Tango Controls Documentation

Release 9.3.4

Tango Community (CC BY 4.0)

Oct 10, 2023

CONTENTS

1 Welcome to Tango Controls documentation! 3
- 1.1 How this documentation is organized . . . 3
- 1.2 Indices and tables . . . 4

2 Authors 5
- 2.1 Acknowledgements . . . 6

3 Overview 7
- 3.1 Introduction . . . 8
- 3.2 Overview of Tango Controls . . . 8
- 3.3 Simplified Tango Device Server Model . . . 9
- 3.4 Ecosystem . . . 11
- 3.5 History . . . 11

4 Installation 13
- 4.1 Overview . . . 13
- 4.2 Source Code . . . 16
- 4.3 Long Term Support Versions . . . 16
- 4.4 Linux . . . 17
- 4.5 Windows . . . 21
- 4.6 Raspberry Pi . . . 30
- 4.7 Amazon Cloud . . . 32
- 4.8 Virtual Machine . . . 36
- 4.9 PyTango and Taurus on Windows . . . 64
- 4.10 Binary packages . . . 65
- 4.11 Patches . . . 65

5 Getting Started 67
- 5.1 First steps with Tango Controls . . . 67
- 5.2 End-user applications guide . . . 68
- 5.3 How to develop for Tango Controls . . . 68
- 5.4 Administration applications guide . . . 84

6 Developer's Guide 87
- 6.1 Overview . . . 87
- 6.2 General guidelines . . . 87
- 6.3 10 things you should know about CORBA . . . 90
- 6.4 Tango Client . . . 91
- 6.5 Device Servers . . . 150
- 6.6 Debugging and Testing . . . 285
- 6.7 Advanced . . . 290

6.8 Tango Core C++ Classes Reference Documentation . . . 372
6.9 Contributing . . . 372

7 Tools and Extensions **379**
7.1 The most popular tools . . . 379
7.2 Built-in tools . . . 379
7.3 GUI building . . . 503
7.4 Web tools . . . 505
7.5 Archiving . . . 509
7.6 Bindings . . . 582
7.7 Other tools . . . 584

8 Administration **587**
8.1 Overview . . . 587
8.2 Deployment . . . 587
8.3 Services . . . 600
8.4 Maintenance . . . 658

9 Tutorials and How-Tos **661**
9.1 Tutorials . . . 661
9.2 HOW-TOs . . . 667
9.3 Tutorials for administrators . . . 749
9.4 HOW-TOs for administrators . . . 749
9.5 HOW-TOs for developers . . . 749

10 Reference **751**
10.1 Map Key . . . 751
10.2 Glossary . . . 752
10.3 Bibliography . . . 754

11 Contents for administrators **755**

12 Contents for developers **757**

13 Contents for users **759**

Bibliography **761**

Index **763**

Intended audience: all

CHAPTER

ONE

WELCOME TO TANGO CONTROLS DOCUMENTATION!

Intended audience: all

1.1 How this documentation is organized

If you identify yourself with one of the following roles you may directly use related links:

End user	*Index*	*End-user applications guide*
Beginner	*Index*	*First steps with Tango Controls*
Developer	*Index*	*How to develop for Tango Controls*
Administrator	*Index*	*Administration applications guide*

The documentation is organized in the following categories (some of them overlap):

- *Overview* will give you a quick overview of what Tango Controls is, its origins and who uses it. Start reading here.
- *First steps* will lead you through getting started with Tango Controls. This category includes an overview of Tango Controls concepts, procedures for installation and starting the system as well as *Getting started* tutorials.
- *Developer's Guide* documents the API and information for **Developers** needed for development of *Device Servers* and client applications.
- *Administration* section is important mainly for **System Administrators**. However, it may provide some information for both **End Users** and **Developers**, too. It contains useful information on Tango Controls system deployment, startup and maintenance.
- *Tools and extensions*. Tango comes with rich set of command line tools, graphical toolkits and programming tools for management, developing graphical applications and connecting with other systems and applications. All, **End Users**, **Developers** and **System Adminstrators** should take a look at the toolkits' manuals.
- *Tutorials and HOWTOs* give step by step guidance and teach you how to work with Tango Controls.
- *Table of Contents* provides access to all documents.
- If you want to contribute to the documentation please read the document *How to work with Tango Controls documentation* and the *Documentation workflow tutorial* .

1.2 Indices and tables

- *Table of Contents*
- genindex
- modindex
- search
- *Glossary*

CHAPTER

TWO

AUTHORS

Intended audience: all

Good documentation needs dedicated authors who spend lots of time writing and reading text instead of code. This labour of love is only rarely appreciated by readers. This section lists the numerous contributors to the Tango documentation. If you are reading this section don't hesitate to send them some positive thoughts and thanks for their "labour of love" right now!

The following people have contributed to the Tango documentation over the years:

- **Gwenaelle Abeille** - for writing the original JTango documentation
- **Reynald Bourtembourg** - for writing the HDB++ documentation
- **Thomas Braun** - for doing the first conversion of the Book to sphinx
- **Alain Buteau** - for the guidelines documentation
- **Matteo di Carlo** - for drawing the Device Server system model
- **Tiago Couthino** - for writing the PyTango documentation and showing the way with Sphinx
- **Lukasz Dudek** - for converting many documents to Sphinx and setting up CI
- **Piotr Goryl** - for reformatting the documentation into Sphinx and documentation master
- **Lajos Fülöp** - for the original layered maps
- **Andy Götz** - for contributing and motivating to have a complete Tango documentation
- **Stuart James** - for editing and updating the layered maps
- **Igor Khokhrakiov** - for writing the new version of JTango documentation, REST api, Amazon cloud etc
- **Nicolas Leclerq** - for the Yat4Tango, bindings documentations
- **Olga Merkulova** - for re-organising the documentation and writing getting started
- **Lorenzo Pivetta** - for writing the HDB++ documentation
- **Faranguiss Poncet** - for writing the ATK documentation
- **Jean-Luc Pons** - for writing the Jive documentation
- **Sergi Rubio** - for the Fandango and Panic documentation
- **Olivier Tachet** - for the how to on installing Tango on a Raspberry Pi
- **Guidelines Team** - the following people contributed to the *device server guidelines*: Alain Buteau, Jens Meyer, Jean Michel Chaize, Emmanuel Taurel, Pascal Verdier, Nicolas Leclerq, M.Lindberg, Sebastien Gara, S. Minolli, and Andy Götz.

- **First Write-the-Doc Team** - the following people assisted to the first Write-the-Doc camp: Piotr Goryl, Olga Merkulova, Lukasz Zytniak, Lukasz Dudek, Matteo di Carlo, Matteo Canzari, Igor Khokhrakiov, Reynald Bourtembourg, Jean-Michel Chaize, Stuart James and Andy Götz
- **Second Write-the-Doc Team** - the following people participated in the second Write-the-Doc camp: Reynald Bourtembourg, Thomas Braun, Sébastien Gara, Philippe Gauron, Andrew Götz, Piotr Goryl, Anton Joubert, Krystian Kędroń, Igor Khokhrakiov, Grzegorz Kowalski, Olga Merkulova, Guillaume Mugerin, Lorenzo Pivetta and Sergi Rubio
- **Emmanuel Taurel** - for writing the first Tango documentation (*The Book*) single handedly!
- **Pascal Verdier** - for writing the Pogo and Astor documentation
- **Lukasz Zytniak** - for converting many documents to Sphinx

Please add your name to the above list if you have contributed to the Tango Documentation.

A big **Thank You** to all of you!

2.1 Acknowledgements

The current Tango documentation would not be possible without the help of:

- **Sphinx** - a big thank you especially to Georg Brandl for inventing Sphinx (by chance Georg is also a member of the Tango community)
- **Github** - for hosting the tango-doc repository
- **Travis** - for the continuous integration of tango-doc
- **Read-the-docs** - for formatting and hosting the online documentation
- **Tango Collaboration** - who sponsored the first Tango Write-the-docs camp and the conversion to Sphinx

CHAPTER

THREE

OVERVIEW

Intended audience: beginners, all

Contents:

3.1 Introduction

Intended audience: beginners, developers, administrators, users, ***Programming language:*** all

Tango Controls is a toolkit for building distributed object based control systems. Distributed objects are an implementation of the Actor model. Actors are primitives of concurrent computation which were proposed in the 70s but have gained renewed interest with massively parallel architectures, IoT, cloud computing etc.

The distributed object in Tango Controls is called a *device* and is created as an object in a container process called a *device server*. The device server implements the network communication and links to the configuration data base and clients. Tango device servers and clients can be written in Python, C++ or Java. Tango comes with a full set of tools for developing, supervising, monitoring and archiving.

The Tango Controls toolkit has been used to build the control systems of large and small physics experiments like synchrotrons, lasers, wind tunnels and radio telescopes. Tango can be used for a single device which requires remote control in a lab or on the internet. Tango can be used as a communication protocol for controlling anything remotely. Tango is ideal for connecting things together and its uses are only limited by your imagination!

We are glad you are with us.

Please, look through the following presentations to get a first overview of **the TANGO Control system**.

- `TANGO introduction`
- `Overview of the TANGO Control system`
- `TANGO basics, technical overview`

Please, also, look on the Tango Controls web page

3.2 Overview of Tango Controls

Intended audience: beginners, developers, administrators, users

3.2.1 What is Tango Controls

Tango Controls is an object oriented, distributed control system framework which defines a communication protocol, an Application Programmers Interface (API) and provides a set of tools and libraries to build software for control systems, especially *SCADA*.

It is build around concept of *devices* and *device classes*. This is unique feature of Tango Controls and make it different to other SCADA software which usually treats a controls system as a set of signals and read and write of process values.

Devices are created by *device servers*. Device servers are processes implementing set of *device classes*. Device classes implement a *state machine*, *command* (actions or methods), *pipes* and *attributes* (data fields) for each class. Each

device therefore has state, zero or more commands, zero or more pipes and zero or more attributes. Device classes are responsible for translating hardware communication protocols into Tango Controls communication. This way you may control and monitor all your equipment like motors, valves, oscilloscopes, etc. Device classes can be used to implement any algorithm or act as a mailbox to any other software program or system.

Tango Controls has been designed to manage small and large systems. Each system has a centralised (MariaDB/MySQL) *database*. The database stores configuration data used at startup of a device server, and acts as name server by storing the dynamic network addresses. The database acts as permanent store of dynamic settings which need to be memorised. Each Tango Control system has a database and is identified by its *Tango Host*. A large system can be made up of tens of thousands or devices (the limit has not been reached yet). Systems of systems are supported by the protocol i.e. the API supports transparent access to devices from multiple systems.

Tango Controls communication protocol defines how all components of the system communicates with each other. Tango uses CORBA for synchronous communications and ZeroMQ for asynchronous communication. The detail of these protocols are hidden from the developer and user of Tango by the API and high level tools.

3.2.2 Tango Technologies

TANGO is based on the 21 century technologies :

- CORBA and ZMQ to communicate between device server and clients
- C++, Python and Java as reference programming languages
- Linux and Windows as operating systems
- Modern object oriented design patterns
- Naturally implements a microservices architecture
- Unit tested, continuous integration enabled
- Hosted on Github (https://github.com/tango-controls)
- Extensive documentation + tools, large community

3.2.3 Tango Community

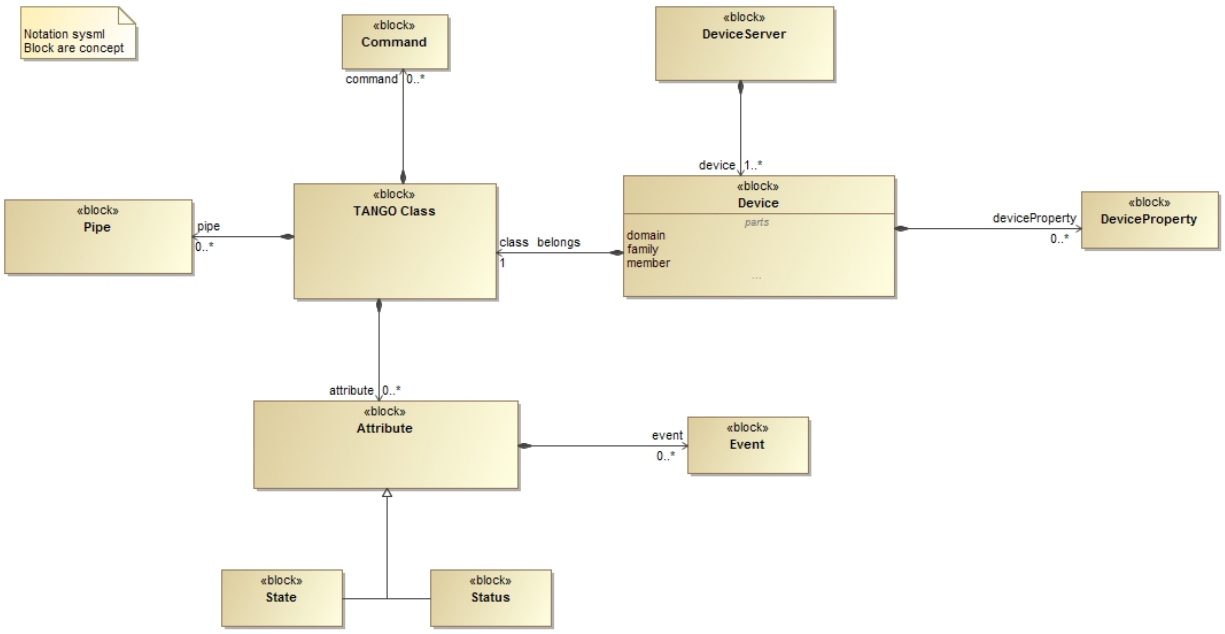
Over the last 17 years that Tango exists over 40 small and large sites (see http://www.tango-controls.org/partners/) have adopted Tango for their control system. Tango is now used to control not only accelerators but also experimental lasers (ELI), wind tunnels (Onera), and most recently has been adopted by the world's largest radio telescope as its core control system (SKA).

3.3 Simplified Tango Device Server Model

Intended audience: beginners, developers, administrators, users

This document is directed to beginner developer.

3.3.1 Primary Presentation



3.3.2 Elements

Block	Description
Device	Abstract concept defined by the TANGO device server object model; it can be a piece of hardware (an interlock bit) a collection of hardware (a screen attached to a stepper motor) a logical device (a taper) or a combination of all these (an accelerator).
TANGO Class	From Object Oriented Programming concept, this is the main class that the developer has to implement
Device-Server	The server (also referred as device server) is a process whose main task is to offer one or more services to one or more clients. To do this, the server has to spend most of its time in a wait loop waiting for clients to connect to it. The devices are hosted in the server process. A server is able to host several classes of devices. In short, it is a process that export devices available to accept requests). Please refer also to the *Glossary*, *device server instance*.
Device-Property	Device specific configuration
Attribute	See *Glossary*, *attribute*.
Pipe	See *Glossary*, *pipe*.
Event	Refer to *Events*.
Command	See *Glossary*.
State	The device state is a number which reflects the availability of the device. Refer to *Events*
Status	The state of the device as a formatted ascii string

3.3.3 Attributes

Block	Attribute	Description
Device	domain/family/member	To identify the device

3.3.4 Relations

Left Block	Right Block	Multiplicity	Description
Device	TANGO Class	1	Every device belongs to a Tango class
Attribute	Event	0..*	An attribute can have more than one event associated
Device	DeviceProperty	0..*	A device can have more than one Device Property associated
DeviceServer	Device	1..*	Every Device server has many devices inside itself
TANGO Class	Attribute	0..*	A TANGO Class can have more than one Attribute associated
TANGO Class	Command	0..*	A TANGO Class can have more than one Command associated
TANGO Class	Pipe	0..*	A TANGO Class can have more than one Pipe associated

3.4 Ecosystem

Intended audience: beginners, all

The Tango ecosystem offers a rich ecosystem for developers and clients alike. The ecosystem is best appreciated via these maps which show what exists and where it is situated in the software stack. Use the *map key* to understand the colour code used.

Tango is a developers toolkit. There are many libraries and tools for implemented device clients and servers. Refer to the *developers guide map* for a quick overview.

Tango is designed to run small and large systems. In order to facilitate managing large systems a number of administrative tools are provided. Refer to the *administrators map* for a quick overview.

All control systems need to be able archive data so they can look back at past data. Tango comes with a number of archiving solutions. Refer to the *archiving map* for a quick overview.

In addition to the rich Python, C++ and Java api's for Tango a number of other languages can be used with the help of one of the many bindings for Tango. Refer to the *bindings map* for a quick overview of the known bindings.

3.5 History

Intended audience: all

The concept of using device servers to access devices was first proposed at the ESRF in 1989. It has been successfully used as the heart of the ESRF Control System of the institute accelerator complex. This control system was called TACO. TACO was based on the SUN RPC (as is the NFS protocol) and C as its core programming language.

In 1999, a renewal of the ESRF distributed control system was started with the aim of replacing SUN/RPC with CORBA, using C++ as the core programming languages. The new software was called TANGO and was developed as a collaboration. In June 2002, Soleil and ESRF offically decide to collaborate to develop this renewal of the old TACO control system. Soleil is a French synchrotron radiation facility based close to Paris. In December 2003, Elettra joined the club. Elettra is an Italian synchrotron radiation facility located in Trieste. Beginning of 2005 ALBA also decided

to join. ALBA is a Spanish synchrotron radiation facility located in Barcelona. DESY and MaxIV were the next big synchrotrons in Europe to join the collaboration. After that things speeded up and more and more sites doing diverse things adopted Tango.

CHAPTER

FOUR

INSTALLATION

Intended audience: administrators, developers, users

Here you will find recipes on how to install the Tango Controls on various platforms.

4.1 Overview

Intended audience: administrators, developers

4.1.1 What is Tango Controls

Tango Controls is an object oriented, distributed control system. It is a framework for building custom SCADA systems. Tango defines communication protocol and API. It provides libraries, set of GUI tools and drivers (so called *Device Servers*) for variety of standard and specific control equipment. For more information see: http://www.tango-controls.org/what-tango-controls/

Your computer may have different (one or more) roles in the Tango CS system. The roles are:

- Tango Host, where configuration of all other components is stored
- Tango Applications, where you run CLI or GUI applications like **`Synoptic`**
- Tango device servers running
- Tango development, where you develop you Device Servers or Tango applications.

Your computer may perform all above roles simultaneously.

Tango Host, DataBaseds

Each Tango Controls system/deployment has to have at least one running DataBaseds *device server*. The machine on which the *device server* is running has a role of so called *Tango Host*. DataBaseds is a device server providing configuration information to all other components of the system as well as a runtime catalog of the components/devices. It allows (among others) client applications to find devices in distributed environment.

The `TANGO_HOST` environment variable is providing information about the address or IP number and the port on which the DataBaseds is listening for Tango connections. The `TANGO_HOST` environment variable is built as follows:

host_name_or_IP:port, example: `localhost:10000`

Tango installation can be very simple running on a single machine for managing a few devices or it can be a fully blown installation managing tens of thousands of devices and multiple Tango control systems.

Tango Host Role

The central role of a Tango control system is Tango Host role, it is created by running the **`DataBaseds`** device server. This device server requires MariaDB or MySQL database in its most common application.

The recommended way of running device servers is to use the *Starter* service.

- a Database server (MariaDB or MySQL)

Warning: root password for database can be different from the computer root password. This password should not be empty. tango database password for tango database can be empty.

- an official Oracle Java JRE (Java Runtime Environment) >= 1.7
- a Tango database. It will ask for a port number, this port will be the one used by the server for Tango requests. The hostname has then to be known from all the computers which will access to Tango Host. It is mandatory to install this tango database **before** every tango client.

Tango development Role

This role is to develop applications and device servers. To play this role, you need:

- the libtango headers for development
- pytango to allow accessing Tango through Python
- an official Oracle Java JRE (Java Runtime Environment) >= 1.7 for development with Java

Tango applications Role

This role is to run CLI and GUI applications. To play this role, you need:

- an official Oracle Java JRE (Java Runtime Environment) >= 1.7 for Java applications
- the libtango java tools (astor, atkpanel, jive, pogo, etc.)
- pytango to allow accessing Tango through Python (if using Python device servers)

Tango device servers Role

This role is to run device servers (drivers): The recommended way of running device servers is to use **`Starter`** service.

To play this role, you need:

- a Tango Starter service
- a TangoTest device server to allow testing
- an official Oracle Java JRE (Java Runtime Environment) >= 1.7 for Java device servers

Every roles

Whatever the role, every computer needs:

- the libtango offline documentation
- the liblog4j package for logging
- to set an environment variable `TANGO_HOST` to the Tango Host and the port, for example

```
TANGO_HOST=mycomputer:10000
```

`mycomputer` is the hostname on which is installed Tango Host, and `10000` is the port defined during the installation of Tango database.

Warning: The choosen port should be defined according to network rules and it should especially be compatible with authorized ports.

4.1.2 Single computer

Installing Tango on a single machine means all roles described above (Tango Host, Tango applications, Tango device servers, Tango development) will be played by the same computer.

The software needed are described in each role.

In this installation type, a `TANGO_HOST` environment variable has to be set to `TANGO_HOST=HOSTNAME:PORT` where HOSTNAME is the name of the computer and PORT is the port on which the server will wait for requests. This will be used to send Tango request. This `TANGO_HOST` environment variable should be loaded at each startup.

4.1.3 Multiple computers

When installing several computers, one should install one Tango host and some clients computers.

Those clients can play different roles (Client computers, Device servers running, and Development).

Moreover, it is possible to start several Tango Host within the same Tango control system in order to keep the control system working if one of them dies. This configuration is described in section *Multiple database servers within a Tango control system*.

4.1.4 Multiple control systems

Several Tango control systems can be used. It means every Tango control systems will have its own Tango Host which will store its own device servers configuration.

In this environment, Tango Host and Tango clients installation is the same as described upside, but `TANGO_HOST` environment variable has to be set on each client according to which server will be used for device servers configuration. The hostnames of the Tango Hosts have then to be known from all the computers which will access to them.

For example, if `testserver` and `productionserver` have been installed as Tango Hosts, each one will propose Tango database as a service, and client can be configured as followed:

- testclient1 with `TANGO_HOST=testserver:10000`
- testclient2 with `TANGO_HOST=testserver:10000`
- operatorclient with `TANGO_HOST=productionserver:10000`

- developerclient with `TANGO_HOST=productionserver:10000`
- dsclient with `TANGO_HOST=productionserver:10000`

In this configuration, one can decide to change `TANGO_HOST` value on a client to use another server. However, this will need to restart every device running on this client.

4.1.5 No database

It is possible to run a device server on some computer without a Tango database.

> **Warning:** A configuration without SQL database can be useful for testing purpose. However, it will not benefit the major part of the Tango functionnalities.

See section *Running a device server without SQL database* to understand how to use this configuration and what are the limitations.

4.2 Source Code

Intended audience: administrators, developers

Tango Controls for C++ and Java is available for download as source code or as pre-compiled binaries.

The latest source code release is 9.3.5.

Always look at the *Patches* page when you download a Tango release. If some patches are available for your release, please, get and apply them.

Previous releases

- 9.3.4
- 9.2.5a
- 9.2.2
- 8.0.5

4.3 Long Term Support Versions

Intended audience: administrators, developers

In 2016, the Tango-Controls Steering Committee requested the introduction of Long Term Support versions for some key components of Tango-Controls like cppTango, the C++ Tango Library and JTango.

Long Term Support (LTS) versions are special versions of Tango components which will be supported for at least 5 years (starting from the day when the next direct major version is released). LTS versions will benefit from critical bug fixes and potentially some patches for simple new features and less critical bugs.

For cppTango, the latest 9.3 version became an LTS version when cppTango 9.4.0 was released on September 30th 2022, three days ahead of schedule. This means that cppTango 9.3.x will be supported until October 2nd 2027.

For the LTS versions only cppTango 9.3.x and starter will stay at C++98, all other projects can require newer C++ standards.

For PyTango there is no LTS policy. PyTango releases target the most recent minor release of cppTango. This means that PyTango currently (July 2022) supports cppTango 9.3.x. After cppTango 9.4.0 is released, future PyTango releases will support cppTango 9.4.x. Critical bug fixes to PyTango for unsupported cppTango releases are not planned.

4.4 Linux

Intended audience: administrators, developers

4.4.1 Debian + Ubuntu

Non-interactive installation

1. Install packages required to compile tango-controls:

```
sudo apt-get install g++ openjdk-8-jdk mariadb-server libmariadb-dev zlib1g-dev␣
↪libomniorb4-dev libcos4-dev omniidl libzmq3-dev make
```

2. Start mariadb :

```
sudo service mariadb start
```

3. Set password for mariabdb root user to 'mypassword' (change as appropriate):

```
sudo mariadb -u root
ALTER USER 'root'@'localhost' IDENTIFIED BY 'mypassword';
# The following 2 lines do not apply on recent mariadb versions (MariaDB-10.4+)
# Please refer to MariaDB documentation to configure correctly your users (https://
↪mariadb.com/kb/en/alter-user/ and https://mariadb.com/kb/en/set-password/)
UPDATE mysql.user SET authentication_string = '' WHERE user = 'root';
UPDATE mysql.user SET plugin = '' WHERE user = 'root';
```

4. Download source tarball from github:

```
wget https://gitlab.com/api/v4/projects/24125890/packages/generic/
↪TangoSourceDistribution/9.3.5/tango-9.3.5.tar.gz
```

5. Unpack in a sub-directory called tango:

```
mkdir tango
cd tango
tar xzvf tango-9.3.5.tar.gz
```

6. Configure tango-controls to build and install in /usr/local/tango (set the DB password as appropriate):

```
./configure --enable-java=yes --enable-mariadb=yes --enable-dbserver=yes --enable-
↪dbcreate=yes --with-mysql-admin=root --with-mysql-admin-passwd='mypassword' --prefix=/
↪usr/local/tango
```

7. Compile tango-controls:

```
make
```

8. Install tango-controls:

```
sudo make install
```

9. Add following lines to start script /usr/local/tango/bin/tango:

```
sudo gedit /usr/local/tango/bin/tango

# add lines near the top:

export MYSQL_USER=root
export MYSQL_PASSWORD=mypassword
```

10. Start tango-controls database server:

```
sudo /usr/local/tango/bin/tango start
```

11. Set the TANGO_HOST variable (note: you can do this in e.g. `~/.bashrc` or wherever it is appropriate for your system):

```
export TANGO_HOST=localhost:10000
```

12. Start test device server:

```
/usr/local/tango/bin/TangoTest test &
```

13. Test Jive:

```
/usr/local/tango/bin/jive &
```

You can now define your device servers and devices, start and test them!

4.4.2 CentOS

RPM packages for RedHat based systems are built from the tango-spec repository using Copr. Copr can be used as a repository but only the latest build is kept forever. To install the packages directly from Copr, please refer to the tango-spec README.

RPM packages from Copr are also available in the MAX-IV's repository. Use yum to install them e.g. to install the TANGO database and test device server:

```
$> sudo yum install -y mariadb mariadb-server
   sudo yum install -y libtango9 tango-db tango-test
```

The above packages install the Tango core C++ libraries, database and TangoTest server.

Installation

If you want to install TANGO on CentOS, here are the steps you should follow:

- add the EPEL repository:

```
$> sudo yum install -y epel-release
```

- add the MAX-IV's public repository by creating the following file:

```
$> sudo nano /etc/yum.repos.d/maxiv.repo
[maxiv-public]
name=MAX IV public RPM Packages - $basearch
baseurl=http://pubrepo.maxiv.lu.se/rpm/el$releasever/$basearch
gpgcheck=0
enabled=1

$> sudo yum makecache
```

- install and start MariaDB:

```
$> sudo yum install -y mariadb-server mariadb
   sudo systemctl start mariadb
   sudo systemctl enable mariadb
```

- run mysql_secure_installation script:

```
$> sudo mysql_secure_installation
```

- install TANGO library:

```
$> sudo yum install -y libtango9 libtango9-devel
```

- install tango-db and tango-common packages:

```
$> sudo yum install -y tango-db tango-common
```

- create TANGO database:

```
$> cd /usr/share/tango-db/
   sudo ./create_db.sh
```

- set up TANGO environment:

> **Note:** You should not use *localhost* as your TANGO_HOST. You can set the machine hostname using **sudo hostnamectl set-hostname tangobox**

```
$> sudo nano /etc/tangorc
```

For example:

```
TANGO_HOST=tangobox:10000
```

- set up environment variables:

```
$> sudo nano /etc/profile.d/tango.sh
```

For example:

```
. /etc/tangorc
export TANGO_HOST
```

- start and enable TANGO database:

```
$> sudo systemctl start tango-db
   sudo systemctl enable tango-db
```

- install Starter and TangoTest:

```
$> sudo yum install -y tango-starter tango-test
```

- start and enable Starter:

```
$> sudo systemctl start tango-starter
   sudo systemctl enable tango-starter
```

- install Java based tools:

```
$> sudo yum install -y tango-java
```

- install PyTango:

```
$> sudo yum install -y python-pytango
```

4.4.3 Arch

An AUR package exists here

Download and install it from there, or use your favourite AUR helper application to do the work for you.

4.4.4 Video

The following video (by Mohamed Cherif Areour, in French with English subtitles) shows you how to install Tango on Ubuntu and LinuxMint.

4.4.5 Testing

How to test that everything was correctly installed

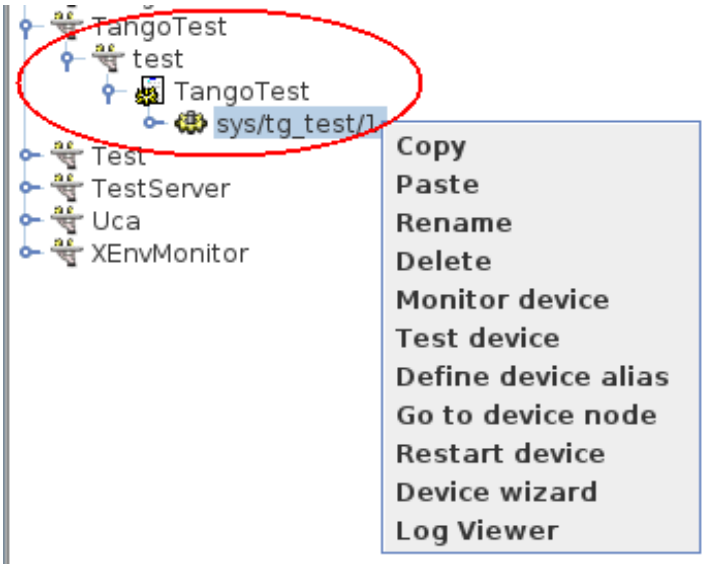
You have to have *"tango-test"* been installed and check where is it located (you can use *"locate TangoTest"* command) and start it with *"test"* instance.

For example:

```
$> /usr/lib/tango/TangoTest test
```

Console should display "Ready to accept request".

After you may go to *Jive* and choose the following window (see the image below):



TangoTest (it is a *server*)-> test (it is an *instance*) -> TangoTest (it is a *class*) -> sys/tg-test/1 (it is a *device*)

Right click on the device and choose *"Test device"*.

You should get a new window with *"Attributes"* where you should see the values. That means you have done everything correct.

4.5 Windows

Intended audience: developers, administrators

This guide provides step by step guide on installation of Tango Controls under Windows operating systems.

4.5.1 Tango package installation

The following video (by Mohamed Cherif Areour, in French with English sub-titles) will help you to install TANGO on Windows.

Prerequisite

Some **Tango Controls** tools require **Java Runtime Environment (JRE) >=1.7**. Please install it first. You may find JRE on https://www.java.com .

The simplest way to have Tango Controls running is to install it from *Binary packages*.

- Download the binary package with your favorite browser.
- Run the downloaded executable file (double-click on it when downloaded).
- Follow instructions provided by the installation wizard.
- Configure `TANGO_HOST` environment variable:
  - On Windows 8 and 10:
    - From the Desktop, right-click the very bottom left corner of the screen to get the *Power User Task Menu*.
    - From the *Power User Task Menu*, click *System*.
  - On Windows XP and 7
    - From the Desktop, right-click the *Computer* icon and select *Properties*. If you don't have a *Computer* icon on your desktop, click *Start* button, right-click the *Computer* option in the *Start* menu, and select *Properties*.
  - Click the *Advanced System Settings* link in the left column.
  - In the System Properties window, click on the *Advanced* tab, then click the *Environment Variables* button near the bottom of that tab.
  - In the *Environment Variables* window click the *New* button.
  - In the field *Name* write `TANGO_HOST`.
  - In the field *Value* write proper value. If it is the only computer in the Tango System provide `localhost:10000`.

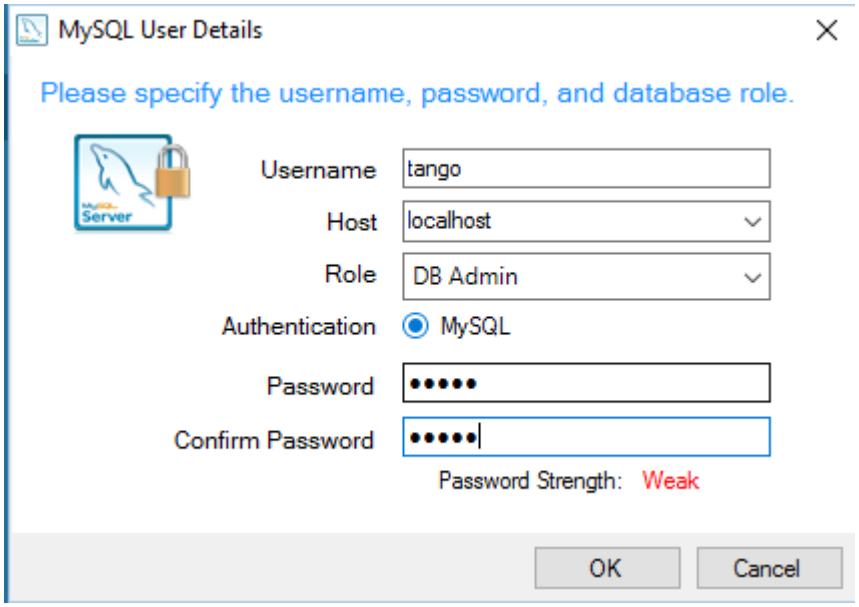
If there is a *Tango Host* already running on some other computer in your deployment and you have provided proper address and port in the `TANGO_HOST` you may start using client and management applications like **Jive**, **Jdraw**/**Synoptic**. In other case you have to configure the system to perform a role of *Tango Host*.

4.5.2 Tango Host role

To make a computer become a Tango Host you need to:

- **Install MySQL server.**
  You may use community version available from https://dev.mysql.com/downloads/mysql/5.7.html . It is suggested to use **`MySQL Installer`** with all tools included. You may read more on MySQL installation topic here: http://dev.mysql.com/doc/refman/5.7/en/windows-installation.html

  It is suggested to create dedicated `tango` user with *DB Admin* priviledges during installation. In the installation wizard on a tab *Accounts and Roles* select button *Add User* and create a dedicated user. See



- **Set up environment variables providing credentials to access MySQL:**
  - Open *Command Line*.
  - Invoke command: **`%TANGO_ROOT%\bin\dbconfig.exe`**.

  Note: This lets you set up two environment variables `MYSQL_USER` and `MYSQL_PASSWORD` used to access the MySQL server. You can check if variables were set correctly, if not you can set it manually. It's recommended to restart computer after operation. You may use `root` credentials provided upon MySQL installation if it is your development workstation. For production environment it is suggested to create an additional user with `DB Admin` privileges. On Windows you may use **`MySQL Installer`** from *Start* menu and select the option *Reconfigure* for MySQL Server. Please refer to: http://dev.mysql.com/doc/refman/5.7/en/adding-users.html

- **Populate database with an initial Tango configuration:**
  - Open a command line.
  - Add MySQL client to be available in the PATH. For MySQL version 5.7 the command should be: **`set PATH=%PATH%;"C:\Program Files\MySQL\MySQL Server 5.7\bin"`**

  Note: Adjust the path according to your MySQL version and the path where it is installed.

  - Invoke **cd "%TANGO_ROOT%\share\tango\db\"**.
  - Call **create_db.bat**.

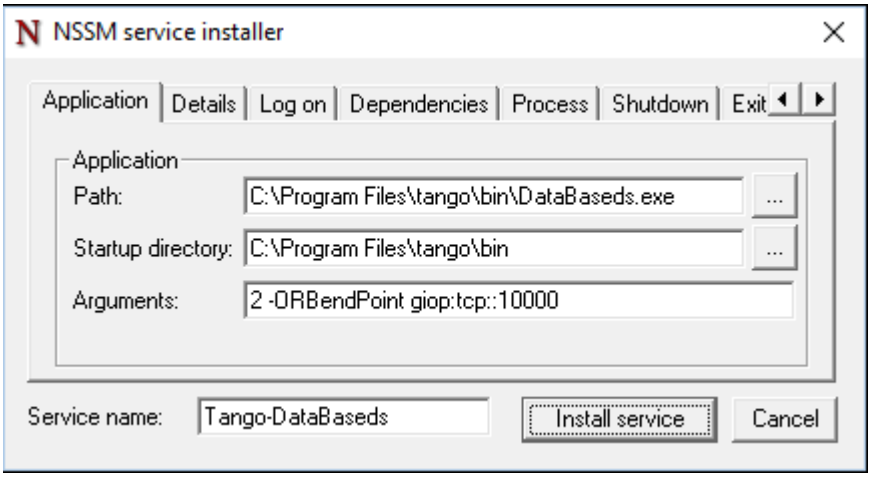
- **Start a DataBaseds *device server*:**
  - Open a new command line window.
  - In the command line call **"%TANGO_ROOT%\bin\start-db.bat"**.

> **Note:** To make your Tango installation operational you have to have this **DataBaseds** running permanently. You may either add the command above to *Autostart* or run it as a service.

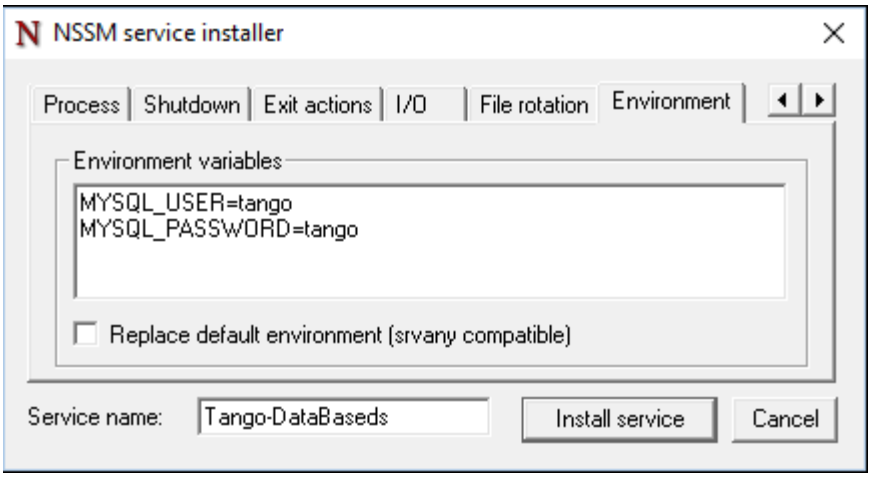
- **Make DataBaseds run as a service**

> **Note:** The proposed solution uses NSSM tool which works on all versions of Windows but you may find some other tools available including native srvany.exe.

  - Download NSSM from http://nssm.cc/.
  - Unpack the file to some convinient location. It is suggested to copy proper (32bit or 64bit) version to the Tango bin folder %TANGO_ROOT%\\bin\\.
  - Open *Command Line* as Administrator.
  - Change current path to where the **nssm** is unpacked or copied, eg. **cd "%TANGO_ROOT%\bin"**.
  - Invoke **nssm.exe install Tango-DataBaseds**. This will open a window where you can define service parameters.
    - In the Application tab provide information as follows (adjust if your installation path is different).



    - In the Environment tab provide variables with credentials used for accessing the MySQL, like:



    * Click *Install Service*.

  - Invoke **`nssm.exe start Tango-DataBaseds`** to start the service.
  - Test if everything is ok. Use *Start* menu to run Jive or in command line call **`"%TANGO_ROOT%\bin\start-jive.bat"`**.

4.5.3 Running Device Servers

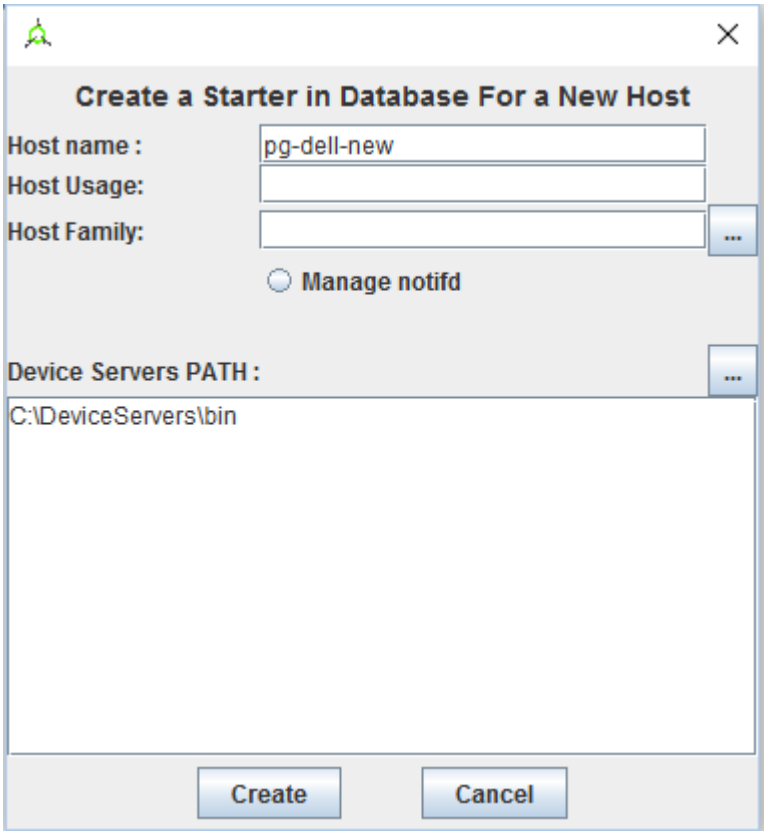
The recommended way of running device servers is to use *Starter* service. Then you may use **`NSSM`** as for **`DataBaseds`**. Assuming you have downloaded it and copied to the Tango bin folder please follow:

- Open Command Line as Administrator (if it is not yet open).
- Prepare folder for *Device Servers* executable:

  Note: To let your device servers start with **`Starter`** service their executables have to be in a path without spaces. This is a limitation of the current **`Starter`** implementation.

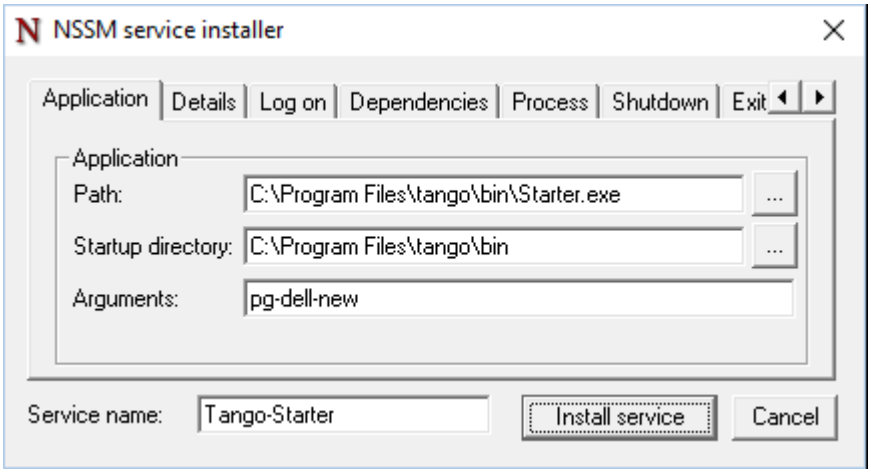
  - Create a directory for *Device Servers*. Let it be `C:\DeviceServers\bin` with **`mkdir c:\DeviceServers\bin`**
  - Change to the Tango bin directory with command (**`cd "%TANGO_ROOT%\bin"`**)
  - Copy **`TangoTest`** *device server* to the newly crated folder: **`copy TangoTest.exe c:\DeviceServers\bin`**

- **Add entry about the Starter device server you will start on your computer:**
  - Start a tool called **`Astor`**. You may use either Windows *Start* menu or call **`tango-astor.bat`**
  - In *Astor* window select menu *Command* → *Add a New Host*
  - In the form that appears provide your *Host name* and *Device Servers PATH*.



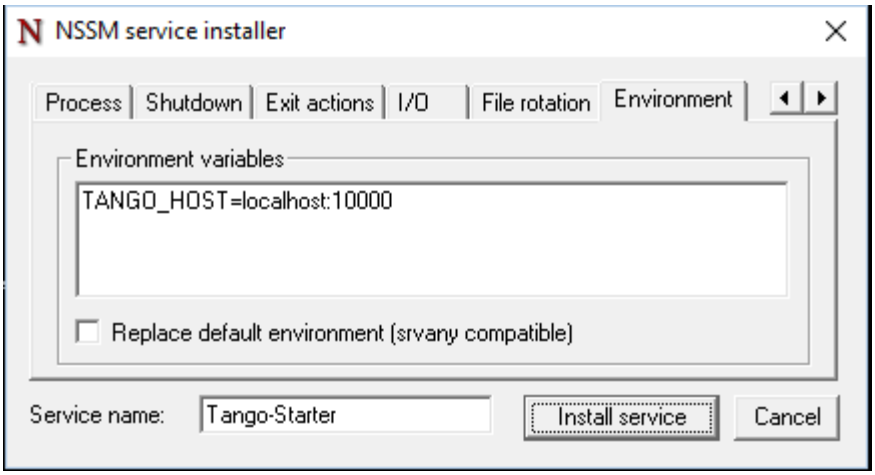
  - Accept with *Create*
  - Go back to **`Command Line`**

- **Install Starter service:**
  - Invoke **`nssm.exe install Tango-Starter`**.
  - In the Application tab provide information as follows:

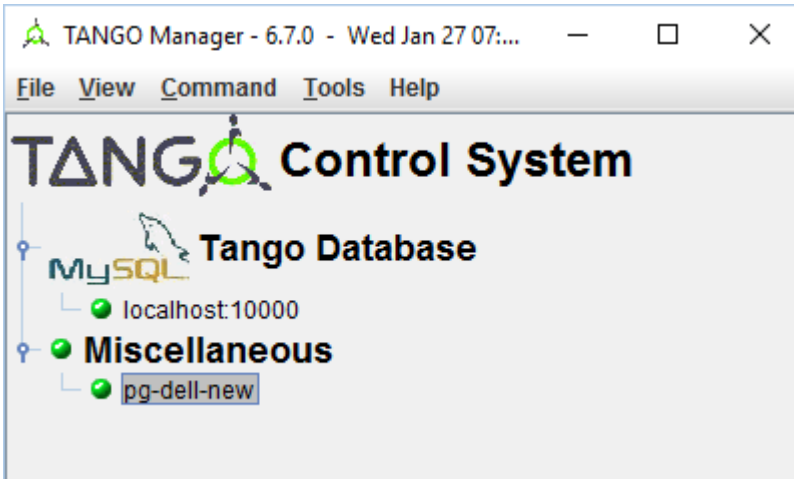


Adjust if your installation path is different. In *Arguments* exchange `pg-dell-new` with the proper name of your host.

  - In the Environment tab provide TANGO_HOST variable, like:



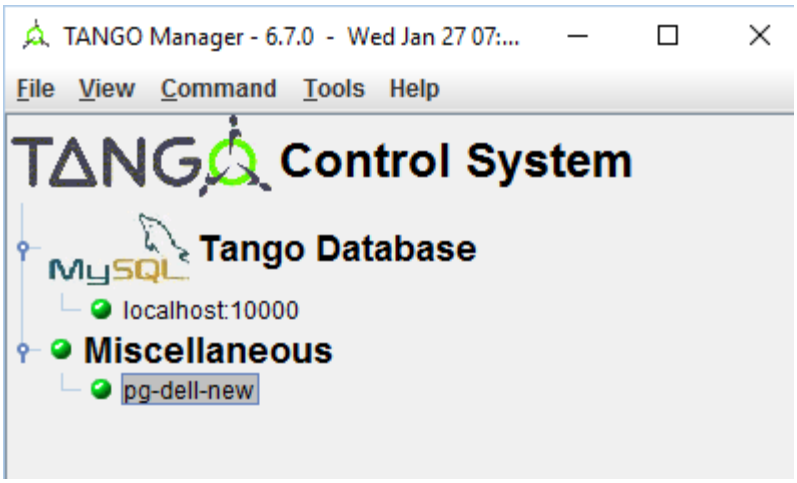
- Click *Install service*
- Start the service: **`nssm.exe start Tango-Starter`**.
- Go back to **`Astor`**.
- After a while you will see a green led next to your host name:



- Run **`TangoTest`** device server:

  You may test the configuration by starting prefigured TangoTest device.

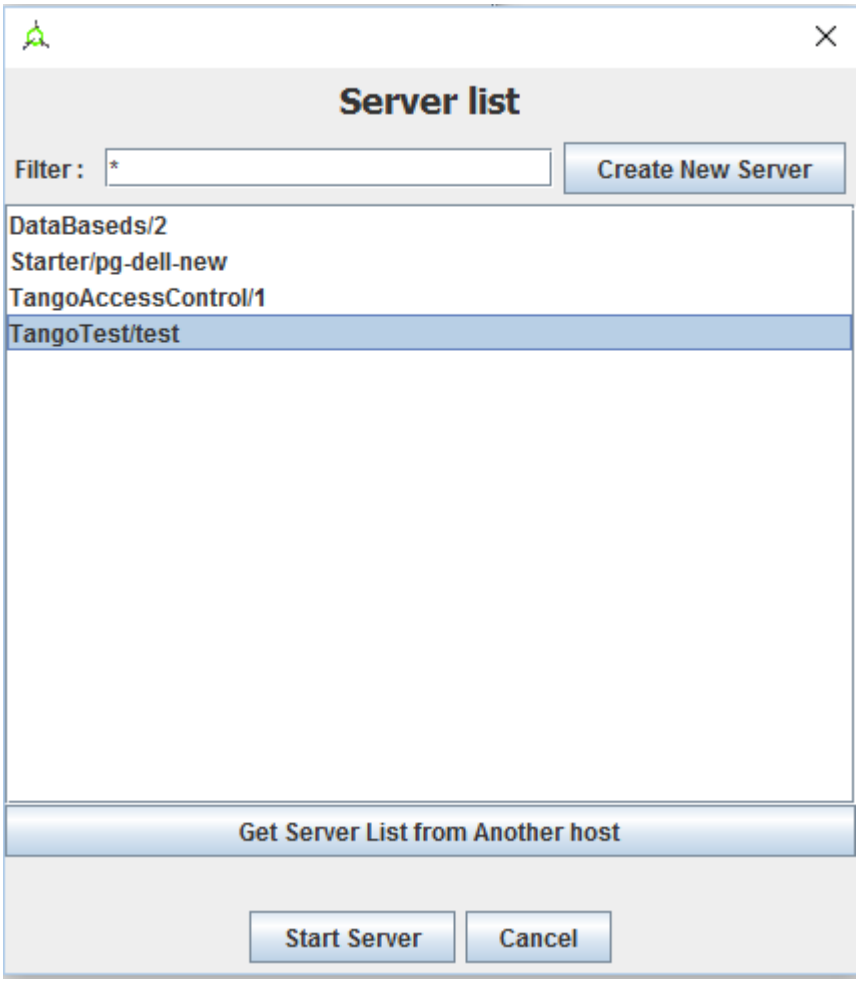
  - Start **`Astor`** if it is not running.



– Double Click on your computer name to open *Control Panel*. It opens a window as below:

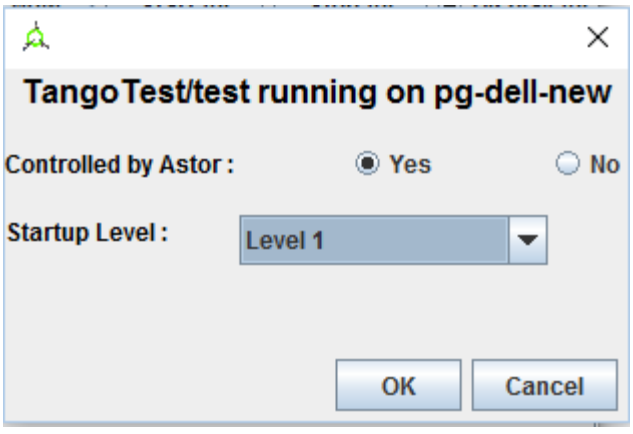
– Click *Start new*.

– In the open window select *TangoTest/test*:



– Click *Start Server*.

– In the open window select *Controlled by Astro -> Yes*, and *Startup Level -> Level 1*.



– When you click *OK* it should start the server. After a while you should see:

- **Running your *Device Servers*:**
  - You need to copy an executable to the folder configured for **`Starter`**. In our example it is `C:\DeviceServers\bin`.
  - Then use **`Astor`**. After opening *Control panel* for your computer (double clicking on a label) and selection *Start New…*
  - Select *Create New Server* and follow a wizard.

4.5.4 What's next

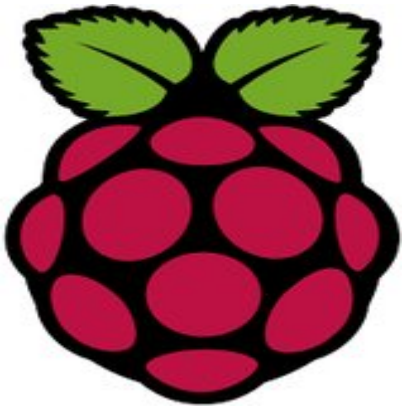
You should check PyTango and Taurus library and tools to cope with scripting and GUIs for Tango *PyTango and Taurus on Windows*.

4.5.5 Typical issues

- **Error when running a command with TANGO_ROOT variable:**
  - Check if environment variable is set correctly.
  - *C:Program is not recognized as an internal or external command, operable program or batch file.*
    - Check if you don't forget about the quotation mark in command.
    - Alternatively, install Tango outside of the `C:\Program Files`.

4.6 Raspberry Pi

Intended audience: developers



Installing Tango on a Raspberry is very simple - just one line command.

4.6.1 Installation with database

If you need the local database, process this installation **before** other installation

```
$ sudo apt install mysql-server mysql-client
$ sudo apt install tango-db tango-common
```

More details on readthedocs

> **Warning:** The ERAS project ReadTheDocs entry does not exist. Please refer to the PDF on the other side of the provided link.

4.6.2 Installation without database

On Raspberry 2 and 3, it is better to wait for the network on booting.

(Raspberry PI Preference menu, wait on boot, check "wait for network")

TANGO installation (one line command) :

```
$> sudo apt install tango-starter tango-test liblog4j1.2-java
```

Graphic tools (Jive, Astor,...) installation :

download the latest version of libtango-java librairies on picca

and installing

```
$> sudo apt install --assume-yes wget\
   wget -c https://people.debian.org/~picca/libtango-java_XX_version.deb\
   sudo dpkg -i ./libtango-java_XX_version.deb
```

4.7 Amazon Cloud

Intended audience: beginner users, beginner developers, beginner administrators

4.7.1 TangoBox 9.3 AMI

The 9.3 release is also available as an AMI image on the AWS. The related AMI-ID is:

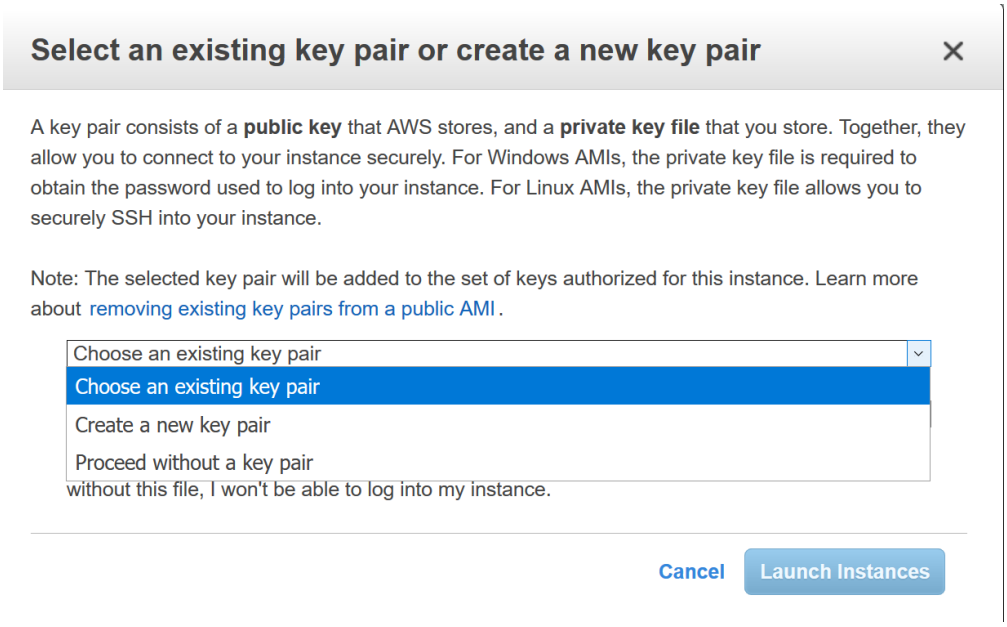
```
ami-0a2e0cddaa68be39f
```

The image contains all the features of the original TangoBox. It requires at least 2 vCPUs and 4GB of memory, which corresponds to *t2.medium* instance type. The running costs apply according to AWS pricing.

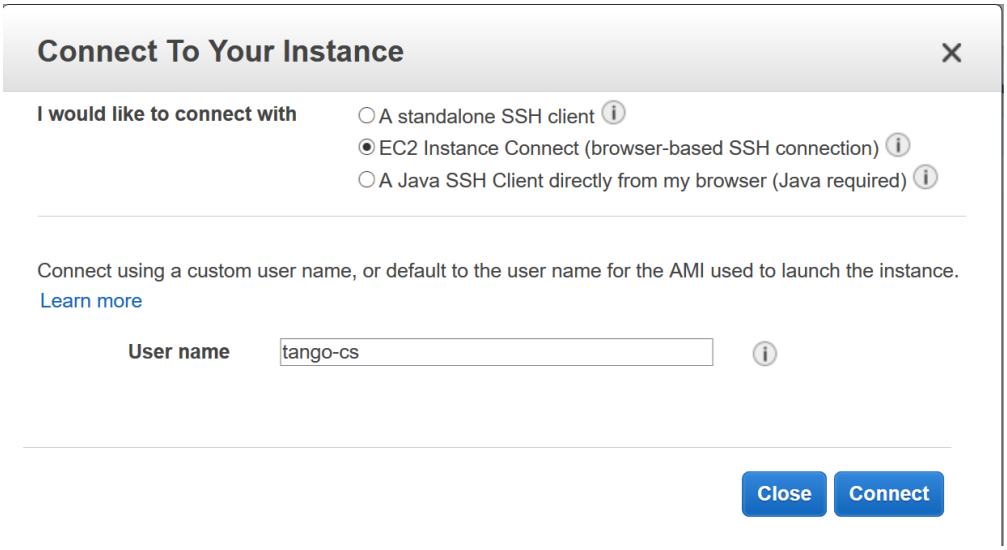
An instance may be accessed with both *SSH access* or *Remote Desktop*.

SSH access

For security reason, the SSH does not accept password authentication. To SSH login to your instance, you need a key-pair configured. The AWS web console asks for the key-pair during the launch process. You may either select exiting or create a new key-par:



Then, you can use the web console *Connect* feature. Please provide the username *tango-cs*:



Remote Desktop

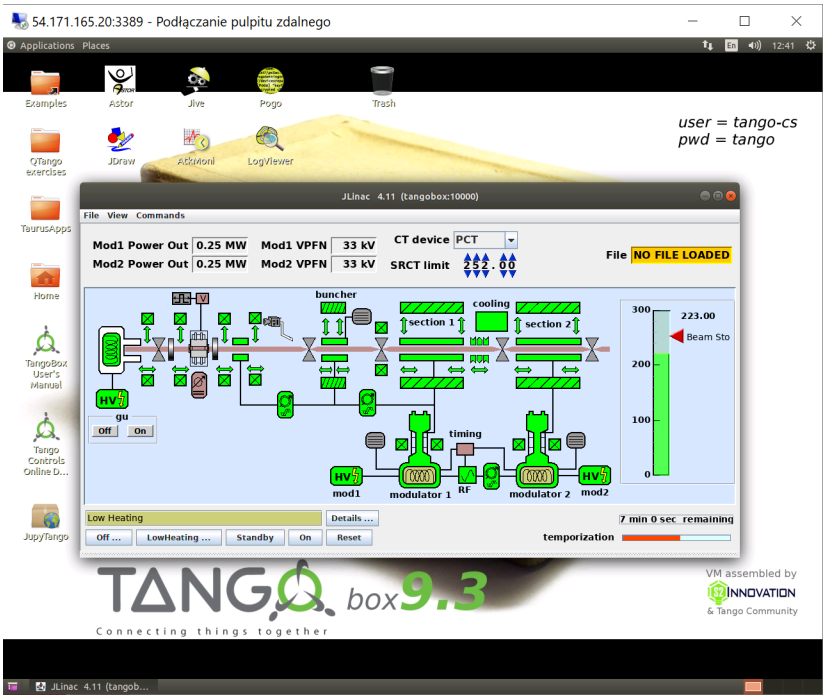
There is also xRDP server installed to enable a desktop connection. So, you can connect to the instance with, for example a windows *Remote Desktop* client. For this feature, the instance *Security Group* settings shall allow for connecting to 3389 port.

Warning: Before enabling the 3389 port, it is recommended to change the default tango-cs user password:

- connect to the instance with the AWS web console *Connect*, as described above.
- call *passwd* and change the password from the default. When prompted for the current password use *tango*.

After enabling the RDP port and connecting with a remote desktop client, you are greeted with the following screen:

After providing the username *tango-cs* and the valid password, you connect to the desktop:



4.7.2 Previous version

The version of TANGO 9.2.5a is also available on the cloud.

An Amazon image running Ubuntu 16.04 with TANGO 9.2.5a is pre-installed and configured to start up at boot time. The image is public and can be found under this id and region:

```
AMI-ID: ami-d503cfba
region=EU-Frankfurt
```

You can find out how to do this here.

Launch VM with this image and you will have TANGO 9.2.5 + PyTango 9.2.0 up and running including the TANGO REST API so you can access it from internet.

Note: the TANGO_HOST is the private IP address of the VM.

This means the TANGO database and device servers are not accessible from the internet but only on the VM or set of VMs which share the same VPN. This can be seen as a security feature. Use the REST api and TANGO security to open up access to the device servers you want to expose.

To experiment with the REST api, start an instance of the AMI image on Amazon cloud.

You can connect to the TangoWebApp as follows:

1. point your browser to this url:

```
http://ec2-35-157-86-137.eu-central-1.compute.amazonaws.com:8080/TangoWebapp/
```

2. click on cancel on the popup login window
3. set the TANGO_REST_URL to

```
http://ec2-35-157-86-137.eu-central-1.compute.amazonaws.com:8080/tango/rest
```

Note: NO spaces before or after and no quotes

5. set the TANGO_HOST to

```
ip-172-31-29-94.eu-central-1.compute.internal:10000
```

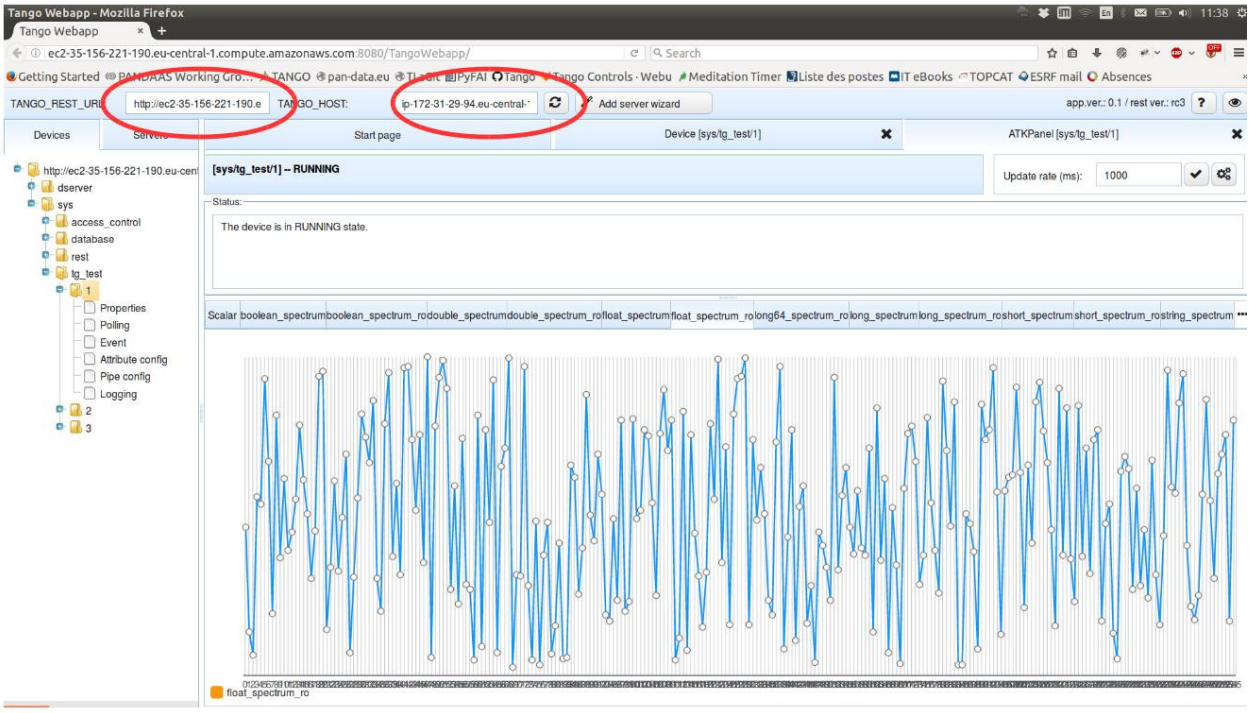
Note: NO spaces or quotes otherwise it won't work!

6. click on the refresh button to the right of the TANGO_HOST field
7. login as user=tango-cs and pw=tango when prompted

Note: If you do not get a new prompt for user name and pwd from the host *ec2-35-156-147-163.eu-central-1.compute.amazonaws.com* then the WebApp is down and it won't work.

8. expand the tree of devices at the top left of the application

See picture below to find out more. You should be able to play with the TangoTest device *sys/tg_test/1*



To see the running DEMO, please, follow the link. Use *tango-cs/tango* to login

4.8 Virtual Machine

Intended audience: developers, administrators, users

The purpose of the **TANGO Box Virtual Machine** is to give you a fast, out-of-the-box experience of a working TANGO system. There is a set of shortcuts to all essential TANGO tools on the virtual machine desktop. Together with the introductory video and user manual, that allow you to experience the power and elegance of a fully configured TANGO system with all the latest tools first hand. After this "guided tour" of the TANGO system, TANGO Box is an excellent tool to make further explorations on your own, to use it for demonstration purposes, to make studies, proof-of-concepts and even production ready systems.

This way, out of this virtual box, another great, sophisticated control system for the real world can be born!

4.8.1 TangoBox 9.3

The latest release of TangoBox is based on Tango Controls v9.3.3 installed on 64 bit Ubuntu 18.04.

- You may download it from here.
- Please read *Tango Controls demo VM's documentation*.
- See also a release note and README.

Minimum Requirements

- 2vCPU
- 2GB (preferred 4GB) RAM
- 30GB of disk space

TangoBox 9.3 on the Amazon cloud

The TangoBox 9.3 release is also available as an AMI image on the AWS (Ireland region). The related AMI-ID is:

```
ami-0a2e0cddaa68be39f
```

Please refer to *Amazon Cloud*.

4.8.2 Previous versions

Previous versions are available below.

TangoBox 9.2

- Download 64 bit Ubuntu virtual machine with TANGO 9.2 RC11.
- Please read *TangoBox 9.2 documentation*.

Elder versions

- Download 64 bit Ubuntu virtual machine with TANGO 9.1.0.

  Note:

  - Use this link to download and read the documentation to install and run the virtual machine on your desktop.
  - Use 7z to unzip it on Linux or unzip on Windows.

- download the 32 bit version running TANGO 7 on Ubuntu 11
- a preconfigured TANGO 8.1.2 64b ubuntu vdi (1.4 GB, for 64 virtual machine)

Note: Read the manual to know more.

We recommend the following videos about TANGO which will help you learn more about TANGO Controls.

TangoBox 9.3

Intended audience: beginner users, beginner developers, beginner administrators

TangoBox is a VM image running Tango Controls system and its various tools. It is intended to be used for demonstration and training. It also simulates distributed deployment by using Docker.

Download

The latest version of the TangoBox 9.3 can be downloaded from here.

What is installed

Below there is list of provide packages/features. Please note that some of them are installed as docker container and maybe switched off (stopped) and requires to be switched on for being explored, see *Switching containers on and off*

- Tango 9.3.3
- Tango Access Control
- PyTango 9.3.3
- Taurus 4.5.1
- QTango
- *Cumbia*
- ITango
- *Sardana 2.7.0*
- *PANIC/PyAlarm*
- *Bensikin*
- *Mambo*
- Docker

- *Linac system simulation* (as docker container tangobox-sim)
- *HDB/TDB, SNAP DS* (as docker container tangobox-archiving)
- *HDB++* (as docker container tangobox-hdbpp)
- *JupyTango* (as docker container tangobox-jupytango)
- SerialLine, Modbus and PyPLC device server (as docker container tangobox-com)
- *mTango + restAPI*, *Tango WebApp* (as docker container tangobox-web)
- *E-giga* (as docker container tangobox-egiga)
- PyCharm
- *ModbusPal to simulate Modbus*

First steps

- First of all you have to download latest release of VirtualBox. It can be downloaded from www.virtualbox.org . Simply install it and start the program.
- TangoBox is released as an **.ova** package so it can be easily imported.
- Select *import* and choose downloaded TangoBox file
- If you want, you can change VM's configuration (i.e graphics, RAM). **It is highly recommended to increase default RAM size**

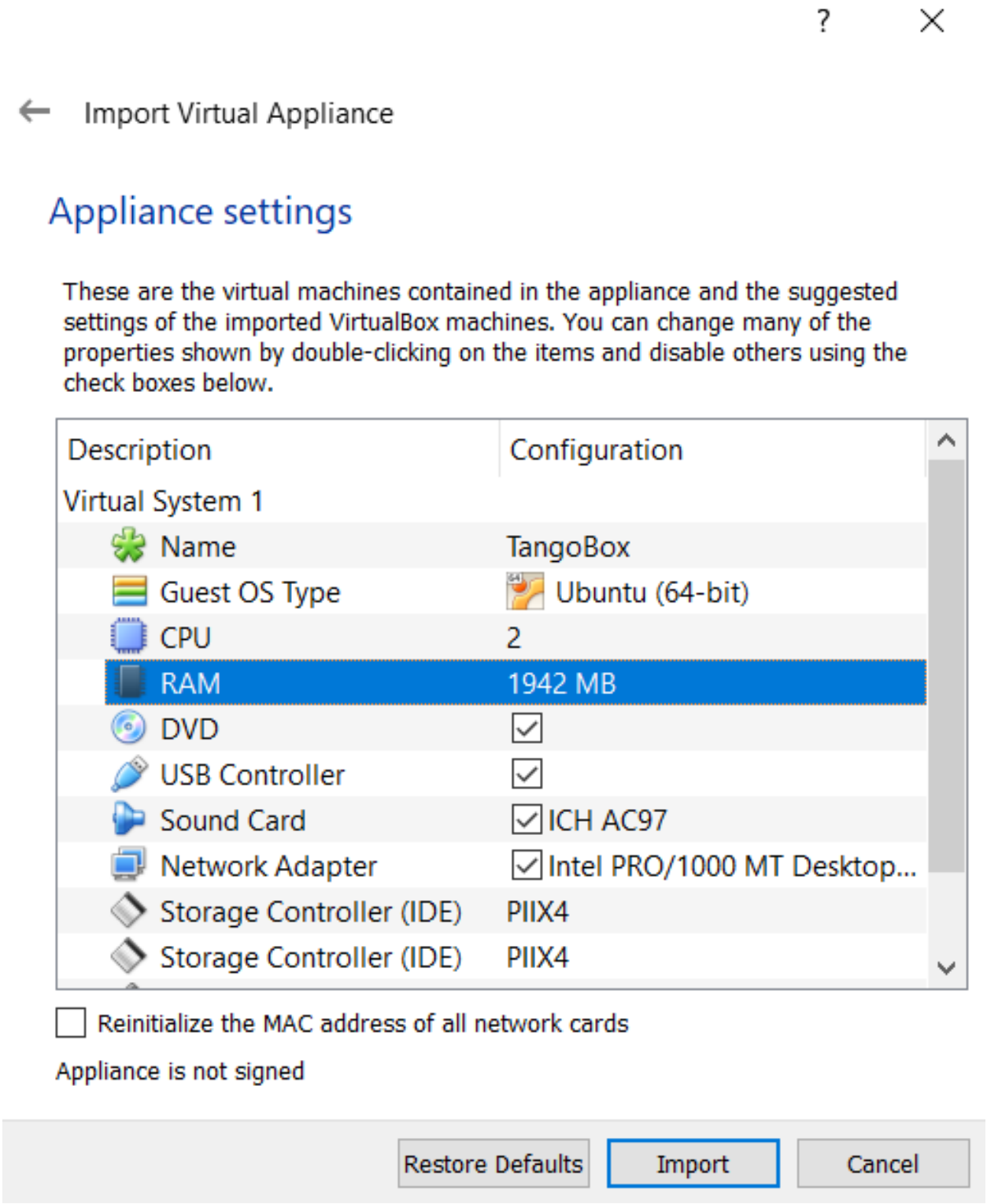


Fig. 1: A virtual machine settings window.
You may change number of CPUs and increase RAM size. Memory size has major impact on VM performance.

- Wait for VirtualBox to import machine

After importing the VM image to VirtualBox you may start it.

- Username is: *tango-cs*
- Password is: *tango*

tango-cs user has sudo privileges, so he may invoke commands as superuser with command **sudo**.

You may explore the Tango Controls feature by clicking related shortcuts on the Desktop.

Note: Plese note that some shortcuts are related to features running on containers. Please start related container first. See the following section.

Switching containers on and off

Some of the features of Tango are provided inside pre-build docker containers. These can be switched on and off by starting or stopping related containers. Containers behave similar to virtual machines with they own network cards and operating system stack, however, lacking full separations.

To start a container, open terminal and invoke **docker start {container-name}**. For example, to star a linac simulation use the following statement:

```
docker start tangobox-sim
```

To stop a container, open terminal and invoke **docker stop {container-name}**. For example, to stop a linac simulation use the following statement:

```
docker stop tangobox-sim
```

To see which containers are running please, call **docker ps**

Deployment structure

Network

Containers are created withing their own subnet: *172.18.0.0/16*. The network is called *tango_nw* The subnet was created with the following docker command:

```
docker network create --driver=bridge --subnet=172.18.0.0/16 --opt com.docker.network.
↪bridge.enable_icc=true \
--opt com.docker.network.bridge.host_binding_ipv4="0.0.0.0" --opt com.docker.network.
↪bridge.mtu=1500 \
--opt com.docker.network.bridge.enable_ip_masquerade=true  tango_nw
```

Containers are assigned static IPs. List of the IPs assignment maybe seen in `/etc/hosts`. Use command **cat /etc/hosts** to see its contents.

Containers and images dependency

Each container is based on its image. All images are already build but, if neccessary, *Dockerfiles* are stored in `home/Dockerfiles` directory. Below is the list of all containers and corresponding images:

Container	Image	Remarks
tangobox-com	*registry*/tangobox-com	-
tangobox-sim	*registry*/tangobox-sim	-
tangobox-archiving	*registry*/tangobox-archive	-
tangobox-hdbpp	*registry*/tangobox-hdbpp	-
tangobox-web	*registry*/tangobox-web	-
tangobox-egiga	*registry*/tangobox-egiga	-
tangobox-jupytango	*registry*/tangobox-jupytango	-
-	*registry*/tangobox-base	Base container
-	ubuntu	Ubuntu image to build others

Currently images are stored in *registry.gitlab.com/s2innovation/tangobox-docker* registry.

Some device servers may be stopped when launching containers. It is so to get better performance (high cpu and ram usage). To control and start/stop particular DS according to your needs, use **Tango Manager (Astor)** to it.

Example applications

Modbus simulation

To simulate Modbus, we suggest to use *ModbusPal*. To do so, use **ModbusPal (simulation)** desktop shortcut. Once it is started, declare new *modbus slave* by clicking **ADD**. Choose 1 and name it **10.0.2.15**. Now click "eye" button in modbus slaves section and add at least 10 holding registers. You can change their values according to your needs.

After that, use **RUN** button to start simulation. Values in registers can be changed by clicking *"eye"* button in *Modbus slaves* section.

Keep in mind that in ModbusPal, registers are counted from **1** while on DS from **0**!

To monitor changes, use ATKPanel started from Jive. Both ModbusComposer and PyPLC have two attributes configured:

- ModbusComposer: Temperature uses **4th**; Pressure uses **5th** register in ModbusPal
- PyPLC: Voltage uses **6th**; Flow uses **7th** register in ModbusPal

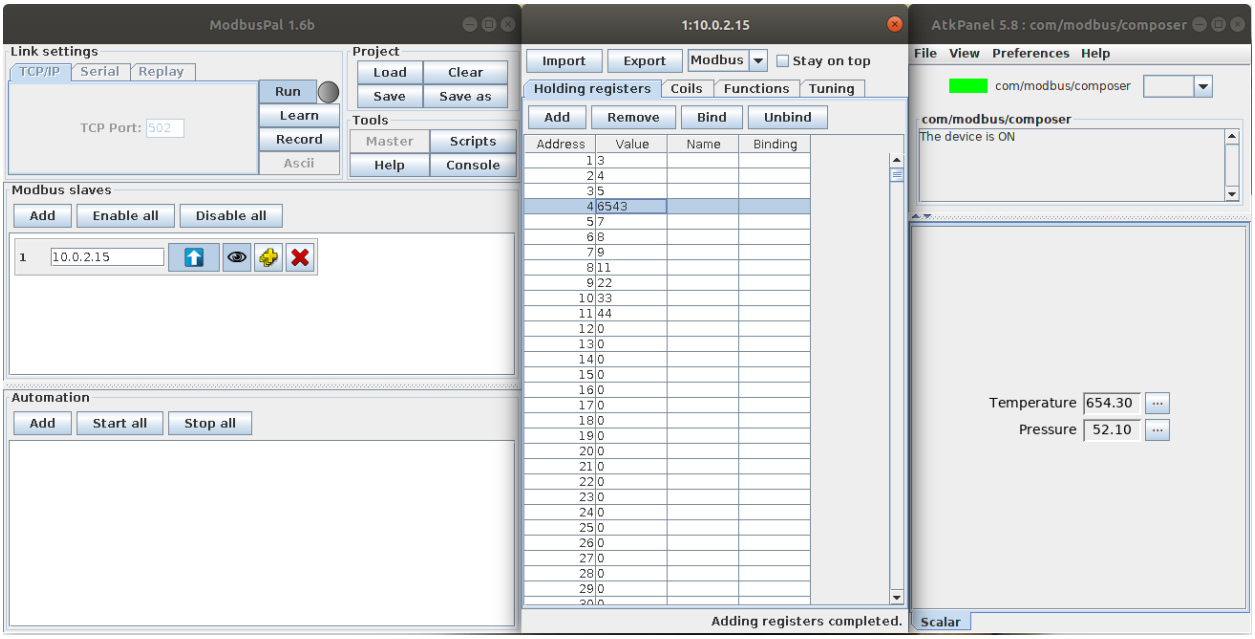


Fig. 2: View on a ModbusComposer device and configured ModbusPal simulator.

JupyTango

JupyTango is a Jupyter featuring Tango related kernels. With JupyterLab you may interact and do scripting for Tango through a web browser.

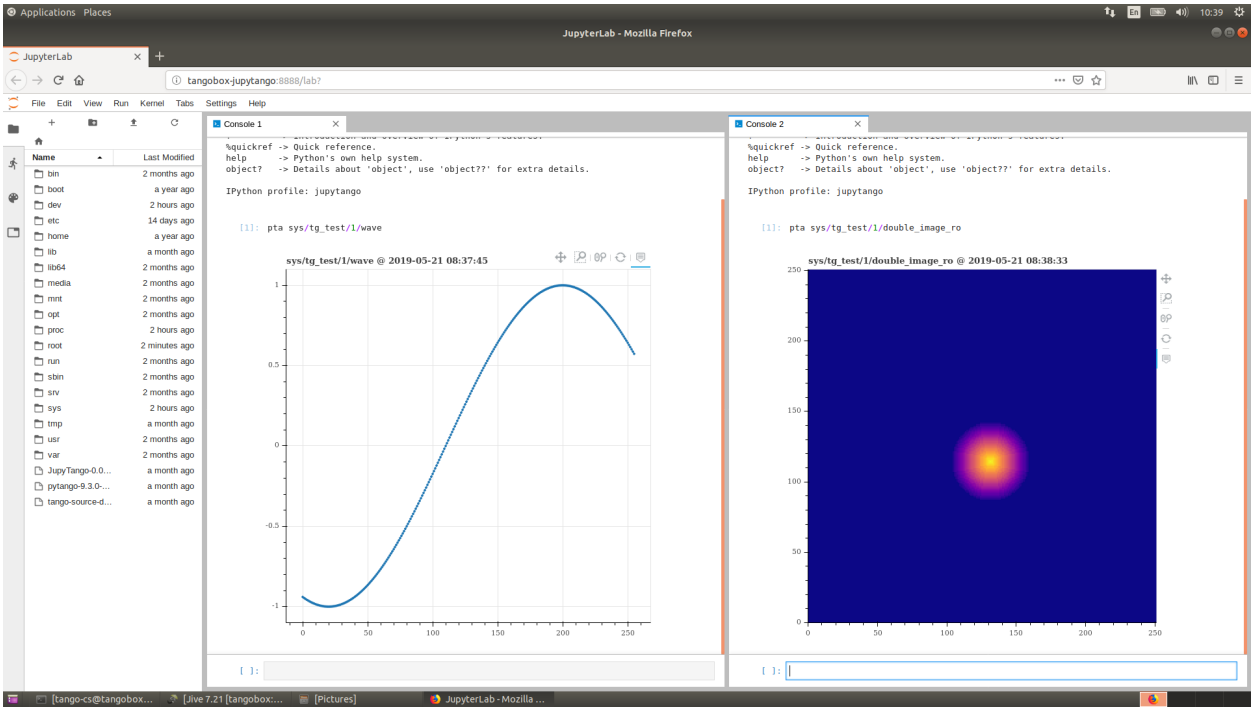


Fig. 3: Browser window with JupyTango in action

In case you want to try it, here's the procedure:

1. start jupyterlab (it is started by default): `docker start tangobox-jupytango`
2. open a new browser window and go to http://tangobox-jupytango:8888/lab
3. enjoy!

Here are the JupyTango additions to itango:

Plotting a tango attribute

Syntax:

```
pta [options] <tab for device selection> + <tab for attribute selection>
```

Supported options:

- *-w* or *–width*: plot width in pixels
- *-h* or *–height*: plot height in pixels

Monitoring a tango attribute: Syntax:

```
tm [options]  + <tab for device selection> + <tab for attribute selection>
```

Supported options:

- *-w* or *–width*: plot width in pixels
- *-h* or *–height*: plot height in pixels
- *-p* or *–period*: plot refresh period in [0.1, 5] seconds - defaults to 1s
- *-d* or *–depth*: scalar attribute history depth in [1, 3600] seconds - defaults to 900s

You can try to kill the monitored device will the JupyTango monitor is running to see how errors are handled.

JLinac simulation

To start simulation, you need to run **tangobox-sim** container (use **`docker start tangobox-sim`** to start it from a terminal). It is also important to make sure that all related device servers are running. The easiest way to do it is to check it in Astor - a bulb next to *tangobox-sim* should be green.

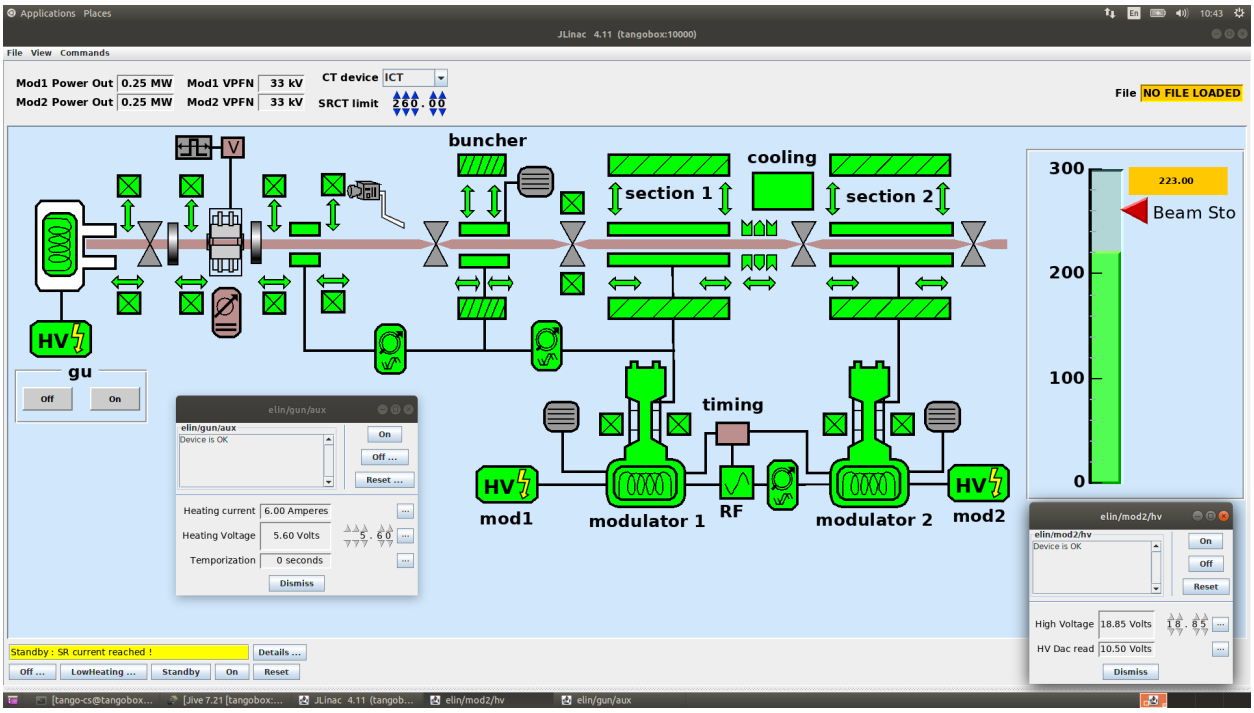


Fig. 4: JLinac simulation running.

HDB/TDB/SNAP Archiving (Mambo, Bensikin)

Prior to use *HDB/TDB* (*Mambo*) or *SNAP* (*Bensikin*) you need to make sure that the **tangobox-archiving** container and related device servers are running:

- Call **`docker start tangobox-archive`** on a terminal.
- Start **`Astor`** and check if the *tangobox-archive* node is green.

Then, you may start **`Mambo`** or **`Bensikin`** by clicking icons on the desktop.

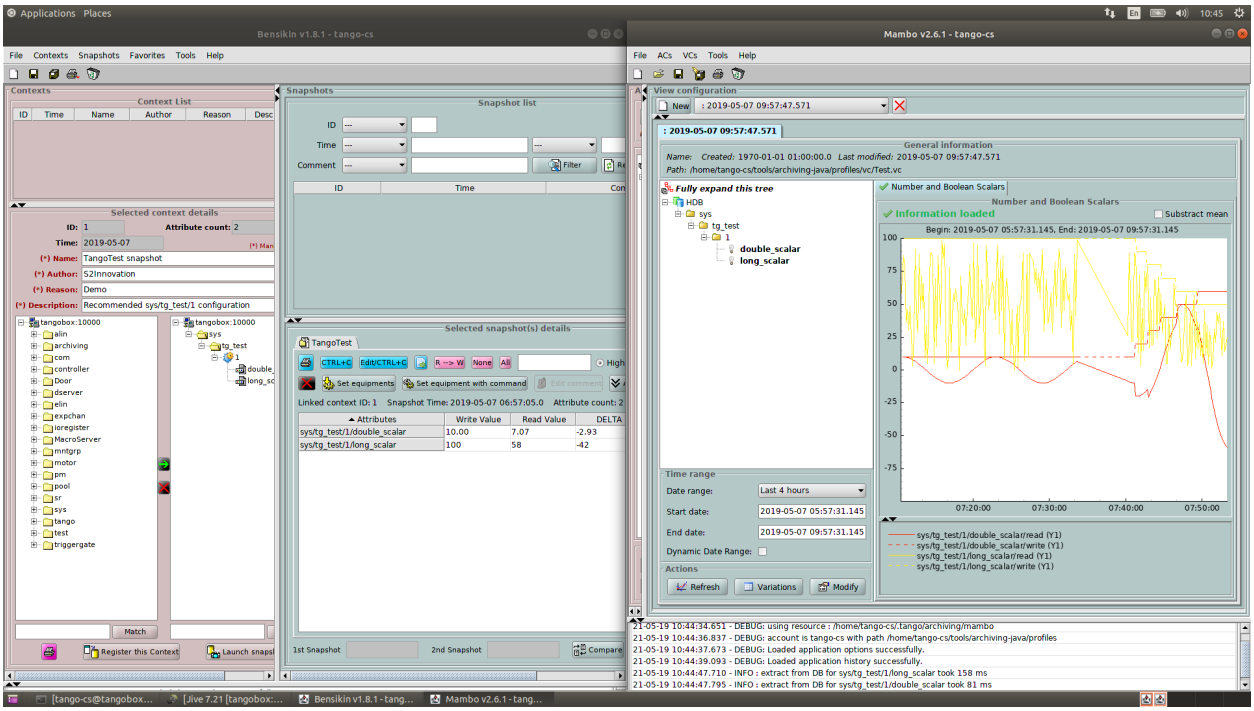


Fig. 5: Screen of running Mambo
Please take note of a green bulb of the *tangobox-archive* node in the Astor window.

HDB++ Archiving

To use *HDB++* and its tools (*HDB Configurator* and *HDB Viewer*) please make sure that the **tangobox-hdbpp** container and related device servers are running:

- Call **`docker start tangobox-hdbpp`** on a terminal.
- Start **`Astor`** and check if the *tangobox-hdbpp* node is green.

Then, you may start **`HDB Configurator`** or **`HDB Viewer`** by clicking icons on the desktop.

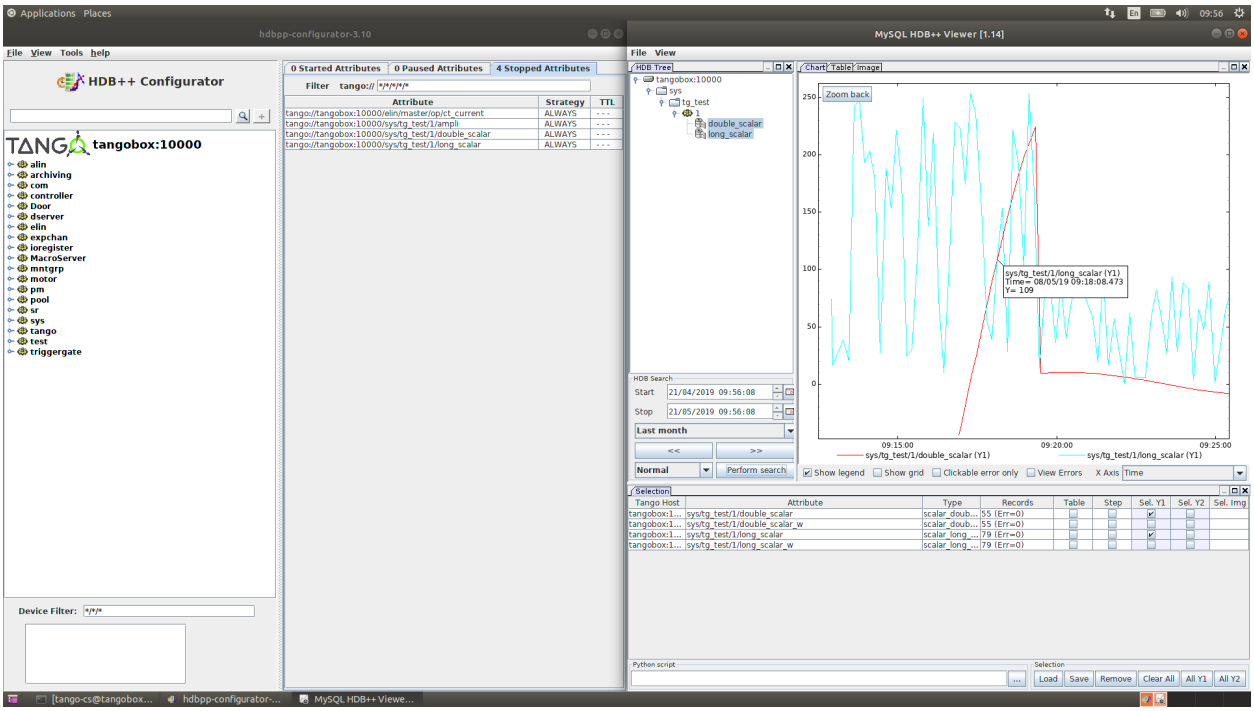


Fig. 6: HDB Configurator and HDB Viewer

E-giga

E-giga is a web application for archiving data visualization (HDB/TDB and HDB++). The TangoBox deployment uses HDB/TDB.

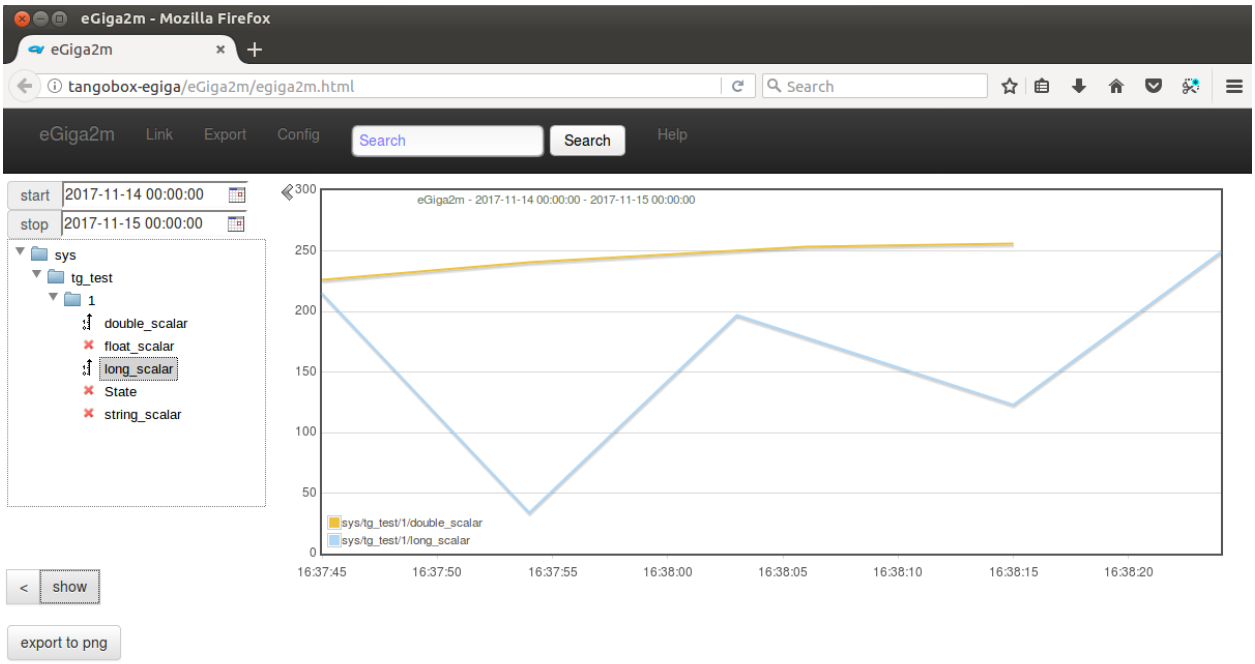


Fig. 7: E-giga in a web browser window

To use e-giga following conditions must be fulfilled:

- *tangobox-archive* and *tangobox-web* containers must be started and archiving device servers must be running
- use i.e. **Mambo** to enable data archiving for HDB database. It is required. If you do not see any attributes in *E-giga* it is probably due to archiving being disabled. Check with **`Astor`** if the tangobox-archive LED is green and with **`Mambo`** if there are any attributes configured to be archivied.

To open browser with **`E-giga`** click on the relate desktop icon.

Tango WebApp

Tango may be available through a web browser. *Tango WebApp* is a general purpose Tango web application. You may try it on the TangoBox.

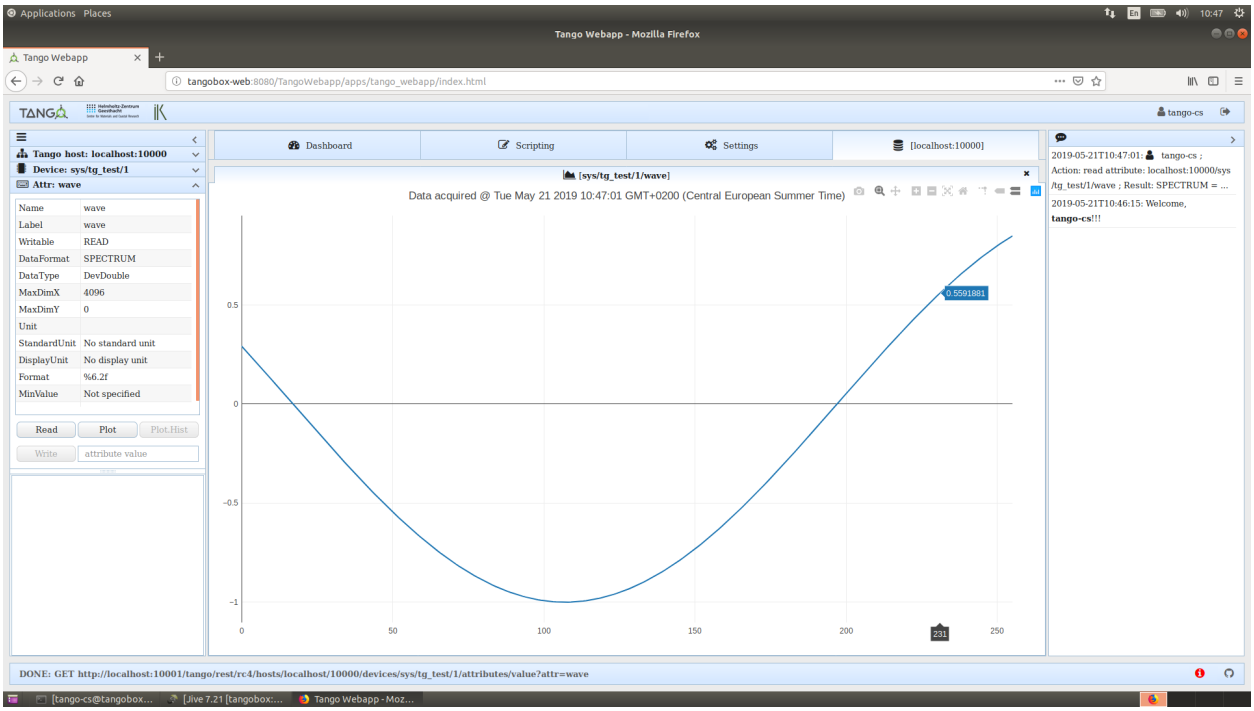


Fig. 8: A screenshot of Tango WebApp in a browser

To play with **Tango WebApp** make sure that the ‘tangobox-web` container is running (use **docker start tangobox-web** to start it from a terminal). Then, you may open a browser with a related desktop icon. Use username *tango-cs* and password *tango* to log-in.

REST API

Tango Controls specifies REST API interface and provides its reference implementation. For details see *REST API documentation*

The **TangoBox** comes with REST API installed. Related desktop icon opens a web browser pointing to REST API interface. The REST server requires authentication. User is *tango-cs* and password is *tango*.

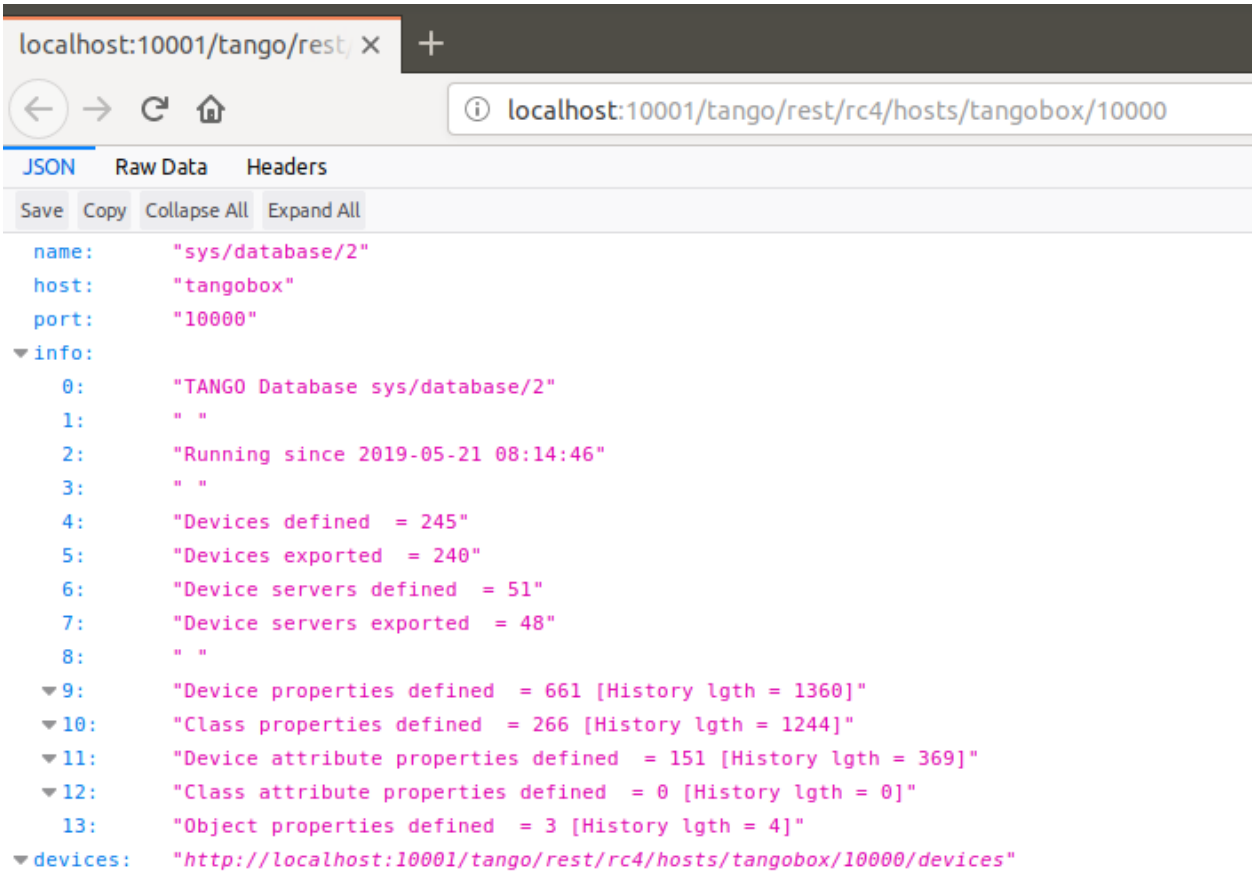


Fig. 9: A web browser window presenting JSON response of the Tango REST server

If you would like to play with it with other tools (Python, curl) it is avaialabe at the following address: *http://localhost:10001/tango/rest/rc4/hosts/tangobox/10000*.

Sardana

Sardana is a software suite for Supervision, Control and Data Acquisition in scientific installations. I t aims to reduce cost and time of design, development and support of the control and data acquisition systems. For more information about it please refer to Sardana documentation.

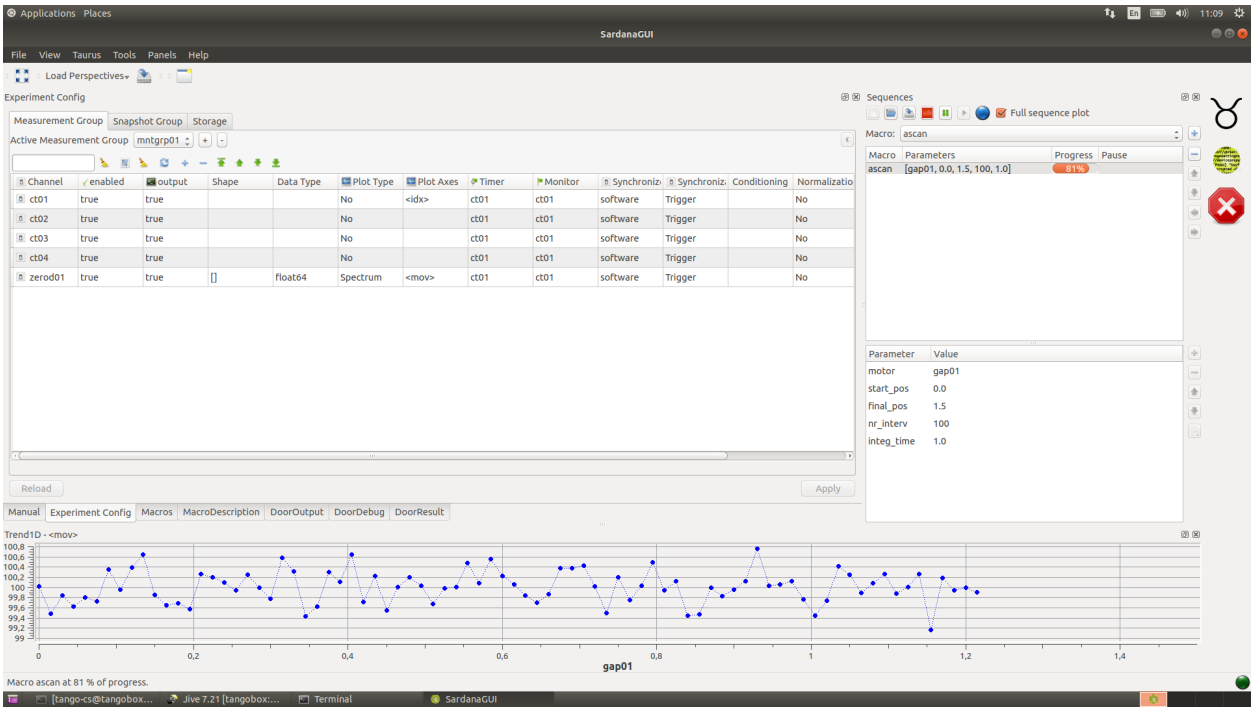


Fig. 10: SardanaGUI in action

To play with Sardana you may double-click the *SardanaGUI* icon on the desktop or run it from a terminal (type **`SardanaGUI`**).

Cumbia

> Cumbia is a new library that offers a carefree approach to multi-threaded application design and implementation. Written from scratch, it can be seen as the evolution of the QTango library, because it offers a more flexible and object oriented multi-threaded programming style.

For more details please check Cumbia webpage and source repository.

Cumbia is installed in `/usr/local/cumbia-libs`. This directory is added to `ld`'s default search path.

To see an example Cumbia application, please run below command or use desktop shortcut *CumbiaClientDemo*:

```
cumbia client sys/tg_test/1/double_scalar
```

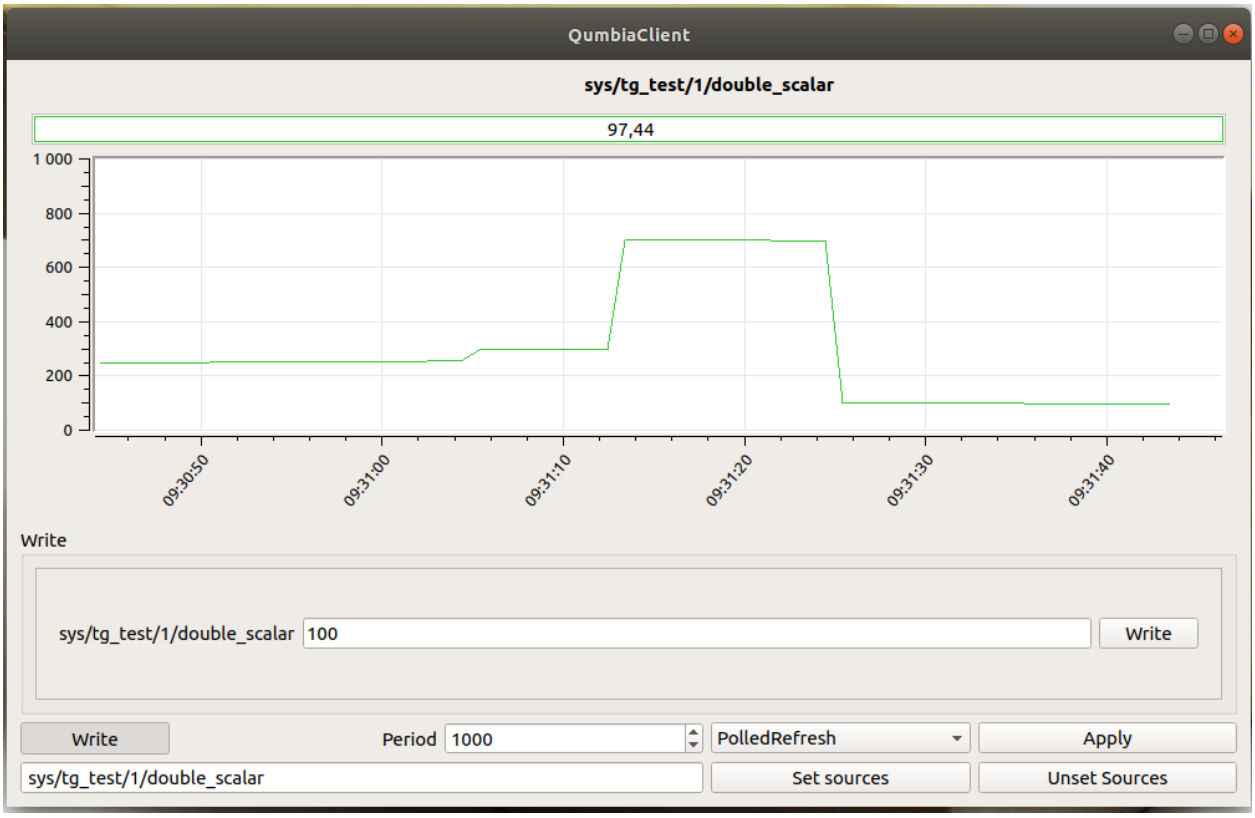


Fig. 11: Cumbia demo application

PANIC/PyAlarm

PANIC Alarm System is a set of tools (api, Tango device server, user interface) that provides: Periodic evaluation of a set of conditions, Notification (email, sms, pop-up, speakers), Notification (email, sms, pop-up, speakers), Keep a log of what happened. (files, Tango Snapshots), Taking automated actions (Tango commands / attributes), Tools for configuration/visualization.

The Panic package contains the python AlarmAPI for managing the PyAlarm device servers from a client application or a python shell. The panic module is used by PyAlarm, Panic Toolbar and Panic GUI.

To launch PANIC GUI, use desktop shortcut *PANIC*.

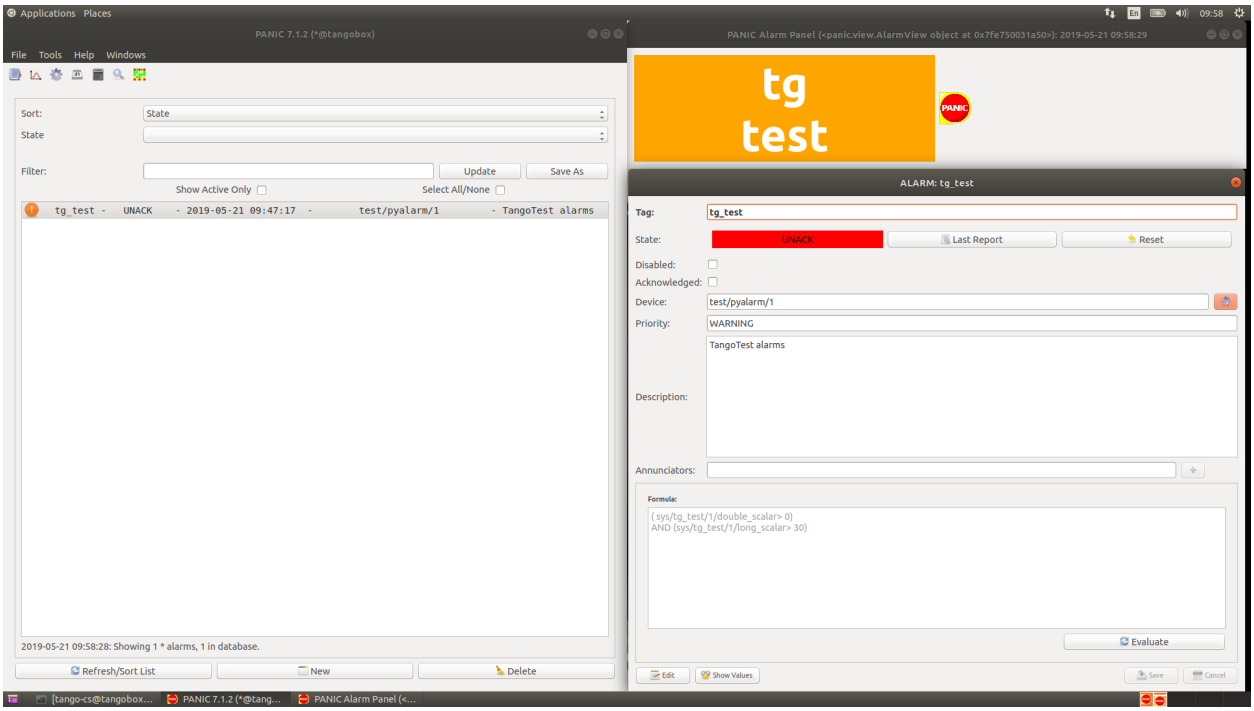


Fig. 12: PANIC GUI application

TangoBox 9.2

Intended audience: beginner users, beginner developers, beginner administrators

TangoBox is a VM image running Tango Controls system and its various tools. It is intended to be used for demonstration and training. It also simulates distributed deployment by using Docker.

Download

The latest version (RC11) of the TangoBox 9.2 can be downloaded from here.

What is installed

Below there is list of provide packages/features. Please note that some of them are installed as docker container and maybe switched off (stopped) and requires to be switched on for being explored, see *Switching containers on and off*

- Tango 9.2.5
- PyTango 9.2.2
- Taurus 4.2.2
- QTango
- iTango
- Tango Access
- *JupyTango*

- PANIC
- *Bensikin*
- *Mambo*
- Docker
- *Linac system simulation* (as docker container tangobox-sim)
- *HDB/TDB, SNAP DS* (as docker container tangobox-archiving)
- *HDB++* (as docker container tangobox-hdbpp)
- SerialLine, Modbus and PyPLC device server (as docker container tangobox-com)
- *mTango + restAPI*, *Tango WebApp* (as docker container tangobox-web)
- *E-giga* (as docker container tangobox-egiga)
- PyAlarm DS on each container
- PyCharm
- Visual Studio Code
- *ModbusPal to simulate Modbus*
- *Sardana 2.3.2*

First steps

- First of all you have to download latest release of VirtualBox. It can be downloaded from www.virtualbox.org . Simply install it and start the program.
- TangoBox is released in **.ova** extension so it can be easily imported.
- Select *import* and choose downloaded TangoBox file
- If you want, you can change VM's configuration (i.e graphics, RAM). **It is highly recommended to increase default RAM size**

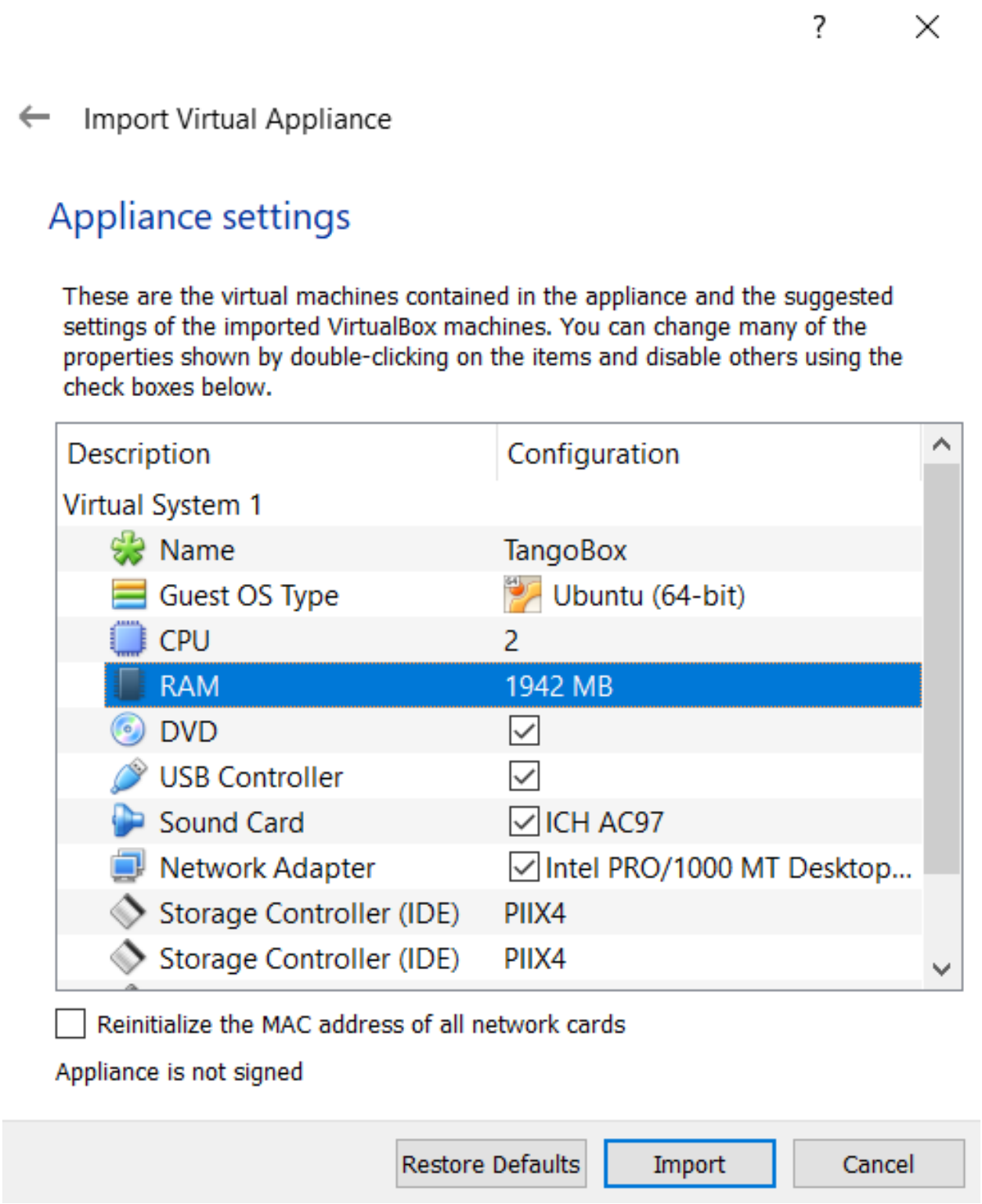


Fig. 13: A virtual machine settings window.
You may change number of CPUs and increase RAM size. Memory size has major impact on VM performance.

- Wait for VirtualBox to import machine

After importing the VM image to VirtualBox you may start it.

- Username is: *tango-cs*
- Password is: *tango*

tango-cs user has sudo rights, so he may invoke commands as superuser with command **sudo**.

You may explore the Tango Controls feature by clicking related shortcuts on the Desktop.

Note: Plese note that some shortcuts are related to features running on containers. Please start related container first. See the following section.

Switching containers on and off

Some of the features of Tango are provided inside pre-build docker containers. These can be switched on and off by starting or stopping related containers. Containers behave similar to virtual machines with they own network cards and operating system stack, however, lacking full separations.

To start a container, open terminal and invoke **docker start {container-name}**. For example, to star a linac simulation use the following statement:

```
docker start tangobox-sim
```

To stop a container, open terminal and invoke **docker stop {container-name}**. For example, to stop a linac simulation use the following statement:

```
docker stop tangobox-sim
```

To see which containers are running please, call **docker ps**

Deployment structure

Network

Containers are created withing their own subnet: *172.18.0.0/16*. The network is called *tango_nw* The subnet was created with the following docker command:

```
docker network create --driver=bridge --subnet=172.18.0.0/16 --opt com.docker.network.
↪bridge.enable_icc=true \
--opt com.docker.network.bridge.host_binding_ipv4="0.0.0.0" --opt com.docker.network.
↪bridge.mtu=1500 \
--opt com.docker.network.bridge.enable_ip_masquerade=true  tango_nw
```

Containers are assigned static IPs. List of the IPs assignment maybe seen in `/etc/hosts`. Use command **cat /etc/hosts** to see its contents.

Containers and images dependency

Each container is based on its image. All images are already build but, if neccessary, *Dockerfiles* are stored in `home/Dockerfiles` directory. Below is the list of all containers and corresponding images:

Container	Image	Remarks
tangobox-com	com	•
tangobox-sim	sim	•
tangobox-archiving	archive	•
tangobox-hdbpp	hdbpp	•
tangobox-web	web	•
tangobox-egiga	egiga	•
tangobox-sardana	sardana	
•	base	Base container
•	ubuntu	Ubuntu image to build others

Some device servers may be stopped when launching containers. It is so to get better performance (high cpu and ram usage). To control and start/stop particular DS according to your needs, use **Tango Manager (Astor)** to it.

Example applications

Modbus simulation

To simulate Modbus, we suggest to use *ModbusPal*. To do so, use **ModbusPal (simulation)** desktop shortcut. Once it is started, declare new *modbus slave* by clicking **ADD**. Choose 1 and name it **10.0.2.15**. Now click "eye" button in modbus slaves section and add at least 10 holding registers. You can change their values according to your needs.

After that, use **RUN** button to start simulation. Values in registers can be changed by clicking *"eye"* button in *Modbus slaves* section.

Keep in mind that in ModbusPal, registers are counted from **1** while on DS from **0**!

To monitor changes, use ATKPanel started from Jive. Both ModbusComposer and PyPLC have two attributes configured:

- ModbusComposer: Temperature uses **4th**; Pressure uses **5th** register in ModbusPal
- PyPLC: Voltage uses **6th**; Flow uses **7th** register in ModbusPal

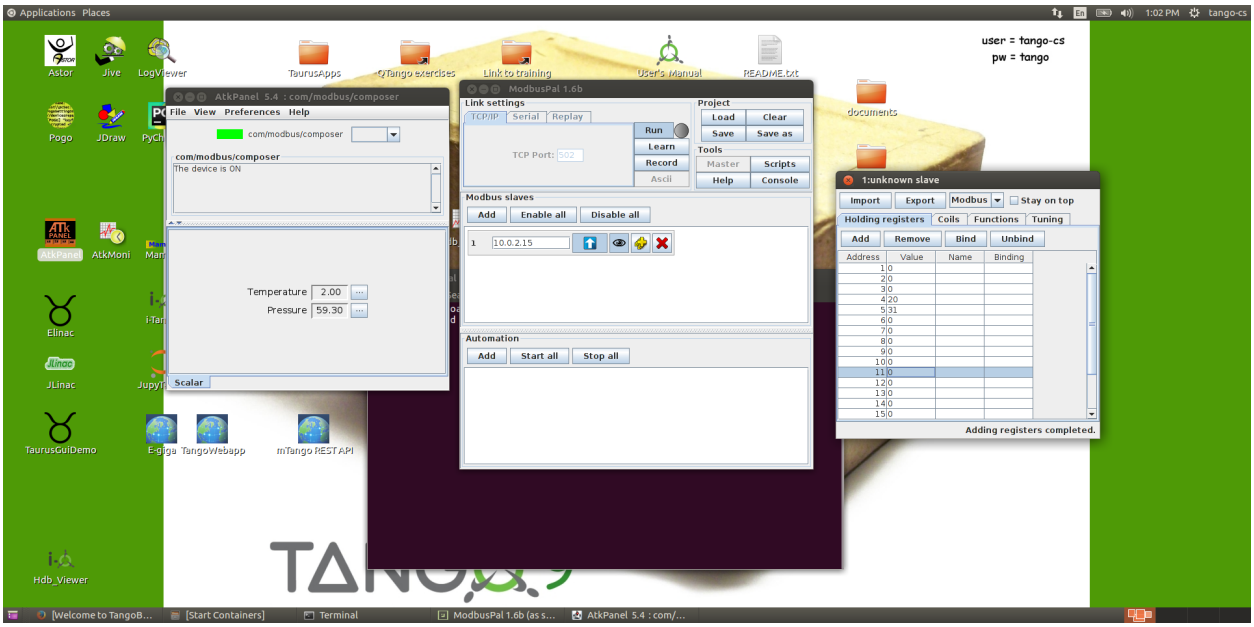


Fig. 14: View on a ModbusComposer device and configured ModbusPal simulator.

JupyTango

JupyTango is a Jupyter featuring Tango related kernels. With JupyterLab you may interact and do scripting for Tango through a web browser.

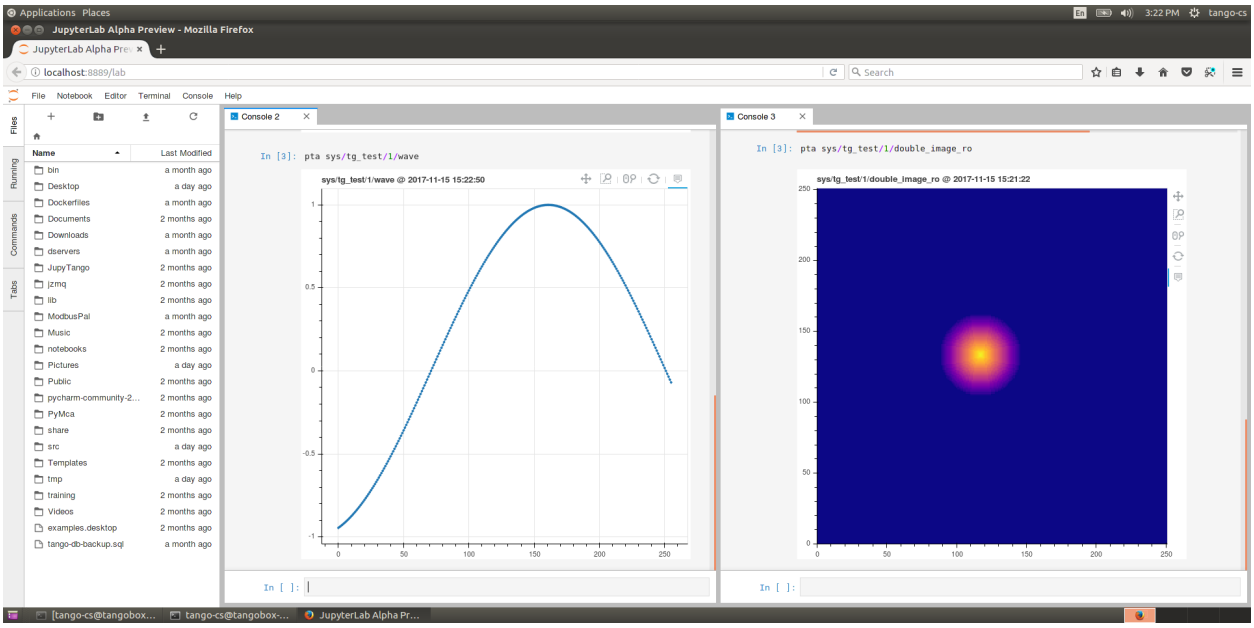


Fig. 15: Browser window with JupyTango in action

In case you want to try it, here's the procedure:

1. start jupyterlab using our dedicated script: *jupytango*

2. a new browser tab is automagically opened with the right URL: `localhost:8888/lab?`. Please be patient, it may take a while on VM.
3. the very first connection to the service requires a 'token' which is printed in the jupytango console
4. once in jupyterlab, click the JupyTango icon to open a notebook with the appropriate kernel
5. enjoy!

Here are the JupyTango additions to itango:

Plotting a tango attribute

Syntax:

```
pta [options] <tab for device selection> + <tab for attribute selection>
```

Supported options:

- *-w* or *–width*: plot width in pixels
- *-h* or *–height*: plot height in pixels

Monitoring a tango attribute: Syntax:

```
tm [options]  + <tab for device selection> + <tab for attribute selection>
```

Supported options:

- *-w* or *–width*: plot width in pixels
- *-h* or *–height*: plot height in pixels
- *-p* or *–period*: plot refresh period in [0.1, 5] seconds - defaults to 1s
- *-d* or *–depth*: scalar attribute history depth in [1, 3600] seconds - defaults to 900s

You can try to kill the monitored device will the JupyTango monitor is running to see how errors are handled.

JLinac and Elinac simulation

To start simulation, you need to run **tangobox-sim** container (use **`docker start tangobox-sim`** to start it from a terminal). It is also important to make sure that all related device servers are running. The easiest way to do it is to check it in Astor - a bulb next to *tangobox-sim* should be green.

Don't worry about warnings during Elinac's initialization.

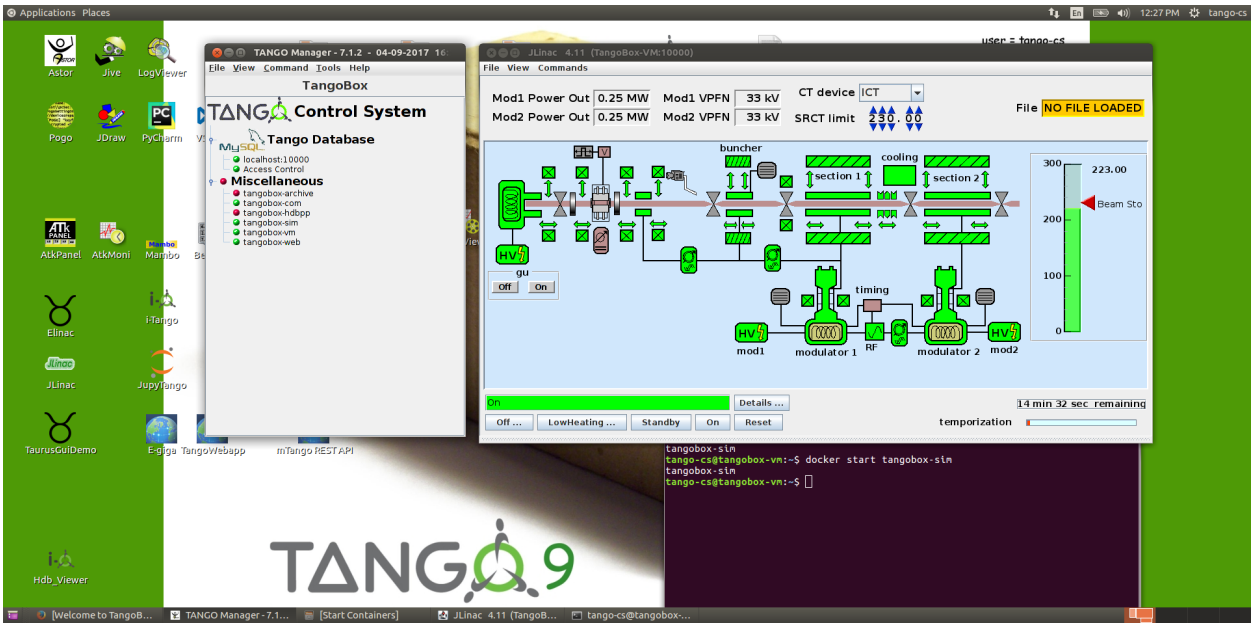


Fig. 16: JLinac simulation running.

HDB/TDB/SNAP Archiving (Mambo, Bensikin)

Prior to use *HDB/TDB* (*Mambo*) or *SNAP* (*Bensikin*) you need to make sure that the **tangobox-archiving** container and related device servers are running:

- Call **`docker start tangobox-archive`** on a terminal.
- Start **`Astor`** and check if the *tangobox-archive* node is green.

Then, you may start **`Mambo`** or **`Bensikin`** by clicking icons on the desktop.

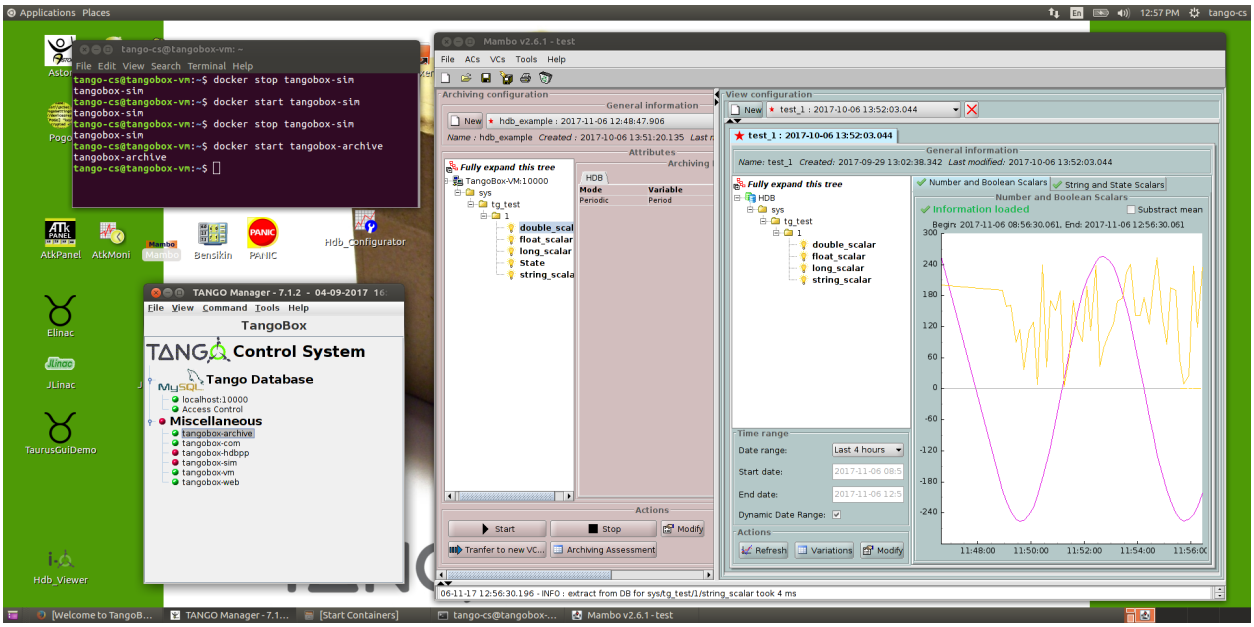


Fig. 17: Screen of running Mambo
Please take note of a green bulb of the *tangobox-archive* node in the Astor window.

HDB++ Archiving

To use *HDB++* and its tools (*HDB Configurator* and *HDB Viewer*) please make sure that the **tangobox-hdbpp** container and related device servers are running:

- Call **`docker start tangobox-hdbpp`** on a terminal.
- Start **`Astor`** and check if the *tangobox-hdbpp* node is green.

Then, you may start **`HDB Configurator`** or **`HDB Viewer`** by clicking icons on the desktop.

E-giga

`E-giga` is a web application for archiving data visualization (HDB/TDB and HDB++). The TangoBox deployment uses HDB/TDB.

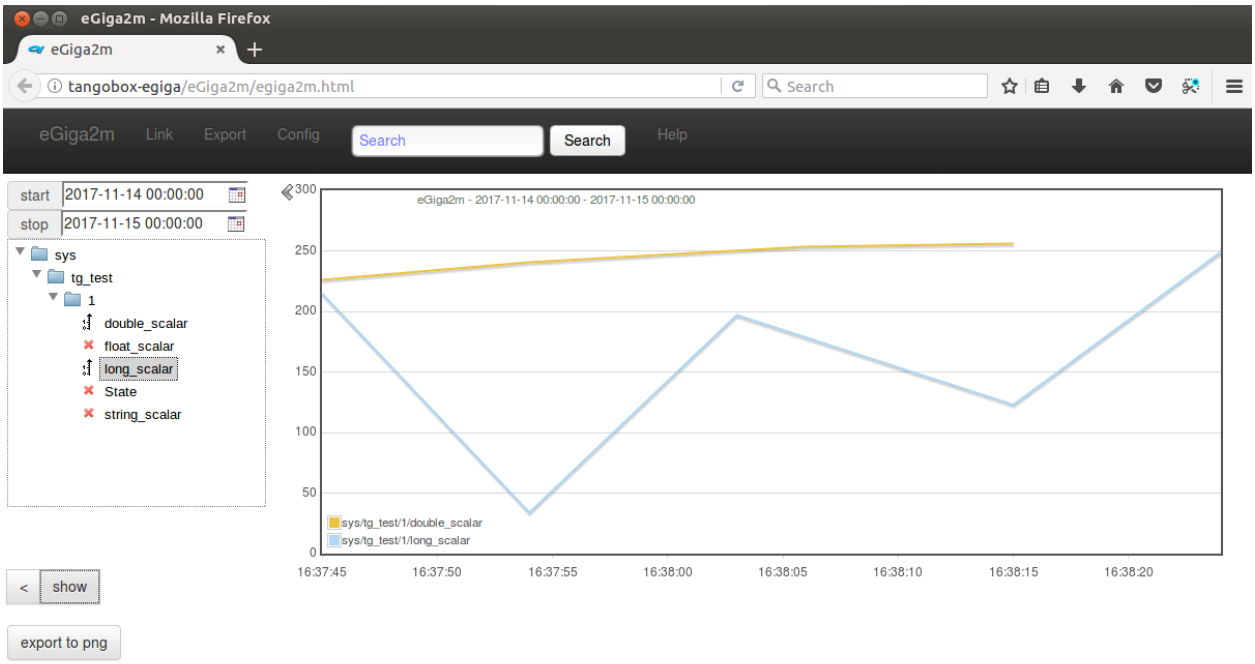


Fig. 18: E-giga in a web browser window

To use e-giga following conditions must be fulfilled:

- *tangobox-archive* and *tangobox-web* containers must be started and archiving device servers must be running
- use i.e. **Mambo** to enable data archiving for HDB database. It is required. If you do not see any attributes in *E-giga* it is probably due to archiving being disabled. Check with **`Astor`** if the tangobox-archive LED is green and with **`Mambo`** if there are any attributes configured to be archivied.

To open browser with **`E-giga`** click on the relate desktop icon.

Note: Please keep in mind that you should not rebuild **tangobox-web** image because its configuration is not included in Dockerfile (it requires in-container config).

Tango WebApp

Tango may be available through a web browser. *Tango WebApp* is a general purpose Tango web application. You may try it on the TangoBox.

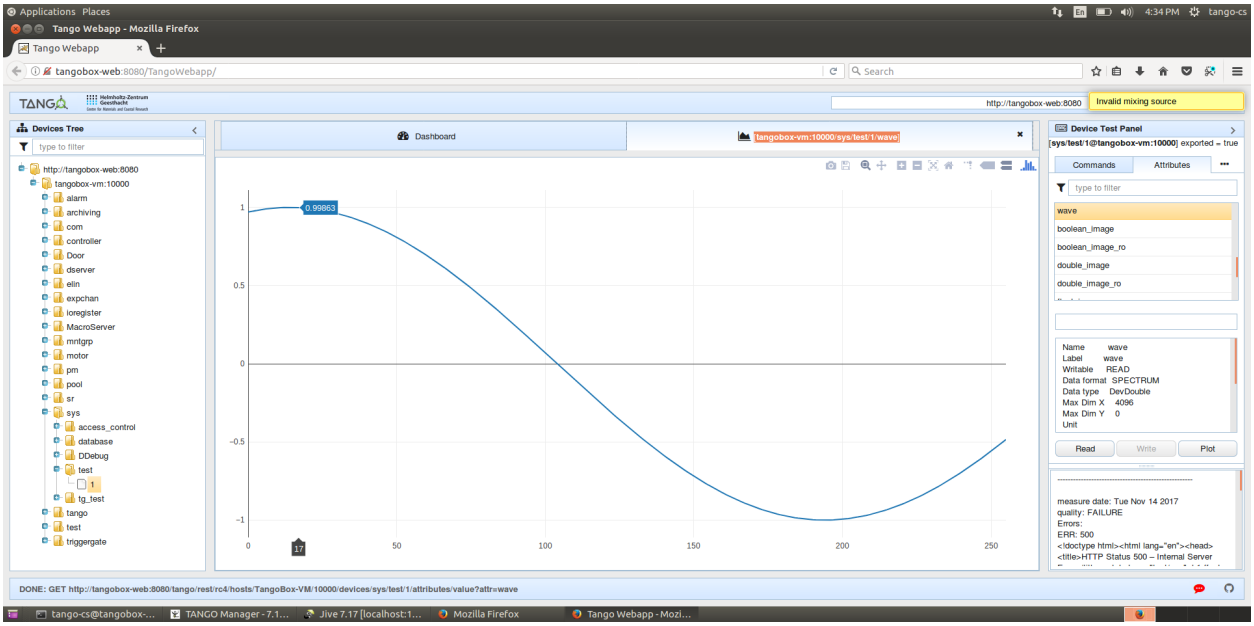


Fig. 19: A screenshot of Tango WebApp in a browser

To play with **Tango WebApp** make sure that the ‘tangobox-web` container is running (use **docker start tangobox-web** to start it from a terminal). Then, you may open a browser with a related desktop icon. Use username *tango-cs* and password *tango* to log-in.

REST API

Tango Controls specifies REST API interface and provides its reference implementation. For details see *REST API documentation*

The **TangoBox** comes with REST API installed. The *tangobox-web* container must be started to play with it. Invoke **docker start tangobox-web**.

Related deskto icon opens a web browser pointing to REST API interface. The REST server requires authentication. User is *tango-cs* and password is *tango*.

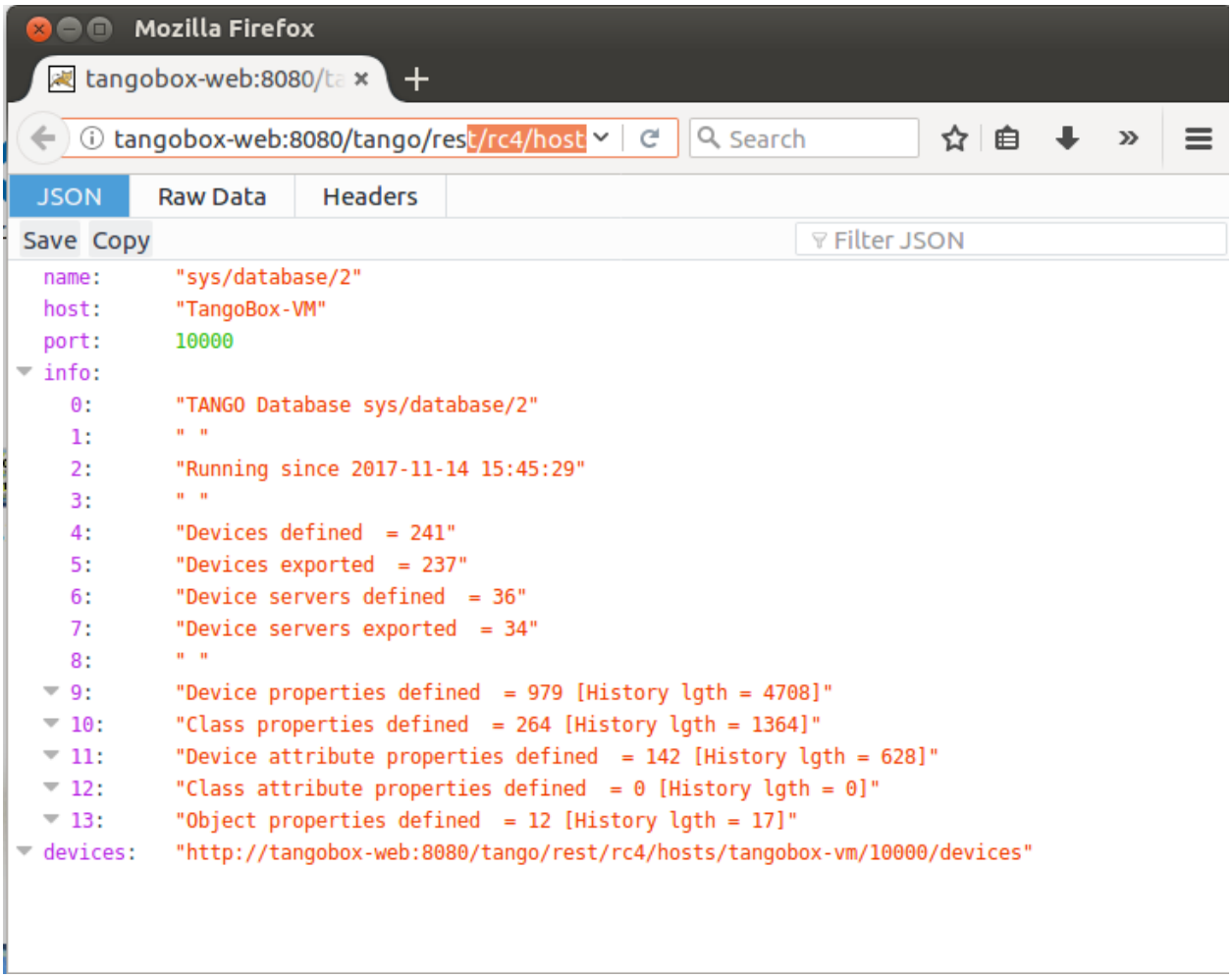


Fig. 20: A web browser window presenting JSON response of the Tango REST server

If you would like to play with it with other tools (Python, curl) it is avaialabe at the following address: *http://tangobox-web:8080/tango/rest/rc4/hosts/tangobox-vm/10000*.

Note: Please keep in mind that you should not rebuild **tangobox-web** image because its configuration is not included in Dockerfile (it requires in-container config).

Sardana

Sardana is a software suite for Supervision, Control and Data Acquisition in scientific installations. I t aims to reduce cost and time of design, development and support of the control and data acquisition systems. For more information about it please refer to Sardana documentation.

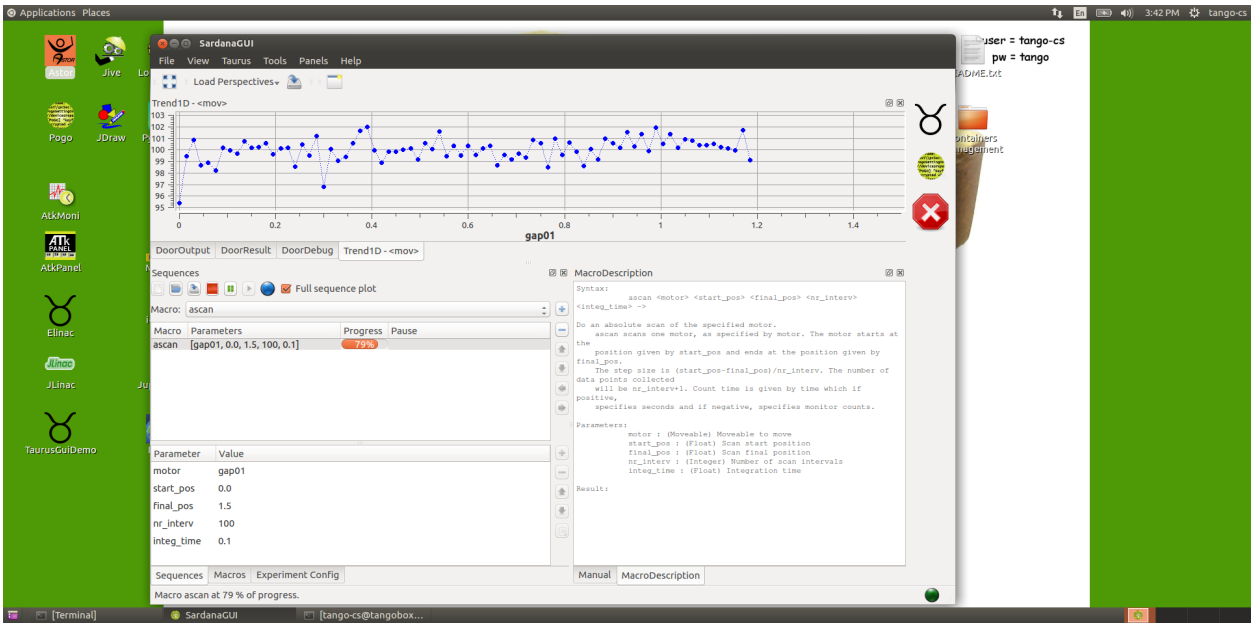


Fig. 21: SardanaGUI in action

To play with Sardana the **tangobox-sardana** container has to be started. Open a terminal and call **`docker start tangobox-sardana`**. Then, you may double-click the *SardanaGUI* icon on the desktop or run it from a terminal (type **`SardanaGUI`**).

4.9 PyTango and Taurus on Windows

Intended audience: administrators, developers

4.10 Binary packages

Intended audience: users, developers, administrators

Table 1: Binary packages for Tango are maintained to make it easier for developers to install the latest builds of Tango without having to compile them.

Tango Source Distribution	C++ source package with Java binaries	
PyTango	Library for device servers and clients in Python	
JTango	Library for device servers and clients in Java	
TangORB	Library for device servers and client in Java on Android	
Windows	Windows 32-bit/64-bit	
Conda packages	Tango conda packages	

4.11 Patches

Intended audience: developers, administrators

To apply any patches downloaded from this page, first go to the directory where Tango source distribution has been extracted e.g. *cd ~/tango-9.3.5*, then type the command :

```
patch -p1 < "patch_file"
```

4.11.1 omniORB patches

omniORB 4.2 bug (described in bug 794) is fixed in omniORB >= 4.2.2 or if you apply dii_race.patch patch file to omniORB < 4.2.2.

To apply this patch, copy the patch file to the directory where you extracted omniORB e.g. ~/omniORB-4.2.1, then type the command:

```
patch -p0 < dii_race.patch
```

4.11.2 Version 9.3.5 source patches

None yet.

4.11.3 Version 9.3.4 source patches

None yet.

4.11.4 Version 9.2.2 source patches

Bug 787, Bug 788, Bug 789, Bug 790, Bug 791 and Bug 792 are all fixed if you apply p922_1.diff patch file.

Then go to your build directory and run *make* followed by *make install*.

4.11.5 Version 9.1.0 source patches

Bug 745, Bug 748, Bug 749, Bug 752 and Bug 753 are all fixed if you apply p910_2.diff patch file.

Then go to your build directory and run *make* followed by *make install*. As usual, this patch assumes the previous patch(es) for the Tango release has been already applied.

Bug 741 is fixed if you apply p910_1.diff patch file. Then go to your build directory and run *make* followed by *make install*.

4.11.6 Version 8.1.2 source patches

Bug 662, Bug 663: These two bugs are fixed if you apply p812_4.diff patch file. Then go to your build directory and run *make* followed by *make install*.

This patch supposed that previous one (p812_3.diff) has been already applied.

Bug 646: This bug is fixed if you apply p812_3.diff patch file. Then go to your build directory and run *make* followed by *make install*.

This patch supposed that previous one (p812_2.diff) has been already applied.

Bug 631, Bug 632, Bug 638: These 3 bugs are all fixed if you apply p812_2.diff patch file. Then go to your build directory and run *make* followed by *make install*.

Bug 624, Bug 625: These 2 bugs are all fixed if you apply p812_1.diff patch file. Then go to your build directory and run *make* followed by *make install*.

4.11.7 Version 8.0.5 source patches

Bug 528, Bug 530, Bug 531, Bug 533, Bug 534, Bug 536: These 6 bugs are all fixed if you apply p805_1.diff patch file. Then go to your build directory and run *make* followed by *make install*.

Bug 545, Bug 546: These 2 bugs and some DeviceProxy class thread safety issues are all fixed if you apply p805_2.diff patch file. Then go to your build directory and run *make* followed by *make install*.

CHAPTER

FIVE

GETTING STARTED

Intended audience: beginners, all, ***Programming language:*** all

In this section we will guide you step-by-step to help you getting started with Tango-Controls.

We assume that Tango-Controls has been already installed in your environment. Otherwise, if you have to install Tango-Controls on your own, please, read the installation guide first: *Installation*

Note: You may identify yourself with one of the following roles and use proviodeds links:

End user	*Index*	*End-user applications guide*
Developer	*Index*	*How to develop for Tango Controls*
Administrator	*Index*	*Administration applications guide*

Table of contents of this section:

5.1 First steps with Tango Controls

Intended audience: all, ***Programming language:*** all

- In the *Overview* you will find basic information on Tango Controls. It will let you understand concepts of Tango Controls and help dancing it.
- You may also start with *trying Tango Controls* either with a preconfigured virtual machine or installing basic setup on your own computer.
- If you would like to install Tango Controls please look on *Installation guides*.
- To start connecting your devices to Tango Controls you should probably:
  - browse Device Classes Catalogue to find device servers for your equipment
  - read *how to start a device server*
  - or read *Your first C++ TANGO device class* and follow a guide *How to write your device class* if your device is not yet supported by any existing *device server*.
- To write your first C++ client see *Writing your first C++ TANGO client*.
- At the beginning you may also be interested in how to use provided tools: *Jive*, *ATKPanel*, *Astor*, *JDraw* or *Pogo*.
- You may be also interested in the content of Tango Controls web page

5.2 End-user applications guide

Intended audience: user

If you are end-user you are probably interested in documentation for *tools delivered with Tango Controls*. Below, you will find a list of tools a beginner user usually needs to know.

5.2.1 Jive

It is a tool used to configure components of the Tango Controls and browse a static *Tango Database*. See *Jive Manual*.

5.2.2 ATKPanel

ATKPanel is a simple application which shows (and allows to modify or invoke) device *state*, *attributes* and *commands*. Thus it allows to test and control all devices in the system. The tool is delivered together with Tango Controls. It may be opened as a stand-alone application or invoked from Jive. See *ATKPanel Manual*.

5.2.3 LogViewer

Tango provides a logging facility. You may use it with the *LogViewer application* delivered with Tango Controls.

5.3 How to develop for Tango Controls

Intended audience: developers, ***Programming language:*** all

Here you will find recipies on how-to develop the Tango Controls on various systems.

5.3.1 Getting started with JTango (Java implementation of Tango-Controls)

Intended audience: beginner developers, ***Programming language:*** java

Developing your first Java TANGO client

Developing your first Java TANGO device class

Intended audience: beginner developers, ***Programming language:*** java

In this section we describe how one can start developing Tango device server using Java.

Three methods will be described:

1. Using jtango-maven-archetype
2. Using POGO
3. Starting from scratch

Prerequisites

- Java >1.7
- Maven >3

- Tango-Controls environment (Tango Database aka Tango host is deployed)

Using jtango-maven-archetype

Perhaps the simplest way to start to develop your first Tango device server in Java is to use jtango-maven-archetype.

Maven is an Apache project and it is widely used in Java development nowdays. More information can be found in the Internet. Here we just name main features of Maven:

First of all Maven is a build system, i.e. it automatizes the build process of the project. As Maven is a plugin platform various plugins are used to achieve the desired result e.g. define compilation target (aka **`javac -target 1.8`**) or package the project into a single executable jar.

Secondly Maven automatically manages dependencies (required versions are being automatically downloaded from so called Maven central repository from the Internet).

Finally Maven provides a way to generate skeleton projects. This section is based on this feature.

So to start execute the following command:

```
$> mvn archetype:generate \
      -DarchetypeGroupId=org.tango-controls \
      -DarchetypeArtifactId=jtango-maven-archetype \
      -DarchetypeVersion=1.4
```

This command generates skeleton project using special Maven artifact that defines the template of the project. While generating new project you have to define several properties:

- **groupId** – target project's groupId. Typically it is reversed domain name of the company e.g. com.company
- **artifactId** – target project's artifactId. This is can be considered as the name of the target executable. This value must follow java class naming conventions e.g. *MyDevice*
- **version** – target project version. Simply the first version of the project e.g. *1.0-SNAPSHOT*
- **package** – Java package for newly generated class. Typically can be left as default i.e. **groupId**
- **license** – name of the license under which the project is distributed e.g. *LGPL-3*, *GPL*, *MIT* etc
- **organization** – name of the organization that maintains the project e.g. *Company*
- **organization-url** – organization's URL e.g. *http://www.company.com*
- **author-name** – name of the author/maintainer e.g. *JoeDoe*
- **author-email** – author/maintainer's email e.g. *joe.doe@company.com*
- **facility** – facility at which project is being developed e.g. *DESY*, *ESRF*, etc
- **platform** – Windows, MacOS, Unix/Linux etc. Typically Java projects will have *All* in this property
- **family** – as in POGO. Typically Java high level projects will have *SoftwareSystem* in this property
- **bus** – bus to the device (underlying hardware) e.g. *Serial*. For Java this might be *NA* if there is no real hardware associated with this Tango server.
- **jtango-version** – a version of JTango dependency or *LATEST* if you are know what are you doing.

Latest version of JTango is

The following output indicates that project has been successfully generated:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Now you can goto to the project folder and build it:

```
$>cd MyDevice
$>mvn package
$>java -jar target/MyDevice-1.0-SNAPSHOT.jar development
```

Assuming that Tango-Controls environment is set up properly (TODO ref) and MyDevice/development (TODO ref) server is defined in the Tango Database the later command will start the device server.

Now using your favorite IDE open the newly generated project and develop your JTango server. Please read more in JTango documentation.

Using POGO

See *POGO documentation*.

Starting from scratch

Example Java device and detailed documentation can be found in the JTango documentation.

For a more complete guide on JTango please refer to the JTango documentation.

5.3.2 Getting started with cppTango (C++ implementation of Tango-Controls)

Intended audience: beginner developers, ***Programming language:*** c++

Writing your first C++ TANGO client

Intended audience: beginer developers, ***Programming language:*** c++

The quickest way of getting started is by studying this example :

```
/*
 * example of a client using the TANGO C++ api.
 */
#include <tango.h>
using namespace Tango;
int main(unsigned int argc, char **argv)
{
    try
    {

//
// create a connection to a TANGO device
//

        DeviceProxy *device = new DeviceProxy("sys/database/2");

//
// Ping the device
//

        device->ping();

//
// Execute a command on the device and extract the reply as a string
//

        string db_info;
        DeviceData cmd_reply;
        cmd_reply = device->command_inout("DbInfo");
        cmd_reply >> db_info;
        cout << "Command reply " << db_info << endl;

//
// Read a device attribute (string data type)
//

        string spr;
        DeviceAttribute att_reply;
        att_reply = device->read_attribute("StoredProcedureRelease");
        att_reply >> spr;
        cout << "Database device stored procedure release: " << spr << endl;
    }
    catch (DevFailed &e)
    {
        Except::print_exception(e);
        exit(-1);
    }
}
```

Modify this example to fit your device server or client's needs, compile it and link with the library -ltango. Forget about those painful early TANGO days when you had to learn CORBA and manipulate Any's. Life's going to easy and fun from now on !

TANGO C++ Quick start

Intended audience: developers

This quick tutorial will guide you to some fundamental TANGO concepts and how to do the first steps.

Fundamental TANGO concepts

Before starting to work in TANGO, it is important to know at least some concepts.

TANGO is a control system framework, composed by a set of devices running somewhere on the network, that communicate each other through on *CORBA* and *ZeroMQ*. Everything which needs to be controlled is modeled as a *Device*.

Every device in the network is identified by the Fully Qualified Domain Name (FQDN), a unique case insensitive name in the format:

tango://hostname.full.domain.name:port/domain/family/member

Usually the environment variable TANGO_HOST is used by clients/servers to know on which host and port the Database server is running:

TANGO_HOST=tango://hostname.full.domain.name:port

or short form

TANGO_HOST=hostname:port

In this way is possible to refers a device as *domain/family/member*

eg. DishArray1/Receiver/01 LAB/PowerSupply/01

Each device has a series of *attributes*, *pipes*, *properties* and *commands*.

An *attribute* is identified by a name in a device and represents the data fields a Device wants clients to Read or Write or receive Events. It has a value that can be read or can also be changed (read-write attributes). Each attribute has a well known, fixed data type.

A *pipe* is a kind of attribute and it consists in a data stream or channel for exchanging a stream of any Tango data type. Data types can be sent individually or grouped together in a Blob

A *property* is identified by a name in a device. It consists in a data stored in the database and used to configure Devices at startup.

A *command* is also identified by a name and is the action of a device the clients needs to execute. A command may or not receive a parameter and may or not return a value when it is executed.

Any device has at least a *State* and *Status* attributes and State, Status and *Init* commands. Reading the State or Status attributes has the same effect as executing the State or Status commands.

Devices belong to a *Device Class* and are hosted in a *Device Server*.

The *device class* implements a generic Device behaviour. Properties are used to configure the specific Device. Device Classes can be implemented in *Python*, *C++* or *Java*.

TANGO devices live inside a operating system process called *TANGO Device Server*. This server acts as a container of devices. A device server can host multiple devices of multiple TANGO classes. Devices are, therefore, only accessible when the corresponding TANGO Device Server is running.

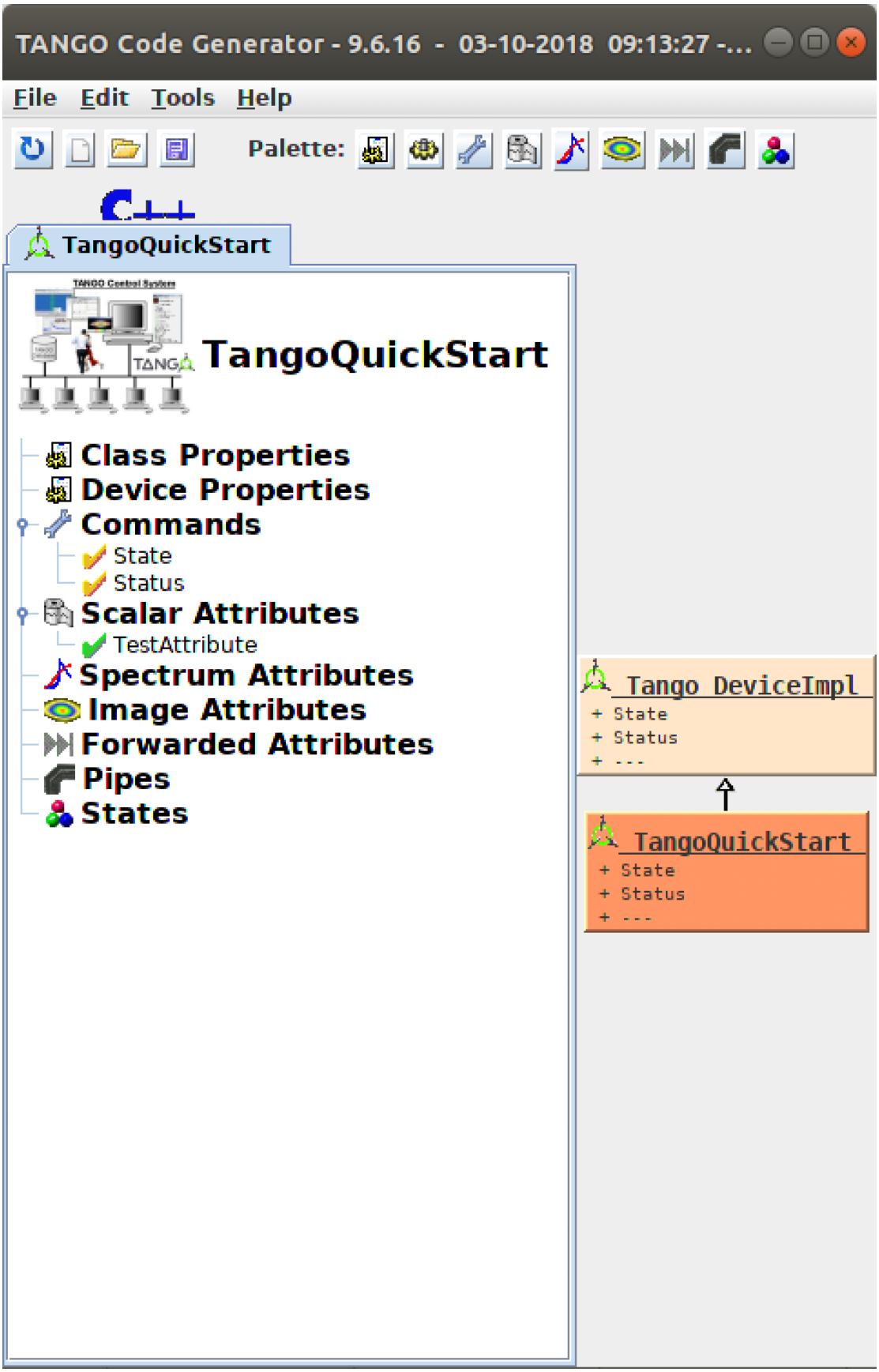
A special TANGO device server called the *TANGO Database Server* will act as a naming service between TANGO servers and clients. This server has a known address where it can be reached that is the addresses setted in *TANGO_HOST* environment variable.

Step 1: installation

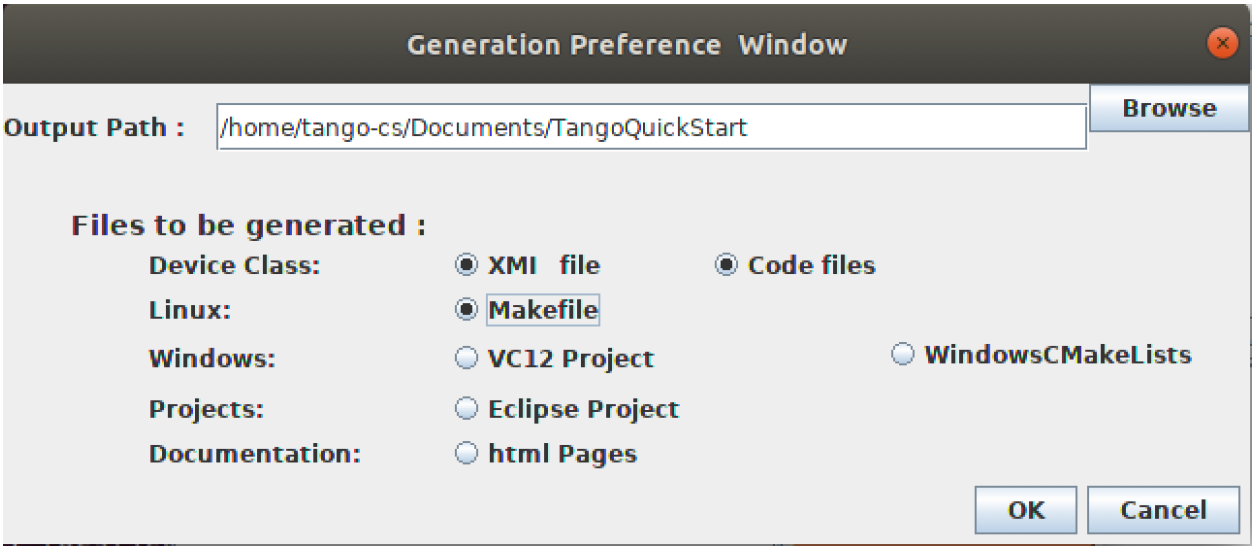
This chapter assumes that you have already installed Tango in your local computer or in your network. If you need to install TANGO, please reference to *the documentation*.

Step 2: create a device class

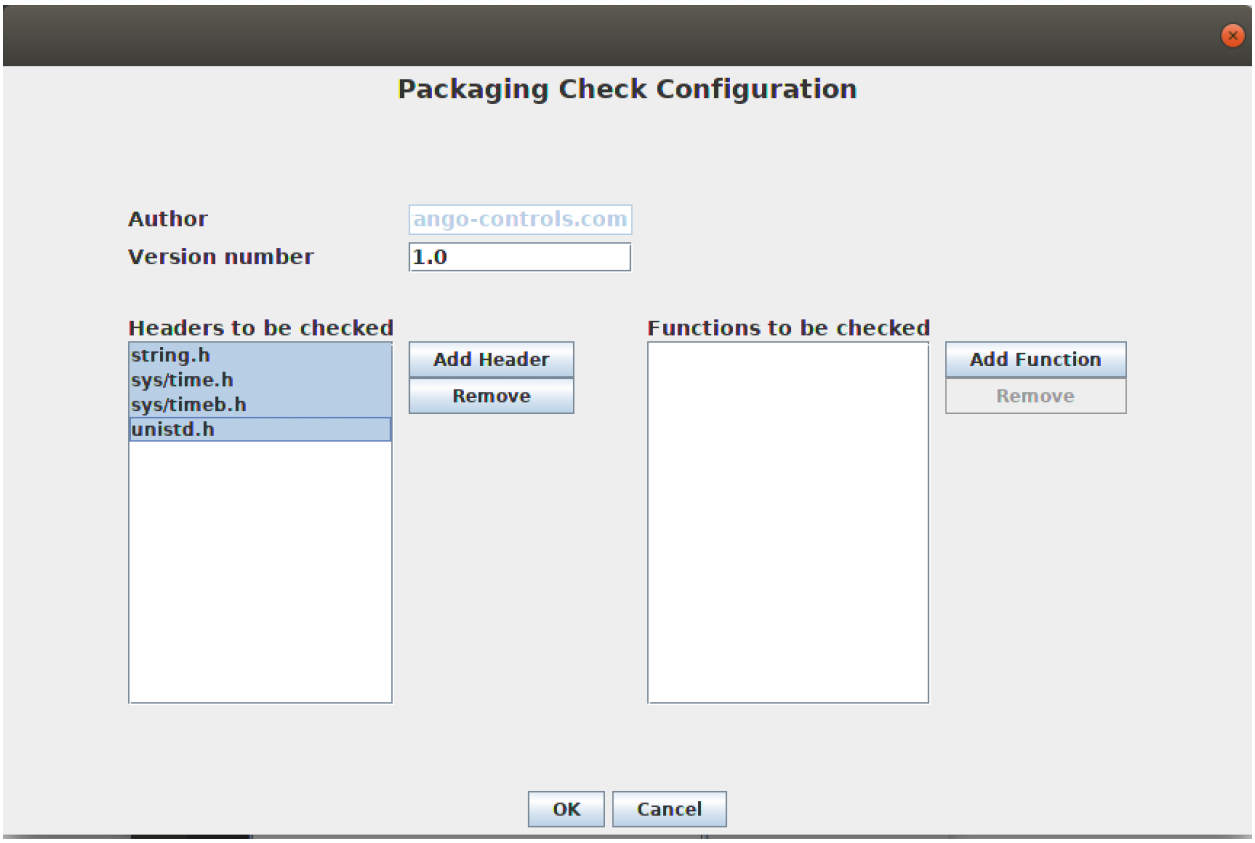
A useful tool to create a TANGO Class skeleton is **POGO**. POGO is a tool that permits you the generate code and the documentation of a class model in the different programming languages (c++, Java and Python).


TANGO Code Generator - 9.6.16 - 03-10-2018 09:13:27 -...
File Edit Tools Help
Palette:
TangoQuickStart
TangoQuickStart
Class Properties
Device Properties
Commands
State
Status
Scalar Attributes
TestAttribute
Spectrum Attributes
Image Attributes
Forwarded Attributes
Pipes
States
Tango DeviceImpl
+ State
+ Status
+ ---
TangoQuickStart
+ State
+ Status
+ ---

In the example a c++ Tango class, named TangoQuickStart, has been created. To generate the code go to *File* -> *Generate* and press *OK* in the window.



In order to compile properly the class, you need to add the packaging. Into POGO, go to *File* -> *Export Package* and check all the headers. The tool will create a new folder with the packages.



Once you created a skeleton and added the header, you can add your code into the class and compile it. In *this section* you can find a guide how to implement a c++ device class. You may find more on device API *here*.

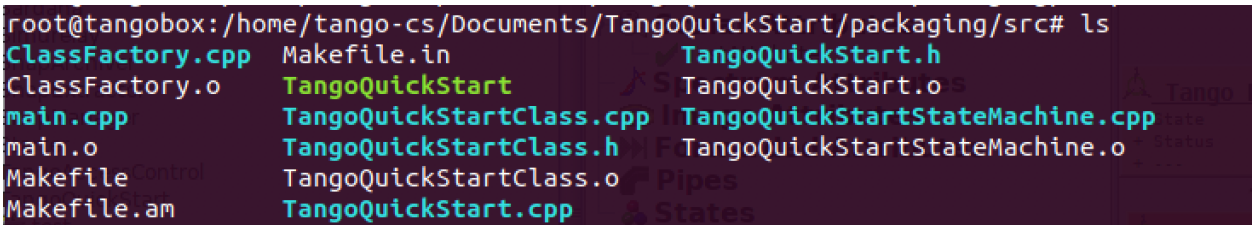
In *this link* you can find the POGO guide.

Step 3: compile the device class

To compile the files, go to the folder that you choose to store your class and execute the following command:

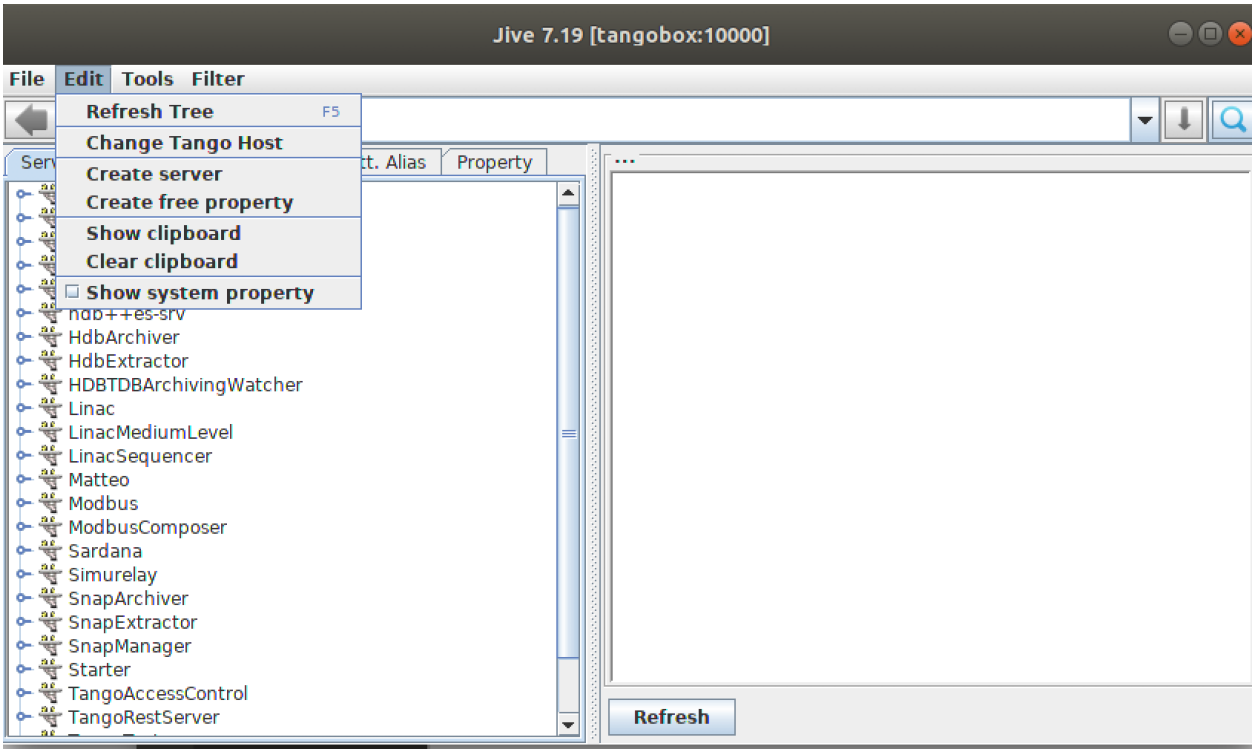
```
cd packaging
./autogen.sh
./configure --prefix=$HOME/packaging
make
make install
```

If everything works, you will see the files in the src directory

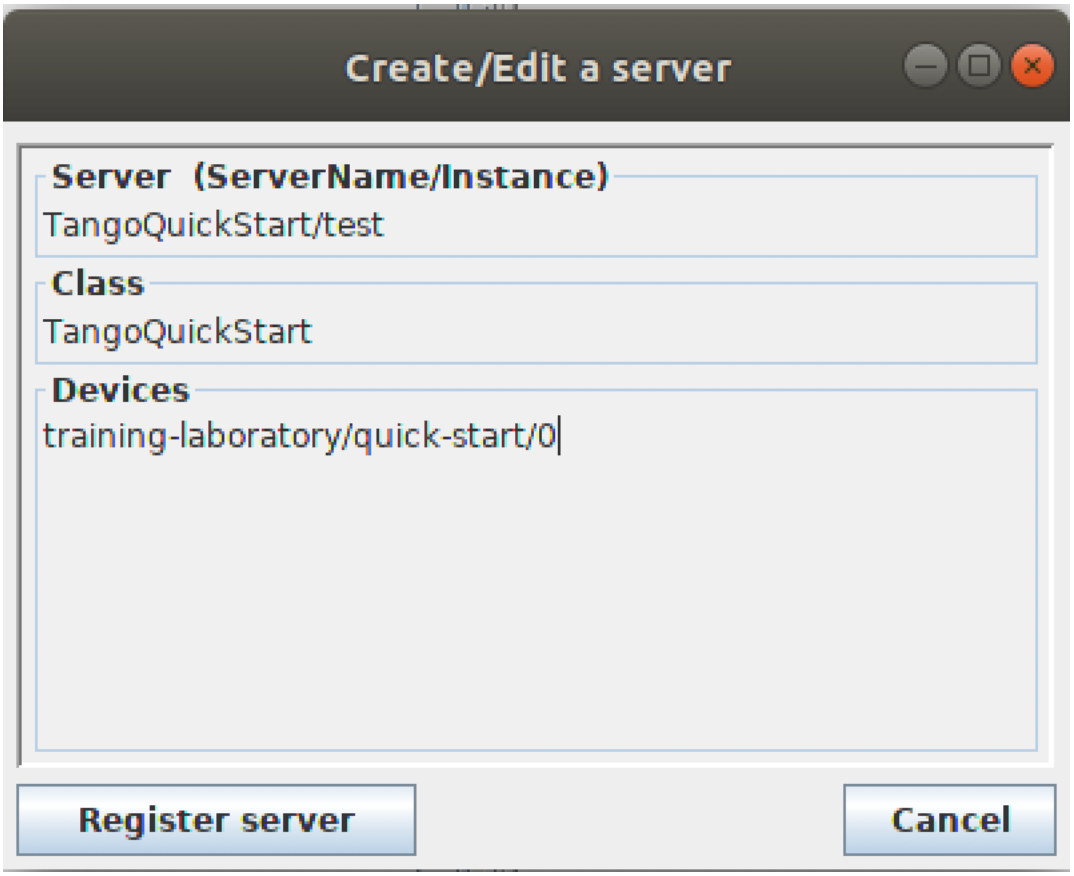


Step 4: register the device

Open Jive and go to **`Edit -> Create Server`**



Fill the form as follow:



Remember that in the server you must add the ServerName, that is the name of the Device Server and che instance. In Class you have to insert che correct ClassName and in Device you can add one or more device, following the convention:

domain/family/member

After, click *Register server*.

Step 5: start the device

In order to start the device, using the command line, go to the folder where you compiled your c++ class

```
cd packaging/src/
```

and use the command with the following sintax:

<TangoClassName> <instance>

Where *<TangoClassName>* is the name of the class that you created, and the instance is the name of the instance into the TANGO Database.

In the example:

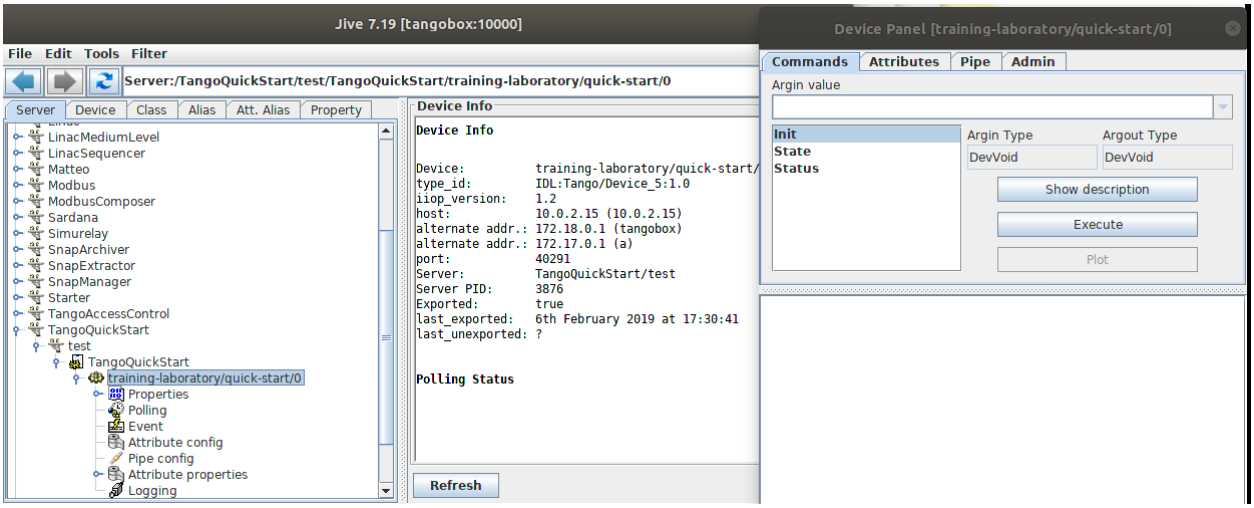
```
root@tangobox:/home/tango-cs/Documents/TangoQuickStart/packaging/src# ./TangoQui
ckStart test
Ready to accept request
```

If everything is ok, the following message appears:

```
Ready to accept request
```

Step 6: explore the device

When finish, you can explore your device using Jive.



Your first C++ TANGO device class

Intended audience: beginner developers, ***Programming language:*** c++

The code given in this chapter as example has been generated using POGO. Pogo is a code generator for Tango device server. See POGO home page for more information about POGO. The following examples briefly describe how to write device class with commands which receives and return different kind of Tango data types and also how to write device attributes The device class implements 5 commands and 3 attributes. The commands are :

- The command **DevSimple** deals with simple Tango data type
- The command **DevString** deals with Tango strings
- **DevArray** receive and return an array of simple Tango data type
- **DevStrArray** which does not receive any data but which returns an array of strings
- **DevStruct** which also does not receive data but which returns one of the two Tango composed types (DevVarDoubleStringArray)

For all these commands, the default behavior of the state machine (command always allowed) is acceptable. The attributes are :

- A spectrum type attribute of the Tango string type called **StrAttr**
- A readable attribute of the Tango::DevLong type called **LongRdAttr**. This attribute is linked with the following writable attribute
- A writable attribute also of the Tango::DevLong type called **LongWrAttr**.

Since release 9, a Tango device also supports pipe. This is an advanced feature reserved for some specific cases. Therefore, there is no device pipe example in this Getting started chapter.

The commands and attributes code

For each command called DevXxxx, pogo generates in the device class a method named dev_xxx which will be executed when the command is requested by a client. In this chapter, the name of the device class is *DocDs*

The DevSimple command

This method receives a Tango::DevFloat type and also returns a data of the Tango::DevFloat type which is simply the double of the input value. The code for the method executed by this command is the following:

```
1  Tango::DevFloat DocDs::dev_simple(Tango::DevFloat argin)
2  {
3          Tango::DevFloat argout ;
4          DEBUG_STREAM << "DocDs::dev_simple(): entering... !" << endl;
5
6          //      Add your own code to control device here
7
8          argout = argin * 2;
9          return argout;
10 }
```

This method is fairly simple. The received data is passed to the method as its argument. It is

doubled at line 8 and the method simply returns the result.

The DevArray command

This method receives a data of the Tango::DevVarLongArray type and also returns a data of the Tango::DevVarLongArray type. Each element of the array is doubled. The code for the method executed by the command is the following :

```
1  Tango::DevVarLongArray *DocDs::dev_array(const Tango::DevVarLongArray *argin)
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //--------------------------------------------------------------
8          Tango::DevVarLongArray  *argout  = new Tango::DevVarLongArray();
9
10         DEBUG_STREAM << "DocDs::dev_array(): entering... !" << endl;
11
12         //      Add your own code to control device here
13
14         long argin_length = argin->length();
15         argout->length(argin_length);
16         for (int i = 0;i < argin_length;i++)
17                 (*argout)[i] = (*argin)[i] * 2;
18
19         return argout;
20 }
```

The argout data array is created at line 8. Its length is set at line 15 from the input argument length. The array is populated at line 16,17 and returned. This method allocates memory for the argout array. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated array without copying. Look at chapter [Data exchange] for all the details.

The DevString command

This method receives a data of the Tango::DevString type and also returns a data of the Tango::DevString type. The command simply displays the content of the input string and returns a hard-coded string. The code for the method executed by the command is the following :

```
1  Tango::DevString DocDs::dev_string(Tango::DevString argin)
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //--------------------------------------------------------------
8          Tango::DevString        argout;
9          DEBUG_STREAM << "DocDs::dev_string(): entering... !" << endl;
10
11         //      Add your own code to control device here
12
13         cout << "the received string is " << argin << endl;
14
```

(continues on next page)

(continued from previous page)

```
15          string str("Am I a good Tango dancer ?");
16          argout = new char[str.size() + 1];
17          strcpy(argout,str.c_str());
18
19          return argout;
20  }
```

The argout string is created at line 8. Internally, this method is using a standard C++ string. Memory for the returned data is allocated at line 16 and is initialized at line 17. This method allocates memory for the argout string. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated string without copying. Look at chapter [Data exchange] for all the details.

The DevStrArray command

This method does not receive input data but returns an array of strings (Tango::DevVarStringArray type). The code for the method executed by this command is the following:

```
1   Tango::DevVarStringArray *DocDs::dev_str_array()
2   {
3           //      POGO has generated a method core with argout allocation.
4           //      If you would like to use a static reference without copying,
5           //      See "TANGO Device Server Programmer's Manual"
6           //              (chapter x.x)
7           //--------------------------------------------------------------
8           Tango::DevVarStringArray        *argout  = new Tango::DevVarStringArray();
9
10          DEBUG_STREAM << "DocDs::dev_str_array(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          argout->length(3);
15          (*argout)[0] = Tango::string_dup("Rumba");
16          (*argout)[1] = Tango::string_dup("Waltz");
17          string str("Jerck");
18          (*argout)[2] = Tango::string_dup(str.c_str());
19          return argout;
20  }
```

The argout data array is created at line 8. Its length is set at line 14. The array is populated at line 15,16 and 18. The last array element is initialized from a standard C++ string created at line 17. Note the usage of the *string_dup* function of the Tango namespace. This is necessary for strings array due to the CORBA memory allocation schema.

The DevStruct command

This method does not receive input data but returns a structure of the Tango::DevVarDoubleStringArray type. This type is a composed type with an array of double and an array of strings. The code for the method executed by this command is the following:

```
1  Tango::DevVarDoubleStringArray *DocDs::dev_struct()
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //--------------------------------------------------------
8          Tango::DevVarDoubleStringArray  *argout  = new␣
↪Tango::DevVarDoubleStringArray();
9
10         DEBUG_STREAM << "DocDs::dev_struct(): entering... !" << endl;
11
12         //      Add your own code to control device here
13
14         argout->dvalue.length(3);
15         argout->dvalue[0] = 0.0;
16         argout->dvalue[1] = 11.11;
17         argout->dvalue[2] = 22.22;
18
19         argout->svalue.length(2);
20         argout->svalue[0] = Tango::string_dup("Be Bop");
21         string str("Smurf");
22         argout->svalue[1] = Tango::string_dup(str.c_str());
23
24         return argout;
25 }
```

The argout data structure is created at line 8. The length of the double array in the output structure is set at line 14. The array is populated between lines 15 and 17. The length of the string array in the output structure is set at line 19. This string array is populated between lines 20 an 22 from a hard-coded string and from a standard C++ string. This method allocates memory for the argout data. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). Note the usage of the *string_dup* function of the Tango namespace. This is necessary for strings array due to the CORBA memory allocation schema.

The three attributes

Some data have been added to the definition of the device class in order to store attributes value. These data are (part of the class definition) :

```
1  protected :
2          //      Add your own data members here
3          //----------------------------------------
4          Tango::DevString        attr_str_array[5];
5          Tango::DevLong          attr_rd;
6          Tango::DevLong          attr_wr;
```

One data has been created for each attribute. As the StrAttr attribute is of type spectrum with a maximum X dimension

of 5, an array of length 5 has been reserved.

Several methods are necessary to implement these attributes. One method to read the hardware which is common to all readable attributes plus one read method for each readable attribute and one write method for each writable attribute. The code for these methods is the following :

```
void DocDs::read_attr_hardware(vector<long> &attr_list)
{
    DEBUG_STREAM << "DocDs::read_attr_hardware(vector<long> &attr_list) entering... "<<␣
↪endl;
// Add your own code here

    string att_name;
    for (long i = 0;i < attr_list.size();i++)
    {
        att_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();

       if (att_name == "LongRdAttr")
       {
           attr_rd = 5;
       }
   }
}

void DocDs::read_LongRdAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "DocDs::read_LongRdAttr(Tango::Attribute &attr) entering... "<<␣
↪endl;

    attr.set_value(&attr_rd);
}

void DocDs::read_LongWrAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "DocDs::read_LongWrAttr(Tango::Attribute &attr) entering... "<<␣
↪endl;

    attr.set_value(&attr_wr);
}

void DocDs::write_LongWrAttr(Tango::WAttribute &attr)
{
    DEBUG_STREAM << "DocDs::write_LongWrAttr(Tango::WAttribute &attr) entering... "<<␣
↪endl;

    attr.get_write_value(attr_wr);
    DEBUG_STREAM << "Value to be written = " << attr_wr << endl;
}

void DocDs::read_StrAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "DocDs::read_StrAttr(Tango::Attribute &attr) entering... "<< endl;

    attr_str_array[0] = const_cast<char *>("Rock");
```

(continues on next page)

(continued from previous page)

```
45      attr_str_array[1] = const_cast<char *>("Samba");
46
47      attr_set_value(attr_str_array, 2);
48  }
```

The *read_attr_hardware()* method is executed once when a client execute the read_attributes CORBA request whatever the number of attribute to be read is. The rule of this method is to read the hardware and to store the read values somewhere in the device object. In our example, only the LongRdAttr attribute internal value is set by this method at line 13. The method *read_LongRdAttr()* is executed by the read_attributes CORBA call when the LongRdAttr attribute is read but after the read_attr_hardware() method has been executed. Its rule is to set the attribute value in the TANGO core classes object representing the attribute. This is done at line 22. The method *read_LongWrAttr()* will be executed when the LongWrAttr attribute is read (after the *read_attr_hardware()* method). The attribute value is set at line 29. In the same manner, the method called *read_StrAttr()* will be executed when the attribute StrAttr is read. Its value is initialized in this method at line 44 and 45. There are several ways to code spectrum or image attribute of the DevString data type. A HowTo related to this topic is available on the Tango control system Web site. The *write_LongWrAttr()* method is executed when the LongWrAttr attribute value is set by a client. The new attribute value coming from the client is stored in the object data at line 36.

Pogo also generates a file called DocDsStateMachine.cpp (for a Tango device server class called DocDs). This file is used to store methods coding the device state machine. By default a allways allowed state machine is provided. For more information about coding the state machine, refer to the chapter Writing a device server.

5.3.3 Getting started with PyTango (Python implementation of Tango-Controls)

Intended audience: beginner developers, ***Programming language:*** python

Developing Python TANGO device class

5.4 Administration applications guide

Intended audience: beginners, administrators

Starting as Administrator for Tango Controls you should look on the following tools and topics:

5.4.1 Astor

Astor is a tool for management of Tango Controls system.

5.4.2 Jive

It is a tool used to configure components of the Tango Controls and browse a static *Tango Database*. See *Jive Manual*.

5.4.3 LogViewer

Tango provides a logging facility. You may use it with the *LogViewer application* delivered with Tango Controls.

5.4.4 Tango Database

Tango Admin is a command-line interface for Tango Database management.

5.4.5 Installation

Recipes on system installation are provided in a *dedicated section*.

Then, *Administration section* contains various related information.

CHAPTER

SIX

DEVELOPER'S GUIDE

Intended audience: developers, ***Programming language:*** all

In this section the process of how to write Tango device servers and clients (applications). In addtion to articles listed below you may be interested in *Tutorials* and *HOW-TOs for developers* or browse *all documents marked for developers*.

The section is organized as follows:

6.1 Overview

Intended audience: developers, ***Programming language:*** all

Tango is a developers toolkit. There are many libraries and tools for implemented device clients and servers.

6.1.1 C++ and Python

This clickable map shows the libraries and tools available for C++ and Python developers.

6.1.2 Java

This clickable map shows the libraries and tools available for Java developers.

6.2 General guidelines

Intended audience: developers, administrators, users, ***Programming language:*** all

6.2.1 Tango object naming (device, attribute and property)

Intended audience: developers, users, administrators, ***Programming language:*** all

Device name

A Tango device name is a three fields name. The field separator is the **/** character. The first field is named **domain**, the second field is named **family** and the last field is named **member**.A tango device name looks like

domain/family/member

It is a hierarchical notation. The member specifies which element within a family. The family specifies which kind of equipment within a domain. The domain groups devices related to which part of the accelerator/experiment they belongs to. At ESRF, some of the machine control system domain name are SR for the storage ring, TL1 for the transfer line 1 or SY for the synchrotron booster. For experiment, ID11 is the domain name for all devices belonging to the experiment behind insertion device 11. Here are some examples of Tango device name used at the ESRF :

- **sr/d-ct/1** : The current transformer. The domain part is sr for storage ring. The family part is d-ct for diagnostic/current transformer and the member part is 1
- **fe/v-pen/id11-1** : A Penning gauge. The domain part is fe for front-end. The family part is v-pen for vacuum/penning and the member name is id11-1 to specify that this is the first gauge on the front-end part after the insertion device 11

Full object name

The device name as described above is not enough to cover all Tango usage like device server without database or device access for multi control system. With the naming schema, we must also be able to name attribute and property. Therefore, the full naming schema is

[protocol://][host:port/]device_name[/attribute][->property][#dbase=xx]

The protocol, host, port, attribute, property and dbase fields are optional. The meaning of these fields are :

protocol: Specifies which protocol is used (Tango or Taco). Tango is the default

#dbase: The supported value for xx is *yes* and *no*. This field is used to specify that the device is a device served by a device server started with or without database usage. The default value is *dbase=yes*

host:port: This field has different meaning according to the dbase value. If *dbase=yes* (the default), the host is the host where the control system database server is running and port is the database server port. It has a higher priority than the value defined by the TANGO_HOST environment variable. If *dbase=no*, host is the host name where the device server process serving the device is running and port is the device server process port.

attribute: The attribute name

property: The property name

The host:port and dbase=xx fields are necessary only when creating the DeviceProxy object used to remotely access the device. The -> characters are used to specify a property name.

Some examples

Full device name examples

- **gizmo:20000/sr/d-ct/1** : Device sr/d-ct/1 running in a specified control system with the database server running on a host called gizmo and using the port number 20000. The TANGO_HOST environment variable will not be used.
- **tango://freak:2345/id11/rv/1#dbase=no** : Device served by a device server started without database. The server is running on a host called freak and use port number 2345. //freak:2345/id11/rv/1#dbase=no is also possible for the same device.
- **Taco://sy/ps-ki/1** : Taco device sy/ps-ki/1

Attribute name examples

- **id11/mot/1/Position** : Attribute position for device id11/mot/1
- **sr/d-ct/1/Lifetime** : Attribute lifetime for Tango device sr/d-ct/1

Attribute property name

- **id11/rv/1/temp->label** : Property label for attribute temp for device id11/rv/1.
- **sr/d-ct/1/Lifetime->unit** : The unit property for the Lifetime attribute of the sr/d-ct/1 device

Device property name

- **sr/d-ct/1->address** : the address property for device sr/d-ct/1

Class property name

- **Starter->doc_url** : The doc_url property for a class called Starter

Device and attribute name alias

Within Tango, each device or attribute can have an alias name defined in the database. Every time a device or an attribute name is requested by the API's, it is possible to use the alias. The alias is simply an open string stored in the database. The rule of the alias is to give device or attribute name a name more natural from the physicist point of view. Let's imagine that for experiment, the sample position is described by angles called teta and psi in physics book. It is more natural for physicist when they move the motor related to sample position to use *teta* and *psi* rather device name like *idxx/mot/1* or *idxx/mot/2*. An attribute alias is a synonym for the four fields used to name an attribute. For instance, the attribute *Current* of a power-supply device called *sr/ps/dipole* could have an alias *DipoleCurrent*. This alias can be used when creating an instance of a AttributeProxy class instead of the full attribute name which is *sr/ps/dipole/Current*. Device alias name are uniq within a Tango control system. Attribute alias name are also uniq within a Tango control system.

Reserved words and characters, limitations

From the naming schema described above, the reserved characters are **:**, **#**, **/** and the reserved string is : **->**. On top of that, the dbt_update tool (tool to fulfill database from the content of a file) reserved the **device** word

The device name, its domain, member and family fields and its alias are stored in the Tango database. The default maximum size for these items are :

Item	max length
device name	255
domain field	85
family field	85
member field	85
device alias name	255

The device name, the command name, the attribute name, the property name, the device alias name and the device server name are **case insensitive**.

6.3 10 things you should know about CORBA

Intended audience: developers, ***Programming language:*** all

1. **You don't need to know CORBA to work with TANGO**
2. CORBA is the acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture and it is a standard defined by the Object Management Group (OMG)
3. CORBA enables communication between software written in different languages and running on different computers
4. CORBA applications are composed of many objects; objects are running software that provides functionalities and that can represent something in the real world
5. Every object has a type which is defined with a language called IDL (Interface Definition Language)
6. An object has an interface and an implementation: this is the essence of CORBA because it allows *interoperability*.
7. CORBA allows an application to request an operation to be performed by a distributed object and for the results of the operation to be returned back to the application making the request.
8. CORBA is based on a *Remote Procedure Call* model
9. The TANGO Device is a CORBA Object
10. The TANGO Device Server is a CORBA Application

6.4 Tango Client

Intended audience: developers, ***Programming language:*** all

6.4.1 Writing a TANGO client using TANGO C++ APIs

Intended audience: developers, ***Programming language:*** c++

Introduction

TANGO devices and database are implemented using the TANGO device server model. To access them the user has the CORBA interface e.g. command_inout(), write_attributes() etc. defined by the idl file. These methods are very low-level and assume a good working knowledge of CORBA. In order to simplify this access, high-level api has been implemented which hides all CORBA aspects of TANGO. In addition the api hides details like how to connect to a device via the database, how to reconnect after a device has been restarted, how to correctly pack and unpack attributes and so on by implementing these in a manner transparent to the user. The api provides a unified error handling for all TANGO and CORBA errors. Unlike the CORBA C++ bindings the TANGO api supports native C++ data types e.g. strings and vectors.

This chapter describes how to use these API's. It is not a reference guide. Reference documentation is available as Web pages in the TANGO home page

Getting Started

Refer to the chapter Getting Started for an example on getting start with the C++ or Java api.

Basic Philosophy

The basic philosophy is to have high level classes to deal with Tango devices. To communicate with Tango device, uses the **DeviceProxy** class. To send/receive data to/from Tango device, uses the **DeviceData, DeviceAttribute** or **DevicePipe** classes. To communicate with a group of devices, use the **Group** class. If you are interested only in some attributes provided by a Tango device, uses the **AttributeProxy** class. Even if the Tango database is implemented as any other devices (and therefore accessible with one instance of a DeviceProxy class), specific high level classes have been developped to query it. Uses the **Database**, **DbDevice**, **DbClass**, **DbServer** or **DbData** classes when interfacing the Tango database. Callback for asynchronous requests or events are implemented via a **CallBack** class. An utility class called **ApiUtil** is also available.

Data types

The definition of the basic data type you can transfert using Tango is:

Tango type name	C++ equivalent type
DevBoolean	`boolean`
DevShort	`short`
DevEnum	enumeration (only for attribute / See chapter on advanced features)
DevLong	`int` (always 32 bits data)
DevLong64	`long long` on 32 bits chip or `long` on 64 bits chip (always 64 bits data)
DevFloat	`float`
DevDouble	`double`
DevString	`char \*`
DevEncoded	`structure` with 2 fields: a string and an array of unsigned char
DevUChar	`unsigned char`
DevUShort	`unsigned short`
DevULong	`unsigned int` (always 32 bits data)
DevULong64	`unsigned long long` on 32 bits chip or `unsigned long` on 64 bits chip (always 64 bits data)
DevState	Tango specific data type

Using commands, you are able to transfert all these data types, array of these basic types and two other Tango specific data types called DevVarLongStringArray and DevVarDoubleStringArray. See chapter [Data exchange] to get details about them. You are also able to create attributes using any of these basic data types to transfer data between clients and servers.

Request model

For the most important API remote calls (command_inout, read_attribute(s) and write_attribute(s)), Tango supports two kind of requests which are the synchronous model and the asynchronous model. Synchronous model means that the client wait (and is blocked) for the server to send an answer. Asynchronous model means that the client does not wait for the server to send an answer. The client sends the request and immediately returns allowing the CPU to do anything else (like updating a graphical user interface). Device pipe supports only the synchronous model. Within Tango, there are two ways to retrieve the server answer when using asynchronous model. They are:

1. The polling mode
2. The callback mode

In polling mode, the client executes a specific call to check if the answer is arrived. If this is not the case, an exception is thrown. If the reply is there, it is returned to the caller and if the reply was an exception, it is re-thrown. There are two calls to check if the reply is arrived:

- Call which does not wait before the server answer is returned to the caller.
- Call which wait with timeout before returning the server answer to the caller (or throw the exception) if the answer is not arrived.

In callback model, the caller must supply a callback method which will be executed when the command returns. They are two sub-modes:

1. The pull callback mode
2. The push callback mode

In the pull callback mode, the callback is triggered if the server answer is arrived when the client decide it by calling a *synchronization* method (The client pull-out the answer). In push mode, the callback is executed as soon as the reply arrives in a separate thread (The server pushes the answer to the client).

Synchronous model

Synchronous access to Tango device are provided using the *DeviceProxy* or *AttributeProxy* class. For the *DeviceProxy* class, the main synchronous call methods are :

- *command_inout()* to execute a Tango device command
- *read_attribute()* or *read_attributes()* to read a Tango device attribute(s)
- *write_attribute()* or *write_attributes()* to write a Tango device attribute(s)
- *write_read_attribute()* or *write_read_attributes()* to write then read Tango device attribute(s)
- *read_pipe()* to read a Tango device pipe
- *write_pipe()* to write a Tango device pipe
- *write_read_pipe()* to write then read Tango device pipe

For commands, data are send/received to/from device using the *DeviceData* class. For attributes, data are send/received to/from device attribute using the *DeviceAttribute* class. For pipes, data are send/receive to/from device pipe using the *DevicePipe* and *DevicePipeBlob* classes.

In some cases, only attributes provided by a Tango device are interesting for the application. You can use the *AttributeProxy* class. Its main synchronous methods are :

- *read()* to read the attribute value
- *write()* to write the attribute value
- *write_read()* to write then read the attribute value

Data are transmitted using the *DeviceAttribute* class.

Asynchronous model

Asynchronous access to Tango device are provided using *DeviceProxy* or *AttributeProxy, CallBack* and *ApiUtil* classes methods. The main asynchronous call methods and used classes are :

- To execute a command on a device
  - *DeviceProxy::command_inout_asynch()* and *DeviceProxy::command_inout_reply()* in polling model.
  - *DeviceProxy::command_inout_asynch()*, *DeviceProxy::get_asynch_replies()* and *CallBack* class in callback pull model
  - *DeviceProxy::command_inout_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model
- To read a device attribute
  - *DeviceProxy::read_attribute_asynch()* and *DeviceProxy::read_attribute_reply()* in polling model
  - *DeviceProxy::read_attribute_asynch()*, *DeviceProxy::get_asynch_replies()* and *CallBack* class in callback pull model.
  - *DeviceProxy::read_attribute_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model

- To write a device attribute
  - *DeviceProxy::write_attribute_asynch()* in polling model
  - *DeviceProxy::write_attribute_asynch()* and *CallBack* class in callback pull model
  - *DeviceProxy::write_attribute_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model

For commands, data are send/received to/from device using the *DeviceData* class. For attributes, data are send/received to/from device attribute using the *DeviceAttribute* class. It is also possible to generate asynchronous request(s) using the *AttributeProxy* class following the same schema than above. Methods to use are :

- *read_asynch()* and *read_reply()* to asynchronously read the attribute value
- *write_asynch()* and *write_reply()* to asynchronously write the attribute value

Events

Introduction

Events are a critical part of any distributed control system. Their aim is to provide a communication mechanism which is fast and efficient.

The standard CORBA communication paradigm is a synchronous or asynchronous two-way call. In this paradigm the call is initiated by the client who contacts the server. The server handles the client's request and sends the answer to the client or throws an exception which the client catches. This paradigm involves two calls to receive a single answer and requires the client to be active in initiating the request. If the client has a permanent interest in a value he is obliged to poll the server for an update in a value every time. This is not efficient in terms of network bandwidth nor in terms of client programming.

For clients who are permanently interested in values the event-driven communication paradigm is a more efficient and natural way of programming. In this paradigm the client registers her interest once in an event (value). After that the server informs the client every time the event has occurred. This paradigm avoids the client polling, frees it for doing other things, is fast and makes efficient use of the network.

The rest of this chapter explains how the TANGO events are implemented and the application programmer's interface.

Event definition

TANGO events represent an alternative channel for reading TANGO device attributes. Device attributes values are sent to all subscribed clients when an event occurs. Events can be an attribute value change, a change in the data quality or a periodically send event. The clients continue receiving events as long as they stay subscribed. Most of the time, the device server polling thread detects the event and then pushes the device attribute value to all clients. Nevertheless, in some cases, the delay introduced by the polling thread in the event propagation is detrimental. For such cases, some API calls directly push the event. Until TANGO release 8, the omniNotify implementation of the CORBA Notification service was used to dispatch events. Starting with TANGO 8, this CORBA Notification service has been replaced by the ZMQ library which implements a Publish/Subscribe communication model well adapted to TANGO events communication.

Event types

The following eight event types have been implemented in TANGO :

1. **change** - an event is triggered and the attribute value is sent when the attribute value changes significantly. The exact meaning of significant is device attribute dependent. For analog and digital values this is a delta fixed per attribute, for string values this is any non-zero change i.e. if the new attribute value is not equal to the previous attribute value. The delta can either be specified as a relative or absolute change. The delta is the same for all clients unless a filter is specified (see below). To easily write applications using the change event, it is also triggered in the following case :
    1. When a spectrum or image attribute size changes.
    2. At event subscription time
    3. When the polling thread receives an exception during attribute reading
    4. When the polling thread detects that the attribute quality factor has changed.
    5. The first good reading of the attribute after the polling thread has received exception when trying to read the attribute
    6. The first time the polling thread detects that the attribute quality factor has changed from INVALID to something else
    7. When a change event is pushed manually from the device server code. (*DeviceImpl::push_change_event()*).
    8. By the methods Attribute::set_quality() and Attribute::set_value_date_quality() if a client has subscribed to the change event on the attribute. This has been implemented for cases where the delay introduced by the polling thread in the event propagation is not authorized.
2. **periodic** - an event is sent at a fixed periodic interval. The frequency of this event is determined by the *event_period* property of the attribute and the polling frequency. The polling frequency determines the highest frequency at which the attribute is read. The event_period determines the highest frequency at which the periodic event is sent. Note if the event_period is not an integral number of the polling period there will be a beating of the two frequencies[1]. Clients can reduce the frequency at which they receive periodic events by specifying a filter on the periodic event counter.
3. **archive** - an event is sent if one of the archiving conditions is satisfied. Archiving conditions are defined via properties in the database. These can be a mixture of delta_change and periodic. Archive events can be send from the polling thread or can be manually pushed from the device server code (*DeviceImpl::push_archive_event()*).
4. **attribute configuration** - an event is sent if the attribute configuration is changed.
5. **data ready** - This event is sent when coded by the device server programmer who uses a specific method of one of the Tango device server class to fire the event (*DeviceImpl::push_data_ready_event()*). The rule of this event is to inform a client that it is now possible to read an attribute. This could be useful in case of attribute with many data.
6. **user** - The criteria and configuration of these user events are managed by the device server programmer who uses a specific method of one of the Tango device server class to fire the event (*DeviceImpl::push_event()*).
7. **device interface change** - This event is sent when the device interface changes. Using Tango, it is possible to dynamically add/remove attribute/command to a device. This event is the way to inform client(s) that attribute/command has been added/removed from a device. Note that this type of event is attached to a device and not to one attribute (like all other event types). This event is triggered in the following case :
    1. A dynamic attribute or command is added or removed. The event is sent after a small delay (50 mS) in order to eliminate the risk of events storm in case several attributes/commands are added/removed in a loop
    2. At the end of admin device RestartServer or DevRestart command

[1] note: the polling is not synchronized is currently not synchronized on the hour

3. After a re-connection due to a device server restart. Because the device interface is not memorized, the event is sent even if it is highly possible that the device interface has not changed. A flag in the data propagated with the event inform listening applications that the device interface change is not guaranteed.
4. At event re-connection time. This case is similar to the previous one (device interface change not guaranteed)

8. **pipe** - This is the kind of event which has to be used when the user want to push data through a pipe. This kind of event is only sent by the user code by using a specific method (*DeviceImpl::push_pipe_event()*). There is no way to ask the Tango kernel to automatically push this kind of event.

The first three above events are automatically generated by the TANGO library or fired by the user code. Events number 4 and 7 are only automatically sent by the library and events 5, 6 and 8 are fired only by the user code.

Event filtering (Removed in Tango release 8 and above)

Please, note that this feature is available only for Tango releases older than Tango 8. The CORBA Notification Service allows event filtering. This means that a client can ask the Notification Service to send the event only if some filter is evaluated to true. Within the Tango control system, some pre-defined fields can be used as filter. These fields depend on the event type.

Event type	Filterable field name	Filterable field value	type
change	delta_change_rel	Relative change (in %) since last even	double
	delta_change_abs	Absolute change since last event	double
	quality	Is set to 1 when the attribute quality factor has changed, otherwise it is 0	double
	forced_event	Is set to 1 when the event was fired on exception or a quality factor set to invalid	double
periodic	counter	Incremented each time the event is sent	long
archive	delta_change_rel	Relative change (in %) since last event	double
	delta_change_abs	Absolute change since last event	double
	quality	**Is set to 1 when the attribute quality** factor has changed, otherwise it is 0	double
	counter	Incremented each time the event is sent for periodic reason. Set to -1 if event sent for change reason	long
	forced_event	Is set to 1 when the event was fired on exception or a quality factor set to invalid	double
	delta_event	Number of milli-seconds since previous event	double

Filter are defined as a string following a grammar defined by CORBA. It is defined in [NotificationService]. The following example shows you the most common use of these filters in the Tango world :

- To receive periodic event one out of every three, the filter must be

  $counter % 3 == 0

- To receive change event only if the relative change is greater than % (positive and negative), the filter must be

  $delta_change_rel >= 20 or $delta_change_rel <= -20

- To receive a change event only on quality change, the filter must be

  $quality == 1

For user events, the filter field name(s) and their value are defined by the device server programmer.

Application Programmer's Interface

How to setup and use the TANGO events ? The interfaces described here are intended as user friendly interfaces to the underlying CORBA calls. The interface is modeled after the asynchronous *command_inout()* interface so as to maintain coherency. The event system supports **push callback model** as well as the **pull callback model.**

The two event reception modes are:

- **Push callback model** : On event reception a callbacks method gets immediately executed.
- **Pull callback model** : The event will be buffered the client until the client is ready to receive the event data. The client triggers the execution of the callback method.

The event reception buffer in the **pull callback model**, is implemented as a round robin buffer. The client can choose the size when subscribing for the event. This way the client can set-up different ways to receive events.

- Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.
- Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.
- Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

Configuring events

The attribute configuration set is used to configure under what conditions events are generated. A set of standard attribute properties (part of the standard attribute configuration) are read from the database at device startup time and used to configure the event engine. If there are no properties defined then default values specified in the code are used.

change

The attribute properties and their default values for the change event are :

1. **rel_change** - a property of maximum 2 values. It specifies the positive and negative relative change of the attribute value w.r.t. the value of the previous change event which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no property is specified, no events are generated.

2. **abs_change** - a property of maximum 2 values.It specifies the positive and negative absolute change of the attribute value w.r.t the value of the previous change event which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.

periodic

The attribute properties and their default values for the periodic event are :

1. **event_period** - the minimum time between events (in milliseconds). If no property is specified then a default value of 1 second is used.

archive

The attribute properties and their default values for the archive event are :

1. **archive_rel_change** - a property of maximum 2 values which specifies the positive and negative relative change w.r.t. the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then no events are generate.
2. **archive_abs_change** - a property of maximum 2 values which specifies the positive and negative absolute change w.r.t the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.
3. **archive_period** - the minimum time between archive events (in milliseconds). If no property is specified, no periodic archiving events are send.

C++ Clients

This is the interface for clients who want to receive events. The main action of the client is to subscribe and unsubscribe to events. Once the client has subscribed to one or more events the events are received in a separate thread by the client.

Two reception modes are possible:

- On event reception a callbacks method gets immediately executed.
- The event will be buffered until the client until the client is ready to receive the event data.

The mode to be used has to be chosen when subscribing for the event.

Subscribing to events

The client call to subscribe to an event is named *DeviceProxy::subscribe_event()* . During the event subscription the client has to choose the event reception mode to use.

Push model:

```
int DeviceProxy::subscribe_event(
            const string &attribute,
            Tango::EventType event,
            Tango::CallBack *callback,
            bool stateless = false);
```

The client implements a callback method which is triggered when the event is received. Note that this callback method will be executed by a thread started by the underlying ORB. This thread is not the application main thread. For Tango releases before 8, a similar call with one extra parameter for event filtering is also available.

Pull model:

```
int DeviceProxy::subscribe_event(
            const string &attribute,
            Tango::EventType event,
            int event_queue_size,
            bool stateless = false);
```

The client chooses the size of the round robin event reception buffer. Arriving events will be buffered until the client uses *DeviceProxy::get_events()* to extract the event data. For Tango releases before 8, a similar call with one extra parameter for event filtering is also available.

On top of the user filter defined by the *filters* parameter, basic filtering is done based on the reason specified and the event type. For example when reading the state and the reason specified is change the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

The stateless flag = false indicates that the event subscription will only succeed when the given attribute is known and available in the Tango system. Setting stateless = true will make the subscription succeed, even if an attribute of this name was never known. The real event subscription will happen when the given attribute will be available in the Tango system.

Note that in this model, the callback method will be executed by the thread doing the *DeviceProxy::get_events()* call.

The CallBack class

In C++, the client has to implement a class inheriting from the Tango CallBack class and pass this to the *DeviceProxy::subscribe_event()* method. The CallBack class is the same class as the one proposed for the TANGO asynchronous call. This is as follows for events :

```
class MyCallback : public Tango::CallBack
{
   .
   .
   .
  public:
  void push_event(Tango::EventData *);
  void push_event(Tango::AttrConfEventData *);
```

(continues on next page)

(continued from previous page)

```
    void push_event(Tango::DataReadyEventData *);
    void push_event(Tango::DevIntrChangeEventData *);
    void push_event(Tango::PipeEventData *);
}
```

where EventData is defined as follows :

```
class EventData
{
    DeviceProxy       *device;
    string            attr_name;
    string            event;
    DeviceAttribute   *attr_value;
    bool              err;
    DevErrorList      errors;
}
```

AttrConfEventData is defined as follows :

```
class AttrConfEventData
{
    DeviceProxy       *device;
    string            attr_name;
    string            event;
    AttributeInfoEx   *attr_conf;
    bool              err;
    DevErrorList      errors;
}
```

DataReadyEventData is defined as follows :

```
class DataReadyEventData
{
    DeviceProxy       *device;
    string            attr_name;
    string            event;
    int               attr_data_type;
    int               ctr;
    bool              err;
    DevErrorList      errors;
}
```

DevIntrChangeEventData is defined as follows :

```
class DevIntrChangeEventData
{
    DeviceProxy           device;
    string                event;
    string                device_name;
    CommandInfoList       cmd_list;
    AttributeInfoListEx   att_list;
    bool                  dev_started;
    bool                  err;
```

(continues on next page)

(continued from previous page)

```
   DevErrorList           errors;
}
```

and PipeEventData is defined as follows :

```
class PipeEventData
{
   DeviceProxy       *device;
   string            pipe_name;
   string            event;
   DevicePipe        *pipe_value;
   bool              err;
   DevErrorList      errors;
}
```

In push model, there are some cases (same callback used for events coming from different devices hosted in device server process running on different hosts) where the callback method could be executed concurently by different threads started by the ORB. The user has to code his callback method in a **thread safe** manner.

Unsubscribing from an event

Unsubscribe a client from receiving the event specified by *event_id* is done by calling the *DeviceProxy::unsubscribe_event()* method :

```
void DeviceProxy::unsubscribe_event(int event_id);
```

Extract buffered event data

When the pull model was chosen during the event subscription, the received event data can be extracted with *DeviceProxy::get_events()*. Two possibilities are available for data extraction. Either a callback method can be executed for every event in the buffer when using

```
int DeviceProxy::get_events(
             int event_id,
             CallBack *cb);
```

Or all the event data can be directly extracted as EventDataList, AttrConfEventDataList , DataReadyEventDataList, DevIntrChangeEventDataList or PipeEventDataList when using

```
int DeviceProxy::get_events(
             int event_id,
             EventDataList &event_list);

int DeviceProxy::get_events(
             int event_id,
             AttrConfEventDataList &event_list);

int DeviceProxy::get_events(
             int event_id,
             DataReadyEventDataList &event_list);
```

(continues on next page)

(continued from previous page)

```
int DeviceProxy::get_events(
            int event_id,
            DevIntrChangeEventDataList &event_list);

int DeviceProxy::get_events(
            int event_id,
            PipeEventDataList &event_list);
```

The event data lists are vectors of EventData, AttrConfEventData, DataReadyEventData or PipeEventData pointers with special destructor and clean-up methods to ease the memory handling.

```
class EventDataList:public vector<EventData *>
class AttrConfEventDataList:public vector<AttrConfEventData *>
class DataReadyEventDataList:public vector<DataReadyEventData *>
class DevIntrChangeEventDataList:public vector<DevIntrChangeEventData *>
class PipeEventDataList:public vector<PipeEventData *>
```

Example

Here is a typical code example of a client to register and receive events. First, you have to define a callback method as follows:

```
class DoubleEventCallBack : public Tango::CallBack
{
   void push_event(Tango::EventData*);
};


void DoubleEventCallBack::push_event(Tango::EventData *myevent)
{
    Tango::DevVarDoubleArray *double_value;
    try
    {
        cout << "DoubleEventCallBack::push_event(): called attribute "
             << myevent->attr_name
             << " event "
             << myevent->event
             << " (err="
             << myevent->err
             << ")" << endl;


         if (!myevent->err)
         {
             *(myevent->attr_value) >> double_value;
             cout << "double value "
                  << (*double_value)[0]
                  << endl;
             delete double_value;
         }
```

(continues on next page)

(continued from previous page)

```
    }
    catch (...)
    {
        cout << "DoubleEventCallBack::push_event(): could not extract data !\n";
    }
}
```

Note: If the event is an error event (myevent->err == true), the attr_value field in myevent EventData object will be empty (null pointer). The same applies for some other kinds of events. For example, attr_conf field in AttrConfEventData object will be a null pointer in case of error event (if the err field is true). As a consequence, the device server programmer should always check the err field before trying to extract the EventData::attr_value or AttrConfEventData::attr_conf fields associated to the event.

Then the main code must subscribe to the event and choose the push or the pull model for event reception.

Push model:

```
DoubleEventCallBack *double_callback = new DoubleEventCallBack;

Tango::DeviceProxy *mydevice = new Tango::DeviceProxy("my/device/1");

int event_id;
const string attr_name("current");
event_id = mydevice->subscribe_event(attr_name,
                        Tango::CHANGE_EVENT,
                        double_callback);
cout << "event_client() id = " << event_id << endl;

// The callback methods are executed by the Tango event reception thread.
// The main thread is not concerned of event reception.
// Whatch out with synchronisation and data access in a multi threaded environment!

sleep(1000); // wait for events

mydevice->unsubscribe_event(event_id);
```

Pull model:

```
DoubleEventCallBack *double_callback = new DoubleEventCallBack;
int event_queue_size = 100; // keep the last 100 events

Tango::DeviceProxy *mydevice = new Tango::DeviceProxy("my/device/1");

int event_id;
const string attr_name("current");
event_id = mydevice->subscribe_event(attr_name,
                        Tango::CHANGE_EVENT,
                        event_queue_size);
cout << "event_client() id = " << event_id << endl;

// Check every 3 seconds whether new events have arrived and trigger the callback method
```

(continues on next page)

(continued from previous page)

```
// for the new events.

for (int i=0; i < 100; i++)
{
    sleep (3);

    // Read the stored event data from the queue and call the callback method for every event.
    mydevice->get_events(event_id, double_callback);
}

event_test->unsubscribe_event(event_id);
```

Group

A Tango Group provides the user with a single point of control for a collection of devices. By analogy, one could see a Tango Group as a proxy for a collection of devices. For instance, the Tango Group API supplies a *command_inout()* method to execute the same command on all the elements of a group.

A Tango Group is also a hierarchical object. In other words, it is possible to build a group of both groups and individual devices. This feature allows creating logical views of the control system - each view representing a hierarchical family of devices or a sub-system.

In this chapter, we will use the term *hierarchy* to refer to a group and its sub-groups. The term *Group* designates to the local set of devices attached to a specific Group.

Getting started with Tango group

The quickest way of getting started is to study an example…

Imagine we are vacuum engineers who need to monitor and control hundreds of gauges distributed over the 16 cells of a large-scale instrument. Each cell contains several penning and pirani gauges. It also contains one strange gauge. Our main requirement is to be able to control the whole set of gauges, a family of gauges located into a particular cell (e.g. all the penning gauges of the 6th cell) or a single gauge (e.g. the strange gauge of the 7th cell). Using a Tango Group, such features are quite straightforward to obtain.

Reading the description of the problem, the device hierarchy becomes obvious. Our gauges group will have the following structure:

```
-> gauges
  |  -> cell-01
  |     |-> inst-c01/vac-gauge/strange
  |     |-> penning
  |     |    |-> inst-c01/vac-gauge/penning-01
  |     |    |-> inst-c01/vac-gauge/penning-02
  |     |    |- ...
  |     |    |-> inst-c01/vac-gauge/penning-xx
  |     |-> pirani
  |          |-> inst-c01/vac-gauge/pirani-01
  |          |-> ...
  |          |-> inst-c01/vac-gauge/pirani-xx
  |  -> cell-02
```

(continues on next page)

(continued from previous page)

```
|      |-> inst-c02/vac-gauge/strange
|      |-> penning
|      |   |-> inst-c02/vac-gauge/penning-01
|      |   |-> ...
|      |
|      |-> pirani
|      |   |-> ...
|   -> cell-03
|      |-> ...
|          | -> ...
```

In the C++, such a hierarchy can be build as follows (basic version):

```
//- step0: create the root group
Tango::Group *gauges = new Tango::Group("gauges");


//- step1: create a group for the n-th cell
Tango::Group *cell = new Tango::Group("cell-01");


//- step2: make the cell a sub-group of the root group
gauges->add(cell);


//- step3: create a "penning" group
Tango::Group *gauge_family = new Tango::Group("penning");


//- step4: add all penning gauges located into the cell (note the wildcard)
gauge_family->add("inst-c01/vac-gauge/penning*");


//- step5: add the penning gauges to the cell
cell->add(gauge_family);


//- step6: create a "pirani" group
gauge_family = new Tango::Group("pirani");


//- step7: add all pirani gauges located into the cell (note the wildcard)
gauge_family->add("inst-c01/vac-gauge/pirani*");


//- step8: add the pirani gauges to the cell
cell->add(gauge_family);


//- step9: add the "strange" gauge to the cell
cell->add("inst-c01/vac-gauge/strange");

```

(continues on next page)

(continued from previous page)

```
40
41 //- repeat step 1 to 9 for the remaining cells
42 cell = new Tango::Group("cell-02");
43 ...
```

Important note: There is no particular order to create the hierarchy. However, the insertion order of the devices is conserved throughout the lifecycle of the Group and cannot be changed. That way, the Group implementation can guarantee the order in which results are returned (see below).

Keeping a reference to the root group is enough to manage the whole hierarchy (i.e. there no need to keep trace of the sub-groups or individual devices). The Group interface provides methods to retrieve a sub-group or an individual device.

Be aware that a C++ group allways gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a Group (respectively a DeviceProxy) returned by a call to *Tango::Group::get_group()* (respectively to *Tango::Group::get_device()*). Use the *Tango::Group::remove()* method instead (see the Tango Group class API documentation for details).

We can now perform any action on any element of our gauges group. For instance, let's ping the whole hierarchy to be sure that all devices are alive.

```
1 //- ping the whole hierarchy
2 if (gauges->ping() == true)
3 {
4     std::cout << "all devices alive" << std::endl;
5 }
6 else
7 {
8     std::cout << "at least one dead/busy/locked/... device" << std::endl;
9 }
```

Enabling and disabling group members

Devices belonging to a group can be temporarily excluded from all operations performed on the group using the *Group::disable* and *Group::enable* calls.

Device name passed to the *disable* (*enable*) methods can contain wildcards (*). Note that only the first matching device will be disabled (enabled). The search algorithm is breadth-first search. All group elements are searched (in the insertion order) for a match before descending recursively to sub-groups. Recursive search can be disabled with the *forward* flag (see a *section dedicated to the forwarding*).

During group operations like attribute read or command calls, entries for disabled elements will be included in the result set, however they will not have any value and will be marked as disabled (*GroupReply::group_element_enabled* will be false).

Note that if *exceptions are enabled*, any attempt to access the result (e.g. via *get_data()*) from a disabled device will raise `Tango::DevFailed`. Otherwise an empty value will be returned.

Below is an example using the gauges group:

```
1 // will disable: inst-c01/vac-gauge/penning-01
2 gauges->disable("inst-c01/*/penn");
3
4 // will disable nothing
```

(continues on next page)

(continued from previous page)

```
const bool forwarded = true;
gauges->disable("inst-c01/vac-gauge/pirani-01", not forwarded);

// will disable: inst-c01/vac-gauge/pirani-01
gauge_family->disable("inst-c01/vac-gauge/pirani-01");

// will enable: inst-c01/vac-gauge/penning-01
gauges->enable("inst-c01/*");

auto states = gauges->command_inout("State");
for (auto& state : states)
{
    if (state.group_element_enabled())
    {
        // it's safe to access the value
        std::cout << state.dev_name() << ": " << state.get_data() << "\n";
    }
    else
    {
        std::cout << state.dev_name() << ": is disabled\n";
    }
}
```

Forward or not forward?

Since a Tango Group is a hierarchical object, any action performed on a group can be forwarded to its sub-groups. Most of the methods in the Group interface have a so-called *forward* option controlling this propagation. When set to *false*, the action is only performed on the local set of devices. Otherwise, the action is also forwarded to the sub-groups, in other words, propagated along the hierarchy. In C++ , the forward option defaults to true (thanks to the C++ default argument value). There is no such mechanism in Java and the forward option must be systematically specified.

Executing a command

As a proxy for a collection of devices, the Tango Group provides an interface similar to the DeviceProxy's. For the execution of a command, the Group interface contains several implementations of the *command_inout* method. Both synchronous and asynchronous forms are supported.

Obtaining command results

Command results are returned using a Tango::GroupCmdReplyList. This is nothing but a vector containing a Tango::GroupCmdReply for each device in the group. The Tango::GroupCmdReply contains the actual data (i.e. the Tango::DeviceData). By inheritance, it may also contain any error occurred during the execution of the command (in which case the data is invalid).

We previously indicated that the Tango Group implementation guarantees that the command results are returned in the order in which its elements were attached to the group. For instance, if g1 is a group containing three devices attached in the following order:

```
g1->add("my/device/01");
g1->add("my/device/03");
g1->add("my/device/02");
```

the results of

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status");
```

will be organized as follows:

crl[0] contains the status of my/device/01
crl[1] contains the status of my/device/03
crl[2] contains the status of my/device/02

Things get more complicated if sub-groups are added between devices.

```
g2->add("my/device/04");
g2->add("my/device/05");


g4->add("my/device/08");
g4->add("my/device/09");


g3->add("my/device/06");
g3->add(g4);
g3->add("my/device/07");


g1->add("my/device/01");
g1->add(g2);
g1->add("my/device/03");
g1->add(g3);
g1->add("my/device/02");
```

The result order in the Tango::GroupCmdReplyList depends on the value of the forward option. If set to *true*, the results will be organized as follows:

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status", true);
```

crl[0] contains the status of my/device/01 which belongs to g1
crl[1] contains the status of my/device/04 which belongs to g1.g2
crl[2] contains the status of my/device/05 which belongs to g1.g2
crl[3] contains the status of my/device/03 which belongs to g1
crl[4] contains the status of my/device/06 which belongs to g1.g3
crl[5] contains the status of my/device/08 which belongs to g1.g3.g4
crl[6] contains the status of my/device/09 which belongs to g1.g3.g
crl[7] contains the status of my/device/07 which belongs to g1.g3
crl[8] contains the status of my/device/02 which belongs to g1

If the forward option is set to *false*, the results are:

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status", false);
```

crl[0] contains the status of my/device/01 which belongs to g
crl[1] contains the status of my/device/03 which belongs to g1
crl[2] contains the status of my/device/02 which belongs to g1

The Tango::GroupCmdReply contains some public members allowing the identification of both the device (Tango::GroupCmdReply::dev_name) and the command (Tango::GroupCmdReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using the Tango::GroupCmdReply::dev_name member.

Case 1: a command, no argument

As an example, we execute the Status command on the whole hierarchy synchronously.

```
Tango::GroupCmdReplyList crl = gauges->command_inout("Status");
```

As a first step in the results processing, it could be interesting to check value returned by the *has_failed()* method of the GroupCmdReplyList. If it is set to true, it means that at least one error occurred during the execution of the command (i.e. at least one device gave error).

```
if (crl.has_failed())
{
    cout << "at least one error occurred" << endl;
}
else
{
    cout << "no error " << endl;
}
```

Now, we have to process each individual response in the list.

A few words on error handling and data extraction

Depending of the application and/or the developer's programming habits, each individual error can be handle by the C++ (or Java) exception mechanism or using the dedicated *has_failed()* method. The GroupReply class - which is the mother class of both GroupCmdReply and GroupAttrReply - contains a static method to enable (or disable) exceptions called *enable_exception()*. By default, exceptions are disabled. The following example is proposed with both exceptions enable and disable.

In C++, data can be extracted directly from an individual reply. The GroupCmdReply interface contains a template operator >> allowing the extraction of any supported Tango type (in fact the actual data extraction is delegated to DeviceData::operator >>). One dedicated extract method is also provided in order to extract DevVarLongStringArray and DevVarDoubleStringArray types to std::vectors.

Error and data handling C++ example:

```
//-------------------------------------------------------
//- synch. group command example with exception enabled
//-------------------------------------------------------
//- enable exceptions and save current mode
bool last_mode = GroupReply::enable_exception(true);
//- process each response in the list ...
for (int r = 0; r < crl.size(); r++)
{
//- enter a try/catch block
   try
   {
//- try to extract the data from the r-th reply
//- suppose data contains a double
       double ans;
       crl[r] >> ans;
       cout << crl[r].dev_name()
            << "::"
            << crl[r].obj_name()
            << " returned "
            << ans
            << endl;
    }
    catch (const DevFailed& df)
    {
//- DevFailed caught while trying to extract the data from reply
      for (int err = 0; err < df.errors.length(); err++)
      {
           cout << "error: " << df.errors[err].desc.in() << endl;
      }
//- alternatively, one can use crl[r].get_err_stack() see below
    }
    catch (...)
    {
       cout << "unknown exception caught";
    }
}
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- Clear the response list (if reused later in the code)
crl.reset();


//-------------------------------------------------------
//- synch. group command example with exception disabled
//-------------------------------------------------------
//- disable exceptions and save current mode bool
last_mode = GroupReply::enable_exception(false);
//- process each response in the list ...
for (int r = 0; r < crl.size(); r++)
{
//- did the r-th device give error?
    if (crl[r].has_failed() == true)
    {
```

(continues on next page)

(continued from previous page)

```
//- printout error description
       cout << "an error occurred while executing "
            << crl[r].obj_name()
            << " on "
            << crl[r].dev_name() << endl;
//- dump error stack
       const DevErrorList& el = crl[r].get_err_stack();
       for (int err = 0; err < el.size(); err++)
       {
           cout << el[err].desc.in();
       }
    }
    else
    {
//- no error (suppose data contains a double)
       double ans;
       bool result = crl[r] >> ans;
       if (result == false)
       {
           cout << "could not extract double from "
                << crl[r].dev_name()
                << " reply"
                << endl;
       }
       else
       {
           cout << crl[r].dev_name()
                << "::"
                << crl[r].obj_name()
                << " returned "
                << ans
                << endl;
       }
    }
}
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- Clear the response list (if reused later in the code)
crl.reset();
```

Now execute the same command asynchronously. C++ example:

```
//-----------------------------------------------------
//- asynch. group command example (C++ example)
//-----------------------------------------------------
long request_id = gauges->command_inout_asynch("Status");
//- do some work
do_some_work();


//- get results
crl = gauges->command_inout_reply(request_id);
```

(continues on next page)

(continued from previous page)

```
//- process responses as previously describe in the synch. implementation
for (int r = 0; r < crl.size(); r++)
{
//- data processing and error handling goes here
//- copy/paste code from previous example
. . .
}
//- clear the response list (if reused later in the code)
crl.reset();
```

Case 2: a command, one argument

Here, we give an example in which the same input argument is applied to all devices in the group (or its sub-groups).

In C++:

```
//- the argument value
double d = 0.1;
//- insert it into the TANGO generic container for command: DeviceData
Tango::DeviceData dd;
dd << d;
//- execute the command: Dev_Void SetDummyFactor (Dev_Double)
Tango::GroupCmdReplyList crl = gauges->command_inout("SetDummyFactor", dd);
```

Since the SetDummyFactor command does not return any value, the individual replies (i.e. the GroupCmdReply) do not contain any data. However, we have to check their *has_failed()* method returned value to be sure that the command completed successfully on each device (acknowledgement). Note that in such a case, exceptions are useless since we never try to extract data from the replies.

In C++ we should have something like:

```
//- no need to process the results if no error occurred (Dev_Void command)
if (crl.has_failed())
{
//- at least one error occurred
    for (int r = 0; r < crl.size(); r++)
    {
//- handle errors here (see previous C++ examples)
    }
}
//- clear the response list (if reused later in the code)
crl.reset();
```

See case 1 for an example of asynchronous command.

Case 3: a command, several arguments

Here, we give an example in which a **specific** input argument is applied to each device in the hierarchy. In order to use this form of command_inout, the user must have an a priori and perfect knowledge of the devices order in the hierarchy. In such a case, command arguments are passed in an array (with one entry for each device in the hierarchy).

The C++ implementation provides a template method which accepts a std::vector of C++ type for command argument. This allows passing any kind of data using a single method.

The size of this vector must equal the number of device in the hierarchy (respectively the number of device in the group) if the forward option is set to true (respectively set to false). Otherwise, an exception is thrown.

The first item in the vector is applied to the first device in the hierarchy, the second to the second device in the hierarchy, and so on...That's why the user must have a perfect knowledge of the devices order in the hierarchy.

Assuming that gauges are ordered by name, the SetDummyFactor command can be executed on group cell-01 (and its sub-groups) as follows:

Remember, cell-01 has the following internal structure:

```
-> gauges
   | -> cell-01
   |    |-> inst-c01/vac-gauge/strange
   |    |-> penning
   |    |   |-> inst-c01/vac-gauge/penning-01
   |    |   |-> inst-c01/vac-gauge/penning-02
   |    |   |-> ...
   |    |   |-> inst-c01/vac-gauge/penning-xx
   |    |-> pirani
   |        |-> inst-c01/vac-gauge/pirani-01
   |        |-> ...
   |        |-> inst-c01/vac-gauge/pirani-xx
```

Passing a specific argument to each device in C++:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the hierarchy (starting at cell-01)
long n_dev = g->get_size(true);
//- Build argin list
std::vector<double> argins(n_dev);
//- argument for inst-c01/vac-gauge/strange
argins[0] = 0.0;
//- argument for inst-c01/vac-gauge/penning-01
argins[1] = 0.1;
//- argument for inst-c01/vac-gauge/penning-02
argins[2] = 0.2;
//- argument for remaining devices in cell-01.penning
. . .
//- argument for devices in cell-01.pirani
. . .
//- the reply list
Tango::GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
```

(continues on next page)

(continued from previous page)

```
//- execute the command
    crl = g->command_inout("SetDummyFactor", argins, true);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (const DevFailed& df)
{
//- see below
}
crl.reset();
```

If we want to execute the command locally on cell-01 (i.e. not on its sub-groups), we should write the following C++ code:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the group (starting at cell-01)
long n_dev = g->get_size(false);
//- Build argin list
std::vector<double> argins(n_dev);
//- argins for inst-c01/vac-gauge/penning-01
argins[0] = 0.1;
//- argins for inst-c01/vac-gauge/penning-02
argins[1] = 0.2;
//- argins for remaining devices in cell-01.penning
. . .
//- the reply list
Tango::GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
//- execute the command
    crl = g->command_inout("SetDummyFactor", argins, false);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (const DevFailed& df)
{
//- see below
}
crl.reset();
```

Note: if we want to execute the command locally on cell-01 (i.e. not on its sub-groups), we should write the following code:

```
//- get a reference to the target group
Group g = gauges.get_group("cell-01");
//- get pre-build arguments list for the group (starting@cell-01)
```

(continues on next page)

(continued from previous page)

```
DeviceData[] argins = g.get_command_specific_argument_list(false);
//- argins for inst-c01/vac-gauge/penning-01
argins[0].insert(0.1);
//- argins for inst-c01/vac-gauge/penning-02
argins[1].insert(0.2);
//- argins for remaining devices in cell-01.penning
. . .
//- the reply list
GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
//- execute the command
    crl = g.command_inout("SetDummyFactor", argins, false, false);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (DevFailed d)
{
//- see below
}
```

This form of *command_inout* (the one that accepts an array of value as its input argument), may throw an exception **before** executing the command if the number of elements in the input array does not match the number of individual devices in the group or in the hierarchy (depending on the forward option).

An asynchronous version of this method is also available. See case 1 for an example of asynchronous command.

Reading attribute(s)

In order to read attribute(s), the Group interface contains several implementations of the *read_attribute()* and *read_attributes()* methods. Both synchronous and asynchronous forms are supported. Reading several attributes is very similar to reading a single attribute. Simply replace the std::string used for attribute name by a vector of std::string with one element for each attribute name. In case of read_attributes() call, the order of attribute value returned in the GroupAttrReplyList is all attributes for first element in the group followed by all attributes for the second group element and so on.

Obtaining attribute values

Attribute values are returned using a GroupAttrReplyList. This is nothing but an array containing a GroupAttrReply for each device in the group. The GroupAttrReply contains the actual data (i.e. the DeviceAttribute). By inheritance, it may also contain any error occurred during the execution of the command (in which case the data is invalid).

Here again, the Tango Group implementation guarantees that the attribute values are returned in the order in which its elements were attached to the group. See Obtaining command results for details.

The GroupAttrReply contains some public methods allowing the identification of both the device (GroupAttrReply::dev_name) and the attribute (GroupAttrReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using the Tango::GroupAttrReply::dev_name member.

A few words on error handling and data extraction

Here again, depending of the application and/or the developer's programming habits, each individual error can be handle by the C++ exception mechanism or using the dedicated *has_failed()* method. The GroupReply class - which is the mother class of both GroupCmdReply and GroupAttrReply - contains a static method to enable (or disable) exceptions called *enable_exception()*. By default, exceptions are disabled. The following example is proposed with both exceptions enable and disable.

In C++, data can be extracted directly from an individual reply. The GroupAttrReply interface contains a template operator>> allowing the extraction of any supported Tango type (in fact the actual data extraction is delegated to DeviceAttribute::operator>>).

Reading an attribute is very similar to executing a command.

Reading an attribute in C++:

```
//-----------------------------------------------------------------
//- synch. read "vacuum" attribute on each device in the hierarchy
//- with exceptions enabled - C++ example
//-----------------------------------------------------------------
//- enable exceptions and save current mode
bool last_mode = GroupReply::enable_exception(true);
//- read attribute
Tango::GroupAttrReplyList arl = gauges->read_attribute("vacuum");
//- for each response in the list ...
for (int r = 0; r < arl.size(); r++)
{
//- enter a try/catch block
   try
   {
//- try to extract the data from the r-th reply
//- suppose data contains a double
      double ans;
      arl[r] >> ans;
      cout << arl[r].dev_name()
           << "::"
           << arl[r].obj_name()
           << " value is "
           << ans << endl;
   }
   catch (const DevFailed& df)
   {
//- DevFailed caught while trying to extract the data from reply
      for (int err = 0; err < df.errors.length(); err++)
      {
         cout << "error: " << df.errors[err].desc.in() << endl;
      }
//- alternatively, one can use arl[r].get_err_stack() see below
   }
   catch (...)
   {
      cout << "unknown exception caught";
   }
}
```

(continues on next page)

(continued from previous page)

```
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- clear the reply list (if reused later in the code)
arl.reset();
```

In C++, an asynchronous version of the previous example could be:

```
//- read the attribute asynchronously
long request_id = gauges->read_attribute_asynch("vacuum");
//- do some work
do_some_work();


//- get results
Tango::GroupAttrReplyList arl = gauges->read_attribute_reply(request_id);
//- process replies as previously described in the synch. implementation
for (int r = 0; r < arl.size(); r++)
{
//- data processing and/or error handling goes here
...
}
//- clear the reply list (if reused later in the code)
arl.reset();
```

Writing an attribute

The Group interface contains several implementations of the *write_attribute()* method. Both synchronous and asynchronous forms are supported. However, writing more than one attribute at a time is not supported.

Obtaining acknowledgement

Acknowledgements are returned using a GroupReplyList. This is nothing but an array containing a GroupReply for each device in the group. The GroupReply may contain any error occurred during the execution of the command. The return value of the *has_failed()* method indicates whether an error occurred or not. If this flag is set to true, the *GroupReply::get_err_stack()* method gives error details.

Here again, the Tango Group implementation guarantees that the attribute values are returned in the order in which its elements were attached to the group. See Obtaining command results for details.

The GroupReply contains some public members allowing the identification of both the device (GroupReply::dev_name) and the attribute (GroupReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using

Case 1: one value for all devices

Here, we give an example in which the same attribute value is written on all devices in the group (or its sub-groups). Exceptions are supposed to be disabled.

Writing an attribute in C++:

```
//-----------------------------------------------------------------
//- synch. write "dummy" attribute on each device in the hierarchy
//-----------------------------------------------------------------
//- assume each device support a "dummy" writable attribute
//- insert the value to be written into a generic container
Tango::DeviceAttribute value(std::string("dummy"), 3.14159);
//- write the attribute
Tango::GroupReplyList rl = gauges->write_attribute(value);
//- any error?
if (rl.has_failed() == false)
{
    cout << "no error" << endl;
}
else
{
    cout << "at least one error occurred" << endl;
//- for each response in the list ...
    for (int r = 0; r < rl.size(); r++)
    {
//- did the r-th device give error?
       if (rl[r].has_failed() == true)
       {
//- printout error description
           cout << "an error occurred while reading "
                << rl[r].obj_name()
                << " on "
                << rl[r].dev_name()
                << endl;
//- dump error stack
           const DevErrorList& el = rl[r].get_err_stack();
           for (int err = 0; err < el.size(); err++)
           {
              cout << el[err].desc.in();
           }
        }
     }
}
//- clear the reply list (if reused later in the code)
rl.reset();
```

Here is a C++ asynchronous version:

```
//- insert the value to be written into a generic container
Tango::DeviceAttribute value(std::string("dummy"), 3.14159);
//- write the attribute asynchronously
long request_id = gauges.write_attribute_asynch(value);
//- do some work
```

(continues on next page)

(continued from previous page)

```
do_some_work();


//- get results
Tango::GroupReplyList rl = gauges->write_attribute_reply(request_id);
//- process replies as previously describe in the synch. implementation ...
```

Case 2: a specific value per device

Here, we give an example in which a **specific** attribute value is applied to each device in the hierarchy. In order to use this form of *write_attribute()*, the user must have an a priori and perfect knowledge of the devices order in the hierarchy.

The C++ implementation provides a template method which accepts a std::vector of C++ type for command argument. This allows passing any kind of data using a single method.

The size of this vector must equal the number of device in the hierarchy (respectively the number of device in the group) if the forward option is set to true (respectively set to false). Otherwise, an exception is thrown.

The first item in the vector is applied to the first device in the group, the second to the second device in the group, and so on... That's why the user must have a perfect knowledge of the devices order in the group.

Assuming that gauges are ordered by name, the dummy attribute can be written as follows on group cell-01 (and its sub-groups) as follows:

Remember, cell-01 has the following internal structure:

```
-> gauges
    | -> cell-01
    |      |-> inst-c01/vac-gauge/strange
    |      |-> penning
    |      |     |-> inst-c01/vac-gauge/penning-01
    |      |     |-> inst-c01/vac-gauge/penning-02
    |      |     |-> ...
    |      |     |-> inst-c01/vac-gauge/penning-xx
    |      |-> pirani
    |            |-> inst-c01/vac-gauge/pirani-01
    |            |-> ...
    |            |-> inst-c01/vac-gauge/pirani-xx
```

C++ version:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the hierarchy (starting at cell-01)
long n_dev = g->get_size(true);
//- Build value list
std::vector<double> values(n_dev);
//- value for inst-c01/vac-gauge/strange
values[0] = 3.14159;
//- value for inst-c01/vac-gauge/penning-01
values[1] = 2 * 3.14159;
//- value for inst-c01/vac-gauge/penning-02
values[2] = 3 * 3.14159;
```

(continues on next page)

(continued from previous page)

```
//- value for remaining devices in cell-01.penning
. . .
//- value for devices in cell-01.pirani
. . .
//- the reply list
Tango::GroupReplyList rl;
//- enter a try/catch block (see below)
try
{
//- write the "dummy" attribute
    rl = g->write_attribute("dummy", values, true);
    if (rl.has_failed())
    {
//- error handling (see previous cases)
    }
}
catch (const DevFailed& df)
{
//- see below
}
rl.reset();
```

Note: if we want to execute the command locally on cell-01 (i.e. not on its sub-groups), we should write the following code

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the group
long n_dev = g->get_size(false);
//- Build value list
std::vector<double> values(n_dev);
//- value for inst-c01/vac-gauge/penning-01
values[0] = 2 * 3.14159;
//- value for inst-c01/vac-gauge/penning-02
values[1] = 3 * 3.14159;
//- value for remaining devices in cell-01.penning
. . .
//- the reply list
Tango::GroupReplyList rl;
//- enter a try/catch block (see below)
try
{
//- write the "dummy" attribute
   rl = g->write_attribute("dummy", values, false);
   if (rl.has_failed())
   {
//- error handling (see previous cases)
   }
}
catch (const DevFailed& df)
{
//- see below
```

(continues on next page)

(continued from previous page)

```
28 }
29 rl.reset();
```

This form of *write_attribute()* (the one that accepts an array of value as its input argument), may throw an exception before executing the command if the number of elements in the input array does not match the number of individual devices in the group or in the hierarchy (depending on the forward option).

An asynchronous version of this method is also available.

Reading/Writing device pipe

Reading or writing device pipe is made possible using DeviceProxy class methods. To read a pipe, you have to use the method **read_pipe()**. To write a pipe, use the **write_pipe()** method. A method **write_read_pipe()** is also provided in case you need to write then read a pipe in a non-interuptible way. All these calls generate synchronous request and support only reading or writing a single pipe at a time. Those pipe related DeviceProxy class methods (read_pipe, write_pipe,...) use DevicePipe class instances. A DevicePipe instance is nothing more than a string for the pipe name and a *DevicePipeBlob* instance called the root blob. In a DevicePipeBlob instance, you have:

- The blob name
- One array of *DataElement*. Each instance of this DataElement class has:
  - A name
  - A value which can be either
    - Scalar or array of any basic Tango type
    - Another DevicePipeBlob

Therefore, this is a recursive data structure and you may have DevicePipeBlob in DevicePipeBlob. There is no limit on the depth of this recursivity even if it is not recommended to have a too large depth. The following figure summarizes DevicePipe data structure

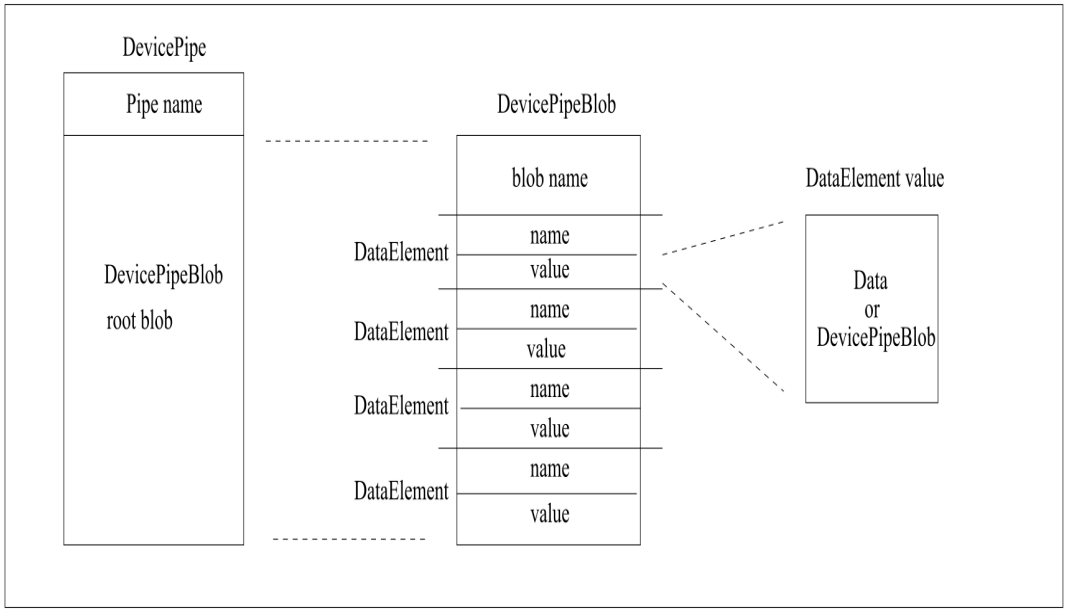


Fig. 1: Figure 4.1: DevicePipe data structure

Many methods to insert/extract data into/from a DevicePipe are available. In the DevicePipe class, these methods simply forward their action to the DevicePipe root blob. The same methods are available in the DevicePipeBlob in case you need to use the recursivity provided by this data structure.

Reading a pipe

When you read a pipe, you have to extract data received from the pipe. Because data transferred through a pipe can change at any moment, two differents cases are possible:

1. The client has a prior knowledge of what should be transferred through the pipe
2. The client does not know at all what has been received through the pipe

Those two cases are detailed in the following sub-chapters.

Extracting data with pipe content prior knowledge

To extract data from a DevicePipe object (or from a DevicePipeBlob object), you have to use its extraction operator >>. Let's suppose that we already know (prior knowledge) that the pipe contains 3 data elements with a Tango long, an array of double and finally an array of unsigned short. The code you need to extract these data is (Without error case treatment detailed in a next sub-chapter)

```
1 DevicePipe dp = mydev.read_pipe("MyPipe");
2
3 DevLong dl;
4 vector<double> v_db;
5 DevVarUShortArray *dvush = new DevVarUShortArray();
6
7 dp >> dl >> v_db >> dvush;
8
9 delete dvush;
```

The pipe is read at line 1. Pipe (or root blob) data extracttion is at line 7. As you can see, it is just a matter of chaining extraction operator (>>) into local data (declared line 3 to 5). In this example, the transported array of double is extracted into a C++ vector while the unsigned short array is extracted in a Tango sequence data type. When you extract data into a vector, there is a unavoidable memory copy between the DevicePipe object and the vector. When you extract data in a Tango sequence data type, there is no memory copy but the extraction method consumes the memory and it is therefore caller responsability to delete the memory. This is the rule of line 9. If there is a DevicePipeBlob inside the DevicePipe, simply extract it into one instance of the DevicePipeBlob class.

You may notice that the pipe root blob data elements name are lost in the previous example. The Tango API also has a DataElement class which allows you to retrieve/set data element name. The following code is how you can extract pipe data and retrieve data element name (same pipe then previously)

```
1 DevicePipe dp = mydev.read_pipe("MyPipe");
2
3 DataElement<DevLong> de_dl;
4 DataElement<vector<double> > de_v_db;
5 DataElement<DevVarUShortArray *> de_dvush(new DevVarUShortArray());
6
7 dp >> de_dl >> de_v_db >> de_dvush;
8
9 delete de_dvush.value;
```

The extraction line (number 7) is similar to the previous case but local data are instances of DataElement class. This is template class and instances are created at lines 4 to 6. Each DataElement instance has only two elements which are:

1. The data element name (a C++ string): *name*
2. The data element value (One instance of the template parameter): *value*

Extracting data in a generic way (without prior knowledge)

Due to the dynamicity of the data transferred through a pipe, the API alows to extract data from a pipe without any prior knowledge of its content. This is achived with methods *get_data_elt_nb()*, *get_data_elt_type()*, *get_data_elt_name()* and the extraction operator >>. These methods belong to the DevicePipeBlob class but they also exist on the DevicePipe class for its root blob. Here is one example of how you use them:

```
1  DevicePipe dp = mydev.read_pipe("MyPipe");
2
3  size_t nb_de = dp.get_data_elt_nb();
4  for (size_t loop = 0;loop < nb_de;loop++)
5  {
6     int data_type = dp.get_data_elt_type(loop);
7     string de_name = dp.get_data_elt_name(loop);
8     switch(data_type)
9     {
10       case DEV_LONG:
11       {
12           DevLong lg;
13           dp >> lg;
14       }
15       break;
16
17       case DEVVAR_DOUBLEARRAY:
18       {
19           vector<double> v_db;
20           dp >> v_db;
21       }
22       break;
23       ....
24    }
25    ...
26 }
```

The number of data element in the pipe root blob is retrieve at line 3. Then a loop for each data element is coded. For each data element, its value data type and its name are retrieved at lines 6 and 7. Then, according to the data element value data type, the data are extracted using the classical extraction operator (lines 13 or 20)

Error management

By default, in case of error, the DevicePipe object throws different kind of exceptions according to the error kind. It is possible to disable exception throwing. If you do so, the code has to test the DevicePipe state after extraction. The possible error cases are:

- DevicePipe object is empty
- Wrong data type for extraction (For instance extraction into a double data while the DataElement contains a string)
- Wrong number of DataElement (Extraction code extract 5 data element while the pipe contains only four)
- Mix of extraction (or insertion) method kind (classical operators << or >>) and [] operator.

Methods *exceptions()* and *reset_exceptions()* of the DevicePipe and DevicePipeBlob classes allow the user to select which kind of error he is interested in. For error treatment without exceptions, methods *has_failed()* and *state()* has to be used. See reference documentation for details about these methods.

Writing a pipe

Writing data into a DevicePipe or a DevicePipeBlob is similar to reading data from a pipe. The main method is the insertion operator <<. Let's have a look at a first example if you want to write a pipe with a Tango long, a vector of double and finally an array of unsigned short.

```
1  DevicePipe dp("MyPipe");
2
3  vector<string> de_names {"FirstDE","SecondDE","ThirdDE"};
4  db.set_data_elt_names(de_names);
5
6  DevLong dl = 666;
7  vector<double> v_db {1.11,2.22};
8  unsigned short *array = new unsigned short [100];
9  DevVarUShortArray *dvush = create_DevVarUShortArray(array,100);
10
11 try
12 {
13    dp << dl << v_db << dvush;
14    mydev.write_pipe(dp);
15 }
16 catch (DevFailed &e)
17 {
18    cout << "DevicePipeBlob insertion failed" << endl;
19    ....
20 }
```

Insertion into the DevicePipe is done at line 12 with the insert operators. The main difference with extracting data from the pipe is at line 3 and 4. When inserting data into a pipe, you need to FIRST define its number od name of data elements. In our example, the device pipe is initialized to carry three data element and the names of these data elements is defined at line 4. This is a mandatory requirement. If you don't define data element number, exception will be thrown during the use of insertion methods. The population of the array used for the third pipe data element is not represented here.

It's also possible to use DataElement class instances to set the pipe data element. Here is the previous example modified to use DataElement class.

```
DevicePipe dp("MyPipe");

DataElement<DevLong> de_dl("FirstElt",666);
vector<double>  v_db {1.11,2.22};
DataElement<vector<double> > de_v_db("SecondElt,v_db");

unsigned short *array = new unsigned short [100];
DevVarUShortArray *dvush = create_DevVarUShortArray(array,100);
DataElement<DevVarUShortArray *> de_dvush("ThirdDE",array);

try
{
   dp << de_dl << de_v_db << de_dvush;
   mydev.write_pipe(dp);
}
catch (DevFailed &e)
{
   cout << "DevicePipeBlob insertion failed" << endl;
   ....
}
```

The population of the array used for the third pipe data element is not represented here. Finally, there is a third way to insert data into a device pipe. You have to defined number and names of the data element within the pipe (similar to first insertion method) but you are able to insert data into the data element in any order using the operator overwritten for the DevicePipe and DevicePipeBlob classes. Look at the following example:

```
1  DevicePipe dp("MyPipe");
2
3  vector<string> de_names {"FirstDE","SecondDE","ThirdDE"};
4  db.set_data_elt_names(de_names);
5
6  DevLong dl = 666;
7  vector<double> v_db = {1.11,2.22};
8  unsigned short *array = new unsigned short [100];
9  DevVarUShortArray *dvush = create_DevVarUShortArray(array,100);
10
11 dp["SecondDE"] << v_db;
12 dp["FirstDE"] << dl;
13 dp["ThirdDE"] << dvush;
```

Insertion into the device pipe is now done at lines 11 to 13. The population of the array used for the third pipe data element is not represented here. Note that the data element name is case insensitive.

Error management

When inserting data into a DevicePipe or a DevicePipeBlob, error management is very similar to reading data from from a DevicePipe or a DevicePipeBlob. The difference is that there is one more case which could trigger one exception during the insertion. This case is

- Insertion into the DevicePipe (or DevicePipeBlob) if its data element number have not been set.

Device locking

Starting with Tango release 7 (and device inheriting from Device_4Impl), device locking is supported. For instance, this feature could be used by an application doing a scan on a synchrotron beam line. In such a case, you want to move an actuator then read a sensor, move the actuator again, read the sensor... You don't want the actuator to be moved by another client while the application is doing the scan. If the application doing the scan locks the actuator device, it will be sure that this device is reserved for the application doing the scan and other client will not be able to move it until the scan application un-locks this actuator.

A locked device is protected against:

- *command_inout* call except for device state and status requested via command and for the set of commands defined as allowed following the definition of allowed command in the Tango control access schema.
- *write_attribute* and *write_pipe* call
- *write_read_attribute, write_read_attributes* and *write_read_pipe* call
- *set_attribute_config* and *set_pipe_config* call
- polling and logging commands related to the locked device

Other clients trying to do one of these calls on a locked device will get a DevFailed exception. In case of application with locked device crashed, the lock will be automatically release after a defined interval. The API provides a set of methods for application code to lock/unlock device. These methods are:

- *DeviceProxy::lock()* and *DeviceProxy::unlock()* to lock/unlock device
- *DeviceProxy::locking_status()*, *DeviceProxy::is_locked()*, *DeviceProxy::is_locked_by_me()* and *DeviceProxy::get_locker()* to get locking information

These methods are precisely described in the API reference chapters.

Reconnection and exception

The Tango API automatically manages re-connection between client and server in case of communication error during a network access between a client and a server. By default, when a communication error occurs, an exception is returned to the caller and the connection is internally marked as bad. On the next try to contact the device, the API will try to re-build the network connection. With the *set_transparency_reconnection()* method of the DeviceProxy class, it is even possible not to have any exception thrown in case of communication error. The API will try to re-build the network connection as soon as it is detected as bad. This is the default mode. See *Reconnection and exception* for more details on this subject.

Thread safety

Starting with Tango 7.2, some classes of the C++ API has been made thread safe. These classes are:

- DeviceProxy
- Database
- Group
- ApiUtil
- AttributeProxy

This means that it is possible to share between threads a pointer to a DeviceProxy instance. It is safe to execute a call on this DeviceProxy instance while another thread is also doing a call to the same DeviceProxy instance. Obviously, this also means that it is possible to create thread local DeviceProxy instances and to execute method calls on these instances. Nevertheless, data local to a DeviceProxy instance like its timeout are not managed on a per thread basis. For a DeviceProxy instance shared between two threads, if thread 1 changes the instance timeout, thread 2 will also see this change.

Compiling and linking a Tango client

Compiling and linking a Tango client is similar to compiling and linking a Tango device server. Please, refer to chapter *Compiling and linking a C++ device server* to get all the details.

6.4.2 TangoATK Programmer's Guide

Intended audience: developers, ***Programming language:*** java

This chapter is only a brief Tango ATK (Application ToolKit) programmer's guide. You can find a reference guide with a full description of TangoATK classes and methods in the ATK JavaDoc (Tango ATK reference on-line documentation).

A tutorial document Tango ATK Tutorial is also provided and includes the detailed description of the ATK architecture and the ATK components. In the ATK Tutorial you can find some code examples and also Flash Demos which explain how to start using Tango ATK.

Introduction

This document describes how to develop applications using the Tango Application Toolkit, TangoATK for short. It will start with a brief description of the main concepts behind the toolkit, and then continue with more practical, real-life examples to explain key parts.

Assumptions

The author assumes that the reader has a good knowledge of the Java programming language, and a thorough understanding of object-oriented programming. Also, it is expected that the reader is fluent in all aspects regarding Tango devices, attributes, and commands.

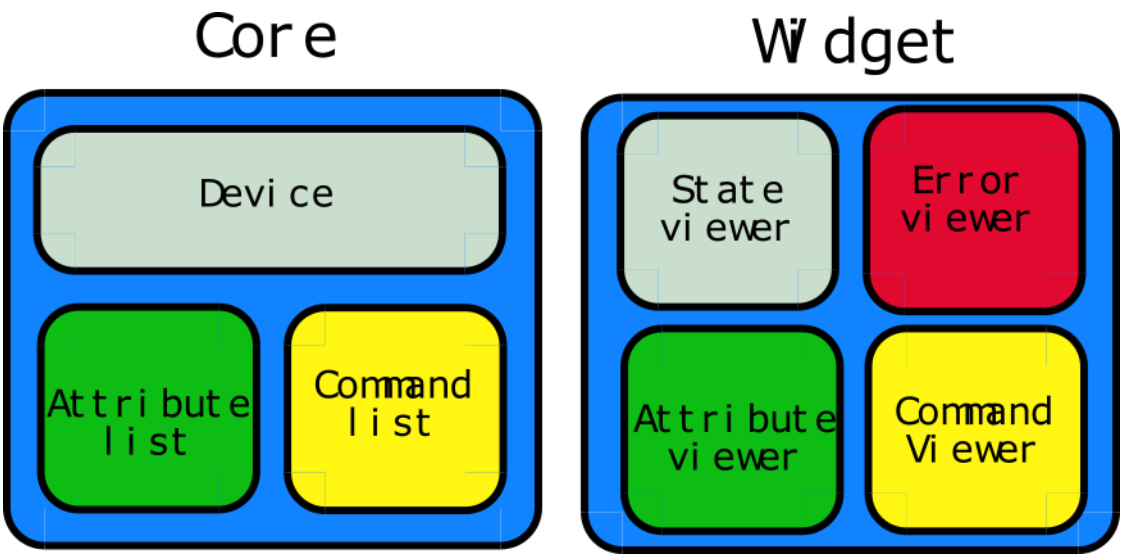
The key concepts of TangoATK

TangoATK was developed with these goals in mind

- TangoATK should help minimize development time
- TangoATK should help minimize bugs in applications
- TangoATK should support applications that contain attributes and commands from several different devices.
- TangoATK should help avoid code duplication.

Since most Tango-applications were foreseen to be displayed on some sort of graphic terminal, TangoATK needed to provide support for some sort of graphic building blocks. To enable this, and since the toolkit was to be written in Java, we looked to Swing to figure out how to do this.

Swing is developed using a variant over a design-pattern the Model-View-Controller (MVC) pattern called *model-delegate*, where the view and the controller of the MVC-pattern are merged into one object.



This pattern made the choice of labor division quite easy: all non-graphic parts of TangoATK reside in the packages beneath *fr.esrf.tangoatk.core*, and anything remotely graphic are located beneath *fr.esrf.tangoatk.widget*. More on the content and organization of this will follow.

The communication between the non-graphic and graphic objects are done by having the graphic object registering itself as a listener to the non-graphic object, and the non-graphic object emmiting events telling the listeners that its state has changed.

Minimize development time

For TangoATK to help minimize the development time of graphic applications, the toolkit has been developed along two lines of thought

- Things that are needed in most applications are included, eg `Splash`, a splash window which gives a graphical way for the application to show the progress of a long operation. The splash window is moslty used in the startup phase of the application.
- Building blocks provided by TangoATK should be easy to use and follow certain patterns, eg every graphic widget has a `setModel` method which is used to connect the widget with its non-graphic model.

In addition to this, TangoATK provides a framework for error handling, something that is often a time consuming task.

Minimize bugs in applications

Together with the Tango API, TangoATK takes care of most of the hard things related to programming with Tango. Using TangoATK the developer can focus on developing her application, not on understanding Tango.

Attributes and commands from different devices

To be able to create applications with attributes and commands from different devices, it was decided that the central objects of TangoATK were not to be the device, but rather the *attributes and the commands*. This will certainly feel a bit awkward at first, but trust me, the design holds.

For this design to be feasible, a structure was needed to keep track of the commands and attributes, so the *command-list and the attribute-list* was introduced. These two objects can hold commands and attributes from any number of devices.

Avoid code duplication

When writing applications for a control-system without a framework much code is duplicated. Anything from simple widgets for showing numeric values to error handling has to be implemented each time. TangoATK supplies a number of frequently used widgets along with a framework for connecting these widgets with their non-graphic counterparts. Because of this, the developer only needs to write the *glue* - the code which connects these objects in a manner that suits the specified application.

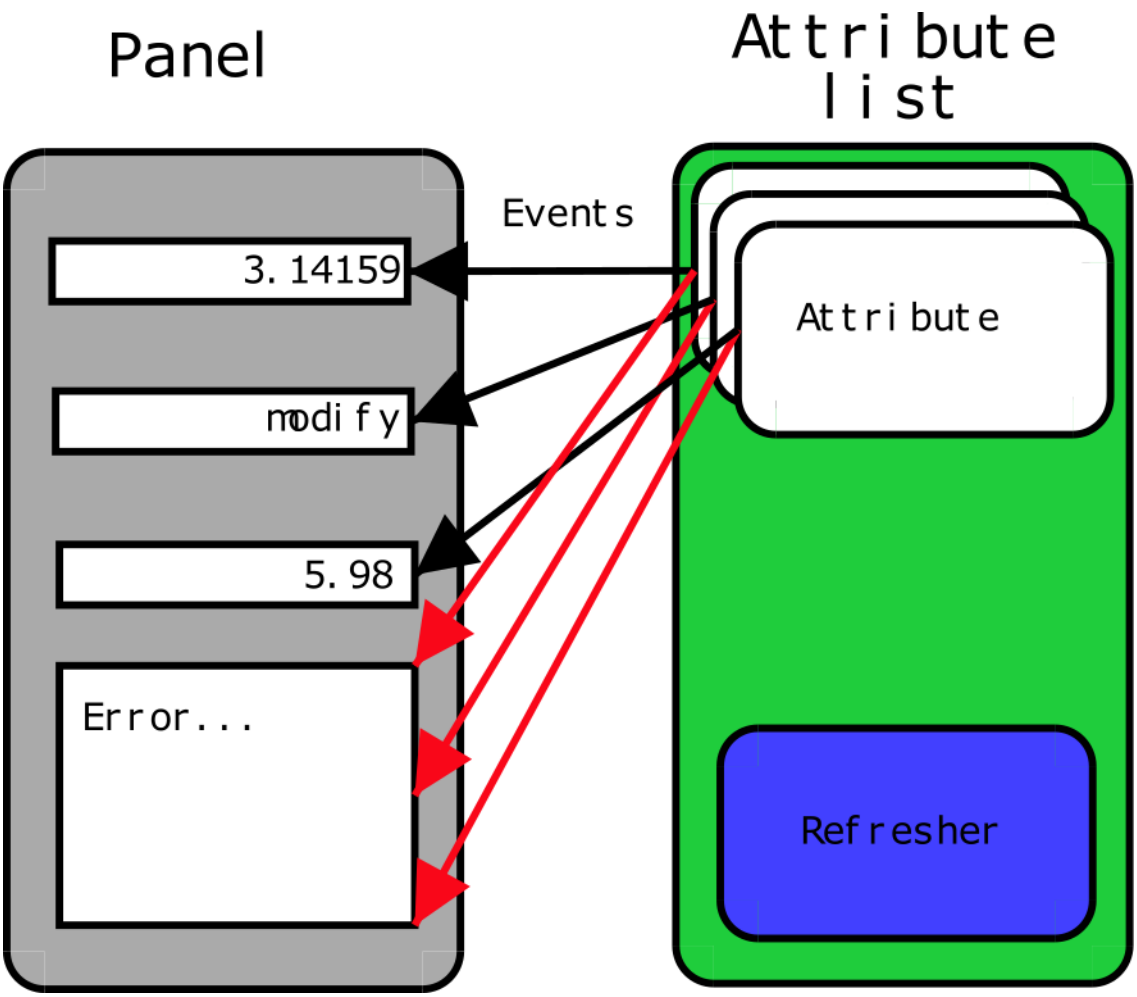
The real getting started

Generally there are two kinds of end-user applications: Applications that only know how to treat one device, and applications that treat many devices.

Single device applications

Single device applications are quite easy to write, even with a gui. The following steps are required

1. Instantiate an AttributeList and fill it with the attributes you want.
2. Instantiate a CommandList and fill it with the commands you want.
3. Connect the whole *AttributeList* with a *list viewer* and / or each *individual attribute* with an *attribute viewer*.
4. Connect the whole *CommandList* to a *command list viewer* and / or connect each *individual command* in the command list with a *command viewer*.



The following program (FirstApplication) shows an implementation of the list mentioned above. It should be rather self-explanatory with the comments.

```
1  package examples;
2
3
4  import javax.swing.JFrame;
5  import javax.swing.JMenuItem;
6  import javax.swing.JMenuBar;
7  import javax.swing.JMenu;
8
9
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
12 import java.awt.BorderLayout;
13
14
15 import fr.esrf.tangoatk.core.AttributeList;
16 import fr.esrf.tangoatk.core.ConnectionException;
17
18
19 import fr.esrf.tangoatk.core.CommandList;
```

(continues on next page)

(continued from previous page)

```
import fr.esrf.tangoatk.widget.util.ErrorHistory;
import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
import fr.esrf.tangoatk.widget.attribute.ScalarListViewer;
import fr.esrf.tangoatk.widget.command.CommandComboViewer;


public class FirstApplication extends JFrame
{
JMenuBar menu;                    // So that our application looks
                                  // halfway decent.
AttributeList attributes;         // The list that will contain our
                                  // attributes
CommandList commands;             // The list that will contain our
                                  // commands
ErrorHistory errorHistory;        // A window that displays errors
ScalarListViewer sListViewer;     // A viewer which knows how to
                                  // display a list of scalar attributes.
                                  // If you want to display other types
                                  // than scalars, you'll have to wait
                                  // for the next example.
CommandComboViewer commandViewer; // A viewer which knows how to display
                                  // a combobox of commands and execute
                                  // them.
String device;                    // The name of our device.


public FirstApplication()
{
   // The swing stuff to create the menu bar and its pulldown menus
   menu = new JMenuBar();
   JMenu fileMenu = new JMenu();
   fileMenu.setText("File");
   JMenu viewMenu = new JMenu();
   viewMenu.setText("View");

   JMenuItem quitItem = new JMenuItem();
   quitItem.setText("Quit");
   quitItem.addActionListener(new
      java.awt.event.ActionListener()
      {
       public void
       actionPerformed(ActionEvent evt)
       {quitItemActionPerformed(evt);}
      });
   fileMenu.add(quitItem);

   JMenuItem errorHistItem = new JMenuItem();
   errorHistItem.setText("Error History");
   errorHistItem.addActionListener(new
           java.awt.event.ActionListener()
           {
            public void
```

(continues on next page)

(continued from previous page)

```
            actionPerformed(ActionEvent evt)
            {errHistItemActionPerformed(evt);}
          });
    viewMenu.add(errorHistItem);
    menu.add(fileMenu);
    menu.add(viewMenu);

    //
    // Here we create ATK objects to handle attributes, commands and errors.
    //
    attributes = new AttributeList();
    commands = new CommandList();
    errorHistory = new ErrorHistory();
    device = "id14/eh3_mirror/1";
    sListViewer = new ScalarListViewer();
    commandViewer = new CommandComboViewer();


//
// A feature of the command and attribute list is that if you
// supply an errorlistener to these lists, they'll add that
// errorlistener to all subsequently created attributes or
// commands. So it is important to do this _before_ you
// start adding attributes or commands.
//

    attributes.addErrorListener(errorHistory);
    commands.addErrorListener(errorHistory);

//
// Sometimes we're out of luck and the device or the attributes
// are not available. In that case a ConnectionException is thrown.
// This is why we add the attributes in a try/catch
//

    try
    {

//
// Another feature of the attribute and command list is that they
// can add wildcard names, currently only `*' is supported.
// When using a wildcard, the lists will add all commands or
// attributes available on the device.
//
    attributes.add(device + "/*");
    }
    catch (ConnectionException ce)
    {
       System.out.println("Error fetching " +
                          "attributes from " +
                          device + " " + ce);
    }
```

(continues on next page)

(continued from previous page)

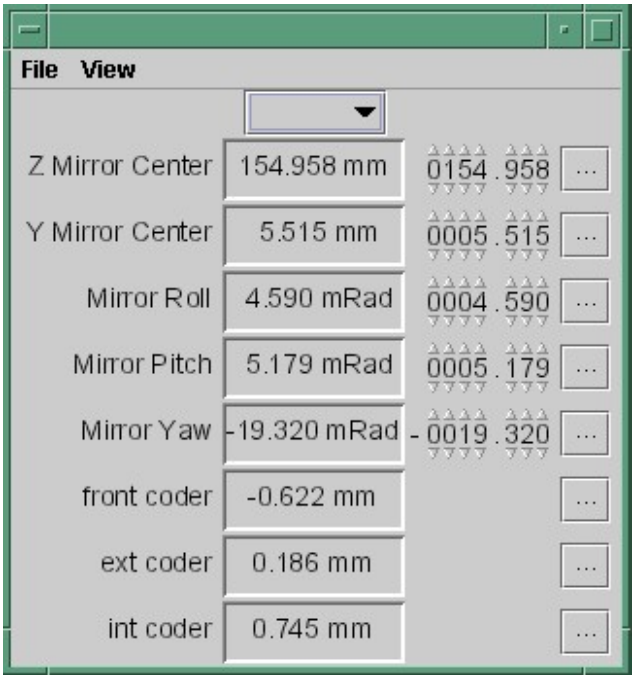
```
124
125
126     //
127     // See the comments for attributelist
128     //
129
130
131        try
132        {
133           commands.add(device + "/*");
134        }
135        catch (ConnectionException ce)
136        {
137           System.out.println("Error fetching " +
138                              "commands from " +
139                              device + " " + ce);
140        }
141
142
143     //
144     // Here we tell the scalarViewer what it's to show. The
145     // ScalarListViewer loops through the attribute-list and picks out
146     // the ones which are scalars and show them.
147     //
148
149        sListViewer.setModel(attributes);
150
151
152     //
153     // This is where the CommandComboViewer is told what it's to
154     // show. It knows how to show and execute most commands.
155     //
156
157
158        commandViewer.setModel(commands);
159
160
161     //
162     // add the menubar to the frame
163     //
164
165
166        setJMenuBar(menu);
167
168
169     //
170     // Make the layout nice.
171     //
172
173
174        getContentPane().setLayout(new BorderLayout());
175        getContentPane().add(commandViewer, BorderLayout.NORTH);
```

(continues on next page)

(continued from previous page)

```
176     getContentPane().add(sListViewer, BorderLayout.SOUTH);
177
178
179  //
180  // A third feature of the attributelist is that it knows how
181  // to refresh its attributes.
182  //
183
184
185     attributes.startRefresher();
186
187
188  //
189  // JFrame stuff to make the thing show.
190  //
191
192
193     pack();
194     ATKGraphicsUtils.centerFrameOnScreen(this); //ATK utility to center window
195
196     setVisible(true);
197     }
198
199
200     public static void main(String [] args)
201     {
202        new FirstApplication();
203     }
204
205     public void quitItemActionPerformed(ActionEvent evt)
206     {
207        System.exit(0);
208     }
209
210     public void errHistItemActionPerformed(ActionEvent evt)
211     {
212        errorHistory.setVisible(true);
213     }
214  }
```

The program should look something like this (depending on your platform and your device)



Multi device applications

Multi device applications are quite similar to the single device applications, the only difference is that it does not suffice to add the attributes by wildcard, you need to add them explicitly, like this:

```
try
{
    // a StringScalar attribute from the device one
   attributes.add("jlp/test/1/att_cinq");
   // a NumberSpectrum attribute from the device one
   attributes.add("jlp/test/1/att_spectrum");
   // a NumberImage attribute from the device two
   attributes.add("sr/d-ipc/id25-1n/Image");
}
catch (ConnectionException ce)
{
   System.out.println("Error fetching " +
       "attributes" + ce);
}
```

The same goes for commands.

More on displaying attributes

So far, we've only considered scalar attributes, and not only that, we've also cheated quite a bit since we just passed the attribute list to the `fr.esrf.tangoatk.widget.attribute.ScalarListViewer` and let it do all the magic. The attribute list viewers are only available for scalar attributes (NumberScalarListViewer and ScalarListViewer). If you have one or several spectrum or image attributes you must connect each spectrum or image attribute to it's corresponding attribute viewer individually. So let's take a look at how you can connect individual attributes (and not a whole attribute list) to an individual attribute viewer (and not to an attribute list viewer).

Connecting an attribute to a viewer

Generally it is done in the following way:

1. You retrieve the attribute from the attribute list
2. You instantiate the viewer
3. Your call the `setModel` method on the viewer with the attribute as argument.
4. You add your viewer to some panel

The following example (SecondApplication), is a Multi-device application. Since this application uses individual attribute viewers and not an attribute list viewer, it shows an implementation of the list mentioned above.

```
1  package examples;
2
3
4  import javax.swing.JFrame;
5  import javax.swing.JMenuItem;
6  import javax.swing.JMenuBar;
7  import javax.swing.JMenu;
8
9
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
12 import java.awt.BorderLayout;
13 import java.awt.Color;
14
15
16 import fr.esrf.tangoatk.core.AttributeList;
17 import fr.esrf.tangoatk.core.ConnectionException;
18
19 import fr.esrf.tangoatk.core.IStringScalar;
20 import fr.esrf.tangoatk.core.INumberSpectrum;
21 import fr.esrf.tangoatk.core.INumberImage;
22 import fr.esrf.tangoatk.widget.util.ErrorHistory;
23 import fr.esrf.tangoatk.widget.util.Gradient;
24 import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
25 import fr.esrf.tangoatk.widget.attribute.NumberImageViewer;
26 import fr.esrf.tangoatk.widget.attribute.NumberSpectrumViewer;
27 import fr.esrf.tangoatk.widget.attribute.SimpleScalarViewer;
28
29 public class SecondApplication extends JFrame
30 {
```

(continues on next page)

(continued from previous page)

```
        JMenuBar            menu;
        AttributeList       attributes;   // The list that will contain our attributes
        ErrorHistory        errorHistory; // A window that displays errors
        IStringScalar        ssAtt;
        INumberSpectrum      nsAtt;
        INumberImage         niAtt;
        public SecondApplication()
        {
           // Swing stuff to create the menu bar and its pulldown menus
           menu = new JMenuBar();
           JMenu fileMenu = new JMenu();
           fileMenu.setText("File");
           JMenu viewMenu = new JMenu();
           viewMenu.setText("View");
           JMenuItem quitItem = new JMenuItem();
           quitItem.setText("Quit");
           quitItem.addActionListener(new java.awt.event.ActionListener()
                                      {
                                       public void actionPerformed(ActionEvent evt)
                                       {quitItemActionPerformed(evt);}
                                      });

           fileMenu.add(quitItem);
           JMenuItem errorHistItem = new JMenuItem();
           errorHistItem.setText("Error History");
           errorHistItem.addActionListener(new java.awt.event.ActionListener()
                   {
                    public void actionPerformed(ActionEvent evt)
                    {errHistItemActionPerformed(evt);}
                   });
           viewMenu.add(errorHistItem);
           menu.add(fileMenu);
           menu.add(viewMenu);
         //
         // Here we create TangoATK objects to view attributes and errors.
         //
           attributes = new AttributeList();
           errorHistory = new ErrorHistory();
         //
         // We create a SimpleScalarViewer, a NumberSpectrumViewer and
         // a NumberImageViewer, since we already knew that we were
         // playing with a scalar attribute, a number spectrum attribute
         // and a number image attribute this time.
         //
         SimpleScalarViewer     ssViewer = new SimpleScalarViewer();
           NumberSpectrumViewer   nSpectViewer = new NumberSpectrumViewer();
           NumberImageViewer      nImageViewer = new NumberImageViewer();
           attributes.addErrorListener(errorHistory);
        //
        // The attribute (and command) list has the feature of returning the last
        // attribute that was added to it. Just remember that it is returned as an
        // IEntity object, so you need to cast it into a more specific object, like
```

(continues on next page)

(continued from previous page)

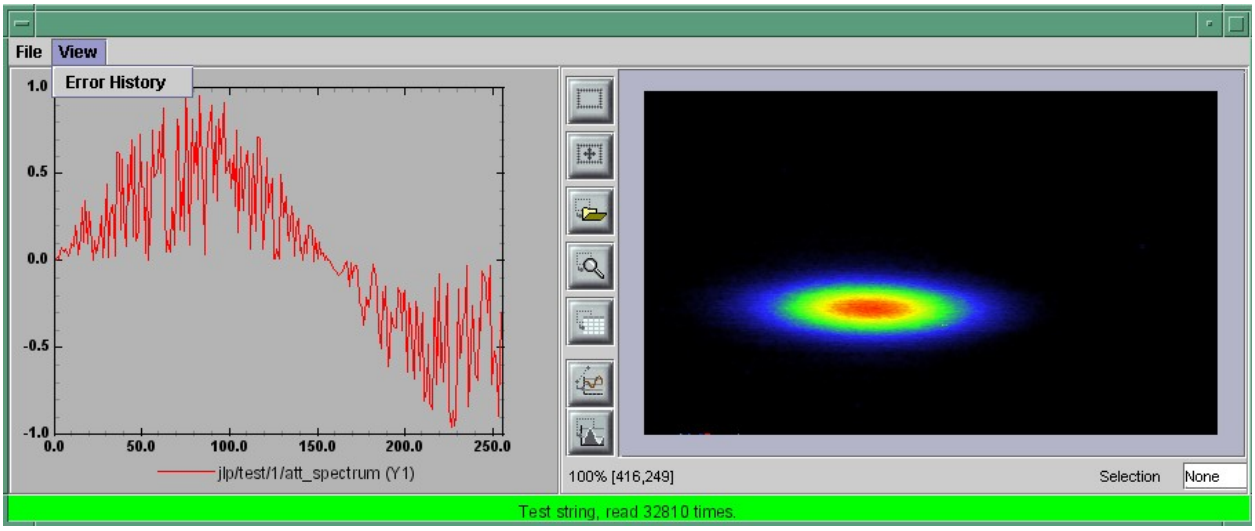
```
        // IStringScalar, which is the interface which defines a string scalar
        //
          try
           {

              ssAtt = (IStringScalar) attributes.add("jlp/test/1/att_cinq");
              nsAtt = (INumberSpectrum) attributes.add("jlp/test/1/att_spectrum");
              niAtt = (INumberImage) attributes.add("sr/d-ipc/id25-1n/Image");
           }
           catch (ConnectionException ce)
           {
              System.out.println("Error fetching one of the attributes  "+" " + ce);
              System.out.println("Application Aborted.");
              System.exit(0);
           }
           //
           // Pay close attention to the following three lines!! This is how it's done!
           // This is how it's always done! The setModelsetModel method of any viewer␣
↪takes care
          // of connecting the viewer to the attribute (model) it's in charge of␣
↪displaying.
          // This is the way to tell each viewer what (which attribute) it has to show.
          // Note that we use a viewer adapted to each type of attribute
          //
           ssViewer.setModel(ssAtt);
           nSpectViewer.setModel(nsAtt);
           nImageViewer.setModel(niAtt);
        //
           nSpectViewer.setPreferredSize(new java.awt.Dimension(400, 300));
           nImageViewer.setPreferredSize(new java.awt.Dimension(500, 300));
           Gradient  g = new Gradient();
           g.buidColorGradient();
           g.setColorAt(0,Color.black);
           nImageViewer.setGradient(g);
           nImageViewer.setBestFit(true);

           //
           // Add the viewers into the frame to show them
           //
           getContentPane().setLayout(new BorderLayout());
           getContentPane().add(ssViewer, BorderLayout.SOUTH);
           getContentPane().add(nSpectViewer, BorderLayout.CENTER);
           getContentPane().add(nImageViewer, BorderLayout.EAST);
           //
           // To have the attributes values refreshed we should start the
           // attribute list's refresher.
           //
           attributes.startRefresher();
           //
           // add the menubar to the frame
           //
           setJMenuBar(menu);
```

(continues on next page)

(continued from previous page)

```
            //
            // JFrame stuff to make the thing show.
            //
            pack();
            ATKGraphicsUtils.centerFrameOnScreen(this); //ATK utility to center window
            setVisible(true);
        }
        public static void main(String [] args)
        {
            new SecondApplication();
        }
        public void quitItemActionPerformed(ActionEvent evt)
        {
            System.exit(0);
        }
        public void errHistItemActionPerformed(ActionEvent evt)
        {
            errorHistory.setVisible(true);
        }
    }
```

This program (SeondApplication) should look something like this (depending on your platform and your device attributes)



Synoptic viewer

TangoATK provides a generic class to view and to animate the synoptics. The name of this class is fr.esrf.tangoatk.widget.jdraw.SynopticFileViewer. This class is based on a "home-made" graphical layer called jdraw. The jdraw package is also included inside TangoATK distribution.

SynopticFileViewer is a sub-class of the class TangoSynopticHandler. All the work for connection to tango devices and run time animation is done inside the TangoSynopticHandler.

The recipe for using the TangoATK synoptic viewer is the following

1. You use Jdraw graphical editor to draw your synoptic
2. During drawing phase don't forget to associate parts of the drawing to tango attributes or commands. Use the "name" in the property window to do this
3. During drawing phase you can also aasociate a class (frequently a "specific panel" class) which will be displayed when the user clicks on some part of the drawing. Use the "extension" tab in the property window to do this.
4. Test the run-time behaviour of your synoptic. Use "Tango Synoptic view" command in the "views" pulldown menu to do this.
5. Save the drawing file.
6. There is a simple synoptic application (SynopticAppli) which is provided ready to use. If this generic application is enough for you, you can forget about the step 7.
7. You can now develop a specific TangoATK based application which instantiates the SynopticFileViewer. To load the synoptic file in the SynopticFileViewer you have the choice : either you load it by giving the absolute path name of the synoptic file or you load the synoptic file using Java input streams. The second solution is used when the synoptic file is included inside the application jarfile.

The SynopticFilerViewer will browse the objects in the synoptic file at run time. It discovers if some parts of the drawing is associated with an attribute or a command. In this case it will automatically connect to the corresponding attribute or command. Once the connection is successful SynopticFileViewer will animate the synoptic according to the default behaviour described below :

- For *tango state attributes* : the colour of the drawing object reflects the value of the state. A mouse click on the drawing object associated with the tango state attribute will instantiate and display the class specified during the drawing phase. If no class is specified the atkpanel generic device panel is displayed.
- For *tango attributes* : the current value of the attribute is displayed through the drawing object
- For *tango commands* : the mouse click on the drawing object associated with the command will launch the device command.
- If the tooltip property is set to "name" when the mouse enters *any tango object* (attribute or command), inside the synoptic drawing the name of the tango object is displayed in a tooltip.

The following example (ThirdApplication), is a Synoptic application. We assume that the synoptic has already been drawn using Jdraw graphical editor.

```
1  package examples;
2  import java.io.*;
3  import java.util.*;
4  import javax.swing.JFrame;
5  import javax.swing.JMenuItem;
6  import javax.swing.JMenuBar;
7  import javax.swing.JMenu;
8  import java.awt.event.ActionListener;
```

(continues on next page)

(continued from previous page)

```
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;
import fr.esrf.tangoatk.widget.util.ErrorHistory;
import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
import fr.esrf.tangoatk.widget.jdraw.SynopticFileViewer;
import fr.esrf.tangoatk.widget.jdraw.TangoSynopticHandler;
public class ThirdApplication extends JFrame
{
     JMenuBar              menu;
     ErrorHistory          errorHistory;  // A window that displays errors
     SynopticFileViewer    sfv;           // TangoATK generic synoptic viewer


     public ThirdApplication()
     {
        // Swing stuff to create the menu bar and its pulldown menus
        menu = new JMenuBar();
        JMenu fileMenu = new JMenu();
        fileMenu.setText("File");
        JMenu viewMenu = new JMenu();
        viewMenu.setText("View");
        JMenuItem quitItem = new JMenuItem();
        quitItem.setText("Quit");
        quitItem.addActionListener(new java.awt.event.ActionListener()
                                   {
                                    public void actionPerformed(ActionEvent evt)
                                    {quitItemActionPerformed(evt);}
                                   });
        fileMenu.add(quitItem);
        JMenuItem errorHistItem = new JMenuItem();
        errorHistItem.setText("Error History");
        errorHistItem.addActionListener(new java.awt.event.ActionListener()
                {
                 public void actionPerformed(ActionEvent evt)
                 {errHistItemActionPerformed(evt);}
                });
        viewMenu.add(errorHistItem);
        menu.add(fileMenu);
        menu.add(viewMenu);
        //
        // Here we create TangoATK synoptic viewer and error window.
        //
        errorHistory = new ErrorHistory();
        sfv = new SynopticFileViewer();
        try
        {
            sfv.setErrorWindow(errorHistory);
        }
        catch (Exception setErrwExcept)
        {
            System.out.println("Cannot set Error History Window");
        }
```

(continues on next page)

(continued from previous page)

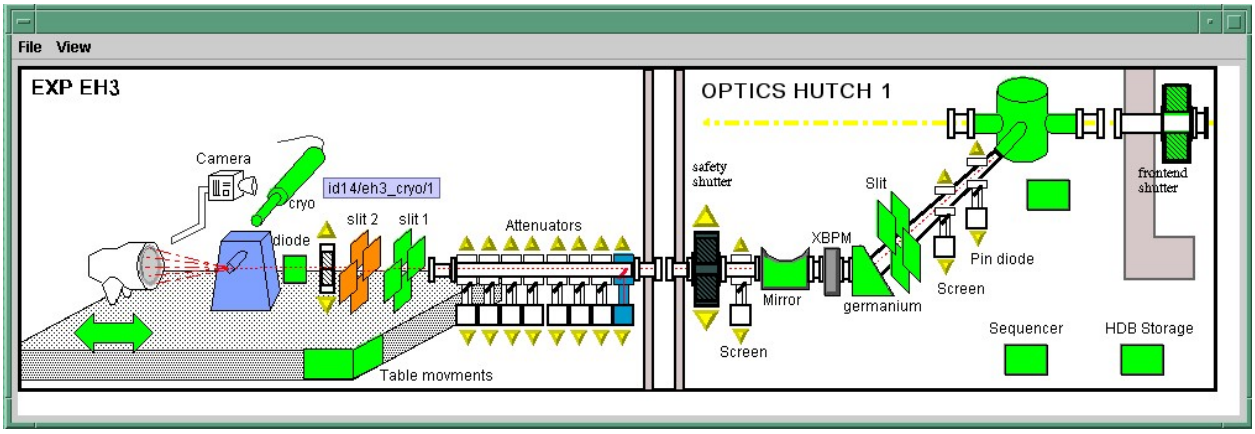
```
61
62          //
63          // Here we define the name of the synoptic file to show and the tooltip mode␣
↪to use
64          //
65          try
66          {
67            sfv.setJdrawFileName("/users/poncet/ATK_OLD/jdraw_files/id14.jdw");
68            sfv.setToolTipMode (TangoSynopticHandler.TOOL_TIP_NAME);
69          }
70          catch (FileNotFoundException  fnfEx)
71          {
72             javax.swing.JOptionPane.showMessageDialog(
73                null, "Cannot find the synoptic file : id14.jdw.\n"
74                      + "Check the file name you entered;"
75                      + " Application will abort ...\n"
76                      + fnfEx,
77                      "No such file",
78                      javax.swing.JOptionPane.ERROR_MESSAGE);
79             System.exit(-1);
80          }
81          catch (IllegalArgumentException  illEx)
82          {
83             javax.swing.JOptionPane.showMessageDialog(
84                null, "Cannot parse the synoptic file : id14.jdw.\n"
85                      + "Check if the file is a Jdraw file."
86                      + " Application will abort ...\n"
87                      + illEx,
88                      "Cannot parse the file",
89                      javax.swing.JOptionPane.ERROR_MESSAGE);
90             System.exit(-1);
91          }
92          catch (MissingResourceException  mrEx)
93          {
94             javax.swing.JOptionPane.showMessageDialog(
95                null, "Cannot parse the synoptic file : id14.jdw.\n"
96                      + " Application will abort ...\n"
97                      + mrEx,
98                      "Cannot parse the file",
99                      javax.swing.JOptionPane.ERROR_MESSAGE);
100            System.exit(-1);
101         }
102         //
103         // Add the viewers into the frame to show them
104         //
105         getContentPane().setLayout(new BorderLayout());
106         getContentPane().add(sfv, BorderLayout.CENTER);
107         //
108         // add the menubar to the frame
109         //
110         setJMenuBar(menu);
111         //
```

(continues on next page)

(continued from previous page)

```
112             // JFrame stuff to make the thing show.
113             //
114             pack();
115             ATKGraphicsUtils.centerFrameOnScreen(this); //TangoATK utility to center window
116             setVisible(true);
117         }
118         public static void main(String [] args)
119         {
120             new ThirdApplication();
121         }
122         public void quitItemActionPerformed(ActionEvent evt)
123         {
124             System.exit(0);
125         }
126         public void errHistItemActionPerformed(ActionEvent evt)
127         {
128             errorHistory.setVisible(true);
129         }
130     }
```

The synoptic application (ThirdApplication) should look something like this (depending on your synoptic drawing file)



A short note on the relationship between models and viewers

As seen in the examples above, the connection between a model and its viewer is generally done by calling `setModel(model)` on the viewer, it is never explained what happens behind the scenes when this is done.

Listeners

Most of the viewers implement some sort of *listener* interface, eg INumberScalarListener. An object implementing such a listener interface has the capability of receiving and treating *events* from a model which emits events.

```
// this is the setModel of a SimpleScalarViewer
  public void setModelsetModel(INumberScalar scalar) {

    clearModel();

    if (scalar != null) {
      format = scalar.getProperty("format").getPresentation();
      numberModel = scalar;

   // this is where the viewer connects itself to the
   // model. After this the viewer will (hopefully) receive
   // events through its numberScalarChange() method

    numberModel.addNumberScalarListener(this);


        numberModel.getProperty("format").addPresentationListener(this);
      numberModel.getProperty("unit").addPresentationListener(this);
    }

  }



// Each time the model of this viewer (the numberscalar attribute) decides it is time, it
// calls the numberScalarChange method of all its registered listeners
// with a NumberScalarEvent object which contains the
// the new value of the numberscalar attribute.
//

  public void numberScalarChange(NumberScalarEvent evt) {
    String val;
    val = getDisplayString(evt);
    if (unitVisible) {
      setText(val + " " + numberModel.getUnit());
    } else {
      setText(val);
    }
  }
```

All listeners in TangoATK implement the `IErrorListener` interface which specifies the `errorChange(ErrorEvent e)` method. This means that all listeners are forced to handle errors in some way or another.

The key objects of TangoATK

As seen from the examples above, the key objects of TangoATK are the `CommandList` and the `AttributeList`. These two classes inherit from the abstract class `AEntityList` which implements all of the common functionality between the two lists. These lists use the functionality of the `CommandFactory`, the `AttributeFactory`, which both derive from `AEntityFactory`, and the `DeviceFactory`.

In addition to these factories and lists there is one (for the time being) other important functionality lurking around, the refreshers.

The Refreshers

The refreshers, represented in TangoATK by the `Refresher` object, is simply a subclass of `java.lang.Thread` which will sleep for a given amount of time and then call a method refresh on whatever kind of `IRefreshee` it has been given as parameter, as shown below

```
1  // This is an example from DeviceFactory.
2  // We create a new Refresher with the name "device"
3  // We add ourself to it, and start the thread
4
5
6  Refresher refresher = new Refresher("device");
7  refresher.addRefreshee(this).start();
```

Both the `AttributeList` and the `DeviceFactory` implement the `IRefreshee` interface which specify only one method, `refresh()`, and can thus be refreshed by the `Refresher`. Even if the new release of TangoATK is based on the Tango Events, the refresher mecanisme will not be removed. As a matter of fact, the method refresh() implemented in AttributeList skips all attributes (members of the list) for which the subscribe to the tango event has succeeded and calls the old refresh() method for the others (for which subscribe to tango events has failed).

In a first stage this will allow the TangoATK applications to mix the use of the old tango device servers (which do not implement tango events) and the new ones in the same code. In other words, TangoATK subscribes for tango events if possible otherwise TangoATK will refresh the attributes through the old refresher mecanisme.

Another reason for keeping the refresher is that the subscribe event can fail even for the attributes of the new Tango device servers. As soon as the specified attribute is not polled the Tango events cannot be generated for that attribute. Therefore the event subscription will fail. In this case the attribute will be refreshed thanks to the ATK attribute list refresher.

The `AttributePolledList` class allows the application programmer to force explicitly the use of the refresher method for all attributes added in an AttributePolledList even if the corresponding device servers implement tango events. Some viewers (fr.esrf.tangoatk.widget.attribute.Trend) need an AttributePolledList in order to force the refresh of the attribute without using tango events.

What happens on a refresh

When `refresh` is called on the `AttributeList` and the `DeviceFactory`, they loop through their objects, `IAttributes` and `IDevices`, respectively, and ask them to refresh themselves if they are not event driven.

When AttributeFactory, creates an `IAttribute`, TangoATK tries to subscribe for Tango Change event for that attribute. If the subscription succeeds then the attribute is marked as event driven. If the subscription for Tango Change event fails, TangoATK tries to subscribe for Tango Periodic event. If the subscription succeeds then the attribute is marked as event driven. If the subscription fails then the attribute is marked as to be " without events".

In the refresh() method of the AttributeList during the loop through the objects if the object is marked event driven then the object is simply skipped. But if the object (attribute) is not marked as event driven, the refresh() method of the AttributeList, asks the object to refresh itself by calling the "refresh()" method of that object (attribute or device). The refresh() method of an attribute will in turn call the "readAttribute" on the Tango device.

The result of this is that the `IAttributes` fire off events to their registered listeners containing snapshots of their state. The events are fired either because the IAttribute has received a Tango Change event, respectively a Tango Periodic event (event driven objects), or because the refresh() method of the object has issued a readAttribute on the Tango device.

The DeviceFactory

The device factory is responsible for two things

1. Creating new devices (Tango device proxies) when needed
2. Refreshing the state and status of these devices

Regarding the first point, new devices are created when they are asked for and only if they have not already been created. If a programmer asks for the same device twice, she is returned a reference to the same device-object.

The `DeviceFactory` contains a Refresher as described above, which makes sure that the all in the updates their state and status and fire events to its listeners.

The AttributeFactory and the CommandFactory

These factories are responsible for taking a name of an attribute or command and returning an object representing the attribute or command. It is also responsible for making sure that the appropriate `IDevice` is already available. Normally the programmer does not want to use these factory classes directly. They are used by TangoATK classes indirectly when the application programmer calls the AttributeList's (or CommandList's) add() method.

The AttributeList and the CommandList

These lists are containers for attributes and commands. They delegate the construction-work to the factories mentioned above, and generally do not do much more, apart from containing refreshers, and thus being able to make the objects they contain refresh their listeners.

The Attributes

The attributes come in several flavors. Tango supports the following types:

- Short
- Long
- Double
- String
- Unsigned Char
- Boolean
- Unsigned Short
- Float

- Unsigned Long

According to Tango specifications, all these types can be of the following formats:

- Scalar, a single value
- Spectrum, a single array
- Image, a two dimensional array

For the sake of simplicity, TangoATK has combined all the numeric types into one, presenting all of them as doubles. So the TangoATK classes which handle the numeric attributes are : NumberScalar, NumberSpectrum and NumberImage (Number can be short, long, double, float, ...).

The hierarchy

The numeric attribute hierarchy is expressed in the following interfaces:

`INumberScalar` extends `INumber`

`INumberSpectrum` extends `INumber`

`INumberImage` extends `INumber`

`INumber` in turn extends `IAttribute`

Each of these types emit their proper events and have their proper listeners. Please consult the javadoc for further information.

The Commands

The commands in Tango are rather ugly beasts. There exists the following kinds of commands

- Those which take input
- Those which do not take input
- Those which do output
- Those which do not do output

Now, for both input and output we have the following types:

- Double
- Float
- Unsigned Long
- Long
- Unsigned Short
- Short
- String

These types can appear in scalar or array formats. In addition to this, there are also four other types of parameters:

1. Boolean
2. Unsigned Char Array
3. The StringLongArray

4. The StringDoubleArray

The last two types mentioned above are two-dimensional arrays containing a string array in the first dimension and a long or double array in the second dimension, respectively.

As for the attributes, all numeric types have been converted into doubles, but there has been made little or no effort to create an hierarchy of types for the commands.

Events and listeners

The commands publish results to their `IResultListener`s, by the means of a `ResultEvent`. The `IResultListener` extends `IErrorListener`, any viewer of command-results should also know how to handle errors. So a viewer of command-results implements IResultListener interface and registers itself as a resultListener for the command it has to show the results.

6.4.3 PyTango - a Python binding to Tango

Python is a commonly used programming language in the scientific community, due to its many advantages (the most important of those is probably simplicity of its syntax). *Tango Controls* also supports it in a form of a Boost-based binding to C++ Tango implementation.

In "pythonic" terms, it is a package available at PyPI that exposes the complete Tango API (both the client and the server parts of it) as well as provides a framework for unit-testing your *device servers*.

Note: You should use PyTango that has major and minor version numbers the same as Tango C++ library that you have. So if you have Tango C++ library version X.Y.Z, you should have PyTango version X.Y.V (where V might equal Z, but its not required).

You can find its full documentation here.

6.5 Device Servers

Intended audience: developers, ***Programming language:*** all

6.5.1 Introduction to device server

Intended audience: developers, ***Programming language:*** all

Device servers were first developed at the European Synchrotron radiation Facility (ESRF) for controlling the 6 Gev synchrotron radiation source. This document is a Programmer's Manual on how to write TANGO device servers. It will not go into the details of the ESRF, nor its Control System nor any of the specific device servers in the Control System. The role of this document is to help programmers faced with the task of writing TANGO device servers.

Device servers have been developed at the ESRF in order to solve the main task of Control Systems viz provide read and write access to all devices in a distributed system. The problem of distributed device access is only part of the problem however. The other part of the problem is providing a programming framework for a large number of devices programmed by a large number of programmers each having different levels of experience and style.

Device servers have been written at the ESRF for a large variety of different devices. Devices vary from serial line devices to devices interfaced by field-bus to memory mapped VME cards or PC cards to entire data acquisition systems. The definition of a device depends very much on the user's requirements. In the simple case a device server can be used to hide the serial line protocol required to communicate with a device. For more complicated devices the device server can be used to hide the entire complexity of the device timing, configuration and acquisition cycle behind a set of high level commands.

This section is organized as follows:

Throughout this section examples of source code will be given in order to illustrate what is meant. Most examples have been taken from the StepperMotor class - a simulation of a stepper motor which illustrates how a typical device server for a stepper motor at the ESRF functions.

6.5.2 TANGO Device Server Guidelines

Intended audience: developers, ***Programming language:*** all

Contents:

Guidelines

Intended audience: developers, ***Programming language:*** all

This chapter describes Guidelines for developing Device Servers. The purpose of this document is not to rewrite the Tango documentation but to propose the community an interpretation of Tango device development. The Tango Device Server Model is flexible and permits different interpretations of how to implement Device Servers. However there is a right way of using Tango to implement device servers. This chapter documents the best practices from experienced developers (some of them thee original developers of Tango) for device development.

Other ways of using Tango which do not follow these guidelines are possible and can be useful but they might run into difficulties because Tango was not designed to be use that way. All developers should start off by first reading these guidelines and then deciding if they want to ignore them or not. We strongly recommend you stick to them to make your Device Classes easier to share and your Tango control system as efficient as possible.

To this aim the document is divided in 3 main chapters:

1. Tango concepts
2. DeviceServers design consideration
3. Implementation good practices

About this document

The document has been initiated within the collaborative framework between SOLEIL and MAX-IV to define common software quality rules for shared software between these 2 institutes. It has since been adopted by the Tango community and is maintained for and by the community.

The objectives are therefore to enhance the general software quality of Device Servers developed by the various sites using Tango. This will also facilitate the reusability of developments between sites by allowing finding "reliable off-the-shelves" Tango servers in public repositories.

Last but not least, this document can be freely distributed (under the Creative Commons license) to subcontractors, students, etc…. Our hope is (*as all writers*) to have as many readers as possible!!

Note: Throughout the rest of the document, the issued recommendations are specified as below:

The recommendation is to …

Note: **Important note:** The content of this document is generally independent of the programming language used. However, there are some "C++ oriented" recommendations. For Java and Python refer to the relevant documentation for language specific issues. In the future we hope to add guidelines for Java and Python too.

The present document refers to the Tango 8 or higher versions features.

Tango Concepts

The following explanations are from the *Tango Device Server Model*.

Tango Control system

The Tango control system is an abstract concept which represents a set of "microservices" based on a common technology: Tango. Tango is itself a control/command oriented specialization of CORBA/ZMQ. CORBA supports the concept of software bus running over a network interconnected machines. It provides transparent access to any software object (or microservice) connected to the bus and abstracts the notions of programming language (C++, Java, Python…) and operating systems (Linux, Windows…) via a binary network protocol (based on CORBA and ZMQ).

Tango hides the complexity of the underlying protocols to the programmer, while adding specific control system features (alarms, events, logging, data archiving…).

Device concept

The *device* is the core concept of Tango. This concept can be directly linked to the notion of microservice: **1 device = 1 microservice**

A device can represent:

- An equipment (eg: a power supply),
- A set of equipments (eg: a set of 4 motors driven by the same controller),
- A set of software functions (eg: image processing),
- A group of devices representing a subsystem

The Tango Device allows making abstraction of the equipment's nature: the device hides the implementation specific details from the user who does not need to care about communication protocols etc. and provides the user with a model which speaks their languages e.g. physical or engineering parameters.

Hierarchy

A Tango control system can be hierarchically organized.

At the lower level, we find elementary devices which are associated with equipments.

- e.g.: a vacuum pump, a motor, an I/O card

At higher levels, the devices are « logical ». These devices, based on the lower-level devices, manage and represent a subset of the control system. This is usually a synthetic view of a set of equipments with a high-level steering (functions can perform sequences of actions on several basic devices).

For example, a high-level device achieves "complex" features. This device is usually bound to evolve regardless of the hardware. Therefore, it is necessary to separate and segregate responsibilities related to the logic functionality and those related to hardware interfaces.

It is possible to access any other device from every device at every level.

The following diagram illustrates the concept of hierarchy of devices:

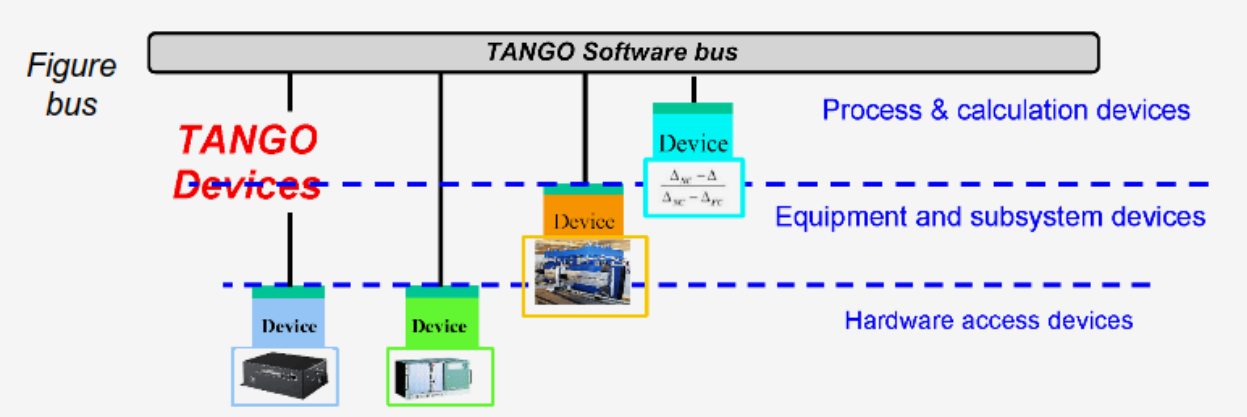


Fig. 2: The software bus view of devices

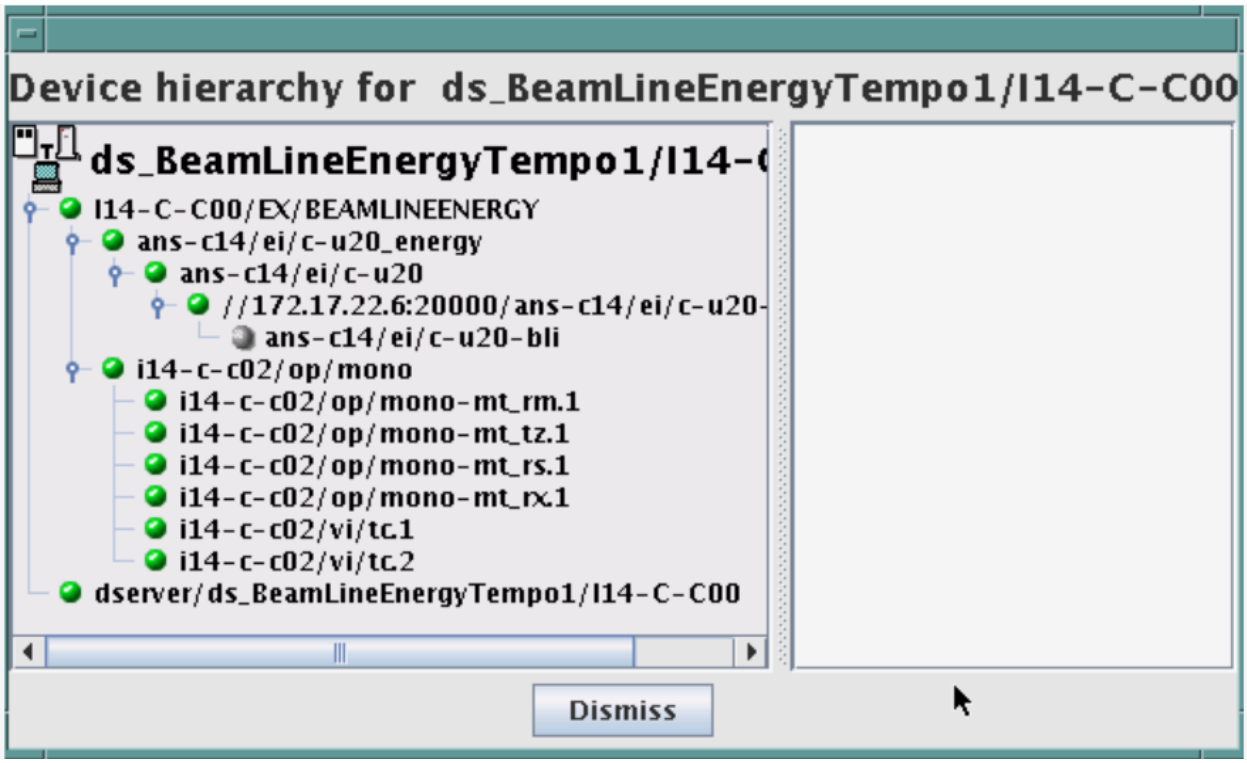


Fig. 3: Hierarchical view of devices

Communication paradigms

Tango offers three communication paradigm: synchronous, asynchronous and publish-subscribe calls.

In the synchronous and asynchronous paradigms the call is initiated by the client who contacts the server. The server handles the client's request and sends the answer to the client or throws an exception which the client catches. This paradigm involves two network calls to receive a single answer and requires the client to be active in initiating the request. The calls initiated by the client may be done by 2 mechanisms:

Note:

1. the **synchronous** mechanism where the client waits (and is blocked) for the server to send the answer or until the timeout is reached
2. the **asynchronous** mechanism where the clients send the request and immediately returns. It is not blocked. It is free to do whatever it has to do like updating a graphical user interface. The client has the choice to retrieve the server answer by checking if the reply is arrived by calling an API specific call or by requesting that a call-back method is executed when the client receives the server answer.

If the client needs to know a value every time it changes or at regular intervals then he is obliged to poll the server for an update in a value every time. This is not efficient in terms of network bandwidth nor in terms of client programming. For this the publish-subscribe events communication is more efficient.

Note:

3. the **publish-subscribe** communication paradigm is a more efficient and natural way of programming. In this paradigm the client registers his interest once in an event (value). An event can be a change in value, a regular

update at a fixed frequency or an archive event. After that the server informs the client every time an event has occurred. This paradigm avoids the client polling, frees it for doing other things, is fast and makes efficient use of the network.

Class, Device and Device Server

Reminders

Sometimes, there are misuses of language regarding the concepts of: device, device server and Tango class.

- **DeviceClass** class: a class defining the interface and state machine.
- **Device** class: a class implementing the device control.
- **Device**: An instance of a Device class giving access to the services of the DeviceClass class.
- **Device Server**: process in which one or more Tango classes are executed (*device server*).

Note: DeviceClass class is only used in C++ device classes

Hint: These four concepts are closely related, and they express very important concepts of Tango. Take time to clearly understand them!

The diagrams below illustrate these concepts:

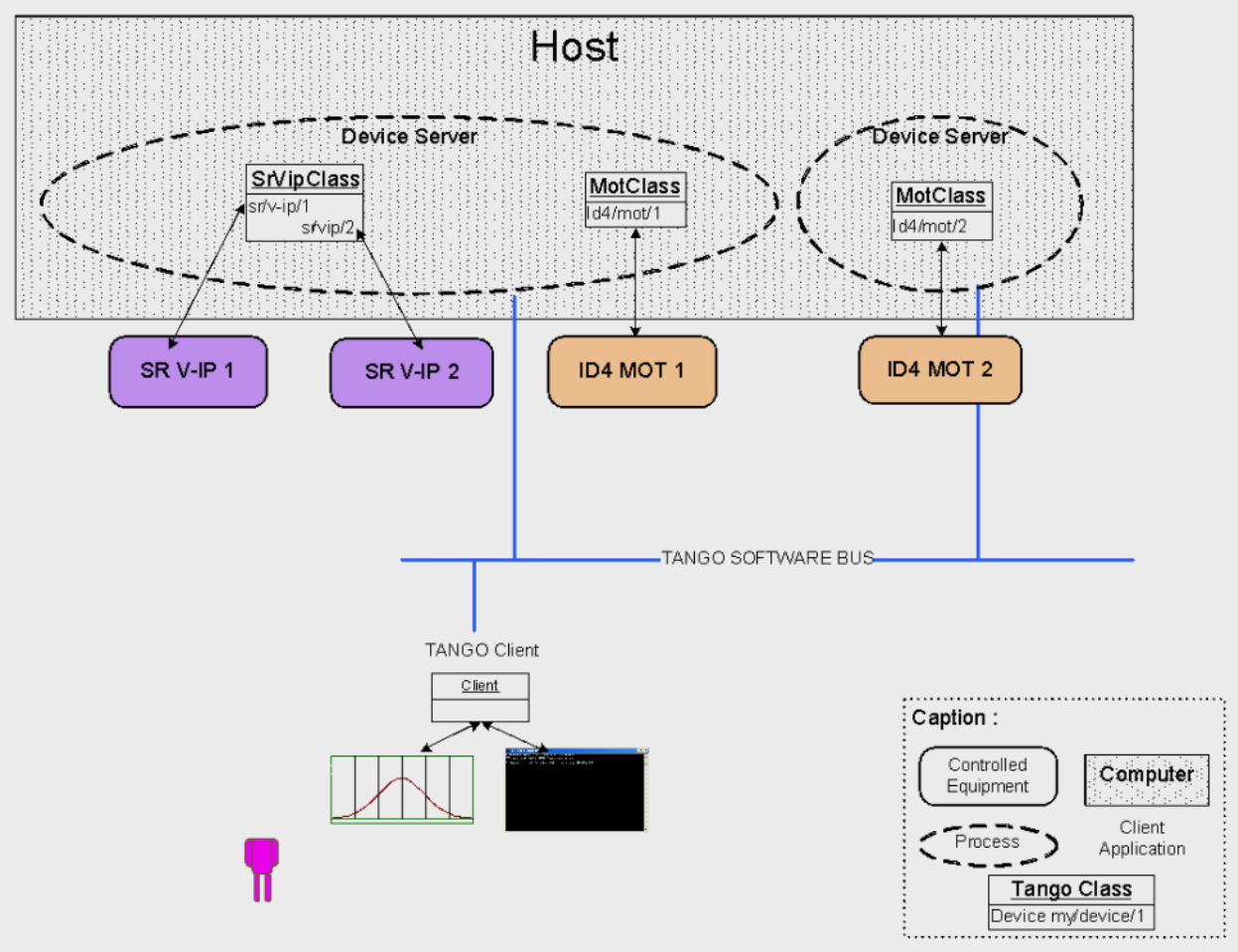


Fig. 4: Tango Deployment

A Device Server can host several Device classes, each class can be instantiated one or more times within the same device server. There are no specific rules regarding the maximum number of classes or the maximum number of instances operating within a single Device Server.

In particular cases, due to limitations imposed by the hardware or software interface, it is not always possible to run several instances of a Device class within the same Device Server:

- **Case of a DLL's use:** some DLLs can't be used by two threads of the same process.

In other cases, it is useful to have multiple devices running in the same Device Server:

- ***Case of motors:** a single axis controller for 4 motors.

Device

Note: This is the basic entity of the control system. In the Tango world, everything is a *device*.

A Tango Device must be "self-consistent". In case it represents a subset of the control system, it must enable the access to all the associated features (unless otherwise specified). The limit of its "responsibilities", meaning "separation of concerns", is clearly defined: 1 Device = 1 microservice = 1 element of the system. The analogy with object-oriented programming is straightforward.

A Device is a **microservice** made available to any number of unspecified clients. Its implementation and/or behaviour must not make **assumptions about the nature and the number of its potential clients**. In all cases, reactivity must be ensured (i.e. the response time of the device, must be minimized).

A Device has an interface composed of *commands* and *attributes*, which provides the service of the device. It also has *properties*, stored in the relational database, which are generally used as configuration settings. These concepts are explained later in this document.

Device attributes

Purpose of an attribute

Attributes correspond to physical quantities carried by the device. Any value that you want available on the Tango bus is an attribute. For example:

- A device associated with a motor **has** a `position` attribute expressed in mm.
- A device associated with a thermocouple **has** a `temperature` attribute expressed in Celsius (or any another suitable unit).

Note: The main purpose of an attribute is to replace getters and setters.

- For example: the position of a motor will be obtained by reading the associated attribute (position) and not by running a command like *get_position.*
- The data associated with the Tango attributes are the only values that can be archived. The Tango *archiving system* (HDB/TDB) doesn't have any functions to archive the result of a command. Similarly, some mechanisms to store the experimental data (such as those implemented by the DataRecorder of SOLEIL) are only based on attributes.

Attribute Properties

A Tango attribute has a group of settings that describe it.

These configuration parameters are called AttributeProperties. They can be considered as meta-data to enhance the semantic and describe the data. They can be used by GUI clients for configuring their viewers in the best manner and displaying extra information.

Those Attribute properties describe the attribute data and define some of its behaviour such as alarm limits, units etc…

The first set of *Attribute Properties* are static metadata. They describe the kind of data carried by the Tango Attribute. The static metadata includes properties such as the name, the type, the dimension, if the attribute is writable or not. These data are hardcoded, defined for the whole life of the attribute and cannot be modified.

The second set of *Attribute Properties*, are dynamic. They describe more precisely the meaning of the data and some behaviour. They are used by GUI viewers to configure themselves. They can be modified at run time.

All these metadata are hosted in the class itself and can be set by the programmer or by a configuration in the Tango database.

Static attribute Properties

- *name*: the attribute name
  - Type: string e.g : OutCurrent, InCurrent...
- *data_type*: the attribute data type
  - Identifier of the Tango numeric type associated to the attribute: *DevBoolean, DevUChar, Dev[U]Short, Dev[U]Long, Dev[U]Long64, DevFloat, DevDouble, DevString, DevEncoded*
  - Note: *Tango::DevEncoded* is the Tango type that encapsulates client data.
- *data_format*: describes the dimension of the data.
  - Type: scalar (value), spectrum (1D array), image (2D array)
- *writable*: defines 4 possible types of access. In practical, we can say that only 2 are really useful and answer to practically all the cases.
  - READ, The attribute can only be read (e.g. a temperature)
  - WRITE, The attribute can only be written (to be used only in very specific cases. the READ_WRITE is generally more suitable for real cases)
  - READ_WRITE, The attribute can be written and read (the most common case) e.g. The current of a powersupply, The position of an axis...
  - READ_WITH_WRITE (deprecated, do not use)
- *max_dim_x*: this property is valid only for data_format spectrum or image. It gives the maximum number of element in the dimension X. e.g. the max length of a spectrum or the maximum number of rows of an image. This property is used to reserve memory space to host the data. Nothing prevent to have a real length much shorter that this maximum.
  - e.g. 0 for a scalar, n for a spectrum of max n elements, n for an image of max n rows
- *max_dim_y*: this property is valid only for data_format image. It gives the maximum number of element in the dimension Y. e.g. the maximum number of columns of an image. This property is used to reserve memory space to host the data. Nothing prevent to have a real length much shorter that this maximum.
  - 0 for a scalar or a spectrum, n for an image of max n columns
- *display_level*: enables to hide the attribute regarding the client mode (expert or not)
  - Tango::OPERATOR or Tango::EXPERT

Warning: *writable_attr_name*: **deprecated since version 8, do not use anymore**

Modifiable attribute properties

These properties carries out information regarding the display of a value (they are editable while the device is running). Those properties enhance the meaning of the attribute and should as much as possible be defined by the device server programmer as default value when known. For instance, in the general case, the programmer knows the unit of the data and is able to describe it. Feeling the attribute property at the development stage will allow all generic clients to display the data in the best manner

- *description*: describes the attribute
  - Type: string e.g. "The powersupply output current"
- *label*: label used on the GUIs
  - Type: string e.g. "Output Current", "Input Current"
- *unit*: attribute unit to be displayed in the client viewer
  - Type: string (eg "mA", "mm"...)
- *standard_unit*: conversion factor to get attribute value into S.I (M.K.S.A)_unit. Be careful this information is intended to be used ONLY by the client (.e.g ATKPanel uses it, but jive->test device does not)
  - Type: string interpreted as a floating point value E.g. If the device attribute gives the current in mA, we have to divide by 1000 to obtain it in Amp. Then we will set this property to 1E-03
- *display_unit*: used by the GUIs to display the attribute into a unit more appropriate for the user. Be careful this information is intended to be used ONLY by the client (e.g ATKPanel uses it, but JiveTest device does not).
  - Type: string interpreted as a floating point value If the device attribute gives a current in mA. If we want to display it in microA, then we have to multiply by 1000 to obtain it in microAmp. Then we will set this property to 1000.0.
- *format*: specifies how a numeric attribute value should be presented
  - Type: string : e.g. « %6.3f »
  - Note: we use a "printf" like syntax
- *min_value* and *max_value*: minimum and maximum allowable value. These properties are automatically checked at each execution of a write attribute. If the value requested is not between the min_value and the max_value, an exception will be returned to the client.
  - Type: string interpreted as a floating point value (e.g. 10.1, 1E01, 0.12.)
  - Note: these properties are valid only for writable attributes

Attributes properties for ALARM configuration

Tango provides an automatic way of defining alarms. An alarm condition will switch the attribute quality factor to alarm and the device state will automatically switched to ALARM in certain conditions. Four properties are available for alarm purpose.

- *min_alarm* and *max_alarm*: Define the range outside which the attribute is considered in alarm. If the value of the attribute is > max_alarm or < min_alarm, then the attribute quality factor will be switched to ALARM.
- *Delta_val* and *delta_t*: (*could also be called maximum noise and time constant*) Valid for a writeable attribute. Define a maximum difference between the set_value and the read_value of an attribute after a standard time.

e.g. the voltage of a powersupply is set via a DAC and read via an ADC convertor. Both values are different due to various factors such as internal resistor or noise on the ADC. Furthermore when setting a voltage, the powersupply may need a certain time to establish its output voltage. The *delta_val* property allows to define the limit of the acceptable difference between set and read values (noise threshold) and *delta_t* defines the time the device needs to establish the voltage after the writing of the setpoint (time constant). When writing a new value of the attribute, if the read value is still not close enough from the set value after the time constant, the attribute quality factor will be set to ALARM.

If these properties are not set, nothing is done. As soon as one of these properties is set, then the attribute quality factor is automatically calculated at each read and is taken into account by the default State attribute method. Device_Impl.dev_state(); The programmer should be aware of possible effect of these mechanisms in the response time of the State method. (Refer to chapter 1.14 of the present guide).

Warning: The behaviour described above is only correct in the case the device's method *Tango::Device_[X]Impl::dev_state()* is executed. In case of overwrite of the dev_state() in the device code, it is recommended to finish the method by calling DeviceImpl::dev_state();

Warning: **min_warning** *and* **max_warning** : lower and upper bound for WARNING (deprecated since version 8)

Attributes properties related to Events configuration

These settings are used for tuning the events related to the attribute.

- *Rel_change:* relative change in the value in percent
- *Abs_change*: absolute change in the value in the standard unit.
- *Period*: period between two consecutive events
- *Archive_rel_change*: relative change in the value
- *Archive_abs_change*: absolute change in the value
- *Archive_period*: period between two consecutives events.

Particular case of a memorized attribute

Note: Memorised attributes are only possible with an attribute with WRITE or READ_WRITE mode and SCALAR type

A memorized attribute can store its last written value in the database (i.e. the last setpoint received by the device for this attribute can optionally persist into the Tango database).

The stored value will be reloaded into the set value associated with this attribute at device start-up and (optionally) upon each execution of the "Init" command. The Tango code generator (Pogo) provides the interface allowing the developer to select the expected behaviour.

Note: **BE CAREFUL:** this mechanism has the following **behaviour**:

- The writing of the memorized attributes is carried out after the function `init_device`, executed by the Tango layer, and not by the Tango DeviceServer code. In case an error occurs during the `init_device` it cannot be caught by the Tango DeviceServer programmer.
- If in the `init_device` method an error occurs that causes a change of state in which the writing of an attribute is impossible, this error will prohibit the restoration of the memorized value of the attribute.
- The order of reloading is deterministic but complex (*order of ClassFactory then device definition in database then attribute definition in Pogo*). Therefore relying on this order might have some side effects particularly in case attributes are modified through Pogo when attributes values are linked (*eg: sampling frequency and number of samples*).

Warning: Performance issues may happen in case the setpoint is written at high frequency, the static Tango database is requested on each write of the memorized attribute. Since Tango 9 the database has been optimised for memorised attributes and it should be possible to update memorised attributes at 10 Hz without taking a performance hit.

Tip: If this standard Tango behaviour for reloading memorized values doesn't fit your need, we recommend to code the reloading of attribute values yourself. This is especially true for fast (> 10 Hz) feedback loops which can trigger the writing of attributes at a high frequency.

Device commands

A *command* is associated with an action. *On, Off, Start, Stop* are commons examples.

A Tango command has, optionally, ONE input argument and ONE output argument.

The different types of data compatible for input and output are:

- void, boolean, short, long, long64, float, double, string, unsigned short, unsigned long, unsigned long64
- *1D array of the followings types :* char, short, long, long64, float, double, unsigned short, unsigned long, unsigned long64, string
- State: enumeration, representing the different states described in the section on *Device State*.
- 2 particular types: longstringarray and doublestringarray. These are structures including one array of long/double and one array of string.

The list of data types is fixed. If you need to add your own data type then use the DevEncoded type and encode your own data type. Or you can use the DevPipe communication channel (avaliable since Tango 9).

For each command to implement, it is essential to generate exceptions depending on possible errors. The error handling is described more in details below.

Device State

State transitions

Note: Every Tango device has a state implemented by *finite state machine*.

The device state is a key element in its integration into the control system. Therefore, **you should be very careful in the management of state transitions** in the device implementation.

The device state must, at any time, reflect the internal state of the system it represents. The state should represent any change made by a client's request.

This is crucial information. Indeed, the "clients" will primarily, or only, use this information to determine the internal state of a system.

The available states are limited to:

- `ON`, `OFF`, `CLOSE`, `OPEN`, `INSERT`, `EXTRACT`, `MOVING`, `STANDBY`, `FAULT`, `INIT`, `RUNNING`, `ALARM`, `DISABLE`, `UNKNOWN`

The main thing is to ensure a predictable behaviour of the device regarding the state transitions.

For example:

- Consider the case of a motor system. The client knows the motor state (*`STANDBY`, `MOVING`, `FAULT`,*)* with a *polling* mechanism (periodic reading of the state attribute of the motor – instead of using the Tango event system).

  In such cases, this can easily lead to inconsistent behaviour due to inappropriate management of the state.

  A typical example is to launch an axis movement through the writing of the position attribute then the client is pending on the `MOVING` state (the motor is supposed to make a transition *STANDBY MOVING*). Such a method will only work if the writing of the position attribute switches the device state to `MOVING` *before* the return of the writing request of the position attribute. Otherwise, the client can read (non-zero probability) the `STANDBY` state, and interpret it as "movement ended" while this one had not even started!

  This behaviour is described in figure 4 below.

Tip: The developer has to guarantee the clients the same behaviour regardless the type of state monitoring (polling or events). This relates to the above rule: **Do not make assumptions about the nature of the clients!**

The state transitions and the "associated guarantees" must be documented. In the previous example, rereading the STANDBY state after performing any movement must ensure that the required movement is completed (and not that it has not yet been started!!).

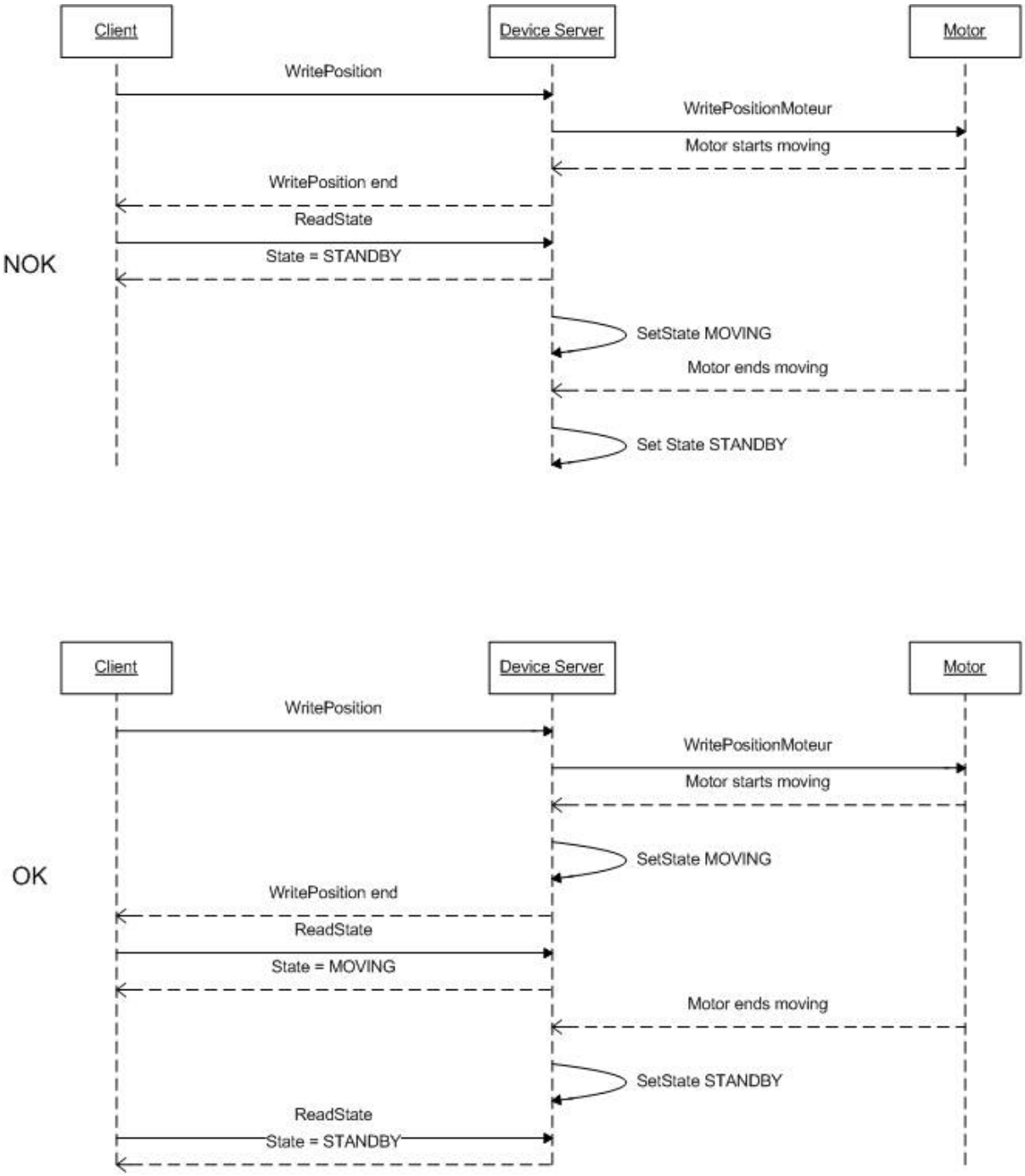


Fig. 5: Example of State transitions

Properties

Concepts

By default Tango is based on a relational database (MySQL) to store configuration information for devices namely the *properties*.

The properties are used to configure a device without changing the Tango class code. Taking an axis controller as example, the controller must be configured for the motor mechanics according to the characteristics of the actuator and the movements to achieve.

Configuration properties are available on different levels:

1. **The device level:** These are properties to configure the device itself and its attributes. The device properties configure the device with the necessary set-up information during initialisation. Attribute properties are used to configure alarms or specify the way the attribute value is displayed to the user (Label, Format, Unit...).
2. **The class level:** Device or attribute properties configured at the class level are valid for all instances of a class. A property defined on the class level will be overwritten by a property of the same name on the device level.
3. **Free properties:** These are configuration values which are not attached to any device or class and can be freely used by programmers.

Class level and device level properties are automatically loaded during device initialisation when starting-up a device server or calling the "Init" command. The reading and writing of free properties must be handled by the programmer.

Configuration properties can have the following data types:

- boolean, short, long, float, double, unsigned short, unsigned long, string
- array of: short, long, float, double, string

On top of those basic concepts, device and class level properties can be initialised with default values which are entered, for example, with *Pogo* at the interface creation time. Default values are stored in the device server code and are overwritten when another value is found in the configuration database.

It is necessary to assign a default value for every property. This value will be used when the property is not defined in the Tango database. If a default value for a device property does not make sense, the property should be declared as mandatory. A mandatory property has to have a value configured in the Tango database. If no value is configured, the device initialisation will stop with an exception on the missing property value.

Device property vs memorized attributes

In some cases, you could be tempted to use a property for a memorized attribute and vice-versa. It is important to distinguish the function of each, and use them wisely.

- The use of a property must be limited to configuration data which value doesn't change at runtime (the IP address of equipment for example).
- The memorized attributes are reserved for physical quantities subject to change at runtime (*attribute read/write*) for which you want to retain (store) the value from one execution to the other.

  e.g. speed or acceleration on a motor.

Tip: In the case you want to manually manage the memorization of the attribute set points, you should use an attribute property called *__value* (as natively done by Tango).

How to configure a new device

To set-up a new device you need to know about all the device properties and their values which must be configured to make the device work. You need to have a description on the property which should indicate clearly its use. Also you need to know about a specified default value.

When creating the device interface with Pogo a description and a default value can be entered for every device property. This information is used by the device installation wizard (available with Jive) to guide you through the configuration.

When creating a new server start the wizard from the Tools menu -> Server Wizard. It allows you to create a new device and to initialise it property by property. For every property the description is displayed and the default value can be viewed. To use the wizard on an already existing device you can right click on the device and choose Device Wizard. You will be guided again through all the properties of the device. At the end the device can be re-started when necessary. Because the wizard is part of Jive, you can test the device configuration immediately.

Tango Device Design

Elements of general design

Reusability

In a Tango control system, each device is a software component potentially reusable.

It is necessary to:

- Systematically evaluate prior the coding of a device, the possibility of reusing a device available in the code repositories (Tango community, local repository), in order to avoid several implementations of the same equipment.
- Design the device as reusable/extensible as possible because it may interest the others developers in the community.

As such, the device must be:

- Configurable: (e.g.: no port number “hard coded”, but use of a parameter via a property),
- Self-supporting: the device must be usable outside the private programming environment (eg: all the necessary elements to use the device (compile, link) must be provided to the community). The use of the GPL should be considered, and the use of proprietary libraries should be avoided if possible
- Portable: the device code must be (as much as possible) independent of the target platform unless it depends on platform specific drivers,
- Documentation in English

Generic interface programming

The device must be as generic as possible which means the definition of its interface should

- Reflect the service rather its underlying implementation. For example, a command named "WriteRead" reflects the communication service of a bus (type: message exchange), while a command named `NI488_Send` reflects a specific implementation of the supplier.
- Show the general characteristics (attributes and commands) of a common type of equipment that it represents. For example, a command `On` reflects the action of powering on a PowerSupply , while a command named `BruckerPSON` reflects a specific implementation which must be avoided.

The device interface must be service oriented, and not implementation oriented.

Abstract interfaces

Singleton device

Tango allows a device server to host several devices which are instantiations of the same Tango class.

However, in particular case some technical constraints may forbid it. In this case, the Device Server programmer must anticipate it in the device design phase (add for example a static variable counting device instances or other) to detect this misconfiguration. For example, it can authorize the creation of a second instance (within the meaning of the device creation) but systematically put the state to FAULT (in the method `init_device`) and indicate the problem in the Status.

In the case where technical constraints prohibit the deployment of multiple instances of a Tango device within the same device server, the developer has to ensure that only one instance can be created and inform the user with a clear message in case more than one device is configured in the database.

Device states

When designing the device, you should clearly define the state machine that will reflect the different states in which the device can be, and also the associated transitions.

The state machine must follow these rules:

- At any time, the device state must reflect the internal state of the system it represents.
- The state should represent any change made by a client's request.
- The device behaviour is specified and documented.

Device interface definition

The first step in designing a device is to define the commands and the attributes via Pogo (use **Pogo** to define the Tango interface).

Except in (very) particular cases, always use an attribute to expose the data produced by the device. The command concept exists (see *Device Commands*) but its use as an attribute substitute is prohibited. Example: a motor must be moved writing its associated 'position' attribute instead of using a 'GotoPosition' command.

The choice will be made following these rules:

- Attribute: for all values to be presented to the "client". **It is imperative to use the attributes and to not use Tango commands that would act like a get/set couple.**

- Command: for every action, of void-void type in most cases.

Any deviation from these rules must be justified in the description of the attribute or command particular case.

Service availability

From the operator perspective, the "**response time**" or "**reactivity**" (i.e. the device is always responsive) is **the** reference metric to describe the performance of a device. Ideally, the device implementation must ensure the service availability regardless of the external client load or the internal load. For the end user, it is always very unpleasant to suffer a Tango timeout and receive an exception instead of the expected response.

The response time of the device should be minimised and in any case lower than the default Tango timeout of 3 seconds.

If the action to be performed takes longer than that, execution should be done asynchronously in the Tango class: its progress being reported in the state/status.

Several technical solutions are available to the device developer to ensure service availability:

- Use the Tango polling mechanism,
- Use a threading mechanism, managed by the developer.

Tango polling mechanism

Polling interest

The polling mechanism is detailed in the Tango documentation *Device Polling*.

Tango implements a mechanism called *polling* which alleviates the problem of equipment response time (which is usually the weak point in terms of performance). The response time of a GPIB link or a RS-232 link is usually one to two orders of magnitude higher than the performance of the Tango code executed by a client request.

Polling limitations

From the perspective of the device activity, the polling is in direct competition with client requests. The client load is therefore competing with the polling activity.

This means that polling activity has to be tuned in order to keep some free time for the device to answer client requests. Do not try to poll a device object with a polling period of let say 200 mS if the object access time is 300 mS (*even if Tango implements some algorithm to minimize the bad behavior of such badly tuned polling*).

For polled Tango device objects (attribute or command), client reading does not generate any activity on the device whatever the client number. The data are returned from the so-called polling buffer instead of coming from the device itself. Therefore, an obvious rule is to poll the key device object (state attribute, pressure attribute for a vacuum valve...)

The recommendation for device polling tuning is to keep the device free 40% of time.

Let's take an example: for a power supply device, you want to poll the device state and its current attribute which for such a device are the device key objects.

- State access needs 100 mS while current attribute reading needs 50 mS.
- Because, you want to poll these two objects, time required on the device by the polling mechanism will be 150 mS (100 + 50).
- In order to keep the 40% ratio, tune the polling period for this device to 250 mS.

- The device is then occupied by the polling mechanism during 150 mS (60 %) but free for other client activity during 100 mS (40 %).

Device polling is easily tunable at run time using Jive and/or Astor Tango tools.

Threading mechanism

Threading is a possible solution for the load problem: a thread (managed by the device developer) supports communication with the material (*polling* or other) and the data obtained are put in the "cache". You can now produce the "last known value" to the client at any time and optimize the response time. This approach, however, has a limit where it is necessary to reread the hardware to assure clients that the returned value is the system "current state".

For a C++ device, the implementation of a threading mechanism can be done via the *DeviceTask* class from the *Yat4Tango library*. This class owns a thread associated with a FIFO message list. Processing messages can be synchronous or asynchronous.

See the complete example in the appendix for the implementation details.

When the design of the Tango class requires threading:

- in case of simple thread usage, in C++ the recommendation is to use a C++11 thread
- In case of acquisition thread with messages exchange in C++ the recommendation is to use Yat4Tango::DeviceTask class.

Tango device implementation

General rules

Language

The Tango community is international and the developments could be shared with the community, so it is recommended to use English for documenting a device development.

English will be used for:

- The interfaces definition (attributes and commands),
- The device documentation (online help for command usage and attributes description),
- The comments inserted in the code by the developer,
- The error messages,
- The name of variables and internal methods added by the developer.

The choice of the language used for the user's documentation of the device server ("Device Server User's Guide") is left free, to focus on the editorial quality. In the case of a joint development with another institute, English will be used.

Types

The types used for the device interface definition are Tango types (`Tango::DevDouble`, `Tango::DevFloat` ...). These types are presented by Pogo and are not modifiable.

The types used by the developer in its own code are left free to choose, as long as they are not platform specific. Standard types of the language used (Boolean, int, double ...), Tango types or types from a common library (Yat, Yat4Tango for C++) can potentially be used.

Direct conversions from the C++ type long to `Tango::DevLong` are only supported on 32-bit platforms and should be avoided.

Generated code

The automatically generated code by Pogo must not be modified by the developer.

The developer must include its own code in the "PROTECTED REGION" specified parts.

Device interface

Naming rules

Having homogeneous conventions for naming attributes, commands and properties is a good way to promote Device-Servers reuse inside the Tango collaboration.

In fact it makes the development done by another institute easier to understand and integrate in another Control System.

Class name

The Tango class name is obtained by concatenating the fields that compose it – each field beginning with a capital letter:

Eg : `MyDeviceClass`

Device attributes

The device command and attributes names must be explicit and should enable to quickly understand the nature of the attribute or the command.

- Eg: for a power supply, you will have an attribute *`outputCurrent`* (not OC1) or a command *`ActivateOutput1`* (not ActO1).

The nomenclature recommendations are in the section *Naming Rules*.

The attribute naming recommendations are:

- Name composed of at least two characters,
- Only alphanumeric characters are allowed (no underscore, no dashes),
- Start with a **lowercase** letter,
- In case of a composite name, each sub-words must be capitalized (except the first letter),

- Prohibit any use of vague terms (eg: readValue).

Device Commands

The recommendations are the same as those proposed for an attribute, except for the first letter of the name.

The command naming recommendations are:

- Name composed of at least two characters,
- Only alphanumeric characters are allowed (no underscore, no dashes),
- Start with a **uppercase** letter,
- In case of a composite name, each sub-words must be capitalized,
- Prohibit any use of vague terms (eg: Control).

Device properties

The recommendations are the same as those proposed for a command.

The property naming recommendations are:

- Name composed of at least two characters,
- Only alphanumeric characters are allowed (no underscore, no dashes),
- Start with a **uppercase** letter,
- In case of a composite name, each sub-words must be capitalized,
- Prohibit any use of vague terms (eg: Prop1).

Device attributes nomenclature

It is a good practice that a particular signal type is always named in a similar way in various DeviceServers.

For example the intensity of a current should always be name *intensity* (and not "**intens**", "**intensity**", "**current**","**I**" depending on the DeviceServers).

This allow the user to quickly make the link between the software information and the physical sensor and reciprocally.

Data types choice

Always use data types consistent with the underlying information

- Unsigned integer must be used for the physical quantities that are suitable.
  - Eg: A number of samples numSamples, where negative values have no meaning, will be a Tango::DevULong (unsigned integer 32 bits) and not a Tango::DevLong (signed integer 32 bits).
  - Similarly, in such a case, the use of a floating point number is to be prohibited, non-integer values having no meaning.
- This rule is applicable to input/output arguments of commands.

Interface level choice

The choice between the *Expert* or the *Operator* level for an interface must be thoughtful.

Only necessary and sufficient commands for a nominal control of the equipment must be accessible to the *Operator* level. The commands for fine control of the equipment (eg: metrology, maintenance, unit test) must only be accessible to the *Expert* level.

Pogo use

Device generation

The use of Pogo is mandatory for creating or modifying the device interface.

Tango is constantly evolving, this tool will support all or part of the porting, associated to the kernel and their consequences on the IDL interface.

In addition, it simplifies maintenance / development operations.

Every command, attribute, property or device state must be fully documented; this documentation is done via the Pogo tool.

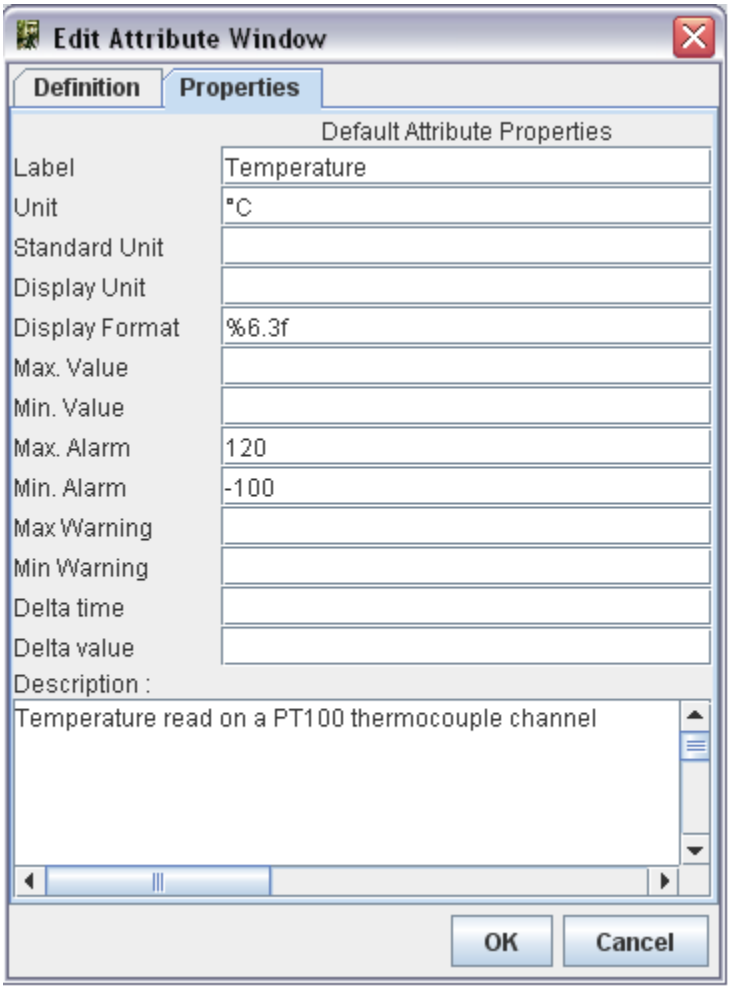
Specifically, when creating an attribute with Pogo, the entire configuration of the attribute must be fully filled in by the developer (maximum possible) to avoid ambiguities.

Similarly, the states and their transitions must be described with precision and clarity.

In fact:

- In operation, this documentation will be the reference for understanding the device behaviour. Remember that the operator will have this information with the generic tools (like *Test Device* from **`Jive`**).
- The html documentations generated by **`Pogo`** can also be accessed from a local server (peculiar to the institute).
- Consider also filling in the alarm values.
  - Eg: set the alarm values according to the specifications of a power supply, ie, 0V-24V for the voltage, or 0A-3A for the output current.

    Example for a temperature reading:



Attributes generation in C++

In C++, Pogo automatically generates **pointers** to the data associated with the attributes values (ie a pointer is generated for the read part). The use of these pointers is not mandatory. The developer is free to use his own data structure in the attribute value affectation.

Internal device implementation

Separation between the Tango interface and the internal system function

Don't forget that the Tango interface is only a means to insert a microservice in a control system. Therefore, it is necessary to think the device internal design like any other application and just add the Tango as an interface on top of it.

As a rule of thumb if the code implemented within the Pogo markers is too long, a good practice is to move it to another class. Then Pogo generated methods will be only a few lines of code long.

In practice, it is necessary to avoid mixing the generated code by Pogo and the developer's one.

The Tango sub-class inherited from `Tango::DeviceImpl[_X]` instantiates a class derived from the model object implementing the system, and ensure the replacement between the external requests (clients) and the implementation

class(es).

In the choice of data structures, we are talking about those of the developer's object model, we will consider the technical constraints imposed by Tango and/or the underlying layers (CORBA/ZMQ). The idea here is to avoid copy and/or reorganization of the data when transferred to the client. For this, the developer needs to know/master the underlying memory management mechanism (especially in C++). The Tango documentation contains a dedicated chapter "*Exchanging data between client and server*".

Details on method for accessing the hardware: always_executed_hook versus read_attr_hardware

It is essential to master the concepts implemented by these two methods (common methods for all Tango devices).

It is also necessary to clearly identify, in the design phase, the possible consequences of implementing these two methods on the device behaviour (remember that they are initially just empty shells generated by Pogo).

- `Always_executed_hook()` method is called before each command execution or each reading/writing of an attribute (*but it is called only once when reading several attributes: see calling sequence below*)
- `Read_attr_hardware()` is called before each reading of attribute(s)(*but it is called only once when reading several attributes: see calling sequence below)*. This method aims to optimize (minimize) the equipment access in case of simultaneous reading of multiple attributes in the same request.

Reminder about the calling sequence of these methods:

- *Command execution*
  - 1 – `always_executed_hook()`
  - 2 – `is_MyCmd_allowed()`
  - 3 – `MyCmd()`
- *Attribute reading*
  - 1 – `always_executed_hook()`
  - 2 – `read_attr_hardware()`
  - 3 – `is_MyAttr_allowed()`
  - 4 – `read_MyAttr()`
- *Attribute writing*
  - 1 – `always_executed_hook()`
  - 2 – `is_MyAttr_allowed()`
  - 3 – `write_MyAttr()`
- *Attributes reading*
  - 1 – `always_executed_hook()`
  - 2 – `read_attr_hardware()`
  - 3 – `is_MyAttr_allowed()`
  - 4 – `read_MyAttr()`
- *Attributes writing*
  - 1 – `always_executed_hook()`
  - 2 – `is_MyAttr_allowed()`

- 3 – `write_MyAttr()`

When reading the sequence above, we understand why the mastery of these concepts is important. Particularly, having "slow code" in the `MyDevice::always_executed_hook` method can have serious consequences on the device performance.

Warning: There is no obligation to use the `read_attr_hardware` method; it depends on the equipment to drive and its communication channel (Ethernet, GPIB, DLL). You can have a call to the equipment in the code of each attribute reading method.

Example: For an attribute "temperature", of READ type, we can insert the call to the equipment in the generated attribute reading method `read_Temperature` instead of `read_attr_hardware`.

Static database as persistent data storage

As noted above the Tango database can (in some cases) be used to ensure persistence of set values, to store the value as a property (of device or attribute).

However, this practice should be reserved for special cases that don't require writing at high frequency. An oversolicitation of the Tango database will penalize the entire control system.

It is therefore recommended to use a property for storage only for methods that are performed rarely, compared to other functions.

For example: storage of calibration operations results

In the general case, we recommend to:

- Use a property to store configuration data,
- Use a memorized attribute to store values changing during the execution,
- Use a memorized attribute to store values that you want to re-inject during a new execution of the device.

Device state management

States choice

In Tango, as already said, the state is seen as an enumerated type with a fix number of values. These states have an implicit default meaning and are not equivalent. Furthermore a color code is associated to each state and is used in the main GUI tools to have a unified manner of representing the state of equipment.

State	Colour	Meaning
ON	green	This state could have been called OK or OPERATIONAL. It means that the device is in its operational state. (E.g. the powersupply is giving its nominal current, the motor is ON and ready to move, the instrument is operating). This state is modified by the Attribute alarm checking of the DeviceImpl:dev_state method. i.e if the state is ON and one attribute has it's quality factor to ALARM, then the state is modified to ALARM
OFF	white	The device is in normal condition but is not active. e.g the powersupply main circuit breaker is open; the RF transmitter has no power etc…
CLOSE	white	Synonym of OFF state. Can be used when OFF is not adequate for the device e.g case of a valve, a door, a relay, a switch.
OPEN	green	Synonym of ON state. Can be used when ON is not adequate for the device e.g case of a valve, a door, a relay, a switch.
INSERT	white	Synonym of OFF state. Can be used when OFF is not adequate for the device. Case of insertable/extractable equipment, absorbers, etc… This state is here for compatibility reason we recommend to use OFF or CLOSE when possible.
EXTRACT	green	Synonym of ON state. Can be used when ON is not adequate for the device Case of insertable/extractable

Unless strictly specified, the developer is free to use the Tango state she considers appropriate to the situation, with all the subjectivity involved.

The only practice that ensures overall consistency is to use a limited number of Tango states, especially for a family of equipment.

It is recommended for an equipment of type motor, slit, monochromator and more generally for any equipment that can change his position, to use the "MOVING" state when the equipment is in "movement" toward his set point.

Semantics of non-nominal states

Although the developer is free to choose the device states, we must define a common error state for all the devices.

In general, any dysfunction is associated with the state *Tango::FAULT*.

The use of the *Tango::ALARM* state should be reserved for very special cases where it is necessary to define an intermediate state between normal operation and fault. Its use must be documented via Pogo in order to define the semantics.

In the case of a problem occurring at initialization, it is recommended to set the device state to *FAULT*.

For the `init_device` method, we recommend:

- If the initialization method is long, thread it.
- The device state INIT must be used only in the start-up of the device.

The device states changes when the init execution is over.

Semantics recommended for *FAULT* and *ALARM* states is as follows:

- *UNKNOWN* (grey): communication problem with the equipment or the "sub"-devices which prevents the device to really know his real state
- *FAULT* (red): A problem which prevents the normal functioning (including during the initialization). Getting out from a FAULT state is possible only by repairing the cause of the problem and/or executing a Reset command.
- *ALARM* (orange): the device is functional but one element is out of range (bad parameters but not preventing the functioning, limit switch of a motor). An attribute is out of range.

State machine management

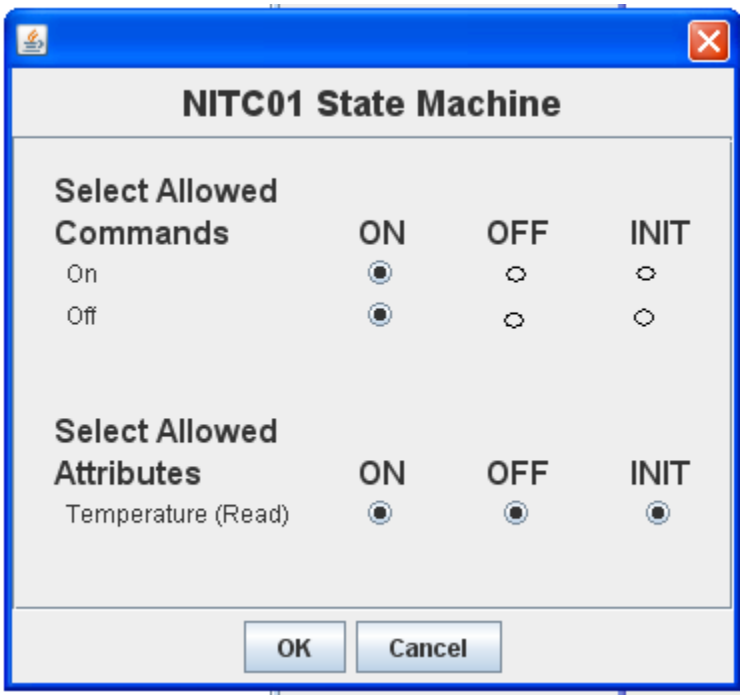
Pogo or developer code

Tango has a basic management of its state machine. `Is_allowed` methods filter the external request depending on the current device state. The developer must define the device behaviour (regarding its internal state) via Pogo.

By default, any request (reading, writing, or command execution) is authorized whatever the current device state is.

The example below illustrates two ways for the state machine management of a device (here NITC01) in C++:

- Managing the "On" command via **Pogo**
- Managing the reading of the attribute "temperature" directly in the code




```
void NITC01::read_attr_hardware(vector<long> &attr_list)
{
    DEBUG_STREAM << "NITC01::read_attr_hardware(vector<long> &attr_list) entering... " << endl;
    /*----- PROTECTED REGION ID(NITC01::read_attr_hardware) ENABLED START -----*/

    //  Add your own code

    int32 read = 0;
    float64 data[1000];
    int32 returnCode = -1;
    char errorString[1000];

    // Read only if state is ON or ALARM or WARNING
    if ((Tango::ON == get_state()) ||
        (Tango::ALARM == get_state()) ||
        (Tango::WARN == get_state()) )
    {
        // Read temperature attribute
        returnCode = DAQmxReadAnalogF64(_taskHandle, -1, 10.0, DAQmx_Val_GroupByScanNumber, data, 1000, &read, NULL);
```

However, the Pogo implementation is “basic”. If, for example, the execution of the `On` command on a power supply is prohibited when the current state is *`Tango::ON`*, then the Tango layer, generated by Pogo, will systematically trigger an exception to the client. From the operator perspective, this may surprise.

In such a case, it is recommended to authorize the command but to ignore it.

Particular case : FAULT state

The *Tango::FAULT* state shouldn't prohibit everything. The attributes and/or commands that are valid and/or allows the device to get out of the *Tango::FAULT* state must remain accessible.

For example, in some cases, when a device used several elementary devices, its state is a combination of the elementary devices states. If one of them is in "FAULT", we must be able to execute commands on others elementary devices, and, in all cases, have a command to get out of this state.

The transition to a *FAULT* state needs reflection and a clear definition of the device management in this state and the output conditions of this state.

Init and error acknowledgement

A common mistake is to associate the generic command `MyDevice::Init` to an acknowledgement mechanism for the current defect.

The execution of the `Init` command must be reserved to the device re-initialization (hardware reconnection after a reboot or reconfiguration following a property modification).

Any device that requires an acknowledgement mechanism must have a dedicated command (like `Reset` or `AcknowledgeError`).

Other implementations

You can also create a specific state machine, without using Tango types, in the interface class with the device. Thus, we use this state machine to determine the Tango state of the device. The aims here is to define an internal state machine (with a design pattern "state" for example) then do a mapping with the existing Tango states to determine the device state.

The developer also has the ability to override the *State* and *Status* methods in order to centralize, in a unique method, the management of the internal device state, which simplifies the update of this fundamental information.

Logging management

The importance of rigorous logging management

The introduction of logging in the device code enables easy development, bug research and the user understanding of the device operations.

The device developer must always use the facilities offered by the *Tango Logging Service* to produce "Run-time" messages, facilitating the understanding of the device operations. Implementations classes can inherit `Tango::LogAdaptater` to redirect the logs to the common service.

The rules to follow are:

- Logs to the console are prohibited. The developer must use the logging stream proposed by Tango (there is a stream for every logging level, the levels being inclusive in the order specified below). : *DEBUG_STREAM, INFO_STREAM, WARN_STREAM, ERROR_STREAM, FATAL_STREAM*
- It is important to use the right level of *logging* : on a higher level than DEBUG, the device should be a little wordy. Beyond the INFO level, it should produce only critical logs.

Recommendations of use:

- `DEBUG_STREAM` : developer information (route trace)

- `INFO_STREAM` : user information (measure, start/stop of a process)
- `WARN_STREAM` : warning (eg deprecated operation)
- `ERROR_STREAM` : general error
- `FATAL_STREAM` : fatal error, shutdown

It is important to use these *streams* early in the development. They allow an easier debugging.

You shouldn't have to modify the code to add traces.

- Eg: use a debug_stream level for the input parameters, the display of a conversion result, the return code from a DLL function…

It is also recommended to adopt a unified formalism for logs, for example:

- "<class_name>::<method_name>() - <text trace with parameter (eventually)>"

  Example of using different logs levels in C++:

```
Tango::DevLong NITCO1::on()
{
    Tango::DevLong argout;
    DEBUG_STREAM << "NITCO1::On()  - " << device_name << endl;
    /*----- PROTECTED REGION ID(NITCO1::on) ENABLED START -----*/

    //  Add your own code
    char errorString[1000];

    // Create acquisition task
    argout = DAQmxCreateTask("", &_taskHandle);

    if (argout != 0)
    {
        // Error on creating acquisition task
        set_state(Tango::FAULT);
        DAQmxGetErrorString(argout, errorString, 1000);
        set_status("Error on creating acquisition task - DAQmxCreateTask() : " + string(errorString));
        ERROR_STREAM << "NITCO1::on() - error on creating acquisition task for " << device_name << endl;
    }
    else
    {
        // Create acquisition channel
        // Use channelAlias property
        argout = DAQmxCreateAIThrmcplChan(_taskHandle, channelAlias.c_str(), "Test", 0.0, 100.0, DAQmx_Val_DegC, DAQmx_Val_J_Type_TC,
            DAQmx_Val_BuiltIn, 25.0, "");

        if (argout != 0)
        {
            // Error on creating acquisition channel
            set_state(Tango::FAULT);
            DAQmxGetErrorString(argout, errorString, 1000);
            set_status("Error on creating acquisition channel DAQmxCreateAIThrmcplChan() : " + string(errorString));
            ERROR_STREAM << "NITCO1::on() - error on creating acquisition channel for " << device_name << endl;
        }
        else
        {
            // Start OK
            set_state(Tango::ON);
            set_status("Acquisition in progress...");
            INFO_STREAM << "NITCO1::On() - Start OK for " << device_name << endl;
        }
    }
```

It is also possible to redirect the stream to a file (via Jive). This can be useful in the case of "random" bugs, for which a long log is required.

Implementation

It is not mandatory, but highly recommended to add an attribute named "log" in the device interface, strings spectrum type, which tracks all the internal activity of the device (as defined in Tango Logging).

- In C++, the class `Yat4Tango::InnerAppender` implements this functionality based on a dynamic attribute (no need to use Pogo).
- This system facilitates the recovery of errors and therefore the problems diagnosis. Problem solving will be faster and optimized.
- This feature is in particular very interesting for devices that manage automatic processes (like doing scans,..) which involve other devices. The operator has then an easy access through this "log" attribute to the behaviour and decisions taken by the device.

Example of using C++ (look at the YAT documentation for further explanations:

In the header file of the device

- Declaration of the service to use

```
//          (c) - Software Engineering Group - ESRF
//=============================================================
#ifndef _ATTRIBUTESEQUENCEWRITER_H
#define _ATTRIBUTESEQUENCEWRITER_H

#include <tango.h>
#include <yat4tango/InnerAppender.h>
#include <yat/memory/DataBuffer.h>
//using namespace Tango;
#include "AttributeSequenceWriterTask.h"
```

In the source code of the device

- `init_device` method: initialization of the "innerAppender"
- `delete_device` method: deletion of the "innerAppender"

```
void AttributeSequenceWriter::init_device()
{
  //- initialize the inner appender (first thing to do)
  try
  {
    yat4tango::InnerAppender::initialize(this, 512);
  }
  catch( Tango::DevFailed& df )
  {
    ERROR_STREAM << df << std::endl;
    this->set_state(Tango::FAULT);
    this->set_status( "initialization failed - could not instanciate the InnerAppender");
    return;
  }
```

```
//
void AttributeSequenceWriter::delete_device()
{
  yat4tango::TraceHelper t("AttributeSequenceWriter::delete_device", this);

    //- release the task
    if (this->m_task)
    {
        //- ask the task to quit
      this->m_task->exit();
    //- !!!!! never try to <delete> a yat4tango::DeviceTask, cause
    //- it commits suicide upon return of its main function !!!!!!
      this->m_task = 0;
    }

  //- remove the inner appender
  yat4tango::InnerAppender::release(this);
}
```

Error handling

The importance of rigorous error handling

The purpose of this paragraph is based on a statement on the Tango developers practice. Indeed, the error handling is often overlooked. A good error handling means easier debugging and maintenance.

This part is important, it is essential for the coding quality. These concepts are detailed in the Tango documentation referenced *"Reporting Error"*.

Typical cases to avoid:

- A device doesn't behave as expected but there is no indication why.
- The device is in *FAULT* state but the *Status* (the attribute) gives no indication on the problem nature, or worse, a bad indication (thus guiding the users in a wrong trail, with a loss of time and energy).
- The error messages are written in the jargon of the developer or the system expert.

The developer has to ensure:

- That any exception is caught, completed (Tango allows it) and spread (use of the `rethrow_exception` method),
- If an error occur it must be logged using the Tango Logging Service
- That the return code of a function is always analyzed,
- That the device *Status* is always coherent with the *State*,
- That the error messages are understandable for the final user and that they are supplemented by *logs* (*ERROR level, use of the error_stream macro*). The *Status* is the indicator that will help the user to find the error reason.
- **Ignore the "ideal situation":** In operation, the ideal setting is often jeopardized.
  - Eg: use of communication sockets: anticipate all the common communication problems: cable not connected, equipment off, sub-devices not started or in *FAULT*.

Implementation

On a more technical view, the Tango exceptions don't provide numerical identifier for discriminating exceptions. In the code, it isn't possible to distinguish two exceptions without having knowledge of the text (as string) conveyed by the said exception.

All exceptions are of type `Tango::DevFailed`. A DevFailed exception consists of these fields:

- Reason: string, defining the error type
  - Aim: refer the **operator** to the root cause
- Description: string, giving a more precise description
  - Aim: refer the **expert** of this system to the root cause.
- Origin: string, method where the exception was thrown
  - Aim : refer the **computer scientist** on the location of the failure in its code
- Severity: enumeration (rarely uses)
- To easily distinguish exceptions, it is recommended to use a finite list of error types for the Reason field, specify in capital letters:

Standardized name for error types

Standardized name for the error types
OUT_OF_MEMORY
HARDWARE_FAILURE
SOFTWARE_FAILURE
HDB_FAILURE
DATA_OUT_OF_RANGE
COMMUNICATION_BROKEN
OPERATION_NOT_ALLOWED
DRIVER_FAILURE
UNKNOW_ERROR
CORBA_TIMEOUT
Tango_CONNECTION_FAILED
Tango_COMMUNICATION_ERROR
Tango_WRONG_NAME_SYNTAX_ERROR
Tango_NON_DB_DEVICE_ERROR
Tango_WRONG_DATA_ERROR
Tango_NON_SUPPORTED_FEATURE_ERROR
Tango_ASYNC_CALL_ERROR
Tango_ASYNC_REPLY_NOT_ARRIVED_ERROR
Tango_EVENT_ERROR
Tango_DEVICE_ERROR
CONFIGURATION_ERROR
DEPENDENCY_ERROR
NO_DEPENDENCY

Table 2 : List of standardized error types for an exception

Example of an exception message:

Reason: DATA_OUT_OF_RANGE

Description: *AxisMotionAccuracy* must be at least of 1 motor step!

Origin: `GalilAxis::write_attr_hardware`

The exception hierarchy defined by Tango has been thought only for internal use (Tango core), the developer can't inherit and define its own inherited exceptions classes. This strong constraint is related to the underlying CORBA IDL.

Always keep the original exception. It must be the first visible item in the device status.

If there is a succession of exceptions, the logic dictates that the first exception has possibly generated all the others. By resolving the first exception, the others can disappear.

Exception handling in `init_device` method:

- no exceptions should be propagated from the method `MyDevice::init_device`. Otherwise, **the device quits.** The device should be kept alive regardless of any failure.
- The code for this method must contain a try / catch block, which guarantees that no exception is propagated in this context
- If an exception is thrown, the developer must set the device state to *FAULT* and update the *Status* to indicate the error nature. (*The goal is to understand easily why the device failed to initialize properly, while still allowing the operator to adjust this or these problems*)

Hint: Examples of error handling in C++:

- If an error occurs, always log it
- Always update *State* **AND** *Status*
- Manage the return code for function that have one
- Manage the exceptions for methods which can throw some

```
void NITCO1::read_attr_hardware(vector<long> &attr_list)
{
    DEBUG_STREAM << "NITCO1::read_attr_hardware(vector<long> &attr_list) entering... " << endl;
    /*----- PROTECTED REGION ID(NITCO1::read_attr_hardware) ENABLED START -----*/

    //  Add your own code

    int32 read = 0;
    float64 data[1000];
    int32 returnCode = -1;
    char errorString[1000];

    // Read only if state is ON or ALARM or WARNING
    if ((Tango::ON == get_state()) ||
        (Tango::ALARM == get_state()) ||
        (Tango::WARN == get_state()) )
    {
        // Read temperature attribute
        returnCode = DAQmxReadAnalogF64(_taskHandle, -1, 10.0, DAQmx_Val_GroupByScanNumber, data, 1000, &read, NULL);

        if (returnCode != 0)
        {
            // Error on reading temperature
            set_state(Tango::FAULT);
            _temperature_read = numeric_limits<double>::infinity();

            DAQmxGetErrorString(returnCode, errorString, 1000);
            set_status("Error on reading temperature - DAQmxReadAnalogF64() : " + string(errorString));
            ERROR_STREAM << "NITCO1::read_attr_hardware() - error on reading temperature for " << device_name << endl;
```

Details for an attribute

Although Tango supports the notion of quality on an attribute value (*Tango::VALID*, *Tango::INVALID*, ...), only few clients use this information to judge the validity of the data returned (which is a shame). So it is best to not make assumptions on the use that would be made (client side) to report an invalid value to the client. In other words, **forcing the attribute quality to :samp:`{Tango::INVALID}` is necessary but not sufficient.**

For float values, it is possible to set the value to "NaN", but there is no equivalent for an integer. To avoid the handling of special cases, it is recommended to throw an exception to indicate the data invalidity.

It is recommended to throw an exception for all invalid values, regardless of their type. There is, however, two exceptions to this rule: State and Status. For these two attributes, always return a value.

This solution has the disadvantage to show a pop-up on the client side, but this is the most effective method to indicate that the attribute reading has failed.

Details for the properties

Properties reading during device initialization

As it stands, the code generated by **Pogo** doesn't wrap in a try / catch block the method which ensures the properties reading in the Tango database (see `MyDevice::init_device`). However, it may fail and cause the generation of an exception. As mentioned above, the developer must ensure that any exception thrown in the `init_device` method (or a method called from it) is catch and not spread.

In case of Tango exception on the *properties* reading, the developer should systematically:

1. detect the error (catch).
2. log it with level ERROR.
3. set the device to the FAULT state.
4. update the Status indicating the problem origin.

Example in C++ :

```
void NITCO1::init_device()
{
    DEBUG_STREAM << "NITCO1::init_device() create device " << device_name << endl;

    /*----- PROTECTED REGION ID(NITCO1::init_device_before) ENABLED START -----*/

    //  Initialization before get_device_property() call

    //  Get the device properties (if any) from database
    try
    {
        get_device_property();
    }
    catch (Tango::DevFailed &ex)
    {
        set_state(Tango::FAULT);
        set_status("Error on initializing device - get_device_property() failed");
        ERROR_STREAM << "NITCO1::init_device() - Error on getting device properties for " << device_name << endl;
        return;
    }
    catch (...)
    {
        set_state(Tango::FAULT);
        set_status("Error on initializing device - unknown error");
        ERROR_STREAM << "NITCO1::init_device() - unknown error occured while initialising " << device_name << endl;
        return;
    }

    /*----- PROTECTED REGION END -----*/    //  NITCO1::init_device_before
```

As a reminder, the default value for a property is defined with Pogo, the value is stored in the database via the `put_property()` method.

Properties without default values

Pogo allows defining a default value for a *property* not present in the Tango database.

For mandatory properties that have no default values, the developer should systematically:

- detect the absence of the value in the database.
- log the problem explicitly with the level ERROR (indicate the missing property).
- set the device to the FAULT state.
- update the Status indicating the problem origin.

Appendices

Appendix 1 – Full code samples

Example C++: AttributeSequenceWriter

6.5.3 The TANGO device server model

Intended audience: developers, ***Programming language:*** all

This chapter will present the TANGO device server object model hereafter referred as TDSOM. First, it will introduce CORBA. Then, it will describe each of the basic features of the TDSOM and their function. The TDSOM can be divided into the following basic elements - the *device*, the *server*, the *database* and the *application programmers interface*. This chapter will treat each of the above elements separately.

Introduction to CORBA

CORBA is a definition of how to write object request brokers (ORB). The definition is managed by the Object Management Group (OMG home page). Various commercial and non-commercial implementations exist for CORBA for all the mainstream operating systems. CORBA uses a programming language independent definition language (called IDL) to defined network object interfaces. Language mappings are defined from IDL to the main programming languages e.g. C++, Java, C, COBOL, Smalltalk and ADA. Within an interface, CORBA defines two kinds of actions available to the outside world. These actions are called **attributes** and **operations**.

Operations are all the actions offered by an interface. For instance, within an interface for a Thermostat class, operations could be the action to read the temperature or to set the nominal temperature. An attribute defines a pair of operations a client can call to send or receive a value. For instance, the position of a motor can be defined as an attribute because it is a data that you only set or get. A read only attribute defines a single operation the client can call to receives a value. In case of error, an operation is able to throw an exception to the client, attributes cannot raises exception except system exception (du to network fault for instance).

Intuitively, IDL interface correspond to C++ classes and IDL operations correspond to C++ member functions and attributes as a way to read/write public member variable. Nevertheless, IDL defines only the interface to an object and say nothing about the object implementation. IDL is only a descriptive language. Once the interface is fully described in the IDL language, a compiler (from IDL to C++, from IDL to Java...) generates code to implement this interface. Obviously, you still have to write how operations are implemented.

The act of invoking an operation on an interface causes the ORB to send a message to the corresponding object implementation. If the target object is in another address space, the ORB run time sends a remote procedure call to the implementation. If the target object is in the same address space as the caller, the invocation is accomplished as an ordinary function call to avoid the overhead of using a networking protocol.

For an excellent reference on CORBA with C++ refer to [Henning]. The complete TANGO IDL file can be found in the TANGO home page or at the end of this document in the appendix 2 chapter.

The model

The basic idea of the TDSOM is to treat each device as an **object**. Each device is a separate entity which has its own data and behavior. Each device has a unique name which identifies it in network name space. Devices are organized according to **classes**, each device belonging to a class. All classes are derived from one root class thus allowing some common behavior for all devices. Four kind of requests can be sent to a device (locally i.e. in the same process, or remotely i.e. across the network) :

- Execute actions via **commands**
- Read/Set data specific to each device belonging to a class via TANGO **attributes**
- Read/Set data specific to each device belonging to a class via TANGO **pipes**
- Read some basic device data available for all devices via CORBA attributes.
- Execute a predefined set of actions available for every devices via CORBA operations

Each device is stored in a process called a **device server**. Devices are configured at runtime via **properties** which are stored in a **database**.

The device

The device is the heart of the TDSOM. A device is an abstract concept defined by the TDSOM. In reality, it can be a piece of hardware (an interlock bit) a collection of hardware (a screen attached to a stepper motor) a logical device (a taper) or a combination of all these (an accelerator). Each device has a unique name in the control system and eventually one alias. Within Tango, a four field name space has been adopted consisting of

[//FACILITY/]DOMAIN/CLASS/MEMBER

Facility refers to the control system instance, domain refers to the sub-system, class the class and member the instance of the device. Device name alias(es) must also be unique within a control system. There is no predefined syntax for device name alias.

Each device belongs to a class. The device class contains a complete description and implementation of the behavior of all members of that class. New device classes can be constructed out of existing device classes. This way a new hierarchy of classes can be built up in a short time. Device classes can use existing devices as sub-classes or as sub-objects. The practice of reusing existing classes is classical for Object Oriented Programming and is one of its main advantages.

All device classes are derived from the same class (the device root class) and implement **the same CORBA interface**. All devices implementing the same CORBA interface ensures all control object support the same set of CORBA operations and attributes. The device root class contains part of the common device code. By inheriting from this class, all devices shared a common behavior. This also makes maintenance and improvements to the TDSOM easy to carry out.

All devices also support a **black box** where client requests for attributes or operations are recorded. This feature allows easier debugging session for device already installed in a running control system.

The commands

Each device class implements a list of commands. Commands are very important because they are the client's major dials and knobs for controlling a device. Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen in a fixed set of data types: All simple types (boolean, short, long (32 bits), long (64 bits), float, double, unsigned short, unsigned long (32 bits), unsigned long (64 bits) and string) and arrays of simple types plus array of strings and longs and array of strings and doubles). Commands can execute any sequence of actions. Commands can be executed synchronously (the requester is blocked until the command ended) or asynchronously (the requester send the request and is called back when the command ended).

Commands are executed using two CORBA operations named **command_inout** for synchronous commands and **command_inout_async** for asynchronous commands. These two operations called a special method implemented in the device root class - the *command_handler* method. The *command_handler* calls an *is_allowed* method implemented in the device class before calling the command itself. The *is_allowed* method is specific to each command[1]. It checks to see whether the command to be executed is compatible with the present device state. The command function is executed only if the *is_allowed* method allows it. Otherwise, an exception is sent to the client.

[1] In contrary to the state_handler method of the TACO device server model which is not specific to each command.

The TANGO attributes

In addition to commands, TANGO devices also support normalized data types called attributes[2]. Commands are device specific and the data they transport are not normalized i.e. they can be any one of the TANGO data types with no restriction on what each byte means. This means that it is difficult to interpret the output of a command in terms of what kind of value(s) it represents. Generic display programs need to know what the data returned represents, in what units it is, plus additional information like minimum, maximum, quality etc. Tango attributes solve this problem.

TANGO attributes are zero, one or two dimensional data which have a fix set of properties e.g. quality, minimum and maximum, alarm low and high. They are transferred in a specialized TANGO type and can be read, write or read-write. A device can support a list of attributes. Clients can read one or more attributes from one or more devices. To read TANGO attributes, the client uses the **read_attributes** operation. To write TANGO attributes, a client uses the **write_attributes** operation. To write then read TANGO attributes within the same network request, the client uses the **write_read_attributes** operation. To query a device for all the attributes it supports, a client uses the **get_attribute_config** operation. A client is also able to modify some of parameters defining an attribute with the **set_attribute_config** operation. These five operations are defined in the device CORBA interface.

TANGO support thirteen data types for attributes (and arrays of for one or two dimensional data) which are: boolean, short, long (32 bits), long (64 bits), float, double, unsigned char, unsigned short, unsigned long (32 bits), unsigned long (64 bits), string, a specific data type for Tango device state and finally another specific data type to transfer data as an array of unsigned char with a string describing the coding of these data.

The TANGO pipes

Since release 9, in addition to commands and attributes, TANGO devices also support pipes.

In some cases, it is required to exchange data between client and device of varrying data type. This is for instance the case of data gathered during a scan on one experiment. Because the number of actuators and sensors involved in the scan may change from one scan to another, it is not possible to use a well defined data type. TANGO pipes have been designed for such cases. A TANGO pipe is basically a pipe dedicated to transfer data between client and device. A pipe has a set of two properties which are the pipe label and its description. A pipe can be read or read-write. A device can support a list of pipes. Clients can read one or more pipes from one or more devices. To read a TANGO pipe, the client uses the **read_pipe** operation. To write a TANGO pipe, a client uses the **write_pipe** operation. To write then read a TANGO pipe within the same network request, the client uses the **write_read_pipe** operation. To query a device for all the pipes it supports, a client uses the **get_pipe_config** operation. A client is also able to modify some of parameters defining a pipe with the **set_pipe_config** operation. These five operations are defined in the device CORBA interface.

In contrary of commands or attributes, a TANGO pipe does not have a pre-defined data type. Data transferred through pipes may be of any basic Tango data type (or array of) and this may change every time a pipe is read or written.

Command, attributes or pipes ?

There are no strict rules concerning what should be returned as command result and what should be implemented as an attribute or as a pipe. Nevertheless, attributes are more adapted to return physical value which have a kind of time consistency. Attribute also have more properties which help the client to precisely know what it represents. For instance, the state and the status of a power supply are not physical values and are returned as command result. The current generated by the power supply is a physical value and is implemented as an attribute. The attribute properties allow a client to know its unit, its label and some other informations which are related to a physical value. Command are well adapted to send order to a device like switching from one mode of operation to another mode of operation. For a power supply, the switch from a STANDBY mode to a ON mode is typically done via a command. Finally pipe is well adapted when the kind and number of data exchanged between the client and the device change with time.

[2] TANGO attributes were known as signals in the TACO device server model

The CORBA attributes

Some key data implemented for each device can be read without the need to call a command or read an attribute. These data are :

- The device state
- The device status
- The device name
- The administration device name called adm_name
- The device description

The device state is a number representing its state. A set of predefined states are defined in the TDSOM. The device status is a string describing in plain text the device state and any additional useful information of the device as a formatted ascii string. The device name is its name as defined in [sec:dev]. For each set of devices grouped within the same server, an administration device is automatically added. This adm_name is the name of the administration device. The device description is also an ascii string describing the device rule.

These five CORBA attributes are implemented in the device root class and therefore do not need any coding from the device class programmer. As explained in [sec:corba], the CORBA attributes are not allowed to raise exceptions whereas command (which are implemented using CORBA operations) can.

The remaining CORBA operations

The TDSOM also supports a list of actions defined as CORBA operations in the device interface and implemented in the device root class. Therefore, these actions are implemented automatically for every TANGO device. These operations are :

ping	to ping a device to check if the device is alive. Obviously, it checks only the connection from a client to the device and not all the device functionalities
command_list_query	request a list of all the commands supported by a device with their input and output types and description
command_query	request information about a specific command which are its input and output type and description
info	request general information on the device like its name, the host where the device server hosting the device is running…
black_box	read the device black-box as an array of strings

The special case of the device state and status

Device state and status are the most important key device informations. Nearly all client software dealing with Tango device needs device(s) state and/or status. In order to simplify client software developper work, it is possible to get these two piece of information in three different manners :

1. Using the appropriate CORBA attribute (state or status)
2. Using command on the device. The command are called State or Status
3. Using attribute. Even if the state and status are not real attribute, it is possible to get their value using the read_attributes operation. Nevertheless, it is not possible to set the attribute configuration for state and status. An error is reported by the server if a client try to do so.

The device polling

Within the Tango framework, it is also possible to force executing command(s) or reading attribute(s) at a fixed frequency. It is called *device polling*. This is automatically handled by Tango core software with a polling threads pool. The command result or attribute value are stored in circular buffers. When a client want to read attribute value (or command result) for a polled attribute (or a polled command), he has the choice to get the attribute value (or command result) with a real access to the device of from the last value stored in the device ring buffer. This is a great advantage for "slow" devices. Getting data from the buffer is much faster than accessing the device itself. The technical disadvantage is the time shift between the data returned from the polling buffer and the time of the request. Polling a command is only possible for command without input arguments. It is not possible to poll a device pipe.

Two other CORBA operations called *command_inout_history_X* and *read_attribute _history_X* allow a client to retrieve the history of polled command or attribute stored in the polling buffers. Obviously, this history is limited to the depth of the polling buffer.

The whole polling system is available only since Tango release 2.x and above in CPP and since TangORB release 3.7.x and above in Java.

The server

Another integral part of the TDSOM is the server concept. The server (also referred as device server) is a process whose main task is to offer one or more services to one or more clients. To do this, the server has to spend most of its time in a wait loop waiting for clients to connect to it. The devices are hosted in the server process. A server is able to host several classes of devices. In the TDSOM, a device of the **DServer** class is automatically hosted by each device server. This class of device supports commands which enable remote device server process administration.

TANGO supports device server process on two families of operating system : Linux and Windows.

The Tango Logging Service

During software life, it is always convenient to print miscellaneous informations which help to:

- Debug the software
- Report on error
- Give regular information to user

This is classically done using `cout` (or C `printf`) in C++ or `println` method in Java language. In a highly distributed control system, it is difficult to get all these informations coming from a high number of different processes running on a large number of computers. Since its release 3, Tango has incorporated a Logging Service called the Tango Logging Service (TLS) which allows print messages to be:

- Displayed on a console (the classical way)
- Sent to a file
- Sent to specific Tango device called log consumer. Tango package has an implementation of log consumer where every consumer device is associated to a graphical interface. This graphical interface display messages but could also be used to sort messages, to filter messages… Using this feature, it is possible to centralise display of these messages coming from different devices embedded within different processes. These log consumers can be:
  - Statically configured meaning that it memorizes the list of Tango devices for which it will get and display messages.
  - Dynamically configured. The user, with the help of the graphical interface, chooses devices from which he want to see messages.

The database

To achieve complete device independence, it is necessary however to supplement device classes with a possibility for configuring device dependencies at runtime. The utility which does this in the TDSOM is the **property database**. Properties[3] are identified by an ascii string and the device name. TANGO attributes are also configured using properties. This database is also used to store device network addresses (CORBA IOR's), list of classes hosted by a device server process and list of devices for each class in a device server process. The database ensure the uniqueness of device name and of alias. It also links device name and it list of aliases.

TANGO uses MySQL (MySQL home page) as its database. MySQL is a relational database which implements the SQL language. However, this is largely enough to implement all the functionalities needed by the TDSOM. The database is accessed via a classical TANGO device hosted in a device server. Therefore, client access the database via TANGO commands requested on the database device. For a good reference on MySQL refer to [MySQLbook].

The controlled access

Tango also provides a controlled access system. It's a simple controlled access system. It does not provide encrypted communication or sophisticated authentification. It simply defines which user (based on computer loggin authentification) is allowed to do which command (or write attribute) on which device and from which host. The information used to configure this controlled access feature are stored in the Tango database and accessed by a specific Tango device server which is not the classsical Tango database device server described in the previous section. Two access levels are defined:

- Everything is allowed for this user from this host
- The write-like calls on the device are forbidden and according to configuration, a command subset is also forbidden for this user from this host

This feature is precisely described in the chapter Advanced features

[3] Properties were known as resources in the TACO device server model

The Application Programmers Interfaces

Rules of the API

While it is true TANGO clients can be programmed using only the CORBA API, CORBA knows nothing about TANGO. This means client have to know all the details of retrieving IORs from the TANGO database, additional information to send on the wire, TANGO version control etc. These details can and should be wrapped in TANGO Application Programmer Interface (API). The API is implemented as a library in C++ and as a package in Java. The API is what makes TANGO clients easy to write. The API's consists the following basic classes :

- DeviceProxy which is a *proxy* to the real device
- DeviceData to encapsulate data send/receive from/to device via commands
- DeviceAttribute to encapsulate data send/receive from/to device via attributes
- Group which is a *proxy* to a group of devices

In addition to these main classes, many other classes allows a full interface to TANGO features. The following figure is a drawing of a typical client/server application using TANGO.

The database is used during server and client startup phase to establish connection between client and server.

Communication between client and server using the API

With the API, it is possible to request command to be executed on a device or to read/write device attribute(s) using one of the two communication models implemented. These two models are:

1. The synchronous model where client waits (and is blocked) for the server to send the answer or until the timeout is reached
2. The asynchronous model. In this model, the clients send the request and immediately returns. It is not blocked. It is free to do whatever it has to do like updating a graphical user interface. The client has the choice to retrieve the server answer by checking if the reply is arrived by calling an API specific call or by requesting that a call-back method is executed when the client receives the server answer.

The asynchronous model is available with Tango release 3 and above.

Tango events

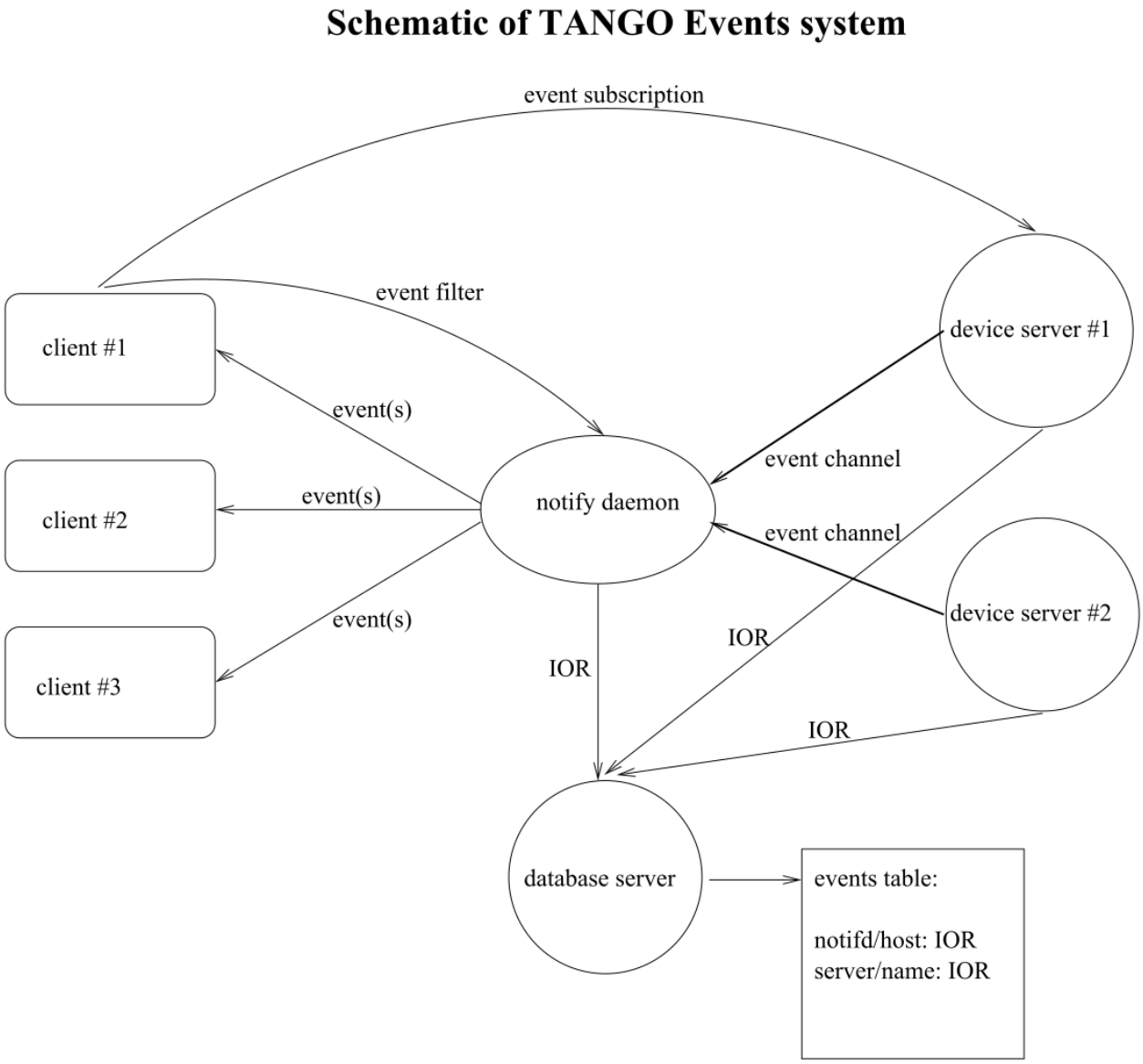
On top of the two communication model previously described, TANGO offers an event system. The standard TANGO communication paradigm is a synchronou/asynchronous two-way call. In this paradigm the call is initiated by the client who contacts the server. The server handles the client's request and sends the answer to the client or throws an exception which the client catches. This paradigm involves two calls to receive a single answer and requires the client to be active in initiating the request. If the client has a permanent interest in a value he is obliged to poll the server for an update in a value every time. This is not efficient in terms of network bandwidth nor in terms of client programming.

For clients who are permanently interested in values the event-driven communication paradigm is a more efficient and natural way of programming. In this paradigm the client registers his interest once in an event (value). After that the server informs the client every time the event has occurred. This paradigm avoids the client polling, frees it for doing other things, is fast and makes efficient use of the network.

Before TANGO release 8, TANGO used the CORBA OMG COS Notification Service to generates events. TANGO uses the omniNotify implementation of the Notification service. omniNotify was developed in conjunction with the omniORB CORBA implementation also used by TANGO. The heart of the Notification Service is the notification daemon. The omniNotify daemons are the processes which receive events from device servers and distribute them to

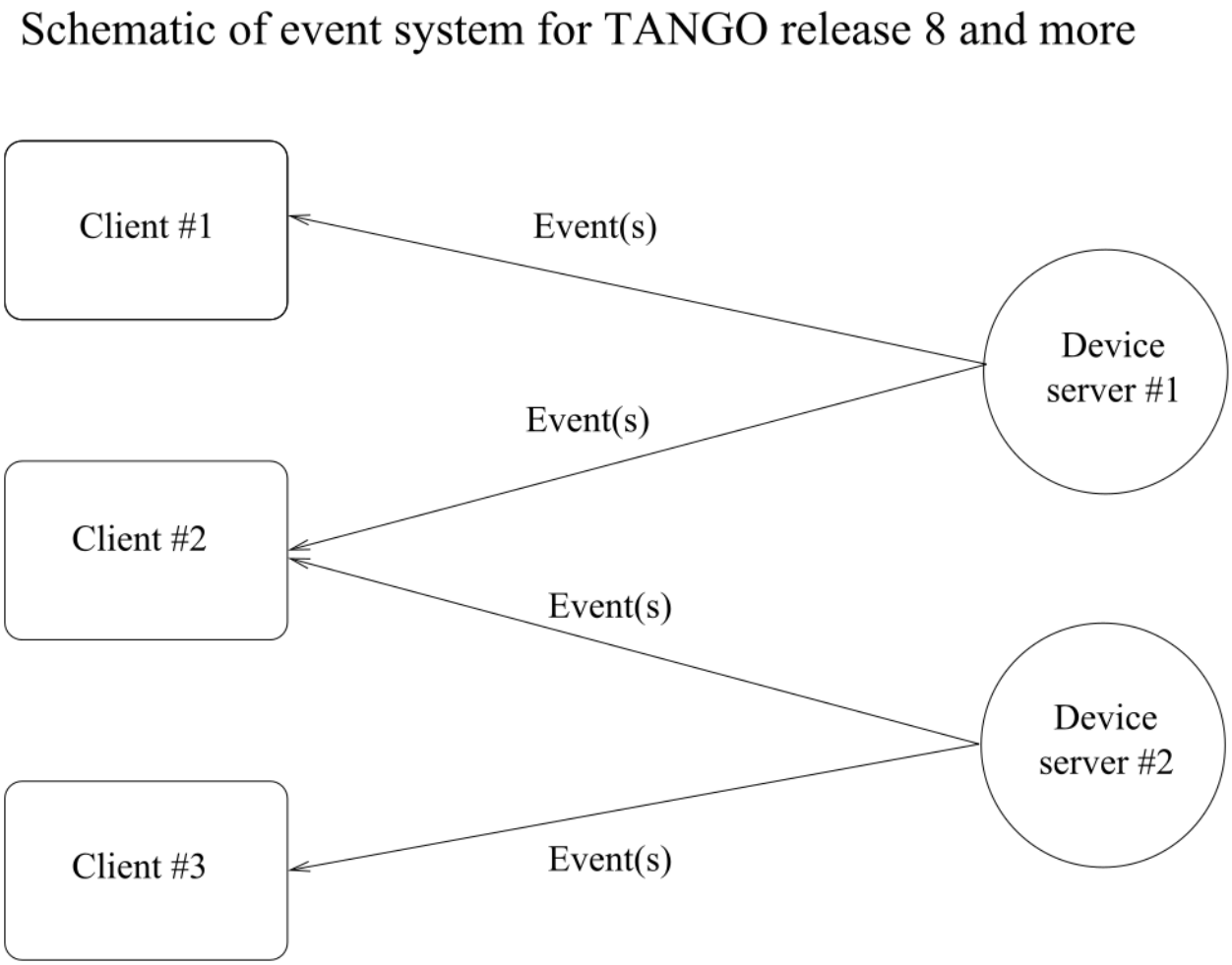
all clients which are subscribed. In order to distribute the load of the events there is one notification daemon per host. Servers send their events to the daemon on the local host. Clients and servers get the IOR for the host from the TANGO database.

The following figure is a schematic of the Tango event system for Tango releases before Tango 8.



Starting with Tango 8, a new design of the event system has been implemented. This new design is based on the ZMQ library. ZMQ is a library allowing users to create communicating system. It implements several well known communication pattern including the Publish/Subscribe pattern which is the basic of the new Tango event system. Using this library, a separate notification service is not needed anymore and event communiction is available with only client and server processes which simplifies the overall design. Starting with Tango 8.1, the event propagation between devices and clients could be done using a multicasting protocol. The aim of this is to reduce both the network bandwidth use and the CPU consumption on the device server side. See chapter on Advanced Features to get all the details on this feature.

The following figure is a schematic of the Tango event system for Tango releases starting with Tango release 8.


Schematic of event system for TANGO release 8 and more
Client #1
Client #2
Client #3
Event(s)
Event(s)
Event(s)
Event(s)
Device server #1
Device server #2

6.5.4 Writing a TANGO device server

Intended audience: developers, ***Programming language:*** c++

The device server framework

This chapter will present the TANGO device server framework. It will introduce what is the device server pattern and then it will describe a complete device server framework. A definition of classes used by the device server framework is given in this chapter. This manual is not intended to give the complete and detailed description of classes data member or methods, refer to [TangoRefMan] to get this full description. But first, the naming convention used in this project is detailed.

The aim of the class definition given in this chapter is only to help the reader to understand how a TANGO device server works. For a detailed description of these classes (and their methods), refer to chapter [Writing_chapter] or to [TangoRefMan].

Naming convention and programming language

TANGO fully supports three different programming languages which are **C++, Java** and **Python**. This documentation focuses on C++ Tango class. For Java and Python Tango class, have a look at the [TangoRefMan] pages where similar chapter for Java and Python are available.

Every software project needs a naming convention. The naming convention adopted for the TDSOM is very simple and only defines two guidelines which are:

- Class names start with uppercase and use capitalization for compound words (For instance MyClassName).
- Method names are in lowercase and use underscores for compound words (For instance my_method_name).

The device pattern

Device server are written using the Device pattern. The aim of this pattern is to provide the control programmer with a framework in which s/he can develop new control objects. The device pattern uses other design patterns like the Singleton and Command patterns. These patterns are fully described in [Patterns]. The device pattern class diagram for stepper motor device is drawn in figure *6.1*

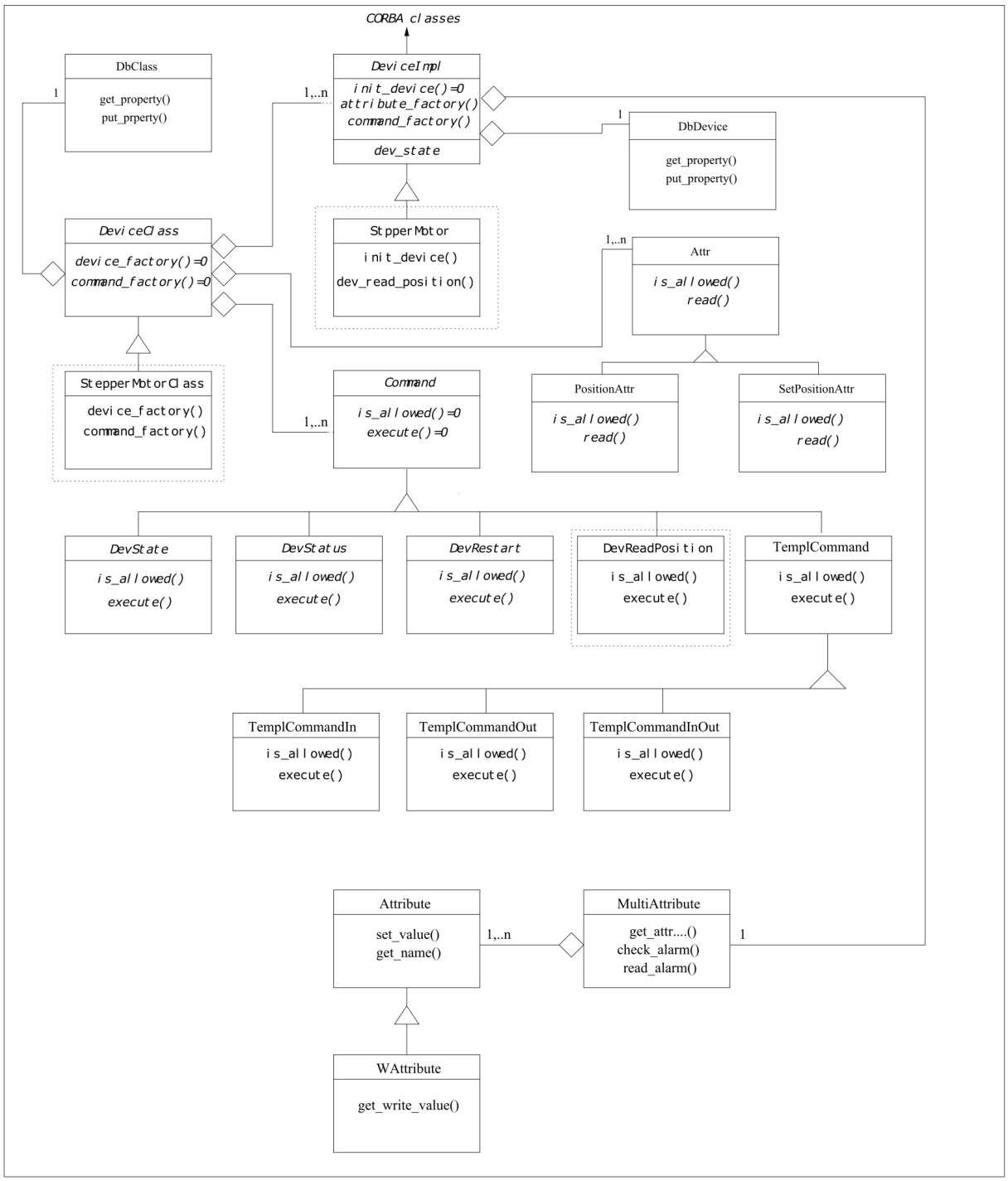


Fig. 6: Figure 6.1: Device pattern class diagram

. In this figure, only classes surrounded with a dash line square are device specific. All the other classes are part of the TDSOM core and are developed by the Tango system team. Different kind of classes are used by the device pattern.

- Three of them are root classes and it is only necessary to inherit from them. These classes are the **DeviceImpl**, **DeviceClass** and **Command** classes.
- Classes necessary to implement commands. The TDSOM supports two ways to create command : Using inher-

itance or using the template command model. It is possible to mix model within the same device pattern

1. Using **inheritance**. This model of creating command heavily used the polymorphism offered by each modern object oriented programming language. In this schema, each command supported by a device via the command_inout or command_inout_async operation is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. A *execute* method must be defined in each sub-class. A *is_allowed* method may also be re-defined in each class if the default one does not fulfill all the needs[1]. In our stepper motor device server example, the DevReadPosition command follows this model.
2. Using the **template command** model. Using this model, it is not necessary to write one class for each command. You create one instance of classes already defined in the TDSOM for each command. The link between command name and method which need to be executed is done through pointers to method. To support different kind of command, four classes are part of the TDSOM. These classes are :
    1. The **TemplCommand** class for command without input or output parameter
    2. The **TemplCommandIn** class for command with input parameter but without output parameter
    3. The **TemplCommandOu**t class for command with output parameter but without input parameter
    4. The **TemplCommandInOut** class for all the remaining commands

- Classes necessary to implement TANGO device attributes. All these classes are part of the TANGO core classes. These classes are the **MultiAttribute**, **Attribute**, **WAttribute**, **Attr**, **SpectrumAttr** and **ImageAttr** classes. The last three are used to create user attribute. Each attribute supported by a device is implemented by a separate class. The Attr class is the root class for each of these classes. According to the attribute data format, the user class implementing the attribute must inherit from the Attr, SpectrumAttr or ImageAtttr class. SpectrumAttr class inherits from Attr class and Image Attr class inherits from the SpectrumAttr class. The Attr base class defined three methods called *is_allowed*, *read* and *write*. These methods may be redefined in sub-classes in order to implement the attribute specific behaviour.
- The other are device specific. For stepper motor device, they are named StepperMotor, StepperMotorClass and DevReadPosition.

The Tango base class (DeviceImpl class)

Description

This class is the device root class and is the link between the Device pattern and CORBA. It inherits from CORBA classes and implements all the methods needed to execute CORBA operations and attributes. For instance, its method *command_inout* is executed when a client requests a command_inout operation. The method *name* of the DeviceImpl class is executed when a client requests the name CORBA attribute. This class also encapsulates some key device data like its name, its state, its status, its black box.... This class is an abstract class and cannot be instantiated as is.

[1] The default is_allowed method behavior is to always allows the command

Contents

The contents of this class can be summarized as :

- Different constructors and one destructor
- Methods to access instance data members outside the class or its derivate classes. These methods are necessary because data members are declared as protected.
- Methods triggered by CORBA attribute request
- Methods triggered by CORBA operation request
- The *init_device()* method. This method makes the class abstract. It should be implemented by a sub-class. It is used by the inherited classes constructors.
- Methods triggered by the automatically added State and Status commands. These methods are declared virtual and therefore can be redefined in sub-classes. These two commands are automatically added to the list of commands defined for a class of devices. They are discussed in chapter [Auto_cmd]
- A method called *always_executed_hook()* always executed for each command before the device state is tested for command execution. This method gives the programmer a hook where he(she) can program some mandatory action which must be done before any command execution. An example of the such action is an hardware access to the device to read its real hardware state.
- A method called *read_attr_hardware()* triggered by the read_attributes CORBA operation. This method is called once for each read_attributes call. This method is virtual and may be redefined in sub-classes.
- A method called *write_attr_hardware()* triggered by the write_attributes CORBA operation. This method is called once for each write_attributes call. This method is virtual and may be redefined in sub-classes.
- Methods for signal management (C++ specific)
- Data members like the device name, the device status, the device state
- Some private methods and data members

The DbDevice class

Each DeviceImpl instance is an aggregate with one instance of the DbDevice class. This DbDevice class can be used to query or modify device properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the Tango API reference documentation available on the Tango WEB pages.

The Command class

Description of the inheritance model

Within the TDSOM, each command supported by a device and implemented using the inheritance model is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. It stores the command name, the command argument types and description and mainly defines two methods which are the *execute* and *is_allowed* methods. The *execute* method should be implemented in each sub-class. A default *is_allowed* method exists for command always allowed. A command also stores a parameter which is the command display type. It is also used to select if the command must be displayed according to the application mode (every day operation or expert mode).

Description of the template model

Using this method, it is not necessary to create a separate class for each device command. In this method, each command is represented by an instance of one of the template command classes. They are four template command classes. All these classes inherits from the Command class. These four classes are :

1. The **TemplCommand** class. One object of this class must be created for each command without input nor output parameters
2. The **TemplCommandIn** class. One object of this class must be created for each command without output parameter but with input parameter
3. The **TemplCommandOut** class. One object of this class must be created for each command without input parameter but with output parameter
4. The **TemplCommandInOut** class. One object of this class must be created for each command with input and output parameters

These four classes redefine the *execute* and *is_allowed* method of the Command class. These classes provides constructors which allow the user to :

- specify which method must be executed by these classes *execute* method
- optionally specify which method must be executed by these classes *is_allowed* method.

The method specification is done via pointer to method.

Remember that it is possible to mix command implementation method within the same device pattern.

Contents

The content of this class can be summarizes as :

- Class constructors and destructor
- Declaration of the *execute* method
- Declaration of the *is_allowed* method
- Methods to read/set class data members
- Methods to extract data from the object used to transfer data on the network
- Methods to insert data into the object used to transfer data on the network
- Class data members like command name, command input data type, command input data description...

The DeviceClass class

Description

This class implements all what is specific for a controlled object class. For instance, every device of the same class supports the same list of commands and therefore, this list of available commands is stored in this DeviceClass. The structure returned by the info operation contains a documentation URL[2]. This documentation URL is the same for every device of the same class. Therefore, the documentation URL is a data member of this class. There should have only one instance of this class per device pattern implementation. The device list is also stored in this class. It is an abstract class because the two methods *device_factory()* and *command_factory()* are declared as pure virtual. The rule of the *device_factory()* method is to create all the devices belonging to the device class. The rule of the *command_factory()*

[2] URL stands for **U**niform **R**esource **L**ocator

method is to create one instance of all the classes needed to support device commands. This class also stored the *attribute_factory* method. The rule of this method is to store in a vector of strings, the name of all the device attributes. This method has a default implementation which is an empty body for device without attribute.

Contents

The contents of this class can be summarize as :

- The *command_handler* method
- Methods to access data members.
- Signal related method (C++ specific)
- Class constructor. It is protected to implements the Singleton pattern
- Class data members like the class command list, the device list...

The DbClass class

Each DeviceClass instance is an aggregate with one instance of the DbClass class. This DbClass class can be used to query or modify class properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the reference Tango C++ API documentation available in the Tango WEB pages.

The MultiAttribute class

Description

This class is a container for all the TANGO attributes defined for the device. There is one instance of this class for each device. This class is mainly an aggregate of Attribute object(s). It has been developed to ease TANGO attribute management.

Contents

The class contents could be summarizes as :

- Miscellaneous methods to retrieve one attribute object in the aggregate
- Method to retrieve a list of attribute with an alarm level defined
- Get attribute number method
- Miscellaneous methods to check if an attribute value is outside the authorized limits
- Method to add messages for all attribute with an alarm set
- Data members with the attribute list

The Attribute class

Description

There is one object of this class for each device attribute. This class is used to store all the attribute properties, the attribute value and all the alarm related data. Like commands, this class also stores th attribute display type. It is foreseen to be used by future Tango graphical application toolkit to select if the attribute must be displayed according to the application mode (every day operation or expert mode).

Contents

- Miscellaneous method to get boolean attribute information
- Methods to access some data members
- Methods to get/set attribute properties
- Method to check if the attribute is in alarm condition
- Methods related to attribute data
- Friend function to print attribute properties
- Data members (properties value and attribute data)

The WAttribute class

Description

This class inherits from the Attribute class. There is one instance of this class for each writable device attribute. On top of all the data already managed by the Attribute class, this class stores the attribute set value.

Contents

Within this class, you will mainly find methods related to attribute set value storage and some data members.

The Attr class

Within the TDSOM, each attribute supported by a device is implemented by a separate class. The Attr class is the root class for each of these classes. It is used in conjonction with the Attribute and Wattribute classes to implement Tango attribute behaviour. It defines three methods which are the *is_allowed, read* and *write* methods. A default *is_allowed* method exists for attribute always allowed. Default *read* and *write* empty methods are defined. For readable attribute, it is necessary to overwrite the *read* method. For writable attribute, it is necessary to overwrite the *write* method and for read and write attribute, both methods must be overwritten.

The SpectrumAttr class

This class inherits from the Attr class. It is the base class for user spectrum attribute. It is used in conjonction with the Attribute and WAttribute class to implement Tango spectrum attribute behaviour. From the Attr class, it inherits the Attr *is_allowed*, *read* and *write* methods.

The ImageAttr class

This class inherits from the SpectrumAttr class. It is the base class for user image attribute. It is used in conjonction with the Attribute and WAttribute class to implement Tango image attribute behaviour. From the Attr class, it inherits the Attr *is_allowed*, *read* and *write* methods.

The StepperMotor class

Description

This class inherits from the DeviceImpl class and is the class implementing the controlled object behavior. Each command will trigger a method in this class written by the device server programmer and specific to the object to be controlled. This class also stores all the device specific data.

Definition

```
class StepperMotor: public TANGO_BASE_CLASS
{
public :
   StepperMotor(Tango::DeviceClass *,string &);
   StepperMotor(Tango::DeviceClass *,const char *);
   StepperMotor(Tango::DeviceClass *,const char *,const char *);
   ~StepperMotor() {};

   DevLong dev_read_position(DevLong);
  DevLong dev_read_direction(DevLong);
  bool direct_cmd_allowed(const CORBA::Any &);

  virtual Tango::DevState dev_state();
  virtual Tango::ConstDevString dev_status();

  virtual void always_executed_hook();

  virtual void read_attr_hardware(vector<long> &attr_list);
  virtual void write_attr_hardware(vector<long> &attr_list);

  void read_position(Tango::Attribute &);
  bool is_Position_allowed(Tango::AttReqType req);
  void write_SetPosition(Tango::WAttribute &);
  void read_Direction(Tango::Attribute &);

  virtual void init_device();
  virtual void delete_device();
```

(continues on next page)

(continued from previous page)

```
28
29    void get_device_properties();
30
31  protected :
32    long axis[AGSM_MAX_MOTORS];
33    DevLong position[AGSM_MAX_MOTORS];
34    DevLong direction[AGSM_MAX_MOTORS];
35    long state[AGSM_MAX_MOTORS];
36
37    Tango::DevLong *attr_Position_read;
38    Tango::DevLong *attr_Direction_read;
39    Tango::DevLong attr_SetPosition_write;
40
41    Tango::DevLong min;
42    Tango::DevLong max;
43
44    Tango::DevLong *ptr;
45  };
46
47  } /* End of StepperMotor namespace */
```

Line 1 : The StepperMotor class inherits from the DeviceImpl class

Line 4-7 : Class constructors and destructor

Line 9 : Method triggered by the DevReadPosition command

Line 10-11 : Methods triggered by the DevReadDirection command

Line 13 : Redefinition of the *dev_state* method of the DeviceImpl class. This method will be triggered by the State command

Line 14 : Redefinition of the *dev_status* method of the DeviceImpl class. This method will be triggered by the Status command

Line 16 : Redefinition of the *always_executed_hook* method.

Line 26 : Definition of the *init_device* method (declared as pure virtual by the DeviceImpl class)

Line 27 : Definition of the *delete_device* method

Line 31-45 : Device data

The StepperMotorClass class

Description

This class inherits from the DeviceClass class. Like the DeviceClass class, there should be only one instance of the StepperMotorClass. This is ensured because this class is written following the Singleton pattern as defined in [Patterns]. All controlled object class data which should be defined only once per class must be stored in this object.

Definition

```
1  class StepperMotorClass : public DeviceClass
2  {
3    public:
4      static StepperMotorClass *init(const char *);
5      static StepperMotorClass *instance();
6      ~StepperMotorClass() {_instance = NULL;}
7
8    protected:
9      StepperMotorClass(string &);
10     static StepperMotorClass *_instance;
11     void command_factory();
12
13   private:
14     void device_factory(Tango_DevVarStringArray *);
15 };
```

Line 1 : This class is a sub-class of the DeviceClass class

Line 4-5 and 9-10: Methods and data member necessary for the Singleton pattern

Line 6 : Class destructor

Line 11 : Definition of the *command_factory* method declared as pure virtual in the DeviceClass call

Line 13-14 : Definition of the *device_factory* method declared as pure virtual in the DeviceClass class

The DevReadPosition class

Description

This is the class for the DevReadPosition command. This class implements the *execute* and *is_allowed* methods defined by the Command class. This class is necessary because this command is implemented using the inheritance model.

Definition

```
1  class DevReadPositionCmd : public Command
2  {
3  public:
4      DevReadPositionCmd(const char *,Tango_CmdArgType, Tango_CmdArgType, const char *,␣
↪const char*);
5      ~DevReadPositionCmd() {};
6
7      virtual bool is_allowed (DeviceImpl *, const CORBA::Any &);
8      virtual CORBA::Any *execute (DeviceImpl *, const CORBA::Any &);
9  };
```

Line 1 : The class is a sub class of the Command class

Line 4-5 : Class constructor and destructor

Line 7-8 : Definition of the *is_allowed* and *execute* method declared as pure virtual in the Command class.

The PositionAttr class

Description

This is the class for the Position attribute. This attribute is a scalar attribute and therefore inherits from the Attr base class. This class implements the *read* and *is_allowed* methods defined by the Attr class.

Definition

```
1  class PositionAttr: public Tango::Attr
2  {
3  public:
4     PositionAttr():Attr("Position",Tango::DEV_LONG,Tango::READ);
5     ~PositionAttr() {};
6
7     virtual void read(Tango::DeviceImpl *dev,Tango::Attribute &att)
8     {(static_cast<StepperMotor *>(dev))->read_Position(att);}
9     virtual bool is_allowed(Tango::DeviceImpl *dev,Tango::AttReqType ty)
10    {return (static_cast<StepperMotor *>(dev))->is_Position_allowed(ty);}
11 };
```

Line 1 : The class is a sub class of the Attr class

Line 4-5 : Class constructor and destructor

Line 7 : Re-definition of the *read* method defined in the Attr class. This is simply a forward to the *read_Position* method of the StepperMotor class

Line 9 : Re-definition of the *is_allowed* method defined in the Attr class. This is also a forward to the *is_Position_allowed* method of the StepperMotor class

Startup of a device pattern

To start the device pattern implementation for stepper motor device, four methods of the StepperMotorClass class must be executed. These methods are :

1. The creation of the StepperMethodClass singleton via its *init*() method
2. The *command_factory*() method of the StepperMotorClass class
3. The *attribute_factory*() method of the StepperMotorClass class. This method has a default empty body for device class without attributes.
4. The *device_factory*() method of the StepperMotorClass class

This startup procedure is described in figure *6.2*

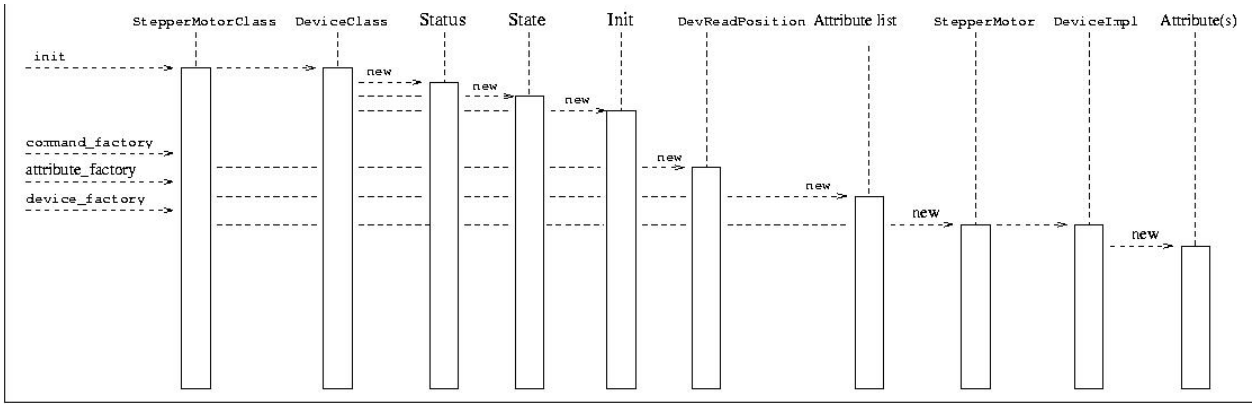


Fig. 7: Figure 6.2: Device pattern startup sequence

. The creation of the StepperMotorClass will automatically create an instance of the DeviceClass class. The constructor of the DeviceClass class will create the Status, State and Init command objects and store them in its command list.

The *command_factory*() method will simply create all the user defined commands and add them in the command list.

The *attribute_factory*() method will simply build a list of device attribute names.

The *device_factory*() method will create each StepperMotor object and store them in the StepperMotorClass instance device list. The list of devices to be created and their names is passed to the *device_factory* method in its input argument. StepperMotor is a sub-class of DeviceImpl class. Therefore, when a StepperMotor object is created, a DeviceImpl object is also created. The DeviceImpl constructor builds all the device attribute object(s) from the attribute list built by the *attribute_factory()* method.

Command execution sequence

The figure *6.3*

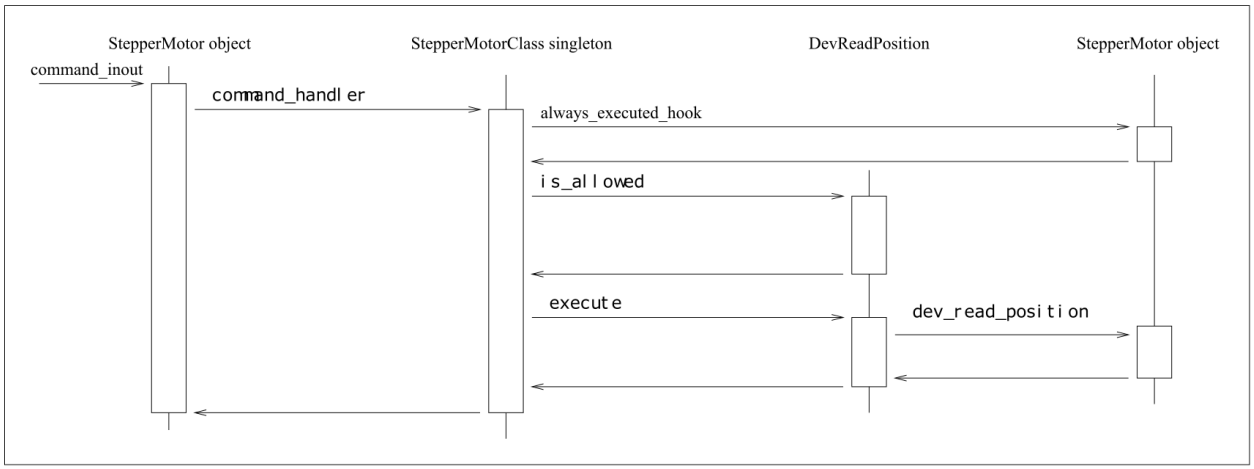


Fig. 8: Figure 6.3: Command execution timing

described how the method implementing a command is executed when a command_inout CORBA operation is requested by a client. The *command_inout* method of the StepperMotor object (inherited from the DeviceImpl class) is triggered by an instance of a class generated by the CORBA IDL compiler. This method calls the

command_handler() method of the StepperMotorClass object (inherited from the DeviceClass class). The *command_handler* method searches in its command list for the wanted command (using its name). If the command is found, the *always_executed_hook* method of the StepperMotor object is called. Then, the *is_allowed* method of the wanted command is executed. If the *is_allowed* method returns correctly, the *execute* method is executed. The *execute* method extracts the incoming data from the CORBA object use to transmit data over the network and calls the user written method which implements the command.

The automatically added commands

In order to increase the common behavior of every kind of devices in a TANGO control system, three commands are automatically added to each class of devices. These commands are :

- State
- Status
- Init

The default behavior of the method called by the State command depends on the device state. If the device state is ON or ALARM, the method will :

- read the attribute(s) with an alarm level defined
- check if the read value is above/below the alarm level and eventually change the device state to ALARM.
- returns the device state.

For all the other device state, the method simply returns the device state stored in the DeviceImpl class. Nevertheless, the method used to return this state (called *dev_state*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default State command and the state CORBA attribute is the ability of the State command to signal an error to the caller by throwing an exception.

The default behavior of the method called by the Status command depends on the device state. If the device state is ON or ALARM, the method returns the device status stored in the DeviceImpl class plus additional message(s) for all the attributes which are in alarm condition. For all the other device state, the method simply returns the device status as it is stored in the DeviceImpl class. Nevertheless, the method used to return this status (called *dev_status*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default Status command and the status CORBA attribute is the ability of the Status command to signal an error to the caller by throwing an exception.

The Init command is used to re-initialize a device without changing its network connection. This command calls the device *delete_device* method and the device *init_device* method. The rule of the *delete_device* method is to free memory allocated in the *init_device* method in order to avoid memory leak.

Reading/Writing attributes

Reading attributes

A Tango client is able to read Tango attribute(s) with the CORBA read_attributes call. Inside the device server, this call will trigger several methods of the device class (StepperMotor in our example) :

1. The *always_executed_hook()* method.
2. A method call *read_attr_hardware()*. This method is called one time per read_attributes CORBA call. The aim of this method is to read the device hardware and to store the result in a device class data member.
3. For each attribute to be read

1. A method called *is_<att name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some attributes which can be read only in some precise conditions. It has one parameter which is the request type (read or write)
2. A method called *read_<att name>()*. The aim of this method is to extract the real attribute value from the hardware read-out and to store the attribute value into the attribute object. It has one parameter which is a reference to the Attribute object to be read.

The figure *6.4* is a drawing of these method calls sequencing. For attribute always readable, a default *is_allowed* method is provided. This method always returns true.

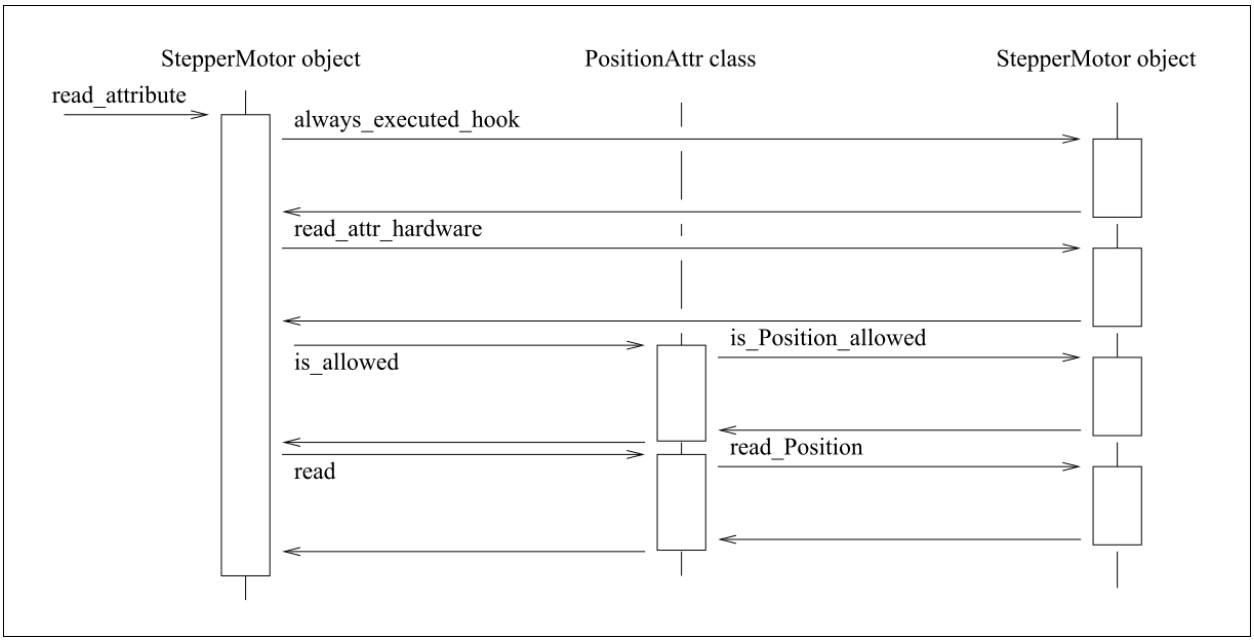


Fig. 9: Figure 6.4: Read attribute sequencing

Writing attributes

A Tango client is able to write Tango attribute(s) with the CORBA write_attributes call. Inside a device server, this call will trigger several methods of the device class (StepperMotor in our example)

1. The *always_executed_hook()* method.
2. For each attribute to be written
    1. A method called *is_<att name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some attributes which can be written only in some precise conditions. It has one parameter which is the request type (read or write)
    2. A method called *write_<att name>()*. It has one parameter which is a reference to the WAttribute object to be written. The aim of this method is to get the data to be written from the WAttribute object and to write this value into the corresponding hardware. If the hardware support writing several data in one go, code the hardware access in the *write_attr_harware()* method.
3. The write_attr_hardware() method. The rule of this method is to effectively write the hardware in case it is able to support writing several data in one go. If this is not the case, don't code this method (a default implementation is coded in the Tango base class) and code the real hardware access in each *write_<att name>()* method.

The figure *6.5* is a drawing of these method calls sequencing. For attribute always writeable, a default is_allowed method is provided. This method always allways returns true.

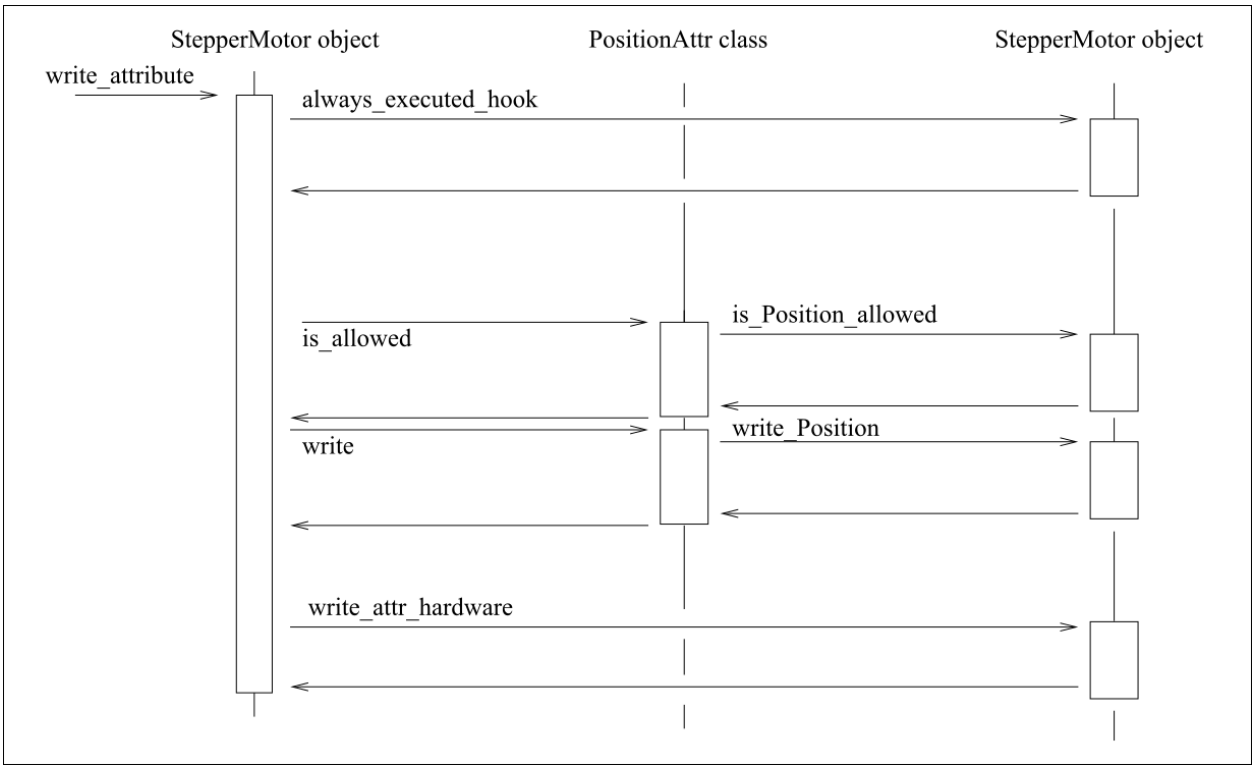


Fig. 10: Write attribute sequencing

The device server framework

Vocabulary

A device server pattern implementation is embedded in a process called a **device server**. Several instances of the same device server process can be used in a TANGO control system. To identify instances, a device server process is started with an **instance name** which is different for each instance. The device server name is the couple device server executable name/device server instance name. For instance, a device server started with the following command

Perkin id11

starts a device server process with an instance name id11, an executable name Perkin and a device server name Perkin/id11.

The DServer class

In order to simplify device server process administration, a device of the DServer class is automatically added to each device server process. Thus, every device server process supports the same set of administration commands. The implementation of this DServer class follows the device pattern and therefore, its device behaves like any other devices. The device name is

dserver/device server executable name/device server instance name

For instance, for the device server process described in chapter [Voc], the dserver device name is dserver/perkin/id11. This name is returned by the adm_name CORBA attribute available for every device. On top of the three automatically added commands, this device supports the following commands :

- DevRestart
- RestartServer
- QueryClass
- QueryDevice
- Kill
- AddLoggingTarget (C++ server only)
- RemoveLoggingTarget (C++ server only)
- GetLoggingTarget (C++ server only)
- GetLoggingLevel (C++ server only)
- SetLoggingLevel (C++ server only)
- StopLogging (C++ server only)
- StartLogging (C++ server only)
- PolledDevice
- DevPollStatus
- AddObjPolling
- RemObjPolling
- UpdObjPollingPeriod
- StartPolling
- StopPolling
- EventSubscriptionChange
- ZmqEventSubscriptionChange
- LockDevice
- UnLockDevice
- ReLockDevices
- DevLockStatus

These commands will be fully described later in this document.

Several controlled object classes can be embedded within the same device server process and it is the rule of this device to create all these device server patterns and to call their command and device factories as described in *Startup of a*

device pattern. The name and number of all the classes to be created is known to this device after the execution of a method called *class_factory*. It is the user responsibility to write this method.

The Tango::Util class

Description

This class merges a complete set of utilities in the same class. It is implemented as a singleton and there is only one instance of this class per device server process. It is mandatory to create this instance in order to run a device server. The description of all the methods implemented in this class can be found in [TangoRefMan].

Contents

Within this class, you can find :

- Static method to create/retrieve the singleton object
- Miscellaneous utility methods like getting the server output trace level, getting the CORBA ORB pointer, retrieving device server instance name, getting the server PID and more. Please, refer to [TangoRefMan] to get a complete list of all these utility methods.
- Method to create the device pattern implementing the DServer class (*server_init()*)
- Method to start the server (*server_run()*)
- TANGO database related methods

A complete device server

Within a complete device server, at least two implementations of the device server pattern are created (one for the dserver object and the other for the class of devices to control). On top of that, one instance of the Tango::Util class must also be created.

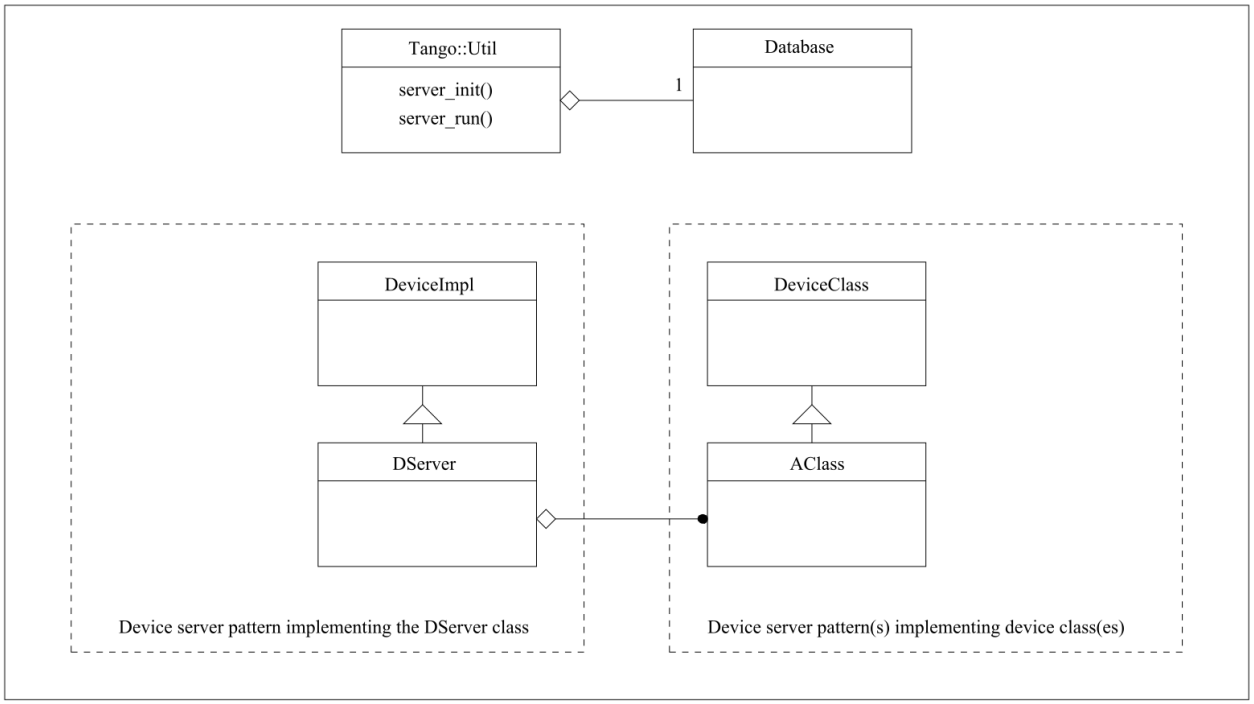


Fig. 11: Figure 6.6: A complete device server

A drawing of a complete device server is in figure *6.6*

Device server startup sequence

The device server startup sequence is the following :

1. Create an instance of the Tango::Util class. This will initialize the CORBA Object Request Broker
2. Called the *server_init* method of the Tango::Util instance The call to this method will :
    1. Create the DServerClass object of the device pattern implementing the DServer class. This will create the dserver object which during its construction will :
        1. Called the *class_factory* method of the DServer object. This method must create all the xxxClass instance for all the device pattern implementation embedded in the device server process.
        2. Call the *command_factory* and *device_factory* of all the classes previously created. The list of devices passed to each call to the *device_factory* method is retrieved from the TANGO database.
3. Wait for incoming request with the *server_run()* method of the Tango::Util class.

Exchanging data between client and server

Exchanging data between clients and server means most of the time passing data between processes running on different computer using the network. Tango limits the type of data exchanged between client and server and defines a way to exchange these data. This chapter details these features. Memory allocation and error reporting are also discussed.

All the rules described in this chapter are valid only for data exchanged between client and server. For device server internal data, classical C++ types can be used.

Command / Attribute data types

Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen out of a fixed set of 24 data types. Attributes support a sub-set of these data types (those are the data type with the (1) note) plus the DevEnum data type. The following table details type name, code and the corresponding CORBA IDL types.

The type name used in the type name column of this table is the C++ name. In the IDL file, all the Tango definition are grouped in a IDL module named Tango. The IDL module maps to C++ namespace. Therefore, all the data type are parts of a namespace called Tango.

Type name	IDL type
Tango::DevBoolean (1)	boolean
Tango::DevShort (1)	short
Tango::DevEnum (2)	short (See chapter on advanced features)
Tango::DevLong (1)	long
Tango::DevLong64 (1)	long long
Tango::DevFloat (1)	float
Tango::DevDouble (1)	double
Tango::DevUShort (1)	unsigned short
Tango::DevULong (1)	unsigned long
Tango::DevULong64 (1)	unsigned long long
Tango::DevString (1)	string
Tango::DevVarCharArray	sequence of unsigned char
Tango::DevVarShortArray	sequence of short
Tango::DevVarLongArray	sequence of long
Tango::DevVarLong64Array	sequence of long long
Tango::DevVarFloatArray	sequence of float
Tango::DevVarDoubleArray	sequence of double
Tango::DevVarUShortArray	sequence of unsigned short
Tango::DevVarULongArray	sequence of unsigned long
Tango::DevVarULong64Array	sequence of unsigned long long
Tango::DevVarStringArray	sequence of string
Tango::DevVarLongStringArray	structure with a sequence of long and a sequence of string
Tango::DevVarDoubleStringArray	structure with a sequence of double and a sequence of string
Tango::DevState (1)	enumeration
Tango::DevEncoded (1)	structure with a string and a sequence of char

The CORBA Interface Definition Language uses a type called **sequence** for variable length array. The Tango::DevUxxx types are used for unsigned types. The Tango::DevVarxxxxArray must be used when the data to be transferred are variable length array. The Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray are structures with two fields which are variable length array of Tango long (32 bits) and variable length array of strings for the Tango::DevVarLongStringArray and variable length array of double and variable length array of string for the Tango::DevVarDoubleStringArray. The Tango::State type is used by the State command to return the device state.

Using data types with C++

Unfortunately, the mapping between IDL and C++ was defined before the C++ class library had been standardized. This explains why the standard C++ string class or vector classes are not used in the IDL to C++ mapping.

TANGO commands/attributes argument types can be grouped on five groups depending on the IDL data type used. These groups are :

1. Data type using basic types (Tango::DevBoolean, Tango::DevShort, Tango::DevEnum, Tango::DevLong, Tango::DevFloat, Tango::DevDouble, Tango::DevUshort and Tango::DevULong)
2. Data type using strings (Tango::DevString type)
3. Data types using sequences (Tango::DevVarxxxArray types except Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray)
4. Data types using structures (Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray types)
5. Data type using IDL enumeration (Tango::DevState type)

In the following sub chapters, only summaries of the IDL to C++ mapping are given. For a full description of the C++ mapping, please refer to [Henning].

Basic types

For these types, the mapping between IDL and C++ is obvious and defined in the following table.

Tango type name	IDL type	C++	typedef
Tango::DevBoolean	boolean	CORBA::Boolean	unsigned char
Tango::DevShort	short	CORBA::Short	short
Tango::DevEnum	short	CORBA::Short	
Tango::DevLong	long	CORBA::Long	int
Tango::DevLong64	long long	CORBA::LongLong	long long or long (64 bits chip)
Tango::DevFloat	float	CORBA::Float	float
Tango::DevDouble	double	CORBA::Double	double
Tango::DevUShort	unsigned short	CORBA::UShort	unsigned short
Tango::DevULong	unsigned long	CORBA::ULong	unsigned long
Tango::DevULong64	unsigned long long	CORBA:ULongLong	unsigned long long or unsigned long (64 bits chip)

The types defined in the column named C++ should be used for a better portability. All these types are defined in the CORBA namespace and therefore their qualified names is CORBA::xxx. The Tango data type DevEnum is a special case described in detail in the chapter about advanced features.

Strings

Strings are mapped to **char ***. The use of *new* and *delete* for dynamic allocation of strings is not portable. Instead, you must use helper functions defined by CORBA (in the CORBA namespace) and Tango. These functions are :

```
1  char *CORBA::string_alloc(unsigned long len);
2  char *Tango::string_dup(const char *);
3  void Tango::string_free(char *);
```

These functions handle dynamic memory for strings. The *string_alloc* function allocates one more byte than requested by the len parameter (for the trailing 0). The function *string_dup* combines the allocation and copy. Both *string_alloc* and *string_dup* return a null pointer if allocation fails. The *string_free* function must be used to free memory allocated with *string_alloc* and *string_dup*. Calling *string_free* for a null pointer is safe and does nothing. *Tango::string_free* is available only since cppTango 9.3.3. *Tango::string_dup* is available only since Tango 9. If you are using an older version of the Tango C++ library, you should use *CORBA::string_free* and *CORBA::string_dup* instead. The following code fragment is an example of the Tango::DevString type usage :

```
1  Tango::DevString str = CORBA::string_alloc(5);
2  strcpy(str,"TANGO");
3
4  Tango::DevString str1 = Tango::string_dup("Do you want to danse TANGO?");
5
6  Tango::string_free(str);
7  Tango::string_free(str1);
```

Line 1-2 : TANGO is a five letters string. The CORBA::string_alloc function parameter is 5 but the function allocates 6 bytes

Line 4 : Example of the Tango::string_dup function

Line 6-7 : Memory deallocation

Sequences

IDL sequences are mapped to C++ classes that behave like vectors with a variable number of elements. Each IDL sequence type results in a separate C++ class. Within each class representing a IDL sequence types, you find the following method (only the main methods are related here) :

1. Four constructors.
    1. A default constructor which creates an empty sequence.
    2. The maximum constructor which creates a sequence with memory allocated for at least the number of elements passed as argument. This does not limit the number of element in the sequence but only the way how memory is allocated to store element
    3. A sophisticated constructor where it is possible to assign the memory used by the sequence with a preallocated buffer.
    4. A copy constructor which does a deep copy
2. An assignment operator which does a deep copy
3. A *length* accessor which simply returns the current number of elements in the sequence
4. A *length* modifier which changes the length of the sequence (which is different than the number of elements in the sequence)
5. Overloading of the [] operator. The subscript operator [] provides access to the sequence element. For a sequence containing elements of type T, the [] operator is overloaded twice to return value of type T & and const T &. Insertion into a sequence using the [] operator for the const T & make a deep copy. Sequence are numbered between 0 and *length*() -1.

Note that using the maximum constructor will not prevent you from setting the length of the sequence with a call to the length modifier. The following code fragment is an example of how to use a Tango::DevVarLongArray type

```
1   Tango::DevVarLongArray *mylongseq_ptr;
2   mylongseq_ptr = new Tango::DevVarLongArray();
3   mylongseq_ptr->length(4);
4
5   (*mylongseq_ptr)[0] = 1;
6   (*mylongseq_ptr)[1] = 2;
7   (*mylongseq_ptr)[2] = 3;
8   (*mylongseq_ptr)[3] = 4;
9
10  // (*mylongseq_ptr)[4] = 5;
11
12  CORBA::Long nb_elt = mylongseq_ptr->length();
13
14  mylongseq_ptr->length(5);
15  (*mylongseq_ptr)[4] = 5;
16
17  for (int i = 0;i < mylongseq_ptr->length();i++)
18       cout << "Sequence elt " << i + 1 << " = " << (*mylongseq_ptr)[i] << endl;
```

Line 1 : Declare a pointer to Tango::DevVarLongArray type which is a sequence of long

Line 2 : Create an empty sequence

Line 3 : Change the length of the sequence to 4

Line 5 - 8 : Initialize sequence elements

Line 10 ; Oups !!! The length of the sequence is 4. The behavior of this line is undefined and may be a core can be dumped at run time

Line 12 : Get the number of element actually stored in the sequence

Line 14-15 : Grow the sequence to five elements and initialize element number 5

Line 17-18 : Print sequence element

Another example for the Tango::DevVarStringArray type is given

```
1   Tango::DevVarStringArray mystrseq(4);
2   mystrseq.length(4);
3
4   mystrseq[0] = Tango::string_dup("Rock and Roll");
5   mystrseq[1] = Tango::string_dup("Bossa Nova");
6   mystrseq[2] = Tango::string_dup("Waltz");
7   mystrseq[3] = Tango::string_dup("Tango");
8
9   CORBA::Long nb_elt = mystrseq.length();
10
11  for (int i = 0;i < mystrseq.length();i++)
12       cout << "Sequence elt " << i + 1 << " = " << mystrseq[i] << endl;
```

Line 1 : Create a sequence using the maximum constructor

Line 2 : Set the sequence length to 4. This is mandatory even if you used the maximum constructor.

Line 4-7 : Populate the sequence

Line 9 : Get how many strings are stored into the sequence

Line 11-12 : Print sequence elements.

Structures

Only three TANGO types are defined as structures. These types are the Tango::DevVarLongStringArray, the Tango::DevVarDoubleStringArray and the Tango::DevEncoded data type. IDL structures map to C++ structures with corresponding members. For the Tango::DevVarLongStringArray, the two members are named *svalue* for the sequence of strings and *lvalue* for the sequence of longs. For the Tango::DevVarDoubleStringArray, the two structure members are called *svalue* for the sequence of strings and *dvalue* for the sequence of double. For the Tango::DevEncoded, the two structure members are called *encoded_format* for a string describing the data coding and *encoded_data* for the data themselves. The encoded_data field type is a Tango::DevVarCharArray. An example of the usage of the Tango::DevVarLongStringArray type is detailed below.

```
1  Tango::DevVarLongStringArray my_vl;
2
3  myvl.svalue.length(2);
4  myvl.svalue[0] = CORBA_string_dup("Samba");
5  myvl.svalue[1] = CORBA_string_dup("Rumba");
6
7  myvl.lvalue.length(1);
8  myvl.lvalue[0] = 10;
```

Line 1 : Declaration of the structure

Line 3-5 : Initialization of two strings in the sequence of string member

Line 7-8 : Initialization of one long in the sequence of long member

The DevState data type

The Tango::DevState data type is used to transfer device state between client and server. It is a IDL enumeration. IDL enumerated types map to C++ enumerations (amazing no!) with a trailing dummy enumerator to force enumeration to be a 32 bit type. The first enumerator will have the value 0, the next one will have the value 1 and so on.

```
1  Tango::DevState state;
2
3  state = Tango::ON;
4  state = Tango::FAULT;
```

Passing data between client and server

In order to have one definition of the CORBA operation used to send a command to a device whatever the command data type is, TANGO uses CORBA IDL **any** object. The IDL type *any* provides a universal type that can hold a value of arbitrary IDL types. Type *any* therefore allows you to send and receive values whose types are not fixed at compile time.

Type *any* is often compared to a void * in C. Like a pointer to void, an *any* value can denote a datum of any type. However, there is an important difference; whereas a void * denotes a completely untyped value that can be interpreted only with advance knowledge of its type, values of type *any* maintain type safety. For example, if a sender places a string value into an *any*, the receiver cannot extract the string as a value of the wrong type. Attempt to read the contents of an *any* as the wrong type cause a run-time error.

Internally, a value of type *any* consists of a pair of values. One member of the pair is the actual value contained inside the *any* and the other member of the pair is the type code. The type code is a description of the value's type. The type description is used to enforce type safety when the receiver extracts the value. Extraction of the value succeeds only if the receiver extracts the value as a type that matches the information in the type code.

Within TANGO, the command input and output parameters are objects of the IDL *any* type. Only insertion/extraction of all types defined as command data types is possible into/from these *any* objects.

C++ mapping for IDL any type

The IDL any maps to the C++ class **CORBA::Any**. This class contains a large number of methods with mainly methods to insert/extract data into/from the any. It provides a default constructor which builds an any which contains no value and a type code that indicates "no value". Such an any must be used for command which does not need input or output parameter. The operator **<<=** is overloaded many times to insert data into an any object. The operator **>>=** is overloaded many times to extract data from an any object.

Inserting/Extracting TANGO basic types

The insertion or extraction of TANGO basic types is straight forward using the <<= or >>= operators. Nevertheless, the Tango::DevBoolean type is mapped to a unsigned char and other IDL types are also mapped to char C++ type (The unsigned is not taken into account in the C++ overloading algorithm). Therefore, it is not possible to use operator overloading for these IDL types which map to C++ char. For the Tango::DevBoolean type, you must use the *CORBA::Any::from_boolean* or *CORBA::Any::to_boolean* intermediate objects defined in the CORBA::Any class.

Inserting/Extracting TANGO strings

The <<= operator is overloaded for const char * and always makes a deep copy. This deep copy is done using the Tango::*string_dup* function. The extraction of strings uses the >>= overloaded operator. The main point is that the Any object retains ownership of the string, so the returned pointer points at memory inside the Any. This means that you must not deallocate the extracted string and you must treat the extracted string as read-only.

Inserting/Extracting TANGO sequences

Insertion and extraction of sequences also uses the overloaded <<= and >>= operators. The insertion operator is overloaded twice: once for insertion by reference and once for insertion by pointer. If you insert a value by reference, the insertion makes a deep copy. If you insert a value by pointer, the Any assumes the ownership of the pointed-to memory.

Extraction is always by pointer. As with strings, you must treat the extracted pointer as read-only and must not deallocate it because the pointer points at memory internal to the Any.

Inserting/Extracting TANGO structures

This is identical to inserting/extracting sequences.

Inserting/Extracting TANGO enumeration

This is identical to inserting/extracting basic types

```
1   CORBA::Any a;
2   Tango::DevLong l1,l2;
3   l1 = 2;
4   a <<= l1;
5   a >>= l2;
6
7   CORBA::Any b;
8   Tango::DevBoolean b1,b2;
9   b1 = true;
10  b <<= CORBA::Any::from_boolean(b1);
11  b >>= CORBA::Any::to_boolean(b2);
12
13  CORBA::Any s;
14  Tango::DevString str1,str2;
15  str1 = "I like dancing TANGO";
16  s <<= str1;
17  s >>= str2;
18
19 //   Tango::string_free(str2);
20 //   a <<= Tango::string_dup("Oups");
21
22  CORBA::Any seq;
23  Tango::DevVarFloatArray fl_arr1;
24  fl_arr1.length(2);
25  fl_arr1[0] = 1.0;
26  fl_arr1[1] = 2.0;
27  seq <<= fl_arr1;
28  const Tango::DevVarFloatArray *fl_arr_ptr;
29  seq >>= fl_arr_ptr;
30
31 //   delete fl_arr_ptr;
```

Line 1-5 : Insertion and extraction of Tango::DevLong type

Line 7-11 Insertion and extraction of Tango::DevBoolean type using the CORBA::Any::from_boolean and CORBA::Any::to_boolean intermediate structure

Line 13-17 : Insertion and extraction of Tango::DevString type

Line 19 : Wrong ! You should not deallocate a string extracted from an any

Line 20 : Wrong ! Memory leak because the <<= operator will do the copy.

Line 22-29 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference and the use of the <<= operator makes a deep copy of the sequence. Therefore, after line 27, it is possible to deallocate the sequence

Line 31: Wrong.! You should not deallocate a sequence extracted from an any

The insert and extract methods of the Command class

In order to simplify the insertion/extraction into/from Any objects, small helper methods have been written in the Command class. The signatures of these methods are :

```
1          void extractextract(const CORBA::Any &,<Tango type> &);
2          CORBA::Any *insertinsert(<Tango type>);
```

An *extract* method has been written for all Tango types. These method extract the data from the Any object passed as parameter and throw an exception if the Any data type is incompatible with the awaiting type. An *insert* method have been written for all Tango types. These method create an Any object, insert the data into the Any and return a pointer to the created Any. For Tango types mapped to sequences or structures, two *insert* methods have been written: one for the insertion from pointer and the other for the insertion from reference. For Tango strings, two *insert* methods have been written: one for insertion from a classical Tango::DevString type and the other from a const Tango::DevString type. The first one deallocate the memory after the insert into the Any object. The second one only inserts the string into the Any object.

The previous example can be rewritten using the insert/extract helper methods (We suppose that we can use the Command class insert/extract methods)

```
1     Tango::DevLong l1,l2;
2     l1 = 2;
3     CORBA::Any *a_ptr = insert(l1);
4     extract(*a_ptr,l2);
5
6     Tango::DevBoolean b1,b2;
7     b1 = true;
8     CORBA::Any *b_ptr = insert(b1);
9     extract(*b_ptr,b2);
10
11    Tango::DevString str1,str2;
12    str1 = "I like dancing TANGO";
13    CORBA::Any *s_ptr = insert(str1);
14    extract(*s_ptr,str2);
15
16    Tango::DevVarFloatArray fl_arr1;
17    fl_arr1.length(2);
18    fl_arr1[0] = 1.0;
19    fl_arr1[1] = 2.0;
20    insert(fl_arr1);
21    CORBA::Any *seq_ptr = insert(fl_arr1);
22    Tango::DevVarFloatArray *fl_arr_ptr;
23    extract(*seq_ptr,fl_arr_ptr);
```

Line 1-4 : Insertion and extraction of Tango::DevLong type

Line 6-9 : Insertion and extraction of Tango::DevBoolean type

Line 11-14 : Insertion and extraction of Tango::DevString type

Line 16-23 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference which makes a deep copy of the sequence. Therefore, after line 20, it is possible to deallocate the sequence

C++ memory management

The rule described here are valid for variable length command data types like Tango::DevString or all the Tango::DevVarxxxxArray types.

The method executing the command must allocate the memory used to pass data back to the client or use static memory (like buffer declares as object data member. If necessary, the ORB will deallocate this memory after the data have been sent to the caller. Fortunately, for incoming data, the method have no memory management responsibilities. The details about memory management given in this chapter assume that the insert/extract methods of the Tango::Command class are used and only the method in the device object is discussed.

For string

Example of a method receiving a Tango::DevString and returning a Tango::DevString is detailed just below

```
1  Tango::DevString MyDev::dev_string(Tango::DevString argin)
2  {
3      Tango::DevString        argout;
4
5      cout << "the received string is " << argin << endl;
6
7      string str("Am I a good Tango dancer ?");
8      argout = new char[str.size() + 1];
9      strcpy(argout,str.c_str());
10
11     return argout;
12 }
```

Note that there is no need to deallocate the memory used by the incoming string. Memory for the outgoing string is allocated at line 8, then it is initialized at the following line. The memory allocated at line 8 will be automatically freed by the usage of the *Command::insert()* method. Using this schema, memory is allocated/freed each time the command is executed. For constant string length, a statically allocated buffer can be used.

```
1  Tango::ConstDevString MyDev::dev_string(Tango::DevString argin)
2  {
3      Tango::ConstDevString   argout;
4
5      cout << "the received string is " << argin << endl;
6
7      argout = "Hello world";
8      return argout;
9  }
```

A Tango::ConstDevString data type is used. It is not a new data Tango data type. It has been introduced only to allows *Command::insert()* method overloading. The argout pointer is initialized at line 7 with memory statically allocated. In this case, no memory will be freed by the *Command::insert()* method. There is also no memory copy in the contrary of the previous example. A buffer defined as object data member can also be used to set the argout pointer.

For array/sequence

Example of a method returning a Tango::DevVarLongArray is detailed just below

```
1  Tango::DevVarLongArray *MyDev::dev_array()
2  {
3      Tango::DevVarLongArray  *argout  = new Tango::DevVarLongArray();
4
5      long output_array_length = ...;
6      argout->length(output_array_length);
7      for (int i = 0;i < output_array_length;i++)
8          (*argout)[i] = i;
9
10     return argout;
11 }
```

In this case, memory is allocated at line 3 and 6. Then, the sequence is populated. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of long data is declared as device data member and named buffer.

```
1  Tango::DevVarLongArray *MyDev::dev_array()
2  {
3      Tango::DevVarLongArray  *argout;
4
5      long output_array_length = ...;
6      argout = create_DevVarLongArray(buffer,output_array_length);
7      return argout;
8  }
```

At line 3 only a pointer to a DevVarLongArray is defined. This pointer is set at line 6 using the *create_DevVarLongArray()* method. This method will create a sequence using this buffer without memory allocation and with minimum copying. The *Command::insert()* method used here is the same than the one used in the previous example. The sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer. The following create_xxx methods are defined in the DeviceImpl class :

Method name	data type
create_DevVarCharArray()	unsigned char
create_DevVarShortArray()	short
create_DevVarLongArray()	DevLong
create_DevVarLong64Array()	DevLong64
create_DevVarFloatArray()	float
create_DevVarDoubleArray()	double
create_DevVarUShortArray()	unsigned short
create_DevVarULongArray()	DevULong
create_DevVarULong64Array()	DevULong64

For string array/sequence

Example of a method returning a Tango::DevVarStringArray is detailed just below

```
1  Tango::DevVarStringArray *MyDev::dev_str_array()
2  {
3     Tango::DevVarStringArray *argout  = new Tango::DevVarStringArray();
4
5     argout->length(3);
6     (*argout)[0] = Tango::string_dup("Rumba");
7     (*argout)[1] = Tango::string_dup("Waltz");
8     string str("Jerck");
9     (*argout)[2] = Tango::string_dup(str.c_str());
10    return argout;
11 }
```

Memory is allocated at line 3 and 5. Then, the sequence is populated at lines 6,7 and 9. The usage of the *Tango::string_dup* function also allocates memory. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. For portability reason, the ORB uses the *CORBA::string_free* function to free the memory allocated for each string. This is why the corresponding *Tango::string_dup* or *CORBA::string_alloc* function must be used to reserve this memory.It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of pointer to char is declared as device data member and named int_buffer.

```
1  Tango::DevVarStringArray *DocDs::dev_str_array()
2  {
3     int_buffer[0] = "first";
4     int_buffer[1] = "second";
5
6     Tango::DevVarStringArray *argout;
7     argout = create_DevVarStringArray(int_buffer,2);
8     return argout;
9  }
```

The intermediate buffer is initialized with statically allocated memory at lines 3 and 4. The returned sequence is created at line 7 with the *create_DevVarStringArray()* method. Like for classical array, the sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer.

For Tango composed types

Tango supports only two composed types which are Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray. These types are translated to C++ structure with two sequences. It is not possible to use memory statically allocated for these types. Each structure element must be initialized as described in the previous sub-chapters using the dynamically allocated memory case.

Reporting errors

Tango uses the C++ try/catch plus exception mechanism to report errors. Two kind of errors can be transmitted between client and server :

1. CORBA system error. These exceptions are raised by the ORB and indicates major failures (A communication failure, An invalid object reference...)
2. CORBA user exception. These kind of exceptions are defined in the IDL file. This allows an exception to contain an arbitrary amount of error information of arbitrary type.

TANGO defines one user exception called **DevFailed**. This exception is a variable length array of **DevError** type (a sequence of DevError). The DevError type is a four fields structure. These fields are :

1. A string describing the type of the error. This string replaces an error code and allows a more easy management of include files.
2. The error severity. It is an enumeration with the three values which are WARN, ERR or PANIC.
3. A string describing in plain text the reason of the error
4. A string describing the origin of the error

The Tango::DevFailed type is a sequence of DevError structures in order to transmit to the client what is the primary error reason when several classes are used within a command. The sequence element 0 must be the DevError structure describing the primary error. A method called *print_exception*() defined in the Tango::Except class prints the content of exception (CORBA system exception or Tango::DevFailed exception). Some static methods of the Tango::Except class called *throw_exception*() can be used to throw Tango::DevFailed exception. Some other static methods called *re_throw_exception()* may also be used when the user want to add a new element in the exception sequence and re-throw the exception. Details on these methods can be found in [TangoRefMan].

Example of throwing exception

This example is a piece of code from the *command_handler*() method of the DeviceImpl class. An exception is thrown to the client to indicate that the requested command is not defined in the command list.

```
    TangoSys_OMemStream o;

    o << "Command " << command << " not found" << ends;
    Tango::Except::throw_exception("API_CommandNotFound",
                                   o.str(),
                                   "DeviceClass::command_handler");


    try
    {
        .....
    }
```

(continues on next page)

(continued from previous page)

```
13      catch (Tango::DevFailed &e)
14      {
15          TangoSys_OMemStream o;
16
17          o << "Command " << command << " not found" << ends;
18          Tango::Except::re_throw_exception(e,
19                                      "API_CommandNotFound",
20                                      o.str(),
21                                      "DeviceClass::command_handler");
22      }
```

Line 1 : Build a memory stream. Use the TangoSys_MemStream because memory streams are not managed the same way between Windows and Unix

Line 3 : Build the reason string in the memory stream

Line 4-5 : Throw the exception to client using one of the *throw_exception* static method of the Except class. This throw_exception method used here allows the definition of the error type string, the reason string and the origin string of the DevError structure. The remaining DevError field (the error severity) will be set to its default value. Note that the first and third parameters are casted to a *const char **. Standard C++ defines that such a string is already a *const char ** but the GNU C++ compiler (release 2.95) does not use this type inside its function overloading but rather uses a *char ** which leads to calling the wrong function.

Line 13-22 : Re-throw an already catched tango::DevFailed exception with one more element in the exception sequence.

The Tango Logging Service

A first introduction about this logging service has been done in chapter [sec:The-Tango-Logging]

The TANGO Logging Service (TLS) gives the user the control over how much information is actually generated and to where it goes. In practice, the TLS allows to select both the logging level and targets of any device within the control system.

Logging Targets

The TLS implementation allows each device logging requests to print simultaneously to multiple destinations. In the TANGO terminology, an output destination is called a **logging target**. Currently, targets exist for console, file and log consumer device.

CONSOLE: logs are printed to the console (i.e. the standard output),

FILE: logs are stored in a XML file. A rolling mechanism is used to backup the log file when it reaches a certain size (see below),

DEVICE: logs are sent to a device implementing a well known TANGO interface (see section [sec:Tango-log-consumer] for a definition of the log consumer interface). One implementation of a log consumer associated to a graphical user interface is available within the Tango package. It is called the LogViewer.

The device's logging behavior can be control by adding and/or removing targets.

Note : When the size of a log file (for file logging target) reaches the so-called rolling-file-threshold (rft), it is backuped as current_log_file_name + _1 and a new current_log_file_name is opened. Obviously, there is only one backup file at a time (i.e. any existing backup is destroyed before the current log file is backuped). The default threshold is 20 Mb, the minimum is 500 Kb and the maximum is 1000 Mb.

Logging Levels

Devices can be assigned a logging level. It acts as a filter to control the kind of information sent to the targets. Since, there are (usually) much more low level log statements than high level statements, the logging level also control the amount of information produced by the device. The TLS provides the following levels (semantic is just given to be indicative of what could be log at each level):

OFF: Nothing is logged

FATAL: A fatal error occurred. The process is about to abort

ERROR: An (unrecoverable) error occurred but the process is still alive

WARN: An error occurred but could be recovered locally

INFO: Provides information on important actions performed

DEBUG: Generates detailed information describing the internal behavior of a device

Levels are ordered the following way:

DEBUG < INFO < WARN < ERROR < FATAL < OFF

For a given device, a level is said to be enabled if it is greater or equal to the logging level assigned to this device. In other words, any logging request which level is lower than the device's logging level is ignored.

Note: The logging level can't be controlled at target level. The device's targets shared the same device logging level.

Sending TANGO Logging Messages

Logging macros in C++

The TLS provides the user with easy to use C++ macros with *printf* and *stream* like syntax. For each logging level, a macro is defined in both styles:

- LOG_{FATAL, ERROR, WARN, INFO or DEBUG}
- {FATAL, ERROR, WARN, INFO or DEBUG}_STREAM

These macros are supposed to be used within the device's main implementation class (i.e. the class that inherits (directly or indirectly) from the Tango::DeviceImpl class). In this context, they produce logging messages containing the device name. In other words, they automatically identify the log source. Section [sub:C++-logging-in] gives a trick to log in the name of device outside its main implementation class. Printf like example:

LOG_DEBUG((Msg#%d - Hello world, i++));

Stream like example:

DEBUG_STREAM << Msg# << i++ << - Hello world << endl;

These two logging requests are equivalent. Note the double parenthesis in the printf version.

C++ logging in the name of a device

A device implementation is sometimes spread over several classes. Since all these classes implement the same device, their logging requests should be associated with this device name. Unfortunately, the C++ logging macros can't be used because they are outside the device's main implementation class. The Tango::LogAdapter class is a workaround for this limitation.

Any method not member of the device's main implementation class, which send log messages associated to a device must be a member of a class inheriting from the Tango::LogAdapter class. Here is an example:

```
class MyDeviceActualImpl: public Tango::LogAdapter
{
public :
   MyDeviceActualImpl(...,Tango::DeviceImpl *device,...)
   :Tango::LogAdpater(device)
   {
         ....
//
// The following log is associated to the device passed to the constructor
//
        DEBUG_STREAM << "In MyDeviceActualImpl constructor" << endl;

        ....
   }
};
```

Writing a device server process

Writing a device server can be made easier by adopting the correct approach. This chapter will describe how to write a device server process. It is divided into the following parts : understanding the device, defining device commands/attributes/pipes, choosing device state and writing the necessary classes. All along this chapter, examples will be given using the stepper motor device server. Writing a device server for our stepper motor example device means writing :

- The *main* function
- The *class_factory* method (only for C++ device server)
- The *StepperMotorClass* class
- The *DevReadPositionCmd* and *DevReadDirectionCmd* classes
- The *PositionAttr*, *SetPositionAttr* and *DirectionAttr* classes
- The *StepperMotor* class.

All these functions and classes will be detailed. The stepper motor device server described in this chapter supports 2 commands and 3 attributes which are :

- Command DevReadPosition implemented using the inheritance model
- Command DevReadDirection implemented using the template command model
- Attribute Position (position of the first motor). This attribute is readable and is linked with a writable attribute (called SetPosition). When the value of this attribute is requested by the client, the value of the associated writable attribute is also returned.
- Attribute SetPosition (writable attribute linked with the Position attribute). This attribute has some properties with user defined default value.

- Attribute Direction (direction of the first motor)

As the reader will understand during the reading of the following sub-chapters, the command and attributes classes (*DevReadPositionCmd*, *DevReadDirectionCmd*, *PositionAttr*, *SetPositionAttr* and *DirectionAttr*) are very simple classes. A tool called **Pogo** has been developped to automatically generate/maintain these classes and to write part of the code needed in the remaining one. See *Pogo manual* to know more on this Pogo tool.

In order to also gives an example of how the database objects part of the Tango device pattern could be used, our device have two properties. These properties are of the Tango long data types and are named "Max" and "Min".

Understanding the device

The first step before writing a device server is to develop an understanding of the hardware to be programmed. The Equipment Responsible should have description of the hardware and its operating modes (manuals, spec sheets etc.). The Equipment Responsible must also provide specifications of what the device server should do. The Device Server Programmer should demand an exact description of the registers, alarms, interlocks and any timing constraints which have to be kept. It is very important to have a good understanding of the device interfacing before starting designing a new class.

Once the Device Server Programmer has understood the hardware the next important step is to define what is a logical device i.e. what part of the hardware will be abstracted out and treated as a logical device. In doing so the following points of the TDSOM should be kept in mind

- Each device is known and accessed by its ascii name.
- The device is exported onto the network to be imported by applications.
- Each device belongs to a class.
- A list of commands exists per device.
- Applications use the device server api to execute commands on a device.

The above points have to be taken into account when designing the level of device abstraction. The definition of what is a device for a certain hardware is primarily the job of the Device Server Programmer and the Applications Programmer but can also involve the Equipment Responsible. The Device Server Programmer should make sure that the Applications Programmer agrees with her definition of what is a device.

Here are some guidelines to follow while defining the level of device abstraction -

- **efficiency**, make sure that not a too fine level of device abstraction has been chosen. If possible group as many attributes together to form a device. Discuss this with the Applications Programmer to find out what is efficient for her application.
- **hardware independency**, one of the main reasons for writing device servers is to provide the Applications Programmer with a *software* interface as opposed to a *hardware* interface. Hide the hardware structure of the device. For example if the user is only interested in a single channel of a multichannel device then define each channel to be a logical device. The user should not be aware of hardware addresses or cabling details. The user is very often a scientist who has a physics-oriented world view and not a hardware-oriented world view. Hardware independency also has the advantage that applications are immune to hardware changes to the device
- **object oriented world view**, another *raison d'etre* behind the device server model is to build up an object oriented view of the world. The device should resemble the user's view of the object as closely as possible. In the case of the ESRF's beam lines for example, the devices should resemble beam line scientist's view of the machine.
- **atomism**, each device can be considered like an atom - is a independent object. It should appear independent to the client even if behind the scenes it shares some hardware or software with other objects. This is often the case with multichannel devices where the user would like to see each channel as a device but it is obvious that the channels cannot be programmed completely independently. The logical device is there to hide or make

transparent this fact. If it is impossible to send commands to one device without modifying another device then a single device should be made out the two devices.

- **tailored** *vs* **general**, one of the philosophies of the TDSOM is to provide tailored solutions. For example instead of writing one *serial line* class which treats the general case of a serial line device and leaving the device protocol to be implemented in the client the TDSOM advocates implementing a device class which handles the protocol of the device. This way the client only has to know the commands of the class and not the details of the protocol. Nothing prevents the device class from using a general purpose serial line class if it exists of course.

Defining device commands

Each device has a list of commands which can be executed by the application across the network or locally. These commands are the Application Programmer's network knobs and dials for interacting with the device.

The list of commands to be implemented depends on the capabilities of the hardware, the list of sensible functions which can be executed at a distance and of course the functionality required by the application. This implies a close collaboration between the Equipment Responsible, Device Server Programmer and the Application Programmer.

When drawing up the list of commands particular attention should be paid to the following points

- **performance**, no single command should monopolize the device server for a long time (a nominal value for long is one second). Commands should be implemented in such a way that it executes immediately returning with a response. At best try to keep command execution time down to less than the typical overhead of an rpc call i.e. som milliseconds. This of course is not always possible e.g. a serial line device could require 100 milliseconds of protocol exchange. The Device Server Programmer should find the best trade-off between the users requirements and the devices capabilities. If a command implies a sequence of events which could last for a long time then implement the sequence of events in another thread - don't block the device server.
- **robustness**, should be provided which allow the client to recover from error conditions and or do a warm startup.

Standard commands

A minimum set of three commands exist for all devices. These commands are

- State which returns the state of a device
- Status which returns the status of the device as a formatted ascii string
- Init which re-initialize a device without changing its network connection

These commands have already been discussed in [Auto_cmd]

Choosing device state

The device state is a number which reflects the availability of the device. To simplify the coding for generic application, a predefined set of states are supported by TANGO. This list has 14 members which are

State name
ON
OFF
CLOSE
OPEN
INSERT
EXTRACT
MOVING
STANDBY
FAULT
INIT
RUNNING
ALARM
DISABLE
UNKNOWN

The names used here have obvious meaning.

Device server utilities to ease coding/debugging

The device server framework supports one set of utilities to ease the process of coding and debugging device server code. This utility is :

1. The device server verbose option

Using this facility avoids the usage of the classical "#ifdef DEBUG" style which makes code less readable.

The device server verbose option

Each device server supports a verbose option called **-v**. Four verbose levels are defined from 1 to 4. Level 4 is the most talkative one. If you use the -v option without specifying level, level 4 will be assumed.

Since Tango release 3, a Tango Logging Service has been introduced (detailed in chapter [The-Tango-Logging chapter]). This -v option set-up the logging service. If it used, it will automatically add a *console* target to all devices embedded within the device server process. Level 1 and 2 will set the logging level to all devices embedded within the device server to INFO. Level 3 and 4 will set the logging level to all devices embedded within the device server to DEBUG. All messages sent by the API layer are associated to the administration device.

C++ utilities to ease device server coding

Some utilities functions have been added in the C++ release to ease Tango device server development. These utilities allow the user to

- Init a C++ vector from a data of one of the Tango DevVarXXXArray data types
- Init a data of one of the Tango::DevVarxxxArray data type from a C++ vector
- Print a data of one of Tango::DevVarxxxArray data type

They mainly used the "<<" operator overloading features. The following code lines are an example of usage of these utilities.

```
1   vector<string> v1;
2   v1.push_back("one");
3   v1.push_back("two");
4   v1.push_back("three");
5
6   Tango::DevVarStringArray s;
7   s << v1;
8   cout << s << endl;
9
10  vector<string> v2;
11  v2 << s;
12
13  for (int i = 0;i < v2.size();i++)
14     cout << "vector element = " << v2[i] << endl;
```

Line 1-4 : Create and Init a C++ string vector

Line 7 : Init a Tango::DevVarStringArray data from the C++ vector

Line 8 : Print all the Tango::DevVarStringArray element in one line of code.

Line 11 : Init a second empty C++ string vector with the content of the Tango::DevVarStringArray

Line 13-14 : Print vector element

Warning: Note that due to a strange behavior of the Windows VC++ compiler compared to other compilers, to use these utilities with the Windows VC++ compiler, you must add the line "using namespace tango" at the beginning of your source file.

Avoiding name conflicts

Namespace are used to avoid name conflicts. Each device pattern implementation is defined within its own namespace. The name of the namespace is the device pattern class name. In our example, the namespace name is *StepperMotor*.

The device server main function

A device server main function (or method) always follows the same framework. It exactly implements all the action described in chapter [Server_startup]. Even if it could be always the same, it has not been included in the library because some linkers are perturbed by the presence of two main functions.

```
1   #include <tango.h>
2
3   int main(int argc,char *argv[])
4   {
5
6       Tango::Util *tg;
7
8       try
9       {
10
11          tg = Tango::Util::init(argc,argv);
12
```

(continues on next page)

(continued from previous page)

```
13          tg->server_init();
14
15          cout << "Ready to accept request" << endl;
16          tg->server_run();
17      }
18      catch (bad_alloc)
19      {
20           cout << "Can't allocate memory!!!" << endl;
21           cout << "Exiting" << endl;
22      }
23      catch (CORBA::Exception &e)
24      {
25           Tango::Except::print_exception(e);
26
27           cout << "Received a CORBA::Exception" << endl;
28           cout << "Exiting" << endl;
29      }
30
31      tg->server_cleanup();
32
33      return(0);
34  }
```

Line 1 : Include the **tango.h** file. This file is a master include file. It includes several other files. The list of files included by tango.h can be found in [TangoRefMan]

Line 11 : Create the instance of the Tango::Util class (a singleton). Passing argc,argv to this method is mandatory because the device server command line is checked when the Tango::Util object is constructed.

Line 13 : Start all the device pattern creation and initialization with the *server_init()* method

Line 16 : Put the server in a endless waiting loop with the *server_run()* method. In normal case, the process should never returns from this line.

Line 18-22 : Catch all exceptions due to memory allocation error, display a message to the user and exit

Line 23 : Catch all standard TANGO exception which could occur during device pattern creation and initialization

Line 25 : Print exception parameters

Line 27-28 : Print an additional message

Line 31 : Cleanup the server before exiting by calling the *server_cleanup()* method.

The DServer::class_factory method

As described in chapter [DServer_class], C++ device server needs a *class_factory*() method. This method creates all the device pattern implemented in the device server by calling their *init*() method. The following is an example of a *class_factory* method for a device server with one implementation of the device server pattern for stepper motor device.

```
1  #include <tango.h>
2  #include <steppermotorclass.h>
3
4  void Tango::DServer::class_factory()
5  {
```

(continues on next page)

(continued from previous page)

```
6
7       add_class(StepperMotor::StepperMotorClass::init("StepperMotor"));
8
9   }
```

Line 1 : Include the Tango master include file

Line 2 : Include the steppermotorclass class definition file

Line 7 : Create the StepperMotorClass singleton by calling its *init* method and stores the returned pointer into the DServer object. Remember that all classes for the device pattern implementation for the stepper motor class is defined within a namespace called *StepperMotor*.

Writing the StepperMotorClass class

The class declaration file

```
1   #include <tango.h>
2
3   namespace StepperMotor
4   {
5
6   class StepperMotorClass : public Tango::DeviceClass
7   {
8   public:
9       static StepperMotorClass *init(const char *);
10      static StepperMotorClass *instance();
11      ~StepperMotorClass() {_instance = NULL;}
12
13  protected:
14      StepperMotorClass(string &);
15      static StepperMotorClass *_instance;
16      void command_factory();
17      void attribute_factory(vector<Tango::Attr *> &);
18
19  public:
20      void device_factory(const Tango::DevVarStringArray *);
21  };
22
23  } /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file

Line 3 : This class is defined within the *StepperMotor* namespace

Line 6 : Class StepperMotorClass inherits from Tango::DeviceClass

Line 9-10 : Definition of the *init* and *instance* methods. These methods are static and can be called even if the object is not already constructed.

Line 11: The destructor

Line 14 : The class constructor. It is protected and can't be called from outside the class. Only the *init* method allows a user to create an instance of this class. See [Patterns] to get details about the singleton design pattern.

Line 15 : The instance pointer. It is static in order to set it to NULL during process initialization phase

Line 16 : Definition of the *command_factory* method

Line 17 : Definition of the *attribute_factory* method

Line 20 : Definition of the *device_factory* method

The singleton related methods

```
1  #include <tango.h>
2
3  #include <steppermotor.h>
4  #include <steppermotorclass.h>
5
6  namespace StepperMotor
7  {
8
9  StepperMotorClass *StepperMotorClass::_instance = NULL;
10
11 StepperMotorClass::StepperMotorClass(string &s):
12 Tango::DeviceClass(s)
13 {
14     INFO_STREAM << "Entering StepperMotorClass constructor" << endl;
15
16     INFO_STREAM << "Leaving StepperMotorClass constructor" << endl;
17 }
18
19
20 StepperMotorClass *StepperMotorClass::init(const char *name)
21 {
22     if (_instance == NULL)
23     {
24           try
25           {
26                string s(name);
27                _instance = new StepperMotorClass(s);
28           }
29           catch (bad_alloc)
30           {
31                throw;
32           }
33     }
34     return _instance;
35 }
36
37 StepperMotorClass *StepperMotorClass::instance()
38 {
39     if (_instance == NULL)
40     {
41           cerr << "Class is not initialised !!" << endl;
42           exit(-1);
43     }
```

(continues on next page)

(continued from previous page)

```
44      return _instance;
45  }
```

Line 1-4 : include files: the Tango master include file (tango.h), the StepperMotorClass class definition file (steppermotorclass.h) and the StepperMotor class definition file (steppermotor.h)

Line 6 : Open the *StepperMotor* namespace.

Line 9 : Initialize the static _instance field of the StepperMotorClass class to NULL

Line 11-18 : The class constructor. It takes an input parameter which is the controlled device class name. This parameter is passed to the constructor of the DeviceClass class. Otherwise, the constructor does nothing except printing a message

Line 20-35 : The *init* method. This method needs an input parameter which is the controlled device class name (StepperMotor in this case). This method checks is the instance is already constructed by testing the _instance data member. If the instance is not constructed, it creates one. If the instance is already constructed, the method simply returns a pointer to it.

Line 37-45 : The *instance* method. This method is very similar to the *init* method except that if the instance is not already constructed. the method print a message and abort the process.

As you can understand, it is not possible to construct more than one instance of the StepperMotorClass (it is a singleton) and the *init* method must be called prior to any other method.

The command_factory method

Within our example, the stepper motor device supports two commands which are called DevReadPosition and DevReadDirection. These two command takes a Tango::DevLong argument as input and output parameter. The first command is created using the inheritance model and the second command is created using the template command model.

```
1   void StepperMotorClass::command_factory()
2   {
3           command_list.push_back(new DevReadPositionCmd("DevReadPosition",
4                                                         Tango::DEV_LONG,
5                                                         Tango::DEV_LONG,
6                                                         "Motor number (0-7)",
7                                                         "Motor position"));
8
9           command_list.push_back(
10              new TemplCommandInOut<Tango::DevLong,Tango::DevLong>
11                  ((const char *)"DevReadDirection",
12                   static_cast<Tango::Lg_CmdMethPtr_Lg>
13                          (&StepperMotor::dev_read_direction),
14                   static_cast<Tango::StateMethPtr>
15                          (&StepperMotor::direct_cmd_allowed))
16                                );
17  }
```

Line 4 : Creation of one instance of the DevReadPositionCmd class. The class is created with five arguments which are the command name, the command type code for its input and output parameters and two strings which are the command input and output parameters description. The pointer returned by the new C++ keyword is added to the vector of available command.

Line 10-14 : Creation of the object used for the DevReadDirection command. This command has one input and output parameter. Therefore the created object is an instance of the TemplCommandInOut class. This class is a C++ template class. The first template parameter is the command input parameter type, the second template parameter is the command output parameter type. The second TemplCommandInOut class constructor parameter (set at line 13) is a pointer to the method to be executed when the command is requested. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class[3]. The third TemplCommandInOut class constructor parameter (set at line 15) is a pointer to the method to be executed to check if the command is allowed. This is necessary only if the default behavior (command always allowed) does not fulfill the needs. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class. When a command is created using the template command method, the input and output parameters type are determined from the template C++ class parameters.

The device_factory method

The *device_factory* method has one input parameter. It is a pointer to Tango::DevVarStringArray data which is the device name list for this class and the instance of the device server process. This list is fetch from the Tango database.

```
1  void StepperMotorClass::device_factory(const Tango::_DevVarStringArray *devlist_ptr)
2  {
3
4      for (long i = 0;i < devlist_ptr->length();i++)
5      {
6           DEBUG_STREAM << "Device name : " << (*devlist_ptr)[i] << endl;
7
8           device_list.push_back(new StepperMotor(this,(*devlist_ptr)[i]));       9
9           if (Tango::Util::_UseDb == true)
10               export_device(device_list.back());
11          else
12               export_device(device_list.back(),(*devlist_ptr[i]));
13     }
14 }
```

Line 4 : A loop for each device

Line 8 : Create the device object using a StepperMotor class constructor which needs two arguments. These two arguments are a pointer to the StepperMotorClass instance and the device name. The pointer to the constructed object is then added to the device list vector

Line 10-13 : Export device to the outside world using the *export_device* method of the DeviceClass class.

The attribute_factory method

The rule of this method is to fulfill a vector of pointer to attributes. A reference to this vector is passed as argument to this method.

```
1  void StepperMotorClass::attribute_factory(vector<Tango::Attr *> &att_list)
2  {
3      att_list.push_back(new PositionAttr());
4
5      Tango::UserDefaultAttrProp def_prop;
6      def_prop.set_label("Set the motor position");
7      def_prop.set_format("scientific;setprecision(4)");
```

(continues on next page)

[3] The StepperMotor class inherits from the DeviceImpl class and therefore is a DeviceImpl

(continued from previous page)

```
8       Tango::Attr *at = new SetPositionAttr();
9       at->set_default_properties(def_prop);
10      att_list.push_back(at);
11
12      att_list.push_back(new DirectcionAttr());
13  }
```

Line 3 : Create the PositionAttr class and store the pointer to this object into the attribute pointer vector.

Line 5-7 : Create a Tango::UserDefaultAttrProp instance and set the label and format properties default values in this object

Line 8 : Create the SetPositionAttr attribute.

Line 9 : Set attribute user default value with the *set_default_properties()* method of the Tango::Attr class.

Line 10 : Store the pointer to this object into the attribute pointer vector.

Line 12 : Create the DirectionAttr class and store the pointer to this object into the attribute pointer vector.

Please, note that in some rare case, it is necessary to add attribute to this list during the device server life cycle. This *attribute_factory()* method is called once during device server start-up. A method *add_attribute()* of the DeviceImpl class allows the user to add a new attribute to the attribute list outside of this *attribute_factory()* method. See [TangoRefMan] for more information on this method.

The DevReadPositionCmd class

The class declaration file

```
1   #include <tango.h>
2
3   namespace StepperMotor
4   {
5
6   class DevReadPositionCmd : public Tango::Command
7   {
8   public:
9       DevReadPositionCmd(const char *,Tango::CmdArgType,
10                          Tango::CmdArgType,
11                          const char *,const char *);
12      ~DevReadPositionCmd() {};
13
14      virtual bool is_allowed (Tango::DeviceImpl *, const CORBA::Any &);
15      virtual CORBA::Any *execute (Tango::DeviceImpl *, const CORBA::Any &);
16  };
17
18  } /* End of StepperMotor namespace */
```

Line 1 : Include the tango master include file

Line 3 : Open the *StepperMotor* namespace.

Line 6 : The DevReadPositionCmd class inherits from the Tango::Command class

Line 9 : The constructor

Line 12 : The destructor

Line 14 : The definition of the *is_allowed* method. This method is not necessary if the default behavior implemented by the default *is_allowed* method fulfill the requirements. The default behavior is to always allows the command execution (always return true).

Line 15: The definition of the *execute* method

The class constructor

The class constructor does nothing. It simply invoke the Command constructor by passing it its five arguments which are:

1. The command name
2. The command input type code
3. The command output type code
4. The command input parameter description
5. The command output parameter description

With this 5 parameters command class constructor, the command display level is not specified. Therefore it is set to its default value (OPERATOR). If the command does not have input or output parameter, it is not possible to use the Command class constructor defined with five parameters. In this case, the command constructor execute the Command class constructor with three elements (class name, input type, output type) and set the input or output parameter description fields with the *set_in_type_desc* or *set_out_type_desc* Command class methods. To set the command display level, it is possible to use a 6 parameters constructor or it is also possible to set it in the constructor code with the *set_disp_level* method. Many Command class constructors are defined. See [TangoRefMan] for a complete list.

The is_allowed method

In our example, the DevReadPosition command is allowed only if the device is in the ON state. This method receives two argument which are a pointer to the device object on which the command must be executed and a reference to the command input Any object. This method returns a boolean which must be set to true if the command is allowed. If this boolean is set to false, the DeviceClass *command_handle*r method will automatically send an exception to the caller.

```
1  bool DevReadPositionCmd::is_allowed(Tango::DeviceImpl *device,
2                                      const CORBA::Any &in_any)
3  {
4       if (device->get_state() == Tango::ON)
5            return true;
6       else
7            return false;
8  }
```

Line 4 : Call the *get_state* method of the DeviceImpl class which simply returns the device state

Line 5 : Authorize command if the device state is ON

Line 7 : Refuse command execution in all other cases.

The execute method

This method receives two arguments which are a pointer to the device object on which the command must be executed and a reference to the command input Any object. This method returns a pointer to an any object which must be initialized with the data to be returned to the caller.

```
1  CORBA::Any *DevReadPositionCmd::execute(
2                          Tango::DeviceImpl *device,
3                          const CORBA::Any &in_any)
4  {
5       INFO_STREAM << "DevReadPositionCmd::execute(): arrived" << endl;
6       Tango::DevLong motor;
7
8       extract(in_any,motor);
9       return insert(
10         (static_cast<StepperMotor *>(device))->dev_read_position(motor));
11 }
```

Line 8 : Extract incoming data from the input any object using a Command class *extract* helper method. If the type of the data in the Any object is not a Tango::DevLong, the *extract* method will throw an exception to the client.

Line 9 : Call the stepper motor object method which execute the DevReadPosition command and insert the returned value into an allocated Any object. The Any object allocation is done by the *insert* method which return a pointer to this Any.

The PositionAttr class

The class declaration file

```
1  #include <tango.h>
2  #include <steppermotor.h>
3
4  namespace StepperMotor
5  {
6
7
8  class PositionAttr: public Tango::Attr
9  {
10 public:
11     PositionAttr():Attr("Position",
12                         Tango::DEV_LONG,
13                         Tango::READ_WITH_WRITE,
14                         "SetPosition") {};
15     ~PositionAttr() {};
16
17     virtual void read(Tango::DeviceImpl *dev,Tango::Attribute &att)
18     {(static_cast<StepperMotor *>(dev))->read_Position(att);}
19     virtual bool is_allowed(Tango::DeviceImpl *dev,Tango::AttReqType ty)
20     {return (static_cast<StepperMotor *>(dev))->is_Position_allowed(ty);}
21 };
22
23 } /* End of StepperMotor namespace */
```

(continues on next page)

(continued from previous page)

```
24
25  #endif // _STEPPERMOTORCLASS_H
```

Line 1-2 : Include the tango master include file and the steppermotor class definition include file

Line 4 : Open the *StepperMotor* namespace.

Line 8 : The PosiitionAttr class inherits from the Tango::Attr class

Line 11-14 : The constructor with 4 arguments

Line 15 : The destructor

Line 17 : The definition of the *read* method. This method forwards the call to a StepperMotor class method called *read_Position()*

Line 19 : The definition of the *is_allowed* method. This method is not necessary if the default behaviour implemented by the default *is_allowed* method fulfills the requirements. The default behaviour is to always allows the attribute reading (always return true). This method forwards the call to a StepperMotor class method called *is_Position_allowed()*

The class constructor

The class constructor does nothing. It simply invoke the Attr constructor by passing it its four arguments which are:

1. The attribute name
2. The attribute data type code
3. The attribute writable type code
4. The name of the associated write attribute

With this 4 parameters Attr class constructor, the attribute display level is not specified. Therefore it is set to its default value (OPERATOR). To set the attribute display level, it is possible to use in the constructor code the *set_disp_level* method. Many Attr class constructors are defined. See [TangoRefMan] for a complete list.

This Position attribute is a scalar attribute. For spectrum attribute, instead of inheriting from the Attr class, the class must inherits from the SpectrumAttr class. Many SpectrumAttr class constructors are defined. See [TangoRefMan] for a complete list.

For Image attribute, instead of inheriting from the Attr class, the class must inherits from the ImageAttr class. Many ImageAttr class constructors are defined. See [TangoRefMan] for a complete list.

The is_allowed method

This method receives two argument which are a pointer to the device object to which the attribute belongs to and the type of request (read or write). In the PositionAttr class, this method simply forwards the request to a method of the StepperMotor class called *is_Position_allowed()* passing the request type to this method. This method returns a boolean which must be set to true if the attribute is allowed. If this boolean is set to false, the DeviceImpl read_attribute method will automatically send an exception to the caller.

The read method

This method receives two arguments which are a pointer to the device object to which the attribute belongs to and a reference to the corresponding attribute object. This method forwards the request to a StepperMotor class called *read_Position()* passing it the reference on the attribute object.

The StepperMotor class

The class declaration file

```
#include <tango.h>

#define AGSM_MAX_MOTORS 8 // maximum number of motors per device

namespace StepperMotor
{

class StepperMotor: public TANGO_BASE_CLASS
{
public :
    StepperMotor(Tango::DeviceClass *,string &);
    StepperMotor(Tango::DeviceClass *,const char *);
    StepperMotor(Tango::DeviceClass *,const char *,const char *);
    ~StepperMotor() {};

    DevLong dev_read_position(DevLong);
    DevLong dev_read_direction(DevLong);
    bool direct_cmd_allowed(const CORBA::Any &);

    virtual Tango::DevState dev_state();
    virtual Tango::ConstDevString dev_status();

    virtual void always_executed_hook();

    virtual void read_attr_hardware(vector<long> &attr_list);
    virtual void write_attr_hardware(vector<long> &attr_list);

    void read_position(Tango::Attribute &);
    bool is_Position_allowed(Tango::AttReqType req);
    void write_SetPosition(Tango::WAttribute &);
    void read_Direction(Tango::Attribute &);

    virtual void init_device();
    virtual void delete_device();

    void get_device_properties();

protected :
    long axis[AGSM_MAX_MOTORS];
    DevLong position[AGSM_MAX_MOTORS];
    DevLong direction[AGSM_MAX_MOTORS];
```

(continues on next page)

(continued from previous page)

```
42      long state[AGSM_MAX_MOTORS];
43
44      Tango::DevLong *attr_Position_read;
45      Tango::DevLong *attr_Direction_read;
46      Tango::DevLong attr_SetPosition_write;
47
48      Tango::DevLong min;
49      Tango::DevLong max;
50
51      Tango::DevLong *ptr;
52  };
53
54  } /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file

Line 5 : Open the *StepperMotor* namespace.

Line 8 : The StepperMotor class inherits from a Tango base class

Line 11-13 : Three different object constructors

Line 14 : The destructor which calls the *delete_device()* method

Line 16 : The method to be called for the execution of the DevReadPosition command. This method must be declared as virtual if it is needed to redefine it in a class inheriting from StepperMotor. See chapter [Inheriting] for more details about inheriting.

Line 17 : The method to be called for the execution of the DevReadDirection command

Line 18 : The method called to check if the execution of the DevReadDirection command is allowed. This method is necessary because the DevReadDirection command is created using the template command method and the default behavior is not acceptable

Line 20 : Redefinition of the *dev_state*. This method is used by the State command

Line 21 : Redefinition of the *dev_status*. This method is used by the Status command

Line 23 : Redefinition of the *always_executed_hook* method. This method is the place to code mandatory action which must be executed prior to any command.

Line 25-31 : Attribute related methods

Line 32 : Definition of the *init_device* method.

Line 33 : Definition of the *delete_device* method

Line 35 : Definition of the *get_device_properties* method

Line 38-50 : Data members.

Line 43-44 : Pointers to data for readable attributes Position and Direction

Line 45 : Data for the SetPosition attribute

Line 47-48 : Data members for the two device properties

The constructors

Three constructors are defined here. It is not mandatory to defined three constructors. But at least one is mandatory. The three constructors take a pointer to the StepperMotorClass instance as first parameter[4]. The second parameter is the device name as a C++ string or as a classical pointer to char array. The third parameter necessary only for the third form of constructor is the device description string passed as a classical pointer to a char array.

```
 #include <tango.h>
 #include <steppermotor.h>

 namespace StepperMotor
 {

 StepperMotor::StepperMotor(Tango::DeviceClass *cl,string &s)
 :TANGO_BASE_CLASS(cl,s.c_str())
 {
   init_device();
}

StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s)
:TANGO_BASE_CLASS(cl,s)
{
   init_device();
}

StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s,const char *d)
:TANGO_BASE_CLASS(cl,s,d)
{
   init_device();
}

void StepperMotor::init_device()
{
   cout << "StepperMotor::StepperMotor() create " << device_name << endl;

   long i;

   for (i=0; i< AGSM_MAX_MOTORS; i++)
   {
      axis[i] = 0;
      position[i] = 0;
      direction[i] = 0;
   }

   ptr = new Tango::DevLong[10];

   get_device_properties();
}

void StepperMotor::delete_device()
{
   delete [] ptr;
```

(continues on next page)

[4] The StepperMotorClass inherits from the DeviceClass and therefore is a DeviceClass

(continued from previous page)

```
46 }
```

Line 1-2 : Include the Tango master include file (tango.h) and the StepperMotor class definition file (steppermotor.h)

Line 4 : Open the *StepperMotor* namespace

Line 7-11 : The first form of the class constructor. It execute the Tango base class constructor with the two parameters. Note that the device name passed to this constructor as a C++ string is passed to the Tango::DeviceImpl constructor as a classical C string. Then the *init_device* method is executed.

Line 13-17 : The second form of the class constructor. It execute the Tango base class constructor with its two parameters. Then the *init_device* method is executed.

Line 19-23: The third form of constructor. Again, it execute the Tango base class constructor with its three parameters. Then the *init_device* method is executed.

Line 25-41 : The *init_device* method. All the device data initialization is done in this method. The device properties are also retrieved from database with a call to the *get_device_properties* method at line 40. The device data member called *ptr* is initialized with allocated memory at line 38. It is not needed to have this pointer, it has been added only for educational purpose.

Line 43-46 : The *delete_device* method. The rule of this method is to free memory allocated in the *init_device* method. In our case , only the device data member *ptr* is allocated in the *init_device* method. Therefore, its memory is freed at line 45. This method is called by the automatically added Init command before it calls the *init_device* method. It is also called by the device destructor.

The methods used for the DevReadDirection command

The DevReadDirection command is created using the template command method. Therefore, there is no specific class needed for this command but only one object of the TemplCommandInOut class. This command needs two methods which are the *dev_read_direction* method and the *direct_cmd_allowed* method. The *direct_cmd_allowed* method defines here implements exactly the same behavior than the default one. This method has been used only for pedagogic issue. The *dev_read_direction* method will be executed by the *execute* method of the TemplCommandInOut class. The *direct_cmd_allowed* method will be executed by the *is_allowed* method of the TemplCommandInOut class.

```
1  DevLong StepperMotor::dev_read_direction(DevLong axis)
2  {
3     if (axis < 0 || axis > AGSM_MAX_MOTORS)
4     {
5         WARNING_STREAM << "Steppermotor::dev_read_direction(): axis out of range !";
6         WARNING_STREAM << endl;
7         TangoSys_OMemStream o;
8
9         o << "Axis number " << axis << " out of range" << ends;
10        throw_exception("StepperMotor_OutOfRange",
11                        o.str(),
12                        "StepperMotor::dev_read_direction");
13     }
14
15     return direction[axis];
16 }
17
18
19 bool StepperMotor::direct_cmd_allowed(const CORBA::Any &in_data)
```

(continues on next page)

(continued from previous page)

```
20  {
21      INFO_STREAM << "In direct_cmd_allowed() method" << endl;
22
23      return true;
24  }
```

Line 1-16 : The *dev_read_direction* method

Line 5-12 : Throw exception to client if the received axis number is out of range

Line 7 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 19-24 : The *direct_cmd_allowed* method. The command input data is passed to this method in case of it is needed to take the decision. This data is still packed into the CORBA Any object.

The methods used for the Position attribute

To enable reading of attributes, the StepperMotor class must re-define two or three methods called *read_attr_hardware(), read_<Attribute_name>()* and if necessary a method called *is_<Attribute_name>_allowed()*. The aim of the first one is to read the hardware. It will be called only once at the beginning of each read_attribute CORBA call. The second method aim is to build the exact data for the wanted attribute and to store this value into the Attribute object. Special care has been taken in order to minimize the number of data copy and allocation. The data passed to the Attribute object as attribute value is passed using pointers. It must be allocated by the method[5] and the Attribute object will not free this memory. Data members called attr_<Attribute_name>_read are foreseen for this usage. The *read_attr_hardware()* method receives a vector of long which are indexes into the main attributes vector of the attributes to be read. The *read_Position()* method receives a reference to the Attribute object. The third method (*is_Position_allowed()*) aim is to allow or dis-allow, the attribute reading. In some cases, some attributes can be read only if some conditions are met. If this method returns true, the *read_<Attribute_name>()* method will be called. Otherwise, an error will be generated for the attribute. This method receives one argument which is an emumeration describing the attribute request type (read or write). In our example, the reading of the Position attribute is allowed only if the device state is ON.

```
1   void StepperMotor::read_attr_hardware(vector<long> &attr_list)
2   {
3      INFO_STREAM << "In read_attr_hardware for " << attr_list.size();
4      INFO_STREAM << " attribute(s)" << endl;
5
6      for (long i = 0;i < attr_list.size();i++)
7      {
8         string attr_name;
9         attr_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();
10
11        if (attr_name == "Position")
12        {
13           attr_Position_read = &(position[0]);
14        }
15        else if (attr_name == "Direction")
```

(continues on next page)

[5] It can also be data declared as object data members or memory declared as static

(continued from previous page)

```
16          {
17              attr_Direction_read = &(direction[0]);
18          }
19      }
20  }
21
22  void read_Position(Tango::Attribute &att)
23  {
24      att.set_value(attr_Position_read);
25  }
26
27  bool is_Position_allowed(Tango::AttReqType req)
28  {
29      if (req == Tango::WRITE_REQ)
30          return false;
31      else
32      {
33          if (get_state() == Tango::ON)
34              return true;
35          else
36              return false;
37      }
38  }
```

Line 6 : A loop on each attribute to be read

Line 9 : Get attribute name

Line 11 : Test on attribute name

Line 13 : Read hardware (pretty simple in our case)

Line 24 : Set attribute value in Attribute object using the *set_value()* method. This method will also initializes the attribute quality factor to Tango::ATTR_VALID if no alarm level are defined and will set the attribute returned date. It is also possible to use a method called *set_value_date_quality()* which allows the user to set the attribute quality factor as well as the attribute date.

Line 33 : Test on device state

The methods used for the SetPosition attribute

To enable writing of attributes, the StepperMotor class must re-define one or two methods called *write_<Attribute_name>()* and if necessary a method called *is_<Attribute_name>_allowed()*. The aim of the first one is to write the hardware. The *write_Position()* method receives a reference to the WAttribute object. The value to write is in this WAttribute object. The third method (*is_Position_allowed()*) aim is to allow or dis-allow, the attribute writing. In some cases, some attributes can be write only if some conditions are met. If this method returns true, the *write_<Attribute_name>()* method will be called. Otherwise, an error will be generated for the attribute. This method receives one argument which is an emumeration describing the attribute request type (read or write). For read/write attribute, this method is the same for reading and writing. The input argument value makes the difference.

For our example, it is always possible to write the SetPosition attribute. Therefore, the StepperMotor class only defines a *write_SetPosition()* method.

```
1  void StepperMotor::write_SetPosition(Tango::WAttribute &att)
2  {
3     att.get_write_value(sttr_SetPosition_write);
4
5     DEBUG_STREAM << "Attribute SetPosition value = ";
6     DEBUG_STREAM << attr_SetPosition_write << endl;
7
8     position[0] = attr_SetPosition_write;
9  }
10
11 void StepperMotor::write_attr_hardware(vector<long> &attr_list)
12 {
13
14 }
```

Line 3 : Retrieve new attribute value

Line 5-6 : Send some messages using Tango Logging system

Line 8 : Set the hardware (pretty simple in our case)

Line 11 - 14: The write_attr_hardware() method.
In our case, we don't have to do anything in the *write_attr_hardware()* method. It is coded here just for educational purpose. When its not needed, this method has a default implementation in the Tango base class and it is not mandatory to declare and defin it in your own Tango class

Retrieving device properties

Retrieving properties is fairly simple with the use of the database object. Each Tango device is an aggregate with a DbDevice object (see figure [Device pattern figure]). This has been grouped in a method called *get_device_properties*(). The classes and methods of the Dbxxx objects are described in the Tango API documentation.

```
1  void DocDs::get_device_property()
2  {
3     Tango::DbData   data;
4     data.push_back(DbDatum("Max"));
5     data.push_back(DbDatum("Min"));
6
7     get_db_device()->get_property(data);
8
9     if (data[0].is_empty()==false)
10       data[0]  >>  max;
11    if (data[1].is_empty()==false)
12       data[1]  >>  min;
13 }
```

Line 4-5 : Two DbDatum (one per property) are stored into a DbData object

Line 7 : Call the database to retrieve properties value

Line 9-10 : If the Max property is defined in the database, extract its value from the DbDatum object and store it in a device data member

Line 11-12 : If the Min property is defined in the database, extract its value from the DbDatum object and store it in a device data member

The remaining methods

The remaining methods are the *dev_state, dev_status, always_executed_hook*, *dev_read_position* and *read_Direction()* methods. The *dev_state* method parameters are fixed. It does not receive any input parameter and must return a Tango_DevState data type. The *dev_status* parameters are also fixed. It does not receive any input parameter and must return a Tango string. The *always_executed_hook* receives nothing and return nothing. The *dev_read_position* method input parameter is the motor number as a long and the returned parameter is the motor position also as a long data type. The *read_Direction()* method is the method for reading the Direction attribute.

```
DevLong StepperMotor::dev_read_position(DevLong axis)
{

   if (axis < 0 || axis > AGSM_MAX_MOTORS)
   {
        WARNING_STREAM << "Steppermotor::dev_read_position(): axis out of range !";
        WARNING_STREAM << endl;

        TangoSys_OMemStream o;

        o << "Axis number " << axis << " out of range" << ends;
        throw_exception("StepperMotor_OutOfRange",
                        o.str(),
                        "StepperMotor::dev_read_position");
   }

   return position[axis];
}

void always_executed_hook()
{
   INFO_STREAM << "In the always_executed_hook method << endl;
}

Tango_DevState StepperMotor::dev_state()
{
   INFO_STREAM << "In StepperMotor state command" << endl;
   return DeviceImpl::dev_state();
}

Tango_DevString StepperMotor::dev_status()
{
   INFO_STREAM << "In StepperMotor status command" << endl;
   return DeviceImpl::dev_status();
}

void read_Direction(Tango::Attribute att)
{
   att.set_value(attr_Direction_read);
}
```

Line 1-18 : The *dev_read_position* method

Line 6-14 : Throw exception to client if the received axis number is out of range

Line 9 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 20-23 : The *always_executed_hook* method. It does nothing. It has been included here only as pedagogic usage.

Line 25-29 : The *dev_state* method. It does exactly what the default *dev_state* does. It has been included here only as pedagogic usage

Line 31-35 : The *dev_status* method. It does exactly what the default *dev_statu*s does. It has been included here only as pedagogic usage

Line 37-40 : The *read_Direction* method. Simply set the Attribute object internal value

Device server under Windows

Two kind of programs are available under Windows. These kinds of programs are called console application or Windows application. A console application is started from a MS-DOS window and is very similar to classical UNIX program. A Windows application is most of the time not started from a MS-DOS window and is generally a graphical application without standard input/output. Writing a device server in a console application is straight forward following the rules described in the previous sub-chapters. Writing a device server in a Windows application needs some changes detailed in the following sub-chapters.

The Tango device server graphical interface

Within the Windows operating system, most of the running application has a window user interface. This is also true for the Windows Tango device server. Using or not this interface is up to the device server programmer. The choice is done with an argument to the *server_init()* method of the Tango::Util class. This interface is pretty simple and is based on three windows which are :

- The device server main window
- The device server console window
- The device server help window

The device server main window

This window looks like :

Fig. 12: Figure 6.7: Tango device server main window

Four menus are available in this window. The File menu allows the user to exit the device server. The View menu allows you to display/hide the device server console window. The Debug menu allows the user to change the server output verbose level. All the outputs goes to the console window even if it is hidden. The Help menu displays the help window. The device server name is displayed in the window title. The text displayed at the bottom of the window has a default value (the one displayed in this window dump) but may be changed by the device server programmer using the *set_main_window_text()* method of the Tango::Util class. If used, this method must be called prior to the call of the *server_init()* method. Refer to [TangoRefMan] for a complete description of this method.

The console window

This window looks like :

It simply displays all the logging** message when a console target is used in the device server.

The help window

This window looks like :

This window displays

- The device server name
- The Tango library release
- The Tango IDL definition release
- The device server release. The device server programmer may set this release number using the *set_server_version()* method of the Tango::Util class. If used, this must be done prior to the call of the *server_init()* method. If the *set_server_version()* method is not used, x.y is displays as version number. Refer to [TangoRefMan] for a complete description of this method.

MFC device server

There is no *main* function within a classical MFC program. Most of the time, your application is represented by one instance of a C++ class which inherits from the MFC CWinApp class. This CWinApp class has several methods that you may overload in your application class. For a device server to run correctly, you must overload two methods of the CWinApp class. These methods are the *InitInstance()* and *ExitInstance()* methods. The rule of these methods is obvious following their names.

Remember that if the Tango device server graphical user interface is used, you must link your device server with the Tango windows resource file. This is done by adding the Tango resource file to the Project Settings/Link/Input/Object, library modules window in VC++.

The InitInstance method

The code to be added here is the equivalent of the code written in a classical *main()* function. Don't forget to add the *tango.h* file in the list of included files.

```
BOOL FluidsApp::InitInstance()
{
   AfxEnableControlContainer();

// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

 #ifdef _AFXDLL
```

(continues on next page)

(continued from previous page)

```
11    Enable3dControls();          // Call this when using MFC in a shared DLL
12  #else
13    Enable3dControlsStatic();    // Call this when linking to MFC statically
14  #endif
15    Tango::Util *tg;
16    try
17    {
18
19        tg = Tango::Util::init(m_hInstance,m_nCmdShow);
20
21        tg->server_init(true);
22
23        tg->server_run();
24
25    }
26    catch (bad_alloc)
27    {
28        MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
29        return(FALSE);
30    }
31    catch (Tango::DevFailed &e)
32    {
33        MessageBox((HWND)NULL,,e.errors[0].desc.in(),"Command line",MB_ICONSTOP);
34        return(FALSE);
35    }
36    catch (CORBA::Exception &)
37    {
38        MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
39        return(FALSE);
40    }
41
42    m_pMainWnd = new CWnd;
43    m_pMainWnd->Attach(tg->get_ds_main_window());
44
45    return TRUE;
46  }
```

Line 19 : Initialise Tango system. This method also analises the argument used in command line.

Line 21 : Create Tango classes requesting the Tango Windows graphical interface to be used

Line 23 : Start Network listener. Note that under NT, this call returns in the contrary of UNIX like operating system.

Line 26-30 : Display a message box in case of memory allocation error and leave method with a return value set to false in order to stop the process

Line 31-35 : Display a message box in case of error during server initialization phase.

Line 36-40 : Display a message box in case of error other than memory allocation. Leave method with a return value set to false in order to stop the process.

Line 37-38 : Create a MFC main window and attach the Tango graphical interface main window to this MFC window.

The ExitInstance method

This method is called when the application is stopped. For Tango device server, its rule is to destroy the Tango::Util singleton if this one has been correctly constructed.

```
1  int FluidsApp::ExitInstance()
2  {
3     bool del = true;
4
5     try
6     {
7         Tango::Util *tg = Tango::Util::instance();
8     }
9     catch(Tango::DevFailed)
10    {
11        del = false;
12    }
13
14    if (del == true)
15        delete (Tango::Util::instance());
16
17    return CWinApp::ExitInstance();
18 }
```

Line 7 : Try to retrieve the Tango::Util singleton. If this one has not been constructed correctly, this call will throw an exception.

Line 9-12 : Catch the exception in case of incomplete Tango::Util singleton construction

Line 14-15 : Delete the Tango::Util singleton. This will unregister the Tango device server from the Tango database.

Line 17 : Execute the *ExitInstance* method of the CWinApp class.

If you don't want to use the Tango device server graphical interface, do not pass any parameter to the *server_init()* method and instead of the code display in lines 37 and 38 in the previous example of the *InitInstance()* method, use your own code to initialize your own application.

Example of how to build a Windows device server MFC based

This sub-chapter gives an example of what it is needed to do to build a MFC Windows device server. Rather than being a list of actions to strictly follow, this is some general rules of how using VC++ to build a Tango device server using MFC.

1. Create your device server using Pogo. For a class named MyMotor, the following files will be needed : *class_factory.cpp*, *MyMotorClass.h*, *MyMotorClass.cpp*, *MyMotor.h* and *MyMotor.cpp*.
2. On a Windows computer running VC++, create a new project of type "MFC app Wizard (exe)" using static MFC libs. Ask for a dialog based project without ActiveX controls.
3. Copy the five files generated by Pogo to the Windows computer and add them to your project
4. Remove the dialog window files (xxxDlg.cpp and xxxDlg.h), the Resource include file and the resource script file from your project
5. Add #include <stdafx.h> as first line of the include files list in *class_factory.cpp*, *MyMotorClass.cpp* and *MyMotor.cpp* file. Also add your own directory and the Tango include directory to the project pre-compiler include directories list.

6. Enable RTTI in your project settings (see chapter [Compiling NT])
7. Change your application class:
    1. Add the definition of an *ExitInstance* method in the declaration file. (xxx.h file)
    2. Remove the include of the dialog window file in the xxx.cpp file and add an include of the Tango master include files (tango.h)
    3. Replace the *InitInstance*() method as described in previous sub-chapter. (xx.cpp file)
    4. Add an *ExitInstance()* method as described in previous sub-chapter (xxx.cpp file)
8. Add all the libraries needed to compile a Tango device server (see chapter [Compiling NT]) and the Tango resource file to the linker Object/Libraries modules.

Win32 application

Even if it is more natural to use the C++ structure of the MFC class to write a Tango device server, it is possible to write a device server as a Win32 application. Instead of having a *main()* function as the application entry point, the operating system, provides a *WinMain()* function as the application entry point. Some code must be added to this *WinMain* function in order to support Tango device server. Don't forget to add the *tango.h* file in the list of included files. If you are using the project files generated by Pogo, don't forget to change the linker SUBSYSTEM option to Windows (Under Linker/System in the project properties window).

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
   MSG msg;
   Tango::Util *tg;

   try
   {
       tg = Tango::Util::init(hInstance,nCmdShow);

       string txt;
       txt = "Blabla first line\n";
       txt = txt + "Blabla second line\n";
       txt = txt + "Blabla third line\n";
       tg->set_main_window_text(txt);
       tg->set_server_version("2.2");

       tg->server_init(true);

       tg->server_run();

   }
   catch (bad_alloc)
   {
       MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
       return (FALSE);
   }
   catch (Tango::DevFailed &e)
```

(continues on next page)

(continued from previous page)

```
31      {
32          MessageBox((HWND)NULL,e.errors[0].desc.in(),"Command line",MB_ICONSTOP);
33          return (FALSE);
34      }
35      catch (CORBA::Exception &)
36      {
37          MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
38          return(FALSE);
39      }
40
41      while (GetMessage(&msg, NULL, 0, 0))
42      {
43          TranslateMessage(&msg);
44          DispatchMessage(&msg);
45      }
46
47      delete tg;
48
49      return msg.wParam;
50  }
```

Line 11 : Create the Tango::Util singleton

Line 13-18 : Set parameters for the graphical interface

Line 20 : Initialize Tango device server requesting the display of the graphical interface

Line 22 : Run the device server

Line 25-39 : Display a message box for all the kinds of error during Tango device server initialization phase and exit WinMain function.

Line 41-45 : The Windows message loop

Line 47 : Delete the Tango::Util singleton. This class destructor unregisters the device server from the Tango database.

Remember that if the Tango device server graphical user interface is used, you must add the Tango windows resource file to your project.

If you don't want to use the tango device server graphical user interface, do not use any parameter in the call of the *server_init()* method and do not link your device server with the Tango Windows resource file.

Device server as service

With Windows, if you want to have processes which survive to logoff sequence and/or are automatically started during computer startup sequence, you have to write them as service. It is possible to write Tango device server as service. You need to

1. Write a class which inherits from a pre-written Tango class called NTService. This class must have a *start* method.
2. Write a main function following a predefined skeleton.

The service class

It must inherits from the *NTService* class and defines a *start* method. The NTService class must be constructed with one argument which is the device server executable name. The *start* method has three arguments which are the number of arguments passed to the method, the argument list and a reference to an object used to log info in the NT event system. The first two args must be passed to the Tango::Util::init method and the last one is used to log error or info messages. The class definition file looks like

```
1  #include <tango.h>
2  #include <ntservice.h>
3
4  class MYService: public Tango::NTService
5  {
6  public:
7     MYService(char *);
8
9     void start(int,char **,Tango::NTEventLogger *);
10 };
```

Line 1-2 : Some include files

Line 4 : The MYService class inherits from *Tango::NTService* class

Line 7 : Constructor with one parameter

Line 9 : The *start()* method

The class source code looks like

```
1  #include <myservice.h>
2  #include <tango.h>
3
4  using namespace std;
5
6  MYService::MYService(char *exec_name):NTService(exec_name)
7  {
8  }
9
10 void MYService::start(int argc,char **argv,Tango::NTEventLogger *logger)
11 {
12    Tango::Util *tg;
13    try
14    {
15       Tango::Util::_service = true;
16
17       tg = Tango::Util::init(argc,argv);
18
19       tg->server_init();
20
21       tg->server_run();
22    }
23    catch (bad_alloc)
24    {
```

(continues on next page)

(continued from previous page)

```
25          logger->error("Can't allocate memory to store device object");
26      }
27      catch (Tango::DevFailed &e)
28      {
29          logger->error(e.errors[0].desc.in());
30      }
31      catch (CORBA::Exception &)
32      {
33          logger->error("CORBA Exception");
34      }
35  }
```

Line 6-8 : The MYService class constructor code.

Line 15 : Set to true the *_service* static variable of the *Tango::Util* class.

Line 17-21 : Classical Tango device server startup code

Line 23-34 : Exception management. Please, note that within a service. it is not possible to print data on a console. This method receives a reference to a logger object. This object sends all its output to the Windows event system. It is used to send messages when an exception has occurred.

The main function

The main function is used to create one instance of the class describing the service, to check the service option and to run the service. The code looks like :

```
1   #include <tango.h>
2   #include <MYService.h>
3
4   using namespace std;
5
6
7   int main(int argc,char *argv[])
8   {
9      MYService service(argv[0]);
10
11     int ret;
12     if ((ret = service.options(argc,argv)) <= 0)
13         return ret;
14
15     service.run(argc,argv);
16
17     return 0;
18  }
```

Line 9 : Create one instance of the MYService class with the executable name as parameter

Line 12 : Check service option with the *options()* method inherited from the NTService class.

Line 15 : Run the service. The *run()* method is inherited from the NTService class. This method will after some NT initialization sequence execute the user *start()* method.

Service options and messages

When a Tango device server is written as a Windows service, it supports several new options. These option are linked to Windows service usage.

Before it can be used, a service must be installed. A name and a title is associated to each service. For Tango device server used as service, the service name is build from the executable name followed by the underscore character and the instance name. For example, a device server service executable file named "opc" and started with "fluids" as instance name, will be named "opc_fluids". The title string is built from the service executable name followed by the sentence "Tango device server" and the instance name between parenthesis. In the previous example, the service title will be "opc Tango device server (fluids)". Once a service is installed, you can configure it with the "Services" application of the control panel. Services title are displayed by this application and allow the user to select one specific service. Once a service is selected, it is possible to start/stop it and to configure its startup type as manual (with the Services application) or as automatic. When the automatic mode is chosen, the service starts when the computer is started. In this case, the service executable code must resides on the computer local disk.

Tango device server logs message in the Windows event system when the service is started or stopped. You can see these messages with the "Event Viewer" application (Start->Programs->Administrative tools->Event Viewer) and choose the Application events.

The new options are -i, -s, -u, -h and -d.

- -i : Install the service
- -s : Install the service and choose the automatic startup mode
- -u : Un-install the service
- -dbg : Run in console mode to debug service. The service must have been installed prior to used it. The classical -v device server option can be used with the -d option.

On the command line, all these options must be used after the device server instance name ("opc fluids -i" to install the service, "opc fluids -u" to un-install the service, "opc fluids -v -d" to debug the service)

Tango device server using MFC as Windows service

If your Tango device server uses MFC and must be written as a Windows NT service, follow these rules :

- Don't forget to add the *stdafx.h* file as the first file included in all the source files making the project.
- Comment out the definition of VC_EXTRALEAN in the *stdafx.h* file.
- Change the pre-processor definitions, replace _WINDOWS by _CONSOLE
- Add the /SUBSYSTEM:CONSOLE option in the linker options window of the project settings.
- Add a call to initialize the MFC (*AfxWinInit()*) in the service main function

```
int main(int argc,char *argv[])
{
   if (!AfxWinInit(::GetModuleHandle(NULL),NULL,::GetCommandLine(),0))
   {
      cerr << "Can't initialise MFC !" << endl;
      return -1;
   }

   service serv(argv[0]);

```

(continues on next page)

(continued from previous page)

```
11      int ret;
12      if ((ret = serv.options(argc,argv)) <= 0)
13           return ret;
14
15      serv.run(argc,argv);
16
17      return 0;
18  }
```

Line 3 : The MFC classes are initialized with the *AfxWinInit()* function call.

Compiling, linking and executing a TANGO device server process

Compiling and linking a C++ device server

On UNIX like operating system

Supported development tools

The supported compiler for Linux is **gcc** release 3.3 and above. Please, note that to debug a Tango device server running under Linux, **gdb** release 7 and above is needed in order to correctly handle threads.

Compiling

TANGO for C++ uses omniORB (release 4) [omniORB] as underlying CORBA Object Request Broker [OMG] and starting with Tango 8, the ZMQ library [ZMQ]. To compile a TANGO device server, your include search path must be set to :

- The omniORB include directory
- The ZMQ include directory
- The Tango include directory
- Your development directory

Linking

To build a running device server process, you need to link your code with several libraries:

- The Tango library (called **libtango**)
- Three omniORB package libraries (called **libomniORB4**, **libomniDynamic4** and **libCOS4**)
- The omniORB threading library (called **libomnithread**)
- The ZMQ library (callled **libzmq**)

On top of that, you need additional libraries depending on the operating system :

- For Linux, add the POSIX thread library (**libpthread**)

The following table summarizes the necessary options to compile a Tango C++ device server. Please, note that starting with Tango 8 and for gcc release 4.3 and later, some C++11 code has been used. This requires the compiler option -std=c++0x (or better). Obviously, the options -I and -L must be updated to reflect your file system organization.

Operating system	Compiling option	Linking option
Linux gcc	-D_REENTRANT -std=c++0x -I..	-L.. -ltango -lomniORB4 -lomniDynamic4 -lCOS4 -lomnithread -lzmq -lpthread

The following is an example of a Makefile for Linux. Obviously, all the paths are set to the ESRF file system structure.

```
#
# Makefile to generate a Tango server
#

CC = c++
BIN_DIR = ubuntu1104
TANGO_HOME = /segfs/tango

INCLUDE_DIRS = -I $(TANGO_HOME)/include/$(BIN_DIR) -I .


LIB_DIRS = -L $(TANGO_HOME)/lib/$(BIN_DIR)


CXXFLAGS = -D_REENTRANT -std=c++0x $(INCLUDE_DIRS)
LFLAGS = $(LIB_DIRS) -ltango \
                     -lomniORB4 \
                     -lomniDynamic4 \
                     -lCOS4 \
                     -lomnithread \
                     -lzmq \
                     -lpthread


SVC_OBJS = main.o \
           ClassFactory.o \
           SteppermotorClass.o \
           Steppermotor.o \
           SteppermotorStateMachine.o


.SUFFIXES: .o .cpp
.cpp.o:
    $(CC) $(CXXFLAGS) -c $<


all: StepperMotor

StepperMotor: $(SVC_OBJS)
    $(CC) $(SVC_OBJS) -o $(BIN_DIR)/StepperMotor $(LFLAGS)

clean:
```

(continues on next page)

(continued from previous page)

```
43      rm -f *.o core
```

Line 5-7 : Define Makefile macros

Line 9-10 : Set the include file search path

Line 12 : Set the linker library search path

Line 15 : The compiler option setting

Line 16-23 : The linker option setting

Line 26-30 : All the object files needed to build the executable

Line 33-35 : Define rules to generate object files

Line 38 : Define a "all" dependency

Line 40-41 : How to generate the StepperMotor device server executable

Line 43-44 : Define a "clean" dependency

On Windows using Visual Studio

Supported Windows compiler for Tango is Visual Studio 2008 (VC 9), Visual Studio 2010 (VC10) and Visual Studio 2013 (VC12). Most problems in building a Windows device server revolve around the /M compiler switch family. This switch family controls which run-time library names are embedded in the object files, and consequently which libraries are used during linking. Attempt to mix and match compiler settings and libraries can cause link error and even if successful, may produce undefined run-time behavior.

Selecting the correct /M switch in Visual Studio is done through a dialog box. To open this dialog box, click on the "Project" menu (once the correct project is selected in the Solution Explorer window) and select the "Properties" option. To change the compiler switch open the "C/C++" tree and select "Code Generation". The following options are supported.

- Multithreaded = /MT
- Multithreaded DLL = /MD
- Debug Multithreaded = /MTd
- Debug Multithreaded DLL = /MDd

Compiling a file with a value of the /M switch family will impose at link phase the use of libraries also compiled with the same value of the /M switch family. If you compiled your source code with the /MT option (Multithreaded), you must link it with libraries also compiled with the /MT option.

On both 32 or 64 bits computer, omniORB and TANGO relies on the preprocessor identifier **WIN32** being defined in order to configure itself. If you build an application using static libraries (option /MT or /MTd), you must add **_WINSTATIC** to the list of the preprocessor identifiers. If you build an application using DLL (option /MD or /MDd), you must add **LOG4TANGO_HAS_DLL** and **TANGO_HAS_DLL** to the list of preprocessor identifiers.

To build a running device server process, you need to link your code with several libraries on top of the Windows libraries. These libraries are:

- The Tango library (called **tango.lib** or **tangod.lib** for debug mode)
- The omniORB package libraries (see next table)

Compile mode	Libraries
Debug Multithreaded	omniORB4d.lib, omniDynamic4d.lib, omnithreadd.lib and COS4d.lib
Multithreaded	omniORB4.lib, omniDynamic4.lib, omnithread.lib and COS4.lib
Debug Multithreaded DLL	omniORB420_rtd.lib, omniDynamic420_rtd.lib, omnithread40_rtd.lib, and COS420_rtd.lib
Multithreaded DLL	omniORB420_rt.lib, omniDynamic420_rt.lib, omnithread40_rt.lib and COS420_rt.lib

- The ZMQ library (**zmq.lib** or **zmqd.lib** for debug mode)
- Windows network libraries (**mswsock.lib** and **ws2_32.lib**)
- Windows graphic library (**comctl32.lib**)

To add these libraries in Visual Studio, open the project property pages dialog box and open the "Link" tree. Select "Input" and add these library names to the list of library in the "Additional Dependencies" box.

The "Win32 Debug" or "Win32 Release" configuration that you change within the Configuration Manager window changes the /M switch compiler. For instance, if you select a "Win32 Debug" configuration in a non-DLL project, use the omniORB4d.lib, omniDynamic4d.lib and omnithreadd.lib libraries plus the tangod.lib and zmqd.lib libraries. If you select the "Win32 Release" configuration, use the omniORB4.lib, omniDynamic4.lib and omnithread.lib libraries plus the tango.lib and zmq.lib libraries.

WARNING: In some cases, the Microsoft Visual Studio wizard used during project creation generates one include file called *Stdafx.h*. If this file itself includes windows.h file, you have to add the preprocessor macro _WIN32_WINNT and set it to 0x0500.

Running a C++ device server

To run a C++ Tango device server, you must set an environment variable. This environment variable is called **TANGO_HOST** and has a fixed syntax which is

TANGO_HOST=<host>:<port>

The host field is the host name where the TANGO database device server is running. The port field is the port number on which this server is listening. For instance, a valid syntax is TANGO_HOST=dumela:10000. For UNIX like operating system, setting environment variable is possible with the *export* or *setenv* command depending on the shell used. For Windows, setting environment variable is possible with the "Environment" tab of the "System" application in the control panel.

If you need to start a Tango device server on a pre-defined port (For Tango database device server or device server without database usage), you must use one of the underlying ORB option *endPoint* like

myserver myinstance_name -ORBendPoint giop:tcp::<port number>

Advanced programming techniques

The basic techniques for implementing device server pattern are required by each device server programmer. In certain situations, it is however necessary to do things out of the ordinary. This chapter will look into programming techniques which permit the device server serve more than simply the network.

Receiving signal

It is **UNSAFE** to use any CORBA call in a signal handler. It is also UNSAFE to use some system calls in a signal handler. Tango device server solved this problem by using threads. A specific thread is started to handle signals. Therefore, every Tango device server is automatically a threaded process. This allows the programmer to write the code which must be executed when a signal is received as ordinary code. All device server threads masks all signals except the specific signal thread which is permanently waiting for signal. If a signal is sent to a device server process, only the signal thread will receive it because it is the single thread which does not mask signals.

Nevertheless, signal management is not trivial and some care have to be taken. The signal management differs from operating system to operating system. It is not recommended that you install your own signal routine using any of the signal routines provided by the operating system calls or library.

Using signal

It is possible for C++ device server to receive signals from drivers or other processes. The TDSOM supports receiving signal at two levels: the device level and the class level. Supporting signal at the device level means that it is possible to specify interest into receiving signal on a device basis. This feature is supported via three methods defined in the DeviceImpl class. These methods are called *register_signal*, *unregister_signal* and s*ignal_handler*.

The ***register_signal*** method has one parameter which is the signal number. This method informs the device server signal system that the device want to be informed when the signal passed as parameter is received by the process. There is a special case for Linux as explained in the previous sub-chapter. It is possible to register a signal to be executed in the a signal handler context (with all its restrictions). This is done with a second parameter to this *register_signal* method. This second parameter is simply a boolean data. If it is true, the signal_handler will be executed in a signal handler context in the device server main thread. A default value (false) has been defined for this parameter.

The ***unregister_signal*** method also have an input parameter which is the signal number. This method removes the device from the list of object which should be warned when the signal is received by the process.

The ***signal_handler*** method is the method which is triggered when a signal is received if the corresponding *register_signal* has been executed. This method is defined as virtual and can be redefined by the user. It has one input argument which is the signal number.

The same three methods also exist in the DeviceClass class. Their action and their usage are similar to the DeviceImpl class methods. Installing a signal at the class level does not mean that all the device belonging to this class will receive the signal. This only means that the *signal_handler* method of the DeviceClass instance will be executed. This is useful if an action has to be executed once for a class of devices when a signal is received.

The following code is an example with our stepper motor device server configured via the database to serve three motors. These motors have the following names : id04/motor/01, id04/motor/02 and id04/motor/03. The signal SIGALRM (alarm signal) must be propagated only to the motor number 2 (id04/motor/02)

```
1  void StepperMotor::init_device()
2  {
3      cout << "StepperMotor::StepperMotor() create motor " << dev_name << endl;
4
5      long i;
6
7      for (i=0; i< AGSM_MAX_MOTORS; i++)
8      {
9          axis[i] = 0;
10         position[i] = 0;
11         direction[i] = 0;
12     }
```

(continues on next page)

(continued from previous page)

```
13
14      if (dev_name == "id04/motor/02")
15          register_signal(SIGALRM);
16  }
17
18  StepperMotor::~StepperMotor()
19  {
20      unregister_signal(SIGALRM);
21  }
22
23  void StepperMotor::signal_handler(long signo)
24  {
25      INFO_STREAM << "Inside signal handler for signal " << signo << endl;
26
27  //  Do what you want here
28
29  }
```

The *init_device* method is modified.

Line 14-15 : The device name is checked and if it is the correct name, the device is registered in the list of device wanted to receive the SIGALARM signal.

The destructor is also modified

Line 20 : Unregister the device from the list of devices which should receives the SIGALRM signal. Note that unregister a signal for a device which has not previously registered its interest for this signal does nothing.

The *signal_handler* method is redefined

Line 25 : Print signal number

Line 27 : Do what you have to do when the signal SIGALRM is received.

If all devices must be warned when the device server process receives the signal SIGALRM, removes line 14 in the *init_device* method.

Exiting a device server gracefully

A device server has to exit gracefully by unregistering itself from the database. The necessary action to gracefully exit are automatically executed on reception of the following signal :

- SIGINT, SIGTERM and SIGQUIT for device server running on Linux
- SIGINT, SIGTERM, SIGABRT and SIGBREAK for device server running on Windows

This does not prevents device server to also register interest at device or class levels for those signals. The user installed *signal_handler* method will first be called before the graceful exit.

Inheriting

This sub-chapter details how it is possible to inherit from an existing device pattern implementation. As the device pattern includes more than a single class, inheriting from an existing device pattern needs some explanations.

Let us suppose that the existing device pattern implementation is for devices of class A. This means that classes A and AClass already exists plus classes for all commands offered by device of class A. One new device pattern implementation for device of class B must be written with all the features offered by class A plus some new one. This is easily done with the inheritance. Writing a device pattern implementation for device of class B which inherits from device of class A means :

- Write the BClass class
- Write the B class
- Write B class specific commands
- Eventually redefine A class commands

The miscellaneous code fragments given below detail only what has to be updated to support device pattern inheritance

Writing the BClass

As you can guess, BClass has to inherit from AClass. The *command_factory* method must also be adapted.

```
1  namespace B
2  {
3
4  class BClass : public A::AClass
5  {
6  .....
7  }
8
9  BClass::command_factory()
10 {
11     A::AClass::command_factory();
12
13     command_list.push_back(....);
14 }
15
16 } /* End of B namespace */
```

Line 1 : Open the B namespace

Line 4 : BClass inherits from AClass which is defined in the A namespace.

Line 11 : Only the *command_factory* method of the BClass will be called at start-up. To create the AClass commands, the *command_factory* method of the AClass must also be executed. This is the reason of the line

Line 13 : Create BClass commands

Writing the B class

As you can guess, B has to inherits from A.

```
1  namespace B
2  {
3
4  class B : public A:A
5  {
6     .....
7  };
8
9  B::B(Tango::DeviceClass *cl,const char *s):A::A(cl,s)
10 {
11    ....
12    init_device();
13 }
14
15 void B::init_device()
16 {
17    ....
18 }
19
20 } /* End of B namespace */
```

Line 1 : Open the B namespace.

Line 4 : B inherits from A which is defined in the A namespace

Line 9 : The B constructor calls the right A constructor

Writing B class specific command

Noting special here. Write these classes as usual

Redefining A class command

It is possible to redefine a command which already exist in class A **only if the command is created using the inheritance model** (but keeping its input and output argument types). The method which really execute the class A command is a method implemented in the A class. This method must be defined as **virtual.** In class B, you can redefine the method executing the command and implement it following the needs of the B class.

Using another device pattern implementation within the same server

It is often necessary that inside the same device server, a method executing a command needs a command of another class to be executed. For instance, a device pattern implementation for a device driven by a serial line class can use the command offered by a serial line class embedded within the same device server process. To execute one of the command (or any other CORBA operations/attributes) of the serial line class, just call it as a normal client will do by using one instance of the DeviceProxy class. The ORB will recognize that all the devices are inside the same process and will execute calls as a local calls. To create the DeviceProxy class instance, the only thing you need to know is the name of the device you gave to the serial line device. Retrieving this could be easily done by a Tango device property. The DeviceProxy class is fully described in Tango Application Programming Interface (API) reference WEB pages

Device pipe

What a Tango device pipe is has been defined in the Chapter 3 about device server model. How you read or write a pipe in a client software is documented in chapter 4 about the Tango API. In this section, we describe how you can read/write into/from a device pipe on the server side (In a Tango class with pipe).

Client reading a pipe

When a client reads a pipe, the following methods are executed in the Tango class:

1. The *always_executed_hook()* method.
2. A method called *is_<pipe_name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some pipes which can be read only in some precise conditions. It has one parameter which is the request type (read or write)
3. A method called *read_<pipe_name>()*. The aim of this method is to store the pipe data in the pipe object. It has one parameter which is a reference to the Pipe object to be read.

The figure *6.8* is a drawing of these method calls sequencing for our class StepperMotor with one pipe named DynData.

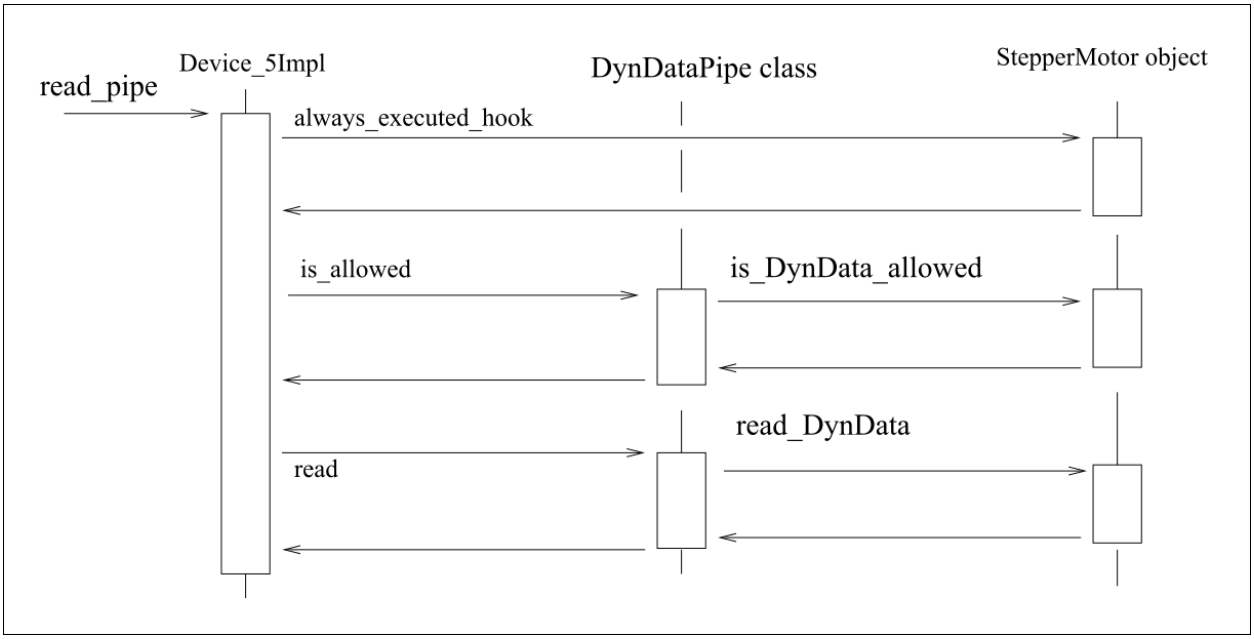


Fig. 13: Figure 6.8: Read pipe sequencing

The class DynDataPipe is a simple class which follow the same skeleton from one Tango class to another. Therefore, this class is generated by the Tango code generator Pogo and the Tango class developper does not have to modify it. The method *is_DynData_allowed()* is relatively simple and in most cases the default code generated by Pogo is enough. The method *read_DynData()* is the method on which the Tango class developper has to concentrate on. The following code is one example of these two methods.

```
1  bool StepperMotor::is_DynData_allowed(Tango::PipeReqType req)
2  {
3      if (get_state() == Tango::ON)
4          return true;
```

(continues on next page)

(continued from previous page)

```
5       else
6           return false;
7   }
8
9   void StepperMotor::read_DynData(Tango::Pipe &pipe)
10  {
11      nb_call++;
12      if (nb_call % 2 == 0)
13      {
14          pipe.set_root_blob_name("BlobCaseEven");
15
16          vector<string> de_names {"EvenFirstDE","EvenSecondDE"};
17          pipe.set_data_elt_names(de_names);
18
19          dl = 666;
20          v_db.clear();
21          v_db.push_back(1.11);
22          v_db.push_back(2.22);
23
24          pipe << dl << v_db;
25      }
26      else
27      {
28          pipe.set_root_blob_name("BlobCaseOdd");
29
30          vector<string> de_names {"OddFirstDE"};
31          pipe.set_data_elt_names(de_names);
32
33          v_str.clear();
34          v_str.push_back("Hola");
35          v_str.push_back("Salut");
36          v_str.push_back("Hi");
37
38          pipe << v_str;
39      }
40  }
```

The *is_DynData_allowed* method is defined between lines 1 and 7. It is allowed to read or write the pipe only is the device state is ON. Note that the input parameter req is not used. The parameter allows the user to know the type of request. The data type PipeReqType is one enumeration with two possible values which are READ_REQ and WRITE_REQ.

The *read_DynData* method is defined between lines 9 and 40. If the number of times this method has been called is even, the pipe contains two data elements. The first one is named EvenFirstDE and its data is a long. The second one is named EvenSecondDE and its data is an array of double. If the number of call is odd, the pipe contains only one data element. Its name is OddFirstDe and its data is an array of strings. Data are inserted into the pipe at lines 24 and 38. The variables nb_call, dl, v_db and v_str are device data member and therefore declare in the .h file. Refer to pipe section in chapter 3 and to the API reference documentation (in Tango WEB pages) to learn more on how you can insert data into a pipe and to know how data are organized within a pipe.

Client writing a pipe

When a client writes a pipe, the following methods are executed in the Tango class:

1. The *always_executed_hook()* method.
2. A method called *is_<pipe_name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some pipes which can be read only in some precise conditions. It has one parameter which is the request type (read or write)
3. A method called *write_<pipe_name>()*. It has one parameter which is a reference to the WPipe object to be written. The aim of this method is to get the data to be written from the WPipe oject and to write them into the corresponding Tango class objects.

The figure 6.9 is a drawing of these method calls sequencing for our class StepperMotor with one pipe named DynData.

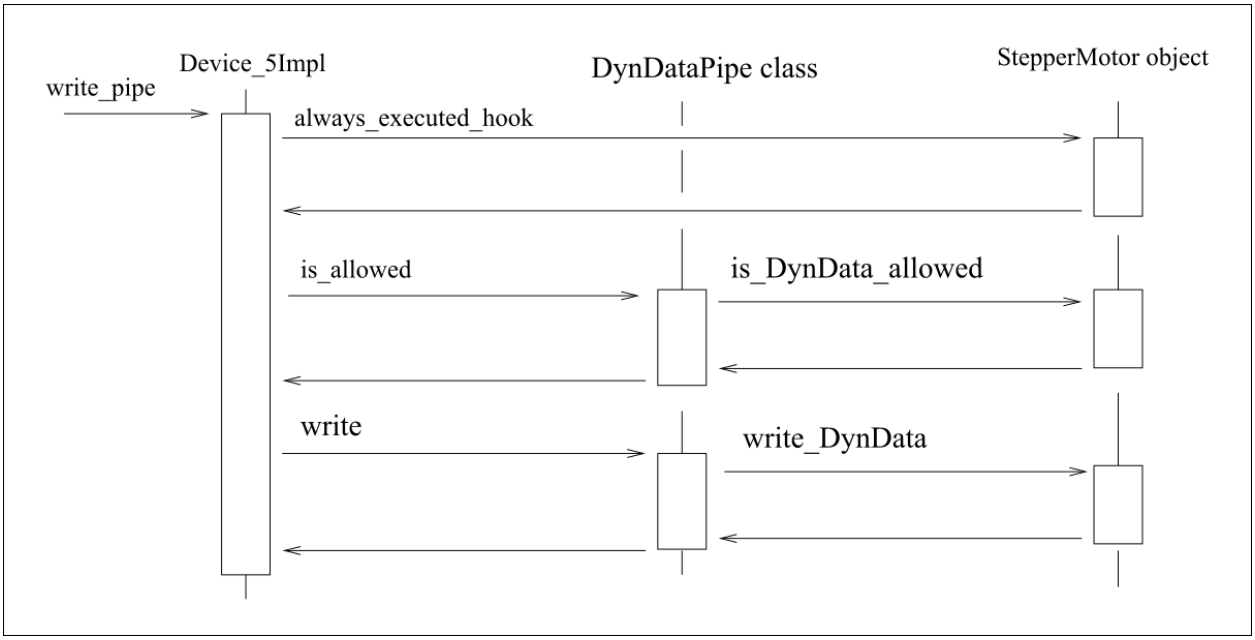


Fig. 14: Figure 6.9: Write pipe sequencing

The class DynDataPipe is a simple class which follow the same skeleton from one Tango class to another. Therefore, this class is generated by the Tango code generator Pogo and the Tango class developper does not have to modify it. The method *is_DynData_allowed()* is relatively simple and in most cases the default code generated by Pogo is enough. The method *write_DynData()* is the method on which the Tango class developper has to concentrate on. The following code is one example of the *write_DynData()* method.

```
1  void StepperMotor::write_DynData(Tango::WPipe &w_pipe)
2  {
3     string str;
4     vector<float> v_fl;
5
6     w_pipe >> str >> v_fl;
7     .....
8  }
```

In this example, we know that the pipe will always contain a srting followed by one array of float. On top of that, we are not niterested by the

data element names. Data are extracted from the pipe at line 6 and are available for further use starting at line 7. If the content of the pipe is not a string followed by one array of float, the data extraction line (6) will throw one exception which will be reported to the client who has tried to write the pipe. Refer to pipe section in chapter 3 and to the API reference documentation (in Tango WEB pages) to learn more on how you can insert data into a pipe and to know how data are organized within a pipe.

6.5.5 Attribute alarms

Intended audience: users, developers

Each Tango attribute two several alarms. These alarms are :

- A four thresholds level alarm
- The read different than set (RDS) alarm

The level alarms

This alarm is defined for all Tango attribute read type and for numerical data type. The action of this alarm depend on the attribute value when it is read :

- If the attribute value is below or equal the attribute configuration **min_alarm** parameter, the attribute quality factor is switched to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is below or equal the attribute configuration **min_warning** parameter, the attribute quality factor is switched to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is above or equal the attribute configuration **max_warning** parameter, the attribute quality factor is switched to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is above or equal the attribute configuration **max_alarm** parameter, the attribute quality factor is switched to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM.

If the attribute is a spectrum or an image, then the alarm is set if any one of the attribute value satisfies the above criterium. By default, these four parameters are not defined and no check will be done.

The following figure is a drawing of attribute quality factor and device state values function of the the attribute value.

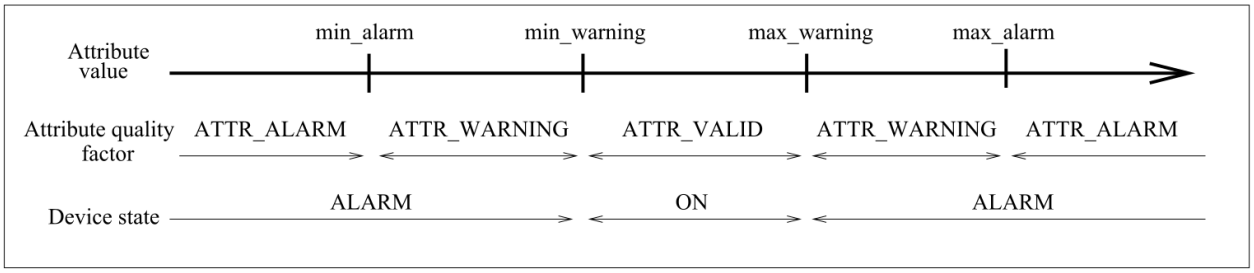


Fig. 15: Figure 7.1: Level alarm

If the min_warning and max_warning parameters are not set, the attribute quality factor will simply change between Tango::ATTR_ALARM and Tango::ATTR_VALID function of the attribute value.

The Read Different than Set (RDS) alarm

This alarm is defined only for attribute of the Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read (or when the device state is requested), if the difference between its read value and the last written value is something more than or equal to an authorized delta and if at least a certain amount of milli seconds occurs since the last write operation, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. If the attribute is a spectrum or an image, then the alarm is set if any one of the attribute value's satisfies the above criterium. This alarm configuration is done with two attribute configuration parameters called **delta_val** and **delta_t**. By default, these two parameters are not defined and no check will be done.

6.5.6 Enumerated attribute

Intended audience: developers, ***Programming language:*** c++

Since Tango release 9, enumerated attribute is supported using the new data type DevEnum. This data type is not a real C++ enumeration because:

1. The enumerated value allways start with 0
2. Values are consecutive
3. It is transferred on the network as DevShort data type

One enumeration label is associated to each enumeration value. For the Tango kernel, it is this list of enumeration labels which will define the possible enumeration values. For instance if the enumeration has 3 labels, its value must be between 0 and 2. There are two ways to define the enumeration labels:

1. At attribute creation time. This is the most common case when the list of possible enumeration values and labels are known at compile time. The Tango code generator Pogo generates for you the code needed to pass the enumeration labels to the Tango kernel.
2. In the user code when the enumeration values and labels are not known at compile time but retrieved during device startup phase. The user gives the possible enumeration values to the Tango kernel using the Attribute class *set_properties()* method.

A Tango client is able to retrieve the enumeration labels in the attribute configuration returned by instance by a call to the *DeviceProxy::get_attribute_config()* method. Using the *DeviceProxy::set_attribute_config()* call, a user may change the enumeration labels but not their number.

Usage in a Tango class

Within a Tango class, you set the attribute value with a C++ enum or a DevShort variable. In case a DevShort variable is used, its value will be checked according to the enumeration labels list given to Tango kernel.

Setting the labels with enumeration compile time knowledge

In such a case, the enumeration labels are given to Tango at the attribute creation time in the *attribute_factory* method of the XXXClass class. Let us take one example

```
enum class Card: short
{
    NORTH = 0,
    SOUTH,
    EAST,
```

(continues on next page)

(continued from previous page)

```
6       WEST
7   };
8
9   struct TheEnumAttrib : Tango::Attr
10  {
11      ...
12
13      // Required for enum attributes
14      virtual bool same_type(const type_info& in_type) override { return typeid(Card) ==␣
↪in_type; }
15      virtual std::string get_enum_type() override { return "Card"; }
16  }
17
18  void XXXClass::attribute_factory(vector<Tango::Attr *> &att_list)
19  {
20      .....
21      TheEnumAttrib   *theenum = new TheEnumAttrib();
22      Tango::UserDefaultAttrProp theenum_prop;
23      vector<string> labels = {"North","South","East","West"};
24      theenum_prop.set_enum_labels(labels);
25      theenum->set_default_properties(theenum_prop);
26      att_list.push_back(theenum);
27      .....
28   }
```

line 1-7 : The definition of the enumeration (C++11 in this example)

line 9-16 : The definition of the enumerated attribute

line 23 : A vector of strings with the enumeration labels is created. Because there is no way to get the labels from the enumeration definition, they are re-defined here.

line 24 : This vector is given to the theenum_prop object which contains the user default properties

line 26 : The user default properties are associated to the attribute
In most cases, all this code will be automatically generated by the Tango code generator Pogo. It is given here for completness.

Setting the labels without enumeration compile time knowledge

In such a case, the enumeration labels are retrieved by the user in a way specific to the device and passed to Tango using the Attribute class *set_properties()* method. Let us take one example

```
1   void MyDev::init_device()
2   {
3       ...
4
5       Attribute &att = get_device_attr()->get_attr_by_name("TheEnumAtt");
6       MultiAttrProp<DevEnum> multi_prop;
7       att.get_properties(multi_prop);
8
```

(continues on next page)

(continued from previous page)

```
9       multi_prop.enum_labels = {....};
10      att.set_properties(multi_prop);
11      ....
12   }
```

line 5 : Get a reference to the attribute object

line 7 : Retrieve the attribute properties

line 9 : Initialise the attribute labels in the set of attribute properties

line 10 : Set the attribute properties

Setting the attribute value

It is possible to set the attribute value using either a classical DevShort variable or using a variable of the C++ enumeration. The following example is when you have compile time knowledge of the enumeration definition. We assume that the enumeration is the same than the one defined above (Card enumeration)

```
1   enum Card points;
2
3   void MyDev::read_TheEnum(Attribute &att)
4   {
5       ...
6       points = SOUTH;
7       att.set_value(&points);
8   }
```

line 1 : One instance of the Card enum is created (named points)

line 6 : The enumeration is initialized

line 7 : The value of the attribute object is set using the enumeration (by pointer)
To get the same result using a classical DevShort variable, the code looks like

```
1   DevShort sh;
2
3   void MyDev::read_TheEnum(Attribute &att)
4   {
5       ...
6       sh = 1;
7       att.set_value(&sh);
8   }
```

line 1 : A DevShort variable is created (named sh)

line 6 : The variable is initialized

line 7 : The value of the attribute object is set using the DevShort variable (by pointer)

Usage in a Tango client

Within a Tango client, you insert/extract enumerated attribute value in/from DeviceAttribute object with a C++ enum or a DevShort variable. The later case is for generic client which do not have compile time knowledge of the enumeration. The code looks like

```
1  DeviceAttribute da = the_dev.read_attribute("TheEnumAtt");
2  Card ca;
3  da >> ca;
4
5  DeviceAttribute db = the_dev.read_attribute("TheEnumAtt");
6  DevShort sh;
7  da >> sh;
```

line 2-3 : The attribute value is extracted in a C++ enumeration variable

line 6-7 : The attribute value is extracted in a DevShort variable

6.5.7 Memorized attribute

Intended audience: advanced developers, ***Programming language:*** c++

It is possible to ask Tango to store in its database the last written value for attribute of the SCALAR data format and obviously only for READ_WRITE or READ_WITH_WRITE attribute. This is fully automatic. During device startup phase, for all device memorized attributes, the value written in the database is fetched and applied. A write_attribute call can be generated to apply the memorized value to the attribute or only the attribute set point can be initialised. The following piece of code shows how the source code should be written to set an attribute as memorized and to initialise only the attribute set point.

```
1  void DevTestClass::attribute_factory(vector<Tango::Attr *> &att_list)
2   {
3       ...
4       att_list.push_back(new String_attrAttr());
5       att_list.back()->set_memorized();
6       att_list.back()->set_memorized_init(false);
7       ...
8   }
```

Line 4 : The attribute to be memorized is created and inserted in the attribute vector.

Line 5 : The *set_memorized()* method of the attribute base class is called to define the attribute as memorized.

Line 6 : The set_memorized_init() method is called with the parameter false to define that only the set point should be initialsied.

6.5.8 Forwarded attribute

Intended audience: developers, ***Programming language:*** c++

Definition

Let's take an example to explain what is a forwarded attribute. We assume we have to write a Tango class for a ski lift in a ski resort somewhere in the Alps. Obviously, the ski lift has a motor for which we already have a Tango class. This motor Tango class has one attribute *speed*. But for the ski lift, the motor speed is not the only thing which has to be controlled. For instance, you also want to give access to the wind sensor data installed on the top of the ski lift. Therefore, you write a ski-lift Tango class representing the whole ski-lift system. This ski-lift class will have at least two attributes which are:

1. The wind speed at the top of the ski-lift
2. The motor speed

The ski-lift Tango class motor speed attribute is nothing more than the motor Tango class speed attribute. All the ski-lift class has to do for this attribute is to forward the request (read/write) to the speed attribute of the motor Tango class. The speed attribute of the ski-lift Tango class is a **forwarded attribute** while the speed attribute of the motor Tango class is its **root attribute**.

A forwarded attribute get its configuration from its root attribute and it forwards to its root attribute

- Its read / write / write_read requests
- Its configuration change
- Its event subscription
- Its locking behavior

As stated above, a forwarded attribute has the same configuration than its root attribute except its *name* and *label* which stays local. All other attribute configuration parameters are forwarded to the root attribute. If a root attribute configuration parameter is changed, the forwarded attribute is informed (via event) and its local configuration is also modified.

The association between the forwarded attribute and its root attribute is done using a property named

__root_att

belonging to the forwarded attribute. This property value is simply the name of the root attribute. Muti-control system is supported and this __root_att attribute property value can be something like *tango://my_tango_host:10000/my/favorite/dev/the_root_attribute*. The name of the root attribute is included in attribute configuration.

It is forbidden to poll a forwarded attribute and one exception is thrown if such a case happens. Polling has to be done on the root attribute. Nevertheless, if the root attribute is polled, a request to read the forwarded attribute with the DeviceProxy object source parameter set to CACHE_DEVICE or CACHE will get its data from the root attribute polling buffer.

If you subscribe to event(s) on a forwarded attribute, the subscription is forwarded to the root attribute. When the event is received by the forwarded attribute, the attribute name in the event data is modified to reflect the forwarded attribute name and the event is pushed to the original client(s).

When a device with forwarded attribute is locked, the device to which the root attribute belongs is also locked.

Coding

As explained in the chapter Writing a Tango device server, each Tango class attribute is implemented via a C++ class which has to inherit from either *Attr*, *SpectrumAttr* or *ImageAttr* according to the attribute data format. For forwarded attribute, the related class has to inherit from the **FwdAttr** class whatever its data format is. For classical attribute, the programmer can define in the Tango class code default value for the attribute properties using one instance of the *UserDefaultAttrProp* class. For forwarded attribute, the programmer has to create one instance of the **UserDefaultFwdAttrProp** class but only the attribute label can be defined. One example of how to program a forwarded attribute is given below

```
1  class MyFwdAttr: public Tango::FwdAttr
2  {
3  public:
4      MyFwdAttr(const string &_n):FwdAttr(_n) {};
5      ~MyFwdAttr() {};
6  };
7
8  void DevTestClass::attribute_factory(vector<Tango::Attr *> &att_list)
9  {
10    ...
11    MyFwdAttr *att1 = new MyFwdAttr("fwd_att_name");
12    Tango::UserDefaultFwdAttrProp att1_prop;
13    att1_prop.set_label("Gasp a fwd attribute");
14    att1->set_default_properties(att1_prop);
15    att_list.push_back(att1);
16    ...
17 }
```

Line 1 : The forwarded attribute class inherits from FwdAttr class.

Line 4-5 : Only constructor and destructor methods are required

Line 11 : The attribute object is created

Line 12-14 : A default value for the forwarded attribute label is defined.

Line 15: The forwarded attribute is added to the list of attribute

In case of error in the forwarded attribute configuration (for instance missing __root_att property), the attribute is not created by the Tango kernel and is therefore not visible for the external world. The state of the device to which the forwarded attribute belongs to is set to ALARM (if not already FAULT) and a detailed error report is available in the device status. In case a device with forwarded attribute(s) is started before the device(s) with the root attribute(s), the same principle is used: forwarded attribute(s) are not created, device state is set to ALARM and device status is reporting the error. When the device(s) with the root attribute will start, the forwarded attributes will automatically be created.

6.5.9 Device polling

Intended audience: developers, ***Programming language:*** cpp, all

Introduction

Each tango device server automatically have a separate polling thread pool. Polling a device means periodically executing command on a device (or reading device attribute) and storing the results (or the thrown exception) in a polling buffer. The aim of this polling is threefold :

- Speed-up response time for slow device
- Get a first-level history of device command output or attribute value
- Be the data source for the Tango event system

Speeding-up response time is achieved because the command_inout or read_attribute CORBA operation is able to get its data from the polling buffer or from the a real access to the device. For "slow" device, getting the data from the buffer is much faster than accessing the device. Returning a first-level command output history (or attribute value history) to a client is possible due to the polling buffer which is managed as a circular buffer. The history is the contents of this circular buffer. Obviously, the history depth is limited to the depth of the circular buffer. The polling is also the data source for the event system because detecting an event means being able to regularly read the data, memorize it and declaring that it is an event after some comparison with older values.

Starting with Tango 9, the default polling algorithm has been modifed. However, it is still possible to use the polling as it was in Tango releases prior to release 9. See chaper on polling configuration to get details on this.

Configuring the polling system

Configuring what has to be polled and how

It is possible to configure the polling in order to poll :

- Any command which does not need input parameter
- Any attribute

Configuring the polling is done by sending command to the device server administration device automatically implemented in every device server process. Seven commands are dedicated to this feature. These commands are

AddObjPolling
: It add a new object (command or attribute) to the list of object(s) to be polled. It is also with this command that the polling period is specified.

RemObjPolling
: To remove one object (command or attribute) from the polled object(s) list

UpdObjPollingPeriod
: Change one object polling period

StartPolling
: Starts polling for the whole process

StopPolling
: Stops polling for the whole process

PolledDevice
: Allow a client to know which device are polled

DevPollStatus
: Allow a client to precisely knows the polling status for a device

All the necessary parameters for the polling configuration are stored in the Tango database. Therefore, the polling configuration is not lost after a device server process stop and restart (or after a device server process crash!!).

It is also possible to automatically poll a command (or an attribute) without sending command to the device server administration device. This request some coding (a method call) in the device server software during the command or attribute creation. In this case, for every devices supporting this command or this attribute, polling configuration will be automatically updated in the database and the polling will start automatically at each device server process startup. It is possible to stop this behavior on a device basis by sending a RemObjPolling command to the device server administration device. The following piece of code shows how the source code should be written.

```
1  void DevTestClass::command_factory()
2  {
3  ...
4      command_list.push_back(new IOStartPoll("IOStartPoll",
5                                             Tango::DEV_VOID,
6                                             Tango::DEV_LONG,
7                                             "Void",
8                                             "Constant number"));
9      command_list.back()->set_polling_period(400);
10 ...
11 }
12
13
14 void DevTestClass::attribute_factory(vector<Tango::Attr *> &att_list)
15 {
16 ...
17     att_list.push_back(new Tango::Attr("String_attr",
18                                        Tango::DEV_STRING,
19                                        Tango::READ));
20     att_list.back()->set_polling_period(250);
21 ...
22 }
```

A polling period of 400 mS is set for the command called "IOStartPoll" at line 10 with the *set_polling_period* method of the Command class. Therefore, for a device of this class, the polling thread will start polling its IOStartPoll command at process start-up except if a RemObjPolling indicating this device and the IOStartPoll command has already been received by the device server administration device. This is exactly the same behavior for attribute. The polling period for attribute called "String_attr" is defined at line 20.

Configuring the polling means defining device attribute/command polling period. The polling period has to be chosen with care. If reading an attribute needs 200 mS, there is no point to poll this attribute with a polling period equal or even below 200 mS. You should also take into account that some free time has to be foreseen for external request(s) on the device. On average, for one attribute needing X mS as reading time, define a polling period which is equal to 1.4 X (280 mS for our example of one attribute needing 200 mS as reading time). In case the polling tuning is given to external user, Tango provides a way to define polling period minimun threshold. This is done using device properties. These properties are named *min_poll_period*, *cmd_min_poll_period* and *attr_min_poll_period*. The property min_poll_period (mS) defined a minimun polling period for the device. The property cmd_min_poll_period allows the definition of a minimun polling period for a specific device command. The property attr_min_poll_period allows the definition of a minimun polling period for one device attribute. In case these properties are defined, it is not possible to poll the device command/attribute with a polling period below those defined by these properties. See Appendix A on device parameter to get a precise syntax description for these properties.

The Jive (*Jive*) tool also allows a graphical device polling configuration.

Configuring the polling threads pool

Starting with Tango release 7, a Tango device server process may have several polling threads managed as a pool. For instance, this could be usefull in case of devices within the same device server process but accessed by different hardware channel when one of the channel is not responding (Thus generating long timeout and de-synchronising the polling thread). By default, the polling threads pool size is set to 1 and all the polled object(s) are managed by the same thread (idem polling system in Tango releases older than release 7) . The configuration of the polling thread pool is done using two properties associated to the device server administration device. These properties are named:

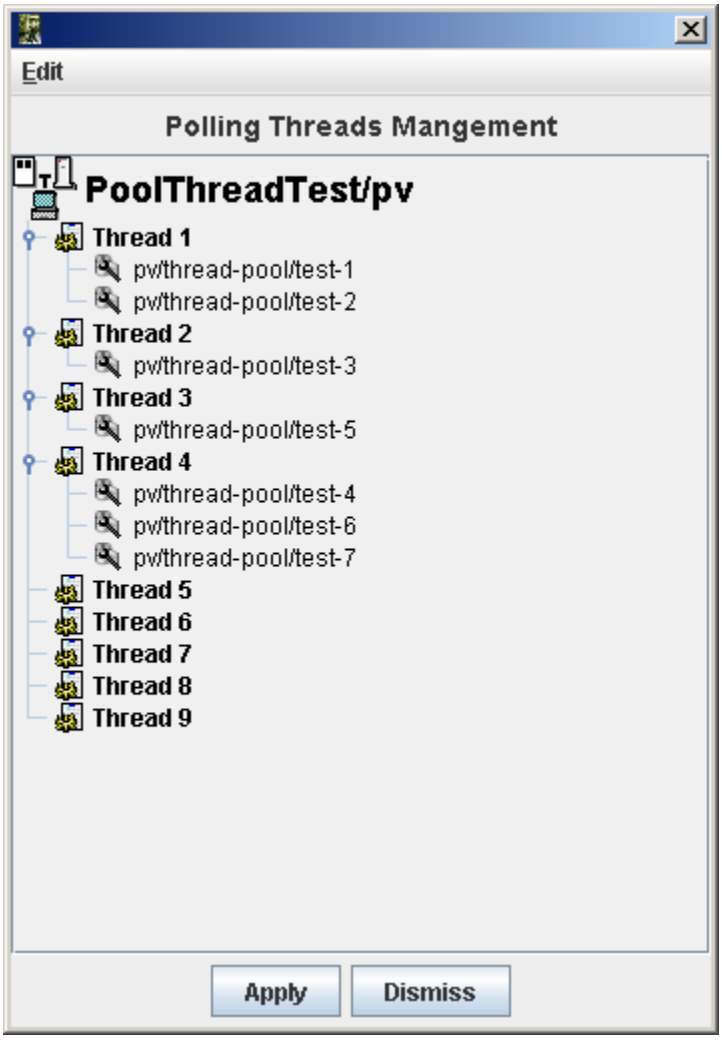
- *polling_threads_pool_size* defining the maximun number of threads that you can have in the pool
- *polling_threads_pool_conf* defining which threads in the pool manages which device

The granularity of the polling threads pool tuning is the device. You cannot ask the polling threads pool to have thread number 1 in charge of attribute *att1* of device *dev1* and thread number 2 to be in charge of *att2* of the same device *dev1*.

When you require a new object (command or attribute) to be polled, two main cases may arrive:

1. Some polled object(s) belonging to the device are already polled by one of the polling threads in the pool: There is no new thread created. The object is simply added to the list of objects to be polled for the existing thread
2. There is no thread already created for the device. We have two sub-cases:
    1. The number of polling threads is less than the polling_threads_pool_size: A new thread is created and started to poll the object (command or attribute)
    2. The number of polling threads is already equal to the polling_threads_pool_size: The software search for the thread with the smallest number of polled objects and add the new polled object to this thread

Each time the polling threads pool configuration is changed, it is written in the database using the polling_threads_pool_conf property. If the behaviour previously described does not fulfill your needs, it is possible to update the polling_threads_pool_conf property in a graphical way using the Tango Astor (ASTOR home page) tool or manually using the Jive tool [Jive]. These changes will be taken into account at the next device server process start-up. At start-up, the polling threads pool will allways be configured as required by the polling_threads_pool_conf property. The syntax used for this property is described in the Reference part of the *Appendix <A-reference.rst>*. The following window dump is the Astor tool window which allows polling threads pool management.



In this example, the polling threads pool size to set to 9 but only 4 polling threads are running. Thread 1 is in charge of all polled objects related to device pv/thread-pool/test-1 and pv/thread-pool/test-2. Thread 2 is in charge of all polled objects related to device pv/thread-pool/test-3. Thread 3 is in charge of all polled objects related to device pv/thread-pool/test-5 anf finally, thread 4 is in charge of all polled objects for devices pv/thread-pool/test-4, pv/thread-pool/test-6 and pv/thread-pool/test-7.

It's also possible to define the polling threads pool size programmatically in the main function of a device server process using the *Util::set_polling_threads_pool_size()* method before the call to the *Util::server_init()* method

Choosing polling algorithm

Starting with Tango 9, you can choose between two different polling algorithm:

- The polling as it was in Tango since it has been introduced. This means:
  - For one device, always poll attribute one at a time even if the polling period is the same (use of read_attribute instead of read_attributes)
  - Do not allow the polling thread to be late: If it is the case (because at the end of polling object 1, the time is greater than the polling date of object 2), discard polling object and inform event user by sending one event with error (Polling thread is late and discard. . . .)

- New polling algorithm introduced in Tango 9 as the default one. This means:
  - For one device, poll all attributes with the same polling period using a single device call (read_attributes)
  - Allow the polling thread to be late but only if number of late objects decreases.

The advantages of new polling algorithm are

1. In case of several attributes polled on the same device at the same period a lower device occupation time by the polling thread (due to a single read_attributes() call instead of several single read_attribute() calls)
2. Less "Polling thread late" errors in the event system in case of device with non constant response time

The drawback is

1. The loss of attribute individual timing data reported in the polling thread status

It is still possible to return to pre-release 9 polling algorithm. To do so, you can use the device server process administration device *polling_before_9* property by setting it to true. It is also possible to choose this pre-release 9 algorithm in device server process code in the main function of the process using the *Util::set_polling_before_9()* method.

Reading data from the polling buffer

For a polled command or a polled attribute, a client has three possibilities to get command result or attribute value (or the thrown exception) :

- From the device itself
- From the polling buffer
- From the polling buffer first and from the device if data in the polling buffer are invalid or if the polling is badly configured.

The choice is done during the command_inout CORBA operation by positioning one of the operation parameter. When reading data from the polling buffer, several error cases are possible

- The data in the buffer are not valid any more. Every time data are requested from the polling buffer, a check is done between the client request date and the date when the data were stored in the buffer. An exception is thrown if the delta is greater than the polling period multiplied by a "too old" factor. This factor has a default value and is up-datable via a device property. This is detailed in the reference part of this manual.
- The polling is correctly configured but there is no data yet in the polling buffer.

Retrieving command/attribute result history

The polling thread stores the command result or attribute value in circular buffers. It is possible to retrieve an history of the command result (or attribute value) from these polling buffers. Obviously the history is limited by the depth of the circular buffer. For commands, a CORBA operation called *command_inout_history_2* allows this retrieval. The client specifies the command name and the record number he want to retrieve. For each record, the call returns the date when the command was executed, the command result or the exception stack in case of the command failed when it was executed by the polling thread. In such a case, the exception stack is sent as a structure member and not as an exception. The same thing is available for attribute. The CORBA operation name is *read_attribute_history_2*. For these two calls, there is no check done between the call date and the record date in contrary of the call to retrieve the last command result (or attribute value).

Externally triggered polling

Sometimes, rather than polling a command or an attribute regulary with a fixed period, it is more interesting to manually decides when the polling must occurs. The Tango polling system also supports this kind of usage. This is called *externally triggered polling*. To define one attribute (or command) as externally triggered, simply set its polling period to 0. This can be done with the device server administration device AddObjPolling or UpdObjPollingPeriod command. Once in this mode, the attribute (or command) polling is triggered with the *trigger_cmd_polling()* method (or *trigger_attr_polling()* method) of the Util class. The following piece of code shows how this method could be used for one externally triggered command.

```
1  .....
2
3  string ext_polled_cmd("MyCmd");
4  Tango::DeviceImpl *device = .....;
5
6  Tango::Util *tg = Tango::Util::instance();
7
8  tg->trigger_cmd_polling(device,ext_polled_cmd);
9
10 .....
```

line 3 : The externally polled command name

line 4 : The device object

line 8 : Trigger polling of command MyCmd

Filling polling buffer

Some hardware to be interfaced already returned an array of pair value, timestamp. In order to be read with the *command_inout_history* or *read_attribute_history* calls, this array has to be transferred in the attribute or command polling buffer. This is possible only for attribute or command configured in the externally triggered polling mode. Once in externally triggered polling mode, the attribute (or command) polling buffer is filled with the *fill_cmd_polling_buffer()* method (or *fill_attr_polling_buffer()* method) of the Util class. For command, the user uses a template class called *TimedCmdData* for each element of the command history. Each element is stored in a stack in one instance of a template class called *CmdHistoryStack*. This object is one of the argument of the fill_cmd_polling_buffer() method. Obviously, the stack depth cannot be larger than the polling buffer depth. See [sub:The-device-polling-prop] to learn how the polling buffer depth is defined. The same way is used for attribute with the *TimedAttrData* and *AttrHistoryStack* template classes. These classes are documented in [TangoRefMan] . The following piece of code fills the polling buffer for a command called MyCmd which is already in externally triggered mode. It returns a DevVarLongArray data type with three elements. This example is not really something you will find in a real hardware interface. It is only to demonstrate the fill_cmd_polling_buffer() method usage. Error management has also been removed.

```
1  ....
2
3  Tango::DevVarLongArray dvla_array[4];
4
5  for(int i = 0;i < 4;i++)
6  {
7      dvla_array[i].length(3);
8      dvla_array[i][0] = 10 + i;
9      dvla_array[i][1] = 11 + i;
10     dvla_array[i][2] = 12 + i;
11 }
```

(continues on next page)

(continued from previous page)

```
12
13  Tango::CmdHistoryStack<DevVarLongArray> chs;
14  chs.length(4);
15
16  for (int k = 0;k < 4;k++)
17  {
18      time_t when = time(NULL);
19
20      Tango::TimedCmdData<DevVarLongArray> tcd(&dvla_array[k],when);
21      chs.push(tcd);
22  }
23
24  Tango::Util *tg = Tango::Util::instance();
25  string cmd_name("MyCmd");
26  DeviceImpl *dev = ....;
27
28  tg->fill_cmd_polling_buffer(dev,cmd_name,chs);
29
30  .....
```

Line 3-11 : Simulate data coming from hardware

Line 13-14 : Create one instance of the CmdHistoryStack class and reserve space for one history of 4 elements

Line 16-17 : A loop on each history element

Line 18 : Get date (hardware simulation)

Line 20 : Create one instance of the TimedCmdData class with data and date

Line 21 : Store this command history element in the history stack. The element order will be the insertion order whatever the element date is.

Line 28 : Fill command polling buffer

After one execution of this code, a command_inout_history() call will return one history with 4 elements. The first array element of the oldest history record will have the value 10. The first array element of the newest history record will have the value 13. A command_inout() call with the data source parameter set to CACHE will return the newest history record (ie an array with values 13,14 and 15). A command_inout() call with the data source parameter set to DEVICE will return what is coded is the command method. If you execute this code a second time, a command_inout_history() call will return an history of 8 elements.

The next example fills the polling buffer for an attribute called MyAttr which is already in externally triggered mode. It is a scalar attribute of the DevString data type. This example is not really something you will find in a real hardware interface. It is only to demonstrate the fill_attr_polling_buffer() method usage with memory management issue. Error management has also been removed.

```
1   ....
2
3   AttrHistoryStack<DevString> ahs;
4   ahs.length(3);
5
6   for (int k = 0;k < 3;k++)
7   {
8       time_t when = time(NULL);
9
```

(continues on next page)

(continued from previous page)

```
10      DevString *ptr = new DevString [1];
11      ptr = Tango::string_dup("Attr history data");
12
13      TimedAttrData<DevString> tad(ptr,Tango::ATTR_VALID,true,when);
14      ahs.push(tad);
15  }
16
17  Tango::Util *tg = Tango::Util::instance();
18  string attr_name("MyAttr");
19  DeviceImpl *dev = ....;
20
21  tg->fill_attr_polling_buffer(dev,attr_name,ahs);
22
23  .....
```

Line 3-4 : Create one instance of the AttrHistoryStack class and reserve space for an history with 3 elements

Line 6-7 : A loop on each history element

Line 8 : Get date (hardware simulation)

Line 10-11 : Create a string. Note that the DevString object is created on the heap

Line 13 : Create one instance of the TimedAttrData class with data and date requesting the memory to be released.

Line 14 : Store this attribute history element in the history stack. The element order will be the insertion order whatever the element date is.

Line 21 : Fill command polling buffer

It is not necessary to return the memory allocated at line 10. The *fill_attr_polling_buffer()* method will do it for you.

Setting and tuning the polling in a Tango class

Even if the polling is normally set and tuned with external tool like Jive, it is possible to set it directly into the code of a Tango class. A set of methods belonging to the *DeviceImpl* class allows the user to deal with polling. These methods are:

- *is_attribute_polled()* and *is_command_polled()* to check if one command/attribute is polled
- *get_attribute_poll_period()* and *get_command_poll_period()* to get polled object polling period
- *poll_attribute()* and *poll_command()* to poll command or attribute
- *stop_poll_attribute()* and *stop_poll_command()* to stop polling a command or an attribute

The following code snippet is just an exmaple of how these methods could be used. They are documented in [TangoRefMan].

```
1   void MyClass::read_attr(Tango::Attribute &attr)
2   {
3       ...
4       ...
5
6       string att_name("SomeAttribute");
7       string another_att_name("AnotherAttribute");
8
```

(continues on next page)

(continued from previous page)

```
        if (is_attribute_polled(att_name) == true)
            stop_poll_attribute(att_name);
        else
            poll_attribute(another_att_name,500);

        ....
        ....

    }

]
```

6.5.10 Generating events in a device server

Intended audience: advanced developers, ***Programming language:*** c++

The server is at the origin of events. It will fire events as soon as they occur. Standard events (*change*, *periodic* and *archive*) are detected automatically in the polling thread and fired as soon as they are detected. The *periodic* events can only be handled by the polling thread. *Change, Data ready* and *archive* events can also be pushed from the device server code. To allow a client to subscribe to events of non polled attributes the server has to declare that events are pushed from the code. Three methods are available for this purpose:

```
Attr::set_change_event(bool implemented, bool detect = true);
Attr::set_archive_event(bool implemented, bool detect = true);
Attr::set_data_ready_event( bool implemented);
```

where *implemented*=true indicates that events are pushed manually from the code and *detect*=true (when used) triggers the verification of the same event properties as for events send by the polling thread. When setting *detect*=false, no value checking is done on the pushed value! The class DeviceImpl also supports the first two methods with an addictional parameter attr_name defining the attribute name.

To push events manually from the code a set of data type dependent methods can be used:

```
DeviceImpl::push_change_event (string attr_name, ....);
DeviceImpl::push_archive_event(string attr_name, ....);
```

For the data ready event, a DeviceImpl class method has to be used to push the event.

```
DeviceImpl::push_data_ready_event(string attr_name,Tango::DevLong ctr);
```

See the class documentation for all available interfaces.

For non-standard events a single call exists for pushing the data to the CORBA Notification Service (omniNotify). Clients who are subscribed to this event have to know what data type is in the DeviceAttribute and unpack it accordingly.

To push non-standard events, use the following api call is available to all device servers :

```
DeviceImpl::push_event( string attr_name,
             vector<string> &filterable_names,
             vector<double> &filterable_vals,
             Attribute &att)
```

where *attr_name* is the name of the attribute. *Filterable_names* and *filterable_vals* represent any filterable data which can be used by clients to filter on. Here is a typical example of what a server will need to do to send its own events. We

are in the read method of the Sinusoide attribute. This attribute is readable as any other attribute but an event is sent if its value is positive when it is read. On top of that, this event is sent with one filterable field called value which is set to the attribute value.

```
1  void MyClass::read_Sinusoide(Tango::Attribute &attr)
2  {
3    ...
4       struct timeval tv;
5       gettimeofday(&tv, NULL);
6       sinusoide = 100 * sin( 2 * 3.14 * frequency * tv.tv_sec);
7
8       if (sinusoide >= 0)
9       {
10         vector<string> filterable_names;
11         vector<double> filterable_value;
12
13         filterable_names.push_back("value");
14         filterable_value.push_back((double)sinusoide);
15
16         push_event( attr.get_name(),
17                     filterable_names, filterable_value,
18                     &sinusoide);
19      }
20   ....
21   ....
22
23 }
```

line 13-14 : The filter pair name/value is initialised

line 16-18 : The event is pushed

6.5.11 Tango Device in Java

For instructions on how to use JTango, the Java binding for Tango Controls, please refer to the JTango documentation.

6.6 Debugging and Testing

Intended audience: developers, ***Programming language:*** all

In the following articles you will find useful information on testing and debugging of your code.

6.6.1 Using Tango docker containers

Intended audience: advanced developers, ***Programming language:*** all

In this section we describe how one can test newly developed tango device server using docker containers.

Tango docker containers provide a lightweight solution for deploying tango, especially useful for local workstation development.

To get info on how to install docker on your machine please refer to the docker documentation.

Once docker is installed one can pull docker images with pre-installed tango.

Tango docker containers

SKAO provides several docker containers for Tango development and deployment, including but not limited to:

- tango-db: a (mysql-compatible) mariadb container with the Tango schema
- tango-cpp: a container with core Tango libraries and dependencies that can run the *DatabaseDS* Device Server
- tango-rest: a Tango REST server
- tango-test: the well-established TangoTest device server

These container images are hosted on SKAO Nexus service: https://artefact.skao.int/#browse/browse:docker-all from where they can be pulled with the prefix `artefact.skao.int/`

The sources for these docker images are hosted and maintained on the SKAO gitlab: https://gitlab.com/ska-telescope/ska-tango-images

Tango docker stack

The easiest way to setup the whole stack on a local development/test computer is to use docker compose

Here is a very small example of docker-compose.yml which runs up the DB and the Database Device Server:

```
version: '3'
networks:
  tango-net:
    name: tango-net
    driver: bridge

services:
  tango-db:
    image: artefact.skao.int/ska-tango-images-tango-db:10.4.19
    platform: linux/x86_64
    networks:
      - tango-net
    ports:
      - "9999:3306"
    environment:
      - MARIADB_ROOT_PASSWORD=root
      - MARIADB_DATABASE=tango
      - MARIADB_USER=tango
      - MARIADB_PASSWORD=tango

  tango-dbds:
    image: artefact.skao.int/ska-tango-images-tango-cpp:9.4.1
    platform: linux/x86_64
    networks:
      - tango-net
    ports:
      - "10000:10000"
    environment:
      - TANGO_HOST=localhost:10000
      - MYSQL_HOST=tango-db:3306
      - MYSQL_USER=tango
      - MYSQL_PASSWORD=tango
```

(continues on next page)

(continued from previous page)

```
      - MYSQL_DATABASE=tango
    links:
      - "tango-db:localhost"
    depends_on:
      - tango-db
    # Alternative to use health-check as a depends_on/condition
    # is to wait for port to be open:
    entrypoint:
      - /usr/local/bin/wait-for-it.sh
      - tango-db:3306 --timeout=30 --strict
      - --
      - /usr/local/bin/DataBaseds 2 -ORBendPoint giop:tcp::10000
```

To start the whole stack execute `docker compose up` in the same directory as the above docker-compose yaml file.

Note: *docker compose* (V2) is installed by default with Docker Desktop. However, if you have another type of docker/container installation then you may be able to use *docker-compose up* instead - or install the newer docker compose V2 manually from here: https://docs.docker.com/compose/

Once docker containers are up and running one can access the database on `localhost:9999` and the tango host on `localhost:10000`

For instance, one can start jive (assuming it is installed on the system) and use the Jive GUI to interact with the Device Servers, running in the containers.

Tango docker stack for Tango REST API

One can setup Tango docker stack for Tango REST API as well. SKAO provides a containerized build of the Tango Rest Server.

The following `compose.yaml` compose.yaml assembles the whole stack:

```
version: '3'
networks:
  tango-net:
    name: tango-net
    driver: bridge

services:
  tango-db:
    image: artefact.skao.int/ska-tango-images-tango-db:10.4.19
    platform: linux/x86_64
    networks:
      - tango-net
    ports:
      - "9999:3306"
    environment:
      - MARIADB_ROOT_PASSWORD=root
      - MARIADB_DATABASE=tango
      - MARIADB_USER=tango
      - MARIADB_PASSWORD=tango
```

(continues on next page)

(continued from previous page)

```
  tango-dbds:
    image: artefact.skao.int/ska-tango-images-tango-cpp:9.4.1
    platform: linux/x86_64
    networks:
      - tango-net
    ports:
      - "10000:10000"
    environment:
      - TANGO_HOST=localhost:10000
      - MYSQL_HOST=tango-db:3306
      - MYSQL_USER=tango
      - MYSQL_PASSWORD=tango
      - MYSQL_DATABASE=tango
    links:
      - "tango-db:localhost"
    depends_on:
      - tango-db
    # The proper way to wait for a service to be ready is to use a container health-
↪check.
    # However, this is not yet available in the container image. So an alternative␣
↪approach
    # is to wait for port to be open:
    entrypoint:
      - /usr/local/bin/wait-for-it.sh
      - tango-db:3306 --timeout=30 --strict
      - --
      - /usr/local/bin/DataBaseds 2 -ORBendPoint giop:tcp:0.0.0.0:10000

  tango-rest:
    image: artefact.skao.int/ska-tango-images-tango-rest:1.14.8
    platform: linux/x86_64
    networks:
      - tango-net
    ports:
      - "8080:8080"
    environment:
      - TANGO_HOST=tango-dbds:10000
    links:
      - "tango-dbds:localhost"
    depends_on:
      - tango-dbds
    entrypoint:
      - /usr/local/bin/wait-for-it.sh
      - tango-dbds:10000 --timeout=30 --strict
      - --
      - /usr/bin/supervisord --configuration /etc/supervisor/supervisord.conf

  tango-test:
    image: artefact.skao.int/ska-tango-images-tango-test:3.0.5
    platform: linux/x86_64
    networks:
```

(continues on next page)

(continued from previous page)

```
      - tango-net
    environment:
      - TANGO_HOST=tango-dbds:10000
    links:
      - "tango-dbds:localhost"
    depends_on:
      - tango-dbds
    entrypoint:
      - /usr/local/bin/wait-for-it.sh
      - tango-dbds:10000 --timeout=30 --strict
      - --
      - /usr/local/bin/TangoTest test
```

Note this is almost the same as the previous, except we have added tango-rest and tango-test containers. A few seconds after the stack has been started with `docker compose up`, the *Tango REST API* can be accessed at http://localhost:8080/tango/rest/rc4/

The default username/password is `tango-cs/tango` and an example query from the host might look like this:

```
curl -s -u "tango-cs:tango" http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/
↪devices/dserver/TangoTest/test/ | python3 -m json.tool
{
    "name": "dserver/TangoTest/test",
    "info": {
        "last_exported": "3rd April 2023 at 14:59:32",
        "last_unexported": "3rd April 2023 at 14:59:23",
        "name": "dserver/tangotest/test",
        "ior":
↪"IOR:010000001700000049444c3a54616e676f2f4465766963655f353a312e3000000100000000000000ad0000000101020000
↪",
        "version": "5",
        "exported": true,
        "pid": 1,
        "server": "TangoTest/test",
        "hostname": "64195fa9b745",
        "classname": "unknown",
        "is_taco": false
    },
    "attributes": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test//attributes",
    "commands": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test//commands",
    "pipes": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test//pipes",
    "properties": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test//properties",
    "state": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test//state",
    "_links": {
        "_self": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/TangoTest/test/",
        "_parent": "http://localhost:8080/tango/rest/rc4/hosts/tango-dbds/10000/devices/
↪dserver/"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

6.7 Advanced

Intended audience: developers

There you will find articles on advanced topics.

Contents:

6.7.1 Threading

Intended audience: developers, ***Programming language:*** all

When used with C++, Tango used omniORB as underlying ORB. This CORBA implementation is a threaded implementation and therefore a C++ Tango device server or client are multi-threaded processes.

Device server process

A classical Tango device server without any connected clients has eight threads. These threads are :

- The main thread waiting in the ORB main loop
- Two ORB implementation threads (the POA thread)
- The ORB scavanger thread
- The signal thread
- The heartbeat thread (needed by the Tango event system)
- Two Zmq implementation threads

On top of these eight threads, you have to add the thread(s) used by the polling threads pool. This number depends on the polling thread pool configuration and could be between 0 (no polling at all) and the maximun number of threads in the pool.

A new thread is started for each connected client. Device server are mostly used to interface hardware which most of the time does not support multi-threaded access. Therefore, all remote calls executed from a client are serialized within the device server code by using mutual exclusion. See chapter [sub:Serialization-model-within] on which serialization model are available. In order to limit thread number, the underlying ORB (omniORB) is configured to shutdown threads dedicated to client if the connection is inactive for more than 3 minutes. To also limit thread number, the ORB is configured to create one thread per connection up to 55 threads. When this level is reached, the threading model is automatically switch to a thread pool model with up to 100 threads. If the number of threads decrease down to 50, the threading model will return to thread per connection model.

If you are using event, the event system for its internal heartbeat system periodically (every 200 seconds) sends a command to the device server administration device. As explained above, a thread is created to execute these command. The omniORB scavanger will terminate this thread before the next event system heartbeat command arrives. For example, if you have a device server with three connected clients using only event, the process thread number will permanently change between 8 and 11 threads.

In summary, the number of threads in a device server process can be evaluated with the following formula:

8 + k + m

k is the number of polling threads used from the polling threads pool and m is the number of threads used for connected clients.

Serialization model within a device server

Four serialization models are available within a device server. These models protect all requests coming from the network but also requests coming from the polling thread. These models are:

1. Serialization by device. All access to the same device are serialized. As an example, let's take a device server implementing one class of device with two instances (dev1 and dev2). Two clients are connected to these devices (client1 and client2). Client2 will not be able to access dev1 if client1 is using it. Nevertheless, client2 is able to access dev2 while client1 access dev1 (There is one mutual exclusion object by device)
2. Serialization by class. With non multi-threaded legacy software, the preceding scenario could generate problem. In this mode of serialization, client2 is not able to access dev2 while client1 access dev1 because dev2 and dev1 are instances of the same class (There is one mutual exclusion object by class)
3. Serialization by process. This is one step further than the previous case. In this mode, only one client can access any device embedded within the device server at a time. There is only one mutual exclusion object for the whole process)
4. No serialization. This is an exotic kind of serialization and **should be used with extreme care** only with device which are fully thread safe. In this model, most of the device access are not serialized at all. Due to Tango internal structure, the *get_attribute_config*, *set_attribute_config*, *read_attributes* and *write_attributes* CORBA calls are still protected. Reading the device state and status via commands or via CORBA attribute is also protected.

By default, every Tango device server is in serialization by device mode. A method of the Tango::Util class allows to change this default behavior.

```
#include <tango.h>

int main(int argc,char *argv[])
{

    try
    {

        Tango::Util *tg = Tango::Util::init(argc,argv);

        tg->set_serial_model(Tango::BY_CLASS);

        tg->server_init();

        cout << "Ready to accept request" << endl;
        tg->server_run();
    }
    catch (bad_alloc)
    {
         cout << "Can't allocate memory!!!" << endl;
         cout << "Exiting" << endl;
    }
    catch (CORBA::Exception &e)
    {
```

(continues on next page)

(continued from previous page)

```
25            Tango::Except::print_exception(e);
26
27            cout << "Received a CORBA::Exception" << endl;
28            cout << "Exiting" << endl;
29        }
30
31        return(0);
32    }
```

The serialization model is set at line 11 before the server is initialized and the infinite loop is started. See [TangoRefMan] for all details on the methods to set/get serialization model.

Attribute Serialization model

Even with the serialization model described previously, in case of attributes carrying a large number of data and several clients reading this attribute, a device attribute serialization has to be followed. Without this level of serialization, for attribute using a shared buffer, a thread scheduling may happens while the device server process is in the CORBA layer transferring the attribute data on the network. Three serialization models are available for attribute serialization. The default is well adapted to nearly all cases. Nevertheless, if the user code manages several attributes data buffer or if it manages its own buffer protection by one way or another, it could be interesting to tune this serialization level. The available models are:

1. Serialization by kernel. This is the default case. The kernel is managing the serialization
2. Serialization by user. The user code is in charge of the serialization. This serialization is done by the use of a omni_mutex object. An omni_mutex is an object provided by the omniORB package. It is the user responsability to lock this mutex when appropriate and to give this mutex to the Tango kernel before leaving the attribute read method
3. No serialization.

By default, every Tango device attribute is in serialization by kernel. Methods of the Tango::Attribute class allow to change the attribute serialization behavior and to give the user omni_mutex object to the kernel.

```
1   void MyClass::init_device()
2   {
3      ...
4      ...
5      Tango::Attribute &att = dev_attr->get_attr_by_name("TheAttribute");
6      att.set_attr_serial_model(Tango::ATTR_BY_USER);
7      ....
8      ....
9
10  }
11
12
13  void MyClass::read_TheAttribute(Tango::Attribute &attr)
14  {
15     ....
16     ....
17     the_mutex.lock();
18     ....
19     // Fill the attribute buffer
```

(continues on next page)

(continued from previous page)

```
20      ....
21      attr.set_value(buffer,....);
22      attr->set_user_attr_mutex(&the_mutex);
23  }
```

The serialization model is set at line 6 in the init_device() method. The user omni_mutex is passed to the Tango kernel at line 22. This omni_mutex object is a device data member. See [TangoRefMan] for all details on the methods to set attribute serialization model.

Client process

Clients are also multi threaded processes. The underlying C++ ORB (omniORB) try to keep system resources to a minimum. To decrease process file descriptors usage, each connection to server is automatically closed if it is idle for more than 2 minutes and automatically re-opened when needed. A dedicated thread is spawned by the ORB to manage this automatic closing connection (the ORB scavenger thread).

Threrefore, a Tango client has two threads which are:

1. The main thread
2. The ORB scavanger thread

If the client is using the event system and as Tango is using the event push-push model, it has to be a server for receiving the events. This increases the number of threads. The client now has 6 threads which are:

- The main thread
- The ORB scavenger thread
- Two Zmq implementation threads
- Two Tango event system related threads (the KeepAliveThread and the EventConsumer thread)

6.7.2 Transferring images

Intended audience: developers, ***Programming language:*** c++

Some optimized methods have been written to optimize image transfer between client and server using the attribute DevEncoded data type. All these methods have been merged in a class called EncodedAttribute. Within this class, you will find methods to:

- Encode an image in a compressed way (JPEG) for images coded on 8 (gray scale), 24 or 32 bits
- Encode a grey scale image coded on 8 or 16 bits
- Encode a color image coded on 24 bits
- Decode images coded on 8 or 16 bits (gray scale) and returned a 8 or bits grey scale image
- Decode color images transmitted using a compressed format (JPEG) and returns a 32 bits RGB image

The following code snippets are examples of how these methods have to be used in a server and in a client. On the server side, creates an instance of the EncodedAttribute class within your object

```
1  class MyDevice::Tango::Device_4Impl
2   {
3       ...
4       Tango::EncodedAttribute jpeg;
```

(continues on next page)

(continued from previous page)

```
5       ...
6   }
```

In the code of your device, use an encoding method of the EncodedAttribute class

```
1   void MyDevice::read_Encoded_attr_image(Tango::Attribute &att)
2   {
3       ....
4       jpeg.encode_jpeg_gray8(imageData,256,256,50.0);
5       att.set_value(&jpeg);
6   }
```

Line 4: Image encoding. The size of the image is 256 by 256. Each pixel is coded using 8 bits. The encoding quality is defined to 50 in a scale of 0 - 100. imageData is the pointer to the image data (pointer to unsigned char)

Line 5: Set the value of the attribute using a *Attribute::set_value()* method.

On the client side, the code is the following (without exception management)

```
1     ....
2     DeviceAttribute da;
3     EncodedAttribute att;
4     int width,height;
5     unsigned char *gray8;
6
7     da = device.read_attribute("Encoded_attr_image");
8     att.decode_gray8(&da,&width,&height,&gray8);
9     ....
10    delete [] gray8;
11    ...
```

The attribute named Encoded_attr_image is read at line7. The image is decoded at line 8 in a 8 bits gray scale format. The image data are stored in the buffer pointed to by gray8. The memory allocated by the image decoding at line 8 is returned to the system at line 10.

6.7.3 Device server with user defined event loop

Intended audience: developers, ***Programming language:*** c++

Sometimes, it could be usefull to write your own process event handling loop. For instance, this feature can be used in a device server process where the ORB is only one of several components that must perform event handling. A device server with a graphical user interface must allow the GUI to handle windowing events in addition to allowing the ORB to handle incoming requests. These types of device server therefore perform non-blocking event handling. They turn the main thread of control over each of the vvarious event-handling sub-systems while not allowing any of them to block for significants period of time. The *Tango::Util* class has a method called *server_set_event_loop()* to deal with such a case. This method has only one argument which is a function pointer. This function does not receive any argument and returns a boolean. If this boolean is true, the device server process exits. The device server core will call this function in a loop without any sleeping time between the call. It is the user responsability to implement in this function some kind of sleeping mechanism in order not to make this loop too CPU consuming. The code of this function is executed by the device server main thread. The following piece of code is an example of how you can use this feature.

```
1  bool my_event_loop()
2  {
3     bool ret;
4
5     some_sleeping_time();
6
7     ret = handle_gui_events();
8
9     return ret;
10 }
11
12 int main(int argc,char *argv[])
13 {
14    Tango::Util *tg;
15    try
16    {
17       // Initialise the device server
18       //----------------------------------------
19       tg = Tango::Util::init(argc,argv);
20
21       tg->set_polling_threads_pool_size(5);
22
23       // Create the device server singleton
24       //        which will create everything
25       //----------------------------------------
26       tg->server_init(false);
27
28       tg->server_set_event_loop(my_event_loop);
29
30       // Run the endless loop
31       //----------------------------------------
32       cout << "Ready to accept request" << endl;
33       tg->server_run();
34    }
35    catch (bad_alloc)
36    {
37    ...
```

The device server main event loop is set at line 29 before the call to the Util::server_run() method. The function used as server loop is defined between lines 2 and 11.

6.7.4 Tango Device Server Model

Intended audience: advanced, developers, ***Programming language:*** all

Primary Presentation

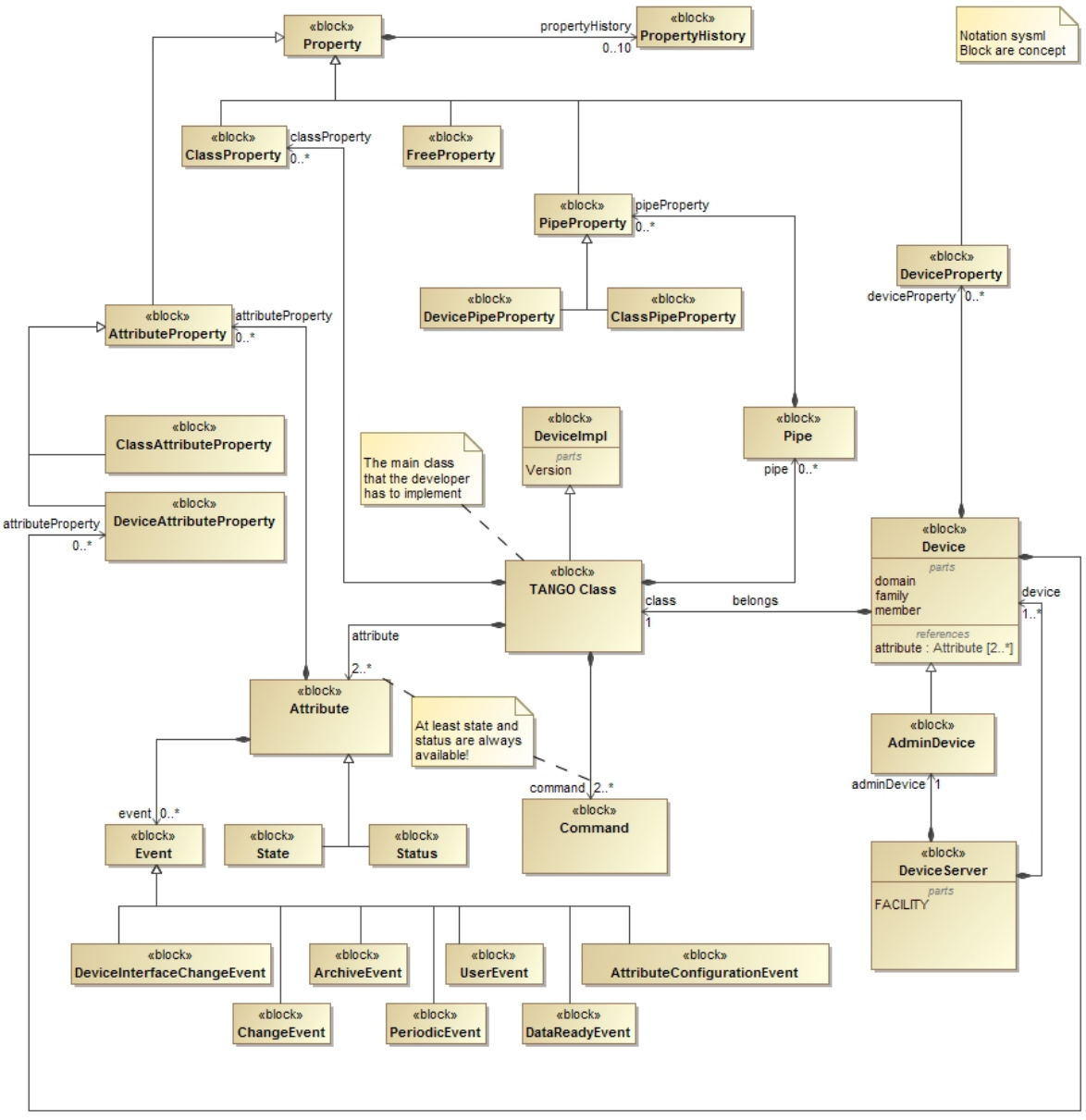


Figure 1: Tango Device Server Model

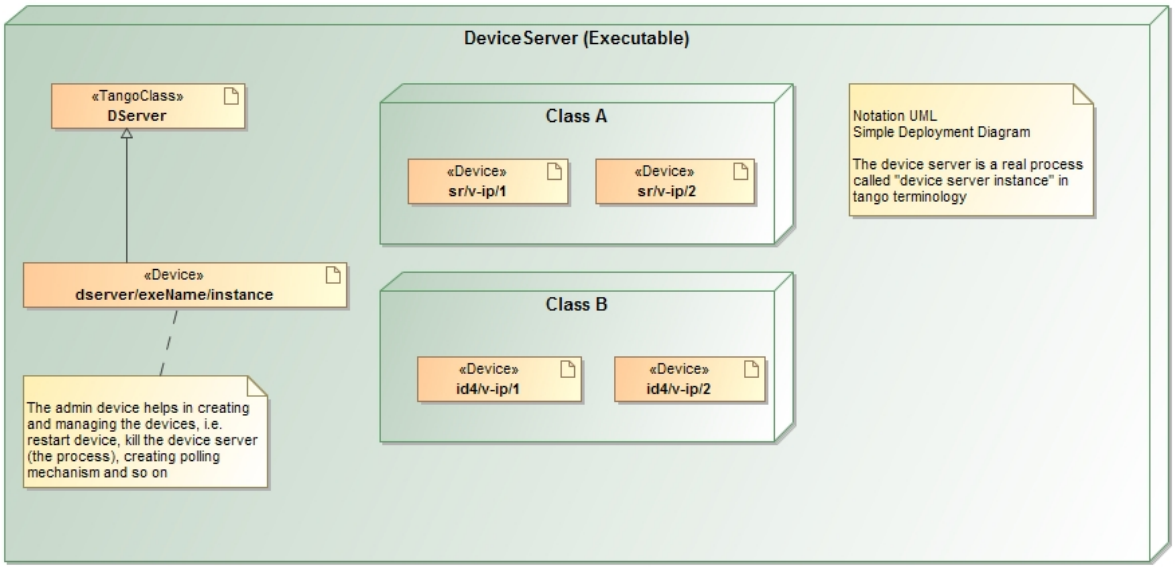


Figure 2: Runtime representation of a Device server

Elements

Block	Description
Device	Abstract concept defined by the TANGO device server object model; it can be a piece of hardware (an interlock bit) a collection of hardware (a screen attached to a stepper motor) a logical device (a taper) or a combination of all these (an accelerator). Please refer also to the *Glossary*, *device*.
TANGO Class	From Object Oriented Programming concept, this is the main class that the developer has to implement
DeviceServer	The server (also referred as device server) is a process whose main task is to offer one or more services to one or more clients. To do this, the server has to spend most of its time in a wait loop waiting for clients to connect to it. The devices are hosted in the server process. A server is able to host several classes of devices. In short, it is a process that export devices available to accept requests). Please refer also to the *Glossary*, *device server instance*.
Property	Store a generic configuration
DeviceProperty	Device specific configuration
ClassProperty	Class specific configuration
AttributeProperty	Attribute specific configuration
ClassAttributeProperty	Attribute specific configuration at class level
DeviceAttributeProperty	Specific configuration for a specific attribute of a specific device
FreeProperty	User-defined specific configuration (for instance GUI, generic system and so on)

continues on next page

Table 1 – continued from previous page

Block	Description
PropertyHistory	History of the values for a property (maximum 10 latest are stored for each property)
Attribute	See *Glossary*, *attribute*.
AttributeAlias	One word which can be used to identify a specific attribute. (shortcut)
Pipe	See *Glossary*, *pipe*.
PipeProperty	Pipe specific configuration
DevicePipeProperty	Configuration of a specific pipe of a specific device
ClassPipeProperty	Configuration of a specific pipe for a specific class
Event	Refer to *Events*.
Command	See *Glossary*; A list of default command are available for the admin device at section *DServer*.
ChangeEvent	It is a type of event that gets fired when the associated attribute changes its value according to its configuration specified in system specific attribute properties (abs_change and rel_change); Refer to *Events*.
ArchiveEvent	It is a type of event that gets fired when the associated attribute should be archived according to its configuration specified in system specific attribute properties (archive_abs_change, archive_rel_change and archive_period); Refer to *Events*.
UserEvent	It is a type of event that gets fired when the device server programmer wants to; Refer to *Events*.
PeriodicEvent	It is a type of event that gets fired at a fixed periodic interval; Refer to *Events*.
DataReadyEvent	It is a type of event that gets fired to inform a client that it is now possible to read an attribute; Refer to *Events*.
AttributeConfigurationEvent	It is a type of event that gets fired if the attribute configuration is changed; Refer to *Events*.
DeviceInterfaceChangeEvent	It is a type of event that gets fired when the device interface changes; Refer to *Events*.
DeviceImpl	Base implementation of every class that will become a device.
State	The device state is a number which reflects the availability of the device. Refer to *Events*
Status	The state of the device as a formatted ascii string
AdminDevice	Special type of Device dedicated to creating and managing the devices, i.e. restart device, kill the device server (the process), creating polling mechanism and so on

Attributes

Block	Attribute	Description
Device	name	Correspond to "Domain/family/member"
	alias	A word that you can use to access the device, like a shortcut. Device name alias(es) must also be unique within a control system. There is no predefined syntax for device name alias.
	domain	Each device has a unique name in the control system. Within Tango, a four field name space has been adopted consisting of [//FACILITY/]DOMAIN/CLASS/MEMBER Facility refers to the control system instance, domain refers to the sub-system, class the class and member the instance of the device.
	family	
	member	
	version	It correspond to the version of base device implementation class (for backward compatibility). It is used to know how to communicate with this device and what features are supported.
	class	Name of the class corresponding to the Device
	ior	Orb Identifier used to connect with the device
	host	Where the device is running
	pid	Id of the specific process
	exported	Means that the device is available to accept request
DeviceServer	Facility	Represent the host where the device server instance (aka process) lives
Attribute	alias	A shortcut that you can use to access the attribute (in the Database there is a specific table)

Relations

Left Block	Right Block	Multi-plicity	Description
Device	TANGO Class	1	Every device belongs to a Tango class
At-tribute	ClassAt-tributeProp-erty	0..*	An attribute can have more than one class attribute property associated
At-tribute	Event	0..*	An attribute can have more than one event associated
Device	DeviceAt-tributeProp-erty	0..*	A device can have more than one Device Attribute Property associated
Device	DeviceProp-erty	0..*	A device can have more than one Device Property associated
Device	Attribute	2..*	A Device can have more than one Attribute associated via reference
Device-Server	AdminDevice	1	Every device server is exporting one admin device
Device-Server	Device	1..*	Every Device server has many devices inside itself
Pipe	PipeProperty	0..*	A pipe can have more than one Pipe Property associated
Property	PropertyHis-tory	0..10	A Property can have more than one Property history associated (this will maintain the history of the change for the property)
TANGO Class	Attribute	2..*	A TANGO Class can have more than one Attribute associated
TANGO Class	ClassProperty	0..*	A TANGO Class can have more than one Class Property associated
TANGO Class	Command	2..*	A TANGO Class can have more than one Command associated
TANGO Class	Pipe	0..*	A TANGO Class can have more than one Pipe associated

Rationale

Please refer to the *Tango Device Model*.

6.7.5 Tango REST API

Intended audience: developers, ***Programming language:*** all

Tango provides REST API specification. This can be used to implement integrations of Tango with 3rd party products using http protocol instead of tango protocol.

Simple example of such 3rd product can be a mobile client application for monitoring Tango.

This section tries to answer the following questions:

- What is REST and why it is useful
- Getting started with simple example
- Example of a safe REST API deployment at ESRF
- Further steps

Tango REST API specification

Shortly speaking REST API in general exports resources. These resources are accessible using URL addressing. Client performs actions on the resources defined by http methods. Typically these are:

GET - request a resource

PUT - update a resource

POST - create a new resource

DELETE - remove a resource

Tango REST API exports Tango hosts; Tango devices; Tango device attributes, commands and pipes. So it follows *Tango Device Server Model*.

For example, one can request *Tango Host* using Tango REST API:

GET tango/rest/rc4/hosts/tango_host/10000

Here one uses Tango REST API implementation related to version RC4 [TODO link to the branch] this relates to tango/rest/rc4 part of the exmaple URL. *hosts* exports all the :term:<Tango Hosts *Tango Host*> accessible through this Tango REST API implementation. tango_host/10000 relates to TANGO_HOST=tango_host:10000.

In the following example client requests for a device attribute:

GET tango/rest/rc4/hosts/tango_host/10000/devices/sys/tg_test/1/attributes/double_scalar/value

Again here *tango/rest/rc4* relates to Tango REST API version RC4 implementation; *hosts/tango_host/10000* relates to TANGO_HOST=tango_host:10000; *devices/sys/tg_test/1* relates to the device *tango://tango_host:10000/sys/tg_test/1* exported through this Tango REST API; finally *attributes/double_scalar/value* relates to the value of the attribute.

The third example is how client executes a command of a device:

PUT tango/rest/rc4/hosts/tango_host/10000/devices/sys/tg_test/1/commands/DevDouble?v=3.14

Here client executes tango://tango_host:10000/sys/tg_test/1/DevDouble with the input arg=3.14

The full reference can be found following this link

Tango REST API implementations

Since Tango REST API itself is only a specification one needs an actual implementation running some where.

Known implementations are listed in the Tango REST API readme.

Please refer to the corresponding implementation documentation on how to install and use it.

In a nutshell download the latest fat jar distribution and start the server via **java -DTANGO_HOST=localhost:10000 -jar mtangorest.server.jar -nodb -dlist sys/rest/0**

Basically one can test the installation trying to read an attribute value from a device. Typically there is a TangoTest server running on the tango host through browser:

http://localhost:10001/tango/rest/rc4/hosts/localhost/10000/devices/sys/tg_test/1/attributes/double_scalar/value


```
{
    "name":"double_scalar",
    "value":179.04696279859678,
    "quality":"ATTR_VALID",
```


(continues on next page)

(continued from previous page)

```
    "timestamp":1493918496122
}
```

Deployment

As Tango REST is supposed to export Tango via http to the Internet the usual question is how to protect Tango from the unwanted activity.

The deployment of the Tango REST API can be quite safe. Usually one wants to put Tango REST API server behind a reverse proxy and restrict its access to a single *Tango Host*. Reverse proxy can also allow connections only via https.

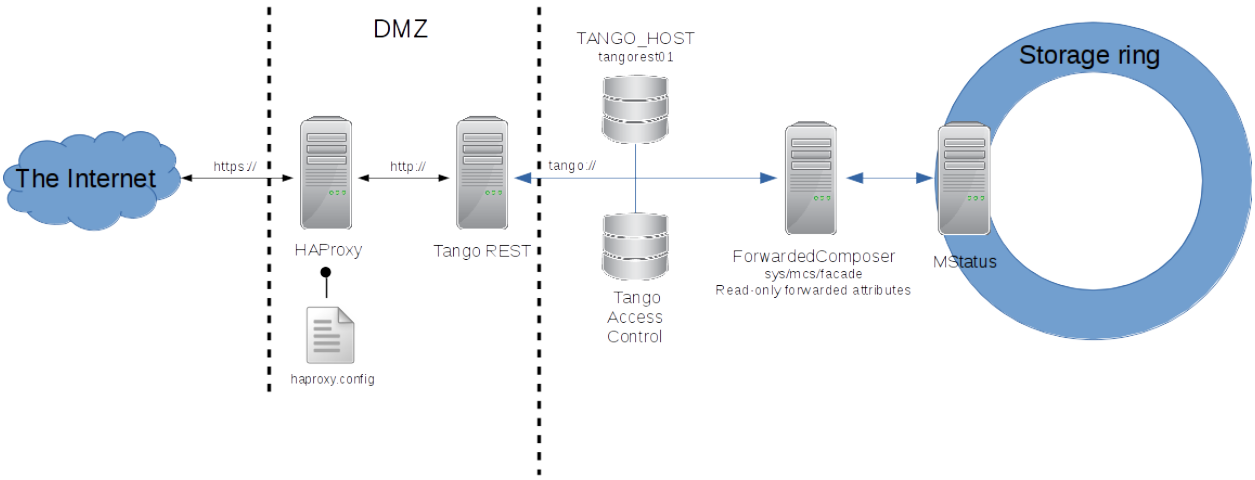
As every request via REST API must be validated against Tango Access Control this adds an extra layer of security.

Below is a deployment scheme of REST API at ESRF:

https://mstatus.esrf.fr/tango/rest

Use tango-cs/tango when prompted.

In this installation REST API exports readonly forwarded attributes and is accessible via secured http connection.



Every request passes HAProxy configured to use https protocol for secure connection. On its backend HAproxy speaks with Tango REST server which in turn can access only one tango host where a device of class ForwardComposer is defined. This device provides read only access to MStatus Tango device with status information about the storage ring at ESRF.

In addition Tango REST API can be integrated with authentication and authorisation services like kerberos.

Finally Tango REST API implementation should use Tango Access Control to validate every request made from the Internet.

Further steps

Install Tango REST API server localy or using docker.

Develop your REST client or use 3rd party frameworks (*see Tango web tools*).

Deploy everything in the local network or in the cloud.

References

[1] Tango REST API specification

[2] Tango REST API specification on GitHub

[3] *REST API installation tutorial, full stack*

[4] Tango REST Server wiki

6.7.6 The TANGO IDL file : Module Tango

Intended audience: developers, ***Programming language:*** all

The fundamental idea of a device as a network object which has methods and data has been retained for TANGO. In TANGO objects are real C++/Java objects which can be instantiated and accessed via their methods and data by the client as if they were local objects. This interface is defined in CORBA IDL. The fundamental interface is Device. All TANGO control objects will be of this type i.e. they will implement and offer the Device interface. Some wrapper classes group in an API will hide the calls to the Device interface from the client so that the client will only see the wrapper classes. All CORBA details will be hidden from the client as far as possible.

Aliases

AttributeConfigList

typedef sequence<AttributeConfig> AttributeConfigList;

AttributeConfigList_2

typedef sequence<AttributeConfig_2> AttributeConfigList_2;

AttributeConfigList_3

typedef sequence<AttributeConfig_3> AttributeConfigList_3;

AttributeConfigList_5

typedef sequence<AttributeConfig_5> AttributeConfigList_5;

AttributeDimList

typedef sequence<AttributeDim> AttributeDimList;

AttributeValueList

typedef sequence<AttributeValue> AttributeValueList;

AttributeValueList_3

typedef sequence<AttributeValue_3> AttributeValueList_3;

AttributeValueList_4

typedef sequence<AttributeValue_4> AttributeValueList_4;

AttributeValueList_5

typedef sequence<AttributeValue_5> AttributeValueList_5;

AttrQualityList

typedef sequence<AttrQuality> AttrQualityList;

CppClntIdent

typedef unsigned long CppClntIdent;

DevAttrHistoryList

typedef sequence<DevAttrHistory> DevAttrHistoryList;

DevAttrHistoryList_3

typedef sequence<DevAttrHistory_3> DevAttrHistoryList_3;

DevBoolean

typedef boolean DevBoolean;

DevCmdHistoryList

typedef sequence<DevCmdHistory> DevCmdHistoryList

DevCmdInfoList

typedef sequence<DevCmdInfo> DevCmdInfoList;

DevCmdInfoList_2

typedef sequence<DevCmdInfo_2> DevCmdInfoList_2;

DevDouble

typedef double DevDouble;

DevErrorList

typedef sequence<DevError> DevErrorList;

DevErrorListList

typedef sequence<DevErrorList> DevErrorListList;

DevFloat

typedef float DevFloat;

DevLong

typedef long DevLong;

DevShort

typedef short DevShort;

DevString

typedef string DevString;

DevULong

typedef unsigned long DevULong;

DevUShort

typedef unsigned short DevUShort;

DevVarCharArray

typedef sequence<octet> DevVarCharArray;

DevVarDoubleArray

typedef sequence<double> DevVarDoubleArray;

DevVarEncodedArray

typedef sequence<DevEncoded> DevVarEncodedArray;

DevVarFloatArray

typedef sequence<float> DevVarFloatArray;

DevVarLongArray

typedef sequence<long> DevVarLongArray;

DevVarPipeDataEltArray

typedef sequence<DevPipeDataElt> DevVarPipeDataEltArray;

DevVarShortArray

typedef sequence<short> DevVarShortArray;
DevVarStateArray

typedef sequence<DevState> DevVarStateArray;

DevVarStringArray

typedef sequence<string> DevVarStringArray;

DevVarULongArray

typedef sequence<unsigned long> DevVarULongArray;

DevVarUShortArray

typedef sequence<unsigned short> DevVarUShortArray;

EltInArrayList

typedef sequence<EltInArray> EltInArrayList;

JavaUUID

typedef unsigned long long JavaUUID[2];
PipeConfigList

typedef sequence<PipeConfig> PipeConfigList;
NamedDevErrorList

typedef sequence<NamedDevError> NamedDevErrorList;

TimeValList

typedef sequence<TimeVal> TimeValList;

Enums

AttrDataFormat

```
enum AttrDataFormat
{
  SCALAR,
  SPECTRUM,
  IMAGE,
  FMT_UNKNOWN
```

};

AttributeDataType

```
enum AttributeDataType
{
  ATT_BOOL,
  ATT_SHORT,
  ATT_LONG,
  ATT_LONG64,
  ATT_FLOAT,
  ATT_DOUBLE,
  ATT_UCHAR,
  ATT_USHORT,
  ATT_ULONG,
  ATT_ULONG64,
  ATT_STRING,
  ATT_STATE,
  DEVICE_STATE,
  ATT_ENCODED,
  ATT_NO_DATA

};
```

AttrQuality

```
enum AttrQuality
{
  ATTR_VALID,
  ATTR_INVALID,
  ATTR_ALARM,
  ATTR_CHANGING,
  ATTR_WARNING

};
```

AttrWriteType

```
enum AttrWriteType
```

```
{
  READ,
  READ_WITH_WRITE,
  WRITE,
  READ_WRITE,
  WT_UNKNOWN

};
```

DispLevel

```
enum DispLevel
{
  OPERATOR,
  EXPERT,
  DL_UNKNOWN

};
```

DevSource

```
enum DevSource
{
  DEV,
  CACHE,
  CACHE_DEV

};
```

DevState

```
enum DevState
{
  ON,
  OFF,
  CLOSE,
  OPEN,
  INSERT,
  EXTRACT,
```

```
  MOVING,
  STANDBY,
  FAULT,
  INIT,
  RUNNING,
  ALARM,
  DISABLE,
  UNKNOWN

};
```

ErrSeverity

```
enum ErrSeverity
{
  WARN,
  ERR,
  PANIC

};
```

LockerLanguage

```
enum LockerLanguage
{
  CPP,
  JAVA

};
```

PipeWriteType

```
enum PipeWriteType
{
  PIPE_READ,
  PIPE_READ_WRITE,
  PIPE_WT_UNKNOWN
};
```

Structs

ArchiveEventProp

```
struct ArchiveEventProp
{
  string rel_change;
  string abs_change;
  string period;
  DevVarStringArray extensions;
        };
```

AttributeAlarm

```
struct AttributeAlarm
{
  string min_alarm;
  string max_alarm;
  string min_warning;
  string max_warning;
  string delta_t;
  string delta_val;
  DevVarStringArray extensions;
        };
```

AttDataReady

```
struct AttributeAlarm {
  string name;
  long data_type;
  long ctr;
}
```

AttributeConfig

```
struct AttributeConfig
{
  string name;
  AttrWriteType writable;
  AttrDataFormat data_format;
  long data_type;
  long max_dim_x;
  long max_dim_y;
  string description;
```

```
  string label;
  string unit;
  string standard_unit;
  string display_unit;
  string format;
  string min_value;
  string max_value;
  string min_alarm;
  string max_alarm;
  string writable_attr_name;
  DevVarStringArray extensions;
};
```

AttributeConfig_2

```
struct AttributeConfig_2
{
  string name;
  AttrWriteType writable;
  AttrDataFormat data_format;
  long data_type;
  long max_dim_x;
  long max_dim_y;
  string description;
  string label;
  string unit;
  string standard_unit;
  string display_unit;
  string format;
  string min_value;
  string max_value;
  string min_alarm;
  string max_alarm;
  string writable_attr_name;
  DispLevel level;
  DevVarStringArray extensions;
};
```

AttributeConfig_3

```
struct AttributeConfig_3
{
  string name;
  AttrWriteType writable;
  AttrDataFormat data_format;
  long data_type;
  long max_dim_x;
  long max_dim_y;
  string description;
  string label;
  string unit;
  string standard_unit;
  string display_unit;
  string format;
  string min_value;
  string max_value;
  string writable_attr_name;
  DispLevel level;
  AttributeAlarm alarm;
  EventProperties event_prop;
  DevVarStringArray extensions;
  DevVarStringArray sys_extensions;
};
```

AttributeConfig_5

```
struct AttributeConfig_5
{
  string name;
  AttrWriteType writable;
  AttrDataFormat data_format;
  long data_type;
  boolean memorized;
  boolean mem_init;
  long max_dim_x;
  long max_dim_y;
  string description;
  string label;
```

```
  string unit;
  string standard_unit;
  string display_unit;
  string format;
  string min_value;
  string max_value;
  string writable_attr_name;
  DispLevel level;
  string root_attr_name;
  DevVarStringArray enum_labels;
  AttributeAlarm att_alarm;
  EventProperties event_prop;
  DevVarStringArray extensions;
  DevVarStringArray sys_extensions;
};
```

AttributeDim

```
struct AttributeDim
{
  long dim_x;
  long dim_y;

};
```

AttributeValue

```
struct AttributeValue
{
  any value;
  AttrQuality quality;
  TimeVal time;
  string name;
  long dim_x;
  long dim_y;

};
```

AttributeValue_3

```
struct AttributeValue_3
{
  any value;
  AttrQuality quality;
  TimeVal time;
  string name;
  AttributeDim r_dim;
  AttributeDim w_dim;
  DevErrorList err_list;
};
```

AttributeValue_4

```
struct AttributeValue_4
{
  AttrValUnion value;
  AttrQuality quality;
  AttrDataFormat data_format;
  TimeVal time;
  string name;
  AttributeDim r_dim;
  AttributeDim w_dim;
  DevErrorList err_list;
};
```

AttributeValue_5

```
struct AttributeValue_5
{
  AttrValUnion value;
  AttrQuality quality;
  AttrDataFormat data_format;
  long data_type;
  TimeVal time;
  string name;
  AttributeDim r_dim;
  AttributeDim w_dim;
  DevErrorList err_list;
};
```

ChangeEventProp

```
struct ChangeEventProp
{
  string rel_change;
  string abs_change;
  DevVarStringArray extensions;
};
```

DevAttrHistory

```
struct DevAttrHistory
{
  boolean attr_failed;
  AttributeValue value;
  DevErrorList errors;

};
```

DevAttrHistory_3

```
struct DevAttrHistory_3
{
  boolean attr_failed;
  AttributeValue_3 value;
};
```

DevAttrHistory_4

```
struct DevAttrHistory_4
{
  string name;
  TimeValList dates;
  any value;
  AttrQualityList quals;
  EltInArrayList quals_array;
  AttributeDimList r_dims;
  EltInArrayList r_dims_array;
  AttributeDimList w_dims;
  EltInArrayList w_dims_array;
  DevErrorListList errors;
  EltInArrayList errors_array;
```

```
};
```

DevAttrHistory_5

```
struct DevAttrHistory_5
{
  string name;
  AttrDataFormat data_format;
  long data_type;
  TimeValList dates;
  any value;
  AttrQualityList quals;
  EltInArrayList quals_array;
  AttributeDimList r_dims;
  EltInArrayList r_dims_array;
  AttributeDimList w_dims;
  EltInArrayList w_dims_array;
  DevErrorListList errors;
  EltInArrayList errors_array;

};
```

DevCmdHistory

```
struct DevCmdHistory
{
  TimeVal time;
  boolean cmd_failed;
  any value;
  DevErrorList errors;

};
```

DevCmdHistory_4

```
struct DevCmdHistory_4
{
  TimeValList dates;
  any value;
  AttributeDimList dims;
```

```
  EltInArrayList dims_array;
  DevErrorListList errors;
  EltInArrayList errors_array;
  long cmd_type;

};
```

DevCmdInfo

```
struct DevCmdInfo
{
  string cmd_name;
  long cmd_tag;
  long in_type;
  long out_type;
  string in_type_desc;
  string out_type_desc;

};
```

DevCmdInfo_2

```
struct DevCmdInfo_2
{
  string cmd_name;
  DispLevel level;
  long cmd_tag;
  long in_type;
  long out_type;
  string in_type_desc;
  string out_type_desc;
};
```

DevEncoded

```
struct DevEncoded
{
  DevString encoded_format;
  DevVarCharArray encoded_data;
};
```

DevError

```
struct DevError
{
  string reason;
  ErrSeverity severity;
  string desc;
  string origin;
};
```

DevInfo

```
struct DevInfo
{
  string dev_class;
  string server_id;
  string server_host;
  long server_version;
  string doc_url;
};
```

DevInfo_3

```
struct DevInfo_3
{
  string dev_class;
  string server_id;
  string server_host;
  long server_version;
  string doc_url;
  string dev_type;
};
```

DevIntrChange

```
struct DevIntrChange
{
  boolean dev_started;
  DevCmdInfoList_2 cmds;
  AttributeConfigList_5 atts;
};
```

DevPipeBlob

```
struct DevPipeBlob
```

```
{
  string name;
  DevVarPipeDataEltArray blob_data;

};
```

DevPipeData

```
struct DevPipeData
{
  string name;
  TimeVal time;
  DevPipeBlob data_blob;
};
```

DevPipeDataElt

```
struct DevPipeDataElt
{
  string name;
  AttrValUnion value;
  DevVarPipeDataEltArray inner_blob;
  string inner_blob_name;
};
```

DevVarDoubleStringArray

```
struct DevVarDoubleStringArray
{
  DevVarDoubleArray dvalue;
  DevVarStringArray svalue;
};
```

DevVarLongStringArray

```
struct DevVarLongStringArray
{
  DevVarLongArray lvalue;
  DevVarStringArray svalue;
};
```

EltInArray

```
struct EltInArray
{
```

```
  long start;
  long nb_elt;

};
```

EventProperties

```
struct EventProperties
{
  ChangeEventProp ch_event;
  PeriodicEventProp per_event;
  ArchiveEventProp arch_event;
};
```

JavaClntIdent

```
struct JavaClntIdent
{
  string MainClass;
  JavaUUID uuid;
};
```

NamedDevError

```
struct NamedDevError
{
  string name;
  long index_in_call;
  DevErrorList err_list;
};
```

PeriodicEventProp

```
struct PeriodicEventProp
{
  string period;
  DevVarStringArray extensions;
};
```

PipeConfig

```
struct PipeConfig
{
  string name;
  string description;
```

```
  string label;
  DispLevel level;
  PipeWriteType writable;
  DevVarStringArray extensions;
};
```

TimeVal

```
struct TimeVal
{
  long tv_sec;
  long tv_usec;
  long tv_nsec;

};
```

ZmqCallInfo

```
struct ZmqCallInfo
{
  long version;
  unsigned long ctr;
  string method_name;
  DevVarCharArray oid;
  boolean call_is_except;
};
```

Unions

AttrValUnion

```
union AttrValUnion switch (AttributeDataType)
{
case ATT_BOOL:
  DevVarBooleanArray bool_att_value;
case ATT_SHORT:
  DevVarShortArray short_att_value;
case ATT_LONG:
  DevVarLongArray long_att_value;
case ATT_LONG64:
  DevVarLong64Array long64_att_value;
```

```
case ATT_FLOAT:
  DevVarFloatArray float_att_value;
case ATT_DOUBLE:
  DevVarDoubleArray double_att_value;
case ATT_UCHAR
  DevVarCharArray uchar_att_value;
case ATT_USHORT:
  DevVarUShortArray ushort_att_value;
case ATT_ULONG:
  DevVarULongArray ulong_att_value;
case ATT_ULONG64:
  DevVarULong64Array ulong64_att_value;
case ATT_STRING:
  DevVarStringArray string_att_value;
case ATT_STATE:
  DevVarStateArray state_att_value;
case DEVICE_STATE:
  DevState dev_state_att;
case ATT_ENCODED:
  DevVarEncodedArray encoded_att_value;
case ATT_NO_DATA:
  DevBoolean union_no_data;
};
```

ClntIdent

```
union ClntIdent switch (LockerLanguage)
{
case CPP:
  CppClntIdent cpp_clnt;
case JAVA:
  JavaClntIdent java_clnt;
};
```

Exceptions

DevFailed

exception DevFailed

{

  DevErrorList errors;

};

MultiDevFailed

exception MultiDevFailed

{

  NamedDevErrorList errors;

};

Interface Tango::Device

The fundamental interface for all TANGO objects. Each Device is a network object which can be accessed locally or via network. The network protocol on the wire will be IIOP. The Device interface implements all the basic functions needed for doing generic synchronous and asynchronous I/O on a device. A Device object has data and actions. Data are represented in the form of Attributes. Actions are represented in the form of Commands. The CORBA Device interface offers attributes and methods to access the attributes and commands. A client will either use these methods directly from C++ or Java or access them via wrapper classes implemented in a API. The Device interface describes only the remote network interface. Implementation features like threads, command security, priority etc. are dealt with in server side of the device server model.

Attributes

adm_name

readonly attribute string adm_name;

adm_name (readonly) - administrator device unique ascii identifier

description

readonly attribute string description;

description (readonly) - general description of device

name

readonly attribute string name;

name (readonly) - unique ascii identifier

state

readonly attribute DevState state;

state (readonly) - device state

status

readonly attribute string status;

status (readonly) - device state as ascii string

Operations

black_box

DevVarStringArray black_box(in long number)

raises(DevFailed);
read list of last N commands executed by clients
Parameters:

number – of commands to return

Returns:

list of command and clients

command_inout

any command_inout(in string command, in any argin)

raises(DevFailed);
execute a command on a device synchronously with no input parameter and one one output parameter
Parameters:

command – ascii string e.g. On

argin – command input parameter e.g. float

Returns:

command result.

command_list_query

DevCmdInfoList command_list_query()

raises(DevFailed);
query device to see what commands it supports
Returns:

list of commands and their types

command_query

DevCmdInfo command_query(in string command)

raises(DevFailed);
query device to see command argument
Parameters:

command – name

Returns:

command and its types

get_attribute_config

AttributeConfigList get_attribute_config(in DevVarStringArray names)

raises(DevFailed);
read the configuration for a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute configurations read

info

DevInfo info()

raises(DevFailed);
return general information about object e.g. class, type, ...
Returns:

device info

ping

void ping()

raises(DevFailed);
ping a device to see if it alive

read_attributes

AttributeValueList read_attributes(in DevVarStringArray names)

raises(DevFailed);
read a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute values read

set_attribute_config

void set_attribute_config(in AttributeConfigList new_conf)

raises(DevFailed);
set the configuration for a variable list of attributes from the device
Parameters:

new_conf – list of attribute configuration to be set

write_attributes

void write_attributes(in AttributeValueList values)

raises(DevFailed);
write a variable list of attributes to a device
Parameters:

values – list of attribute values to write

Interface Tango::Device_2

interface Device_2 inherits from Tango::Device The updated Tango device interface. It inherits from Tango::Device and therefore supports all attribute/operation defined in the Tango::Device interface. Two CORBA operations have been modified to support more parameters (command_inout_2 and read_attribute_2). Three CORBA operations now retrun a different data type (command_list_query_2, command_query_2 and get_attribute_config)

Operations

command_inout_2

any command_inout_2(in string command, in any argin, in DevSource source)

raises(DevFailed); execute a command on a device synchronously with no input parameter and one one output parameter

Parameters:

command – ascii string e.g. On

argin – command input parameter

source – data source

Returns:

command result.

command_inout_history_2

DevCmdHistoryList command_inout_history_2(in string command, in long n)

raises(DevFailed);
Get command result history from polling buffer. Obviously, the command must be polled.
Parameters:

command – ascii string e.g. On

n – record number

Returns:

list of command result (or exception parameters if the command failed).

command_list_query_2

DevCmdInfoList_2 command_list_query_2()

raises(DevFailed);
query device to see what commands it supports

Returns:

list of commands and their types

command_query_2

DevCmdInfo_2 command_query_2(in string command)

raises(DevFailed);
query device to see command argument
Parameters:

command – name

Returns:

command and its types

get_attribute_config_2

AttributeConfigList_2 get_attribute_config_2(in DevVarStringArray names)

raises(DevFailed);
read the configuration for a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute configurations read

read_attributes_2

AttributeValueList read_attributes_2(in DevVarStringArray names, in DevSource source)

raises(DevFailed)
read a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute values read

read_attribute_history_2

DevAttrHistoryList read_attributes_history_2(in string name, in long n)

raises(DevFailed)
Get attribute value history from polling buffer. Obviously, the attribute must be polled.
Parameters:

name – Attribute name to read history

n – Record number

Returns:

list of attribute value (or exception parameters if the attribute failed).

Interface Tango::Device_3

interface Device_3 inherits from Tango::Device_2
The updated Tango device interface for Tango release 5. It inherits from Tango::Device_2 and therefore supports all attribute/operation defined in the Tango::Device_2 interface. Six CORBA operations now return a different data type (read_attributes_3, write_attributes_3, read_attribute_history_3, info_3, get_attribute_config_3 and set_attribute_config_3)

Operations

read_attributes_3

AttributeValueList_3 read_attributes_3(in DevVarStringArray names, in DevSource source)

raises(DevFailed);
read a variable list of attributes from a device
Parameters:

name – list of attribute names to read

source – data source

Returns:

list of attribute values read

write_attributes_3

void write_attributes_3(in AttributeValueList values)

raises(DevFailed, MultiDevFailed);
write a variable list of attributes to a device
Parameters:

values – list of attribute values to write

read_attribute_history_3

DevAttrHistoryList_3 read_attributes_history_3(in string name, in long n)

raises(DevFailed)
Get attribute value history from polling buffer. Obviously, the attribute must be polled.
Parameters:

name – Attribute name to read history

n – Record number

Returns:

list of attribute value (or exception parameters if the attribute failed).

info_3

DevInfo_3 info()

raises(DevFailed);
return general information about object e.g. class, type, ...
Returns:

device info

get_attribute_config_3

AttributeConfigList_3 get_attribute_config_3(in DevVarStringArray names)

raises(DevFailed);
read the configuration for a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute configurations read

set_attribute_config_3

void set_attribute_config_3(in AttributeConfigList_3 new_conf)

raises(DevFailed);
set the configuration for a variable list of attributes from the device
Parameters:

new_conf – list of attribute configuration to be set

Interface Tango::Device_4

interface Device_4 inherits from Tango::Device_3
The updated Tango device interface for Tango release 7. It inherits from Tango::Device_3 and therefore supports all attribute/operation defined in the Tango::Device_3 interface.

Operations

read_attributes_4

AttributeValueList_4 read_attributes_4(in DevVarStringArray names, in DevSource source,in ClntIdent cl_ident)

raises(DevFailed);
read a variable list of attributes from a device
Parameters:

name – list of attribute names to read

source – data source

cl_ident – client identificator

Returns:

list of attribute values read

write_attributes_4

void write_attributes_3(in AttributeValueList_4 values, in ClniIdent cl_ident)

raises(DevFailed, MultiDevFailed);

write a variable list of attributes to a device
Parameters:

values – list of attribute values to write

cl_ident – client identificator

command_inout_4

any command_inout_4(in string command, in any argin, in DevSource source, In ClntIdent cl_ident)

raises(DevFailed);
Execute a command on a device synchronously with one input parameter and one one output parameter
Parameters:

command – ascii string e.g. On

argin – command input parameter

source – data source

cl_ident – client identificator

Returns:

command result

read_attribute_history_4

DevAttrHistory_4 read_attributes_history_4(in string name, in long n)

raises(DevFailed)
Get attribute value history from polling buffer. Obviously, the attribute must be polled.
Parameters:

name – Attribute name to read history

n – Record number

Returns:

Attribute value (or exception parameters if the attribute failed) coded in a structure.

command_inout_history_4

DevCmdHistory_4 command_inout_history_4(in string command, in long n)

raises(DevFailed);
Get command value history from polling buffer. Obviously, the command must be polled.
Parameters:

name – Command name to read history

n – Record number

Returns:

Command value (or exception paramteters) coded in a structure

write_read_attribute_4

AttributeValueList_4 write_read_attribute_4(in AttributeValueList_4 values, in ClntIdent cl_ident)

raises(DevFailed,MultiDevFailed);
Write then read a variable list of attributes from a device
Parameters:

values – list of attribute values to write

cl_ident – client identificator

Returns:

list of attribute values read

set_attribute_config_4

void set_attribute_config_4(in AttributeConfigList_3 new_conf, in ClntIdent cl_ident)

raises(DevFailed);
set the configuration for a variable list of attributes from the device
Parameters:

new_conf – list of attribute configuration to be set

cl_ident – client identificator
Interface Tango::Device_4

interface Device_4 inherits from Tango::Device_3
The updated Tango device interface for Tango release 7. It inherits from Tango::Device_3 and therefore supports all attribute/operation defined in the Tango::Device_3 interface.

Interface Tango::Device_5

interface Device_5 inherits from Tango::Device_4
The updated Tango device interface for Tango release 9. It inherits from Tango::Device_4 and therefore supports all attribute/operation defined in the Tango::Device_4 interface.

operations

get_attribute_config_5

AttributeConfigList_5 get_attribute_config_5(in DevVarStringArray names)

raises(DevFailed);
read the configuration for a variable list of attributes from a device
Parameters:

name – list of attribute names to read

Returns:

list of attribute configurations read

set_attribute_config_5

void set_attribute_config_5(in AttributeConfigList_5 new_conf, in ClntIdent cl_ident)

raises(DevFailed);
set the configuration for a variable list of attributes from the device
Parameters:

new_conf – list of attribute configuration to be set

cl_ident – client identificator

read_attributes_5

AttributeValueList_5 read_attributes_5(in DevVarStringArray names, in DevSource source,in ClntIdent cl_ident)

raises(DevFailed);
read a variable list of attributes from a device
Parameters:

name – list of attribute names to read

source – data source

cl_ident – client identificator

Returns:

list of attribute values read

write_read_attributes_5

AttributeValueList_5 write_read_attributes_5(in AttributeValueList_4 values, in DevVarStringArray r_names, in ClntIdent cl_ident)

raises(DevFailed,MultiDevFailed);
Write then read a variable list of attributes from a device
Parameters:

values – list of attribute values to write

r_names – list of attribute to read

cl_ident – client identificator

Returns:

list of attribute values read

read_attribute_history_5

DevAttrHistory_5 read_attributes_history_5(in string name, in long n)

raises(DevFailed)
Get attribute value history from polling buffer. Obviously, the attribute must be polled.
Parameters:

name – Attribute name to read history

n – Record number

Returns:

Attribute value (or exception parameters if the attribute failed) coded in a structure.

get_pipe_config_5

PipeConfigList get_pipe_config_5(in DevVarStringArray names)

raises(DevFailed);

read the configuration for a variable list of pipes from a device
Parameters:

name – list of pipe names to read

Returns:

list of pipe configurations

set_pipe_config_5

void set_pipe_config_5(in PipeConfigList new_conf, in ClntIdent cl_ident)

raises(DevFailed);
set the configuration for a variable list of pipes from the device
Parameters:

new_conf – list of pipe configuration to be set

cl_ident – client identificator

read_pipe_5

DevPipeData read_pipe_5(in string name, in ClntIdent cl_ident)

raises(DevFailed);
read a pipe from a device
Parameters:

name – pipe name to read

cl_ident – client identificator

Returns:

Pipe value

write_pipe_5

void write_pipe_5(in DevPipeData value, in ClniIdent cl_ident)

raises(DevFailed);
write a pipe to a device
Parameters:

value – new pipe value to write

cl_ident – client identificator

write_read_pipe_5

DevPipeData write_read_pipe_5(in DevPipeData value, in ClntIdent cl_ident)

raises(DevFailed);
Write then read a pipe from a device
Parameters:

value – New pipe value to write

cl_ident – client identificator

Returns:

pipe values read

6.7.7 Reference part

Intended audience: developers, ***Programming language:*** all

This chapter is only part of the TANGO device server reference guide. To get reference documentation about the C++ library classes, see [TangoRefMan] To get reference documentation about the Java classes, also see [TangoRefMan]

Device parameter

A black box, a device description field, a device state and status are associated with each TANGO device.

The device black box

The device black box is managed as a circular buffer. It is possible to tune the buffer depth via a device property. This property name is

device name->blackbox_depth

A default value is hard-coded to 50 if the property is not defined. This black box depth property is retrieved from the Tango property database during the device creation phase.

The device description field

There are two ways to intialise the device description field.

- At device creation time. Some constructors of the DeviceImpl class supports this field as parameter. If these constructor are not used, the device description field is set to a default value which is *A Tango device*.
- With a property. A description field defines with this method overrides a device description defined at construction time. The property name is

  device name->description

The device state and status

Some constructors of the DeviceImpl class allows the initialisation of device state and/or status or device creation time. If these fields are not defined, a default value is applied. The default state is Tango::UNKOWN, the default status is *Not Initialised.*

The device polling

Seven device properties allow the polling tunning. These properties are described in the following table

Property name	property rule	default value
poll_ring_depth	Polling buffer depth	10
cmd_poll_ring_depth	Cmd polling buffer depth	
attr_poll_ring_depth	Attr polling buffer depth	
poll_old_factor	Data too old factor	4
min_poll_period	Minimun polling period	
cmd_min_poll_period	Min. polling period for cmd	
attr_min_poll_period	Min. polling period for attr	

The rule of the poll_ring_depth property is obvious. It defines the polling ring depth for all the device polled command(s) and attribute(s). Nevertheless, when filling the polling buffer via the fill_cmd_polling_buffer() (or fill_attr_polling_buffer()) method, it could be helpfull to define specific polling ring depth for a command (or an attribute). This is the rule of the cmd_poll_ring_depth and attr_poll_ring_depth properties. For each polled object with specific polling depth (command or attribute), the syntax of this property is the object name followed by the ring depth (ie State,20,Status,15). If one of these properties is defined, for the specific command or attribute, it will overwrite the value set by the poll_ring_depth property. The poll_old_factor property allows the user to tune how long the data recorded in the polling buffer are valid. Each time some data are read from the polling buffer, a check is done between the date when the data were recorded in the polling buffer and the date when the user request these data. If the interval is greater than the object polling period multiply by the value of the poll_old_factor factory, an exception is returned to the caller. These two properties are defined at device level and therefore, it is not possible to tune this parameter for each polled object (command or attribute). The last 3 properties are dedicated to define a polling period minimum threshold. The property min_poll_period defines in (mS) a device minimum polling period. Property cmd_min_poll_period defines (in mS) a minimum polling period for a specific command. The syntax of this property is the command name followed by the minimum polling period (ie MyCmd,400). Property attr_min_poll_period defines (in mS) a minimum polling period for a specific attribute. The syntax of this property is the attribute name followed by the minimum polling period (ie MyAttr,600). These two properties has a higher priority than the min_poll_period property. By default these three properties are not defined mening that there is no minimun polling period.

Four other properties are used by the Tango core classes to manage the polling thread. These properties are :

- polled_cmd to memorize the name of the device polled command
- polled_attr to memorize the name of the device polled attribute
- non_auto_polled_cmd to memorize the name of the command which shoule not be polled automatically at the first request
- non_auto_polled_attr to memorize the name of the attribute which should not be polled automatically at the first request

You don't have to change these properties values by yourself. They are automatically created/modified/deleted by Tango core classes.

The device logging

The Tango Logging Service (TLS) uses device properties to control device logging at startup (static configuration). These properties are described in the following table

Property name	property rule	default value
logging_level	Initial device logging level	WARNING
logging_target	Initial device logging target	No default
logging_rft	Logging rolling file threshold	20 Mega bytes
logging_path	Logging file path	/tmp/tango-<logging name> or C:/tango-<logging name> (Windows)

- The logging_level property controls the initial logging level of a device. Its set of possible values is: OFF, FATAL, ERROR, WARNING, INFO or DEBUG. This property is overwritten by the verbose command line option (-v).
- The logging_target property is a multi-valued property containing the initial target list. Each entry must have the following format: target_type::target_name (where target_type is one of the supported target types and target_name, the name of the target). Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in [sec:Tango-log-consumer]). For a

file target, target_name is the name of the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target. The TLS does not report any error occurred while trying to setup the initial targets.

  - Logging_target property example :
    logging_target = [console, file, file::/home/me/mydevice.log, device::tmp/log/1
    In this case, the device will automatically logs to the standard output, to its default file (which is something like domain_family_member.log), to a file named mydevice.log and located in /home/me. Finally, the device logs are also sent to a log consumer device named tmp/log/1.

- The logging_rft property specifies the rolling file threshold (rft), of the device's file targets. This threshold is expressed in Kb. When the size of a log file reaches the so-called rolling-file-threshold (rft), it is backuped as *current_log_file_name* + *_1* and a new current_log_file_name is opened. Obviously, there is only one backup file at a time (i.e. any existing backup is destroyed before the current log file is backuped). The default threshold is 20 Mb, the minimum is 500 Kb and the maximum is 1000 Mb.
- The logging_path property overwrites the TANGO_LOG_PATH environment variable. This property can only be applied to a DServer class device and has no effect on other devices.

Device attribute

Attribute are configured with two kind of parameters: Parameters hard-coded in source code and modifiable parameters

Hard-coded device attribute parameters

Seven attribute parameters are defined at attribute creation time in the Tango class source code. Obviously, these parameters are not modifiable except with a new source code compilation. These parameters are

Parameter name	Parameter description
name	Attribute name
data_type	Attribute data type
data_format	Attribute data format
writable	Attribute read/write type
max_dim_x	Maximum X dimension
max_dim_y	Maximum Y dimension
writable_attr_name	Associated write attribute
level	Attribute display level
root_attr_name	Root attribute name

The Attribute data type

Thirteen data types are supported. These data types are

- Tango::DevBoolean
- Tango::DevShort
- Tango::DevLong
- Tango::DevLong64
- Tango::DevFloat
- Tango::DevDouble

- Tango::DevUChar
- Tango::DevUShort
- Tango::DevULong
- Tango::DevULong64
- Tango::DevString
- Tango::DevState
- Tango::DevEncoded

The attribute data format

Three data format are supported for attribute

Format	Description
Tango::SCALAR	The attribute value is a single number
Tango::SPECTRUM	The attribute value is a one dimension number
Tango::IMAGE	The attribute value is a two dimension number

The max_dim_x and max_dim_y parameters

These two parameters defined the maximum size for attributes of the SPECTRUM and IMAGE data format.

data format	max_dim_x	max_dim_y
Tango::SCALAR	1	0
Tango::SPECTRUM	User Defined	0
Tango::IMAGE	User Defined	User Defined

For attribute of the Tango::IMAGE data format, all the data are also returned in a one dimension array. The first array is value[0],[0], array element X is value[0],[X-1], array element X+1 is value[1][0] and so forth.

The attribute read/write type

Tango supports four kind of read/write attribute which are :

- Tango::READ for read only attribute
- Tango::WRITE for writable attribute
- Tango::READ_WRITE for attribute which can be read and write
- Tango::READ_WITH_WRITE for a readable attribute associated to a writable attribute (For a power supply device, the current really generated is not the wanted current. To handle this, two attributes are defined which are *generated_current* and *wanted_current*. The *wanted_current* is a Tango::WRITE attribute. When the *generated_current* attribute is read, it is very convenient to also get the *wanted_current* attribute. This is exactly what the Tango::READ_WITH_WRITE attribute is doing)

When read, attribute values are always returned within an array even for scalar attribute. The length of this array and the meaning of its elements is detailed in the following table for scalar attribute.

Name	Array length	Array[0]	Array[1]
Tango::READ	1	Read value	
Tango::WRITE	1	Last write value	
Tango::READ_WRITE	2	Read value	Last write value
Tango::READ_WITH_WRITE	2	Read value	Associated attribute last write value

When a spectrum or image attribute is read, it is possible to code the device class in order to send only some part of the attribute data (For instance only a Region Of Interest for an image) but never more than what is defined by the attribute configuration parameters max_dim_x and max_dim_y. The number of data sent is also transferred with the data and is named **dim_x** and **dim_y**. When a spectrum or image attribute is written, it is also possible to send only some of the attribute data but always less than max_dim_x for spectrum and max_dim_x * max_dim_y for image. The following table describe how data are returned for spectrum attribute. dim_x is the data size sent by the server when the attribute is read and dim_x_w is the data size used during the last attribute write call.

Name	Array length	Array[0->dim_x-1]	Array[dim_x-> dim_x + dim_x_w -1]	
Tango::READ	dim_x	Read values		
Tango::WRITE	dim_x_w	Last write values		
Tango::READ_WRITE	dim_x + dim_x_w	Read value	Last write values	
Tango::READ_WITH_WRITE	dim_x + dim_x_w	Read value	Associated attributelast write values	

The following table describe how data are returned for image attribute. dim_r is the data size sent by the server when the attribute is read (dim_x * dim_y) and dim_w is the data size used during the last attribute write call (dim_x_w * dim_y_w).

Name	Array length	Array[0->dim_r-1]	Array[dim_r->dim_r + dim_w -1]
Tango::READ	dim_r	Read values	
Tango::WRITE	dim_w	Last write values	
Tango::READ_WRITE	dim_r + dim_w	Read value	Last write values
Tango::READ_WITH_WRITE	dim_r + dim_w	Read value	Associated attributelast write values

Until a write operation has been performed, the last write value is initialized to *0* for scalar attribute of the numeriacal type, to *Not Initialised* for scalar string attribute and to *true* for scalar boolean attribute. For spectrum or image attribute, the last write value is initialized to an array of one element set to *0* for numerical type, to an array of one element set to *true* for boolean attribute and to an array of one element set to *Not initialized* for string attribute

The associated write attribute parameter

This parameter has a meaning only for attribute with a Tango::READ_WITH_WRITE read/write type. This is the name of the associated write attribute.

The attribute display level parameter

This parameter is only an help for graphical application. It is a C++ enumeration starting at 0. The code associated with each attribute display level is defined in the following table (Tango::DispLevel).

name	Value
Tango::OPERATOR	0
Tango::EXPERT	1

This parameter allows a graphical application to support two types of operation :

- An operator mode for day to day operation
- An expert mode when tuning is necessary

According to this parameter, a graphical application knows if the attribute is for the operator mode or for the expert mode.

The root attribute name parameter

In case the attribute is a forwarded one, this parameter is the name of the associated root attribute. In case of classical attribute, this string is set to Not specified.

Modifiable attribute parameters

Each attribute has a configuration set of 20 modifiable parameters. These can be grouped in three different purposes:

1. General purpose parameters
2. Alarm related parameters
3. Event related parameters

General purpose parameters

Eight attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
description	Attribute description
label	Attribute label
unit	Attribute unit
standard_unit	Conversion factor to MKSA unit
display_unit	The attribute unit in a printable form
format	How to print attribute value
min_value	Attribute min value
max_value	Attribute max value
enum_labels	Enumerated labels
memorized	Attribute memorization

The **description** parameter describes the attribute. The **label** parameter is used by graphical application to display a label when this attribute is used in a graphical application. The **unit** parameter is the attribute value unit. The **standard_unit** parameter is the conversion factor to get attribute value in MKSA units. Even if this parameter is a number, it is returned as a string by the device *get_attribute_config* call. The **display_unit** parameter is the string used by graphical application to display attribute unit to application user. The **enum_labels** parameter is defined only for attribute of the DEV_ENUM data type. This is a vector of strings with one string for each enumeration label. It is an ordered list.

The format attribute parameter

This parameter specifies how the attribute value should be printed. It is not valid for string attribute. This format is a string of C++ streams manipulators separated by the **;** character. The supported manipulators are :

- fixed
- scientific
- uppercase
- showpoint
- showpos
- setprecision()
- setw()

Their definition are the same than for C++ streams. An example of format parameter is

`scientific;uppercase;setprecision(3)`.

A class called Tango::AttrManip has been written to handle this format string. Once the attribute format string has been retrieved from the device, its value can be printed with

```
cout << Tango::AttrManip(format) << value << endl;
```

The min_value and max_value parameters

These two parameters have a meaning only for attribute of the Tango::WRITE read/write type and for numerical data types. Trying to set the value of an attribute to something less than or equal to the min_value parameter is an error. Trying to set the value of the attribute to something more or equal to the max_value parameter is also an error. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

These two parameters have no meaning for attribute with data type DevString, DevBoolean or DevState. An exception is thrown in case the user try to set them for attribute of these 3 data types.

The memorized attribute parameter

This parameter describes the attribute memorization. It is an enumeration with the following values:

- NOT_KNOWN : The device is too old to return this information.
- NONE : The attribute is not memorized
- MEMORIZED : The attribute is memorized
- MEMORIZED_WRITE_INIT : The attribute is memorized and the memorized value is applied at device initialization time.

The alarm related configuration parameters

Six alarm related attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
min_alarm	Attribute low level alarm
max_alarm	Attribute high level alarm
min_warning	Attribute low level warning
max_warning	Attribute high level warning
delta_t	delta time for RDS alarm (mS)
delta_val	delta value for RDS alarm (absolute)

These parameters have no meaning for attribute with data type DevString, DevBoolean or DevState. An exception is thrown in case the user try to set them for attribute of these 3 data types.

The min_alarm and max_alarm parameters

These two parameters have a meaning only for attribute of the Tango::READ, Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read, if its value is something less than or equal to the min_alarm parameter or if it is something more or equal to the max_alarm parameter, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

The min_warning and max_warning parameters

These two parameters have a meaning only for attribute of the Tango::READ, Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read, if its value is something less than or equal to the min_warning parameter or if it is something more or equal to the max_warning parameter, the attribute quality factor will be set to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

The delta_t and delta_val parameters

These two parameters have a meaning only for attribute of the Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. They specify if and how the RDS alarm is used. When the attribute is read, if the difference between its read value and the last written value is something more than or equal to the delta_val parameter and if at least delta_val milli seconds occurs since the last write operation, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

The event related configuration parameters

Six event related attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
rel_change	Relative change triggering change event
abs_change	Absolute change triggering change event
period	Period for periodic event
archive_rel_change	Relative change for archive event
archive_abs_change	Absolute change for archive event
archive_period	Period for change archive event

The rel_change and abs_change parameters

Rel_change is a property with a maximum of 2 values (comma separated). It specifies the increasing and decreasing relative change of the attribute value (w.r.t. the value of the previous change event) which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. It's the absolute value of these values which is taken into account. If only one value is specified then it is used for the increasing and decreasing change.

Abs_change is a property of maximum 2 values (comma separated). It specifies the increasing and decreasing absolute change of the attribute value (w.r.t the value of the previous change event) which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one value is specified then it is used for the increasing and decreasing change. If no values are specified then the relative change is used.

The periodic period parameter

The minimum time between events (in milliseconds). If no property is specified then a default value of 1 second is used.

The archive_rel_change, archive_abs_change and archive_period parameters

archive_rel_change is an array property of maximum 2 values which specifies the positive and negative relative change w.r.t. the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then a default fo +-10% is used

archive_abs_change is an array property of maximum 2 values which specifies the positive and negative absolute change w.r.t the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.

archive_period is the minimum time between archive events (in milliseconds). If no property is specified, no periodic archiving events are send.

Setting modifiable attribute parameters

A default value is given to all modifiable attribute parameters by the Tango core classes. Nevertheless, it is possible to modify these values in source code at attribute creation time or via the database. Values retrieved from the database have a higher priority than values given at attribute creation time. The attribute parameters are therefore initialized from:

1. The Database
2. If nothing in database, from the Tango class default
3. If nothing in database nor in Tango class default, from the library default value

The default value set by the Tango core library are

Parameter type	Parameter name	Library default value
general purpose	description	No description
	label	attribute name
	unit	One empty string
	standard_unit	No standard unit
	display_unit	No display unit
	format	6 characters with 2 decimal
	min_value	Not specified
	max_value	Not specified
alarm parameters	min_alarm	Not specified
	max_alarm	Not specified
	min_warning	Not specified
	max_warning	Not specified
	& delta_t	Not specified
	delta_val	Not specified
event parameters	rel_change	Not specified
	abs_change	Not specified
	period	1000 (mS)
	archive_rel_change	Not specified
	archive_abs_change	Not specified
	archive_period	Not specified

It is possible to set modifiable parameters via the database at two levels :

1. At class level
2. At device level. Each device attribute have all its modifiable parameters sets to the value defined at class level. If the setting defined at class level is not correct for one device, it is possible to re-define it.

If we take the example of a class called *BumperPowerSupply* with three devices called *sr/bump/1*, *sr/bump/2* and *sr/bump/3* and one attribute called *wanted_current*. For the first two bumpers, the max_value is equal to 500. For the third one, the max_value is only 400. If the max_value parameter is defined at class level with the value 500, all devices will have 500 as max_value for the *wanted_current* attribute. It is necessary to re-defined this parameter at device level in order to have the max_value for device sr/bump/3 set to 400.

For the description, label, unit, standard_unit, display_unit and format parameters, it is possible to return them to their default value by setting them to an empty string.

Resetting modifiable attribute parameters

It is possible to reset attribute parameters to their default value at any moment. This could be done via the network call available through the DeviceProxy::set_attribute_config() method family. This call takes attribute parameters as strings. The following table describes which string has to be used to reset attribute parameters to their default value. In this table, the user default are the values given within Pogo in the Properties tab of the attribute edition window (or in in Tango class code using the Tango::UserDefaultAttrProp class).

Input string	Action
'Not specified'	Reset to **library** default
''(empty string)	Reset to **user** default if any. Otherwise, reset to **library** default
'NaN'	Reset to Tango **class** default if any. Otherwise, reset to **user** default (if any) or to **library** default

Let's take one exemple: For one attribute belonging to a device, we have the following attribute parameters:

Parameter name	Def. class	Def. user	Def. lib
standard_unit			No standard unit
min_value		5	Not specified
max_value	50		Not specified
rel_change	5	10	Not specified

The string Not specified sent to each attribute parameter will set attribute parameter value to No standard unit for standard_unit, Not specified for min_value, Not specified for max_value and Not specified as well for rel_change. The empty string sent to each attribute parameter will result with No stanadard unit for standard_unit, 5 for min_value, Not specified for max_value and 10 for rel_change. The string NaN will give No standard unit for standard_unit, 5 for min_value, 50 for max_value and 5 for rel_change.

C++ specific: Instead of the string Not specified and NaN, the preprocessor define **AlrmValueNotSpec** and **NotANumber** can be used.

Device pipe

Pipe are configured with two kind of parameters: Parameters hard-coded in source code and modifiable parameters

Hard-coded device pipe parameters

Three pipe parameters are defined at pipe creation time in the Tango class source code. Obviously, these parameters are not modifiable except with a new source code compilation. These parameters are

Parameter name	Parameter description
name	Pipe name
writable	Pipe read/write type
disp_level	Pipe display level

The pipe read/write type.

Tango supports two kinds of read/write pipe which are :

- Tango::PIPE_READ for read only pipe
- Tango::PIPE_READ_WRITE for pipe which can be read and written

The pipe display level parameter

This parameter is only an help for graphical application. It is a C++ enumeration starting at 0. The code associated with each pipe display level is defined in the following table (Tango::DispLevel).

name	Value
Tango::OPERATOR	0
Tango::EXPERT	1

This parameter allows a graphical application to support two types of operation :

- An operator mode for day to day operation
- An expert mode when tuning is necessary

According to this parameter, a graphical application knows if the pipe is for the operator mode or for the expert mode.

Modifiable pipe parameters

Each pipe has a configuration set of 2 modifiable parameters. These parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
description	Pipe description
label	Pipe label

The **description** parameter describes the pipe. The **label** parameter is used by graphical application to display a label when this pipe is used in a graphical application.

Setting modifiable pipe parameters

A default value is given to all modifiable pipe parameters by the Tango core classes. Nevertheless, it is possible to modify these values in source code at pipe creation time or via the database. Values retrieved from the database have a higher priority than values given at pipe creation time. The pipe parameters are therefore initialized from:

1. The Database
2. If nothing in database, from the Tango class default
3. If nothing in database nor in Tango class default, from the library default value

The default value set by the Tango core library are

Parameter name	Library default value
description	No description
label	pipe name

It is possible to set modifiable parameters via the database at two levels :

1. At class level
2. At device level. Each device pipe have all its modifiable parameters sets to the value defined at class level. If the setting defined at class level is not correct for one device, it is possible to re-define it.

This is the same principle than the one used for attribute configuration modifiable parameters.

Resetting modifiable pipe parameters

It is possible to reset pipe parameters to their default value at any moment. This could be done via the network call available through the DeviceProxy::set_pipe_config() method family. It uses the same principle than the one used for resetting modifiable attribute pipe parameters. Refer to their documentation if you want to know details about this feature.

Device class parameter

A device documentation field is also defined at Tango device class level. It is defined as Tango device class level because each device belonging to a Tango device class should have the same behaviour and therefore the same documentation. This field is store in the DeviceClass class. It is possible to set this field via a class property. This property name is

class name->doc_url

and is retrieved when instance of the DeviceClass object is created. A default value is defined for this field.

The device black box

This black box is a help tool to ease debugging session for a running device server. The TANGO core software records every device request in this black box. A tango client is able to retrieve the black box contents with a specific CORBA operation availabble for every device. Each black box entry is returned as a string with the following information :

- The date where the request has been executed by the device. The date format is dd/mm/yyyy hh24:mi:ss:SS (The last field is the second hundredth number).
- The type of CORBA requests. In case of attributes, the name of the requested attribute is returned. In case of operation, the operation type is returned. For “command_inout” operation, the command name is returned.
- The client host name

Automatically added commands

As already mentionned in this documentation, each Tango device supports at least three commands which are State, Status and Init. The following array details command input and output data type

Command name	Input data type	Output data type
State	void	Tango::DevState
Status	void	Tango::DevString
Init	void	void

The State command

This command gets the device state (stored in its *device_state* data member) and returns it to the caller. The device state is a variable of the Tango_DevState type (packed into a CORBA Any object when it is returned by a command)

The Status command

This command gets the device status (stored in its *device_status* data member) and returns it to the caller. The device status is a variable of the string type.

The Init command

This commands re-initialise a device keeping the same network connection. After an Init command executed on a device, it is not necessary for client to re-connect to the device. This command first calls the device *delete_device()* method and then execute its *init_device()* method. For C++ device server, all the memory allocated in the *init_device()* method must be freed in the *delete_device()* method. The language device desctructor automatically calls the *delete_device()* method.

DServer class device commands

As already explained in [DServer_class], each device server process has its own Tango device. This device supports the three commands previously described plus 32 commands which are DevRestart, RestartServer, QueryClass, QueryDevice, Kill, QueryWizardClassProperty, QueryWizardDevProperty, QuerySubDevice, the polling related commands which are StartPolling, StopPolling, AddObjPolling, RemObjPolling, UpdObjPollingPeriod, PolledDevice and DevPollStatus, the device locking related commands which are LockDevice, UnLockDevice, ReLockDevices and DevLockStatus, the event related commands called EventSubscriptionChange, ZmqEventSubscriptionChange and EventConfirmSubscription and finally the logging related commands which are AddLoggingTarget, RemoveLoggingTarget, GetLoggingTarget, GetLoggingLevel, SetLoggingLevel, StopLogging and StartLogging. The following table give all commands input and output data types

Command name	Input data type	Output data type
State	void	Tango::DevState
Status	void	Tango::DevString
Init	void	void
DevRestart	Tango::DevString	void
RestartServer	void	void
QueryClass	void	Tango::DevVarStringArray
QueryDevice	void	Tango::DevVarStringArray
Kill	void	void
QueryWizardClassProperty	Tango::DevString	Tango::DevVarStringArray
QueryWizardDevProperty	Tango::DevString	Tango::DevVarStringArray
QuerySubDevice	void	Tango::DevVarStringArray
StartPolling	void	void
StopPolling	void	void
AddObjPolling	Tango::DevVarLongStringArray	void
RemObjPolling	Tango::DevVarStringArray	void
UpdObjPollingPeriod	Tango::DevVarLongStringArray	void
PolledDevice	void	Tango::DevVarStringArray
DevPollStatus	Tango::DevString	Tango::DevVarStringArray
LockDevice	Tango::DevVarLongStringArray	void
UnLockDevice	Tango::DevVarLongStringArray	Tango::DevLong
ReLockDevices	Tango::DevVarStringArray	void
DevLockStatus	Tango::DevString	Tango::DevVarLongStringArray
EventSubscribeChange	Tango::DevVarStringArray	Tango::DevLong
ZmqEventSubscriptionChange	Tango::DevVarStringArray	Tango::DevVarLongStringArray

continues on next page

Table 2 – continued from previous page

Command name	Input data type	Output data type
EventConfirmSubscription	Tango::DevVarStringArray	void
AddLoggingTarget	Tango::DevVarStringArray	void
RemoveLoggingTarget	Tango::DevVarStringArray	void
GetLoggingTarget	Tango::DevString	Tango::DevVarStringArray
GetLoggingLevel	Tango::DevVarStringArray	Tango::DevVarLongStringArray
SetLoggingLevel	Tango::DevVarLongStringArray	void
StopLogging	void	void
StartLogging	void	void

The device description field is set to "A device server device". Device server started with the -file command line option also supports a command called QueryEventChannelIOR. This command is used interanally by the Tango kernel classes when the event system is used with device server using database on file.

The State command

This device state is always set to ON

The Status command

This device status is always set to "The device is ON" followed by a new line character and a string describing polling thread status. This string is either "The polling is OFF" or "The polling is ON" according to polling state.

The DevRestart command

The DevRestart command restart a device. The name of the device to be re-started is the command input parameter. The command destroys the device by calling its destructor and re-create it from its constructor.

The RestartServer command

The DevRestartServer command restarts all the device pattern(s) embedded in the device server process. Therefore, all the devices implemented in the server process are destroyed and re-built[1]. The network connection between client(s) and device(s) implemented in the device server process is destroyed and re-built.

Executing this command allows a complete restart of the device server without stopping the process.

The QueryClass command

This command returns to the client the list of Tango device class(es) embedded in the device server. It returns only class(es) implemented by the device server programmer. The DServer device class name (implemented by the TANGO core software) is not returned by this command.

[1] Their black-box is also destroyed and re-built

The QueryDevice command

This command returns to the client the list of device name for all the device(s) implemented in the device server process. Each device name is returned using the following syntax :

<class name>::<device name>

The name of the DServer class device is not returned by this command.

The Kill command

This command stops the device server process. In order that the client receives a last answer from the server, this command starts a thread which will after a short delay, kills the device server process.

The QueryWizardClassProperty command

This command returns the list of property(ies) defined for a class stored in the device server process property wizard. For each property, its name, a description and a default value is returned.

The QueryWizardDevProperty command

This command returns the list of property(ies) defined for a device stored in the device server process property wizard. For each property, its name, a description and a default value is returned.

The QuerySubDevice command

This command returns the list of sub-device(s) imported by each device within the server. A sub-device is a device used (to execute command(s) and/or to read/write attribute(s)) by one of the device server process devices. There is one element in the returned strings array for each sub-device. The syntax of each string is the device name, a space and the sub-device name. In case of device server process starting threads using a sub-device, it is not possible to link this sub-device to any process devices. In such a case, the string contains only the sub-device name

The StartPolling command

This command starts the polling thread

The StopPolling command

This command stops the polling thread

The AddObjPolling command

This command adds a new object in the list of object(s) to be polled. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and three strings. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
svalue[1]	Object type (“command“ or “attribute“)
svalue[2]	Object name
lvalue[0]	polling period in mS

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command does not start polling if it is stopped. This command is not allowed in case the device is locked and the command requester is not the lock owner.

The RemObjPolling command

This command removes an object of the list of polled objects. The command input data type is a Tango::DevVarStringArray with three strings. These strings meaning are :

String	Meaning
string[0]	Device name
string[1]	Object type (“command“ or “attribute“)
string[2]	Object name

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command is not allowed in case the device is locked and the command requester is not the lock owner.

The UpdObjPollingPeriod command

This command changes the polling period for a specified object. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and three strings. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
svalue[1]	Object type (“command“ or “attribute“)
svalue[2]	Object name
lvalue[0]	new polling period in mS

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command does not start polling if it is stopped. This command is not allowed in case the device is locked and the command requester is not the lock owner.

The PolledDevice command

This command returns the name of device which are polled. Each string in the Tango::DevVarStringArray returned by the command is a device name which has at least one command or attribute polled. The list is alphabetically sorted.

The DevPollStatus command

This command returns a polling status for a specific device. The input parameter is a device name. Each string in the Tango::DevVarStringArray returned by the command is the polling status for each polled device objects (command or attribute). For each polled objects, the polling status is :

- The object name
- The object polling period (in mS)
- The object polling ring buffer depth
- The time needed (in mS) for the last command execution or attribute reading
- The time since data in the ring buffer has not been updated. This allows a check of the polling thread
- The delta time between the last records in the ring buffer. This allows checking that the polling period is respected by the polling thread.
- The exception parameters in case of the last command execution or the last attribute reading failed.

A new line character is inserted between each piece of information.

The LockDevice command

This command locks a device for the calling process. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and one string. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
lvalue[0]	Lock validity

The UnLockDevice command

This command unlocks a device. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and one string. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
lvalue[0]	Force flag

The force flag parameter allows a client to unlock a device already locked by another process (for admin usage only)

The ReLockDevices command

This command re-lock devices. The input argument is the list of devices to be re-locked. It's an error to re-lock a device which is not already locked.

The DevLockStatus command

This command returns a device locking status to the caller. Its input parameter is the device name. The output parameters are embedded within a Tango::DevVarLongStringArray data type with three strings and six long. These data are

Command parameter	Parameter meaning
svalue[0]	Locking string
svalue[1]	CPP client host IP address or Not defined
svalue[2]	Java VM main class for Java client or Not defined
lvalue[0]	Lock flag (1 if locked, 0 othterwise)
lvalue[1]	CPP client host IP address or 0 for Java locker
lvalue[2]	Java locker UUID part 1or 0 for CPP locker
lvalue[3]	Java locker UUID part 2 or 0 for CPP locker
lvalue[4]	Java locker UUID part 3 or 0 for CPP locker
lvalue[5]	Java locker UUID part 4 or 0 for CPP locker

The EventSubscriptionChange command (C++ server only)

This command is used as a piece of the heartbeat system between an event client and the device server generating the event. There is no reason to generate events if there is no client which has subscribed to it. It is used by the *DeviceProxy::subscribe_event()* method and one of the event thread on the client side to inform the server to keep on generating events for the attribute in question. It reloads the subscription timer with the current time. Events are not generated when there are no clients subscribed within the last 10 minutes. The input parameters are:

Command parameter	Parameter meaning
argin[0]	Device name
argin[1]	Attribute name
argin[2]	action (subscribe or unsubsribe)
argin[3]	event name (change, periodic, archive,attr_conf)

The command output data is the simply the Tango release used by the device server process. This is necessary for compatibility reason.

The ZmqEventSubscriptionChange command

This command is used as a piece of the heartbeat system between an event client and the device server generating the event when client and/or device server uses Tango release 8 or above. There is no reason to generate events if there is no client which has subscribed to it. It is used by the *DeviceProxy::subscribe_event()* method and one of the event thread on the client side to inform the server to keep on generating events for the attribute in question. It reloads the subscription timer with the current time. Events are not generated when there are no clients subscribed within the last 10 minutes. The input parameters are the same than the one used for the EventSubscriptionChange command. They are:

Command in parameter	Parameter meaning
argin[0]	Device name
argin[1]	Attribute/pipe name
argin[2]	action (subscribe or unsubsribe)
argin[3]	event name (change, quality, periodic, archive, user_event, attr_conf, data_ready, intr_change, pipe)
argin[4]	<Tango client IDL version>

The command output parameters aer all the necessary data to build one event connection between a client and the device server process generating the events. This means:

Command out parameter	Parameter meaning
svalue[0]	Heartbeat ZMQ socket connect end point
svalue[1]	Event ZMQ socket connect end point
svalue[2]	<Alternate Heartbeat pub endpoint>
svalue[3]	<Alternate Event pub endpoint>
...	
svalue[n-3]	<Alternate Heartbeat pub endpoint>
svalue[n-2]	<Alternate Event pub endpoint>
svalue[n-1]	event name used by this server as zmq topic to send events
svalue[n]	channel name used by this server to send heartbeat events
lvalue[0]	Tango lib release used by device server
lvalue[1]	Device IDL release
lvalue[2]	Subscriber HWM
lvalue[3]	Rate (Multicasting related)
lvalue[4]	IVL (Multicasting related)
lvalue[5]	ZMQ release

The EventConfirmSubscription command

This command is used by client to regularly notify to device server process their interest in receiving events. If this command is not received, after a delay of 600 sec (10 mins), event(s) will not be sent any more. The input parameters for the EventConfirmSubscription command must be a multiple of 3. They are 3 parameters for each event confirmed by this command. Per event, these parameters are:

Command in parameter	Parameter meaning
argin[x]	Device name
argin[x + 1]	Attribute name
argin[x + 2]	Event name

The AddLoggingTarget command

This command adds one (or more) logging target(s) to the specified device(s). The command input parameter is an array of string logically composed of {device_name, target_type::target_name} groups where the elements have the following semantic:

- device_name is the name of the device which logging behavior is to be controlled. The wildcard is supported to apply the modification to all devices encapsulated within the device server (e.g. to ask all devices to log to the same device target).
- target_type::target_name: target_type is one of the supported target types and target_name, the name of the target. Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in [sec:Tango-log-consumer]). For a file target, target_name is the full path to the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target and can be omitted.

This command is not allowed in case the device is locked and the command requester is not the lock owner.

The RemoveLoggingTarget command

Remove one (or more) logging target(s) from the specified device(s).The command input parameter is an array of string logically composed of {device_name, target_type::target_name} groups where the elements have the following semantic:

- device_name: the name of the device which logging behavior is to be controlled. The wildcard is supported to apply the modification to all devices encapsulated within the device server (e.g. to ask all devices to stop logging to a given device target).
- target_type::target_name: target_type is one of the supported target types and target_name, the name of the target. Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in [sec:Tango-log-consumer]). For a file target, target_name is the full path to the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target and can be omitted.

The wildcard is supported for target_name. For instance, RemoveLoggingTarget ([, device::*) removes all the device targets from all the devices running in the device server. This command is not allowed in case the device is locked and the command requester is not the lock owner.

The GetLoggingTarget command

Returns the current target list of the specified device. The command parameter device_name is the name of the device which logging target list is requested. The list is returned as a DevVarStringArray containing target_type::target_name elements.

The GetLoggingLevel command

Returns the logging level of the specified devices. The command input parameter device_list contains the names of the devices which logging target list is requested. The wildcard is supported to get the logging level of all the devices running within the server. The string part of the result contains the name of the devices and its long part contains the levels. Obviously, result.lvalue[i] is the current logging level of the device named result.svalue[i].

The SetLoggingLevel command

Changes the logging level of the specified devices. The string part of the command input parameter contains the device names while its long part contains the logging levels. The set of possible values for levels is: 0=OFF, 1=FATAL, 2=ERROR, 3=WARNING, 4=INFO, 5=DEBUG.

The wildcard is supported to assign all devices the same logging level. For instance, SetLoggingLevel ([3]) set the logging level of all the devices running within the server to WARNING. This command is not allowed in case the device is locked and the command requester is not the lock owner.

The StopLogging command

For all the devices running within the server, StopLogging saves their current logging level and set their logging level to OFF.

The StartLogging command

For each device running within the server, StartLogging restores their logging level to the value stored during a previous StopLogging call.

DServer class device properties

This device has two properties related to polling threads pool management plus another one for the choice of polling algorithm. These properties are described in the following table

Property name	property rule	default value
polling_threads_pool_size	Max number of thread in the polling pool	1
polling_threads_pool_conf	Polling threads pool configuration	
polling_before_9	Choice of the polling algorithm	false

The rule of the polling_threads_pool_size is to define the maximun number of thread created for the polling threads pool size. The rule of the polling_threads_pool_conf is to define which thread in the pool is in charge of all the polled object(s) of which device. This property is an array of strings with one string per used thread in the pool. The content of the string is simply a device name list with device name splitted by a comma. Example of polling_threads_pool_conf property for 3 threads used:

```
dserver/<ds exec name>/<inst. name>/polling_threads_pool_conf-> the/dev/01
                    the/dev/02,the/dev/06
                    the/dev/03
```

Thread number 2 is in charge of 2 devices. Note that there is an entry in this list only for the used threads in the pool.

The rule of the polling_before_9 property is to select the polling algorithm which was used in Tango device server process before Tango release 9.

Tango log consumer

The available Log Consumer

One implementation of a log consumer associated to a graphical user interface is available within Tango. It is a standalone java application called **LogViewer** based on the publicly available chainsaw application from the log4j package. It supports two way of running which are:

- The static mode: In this mode, LogViewer is started with a parameter which is the name of the log consumer device implemented by the application. All messages sent by devices with a logging target type set to *device* and with a logging target name set to the same device name than the device name passed as application parameter will be displayed (if the logging level allows it).
- The dynamic mode: In this mode, the name of the log consumer device implemented by the application is build at application startup and is dynamic. The user with the help of the graphical interface chooses device(s) for which he want to see log messages.

The Log Consumer interface

A Tango Log Consumer device is nothing but a tango device supporting the following tango command :

void log (Tango::DevVarStringArray details)

where details is an array of string carrying the log details. Its structure is:

- details[0] : the timestamp in millisecond since epoch (01.01.1970)
- details[1] : the log level
- details[2] : the log source (i.e. device name)
- details[3] : the log message
- details[4] : the log NDC (contextual info) - Not used but reserved
- details[5] : the thread identifier (i.e. the thread from which the log request comes from)

These log details can easily be extended. Any tango device supporting this command can act as a device target for other devices.

Control system specific

It is possible to define a few control system parameters. By control system, we mean for each set of computers having the same database device server (the same TANGO_HOST environment variable)

The device class documentation default value

Each control system may have it's own default device class documentation value. This is defined via a class property. The property name is

Default->doc_url

It's retrieved if the device class itself does not define any doc_url property. If the Default->doc_url property is also not defined, a hard-coded default value is provided.

The services definition

The property used to defined control system services is named **Services** and belongs to the free object **CtrlSystem**. This property is an array of strings. Each string defines a service available within the control system. The syntax of each service definition is

Service name/Instance name:service device name

Tuning the event system buffers (HWM)

Starting with Tango release 8, ZMQ is used for the event based communication between clients and device server processes. ZMQ implementation provides asynchronous communication in the sense that the data to be transmitted is first stored in a buffer and then really sent on the network by dedicated threads. The size of this buffers (on client and device server side) is called High Water Mark (HWM) and is tunable. This is tunable at several level.

1. The library set a default value of **1000** for both buffers (client and device server side)
2. Control system properties used to tune these size are named **DSEventBufferHwm** (device server side) and **EventBufferHwm** (client side). They both belongs to the free object **CtrlSystem**. Each property is the max number of events storable in these buffer.
3. At client or device server level using the library calls *Util::set_ds_event_buffer_hwm()* documented in [TangoRefMan] or *ApiUtil::set_event_buffer_hwm()* documented in `Tango::ApiUtil`
4. Using environment variables TANGO_DS_EVENT_BUFFER_HWM or TANGO_EVENT_BUFFER_HWM

Allowing NaN when writing attributes (floating point)

A property named **WAttrNaNAllowed** belonging to the free object **CtrlSystem** allows a Tango control system administrator to allow or disallow NaN numbers when writing attributes of the DevFloat or DevDouble data type. This is a boolean property and by default, it's value is taken as false (Meaning NaN values are rejected).

Tuning multicasting event propagation

Starting with Tango 8.1, it is possible to transfer event(s) between devices and clients using a multicast protocol. The properties **MulticastEvent**, **MulticastRate**, **MulticastIvl** and **MulticastHops** also belonging to the free object **CtrlSystem** allow the user to configure which events has to be sent using multicasting and with which parameters. See chapter Advanced features/Using multicast protocol to transfer events to get details about these properties.

Summary of CtrlSystem free object properties

The following table summarizes properties defined at control system level and belonging to the free object CtrlSystem

Property name	property rule	default value
Services	List of defined services	No default
DsEventBufferHwm	DS event buffer high water mark	1000
EventBufferHwm	Client event buffer high water mark	1000
WAttrNaNAllowed	Allow NaN when writing attr.	false
MulticastEvent	List of multicasting events	No default
MulticastRate	Rate for multicast event transport	80
MulticastIvl	Time to keep data for re-transmission	20
MulticastHops	Max number of elements to cross	5

C++ specific

The Tango master include file (tango.h)

Tango has a master include file called

tango.h

This master include file includes the following files :

- Tango configuration include file : **tango_config.h**
- CORBA include file : **idl/tango.h**
- Some network include files for WIN32 : **winsock2.h** and **mswsock.h**
- C++ streams include file :
  - **iostream**, **sstream** and **fstream**
- Some standard C++ library include files : **memory, string** and **vector**
- A long list of other Tango include files

Tango specific pre-processor define

The tango.h previously described also defined some pre-processor macros allowing Tango release to be checked at compile time. These macros are:

- TANGO_VERSION_MAJOR
- TANGO_VERSION_MINOR
- TANGO_VERSION_PATCH

For instance, with Tango release 8.1.2, TANGO_VERSION_MAJOR will be set to 8 while TANGO_VERSION_MINOR will be 1 and TANGO_VERSION_PATCH will be 2.

Tango specific types

Operating system free type

Some data type used in the TANGO core software have been defined. They are described in the following table.

Type name	C++ name
TangoSys_MemStream	stringstream
TangoSys_OMemStream	ostringstream
TangoSys_Pid	int
TangoSys_Cout	ostream

These types are defined in the tango_config.h file

Template command model related type

As explained in [Command fact], command created with the template command model uses static casting. Many type definition have been written for these casting.

Class name	Command allowed method (if any)	Command execute method
TemplCommand	Tango::StateMethodPtr	Tango::CmdMethPtr
TemplCommandIn	Tango::StateMethodPtr	Tango::CmdMethPtr_xxx
TemplCommandOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr
TemplCommandInOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr_yyy

The **Tango::StateMethPtr** is a pointer to a method of the DeviceImpl class which returns a boolean and has one parameter which is a reference to a const CORBA::Any obect.

The **Tango::CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns nothing and needs nothing as parameter.

The **Tango::CmdMethPtr_xxx** is a pointer to a method of the DeviceImpl class which returns nothing and has one parameter. xxx must be set according to the method parameter type as described in the next table

Tango type	short cut (xxx)
Tango::DevBoolean	Bo
Tango::DevShort	Sh
Tango::DevLong	Lg
Tango::DevFloat	Fl
Tango::DevDouble	Db
Tango::DevUshort	US
Tango::DevULong	UL
Tango::DevString	Str
Tango::DevVarCharArray	ChA
Tango::DevVarShortArray	ShA
Tango::DevVarLongArray	LgA
Tango::DevVarFloatArray	FlA
Tango::DevVarDoubleArray	DbA
Tango::DevVarUShortArray	USA
Tango::DevVarULongArray	ULA
Tango::DevVarStringArray	StrA
Tango::DevVarLongStringArray	LSA
Tango::DevVarDoubleStringArray	DSA
Tango::DevState	Sta

For instance, a pointer to a method which takes a Tango::DevVarStringArray as input parameter must be statically casted to a Tango::CmdMethPtr_StrA, a pointer to a method which takes a Tango::DevLong data as input parameter must be statically casted to a Tango::CmdMethPtr_Lg.

The **Tango::xxx_CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has no input parameter. xxx must be set according to the method return data type following the same rules than those described in the previous table. For instance, a pointer to a method which returns a Tango::DevDouble data must be statically casted to a Tango::Db_CmdMethPtr.

The **Tango::xxx_CmdMethPtr_yyy** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has one input parameter of one of the Tango data type. xxx and yyy must be set according to the method return data type and parameter type following the same rules than those described in the previous table. For instance, a

pointer to a method which returns a Tango::DevDouble data and which takes a Tango::DevVarLongStringArray must be statically casted to a Tango::Db_CmdMethPtr_LSA.

All those type are defined in the tango_const.h file.

Tango device state code

The Tango::DevState type is a C++ enumeration starting at 0. The code associated with each state is defined in the following table.

State name	Value
Tango::ON	0
Tango::OFF	1
Tango::CLOSE	2
Tango::OPEN	3
Tango::INSERT	4
Tango::EXTRACT	5
Tango::MOVING	6
Tango::STANDBY	7
Tango::FAULT	8
Tango::INIT	9
Tango::RUNNING	10
Tango::ALARM	11
Tango::DISABLE	12
Tango::UNKNOWN	13

A strings array called **Tango::DevStateName** can be used to get the device state as a string. Use the Tango device state code as index into the array to get the correct string.

Tango data type

A "define" has been created for each Tango data type. This is summarized in the following table

Type name	Type code	Value
Tango::DevBoolean	Tango::DEV_BOOLEAN	1
Tango::DevShort	Tango::DEV_SHORT	2
Tango::DevLong	Tango::DEV_LONG	3
Tango::DevFloat	Tango::DEV_FLOAT	4
Tango::DevDouble	Tango::DEV_DOUBLE	5
Tango::DevUShort	Tango::DEV_USHORT	6
Tango::DevULong	Tango::DEV_ULONG	7
Tango::DevString	Tango::DEV_STRING	8
Tango::DevVarCharArray	Tango::DEVVAR_CHARARRAY	9
Tango::DevVarShortArray	Tango::DEVVAR_SHORTARRAY	10
Tango::DevVarLongArray	Tango::DEVVAR_LONGARRAY	11
Tango::DevVarFloatArray	Tango::DEVVAR_FLOATARRAY	12
Tango::DevVarDoubleArray	Tango::DEVVAR_DOUBLEARRAY	13
Tango::DevVarUShortArray	Tango::DEVVAR_USHORTARRAY	14
Tango::DevVarULongArray	Tango::DEVVAR_ULONGARRAY	15
Tango::DevVarStringArray	Tango::DEVVAR_STRINGARRAY	16

continues on next page

Table 3 – continued from previous page

Type name	Type code	Value
Tango::DevVarLongStringArray	Tango::DEVVAR_LONGSTRINGARRAY	17
Tango::DevVarDoubleStringArray	Tango::DEVVAR_DOUBLESTRINGARRAY	18
Tango::DevState	Tango::DEV_STATE	19
Tango::ConstDevString	Tango::CONST_DEV_STRING	20
Tango::DevVarBooleanArray	Tango::DEVVAR_BOOLEANARRAY	21
Tango::DevUChar	Tango::DEV_UCHAR	22
Tango::DevLong64	Tango::DEV_LONG64	23
Tango::DevULong64	Tango::DEV_ULONG64	24
Tango::DevVarLong64Array	Tango::DEVVAR_LONG64ARRAY	25
Tango::DevVarULong64Array	Tango::DEVVAR_ULONG64ARRAY	26
Tango::DevInt	Tango::DEV_INT	27
Tango::DevEncoded	Tango::DEV_ENCODED	28
Tango::DevEnum	Tango::DEV_ENUM	29
Tango::DevPipeBlob	Tango::DEV_PIPE_BLOB	30
Tango::DevVarStateArray	Tango::DEVVAR_STATEARRAY	31

For command which do not take input parameter, the type code Tango::DEV_VOID (value = 0) has been defined.

Use **Tango::data_type_to_string** to convert the tango data type to a string. For older tango versions without that function a strings array called **Tango::CmdArgTypeName** can be used to get the data type as a string. Use the Tango data type code as index into the array to get the correct string.

Tango command display level

Like attribute, Tango command has a display level. The Tango::DispLevel type is a C++ enumeration starting at 0. The code associated with each command display level is already described in page

As for attribute, this parameter allows a graphical application to support two types of operation :

- An operator mode for day to day operation
- An expert mode when tuning is necessary

According to this parameter, a graphical application knows if the command is for the operator mode or for the expert mode.

Device server process option and environment variables

Classical device server

The synopsis of a device server process is

ds_name instance_name [OPTIONS]

The supported options are :

- **-h, -? -help**
  Print the device server synopsis and a list of instance name defined in the database for this device server. An instance name in not mandatory in the command line to use this option
- **-v[trace level]**
  Set the verbose level. If no trace level is given, a default value of 4 is used
- **-file=<file name path>**

Start a device server using an ASCII file instead of the Tango database.

- **-nodb**
  Start a device server without using the database.
- **-dlist <device name list>**
  Give the device name list. This option is supported only with the -nodb option.
- **ORB options** (started with -ORBxxx)
  Options directly passed to the underlying ORB. Should be rarely used except the -ORBendPoint option for device server not using the database

Device server process as Windows service

When used as a Windows service, a Tango device server supports several new options. These options are :

- **-i**
  Install the service
- **-s**
  Install the service and choose the automatic startup mode
- **-u**
  Un-install the service
- **-dbg**
  Run in console mode to debug service. The service must have been installed prior to use it.

Note that these options must be used after the device server instance name.

Environment variables

A few environment variables can be used to tune a Tango control system. TANGO_HOST is the most important one but on top it, some Tango features like Tango logging service or controlled access (if used) can be tuned using environment variable. If these environment variables are not defined, the software searches in the file **$HOME/.tangorc** for its value. If the file is not defined or if the environment variable is also not defined in this file, the software searches in the file **/etc/tangorc** for its value. For Windows, the file is **$TANGO_ROOT/tangorc** TANGO_ROOT being the mandatory environment variable of the Windows binary distribution.

TANGO_HOST

This environment variable is the anchor of the system. It specifies where the Tango database server is running. Most of the time, its syntax is

TANGO_HOST=<host>:<port>

host is the name of the computer where the database server is running and port is the port number on which it is listening. <host> should be a FQDN (Fully Qualified Domain Name). If you want to have a Tango control system which has several database servers (but only one database) in order to survive a database server crash, use the following syntax

TANGO_HOST=<host_1>:<port_1>,<host_2>:<port_2>,<host_3>:<port_3>

Obviously, host_1 is the name of the computer where the first database server is running, port_1 is the port number on which this server is listening. host_2 is the name of the computer where the second database server is running and port_2 is its port number. All access to database will automatically switch from one server to another one in the list if the one which was used has died.

Tango Logging Service (TANGO_LOG_PATH)

The TANGO_LOG_PATH environment variable can be used to specify the log files location. If not set it defaults to /tmp/tango-<user logging name> under Unix and C:/tango-<user logging name> under Windows. For a given device-server, the files are actually saved into $TANGO_LOG_PATH/{ server_name}/{ server_instance_name}. This means that all the devices running within the same process log into the same directory.

The database and controlled access server (MYSQL_USER, MYSQL_PASSWORD, MYSQL_HOST and MYSQL_DATABASE)

The Tango database server and the controlled access server (if used) need to connect to the MySQL database. They are using four environment variables called MYSQL_USER, MYSQL_PASSWORD to know which user/password they must use to access the database, MYSQL_HOST in case the MySQL database is running on another host and MYSQL_DATABASE to specify the name of the database to connect to. The MYSQL_HOST environment variable allows you to specify the host and port number where MySQL is running. Its syntax is

host:port

The port definition is optional. If it is not specified, the default MySQL port will be used. If these environment variables are not defined, they will connect to the DBMS using the root login on localhost with the MySQL default port number (3306). The MYSQL_DATABASE environment variable has to be used in case your are using the same Tango Database device server executable code to connect to several Tango databases each of them having a different name.

The controlled access

Even if a controlled access system is running, it is possible to by-pass it if in the environment of the client application the environment variable SUPER_TANGO is defined to true.

The event buffer size

If required, the event buffer used by the ZMQ software could be tuned using environment variables. These variables are named TANGO_DS_EVENT_BUFFER_HWM for the event buffer on a device server side and TANGO_EVENT_BUFFER_HWM for the event buffer on the client size. Both of them are a number which is the maximum number of events which could be stored in these buffers.

6.7.8 Known issues

Intended audience: developers

List of known issues and workarounds

- *Tango 9*
  - *Unable to launch Jive/DatabaseDS on a linux laptop without network enabled*
  - *Creating/Destroying device proxies on event_received hooks may cause deadlock*
  - *Setting event properties for state/string/boolean attributes will crash device server init*
  - *Pushing attributes outside of main device server thread may fail*

  - *Pushing attributes events on* read_$attribute *method after setting* attr_value *may crash the device*
  - *Pushing attributes within other attributes* read_attribute *methods may cause deadlock*
- *Tango 8*

Tango 9

Some of these issues are solved using EnsureOmniThread context manager on Tango >=9.4

Unable to launch Jive/DatabaseDS on a linux laptop without network enabled

- **known workaround:** Set TANGO_HOST=127.0.0.1:10000 (local loopback).

Creating/Destroying device proxies on event_received hooks may cause deadlock

- **known workaround:** Avoid creating/destroying proxies outside of the main thread.

Setting event properties for state/string/boolean attributes will crash device server init

- **known workaround:** Cleanup tango database from event config properties for the device before restart.

Pushing attributes outside of main device server thread may fail

- **known workaround:** Attribute pushing should be preferably done from *read_attribute*, *always_executed_hook*, *read_attr_hardware* or any device command executed from the main server thread.

Pushing attributes events on *read_$attribute* method after setting *attr_value* may crash the device

- **known workaround:** *attr.set_value()* must be called after *push_change_event*, otherwise device will crash.

Pushing attributes within other attributes *read_attribute* methods may cause deadlock

- **known workaround**: Push events for commands or own *read_attribute* method only.

Tango 8

...

6.8 Tango Core C++ Classes Reference Documentation

- Please refer to: C++ API documentation generated by Doxygen.

6.9 Contributing

Intended audience: developers

6.9.1 How to work with Tango Controls documentation

Intended audience: developers

When writing or improving the Tango Controls documentation it is worth to follow these guidelines. This will help in keeping it as consistent as possible. It is also important to know how the contents and the sources are structured to make your effort efficient. You will find necessary information below.

About this documentation

The documentation is written with the Sphinx markup language. It is a documentation framework based on Docutils and uses reStructuredText for providing content. For details please refer to Sphinx webpage.

The documentation sources are stored on GitHub: https://github.com/tango-controls/tango-doc .

It is publicised in HTML, PDF and EPUB formats on the readthedoc.io: http://tango-controls.readthedocs.io/

Some of the documents were not originally writen in Sphinx. These have been converted from other formats like HTML, LaTeX or Word. These may contain some residual bugs and syntax errors left after conversion. They will be consequently corrected when found.

Updating the documentation

If you find that some useful information is missing, misleading or you can think about any potential improvements please do either:

- send a request through the github project: https://github.com/tango-controls/tango-doc/issues
- or do correction by yourself.

If you decide to contribute by writing, the preferred way is to:

- clone or fork the repository,
- create your own local fix branch,
- when finished, send a pull request to the *origin/master* branch.

Note:

GitHub online edit

For small fixes, you may use GitHub online editing feature. It is a good practice to avoid direct commits to 'dev' nor to 'master' branch. Please select *Create a new branch and start pull request* before sending the change.

For details see *Documentation workflow tutorial*.

Building/previewing documentation locally

To build the documentation you will need Sphinx environment which is a Python package. Please consult Sphinx webpage for details on how to install it.

> **Warning:** Some standard Python packages contain a buggy version of **Breathe**. You may need to install it from sources. You may use the following command:
>
> **pip install --upgrade git+git://github.com/michaeljones/breathe@cc8f830**

After having Sphinx and Breathe installed you will be able to build the documentation:

- go to (**cd**) folder where you have cloned the repository
- call **sphinx-build source build**

This will build HTML output in the `build` folder.Then you may use any web browser to view the documentation.

Note: The build process generates a lot of warnings. Don't worry. We will work to remove as much of them as possible but some are inevitable.

For more details and step-by-step guidance pleas refer to *Documentation workflow tutorial*.

Sources structure

Tango Controls versions

The *readthedocs.io* allows to publish various versions of the documentation. It is achieved by providing branches in a git repository. The official version branches are these named numerically as Tango Controls versions: #.#.#.

Chapters and headers

Chapters' order is defined by the main table of contents. It is contained in a file `source/contents.rst` and referenced index.rst files.

To keep chapters levels consistent please use the following underlining schema:

- First level underline: ==== (equal signs)
- Second level: —— (dashes)
- Third level: ~~~ (waves)

References

Basic list of Reference Names is provided within `source/conf.py` as a *rst_epilog* variable. The contents of this variable is dynamically concatenated to the end of each *.rst* file during the building process. As of today, it provides some common hyperlink targets. However, it is planned to include some common substitutions. The list allows to use some entries like *`Tango webpage`_* which will be rendered as Tango webpage

Glossary

Glossary entries (definitions) may be provided as content of any document. However, there is a `source/reference/glossary.rst` file. Its purpose is to centralise short definitions of main concepts of Tango Controls. Entries defined there may be referenced as *:term:`...`* at any location in the documentation.

Images

Fore each document, images should be stored in a sub-folder of the folder where the document is stored. As an example, please refer to `source/tools-and-extensions/astor`. When a folder contains more than one document the images folder should be named as the document itself. See `source/getting-started/installation/tango-on-windows` as an example.

Configuration

sources/conf.py

This is a standard *build configuration file* used by Sphinx. Among others the project name, version and copyright info are defined there. Please refer to conf.py documentation.

requirements.txt

This is a standard **`pip`** requirements file used to fix packages version. Currently it contains Sphinx only.

readthedocs.yml

This is a configuration file for the *readthedocs* application. It provides some fine-grain settings. For Tango Controls it limits output formats to standard HTML, PDF and EPUB. Leaving this setting blank will lead to some problems with the build process at readthedocs.

6.9.2 How to contribute to Tango Controls C++ library (cppTango)

Intended audience: developers

Everyone is welcome to contribute to cppTango project. If you don't feel comfortable with writing core cppTango we are looking for contributors to documentation or/and tests.

Workflow

- Fork tango-controls/cppTango repository
- Clone fork
- If you want to work on TANGO 9 LTS, switch to tango-9-lts branch if needed: `git checkout tango-9-lts`
- Add remote: `git remote add upstream git@github.com:tango-controls/cppTango.git`
- Start a new branch for your contribution and switch to the new branch `git checkout -b my-contrib`
- Reset your new branch to align it with upstream/tango-9-lts branch: * `git fetch upstream` * `git reset --hard upstream/tango-9-lts`
- Implement your changes:
  1. increment version (patch or minor)
  2. update RELEASE_NOTES.md - add reasonable description of the changes
  3. implement changes
- Commit and push
- Before submitting a new pull request:
  - `git fetch upstream`
  - `git rebase upstream/tango-9-lts`

  OR
  - `git pull --rebase upstream/tango-9-lts`
- Create pull request
- Pull requests should be ALWAYS made to tango-9-lts branch, not to a master branch, unless they are related to Tango V10 development.
- You can have a look at GitHub Standard Fork & Pull Request Workflow gist to get more details on how to fork and submit a Pull Request:

PLEASE PROVIDE TEST CASES WHEN RELEVANT FOR YOUR CONTRIBUTION

reStructuredText and Sphinx

Documentation is written in reStructuredText and built with Sphinx - it's easy to contribute. Please refer to *How to work with Tango Controls documentation* for more details on how to contribute to the documentation.

Releasing a new version

From time to time a new version should be released. Anyone who wishes to see some features of the tango-9-lts branch released is free to make a new release. The basic steps required are as follows:

Pick a version number

- Semantic version numbering is used: <major>.<minor>.<patch>
- Major version number is systematically increased when there is a new version of the TANGO IDL.

Create an issue in Github

- This is to inform the community that a release is planned.

- Use a checklist similar to the one below:

```
Task list:
- [ ] Read steps in the ["Releasing a new version" documentation](https://tango-
↪controls.readthedocs.io/en/latest/development/contributing/cpptango-contribution.
↪html#releasing-a-new-version) for making a release
- [ ] Bump the version (if not already done) on tango-9-lts branch
- [ ] Merge the wished Pull Requests into tango-9-lts (Make sure Travis and␣
↪appveyor are OK on the PR before the merge)
- [ ] Make sure Travis and appveyor are OK on tango-9-lts branch
- [ ] Edit the CHANGELOG.md file
- [ ] Edit the RELEASE_NOTES file
- [ ] Make sure the documentation is updated (readthedocs)
- [ ] Create a release tag on GitHub, from tango-9-lts branch. Flag it as a pre-
↪release.
- [ ] Advertise the pre-release on the mailing list to ask for feedback and testers
- [ ] Wait for at least 2 weeks before to do the next steps to give some time to␣
↪get feedback from the first testers
- [ ] If no major bug was reported, the GitHub release can be flagged as "Latest␣
↪release",
else the bugs should be fixed first and a new pre-release should be created.
- [ ] Add Windows installers as assets to the GitHub release
- [ ] Generate Doxygen documentation for this new release and update cppTango-docs␣
↪with the new generated documentation
- [ ] Update cppTango version in TangoSourceDistribution repository
- [ ] Advertise the release as the new stable release on the mailing list and on␣
↪the tango-controls website (News)
- [ ] Close this issue
```

- A check list in this form on github can be ticked off as the work progresses.

Bump the version on tango-9-lts branch

- Update the following lines at the beginning of CMakeLists.txt to match the new release number
- For cppTango 9.3.3, you will get:

```
#need to define the version of the library
set(MAJOR_VERSION "9")
set(MINOR_VERSION "3")
set(PATCH_VERSION "3")
```

Merge the wished Pull Requests

- Create/Review/Merge the Pull Requests containing the bug fixes and new features you would like to see in the new release. Do this with the help/review of the cppTango developers team.
- Before merging a Pull Request into tango-9-lts branch:
  - Travis and appveyor tests must pass on this Pull Request
  - the Pull Request must have been approved by a member of the cppTango developers team (different than the Pull Request creator)

Make sure Travis and appveyor are OK on tango-9-lts branch

- Once the Pull Requests have been merged on tango-9-lts, make sure that Travis and appveyor builds are still OK on that branch. If not, bad luck - you'll have to fix it first and go back a few steps…

Edit the changelog

- Create a Pull Request to edit CHANGELOG.md file.
- List all the Pull Requests which have been merged since the last release in this CHANGELOG.md file.
- Get these changes reviewed before proceeding.

Edit the release notes

- Create a Pull Request to edit RELEASE_NOTES.md file.
- We took the slack release notes as inspiration for the first cppTango 9.3 release notes. According to slack team:

  They should take the basic facts, reduce jargon, put them into words people could relate to; words that might be lightly poetic and slightly absurd, but stopped short of grating, and... nothing less than informative.

- You can refer to slack: a little thing about release notes for more details.
- Get these changes reviewed before proceeding.

Make sure the documentation is updated (readthedocs)

- If needed, update the documentation which is on readthedocs
- Please refer to *How to work with Tango Controls documentation* for more details on how to contribute to the documentation.

Create a release tag on GitHub, from tango-9-lts branch. Flag it as a pre-release.

- Go to GitHub cppTango releases page and click on `Draft a new release`
- Create a tag for the new version from tango-9-lts target and and prefix the tag name with the letter v. If the tag isn't meant for production use, add a pre-release version after the version name. For instance, v9.3.3-rc1
- Copy/paste the release notes from RELEASE_NOTES.md for this version in the release description
- Flag this version as pre-release by ticking the checkbox

Advertise the pre-release

- Advertise the pre-release on the tango mailing list
- Eventually submit a news on tango-controls web site to ask for feedback and testers

Wait for at least 2 weeks before to do the next steps

- This is to give some time to get some feedback from the first testers

After at least 2 weeks

- If no major bug was reported, the GitHub release can be flagged as "Latest release". If a pre-release version tag was used before (like v9.3.3-rc1), you should create a new tag corresponding to the same software version but with a name corresponding to a stable release (e.g. v9.3.3). Be careful if some new Pull Requests got merged into tango-9-lts branch in the meantime. What you want is to use a tag which corresponds to the version which has been tested successfully during the previous weeks.
- Else the major bugs should be fixed first and a new pre-release/release candidate should be created before to proceed with the final steps.

Add Windows installers as assets to the GitHub release

- Since appveyor artifacts are only kept for 6 months, provide Windows installers generated by the appveyor build of the release as assets of this new GitHub release so they are always available from the GitHub cppTango releases page.

Generate Doxygen documentation

- for this new release and update cppTango-docs with the new generated documentation

Update cppTango version in TangoSourceDistribution repository

- If a stable release has been created, update tango-controls/TangoSourceDistribution distribution.properties file
- In particular, the following variables must be updated:
  - cppTango - e.g. `cppTango=v9.3.3`
  - version-info - version_info must be computed depending on the cppTango library version using the following formula:

```
#VERSION_INFO = current:revision:age
# where
#current = tango lib major + $age
#revision = tango lib patch
#age = tango lib minor.
```

So for tango 9.3.2 it would be:

```
#current = 9 + 3 = 12
#revision = 2
#age = 3
version-info=12.2.3
```

The above formula must be used to compute version-info property to ensure the installed libtango shared library file will contain the cppTango release version in its name.

Advertise the release as the new stable release

- Advertise the release on the tango mailing list
- Submit a release news on tango-controls web site

Close off release issue

- All the items on the check list should be ticked off by now.
- Close the issue.

CHAPTER

SEVEN

TOOLS AND EXTENSIONS

Intended audience: all

7.1 The most popular tools

Intended audience: beginners, all

Tango Controls provides various tools. You may refer to *Index of tools* for a detailed list of available tools, applications and libraries or look below for the most used ones.

- *Astor* which is a tool for management of Tango Controls system.
- *Jive* which is used to configure components of the Tango Controls and browse a static *Tango Database*.
- *AtkPanel* which is a generic control panel.
- *Pogo* which is a *device class* and *device server* generator.
- *JDraw and Synpotic* which is a synoptic panels builder and viewer.
- *Taurus* is a Python library for creating GUI applications.

7.2 Built-in tools

These tools are delivered as a part of Tango Controls core package.

7.2.1 Astor (TANGO Manager)

Intended audience: administrators, developers, users

Introduction to Astor

Intended audience: administrators, developers, users

Goal

- The first goal is to know at a quick glance, if everything is OK in a control system, and otherwise to be able to diagnose a problem and solve it.
- The second goal is to configure the control system and its components.
- The third goal is to have long term analysis on components (logs, statistics, usage,. . . .)

Principle

- On each host to be controlled, a device server (called *Starter*) takes care of all device servers running (or supposed to) on this computer.
- The controlled server list is read from the TANGO database.
- A graphical client (called Astor) is connected to all Starter servers and is able to:
  - Display the control system status and component status using coloured icons.
  - Execute actions on components (start, stop, test, configure, display information, ...
  - Execute diagnostics on components.
  - Execute global analysis on a large number of crates or database.

To control a host in remote, the TANGO device server Starter must be running on it.

Warning: The starter device must have a specific name to be recognized by astor. This name must be **tango/admin/{hostname}** (e.g. *tango/admin/hal*).

Running Astor

- **`Astor`** is a **Java** program using **Swing** classes. Classes has been compiled and the jar file has been built whith **`java-1.7`**.
- To start the application, start the script file: **`$TANGO_HOME/bin/astor`**
- **There are 3 modes to start Astor:**

Parameter	Mode
-rw or none	**`Astor`** is fully **READ/WRITE**"
-db_ro	**`Astor`** is **READ/WRITE** but **`Database`** is **READ_ONLY**
-ro	**`Astor`** is fully **READ_ONLY**

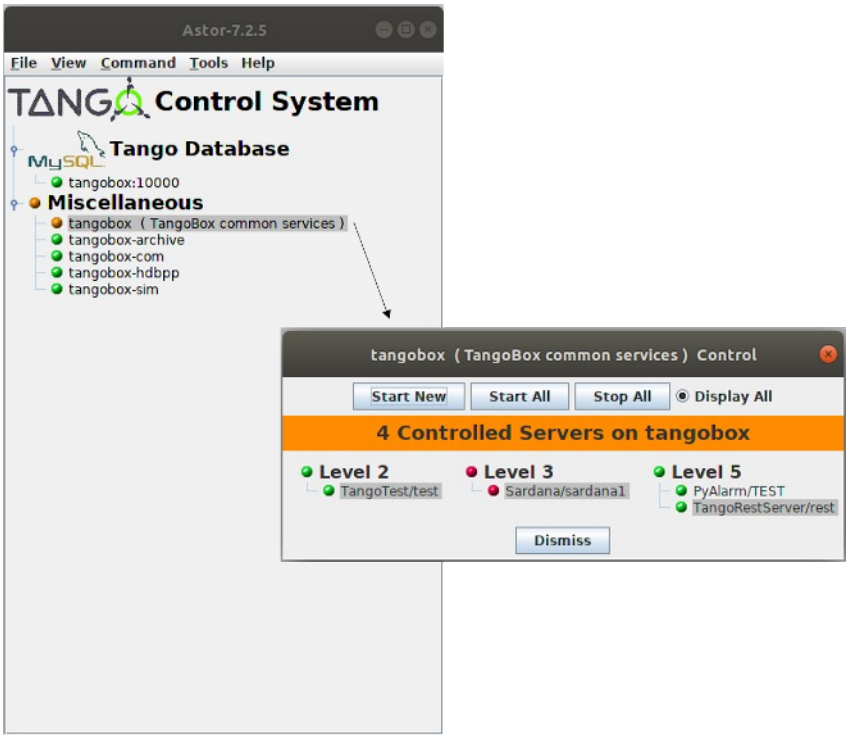
Display

- At startup, Astor display a tree where node could be a family of hosts (see Starter properties), and leaf are hosts where a Starter device server is registred in database.
- The icon of the leaf depends on the controlled device servers status as the following definition:

For Hosts
All controlled servers are running.
Starter is starting server(s).
At least, one controlled server is stopped and one is running.
All controlled servers are stopped.
Starter is not running on host.

For Servers
Server is running
Server is running but not alive (Starting ?)
Server is not running.

Host Control



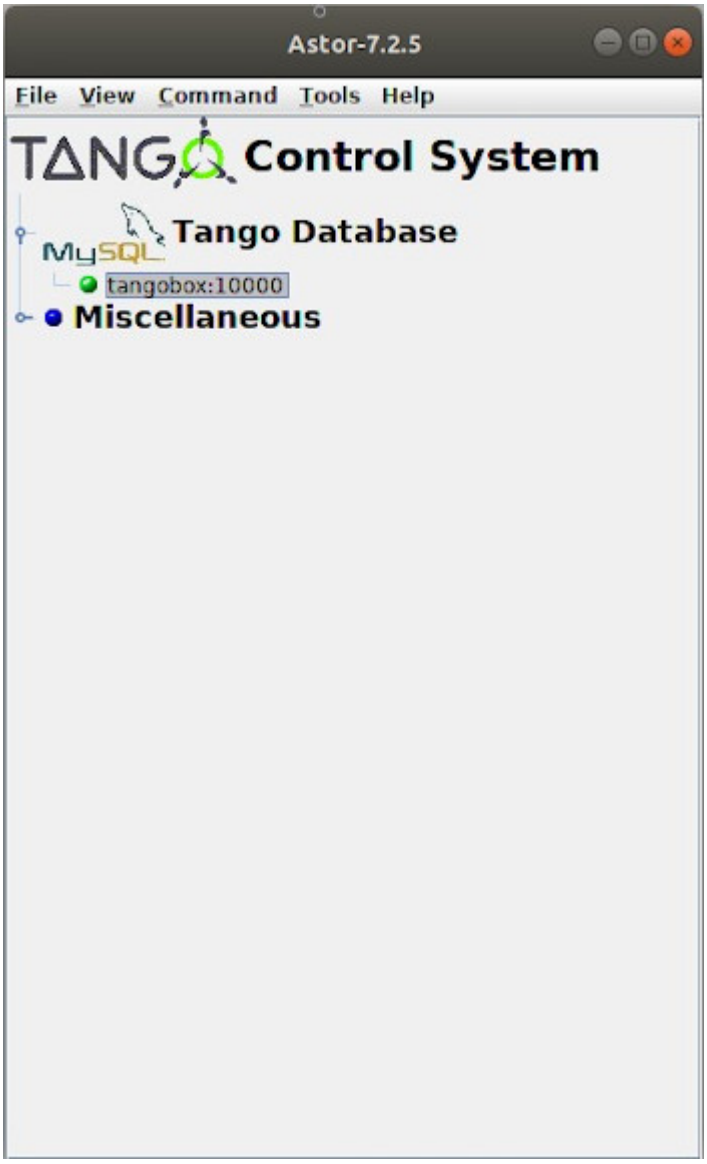
Source

You can download the project here

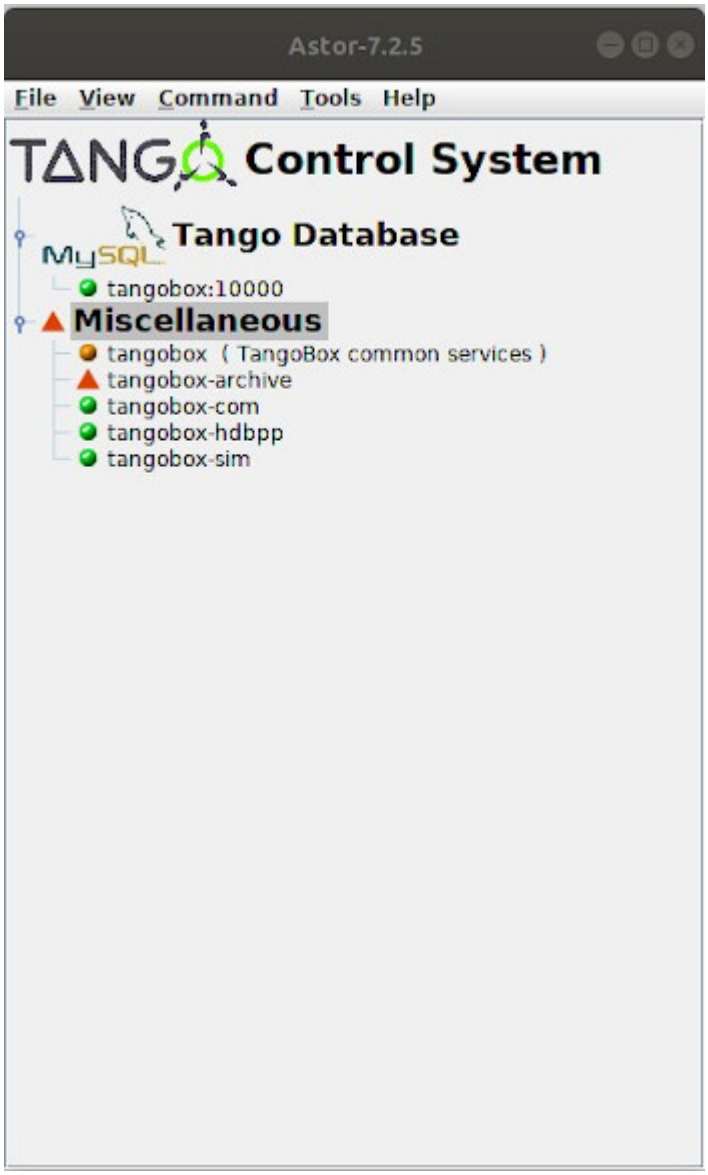
Astor Main Window

Intended audience: administrators, developers, users

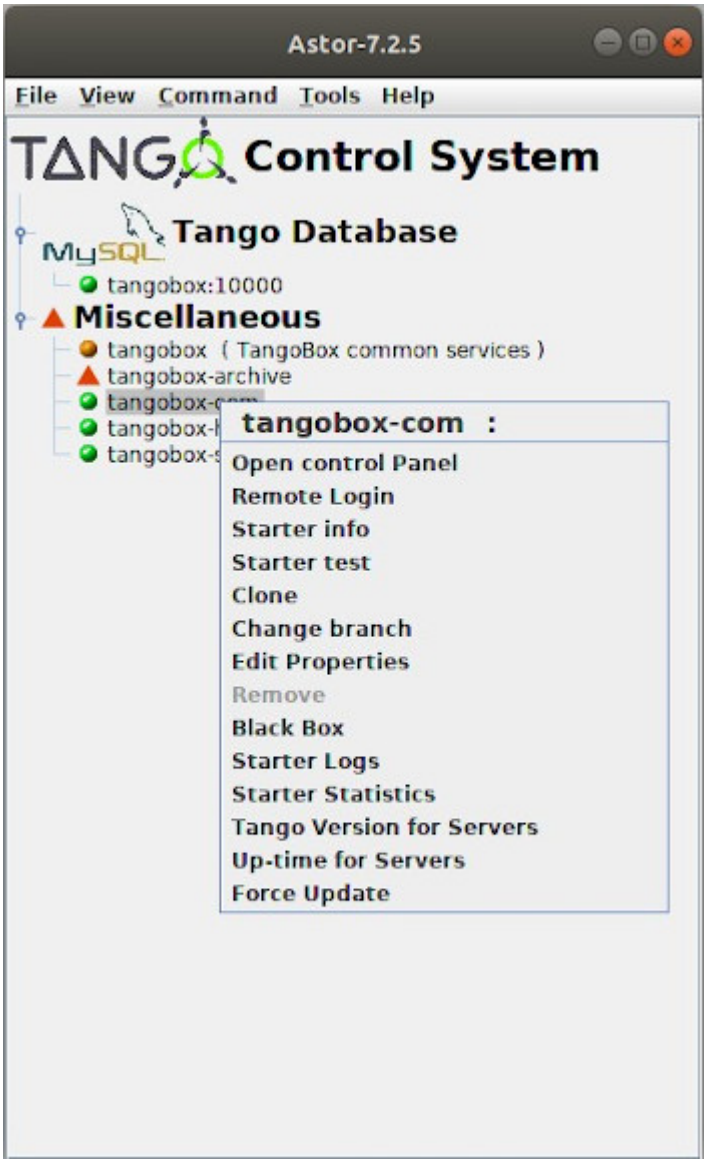
- On the main window, the control system is displayed as a tree.
- Where the root is TANGO, the branch are hosts family and leaf are hosts.
- The Astor main window is able to give a global status of the control system.



- The tree is collapsed, but the node icon display the status of the hosts under.



- An individual control on a host could be done by a click on a host.
- A popup menu will be displayed with a right click on a host.



Host Status Window

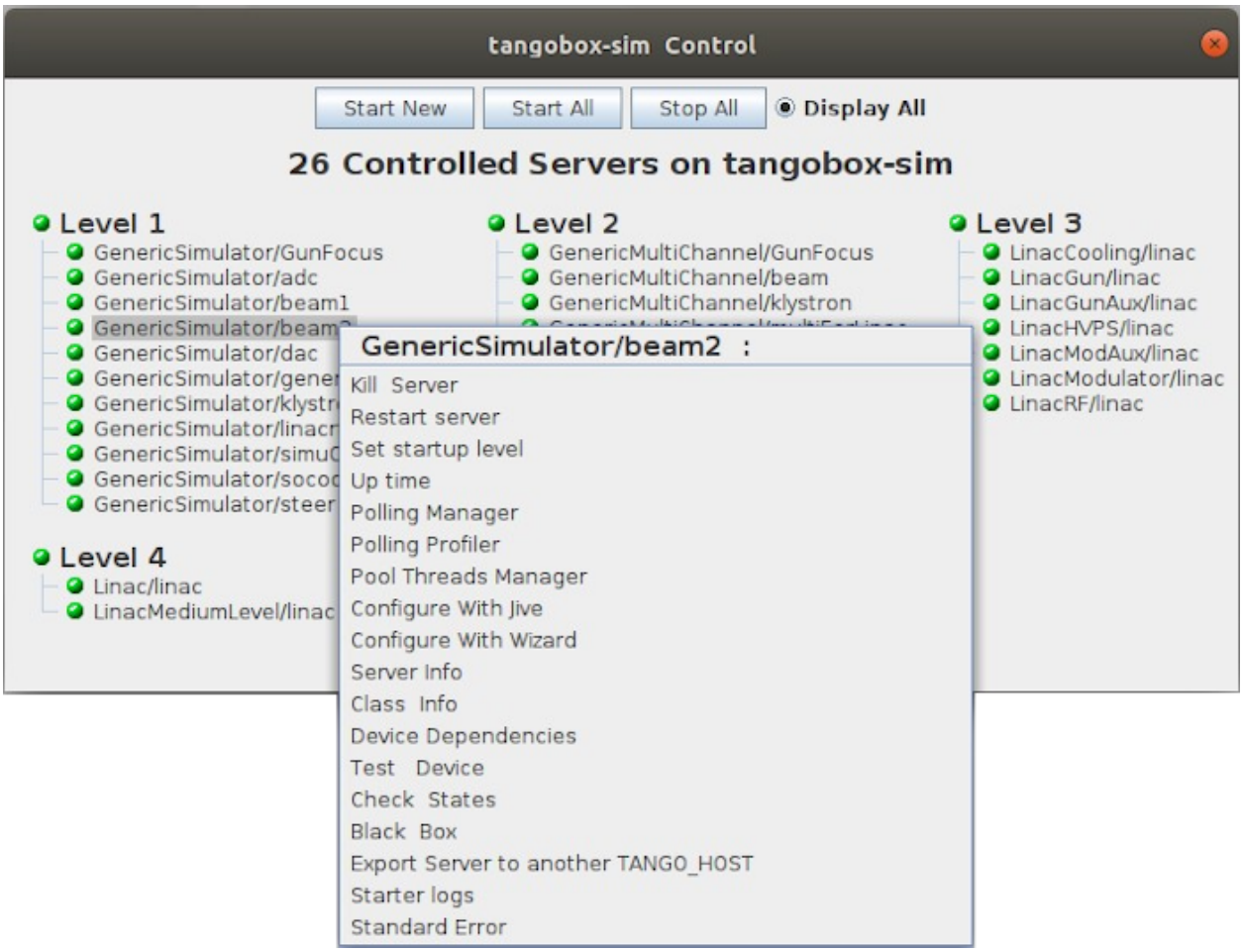
Intended audience: administrators, developers, users

- When a host window has been opened, all servers controlled on this host are displayed.
- The color of the server define its state:

Server is running
Server is running but not alive (Starting ?)
Server is not running.

- These servers are ordered by startup level.

- A popup menu will be displayed with a right click on a server to Start/stop/test… it.



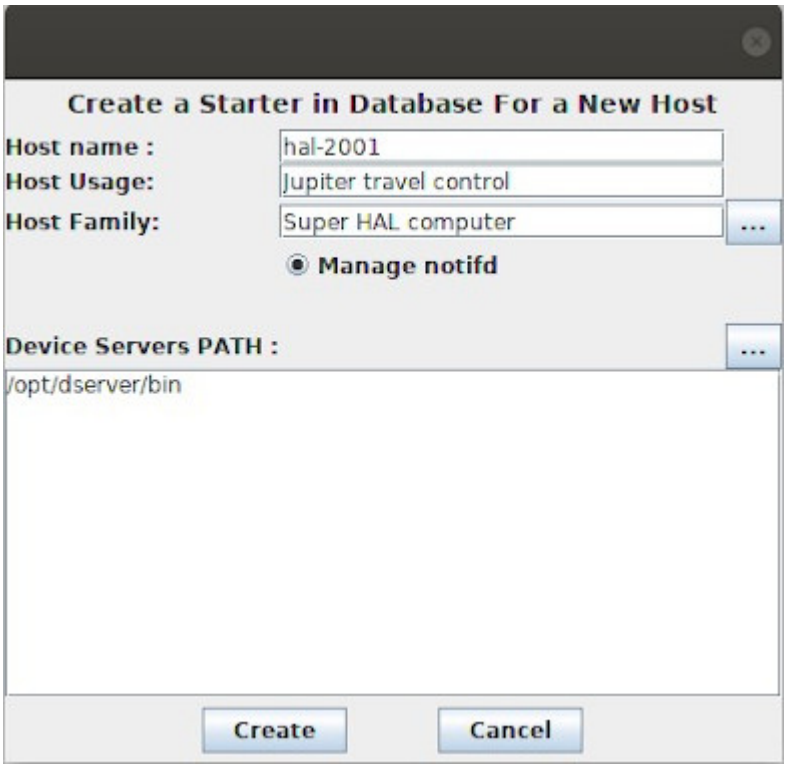
Note:

- It is possible to display "Not Controlled" servers if any.
- These servers are not taken into account to compute host state.

Add a new controlled host

Intended audience: administrators, developers

- Use the *Add a New Host* in *Command* menu.
- Or clone an existing one using host popup menu. And fill the fields of following window:



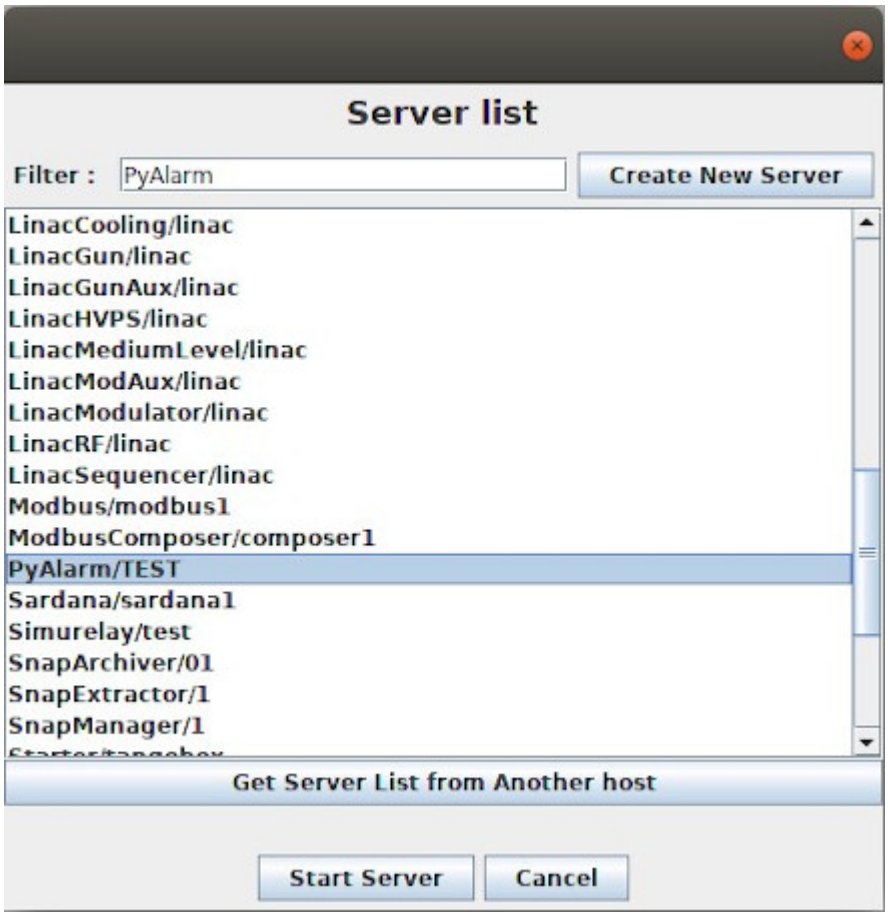
- Click on *Create* button.
- A new server *Starter* will be created in TANGO database.
- And the new host will appear in the tree under the specified branch.
- Start a new remote loggin session to start it from host popup menu
- Finally start Starter on host.

Start a new server with Astor

Intended audience: administrators, developers

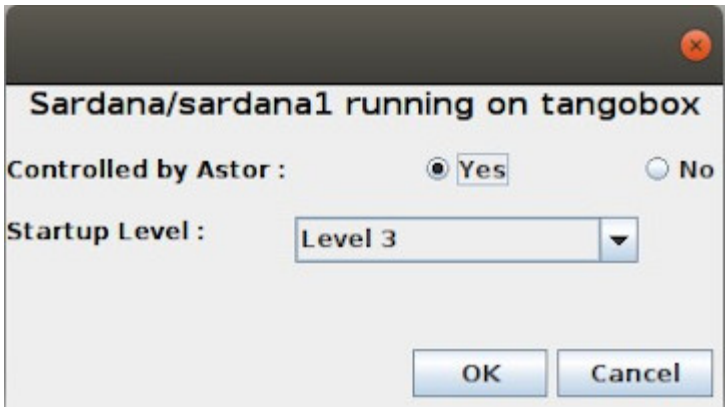
If the server is already defined in database

- Be sure that a **`Starter`** device server is running on the same host. If there is no **`Starter`** running, start it.
- On **`Astor`** main window, click on the host to open *host window*. On this window, click on *Start New* button.
- A window with a list of all servers defined in TANGO database will be displayed.



- Select the expected server and click on **Start server** button.

- And configure **Controlled by Astor** and **Startup Level**.



If the server is NOT already defined in database

- You can declare the server in database using Jive.
- Or click on *Create New Server* button to start the *wizard*.

If the server has been started from shell

The default startup level is not set and the server is not controlled by Astor.

That means that it does not appear in *host window*.

To see it click open **Not Controlled** level. And change it startup level if needed.

New Server Wizard

Intended audience: administrators, developers

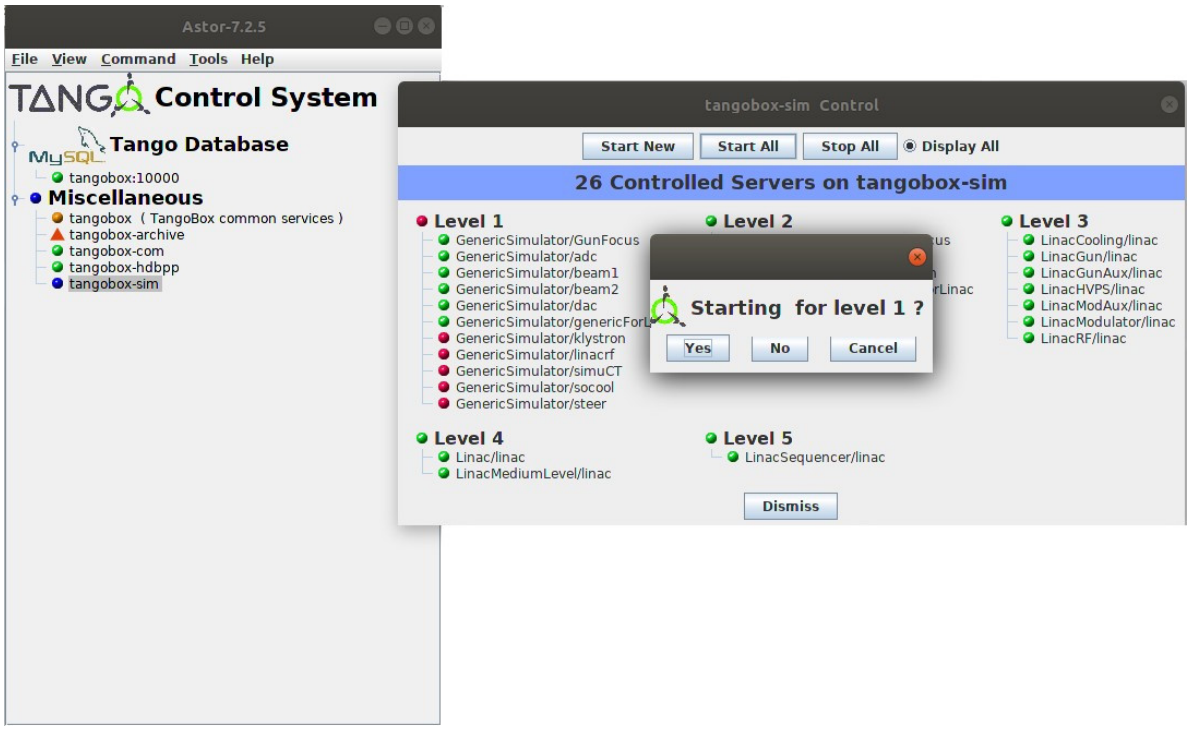
Multi Servers Start/Stop

Intended audience: administrators, developers, users

- You can start or stop:
  - All servers on a host by clicking on *Start All* or *Stop All* button on *host window*.
  - All servers on all hosts of a hosts family by using popup menu on a hosts family.
  - All servers on all hosts of the control system by using *Command* menu in main window.

  The servers will be started in ascending order of startup level and stopped in descending order.

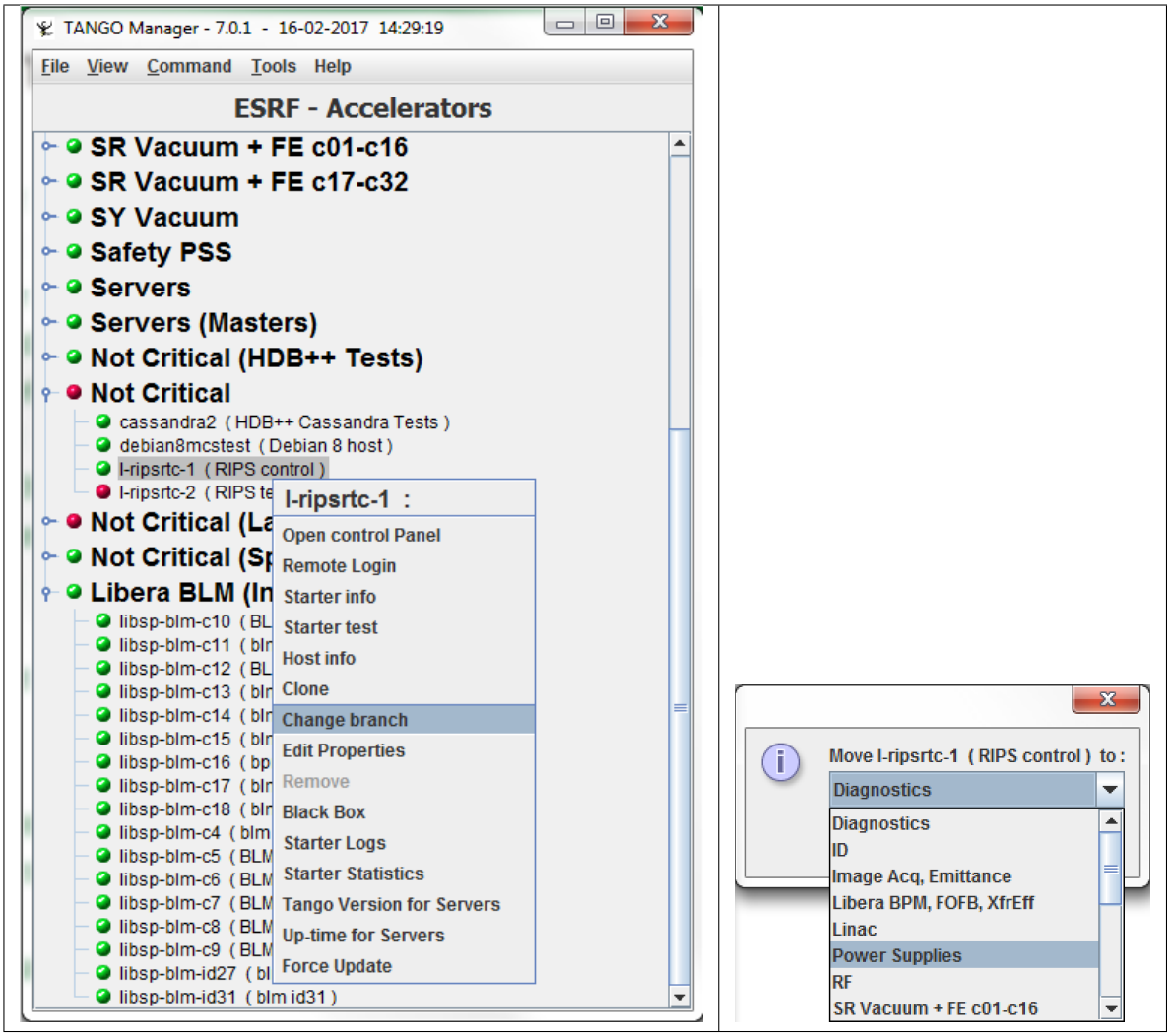
  For each level a confirmation dialogbox will be popuped.



Tree Branch Management

Intended audience: administrators, developers

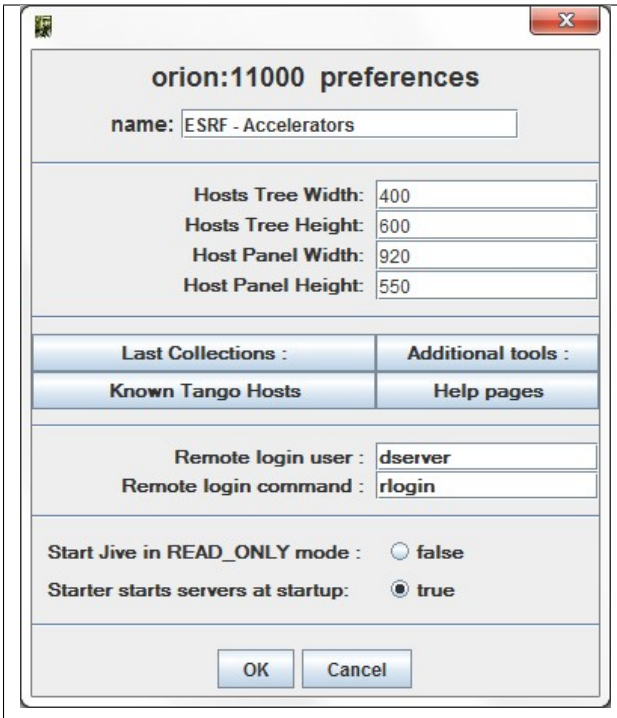
- On the main window, the control system is displayed as a tree. Where the root is TANGO, the branch are hosts family and leaf are hosts.
- The host's family is defined by the Starter property called HostCollection. But it is easier to use cut and paste option on popup menu on host as follow:



Astor Configuration

Intended audience: administrators, developers

Astor can be configured using the following window. Click on *File* → *Ctrl System Preferences* menu to open



- *Define control system name*
- *Define default window sizes*
- *Additional java tools.*
- *Additional html pages.*
- *Multi Tango Host.*
- *Host remote loggin command.*
- *Host remote loggin user.*
- *Jive in READ_ONLY mode.*
- *Starter starts servers at startup*

Note: To disable the *Preference* menu, start Astor with *-DNO_PREF=true*.

Control system name

This name will be displayed on top of Astor main window.

It could be useful when user has to manage several control systems.

Default window sizes

Hosts tree Width/Height defines the preferred size for the tree object in main window.

Host window Width/Height defines default maximum size for host window.

If the number of hosts needs a bigger window, scroll bars will be added.

Last Collections

Hosts can be distributed in families.

In the tree, the families are sorted by alphabetic order.

The property LastCollections give the possibility put collections (families) at the end of the tree.

It could be useful to put at the end of the tree, families like Not Critical, In Test, ...

Additional html pages

The property *HtmlHelps* give the possibility to add specific html pages. This is a string array property.

- The first line is the message displayed in help menu.
- The second one is the URL address for the specified page.

The following example add a link to the Tango device servers pages:

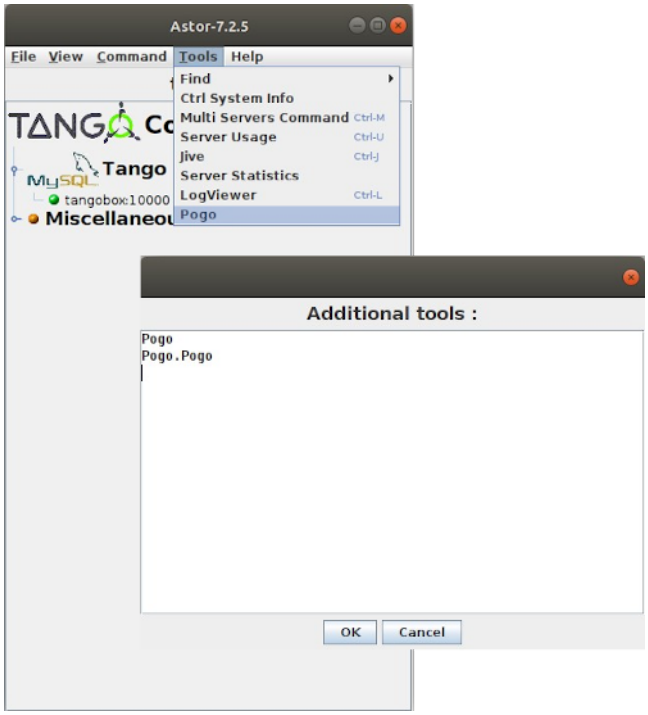
Additional java Tools

The property *Tools* give the possibility to add java class to tools menu. These additional tools could be specific for one control system This is a string array property.

One tool (java class) is defined by two strings:

- The first line is the message displayed in tools menu.
- The second one is the class (and package) to be instancied.

The following example add 3 java tools used on ESRF machine control system for TACO HDB:



Remarks

- The tools java classes need to have a constructor with a *JFrame* parameter as parent.
- It needs also to not to exit if it has been instancied from a parent application.
- The tools class or jar file needs to be found in `$CLASSPATH` of **`Astor`** startup.

KnownTangoHosts

This property give a list of `TANGO_HOST` to be controlled by Astor. The `TANGO_HOST` can be changed by typing a new name but the specified list will be available in a combo box.

RloginCmd

This property give the possibility to change the default remote loggin command. The default command is **`telnet`** (or **`rlogin`** if *RloginUser* is defined).

RloginUser

This property gives the possibility to set the remote loggin to the specified user.

Jive in READ_ONLY mode

If this property is true the **Jive** application will be instancied in READ_ONLY mode. If the property is not set or false, **Jive** is in READ_WRITE mode.

Starter starts servers at startup

This boolean property allows the *Starter* to start the device servers when it starts.

If it is false, the starter will not start any server when it will be launched.

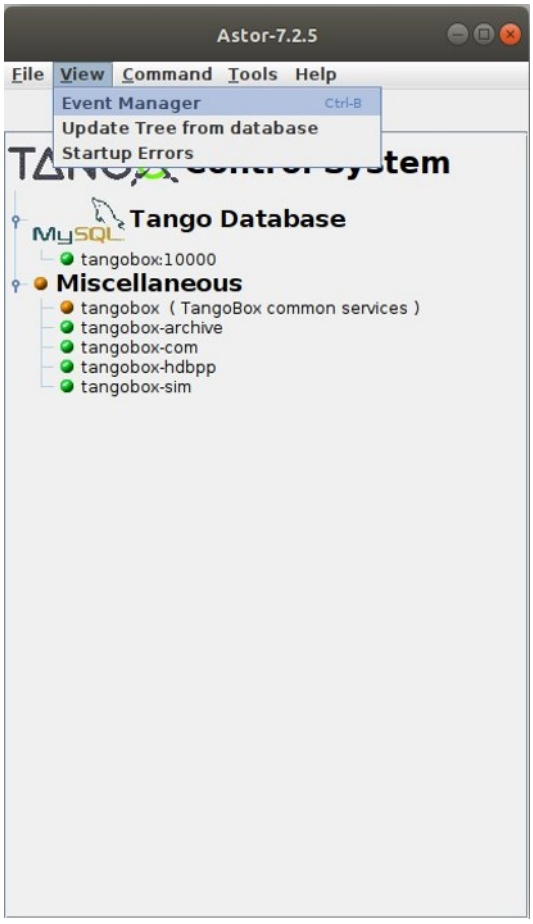
It could be useful when a large control system is re-started (e.g. after an electrical power cut) to do not overload the Tango database.

Event Manager

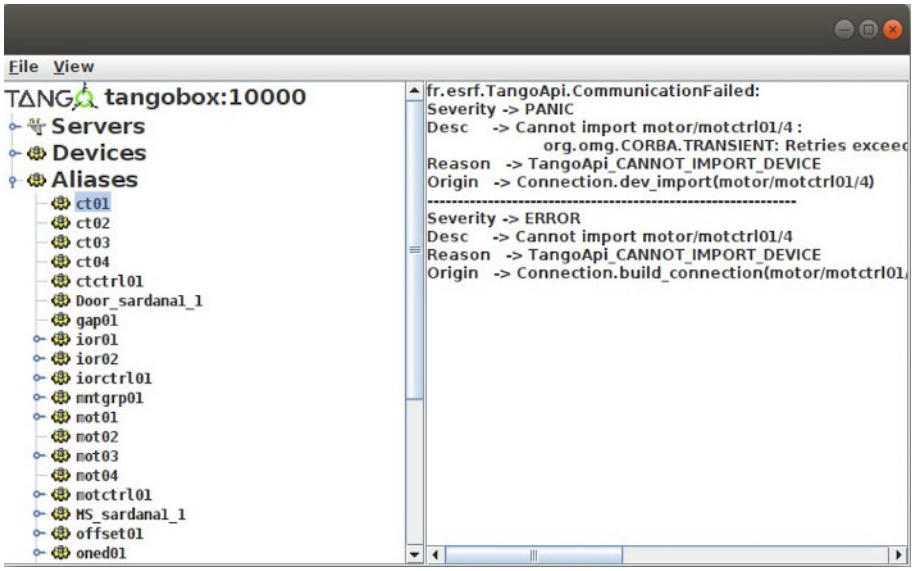
Intended audience: administrators, developers

Astor proposes to browse by server, by device or by alias. It could be useful:

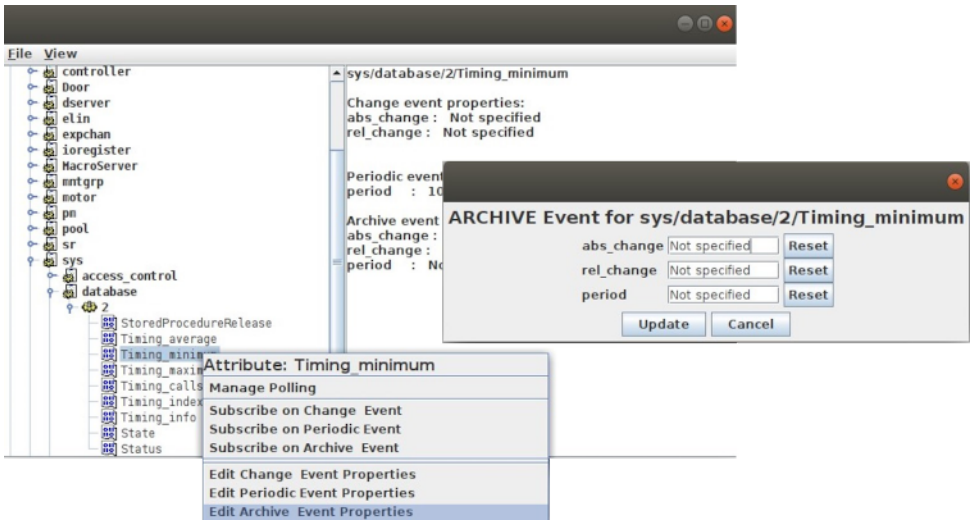
- *to see information*
- *to configure events*
- *to manage polling*
- *to check events*



Server, Device or Alias information



Configure the events for specified attribute

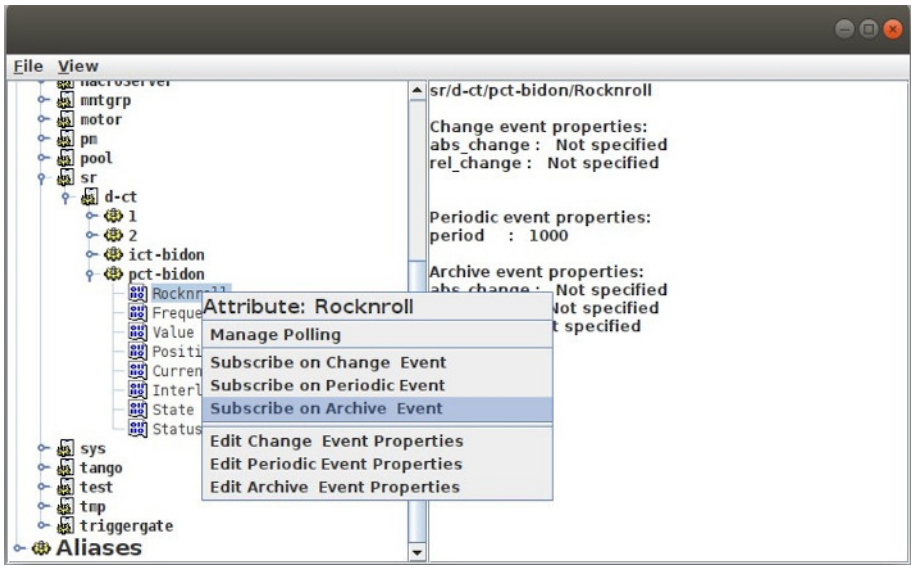


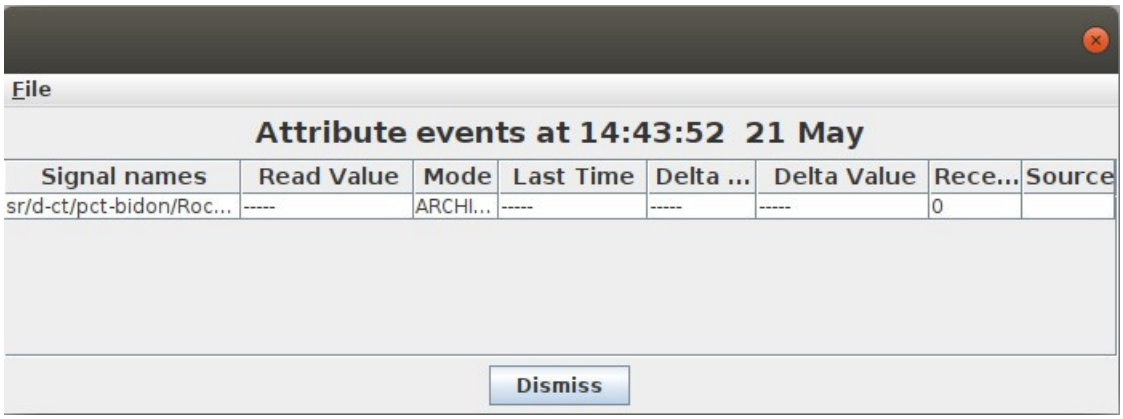
Manage polling for Server, Device or Alias

By a click on server or device menu, the *polling window* will be displayed.

Configure and Test events

By a click on attribute menu, the event tester window will be displayed with event information.





Polling Management

Intended audience: administrators, developers

A Tango device attribute can be polled periodically. To configure this polling, a diagnostic tool is available. It displays when each attribute has been polled the last four times (see following figure) and how much time was required to read and set it.

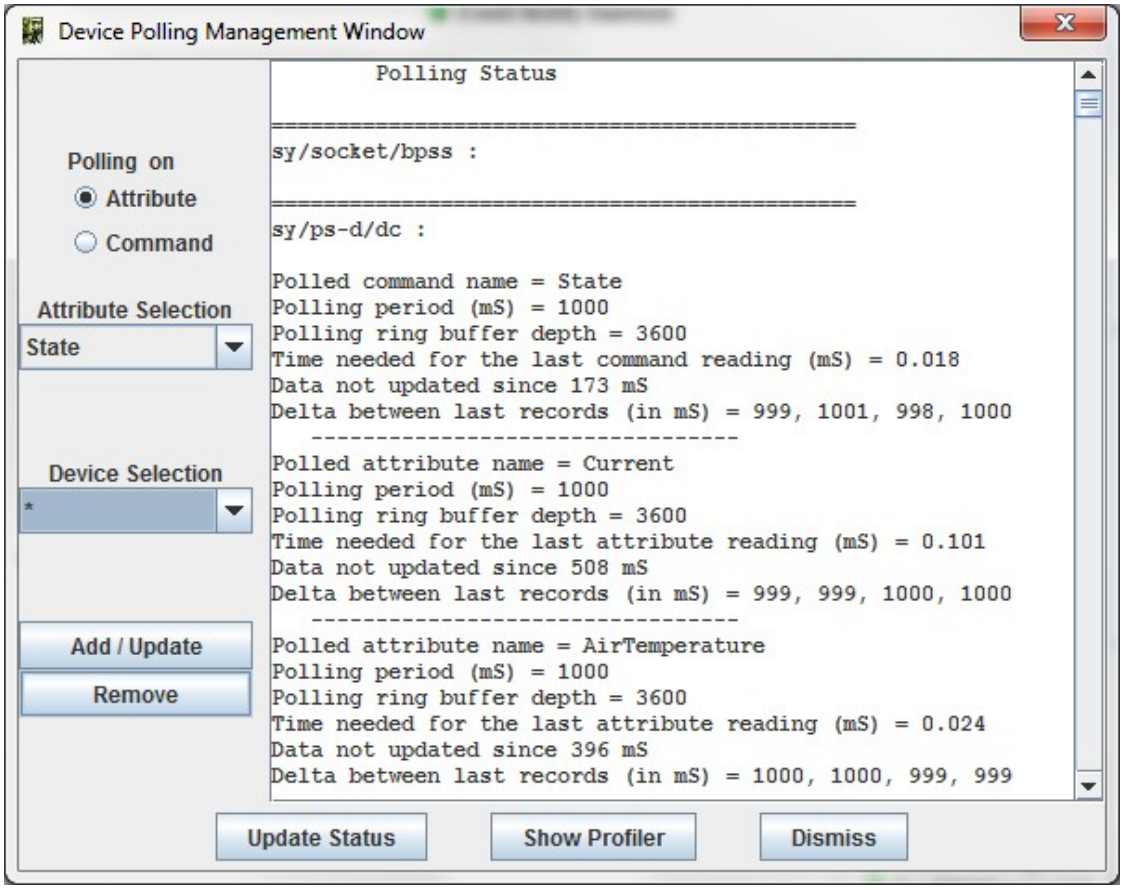
It is very useful to tune the device attribute polling when a server has several devices with many attributes each.

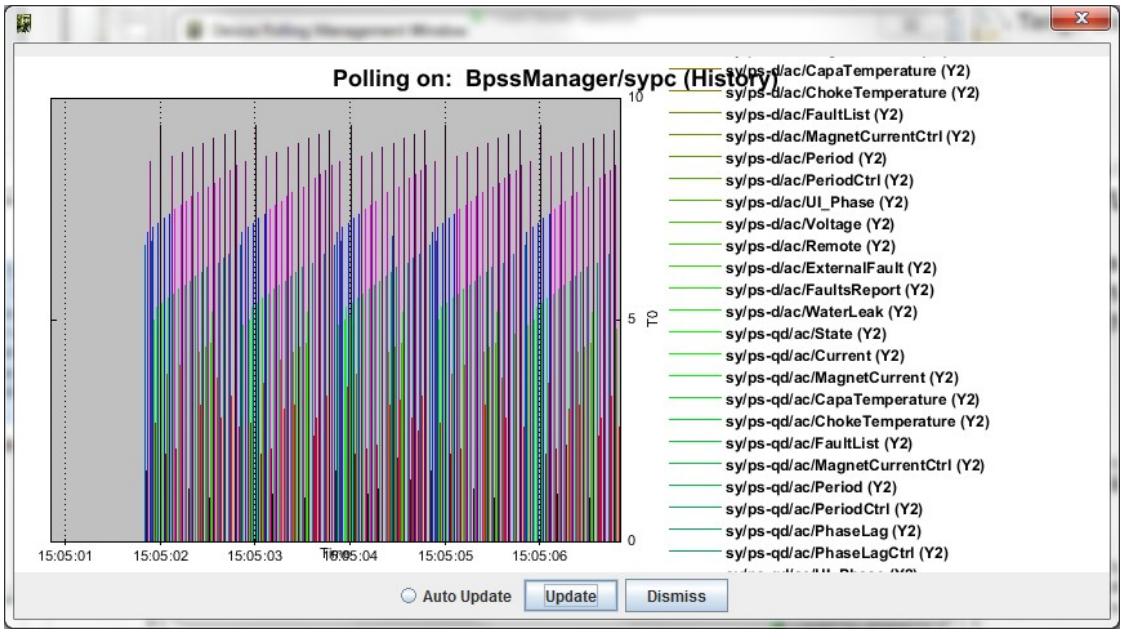
Or in case of a hardware problem with time out in reading. This window could be popuped from *host window* or from *event manager*.

It display polling information for one or all attributes for one (or all) devices.

It allows to remove, add or change polling.

It allows to display polling graphically as a profiler.


Device Polling Management Window
Polling Status
Polling on
Attribute
Command
Attribute Selection
State
Device Selection
*
Add / Update
Remove
sy/socket/bpss :
sy/ps-d/dc :
Polled command name = State
Polling period (mS) = 1000
Polling ring buffer depth = 3600
Time needed for the last command reading (mS) = 0.018
Data not updated since 173 mS
Delta between last records (in mS) = 999, 1001, 998, 1000
Polled attribute name = Current
Polling period (mS) = 1000
Polling ring buffer depth = 3600
Time needed for the last attribute reading (mS) = 0.101
Data not updated since 508 mS
Delta between last records (in mS) = 999, 999, 1000, 1000
Polled attribute name = AirTemperature
Polling period (mS) = 1000
Polling ring buffer depth = 3600
Time needed for the last attribute reading (mS) = 0.024
Data not updated since 396 mS
Delta between last records (in mS) = 1000, 1000, 999, 999
Update Status
Show Profiler
Dismiss


Polling on: BpssManager/sypc (History)
sy/ps-d/ac/CapaTemperature (Y2)
sy/ps-d/ac/ChokeTemperature (Y2)
sy/ps-d/ac/FaultList (Y2)
sy/ps-d/ac/MagnetCurrentCtrl (Y2)
sy/ps-d/ac/Period (Y2)
sy/ps-d/ac/PeriodCtrl (Y2)
sy/ps-d/ac/UI_Phase (Y2)
sy/ps-d/ac/Voltage (Y2)
sy/ps-d/ac/Remote (Y2)
sy/ps-d/ac/ExternalFault (Y2)
sy/ps-d/ac/FaultsReport (Y2)
sy/ps-d/ac/WaterLeak (Y2)
sy/ps-qd/ac/State (Y2)
sy/ps-qd/ac/Current (Y2)
sy/ps-qd/ac/MagnetCurrent (Y2)
sy/ps-qd/ac/CapaTemperature (Y2)
sy/ps-qd/ac/ChokeTemperature (Y2)
sy/ps-qd/ac/FaultList (Y2)
sy/ps-qd/ac/MagnetCurrentCtrl (Y2)
sy/ps-qd/ac/Period (Y2)
sy/ps-qd/ac/PeriodCtrl (Y2)
sy/ps-qd/ac/PhaseLag (Y2)
sy/ps-qd/ac/PhaseLagCtrl (Y2)
10
5
0
T0
15:05:01
15:05:02
15:05:03
15:05:04
15:05:05
15:05:06
Auto Update
Update
Dismiss

Pool of polling threads

	By default, a single thread is started to poll attributes. In case of several devices, a pool of threads is available. Astor proposes a graphic tool to distribute device(s) by thread. Use drag and drop to configure the pool.

Server Statistics

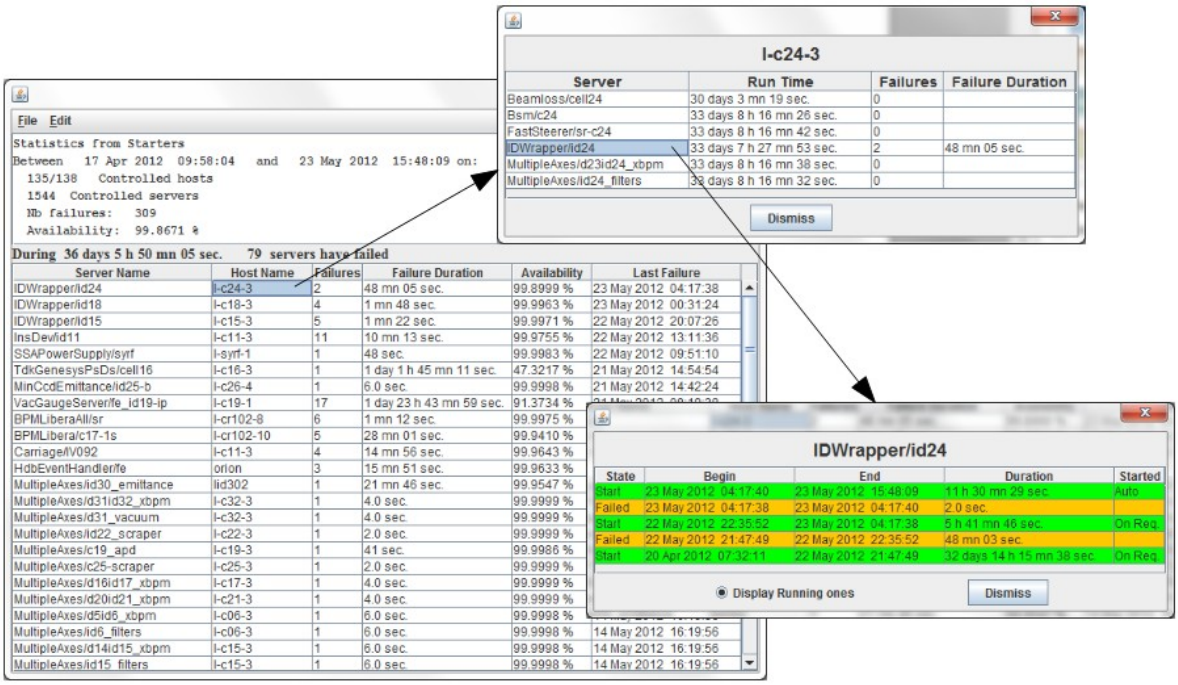
Intended audience: administrators, developers

When a server, started by the Starter, is stopped by another way than a command on the Starter (kill signal, core dump, . . .), this event is logged in a file.

When the server is re started by the Starter, this information is logged too.

Astor proposes to get information from this file for :

- All controlled hosts
- One host.
- Compute statistics and availability for servers.



Reset statistics:

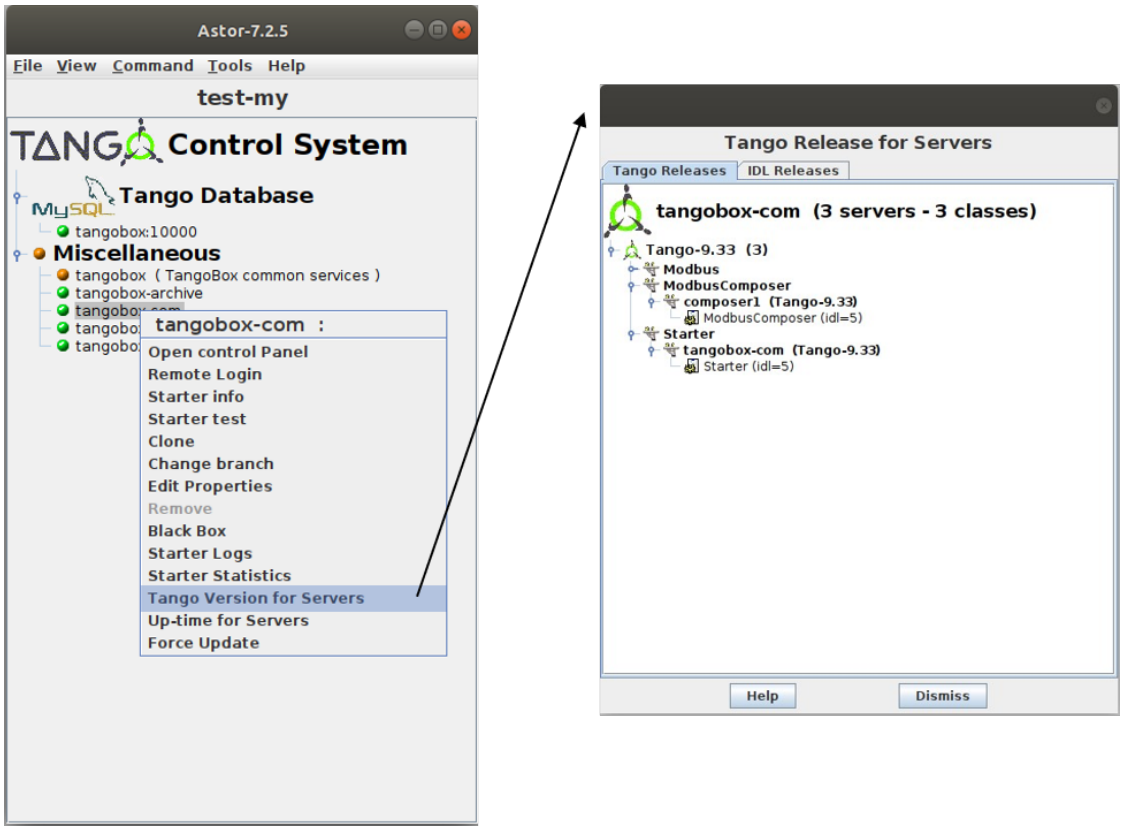
The reset of servers statistics is system specific. It must reserved to a TANGO administrator and it is protected by a password.

Server TANGO versions

Intended audience: administrators, developers

On a host you can have an overview of TANGO version for servers

(See TANGO releases history)



Tango Access Control (TAC) Configuration

Intended audience: administrators, developers

TANGO provides an access control (see *The Tango controlled access system*)

TAC Configuration Tool:

Astor provides a tool to configure it. This tool is reserved to the TANGO administrator and needs a password to be opened.

Configure TAC by IP address and user:

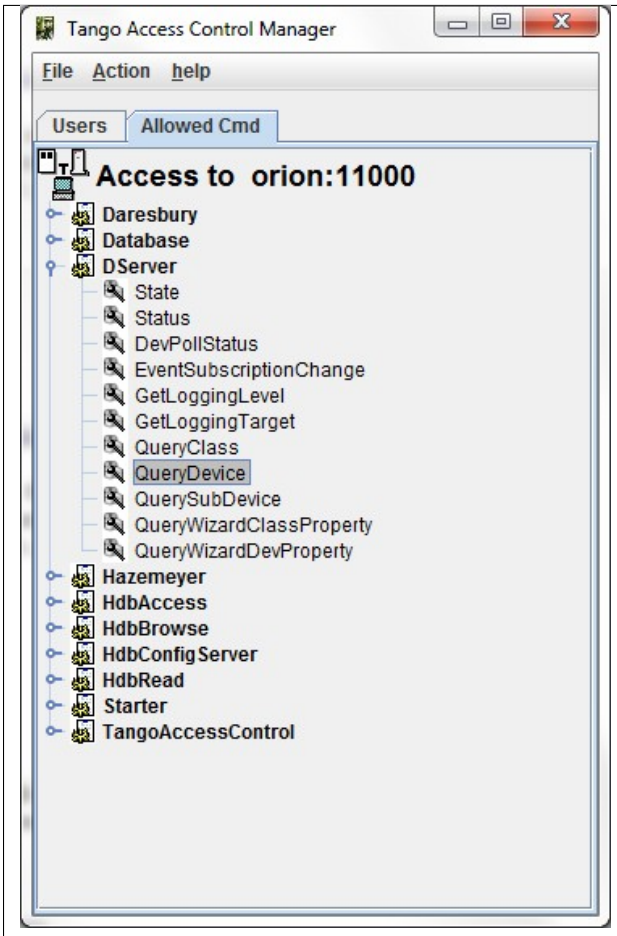
User list is displayed in a tree under groups.

To configure you can Add/Remove a user and its group.

And define this user rights.

	• **All Users:** Have only read access for all devices • **User opid02** Has write access only for devices – fe/id/2 – id/id/2 – sys/hdb-push/* and only for clients running on machines with IP address as xxx.yyy.22.*

Configure TAC for allowed commands:



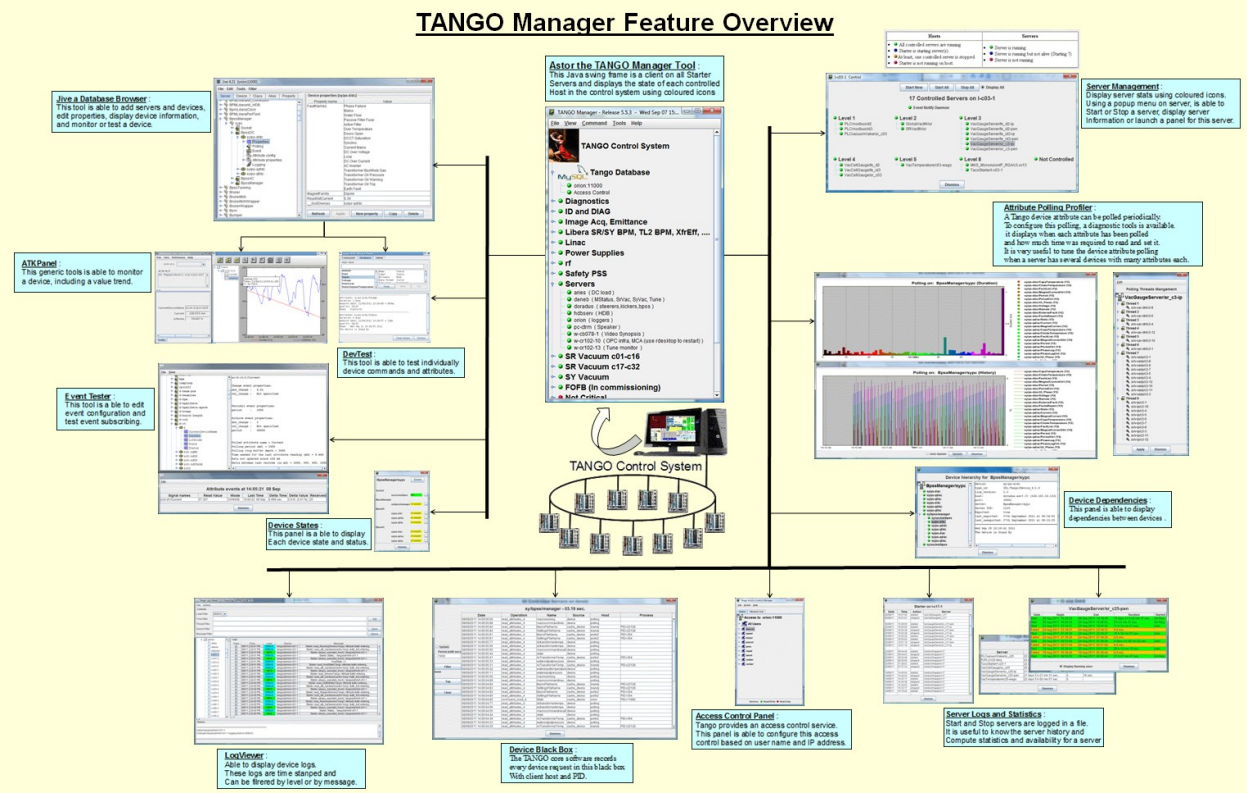
The access control allows **Write** access or **Read Only** access. The **Read Only** access means:

- Read only device attributes.
- Cannot write device attributes.
- Cannot execute commands (a command could be a write action).

Some commands could be a simple read action. For instance on **Dserver** class, the command **QueryDevice** does not write anything and returns the device list. This class is needed to establish a connection between a client and a device, and it is useful even if user is in read only access. This tool provide a list of allowed commands for a specified class. You can add or reove class by a right click menu.

Features overview

Intended audience: administrators, developers, users



ICALEPCS 2011 paper

Intended audience: administrators, developers, users

You may be also interested in an `ICALEPCS paper` about **`Astor`** and **`Starter`**.

7.2.2 Jive

Intended audience: administrators, developers, users

Welcome to Jive documentation!

Jive is a standalone JAVA application designed to browse and edit the static **TANGO database**. Jive has been written using **Swing** and need a JVM higher than 1.6.0. You will find in this documents the way to manage and create devices, properties and classes. Jive also offers advanced search/selection features.

Download the project here

Find this project on GitHub

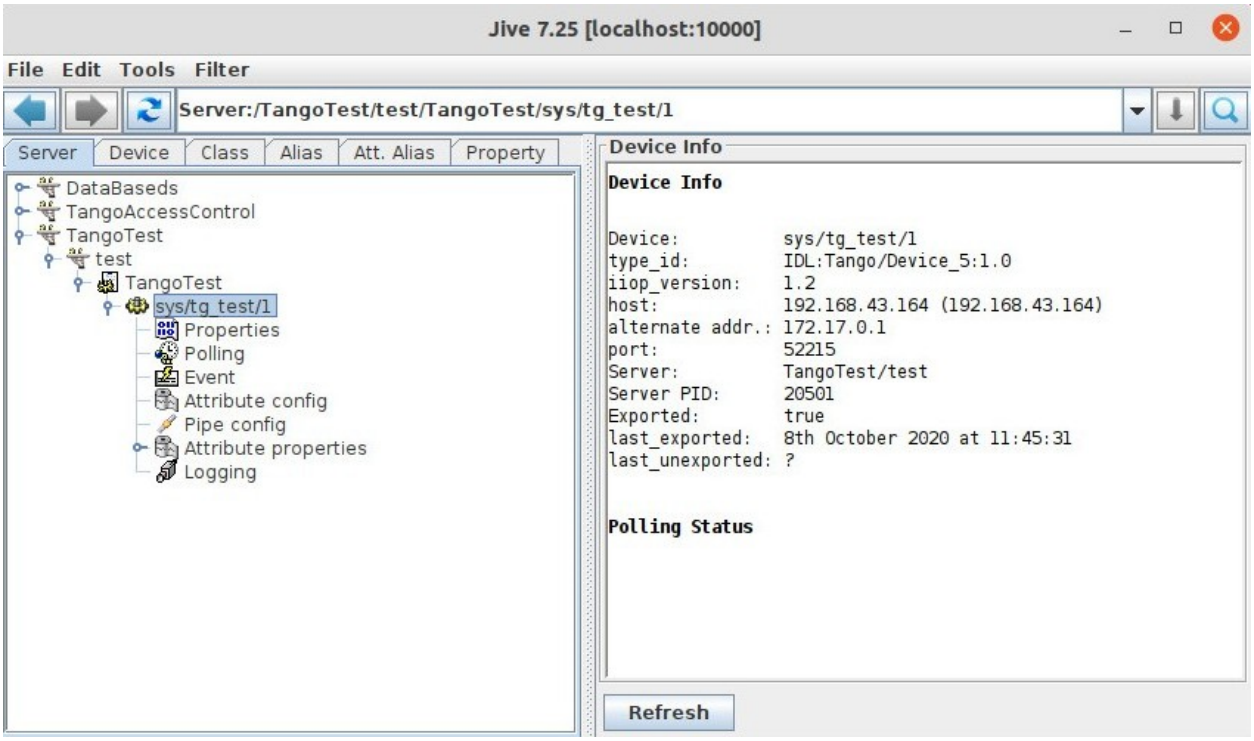


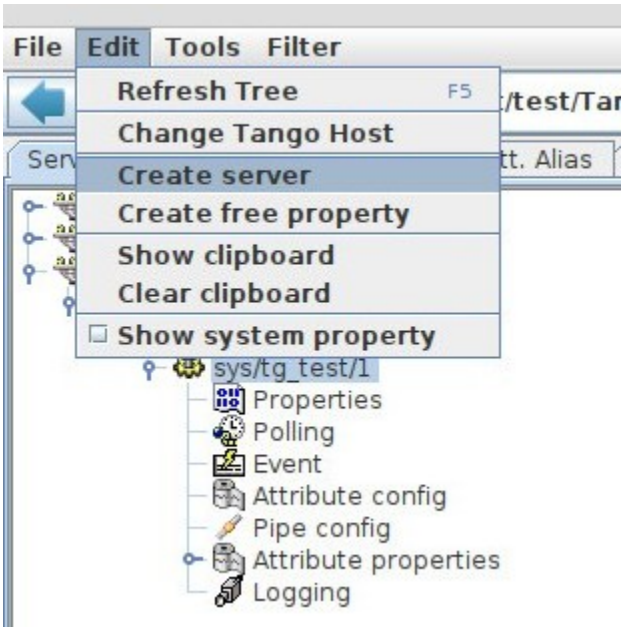
Fig. 1: **Jive: A Tango Database browser**

Contents

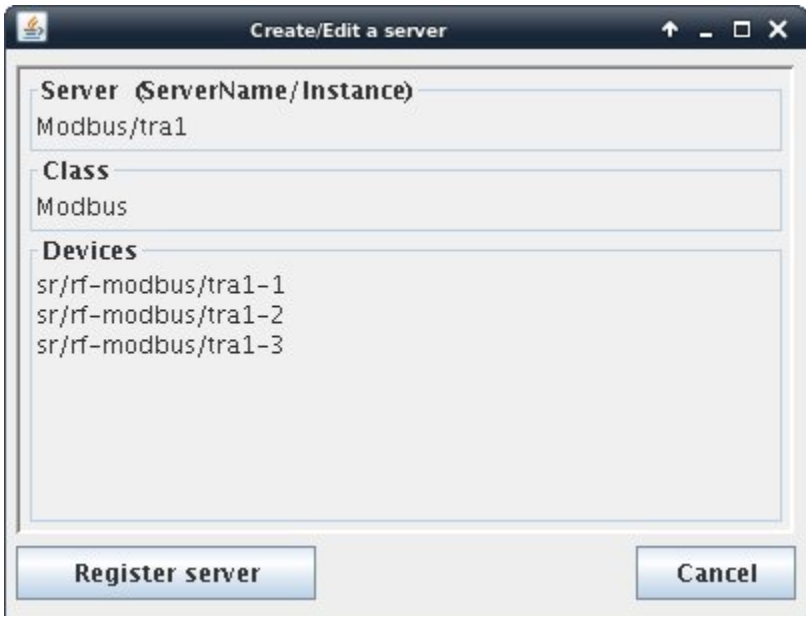
Manage Servers

Intended audience: developers, administrators

Creating a Tango server

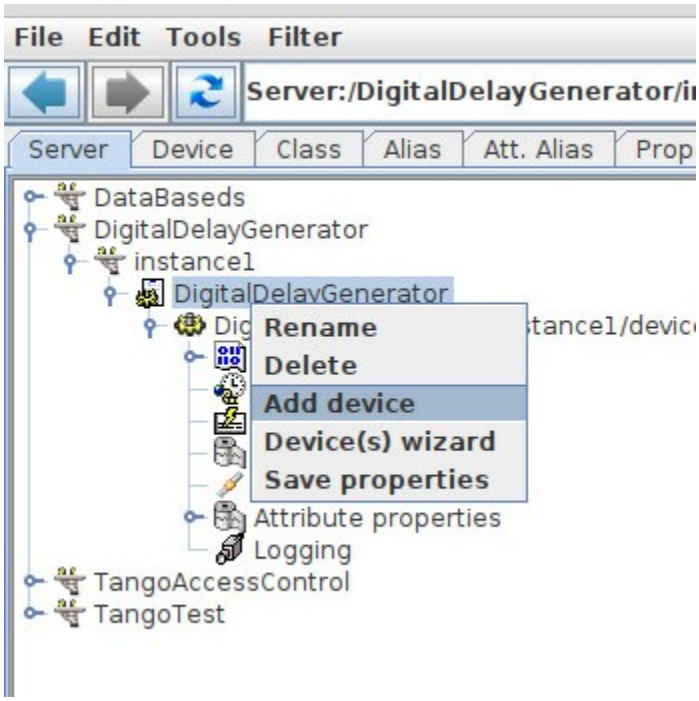


To Create a server , Open the server creation dialog within the **Edit** menu.



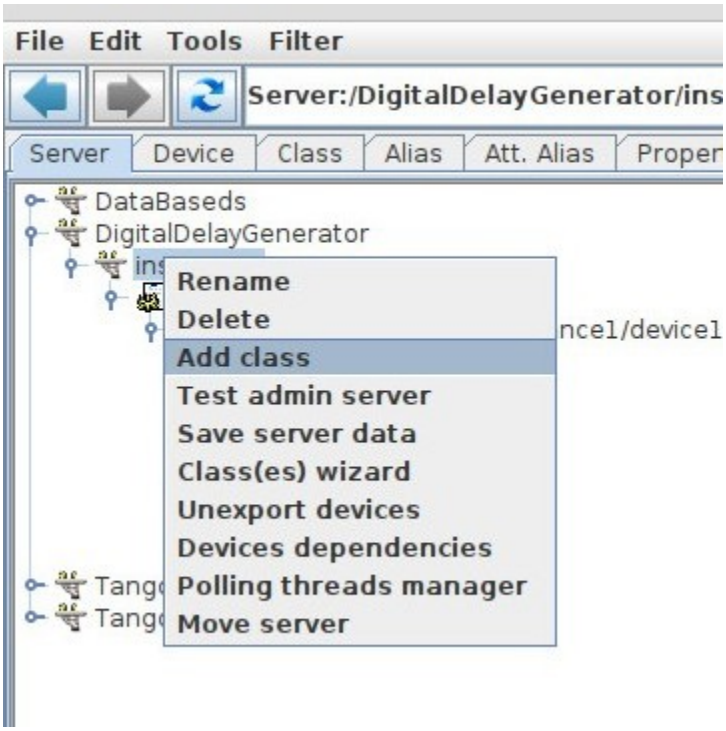
- Enter the server name including its instance name: Ex: Modbus/tra1 where Modbus is the name of the process (the executable name) and tra1 its instance name.Then you can run your server by launching “Modbus tra1”.
- Enter the class name
- Enter all devices of this class
- Click on Register server

Adding or Removing device to/from an existing server



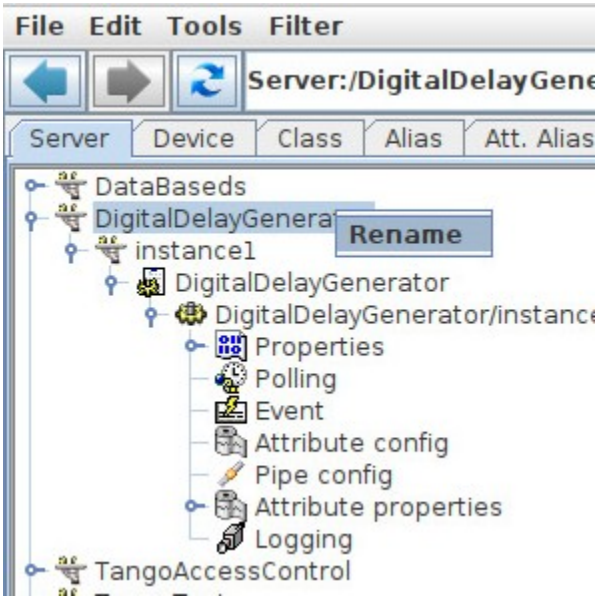
Select the class in the server you want to add a device, then right click on it and select "Add a device". This will ask you for a device name. Select "Delete" if you want to remove the device. Note that when you remove a device all its properties are also removed.

Adding or Removing class to/from an existing server

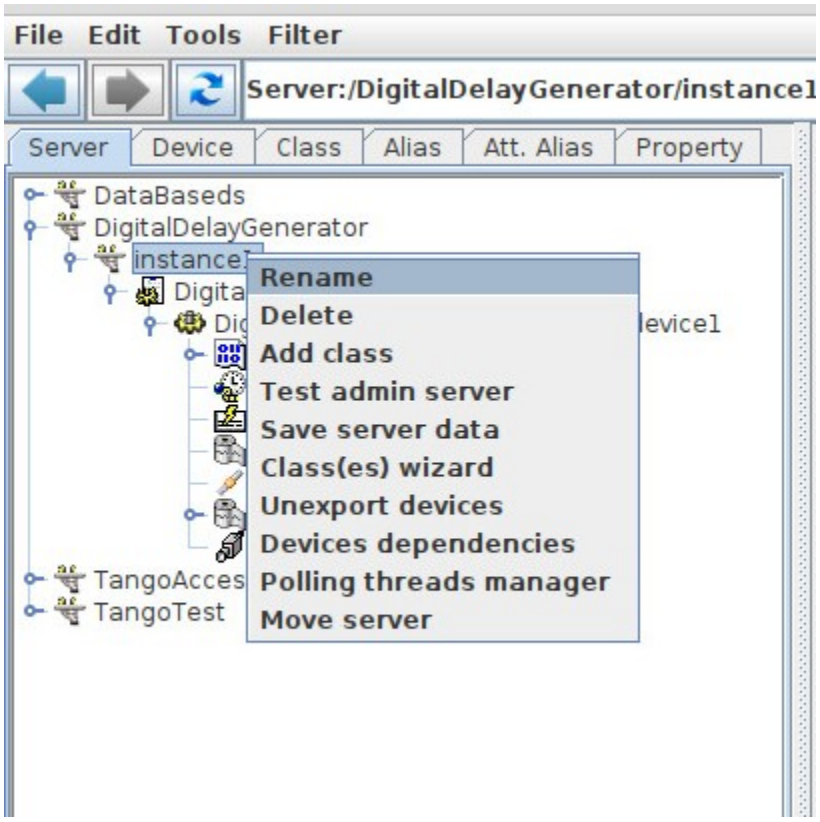


Select the server you want to add a class, then right click on it and select "Add class". This will show you the Create/Edit server window with the server name locked. Enter the class and devices name and click "Register Server".

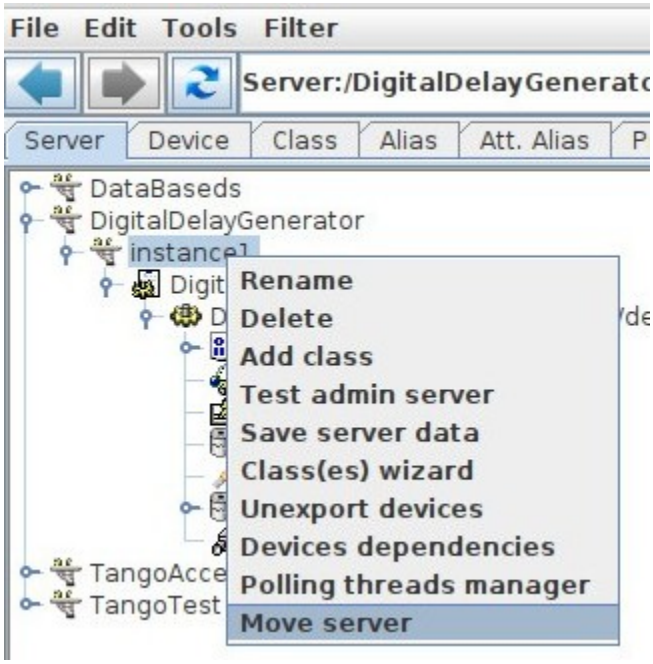
Renaming a server



This is a global rename of a server (executable name), all instances are affected.



This allows to rename the instance of one specific server.

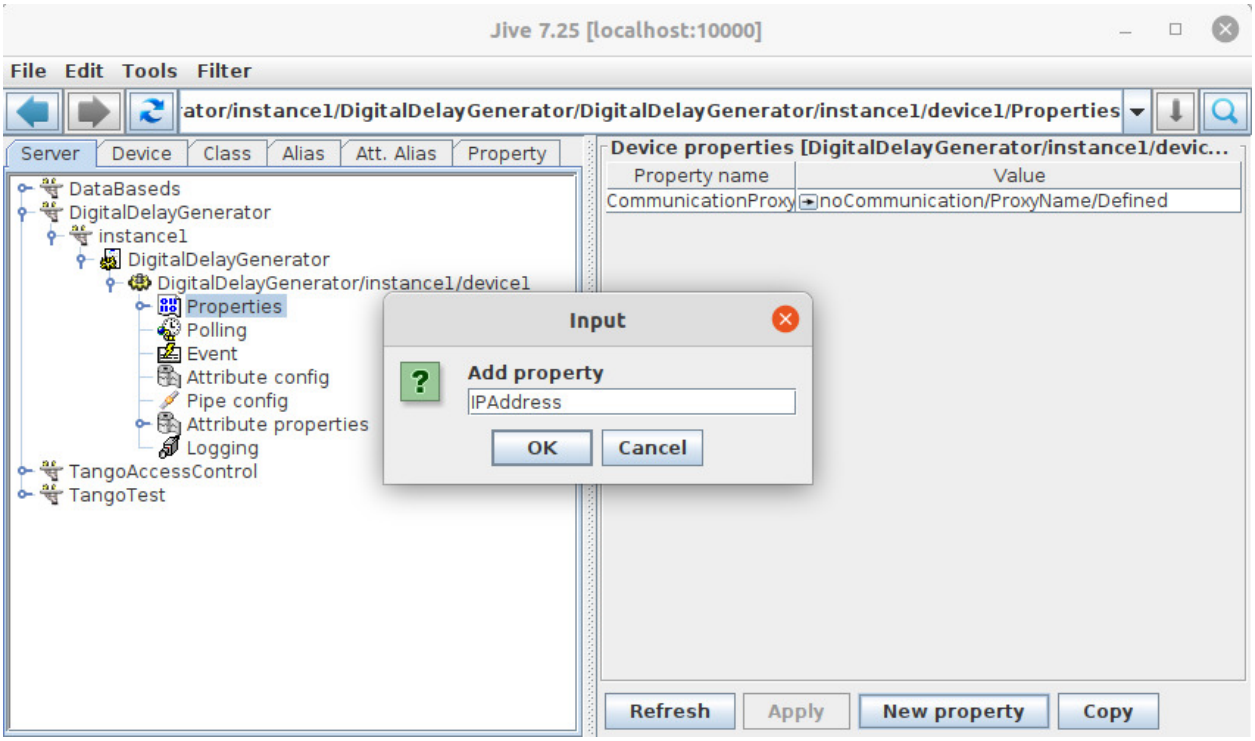


This allows to rename the server name and instance name of one specific server.

Manage Properties

Intended audience: administrators, developers

Creating a Property



Go in the device tree and select the Properties node, then click on the "New property" button. This will prompt an input dialog where you can enter the property name.

Tip:

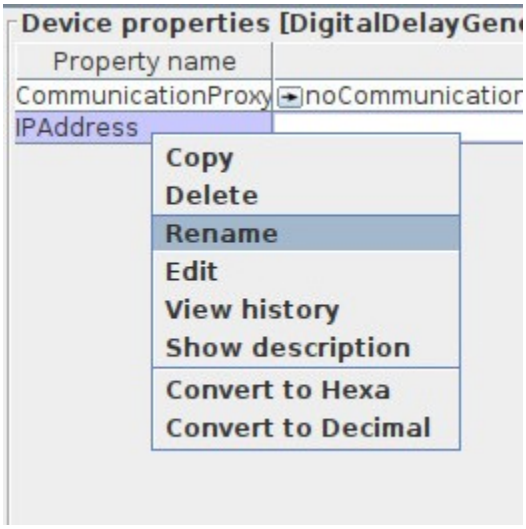
Since cppTango 9.3.3, you can use the keywords *NaN*, *-NaN*, *inf*, *-inf* and *+inf* when writing double or float scalar and array properties from jive to represent *Not A Number* and *infinite* values.

This might not work (yet) for a Java device server.

It is not case sensitive, so *nan*, *-NAN*, *nAn*, *iNF*, *-Inf* and *+INF* are also supported for instance.

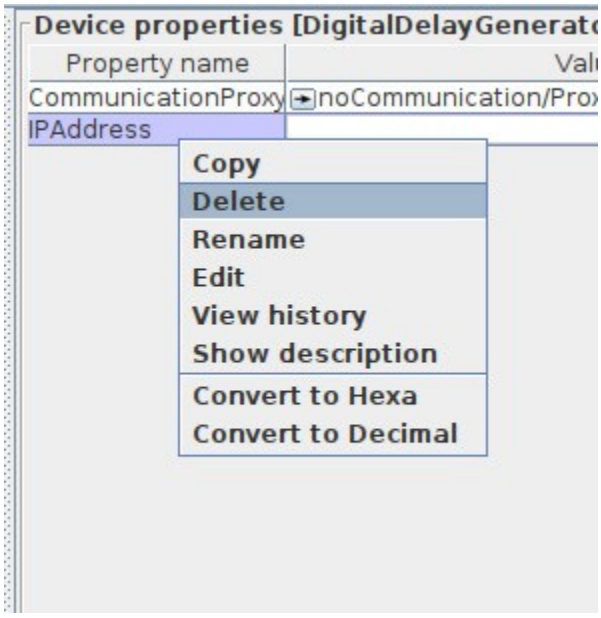
A negative *NaN* is extracted as *NaN*.

Rename a Property



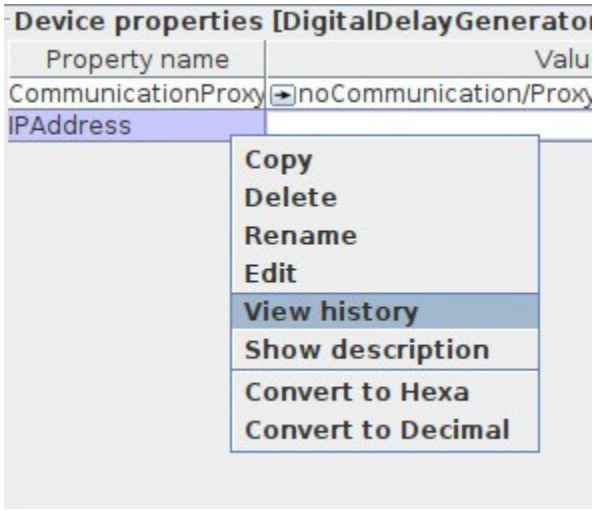
Rigth click on the property name in the property table and select “Rename”, This will allow you to give a new name to your property.

Delete a Property



You can select “Delete” (right click on the property item) to remove one propery. You can also right click on the device node and select “Delete”. This will erase all properties (including attribute properties) for the device. Note that if you want to remove the device itself, you have to remove it from its class in the server definition.

History of a Property



You can access to the change history of a property by selecting “View History”. This opens the history dialog.

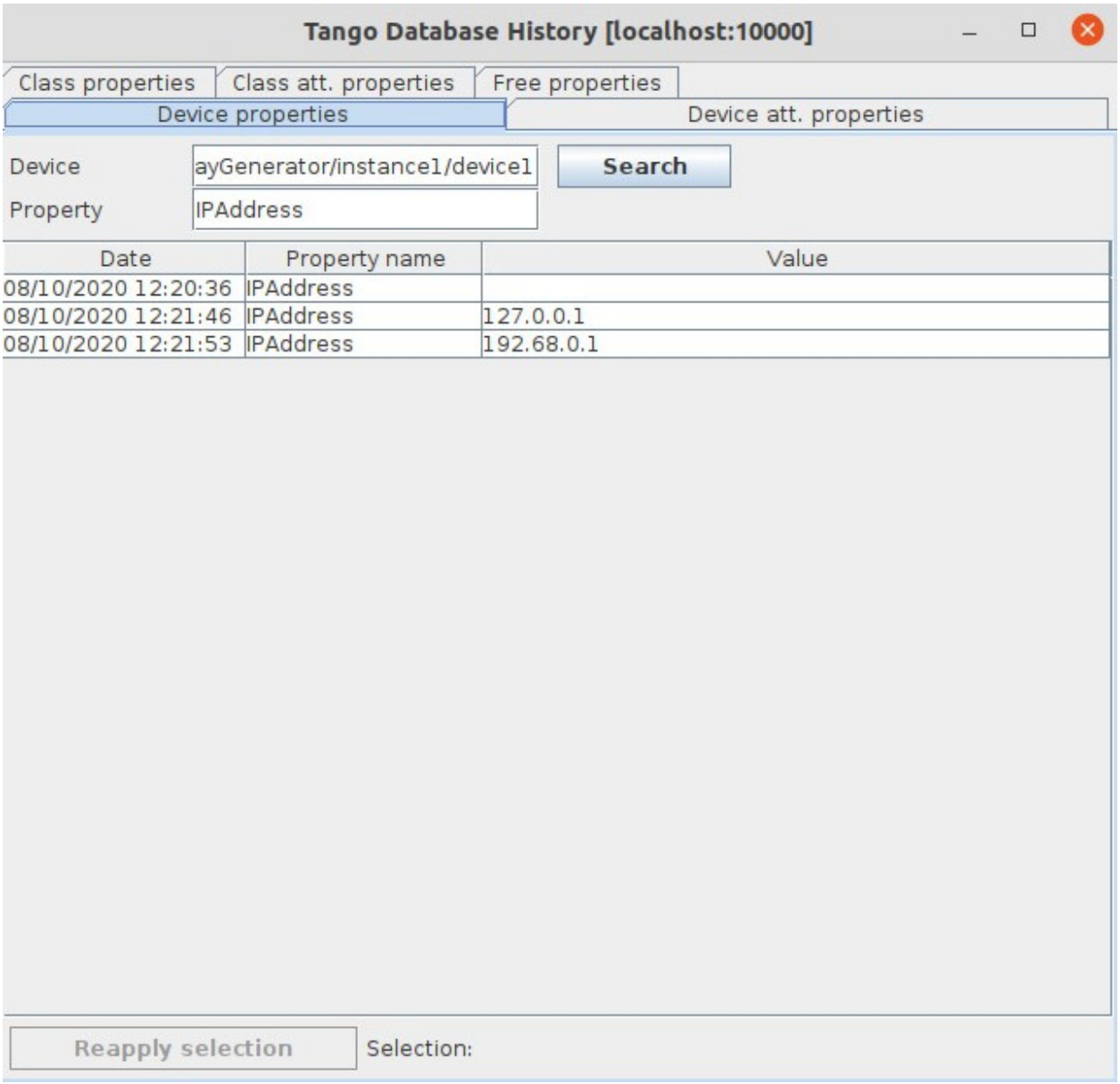
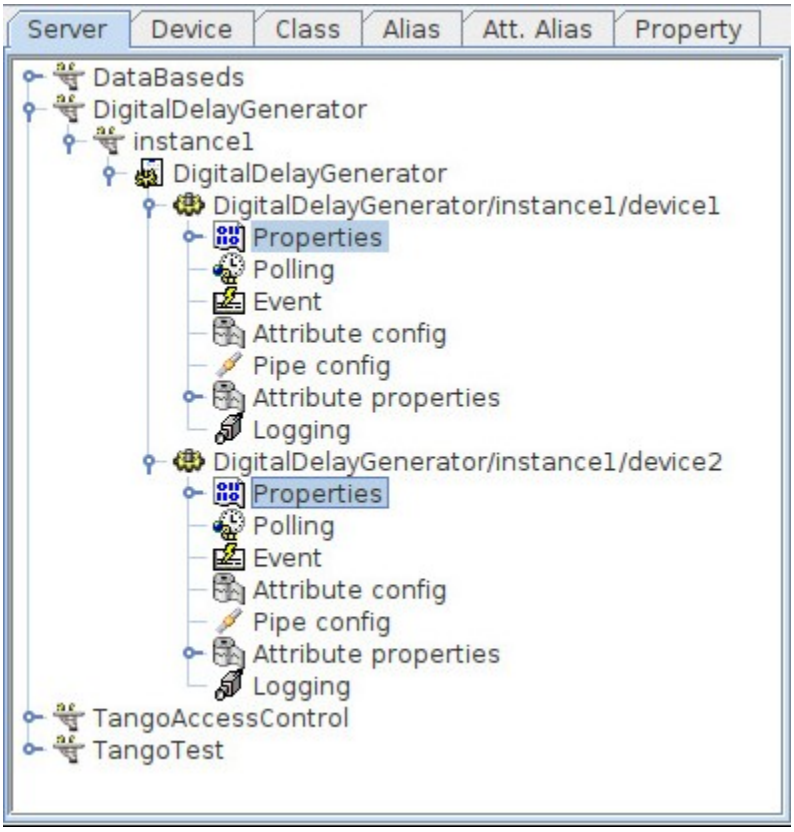


Fig. 2: **Tango Database History dialog**

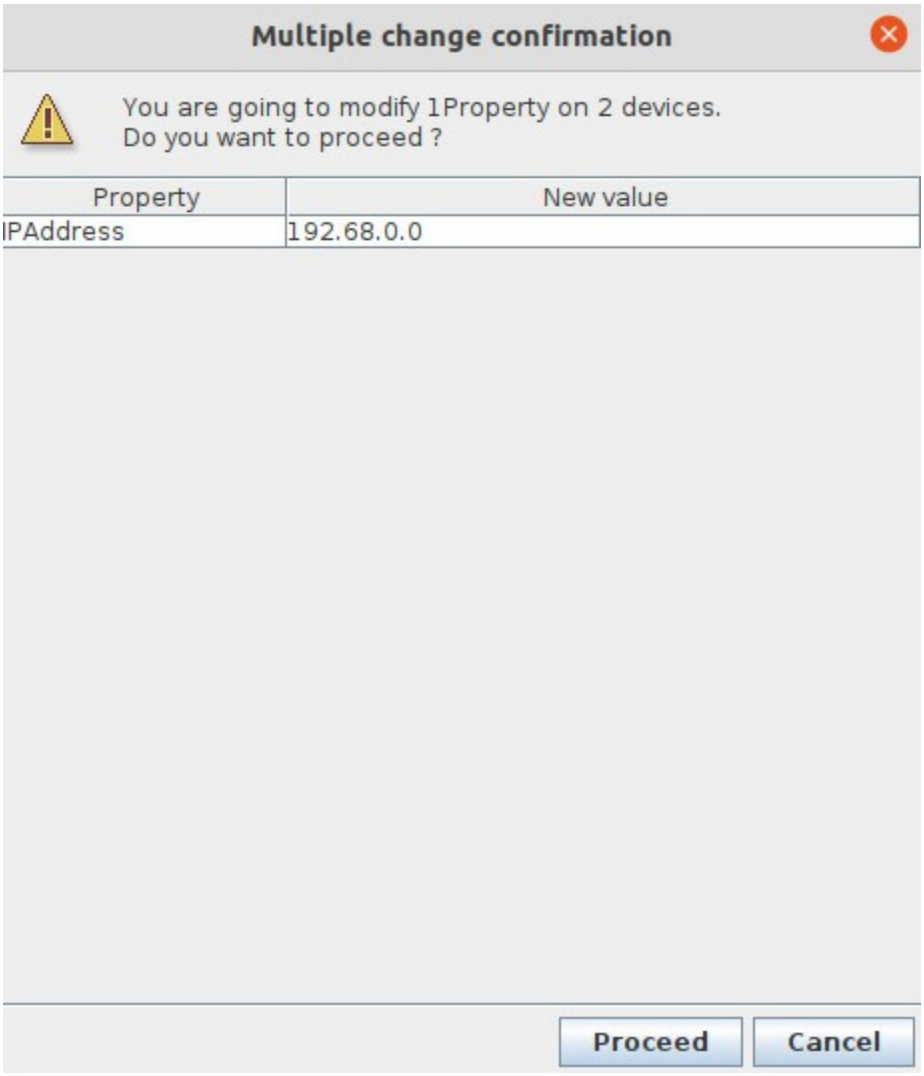
Advanced Editing

Intended audience: administrators, developers

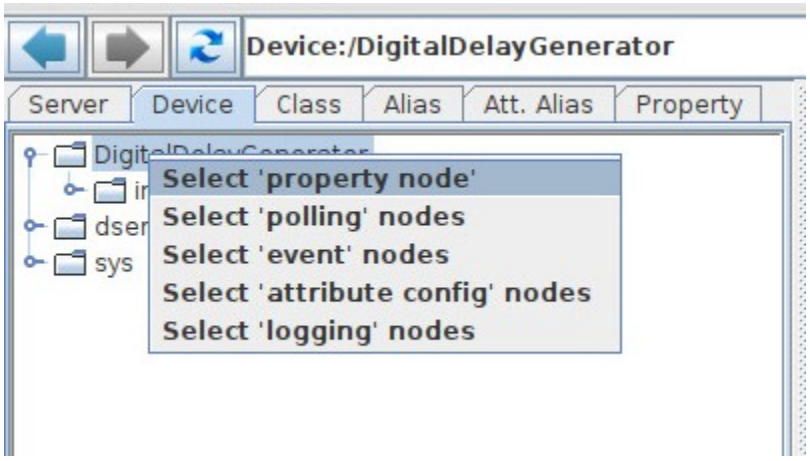
Multiple Editing



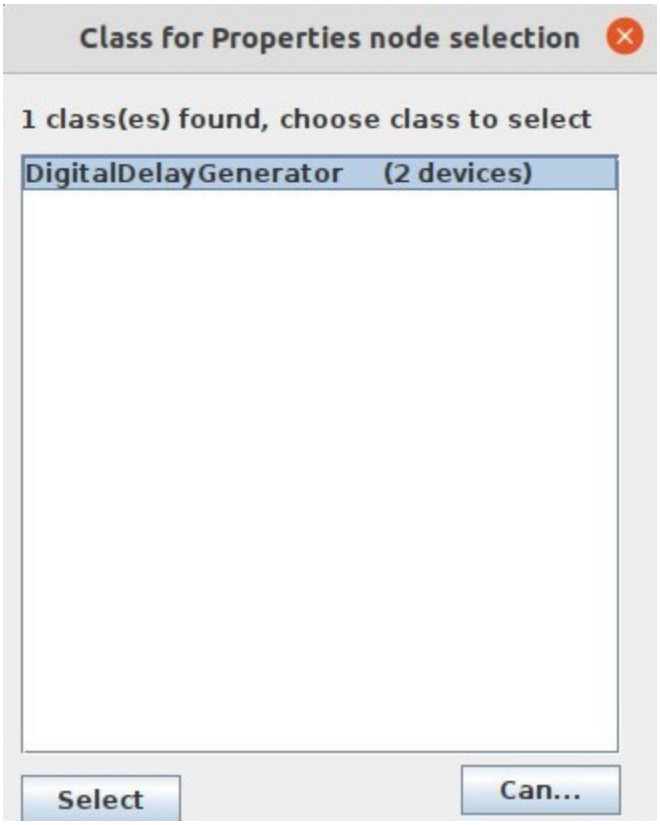
You can select several property nodes by holding the CTRL key down. Then if you edit one or more properties, all properties will be affected in all selected nodes. If several nodes are selected a confirmation dialog will appear as shown below.



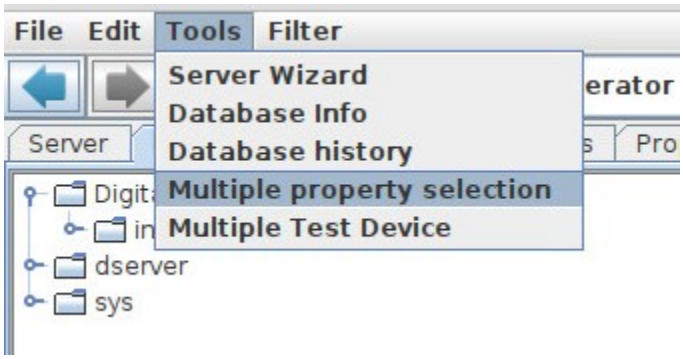
Multiple Selection



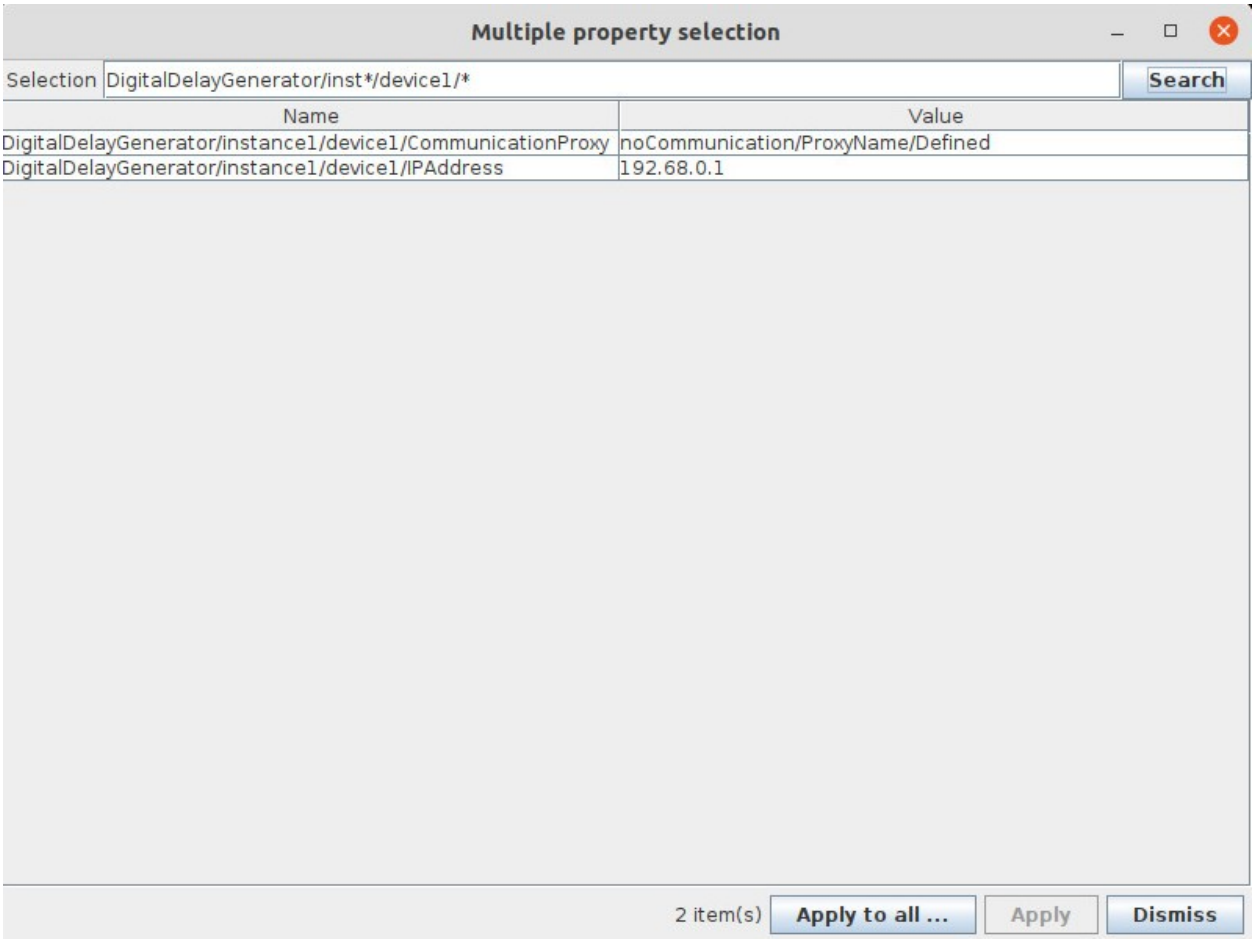
You can by rigth clicking on a domain or family node select a type of node to select. Then a dialog will ask you which class you want to select. All classes present in the tree under the selection root will be selectable as shown below.



Multiple Selection in a table



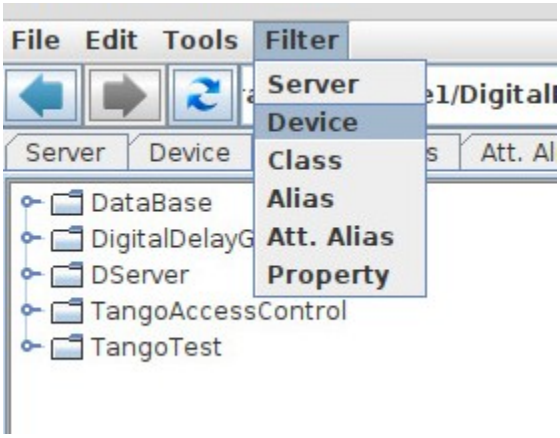
You can select multiple properties by using a filter. Select “Multiple selection” from the tools menu to display the multiple selection table.



- Enter a selection filter using wildcard '*' to get the list of desired properties.
- You can edit a single propery in the talbe or Apply a global value to all the selection by clicking on the "Apply to all" button. "Apply to all" will first ask for the value to be applied.

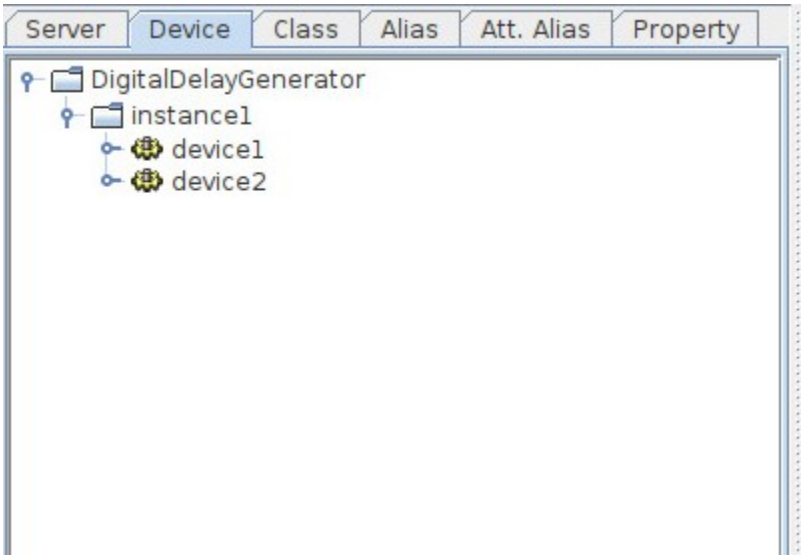
Filtering the nodes in trees

Intended audience: administrators, developers, users



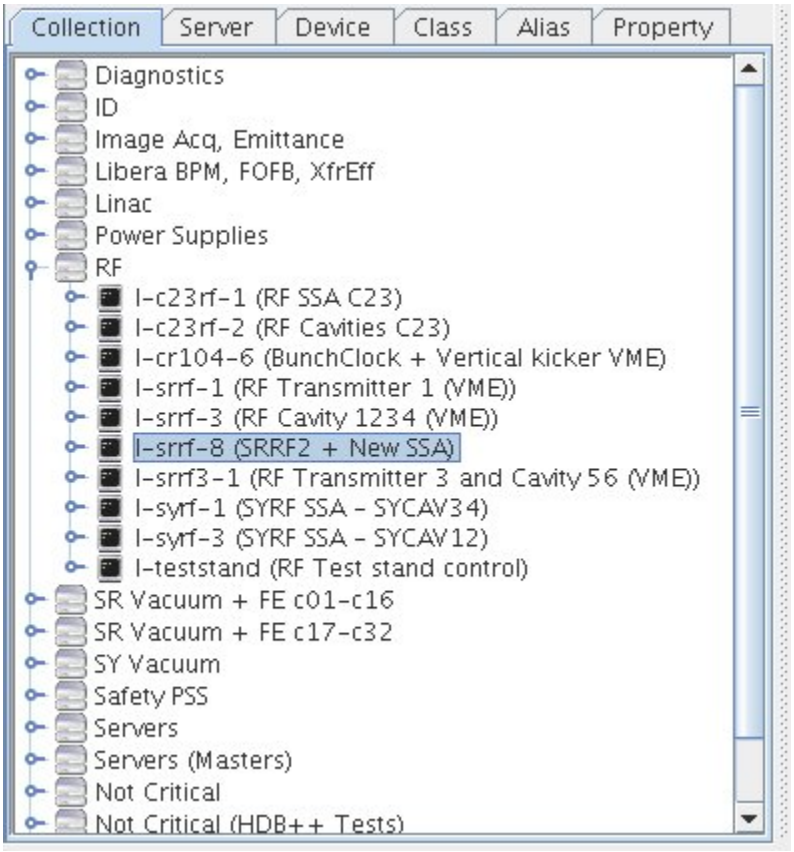
It is possible to filter some nodes in selection trees. Select which tree you want to be filtered then enter a filter (using wildcard) in the input dialog.

Then the device tree will be displayed as below:



Displaying collection

Intended audience: administrators, developers



It is possible to display the servers tree organized as host collection (Astor view). To enable this, you can use the -p option (see command line options). To configure host collection, you have to follow the Astor configuration process and to configure Starter properties.

Command line

Intended audience: administrators, developers

Dependencies (CLASSPATH)

```
ASTOR=$TANGO_JAVA_APPLIS/Astor.jar
JSSHTERMINAL=$TANGO_JAVA_APPLIS/JSSHTerminal.jar
LOGVIEWER=$TANGO_JAVA_APPLIS/LogViewer.jar
LOG4J=$TANGO_JAVA_LIBS/log4j.jar
TANGO=$TANGO_JAVA_LIBS/JTango.jar
TANGOATK=$TANGO_JAVA_LIBS/ATKCore.jar:$TANGO_JAVA_LIBS/ATKWidget.jar
ATKPANEL=$TANGO_JAVA_APPLIS/atkpanel.jar
JIVE=$TANGO_JAVA_APPLIS/Jive.jar

CLASSPATH=$TANGO:$TACO:$TANGOATK:$ATKPANEL:$JIVE:$LOGVIEWER:$LOG4J:$ASTOR
```

Note: Astor.jar is needed for “Polling thread manager” and “Device dependencies” features.

Command line

```
java -DTANGO_HOST=host:port jive3.MainPanel
```

Command line options

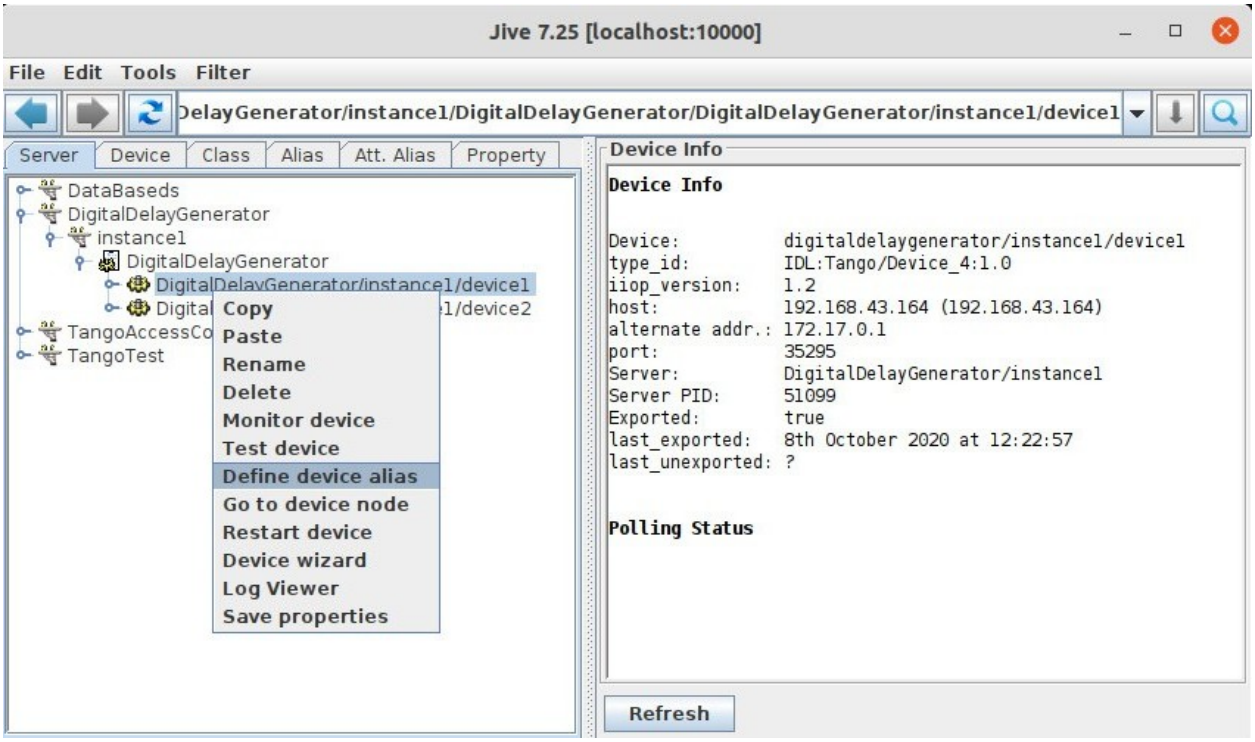
```
Usage: jive [-r] [-s server] [-d device] [-fxx filter] [-p panel]
 -r        Read only mode (No write access to database allowed)
 -s server Open jive and show specified server node (server=ServerName/instance)
 -d device Open jive and show specified device node (device=domain/family/member)
 -fs filter Default server filter
 -fd filter Default device filter
 -fc filter Default class filter
 -fa filter Default alias filter
 -faa filter Default attribute alias filter
 -fp filter Default property filter
 -p panelmask (1=Collection 2=Server 4=Device 8=Class 16=DevAlias 32=AttAlias␣
↪64=FreeProperty)
```

Device Alias

Intended audience: users

How to create a device alias

A device alias can be created from jive by right-clicking on a device node and by selecting “Define device alias” right-click menu item.



7.2.3 ATKPanel

Intended audience: all

Overview

AtkPanel is a generic control panel application. It can be used to control any Tango device. It supports most of Tango features and data types. Using AtkPanel you can view and set attribute values, execute commands, test the device and access diagnostic data.

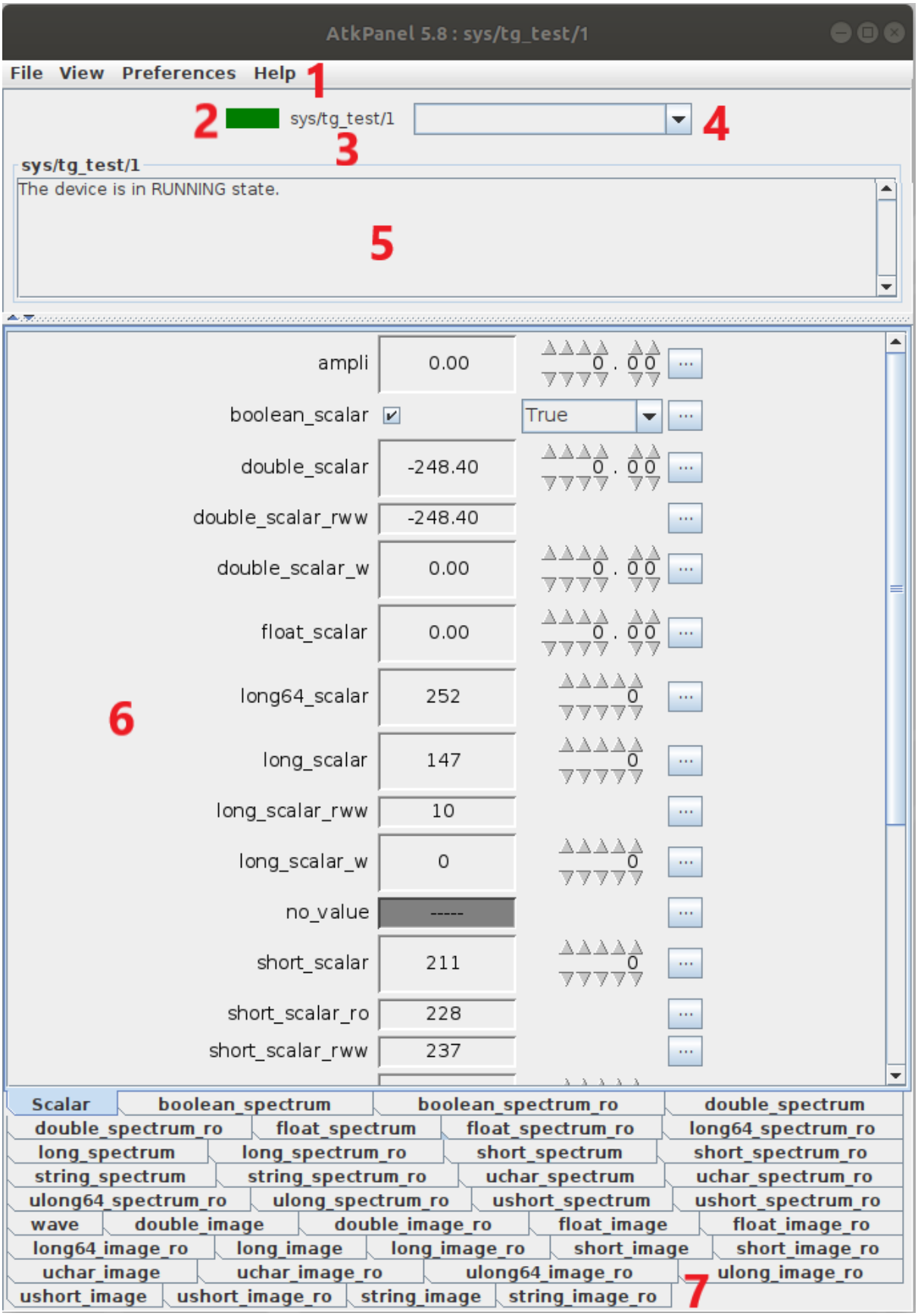


Fig. 3: AtkPanel 5.5 main window

1. Menu bar
2. Device state
3. Device name
4. Commands drop-down
5. Device status
6. Device attributes
7. Attribute switcher

By default AtkPanel opens the scalar attribute view. If your device has attributes of non-scalar types, they will appear on the attribute switcher on the bottom of the window.

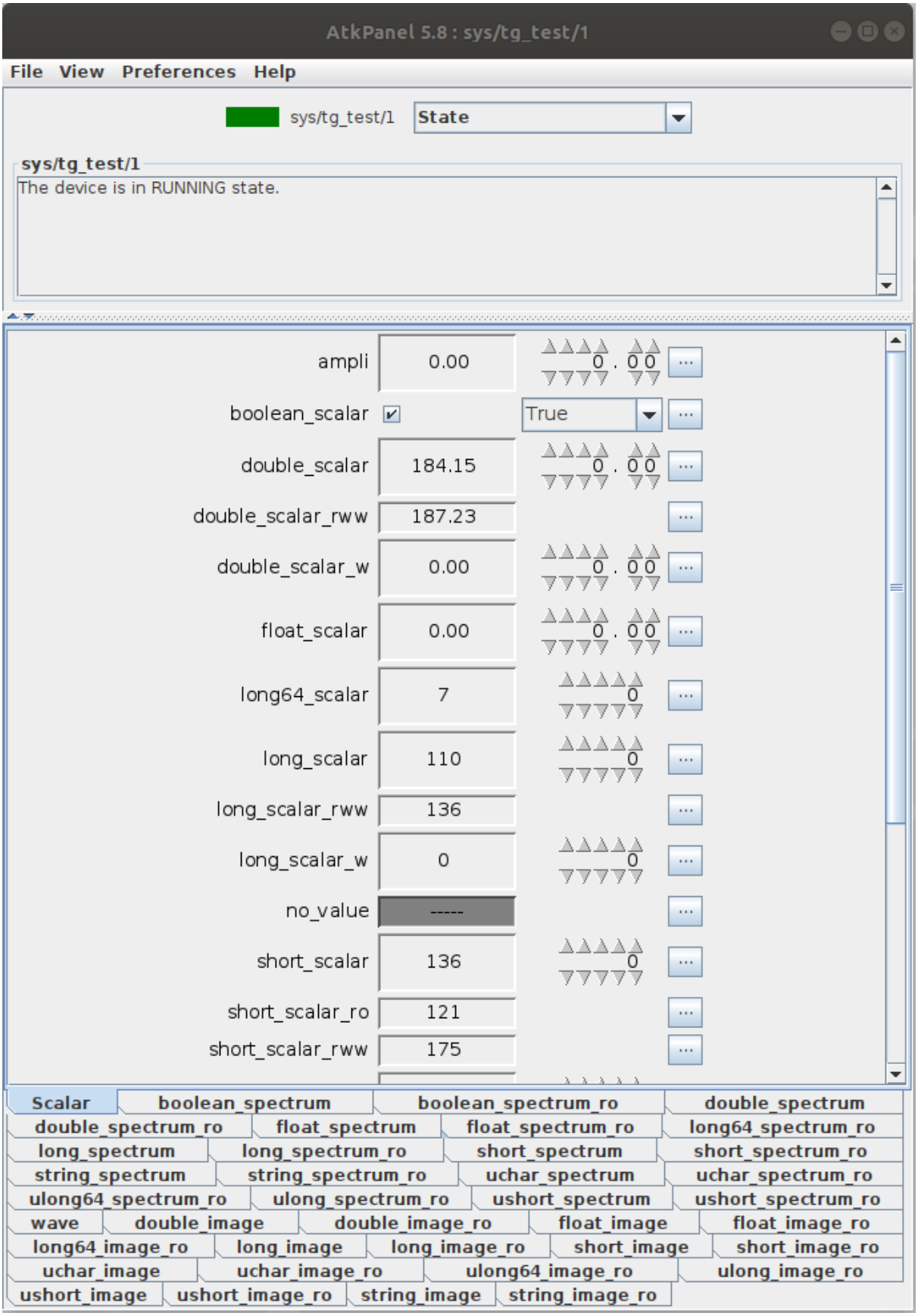


Fig. 4: Scalar attributes view

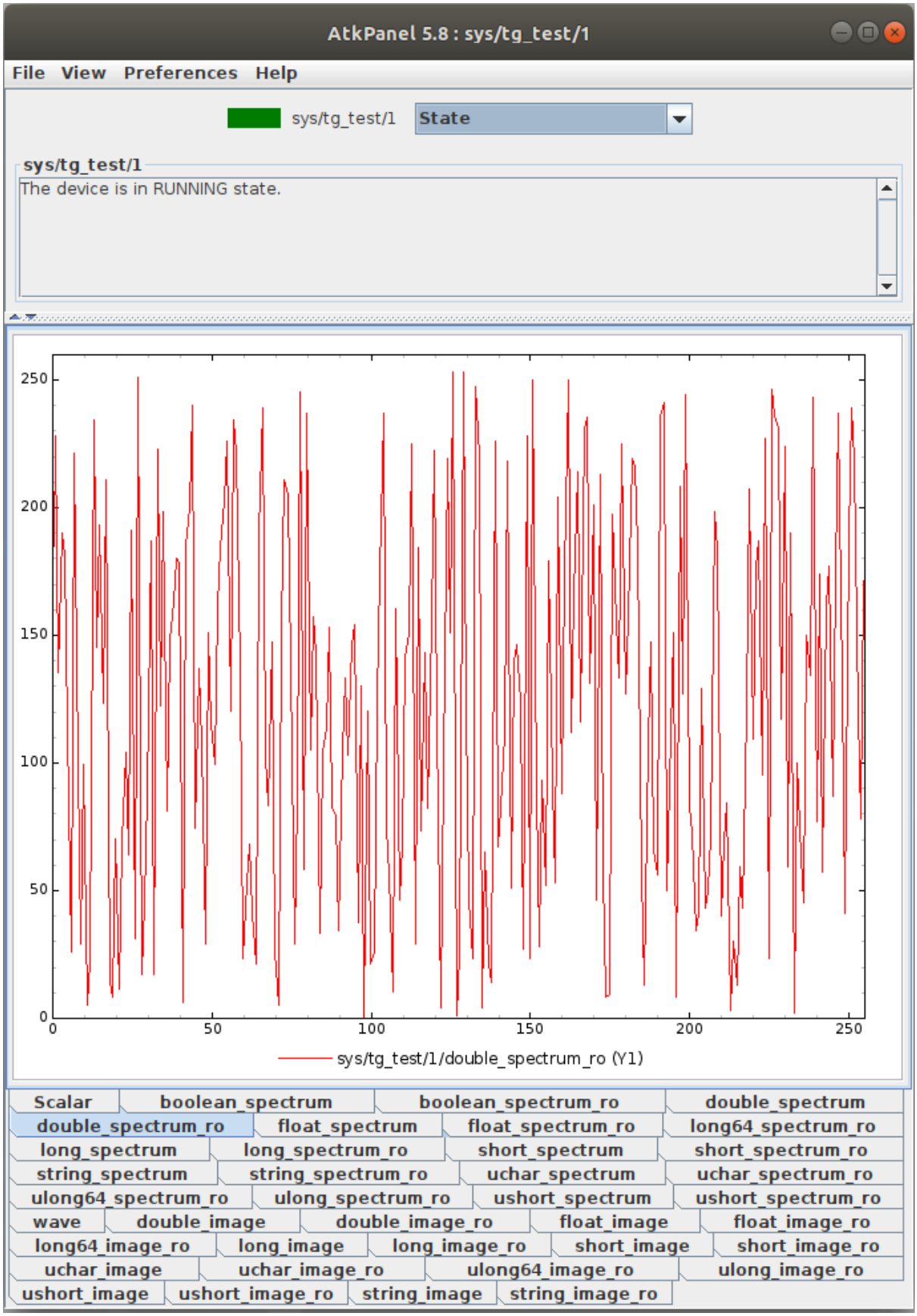


Fig. 5: Double array attribute view

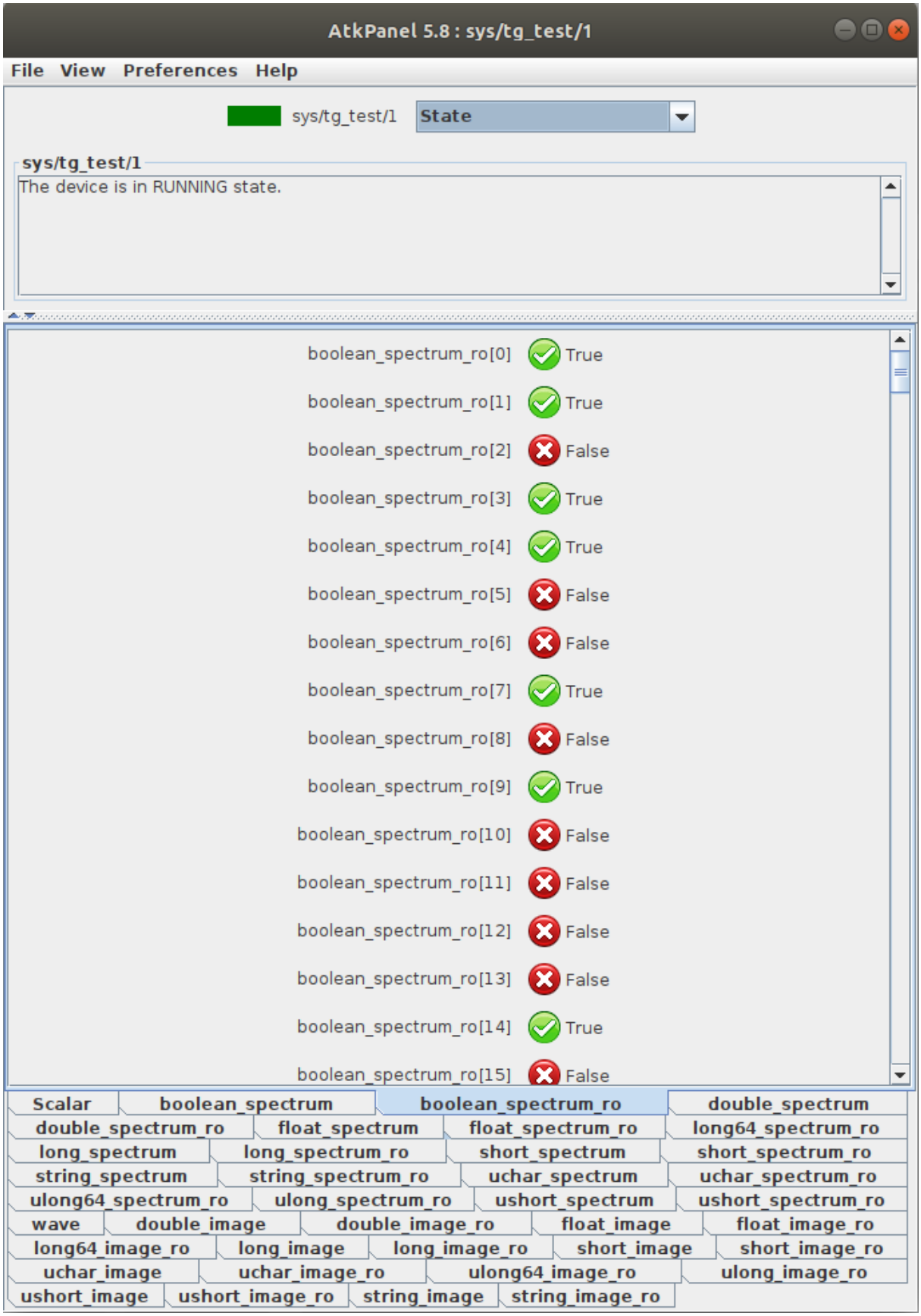


Fig. 6: Boolean array attribute view

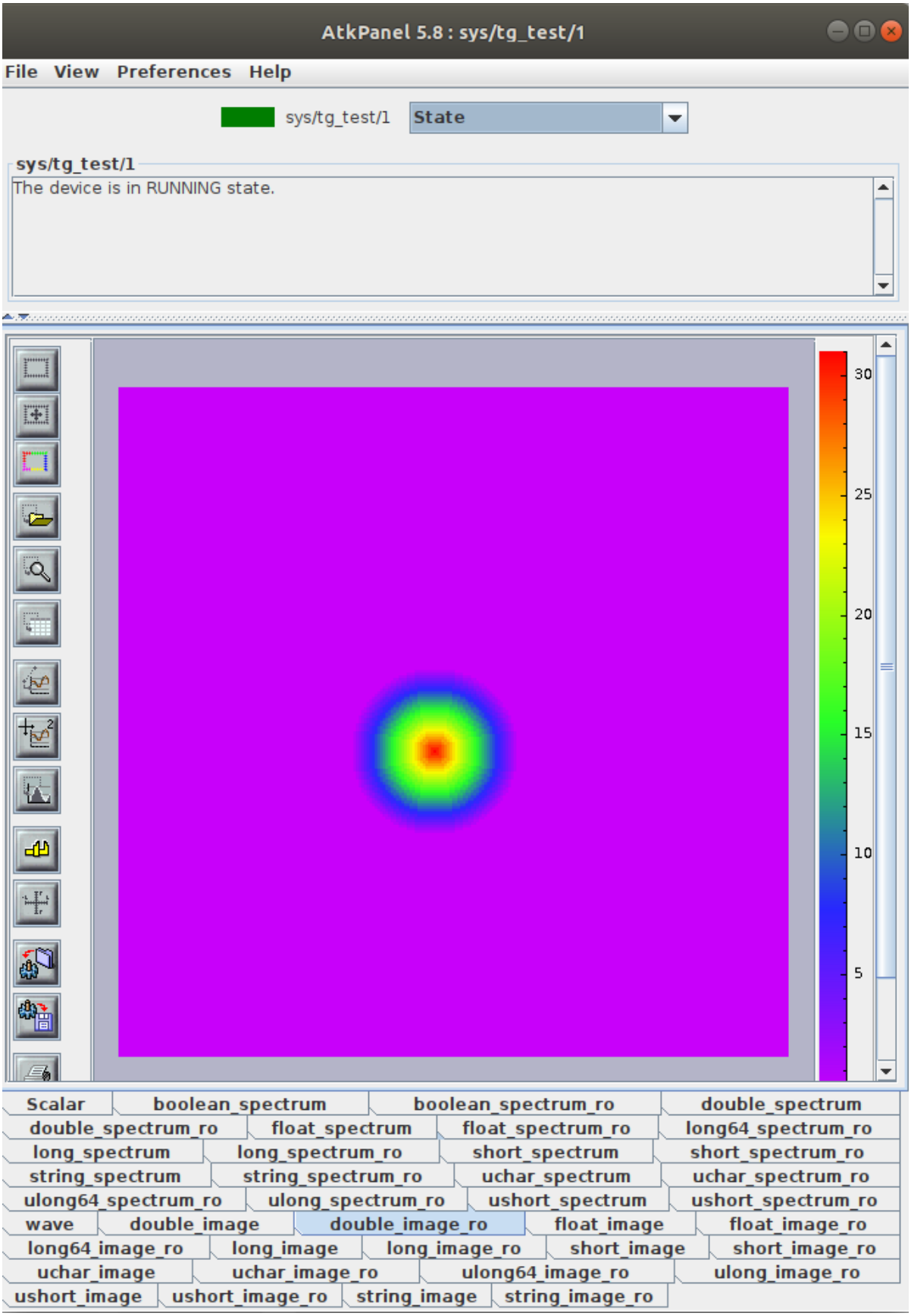


Fig. 7: Image attribute view

You can run a command by selecting the command name from the command drop-down list.

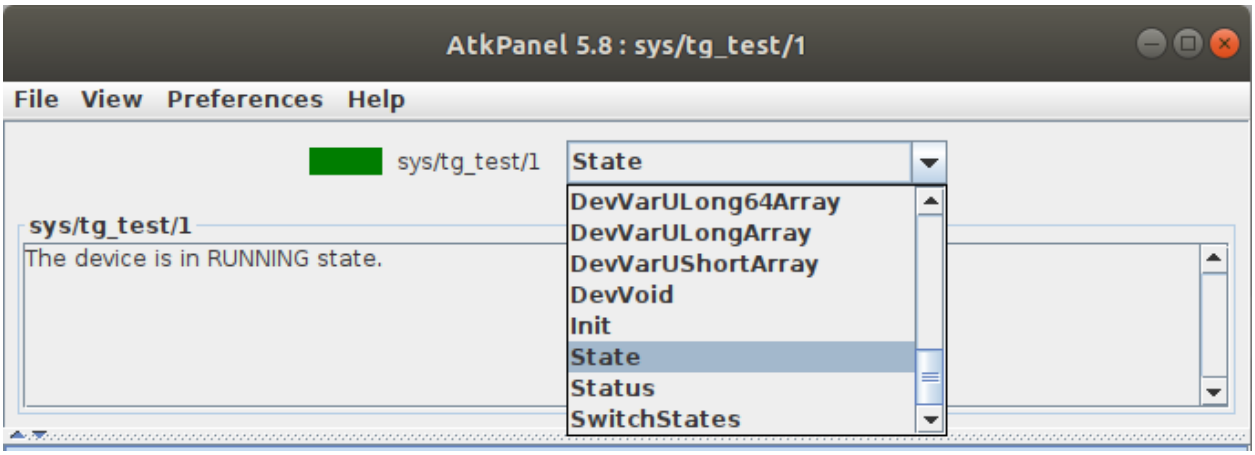


Fig. 8: Command drop-down list

Device testing

By selecting *View* > *Test device* from the menu you can open the device testing panel. In this panel you can check how the device is reponding to different commands and attribute values and how much time the requests take.

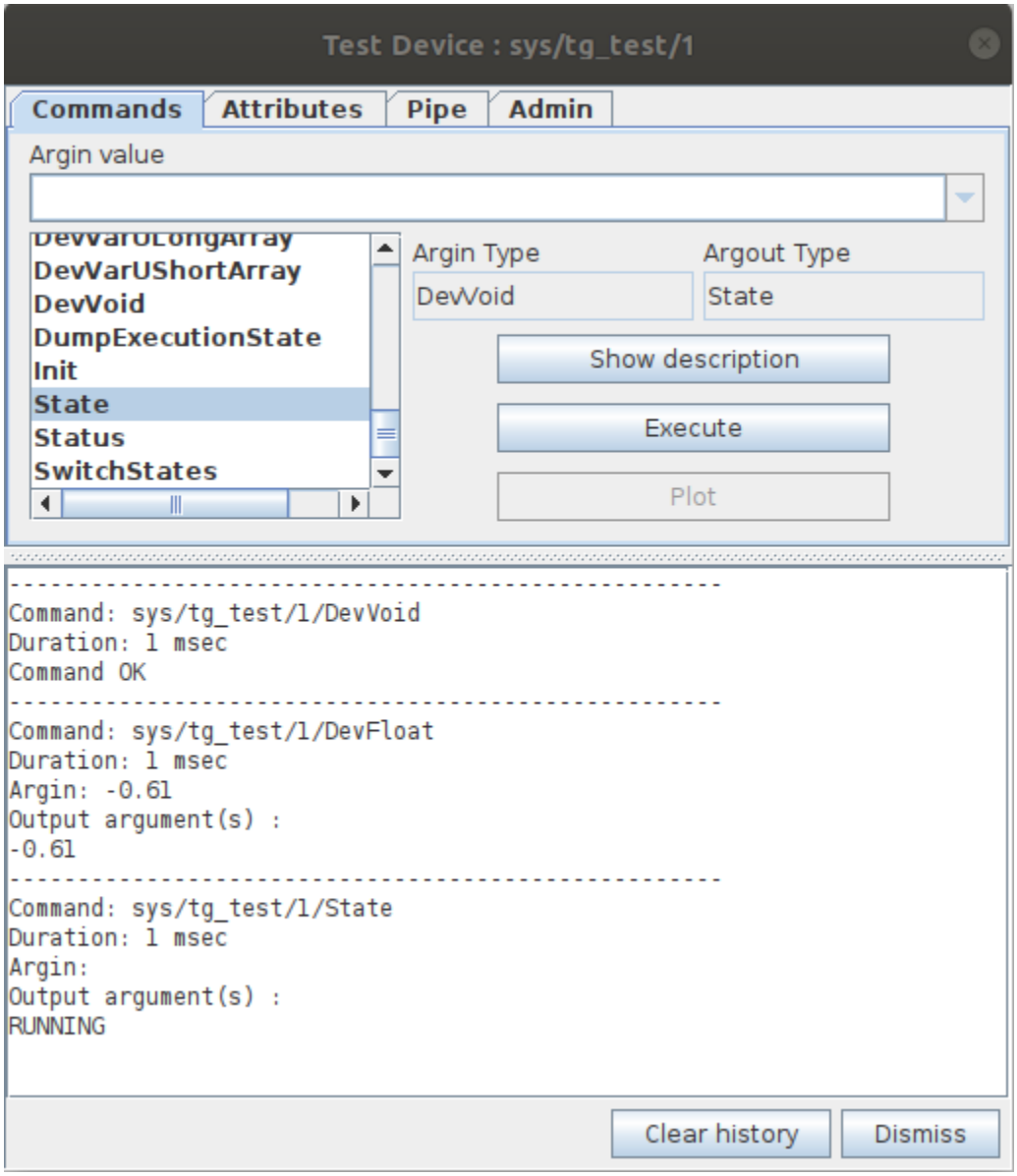


Fig. 9: Test commands view

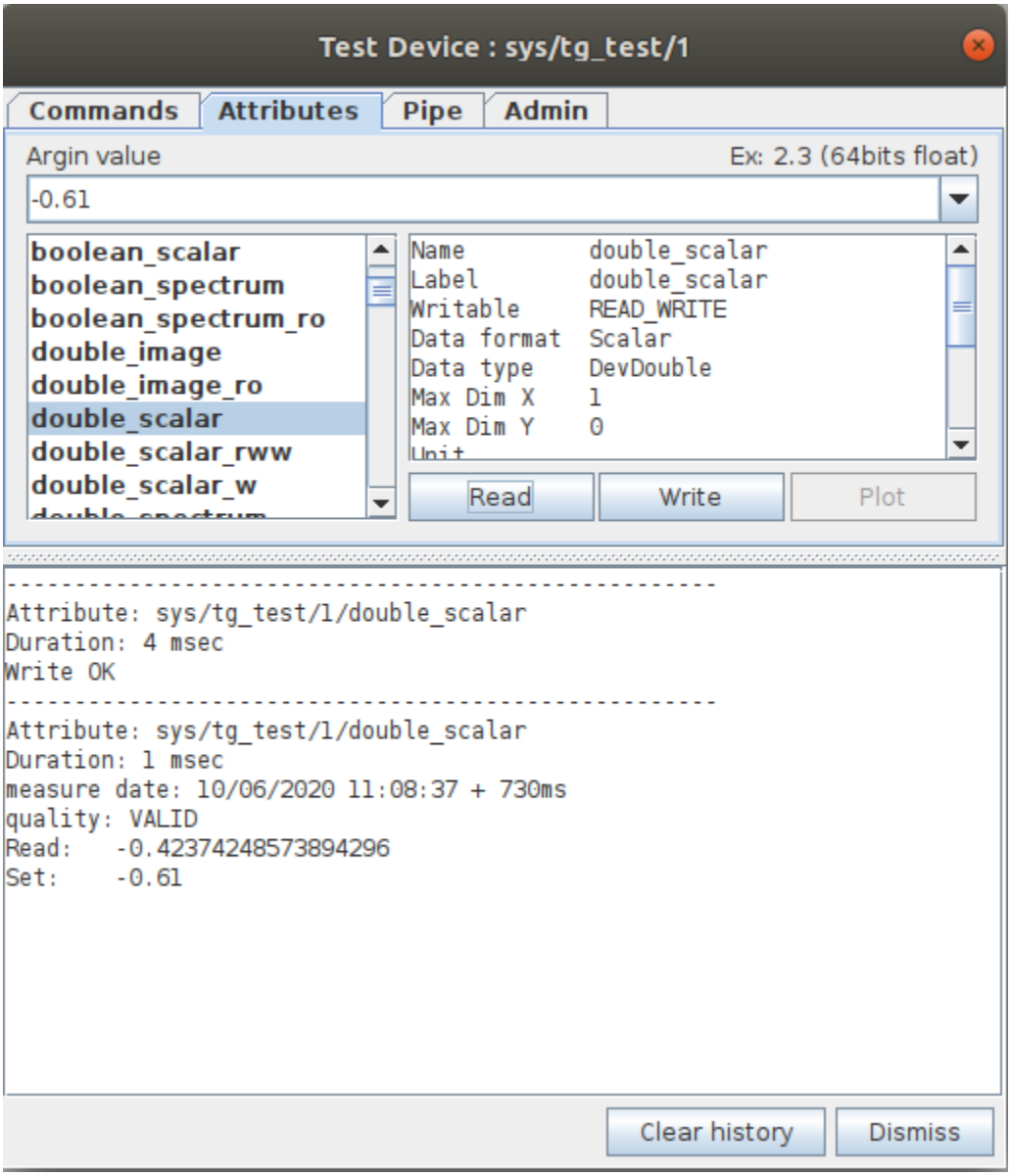


Fig. 10: Test attributes view

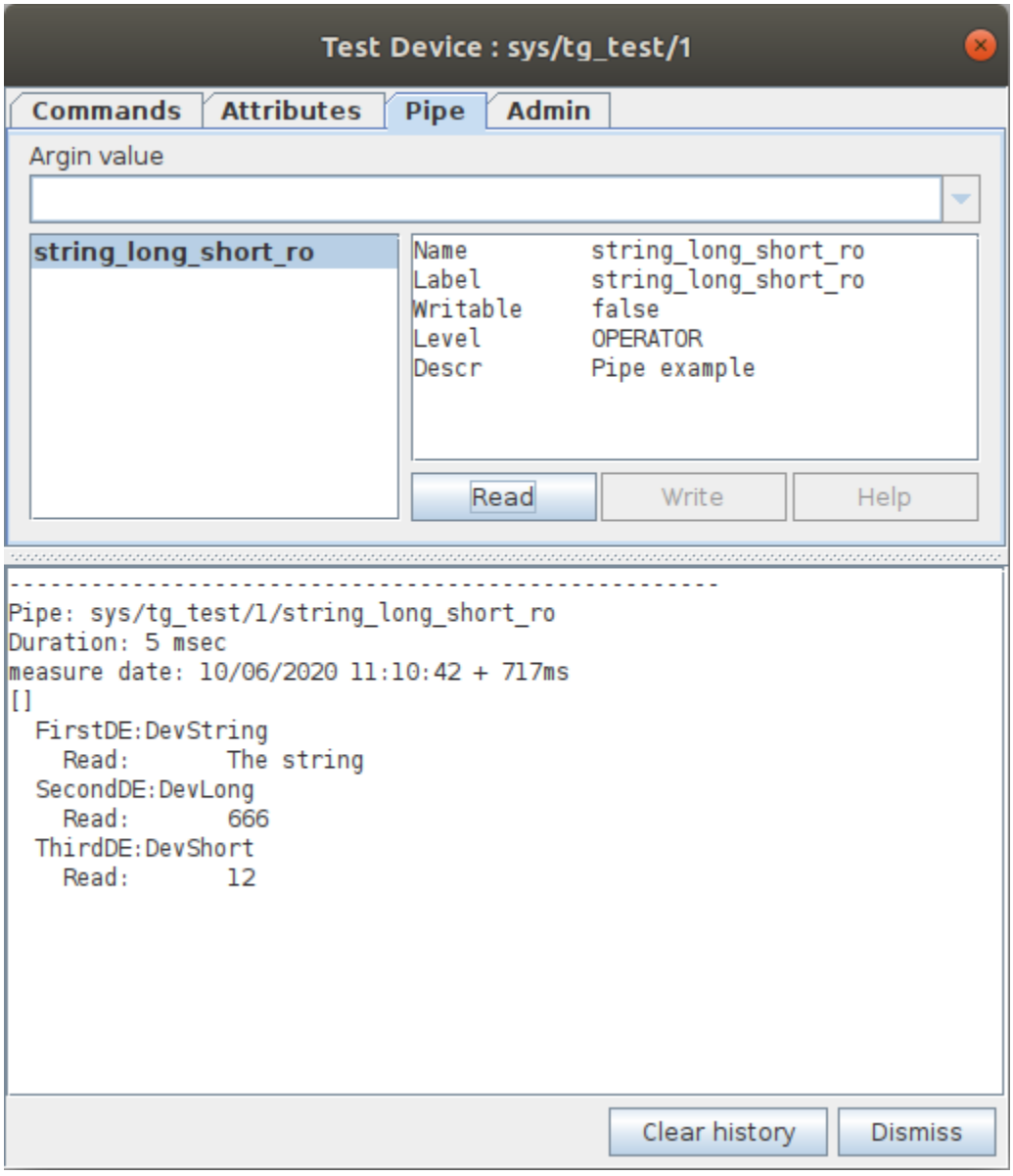


Fig. 11: Test pipes view

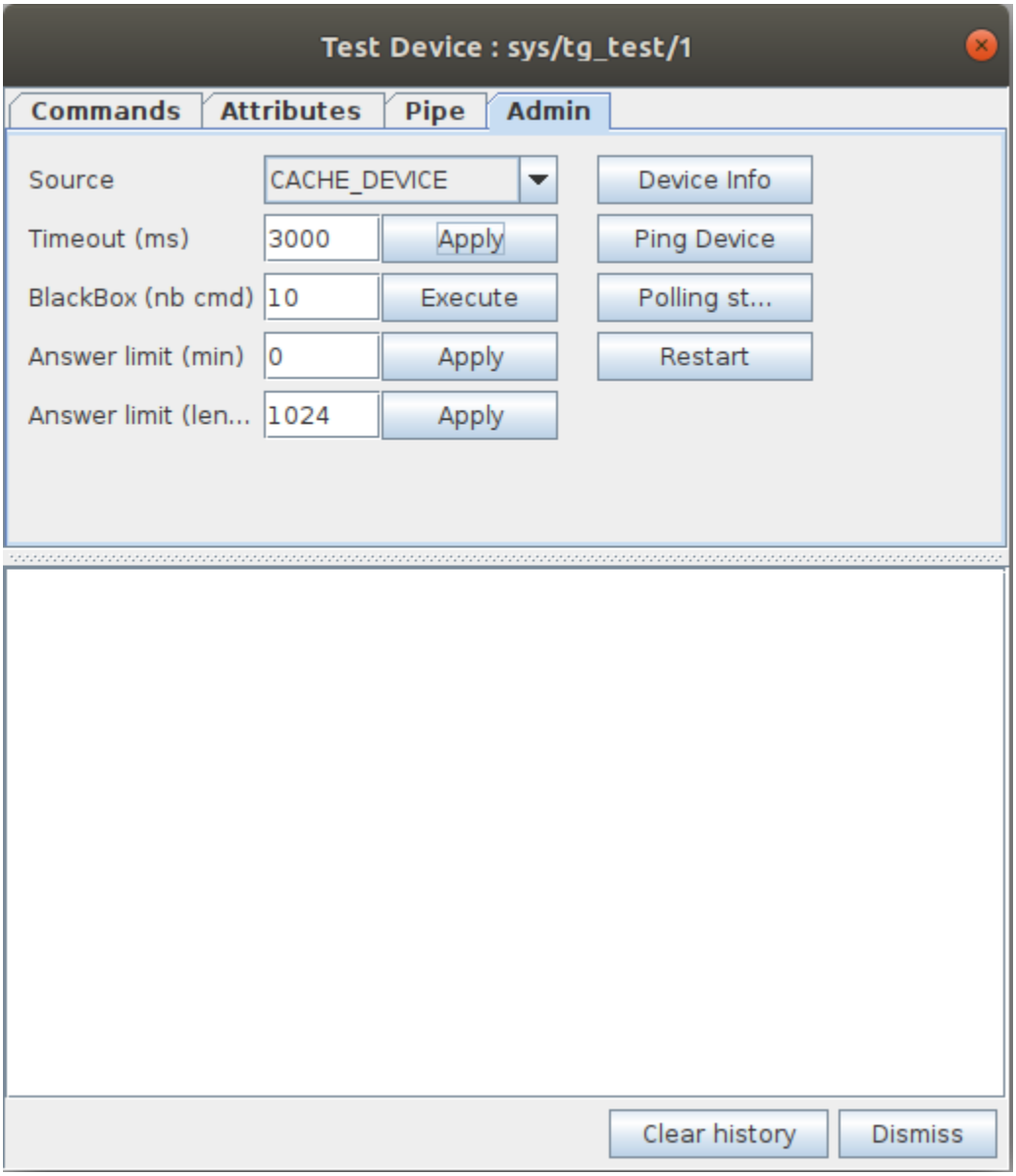


Fig. 12: Test admin view

On the *Admin* tab of *Test Device* window you can set the device's administrative configuration such as value source, timeout, etc.

Trends

By choosing *View* > *Numeric & State Trend* or *View* > *Boolean Trend* you can see the numeric & state or boolean trend of selected attributes.

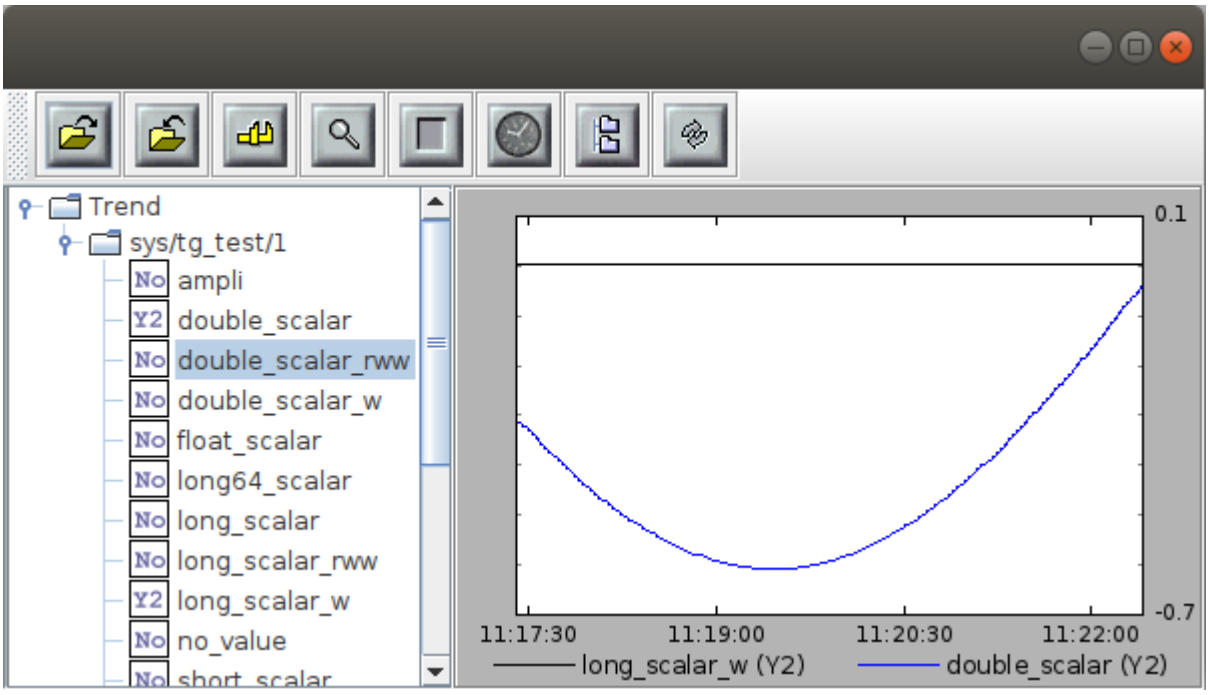


Fig. 13: Numeric & State trend

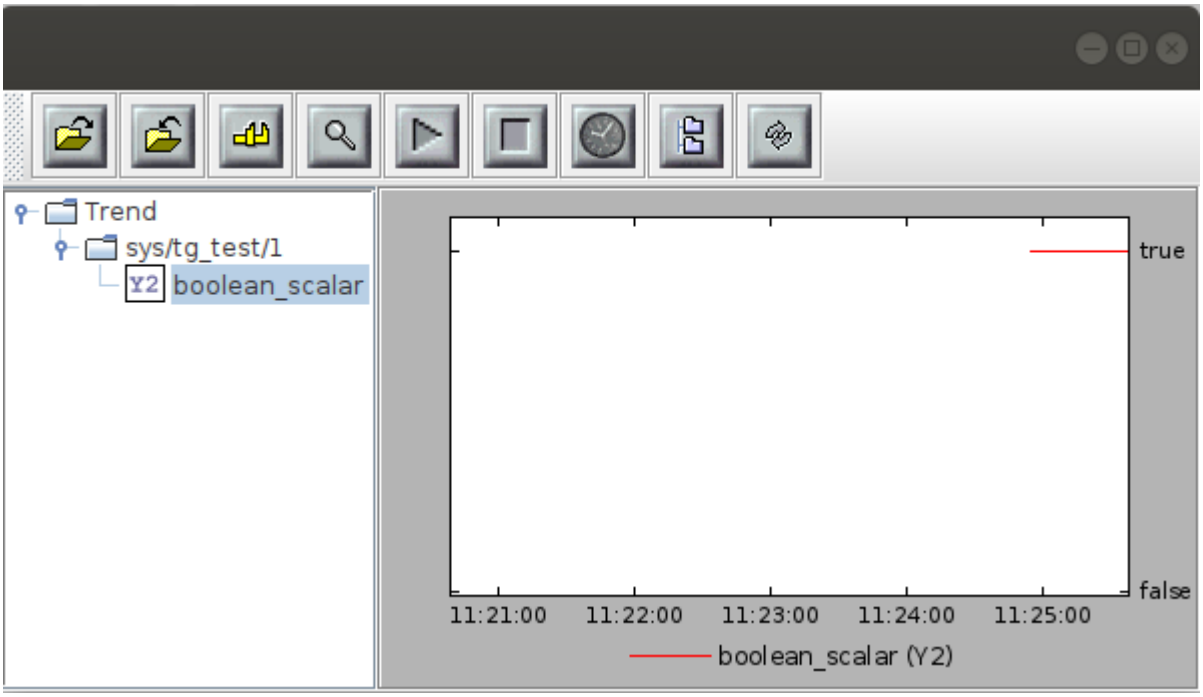


Fig. 14: Boolean trend

You can add attributes to the trend by right-clicking the attribute and selecting the desired axis. You can plot the data on X and two Y axes.

Error History

The *View* > *Error history* menu option opens the list of recent Tango errors that occured with the device.

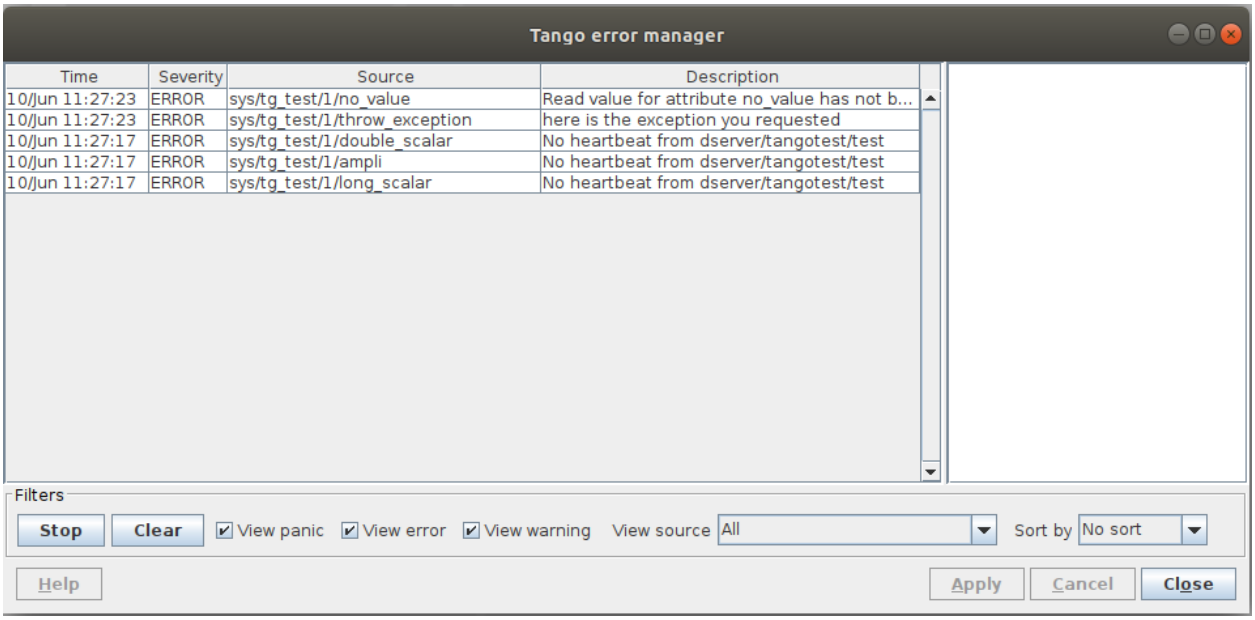


Fig. 15: Error history

Diagnostics

The *View* > *Diagnostic* menu option provides an overview on device diagnostic information. You can check the device interface version, events and polling statistics and command execution counts.

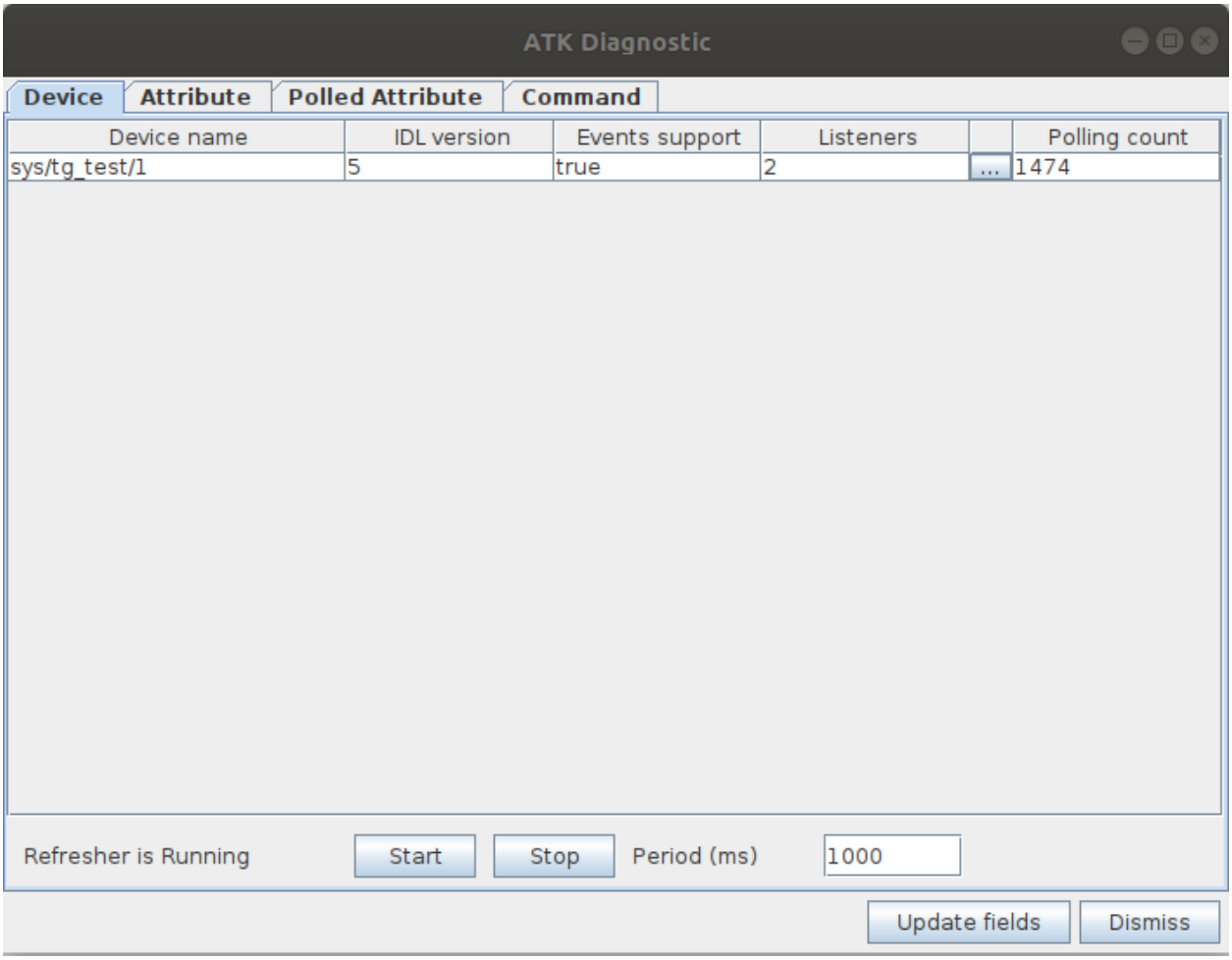


Fig. 16: Device diagnostics

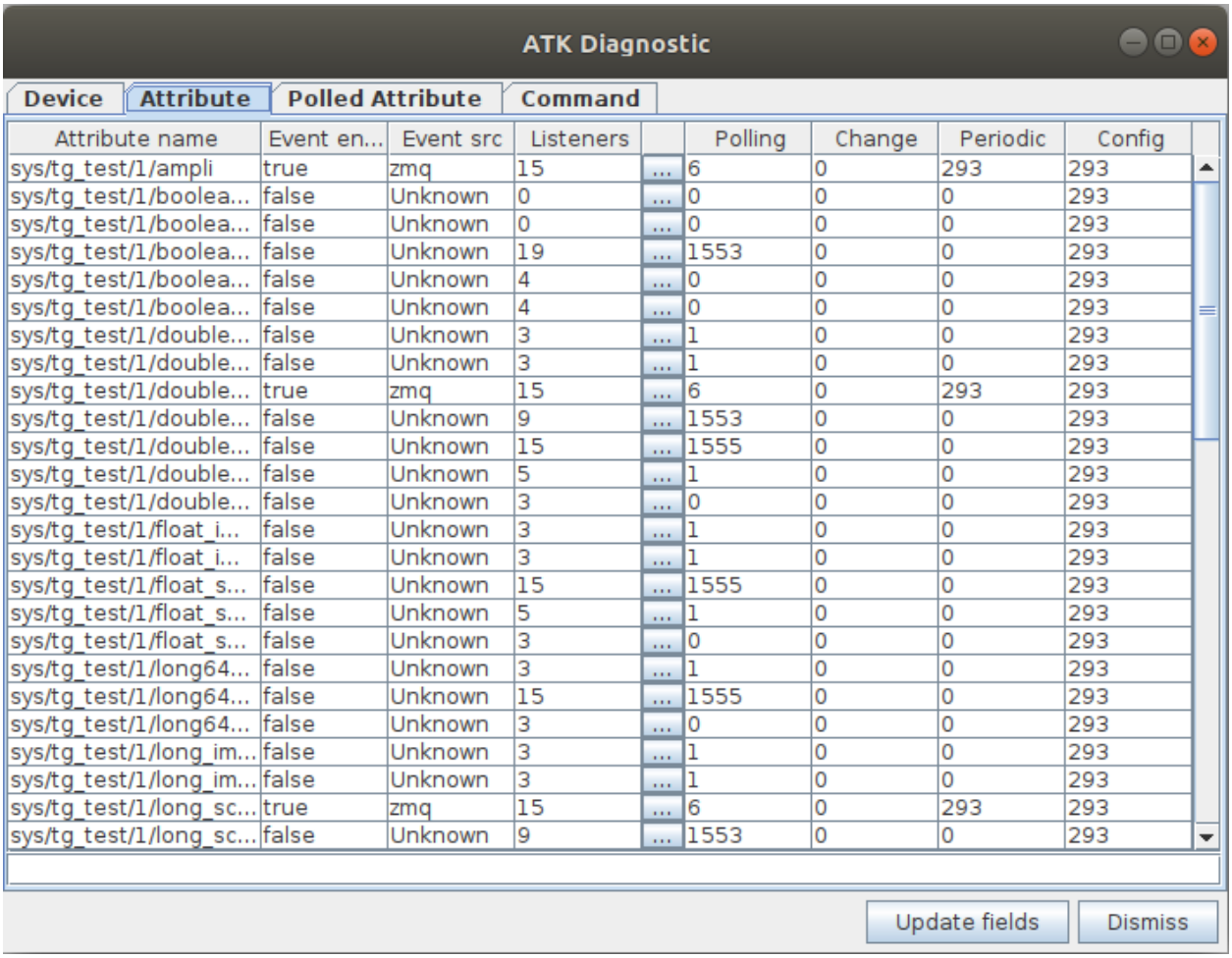


Fig. 17: Attribute diagnostics

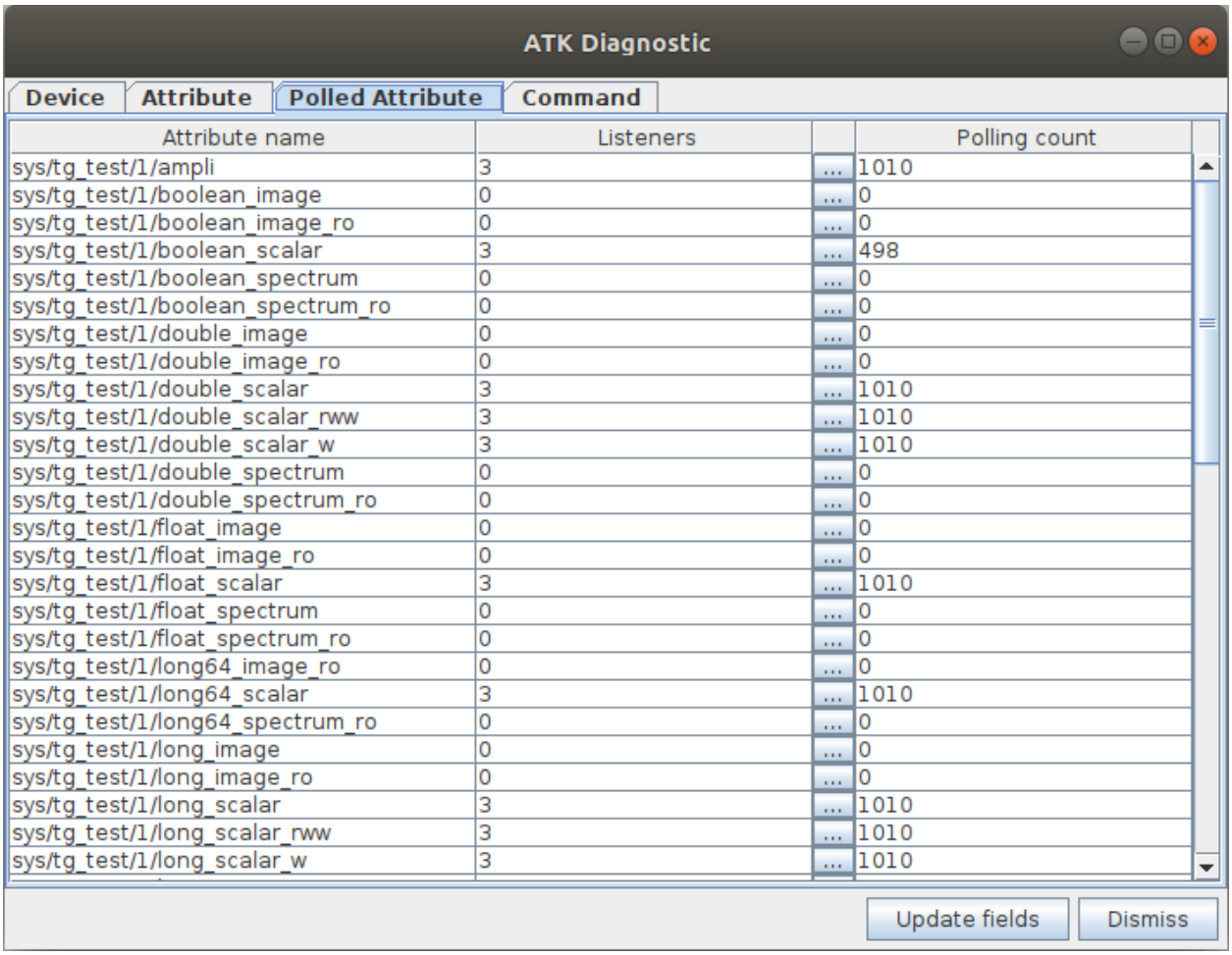


Fig. 18: Polled attribute diagnostics

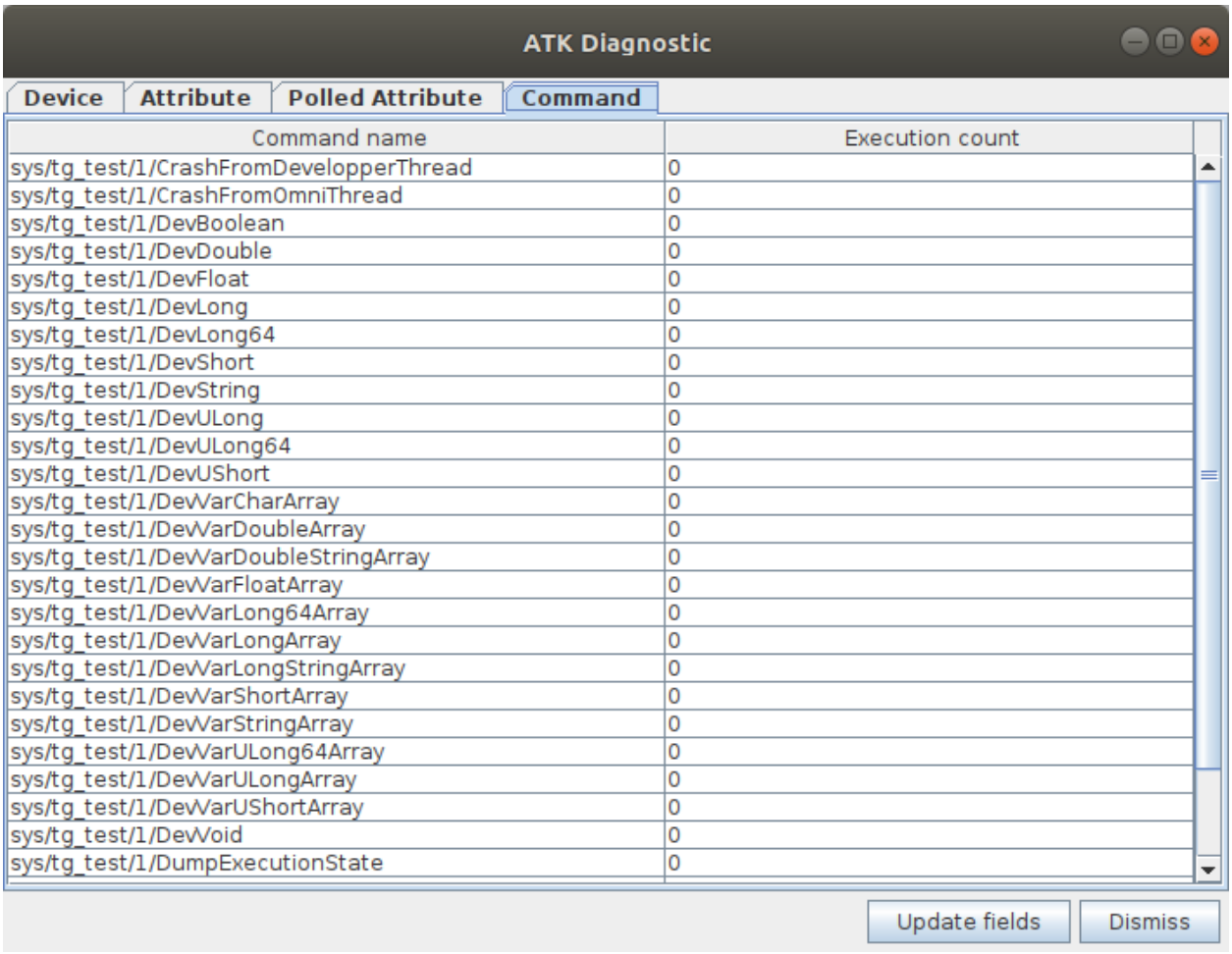


Fig. 19: Command diagnostics

Preferences

From the *Preferences* menu you can tweak AtkPanel refreshing, set the timeout and switch between *Operator View* and *Expert View*. In *Expert View* you can see the attributes that have display level set to `EXPERT`.

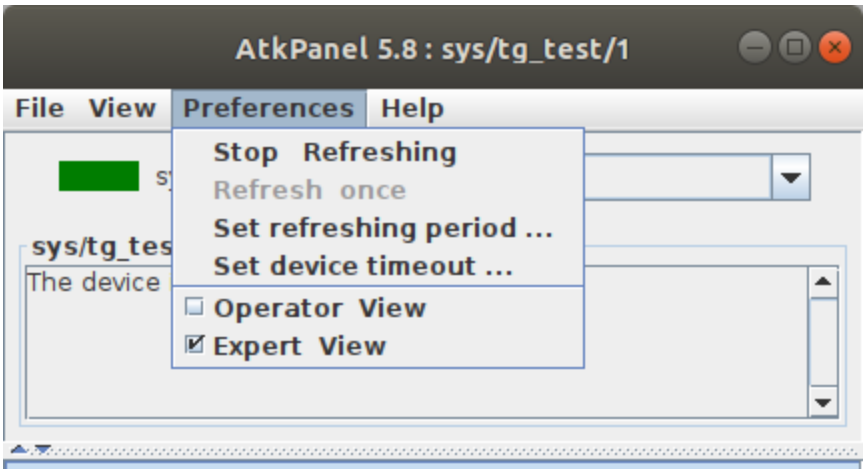


Fig. 20: AtkPanel Preferences

7.2.4 Pogo (Class Generator)

Intended audience: developers, ***Programming language:*** c++, java, python

Pogo is the TANGO code generator.

Introduction

Intended audience: developers, ***Programming language:*** c++, java, python

- Pogo is the TANGO code generator written in Java swing.
- This graphical interface allows to define a TANGO class model.
- This class model will be saved in a **.xmi** file.
- **From this TANGO class model Pogo is able to generate:**
  - a device server in C++, Java or Python.
  - An *html* documentation based on information entered during class model creation.
- The code generation part is based on EMF (Eclipse Model Framework) associated with Xtext and Xtend classes.
- Requirement: This tool needs java 1.7 or higher to be able to run.
- Source:

  project: https://github.com/tango-controls/pogo

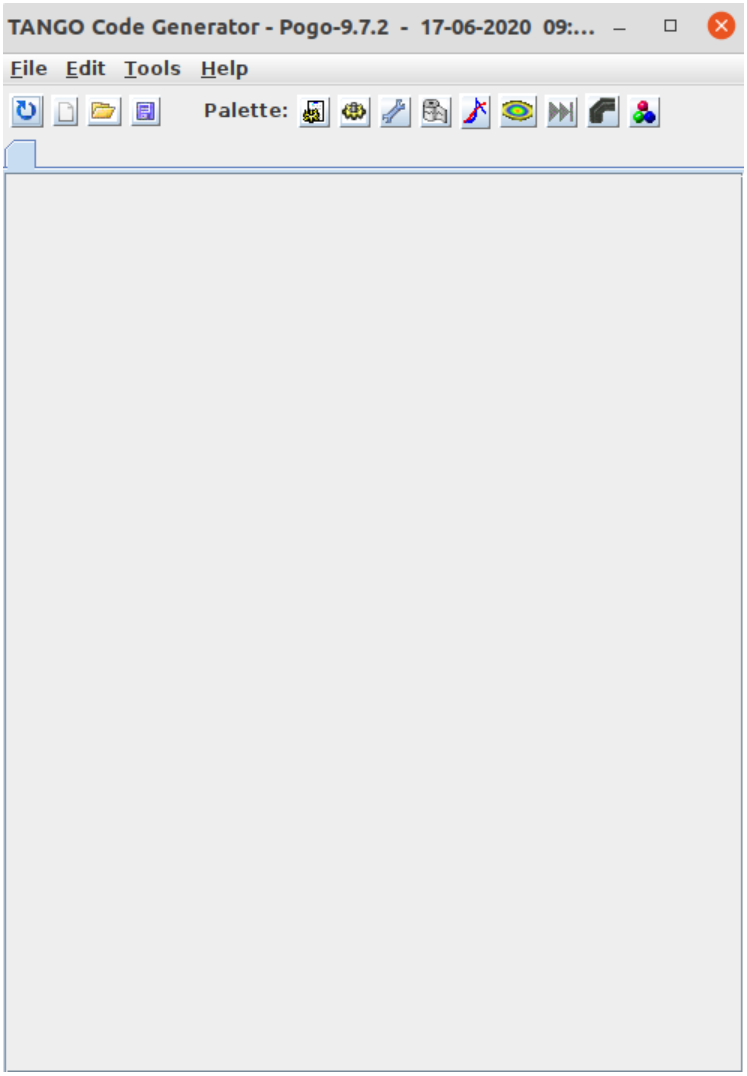
  download: https://bintray.com/tango-controls/maven/Pogo/_latestVersion

Starting with Pogo

Intended audience: beginners, developers, ***Programming language:*** c++, java, python

TANGO class creation

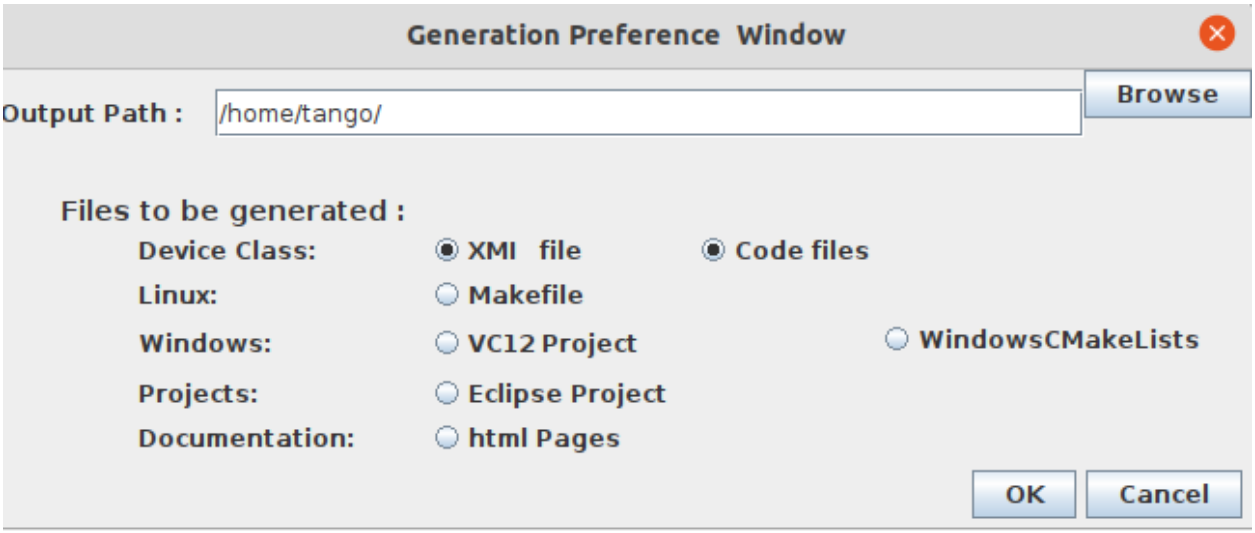
Pogo will start as follow.



Then, create a new class using File/New menu and add your:

- Properties
- Commands
- Attributes
- Pipes
- States

When your class is defined, you can generate code:



Select what you want to generate:

- **.xmi** file is the project itself, it will be loaded when you re-open your project with Pogo.
- Linux **Makefile** or **CMakeList.txt**
- **VC12** project (Windows project)
- **pom.xml** (if Java and maven path structure found)
- **html documentation** : html pages based on information entered during class model creation.

You are now able to compile the generated code and if the server is defined in database (using Jive) you are also able to run it.

Using templates

If you define an environment variable TEMPLATES_PATH to a directory containing `xmi` files,
you will be able to use these `xmi` files as templates.

e.g.: export TEMPLATES_PATH=$TANGO_HOME/templates/pogo

Generated files

Intended audience: developers, ***Programming language:*** c++, java, python

- The source code will not be read when you re-open your project. Only the **.xmi** will be re-loaded.
- Your own code must be added only between specific tags: On following example, only the grey part will not be overwritten at next code generation.

```
844  //--------------------------------------------------------
845  void PowerSupply::on()
846  {
847      DEBUG_STREAM << "PowerSupply::On()  - " << device_name << endl;
848      /*----- PROTECTED REGION ID(PowerSupply::on) ENABLED START -----*/
849
850      set_state(Tango::ON);
851      set_status("The power supply is ON");
852
853      /*----- PROTECTED REGION END -----*/    //  PowerSupply::on
854  }
855  //--------------------------------------------------------
856  /**
```

Generated code:

- *C++ code*
- *Python code*
- *Java code*

C++ Generated files

If no inheritance has been specified, the generated classes will inherit from Device_4Impl (Tango-7.x.x or above).

The generated code structure will look like to the Pogo-6 code.

`MyObject.h`	Containing created class data members and prototypes.
`MyObject.cpp`	Containing created class methods for init, commands, read/write attributes, …..
`MyObjectClass.h`	Containing data members and prototypes for MyObject-Class.cpp. Containing also the Command and Attribute class definitions.
`MyObjectClass.cpp`	A singleton class derived from DeviceClass. It implements the command and attribute lists and all properties and methods required by the created class once per process.
`MyObjectStateMachine.cpp`	Containing created class methods for the state machine.
`ClassFactory.cpp`	Containing created class methods creating used class. In case of multi class server, add other class(es) in the factory.
`main.cpp`	Start point of the device server. Most of the time, not touched by the programmer.

- A method called `add_dynamic_attributes()` has been added to the `MyObject.cpp`. It will be called at startup to create dynamic attributes if any. .. warning:: It is NOT generated if the class is abstract !

Python Generated files

If no inheritance has been specified, the generated classes will inherit from `Device_4Impl` (Tango-7.x.x or above).

The generated code structure will look like to the Pogo-6 code.

The python templates have been implemented by Sebastien Gara at Nexeya

`MyObject.py`	Containing created class python code.

Java Generated files

The generated Java classes are not compatible with the server API from TangORB.

They are compatible only with the new design from Gwenaelle Abeille at Soleil. See Java servers

`org.tango.myobject.MyObject.java`	Containing created class java code.
`org.tango.myobject.MyDynamicAttribute.java`	Containing created java code for dynamic attribute class if any.

Projects:

- *pom.xml*
- *Windows projects*

pom.xml (MAVEN) file

- Pom.xml is MAVEN project file. It can be loaded as project by IDE (like IntellijIDEA).
- To generate this file, you must generate the xmi file in a path like *../../src/main/java*

Windows project files

Pogo supports **Visual C++** projects.

It will generate files in a directory name `vcxx_proj` (where xx is the Visual C release. e.g. `vc12_proj`)

Projects use the `TANGO_ROOT` environment variable to find include and library files.

It provides 64 bits and debug/release modes for each solution.

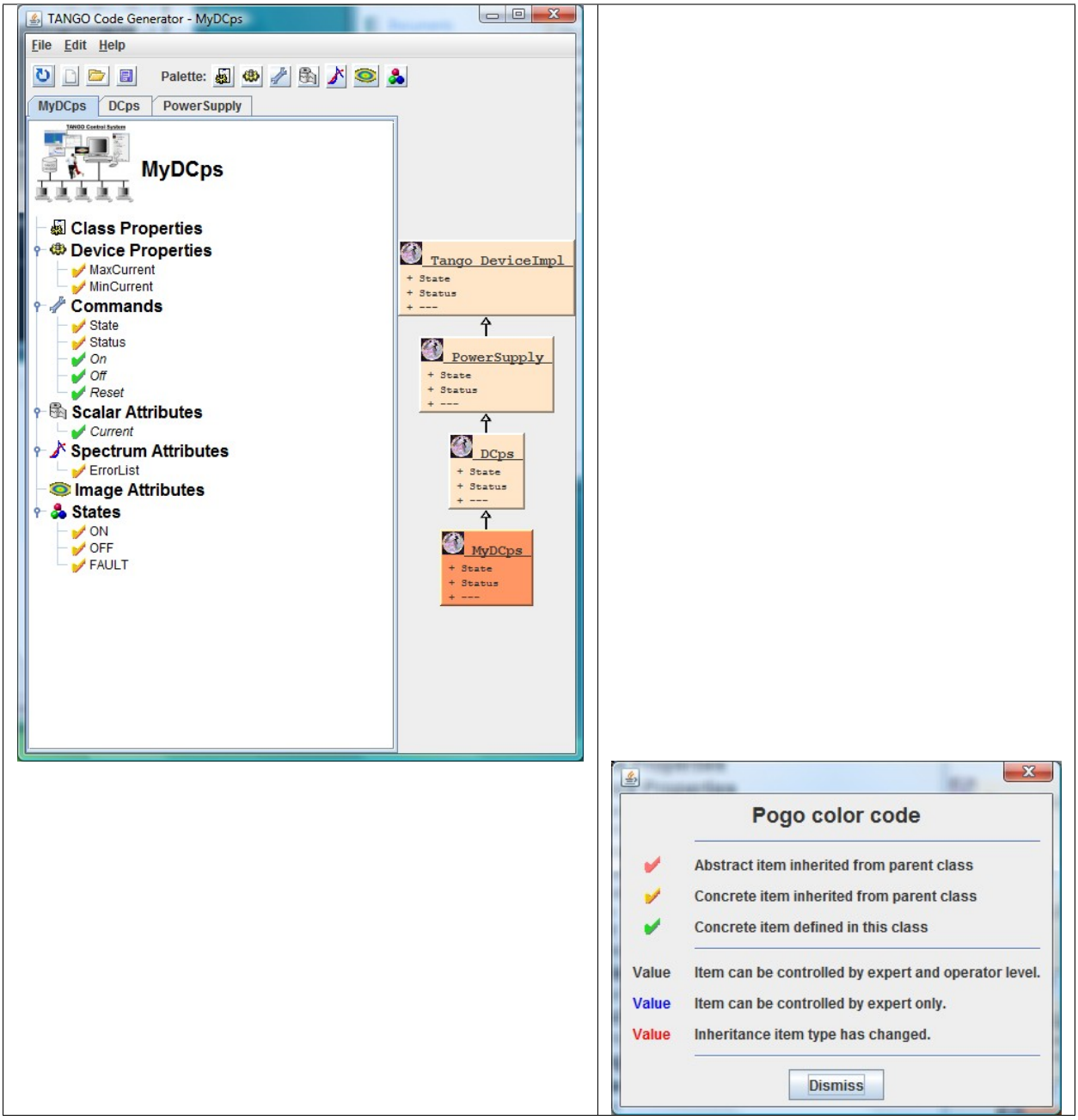
In this directory 5 files are generated:

`MyObject.sln`	Global solution project
`Class_lib.vcxproj`	Project to create a static library for the class
`Class_dll.vcxproj`	Project to create a dynamic-link library for the class
`Server_static.vcxproj`	Project to create a static server (using static library)
`Server_shared.vcxproj`	Project to create a dynamic server (using dll)

Inheritance Management

Intended audience: developers, ***Programming language:*** c++, java, python

- A TANGO device class inherite from a DeviceImpl class. It could also inherite from another TANGO device class.
- To inherite from another TANGO device class, use **Add Inheritance Class** in class creation window.
- **Inheritance is not simply used as template.**
  That means that `*.h` and `*.cpp` files of inherited class must be in `Makefile`.
- Inheritance could be done from a class containing code and not only from pure abstract class.
- An inheritance diagram is displayed on right side and all classes in this diagram are edited.
- A color code display if items are abstract or concrete.

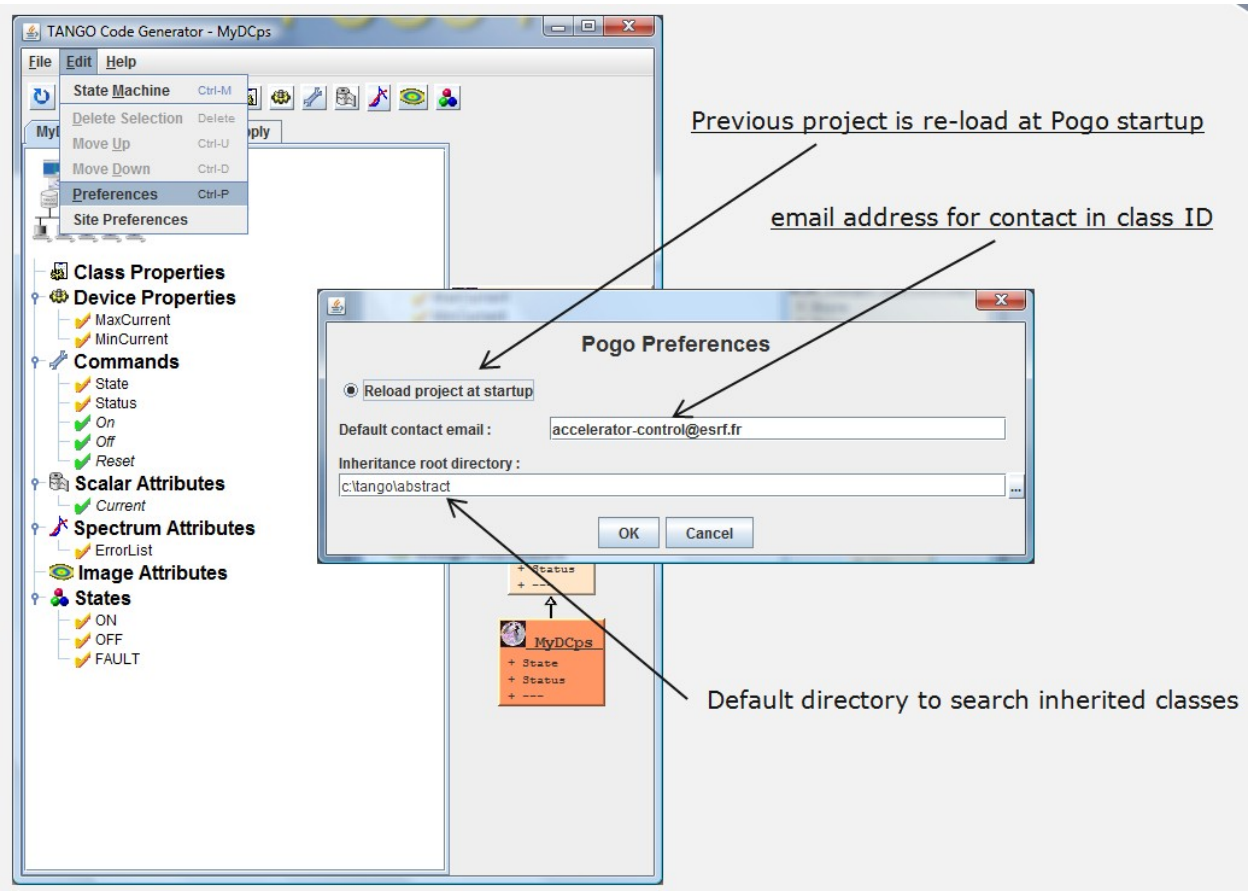

TANGO Code Generator - MyDCps
File Edit Help
Palette:
MyDCps
DCps
PowerSupply
MyDCps
Class Properties
Device Properties
MaxCurrent
MinCurrent
Commands
State
Status
On
Off
Reset
Scalar Attributes
Current
Spectrum Attributes
ErrorList
Image Attributes
States
ON
OFF
FAULT
Tango DeviceImpl
+ State
+ Status
+ ---
PowerSupply
+ State
+ Status
+ ---
DCps
+ State
+ Status
+ ---
MyDCps
+ State
+ Status
+ ---
Pogo color code
Abstract item inherited from parent class
Concrete item inherited from parent class
Concrete item defined in this class
Value
Item can be controlled by expert and operator level.
Value
Item can be controlled by expert only.
Value
Inheritance item type has changed.
Dismiss

Preferences

Intended audience: developers, ***Programming language:*** c++, java, python

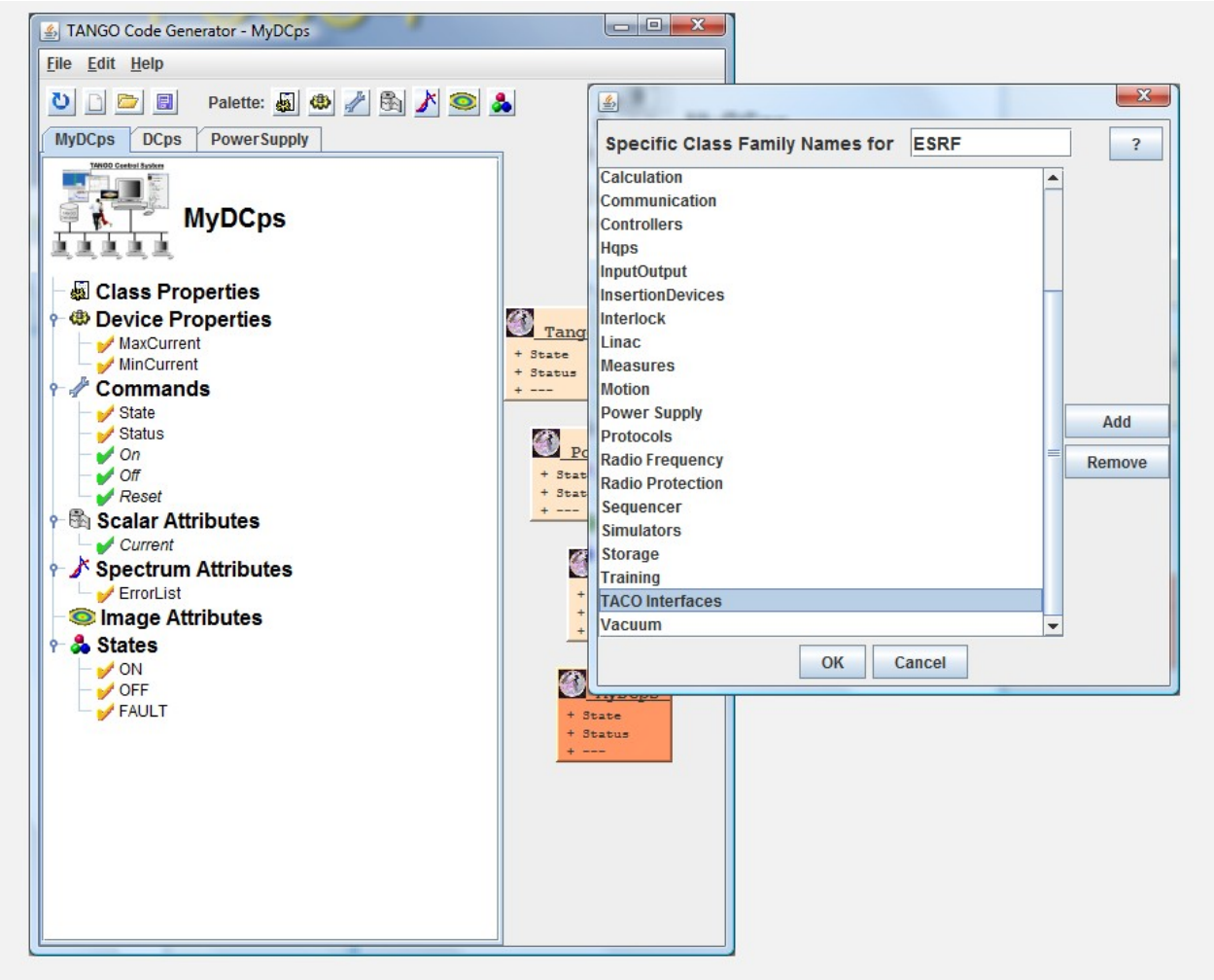
Programmer Preferences

You can customize your application as follow.



Institute Preferences

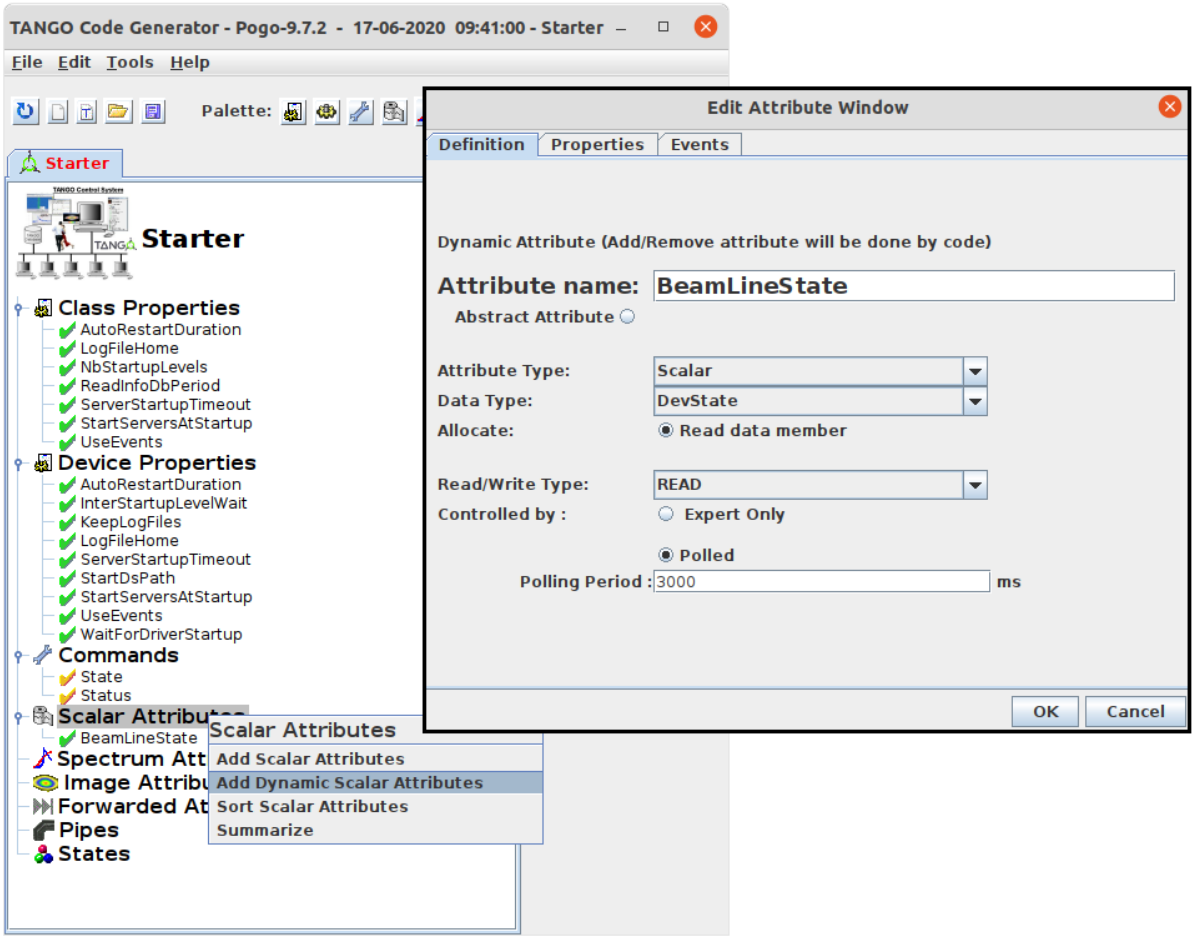
If you use your own repository with specific class families, manage the family list as follow.



Dynamic Attributes

Intended audience: developers, ***Programming language:*** c++, java, python

Pogo allows you to create and manage dynamic attributes You can create a dynamic attribute like a static one.



An additional file will be generated with dynamic attribute utility methods.

The programmer will be able to add and remove every where in his code. For instance during a command execution or with an external trig.

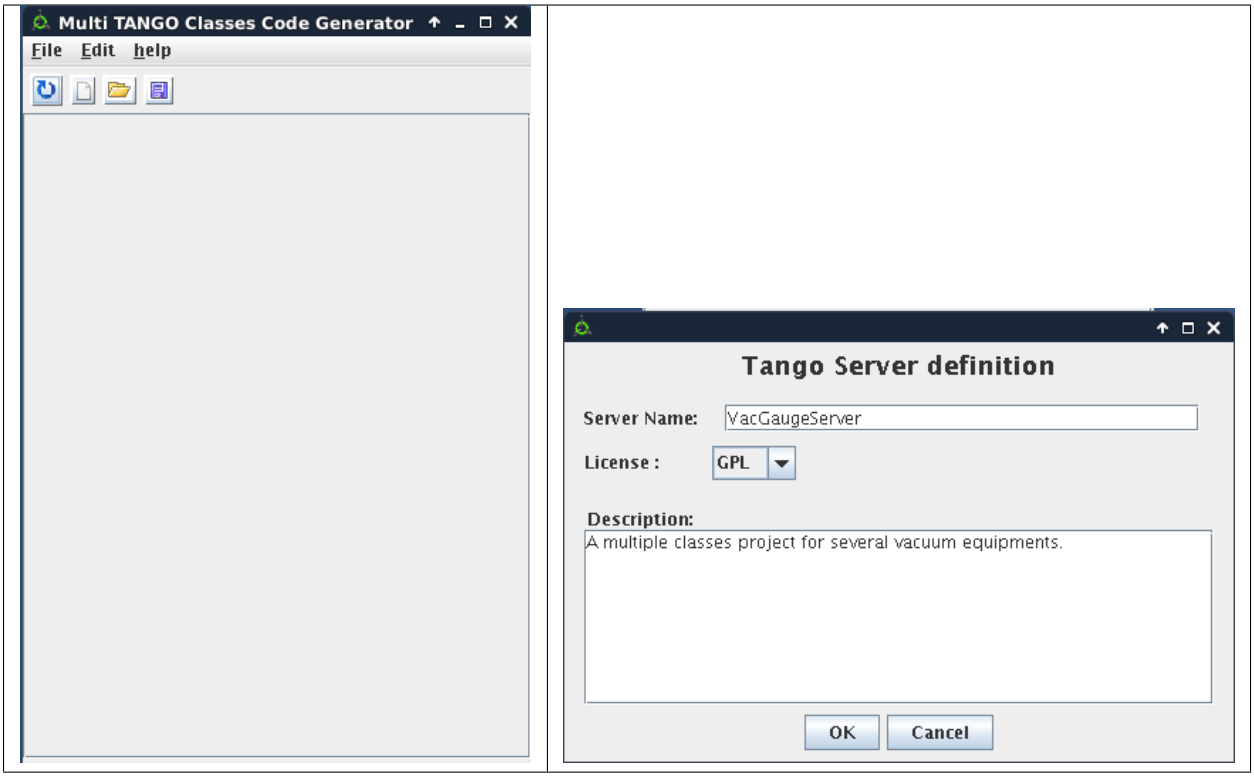
Pogo will generate a method to add dynamic attribute at server startup.

```
/**
 *  Read BeamLineState attribute
 *  Description:
 *
 *  Data type:  Tango::DevState
 *  Attr type:  Scalar
 */
//--------------------------------------------------------
void BLsurvey::read_BeamLineState(Tango::Attribute &attr)
{
    DEBUG_STREAM << "BLsurvey::read_BeamLineState(Tango::Attribute &attr) entering... " << endl;
    Tango::DevState *att_value = get_BeamLineState_data_ptr(attr.get_name());

    /*----- PROTECTED REGION ID(BLsurvey::read_BeamLineState) ENABLED START -----*/

    //  Set the attribute value
    for (unsigned int i=0 ; i<beamLines.size() ; i++)
    {
        string  attName = beamLines[i]->name;
        attName += "State";
        if (attName==attr.get_name())
        {
            *att_value = beamLines[i]->getState();
            attr.set_value(att_value);
        }
    }

    /*----- PROTECTED REGION END -----*/    //  BLsurvey::read_BeamLineState
}

//--------------------------------------------------------
/**
 *  Method      : BLsurvey::add_dynamic_attributes()
 *  Description : Create the dynamic attributes if any at server startup
 *                for specified device.
 */
//--------------------------------------------------------
void BLsurvey::add_dynamic_attributes()
{
    //  Example to add dynamic attribute:
    //  add_BeamLineState_dynamic_attribute("MyAttribute");

    /*----- PROTECTED REGION ID(BLsurvey::add_dynamic_attributes) ENABLED START -----*/

    for (unsigned int i=0 ; i<beamLines.size() ; i++)
    {
        string  attStateName = beamLines[i]->name;
        attStateName += "State";
        add_BeamLineState_dynamic_attribute(attStateName);
    }
    /*----- PROTECTED REGION END -----*/    //  BLsurvey::add_dynamic_attributes()
}
```

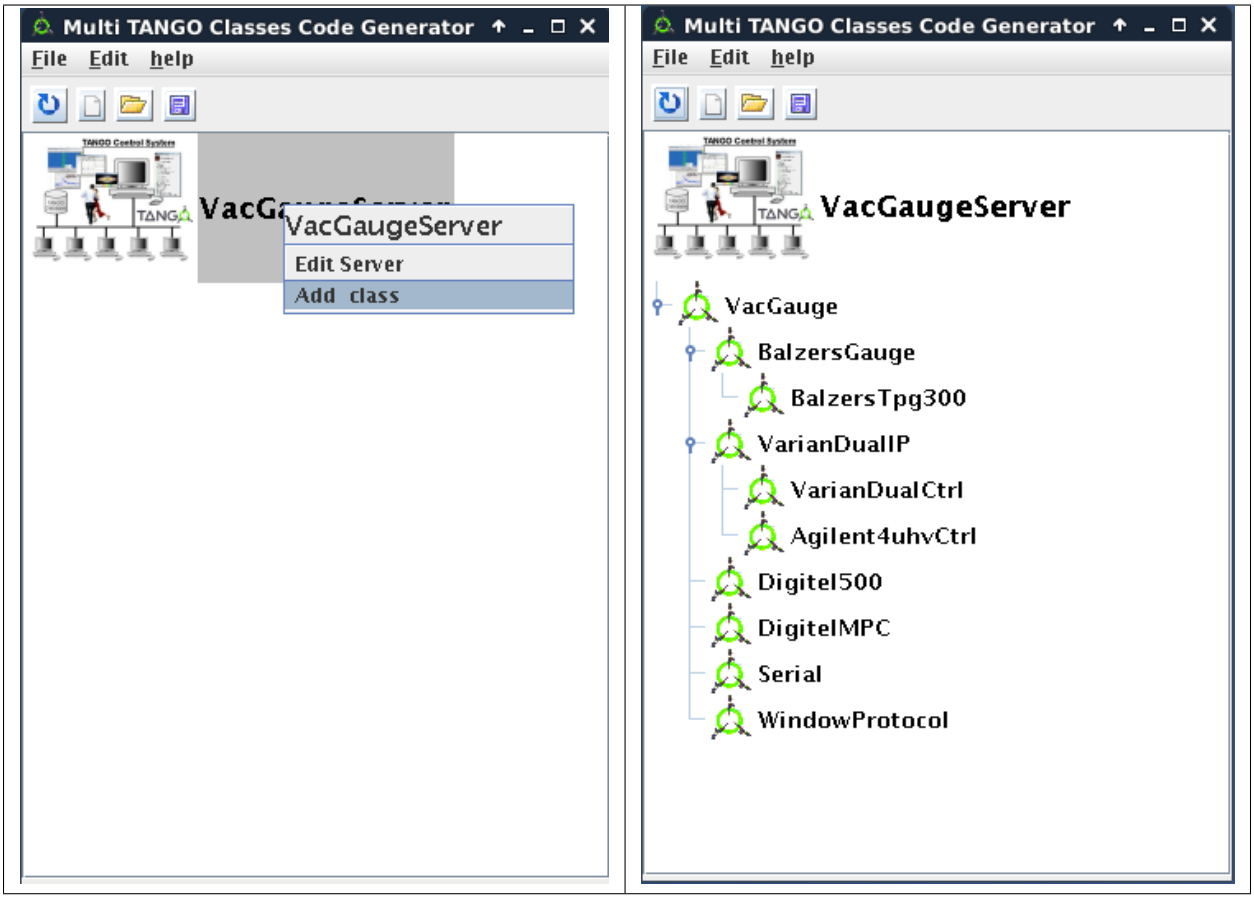
Multiple C++ Classes Manager

Intended audience: developers, ***Programming language:*** c++, java, python

- **Since `Pogo-7.2` on Linux, Pogo allows you to create multiple**
  C++ classes server project.
- **Use the *tools* menu (or `pogo -multi` shell command) to launch**
  the multiple classes GUI.
- Then create a new server project.
- The server name could have the same name of one class in project. But it is not mandatory.



- Then you can add several classes by selecting `xmi` files.



- Now, you can generate code where a class is defined or in an independent one.
- The 4 generated files are:
  - `VacGaugeServer.multi.xmi`: Multiple classes project file
  - `MultiClassesFactory.cpp`: Equivalent to the ClassFactory.cpp containing all specified class dependencies
  - `main.cpp`: The same main.cpp as a single class project
  - `Makefile.multi`: A Makefile containing all specified class dependencies

Tango Server Packaging

Intended audience: developers, ***Programming language:*** c++, java, python

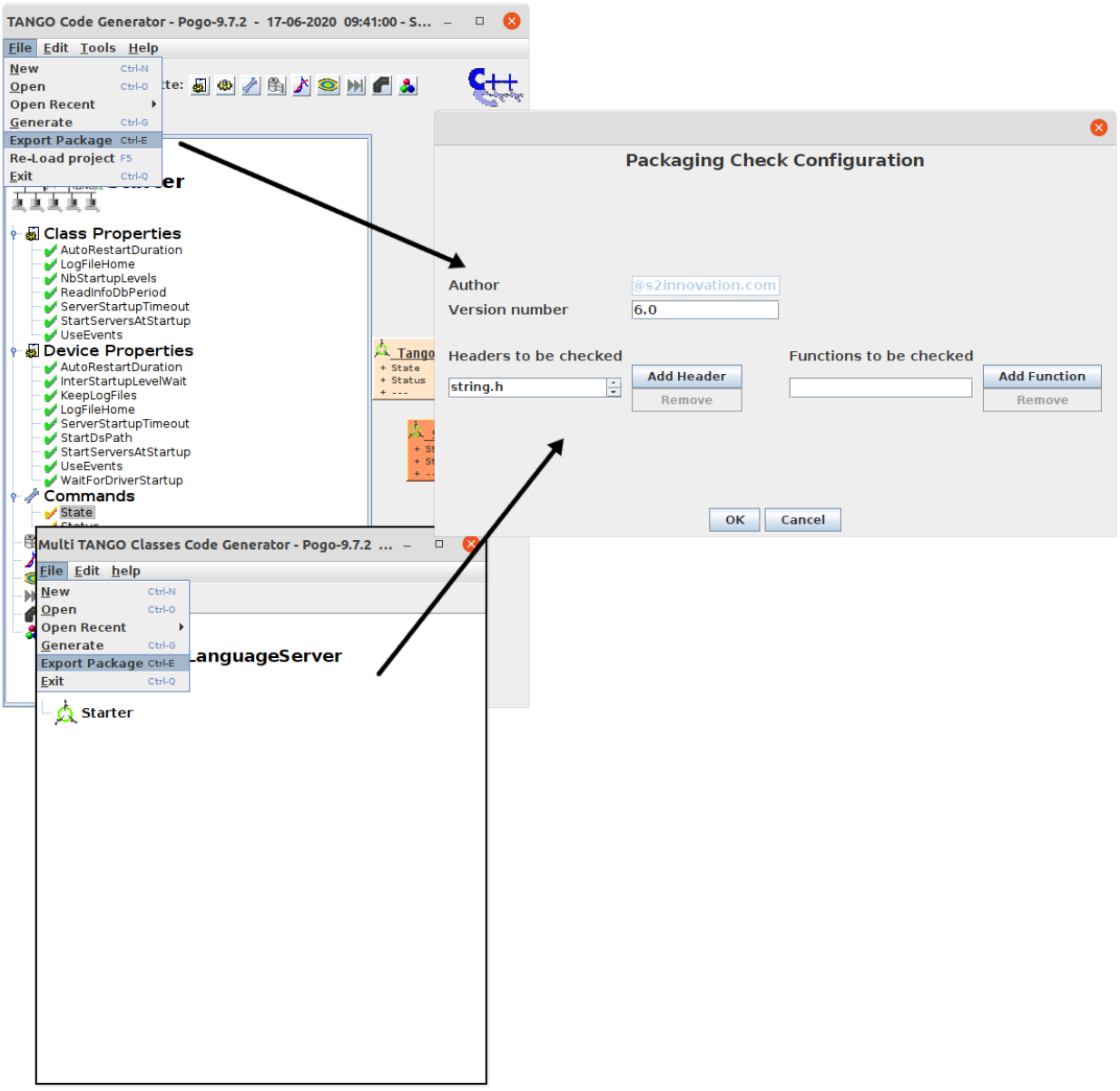
Since **`Pogo-8.3.0`** on **Linux**, **`Pogo`** allows you to create packaging files to be used with autotools

Set the environment variable `PKG_CONFIG_PATH`

- e.g. for tango.pc file : **`export PKG_CONFIG_PATH=$TANGO_LIBS/pkgconfig`**

You can find many examples on the web on pkg_config_path management. In **`Pogo`** or **`Multi classes GUI`**, use *File* menu and *Export Package* item.

A panel will be launched to define to configure packaging.



After clicking on *OK* button, a new directory `packaging` will be created.

You can use **`autotools`** commands (**`autogen`**, **`configure`**, **`make`**, **`make install`**, command:*make distcheck* to build a reliable packaging for your server.

Pogo-6 Compatibility

Intended audience: developers, ***Programming language:*** c++, java, python

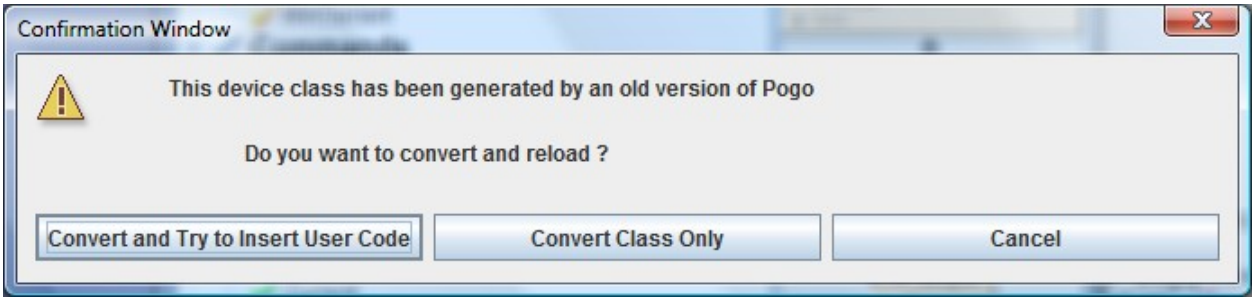
This application is supposed to be able to re-load a class generated with a **Pogo-6.x** application. If **Pogo-6.x** can load it, **Pogo** is also able to load it. *Open* the `XXXX.h` file.

BUT:

It will try to insert your own code at right place. Of course all cases cannot be tested:

- Something could be missing (test your server before other modication)
- Sometimes it could fail and do not insert part of code.

When it load an old project it will propose:



The code insertion is available only for **C++**. If code insertion fails, or for **Python** and **Java**, generate class only and add your code by copy/paste action.

For C++

If insertion works and compilation fails, most of the time it is due to:

- Declaration duplicated (one by generator, one by code insertion).
- For writable attribute, in `write_MyAttribute method`, the `attr.get_write_value()` method management has changed. It is now returned in a local variable. Your old global variable does not exist any more (it is not necessary to be global in 99% of cases). It now local and named `w_val`.
- Take care in `dev_state()` and `dev_status()` methods if you override the default.

Of course a class generated by this **Pogo** version is not able to be re-loaded by **Pogo-6.x.x** application.

7.2.5 Synoptic and JDraw

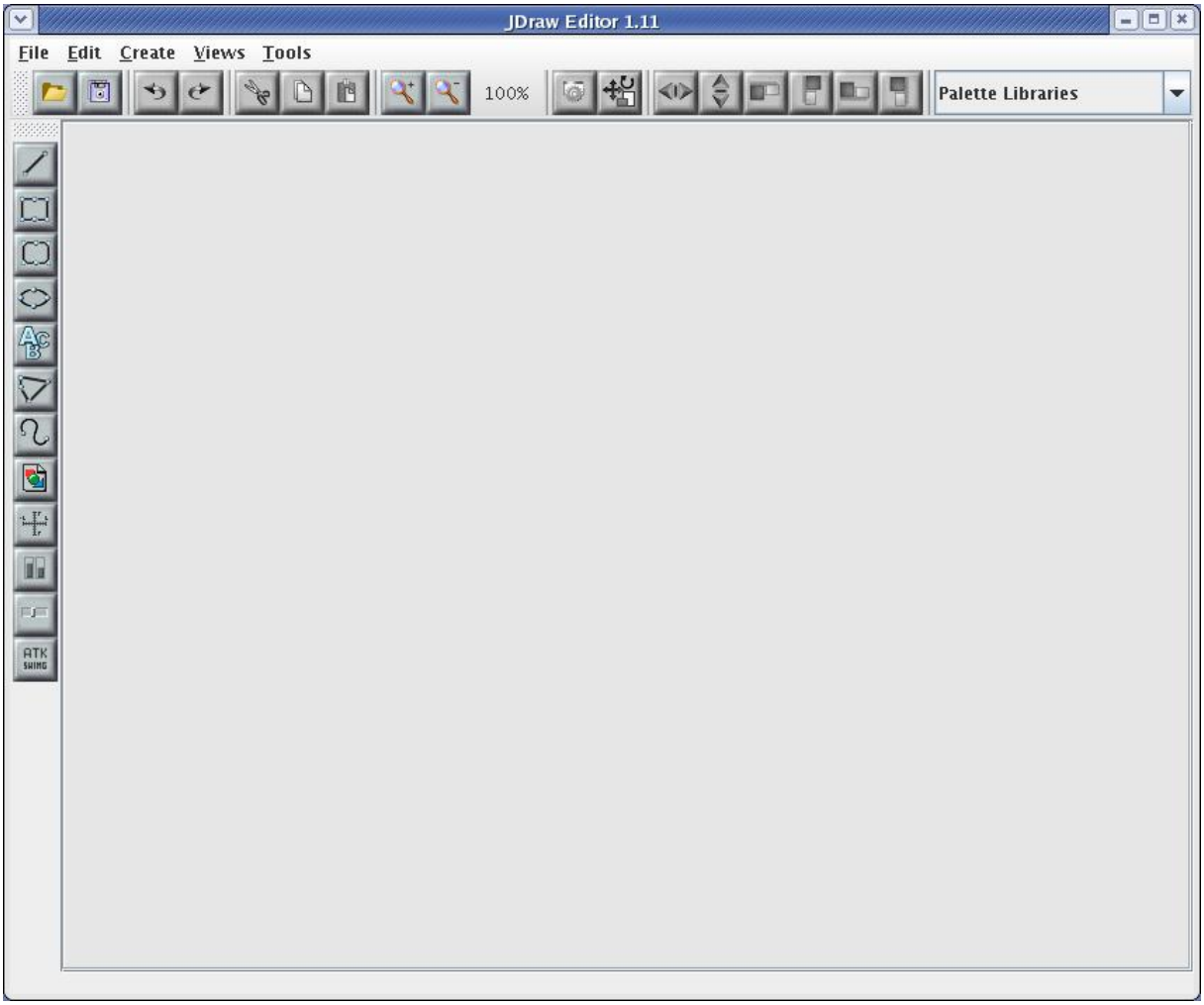
Intended audience: developers, users

ATK includes a graphical editor, to design the graphic shape of the synoptic. The editor is called **Jdraw**. This editor is included in ATK so you don't need to download any specific jarfile.

To launch **jdraw** you should start the following class :

fr.esrf.tangoatk.widget.util.jdraw.JdrawEditorFrame

Once the jdraw is started you will see the following window :

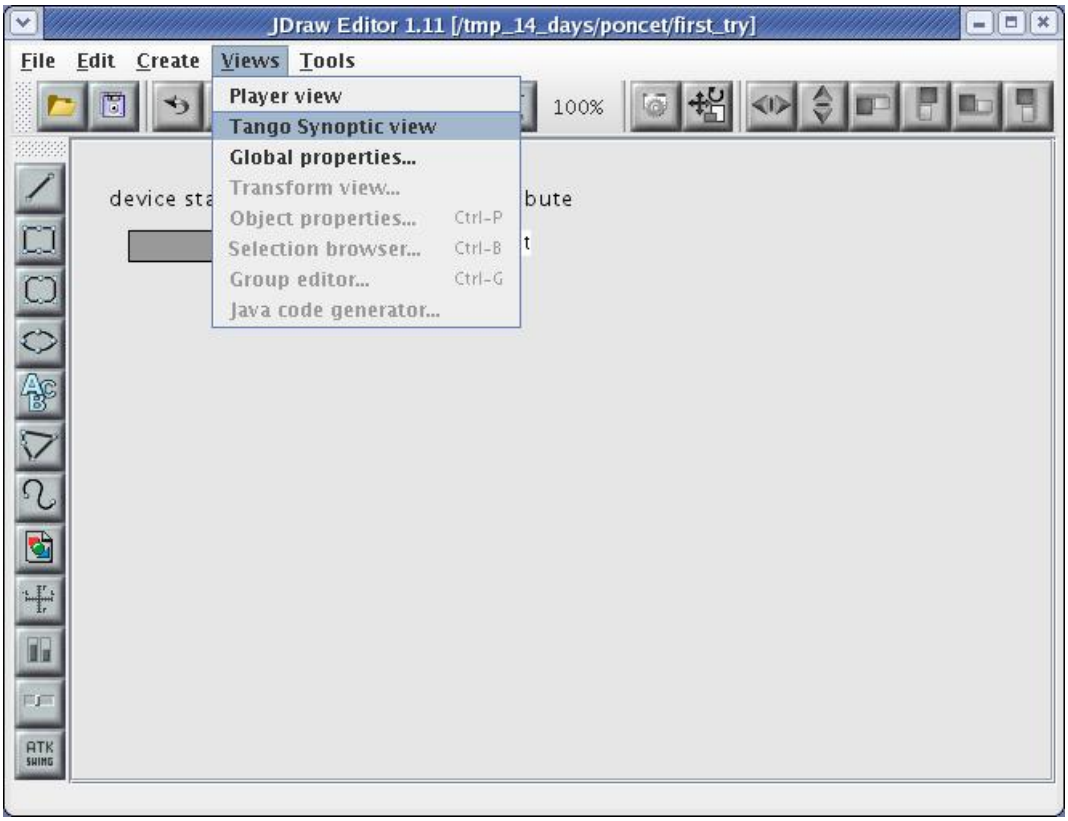


You can now start to draw the synoptic. As in any drawing editor you can group basic objects (like rectangle, circle, lines, ...) to obtain more elaborated shapes. Once the drawing is done you need to associate parts of the drawing to a Tango control system object. Click on the following link to view a Flash demo of how to draw a simple tango synoptic.

First synoptic (Flash Demo)

Test the synoptic : "Tango Synoptic view"

As soon as the synoptic is saved in a file, it's run time behavior can be tested. Inside Jdraw, select "Tango Synoptic view" from the "Views" pulldown menu to test the run time behavior of the synoptic.

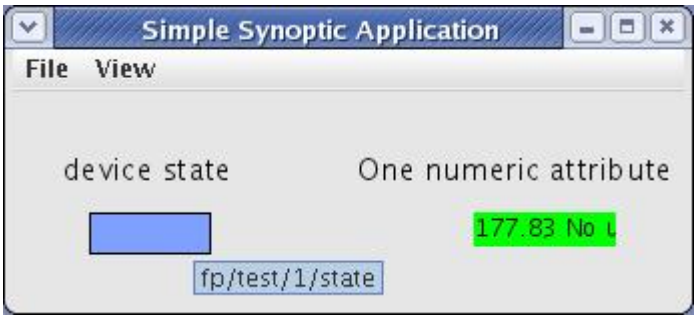


You can also start the same simple synoptic application outside jdraw editor. This application is called SimpleSynopticAppli and is included in ATK. You don't need any specific jarfile to use it.To launch **SimpleSynopticAppli** you should start the following class :

fr.esrf.tangoatk.widget.jdraw.SimpleSynopticAppli

You can pass the absolute path of the synoptic file as the argument to the SimpleSynopticAppli. In case no argument is passed on the command line, the application will popup a file selection dialog in order to get the name of the synoptic file to load.

The following screenshot shows the SimpleSynopticAppli with the synoptic file loaded.



As you can see the rectangle shows the value of fp/test/1/state attribute. The state attribute value is represented by it's corresponding color (blue). Moreover you can see the value of the *"fp/test/1/double_scalar"* value displayed by the SimpleScalarViewer inside the synoptic.

If the mouse enters the blue rectangle (while the focus is inside the synoptic window), a tooltip will display the name of the Tango attribute associated to the rectangle. The same goes for the region where the numeric value (177.83) is displayed.

A mouse click inside the blue rectangle will launch an AtkPanel for the device fp/test/1. The AtkPanel is started by default in read-only mode : device commands and attribute setters are not displayed.

Conclusion :

- A synoptic drawing can easily be made using the drawing tool Jdraw included in ATK.
- The association between the graphic components inside the drawing and the control system Tango objects are made through the name of the graphic components.
- A Simple Synoptic application is provided to test the run-time behaviour of the synoptic. It can be started directly from the Jdraw editor's menubar or outside Jdraw.

Jdraw editor

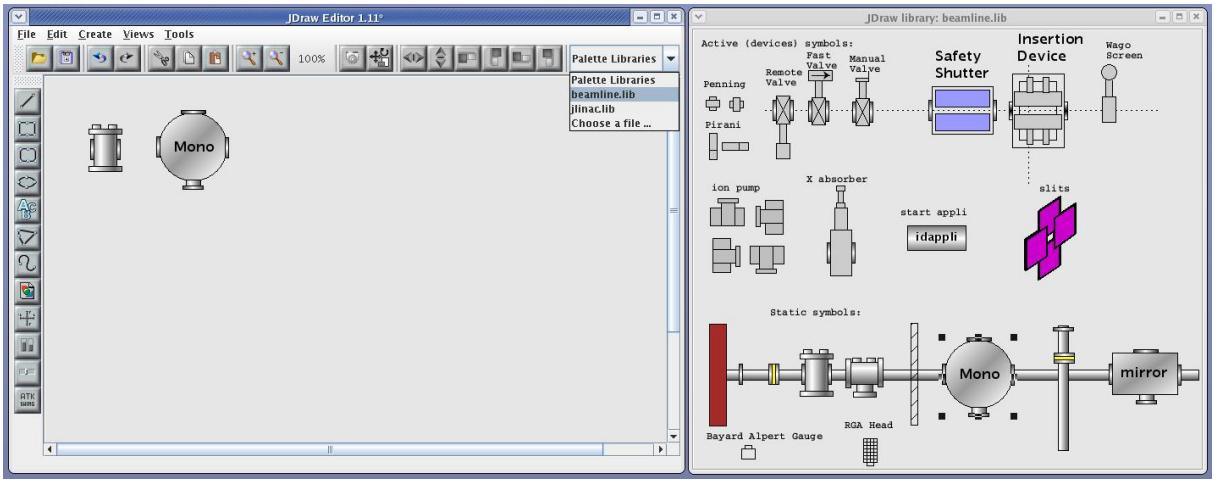
We will not explain in detail Jdraw editor. In fact, it is a very intuitive editor and you can just try it to get experience with it. Neverthelesse there are some Jdraw features which will be described in this section.

Jdraw libraries

You can draw your own standrad shapes and save them in a file such that they can be used in other synoptic files.

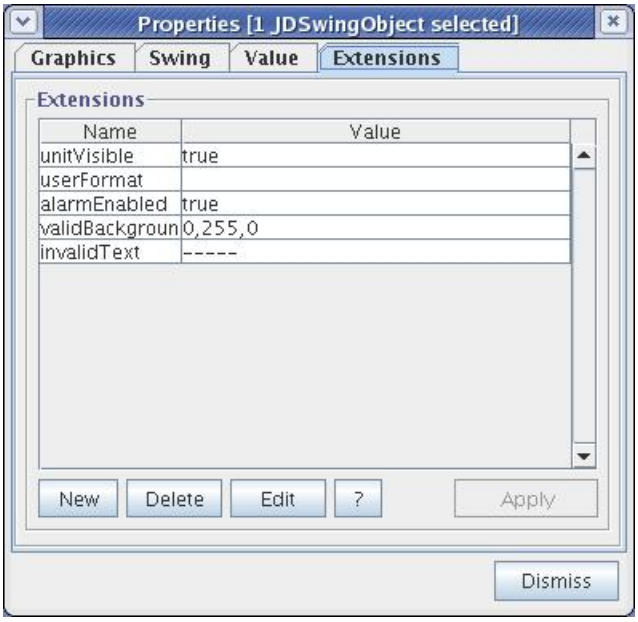
1. Draw your standrard / predefined shapes in the Jdraw editor.
2. Save the file by naming it with a **".lib"** suffix.
3. Move your **".lib"** file to a well defined location on your disk.
4. Set the "LIBPATH" environment variable to the folder where is located your **".lib"** file.
5. Start the jdraw java machine with "-DLIBPATH=$LIBPATH".
6. Note on the top right corner the "Palette Libraries" ComboBox. You should see inside the drop down list the name of the **".lib"** file.

When the library name is selected it's content is displayed in a separate window. You can simply click one component in the library window and click the jdraw window to add it into your drawing. See the screenshot below :



ATK Viewers in Jdraw

A small set of ATK viewers are available in Jdraw so that they can be added inside the synoptic drawing. To add one of them click on "ATK Swing" button and select an appropriate viewer from the list.When using the ATK viewers you may need to set some of their "bean properties" to make them behave as you wish. A subset of the properties of each viewer is accessible through jdraw. To see and to edit those properties, double click the atk viewer, then select the "Extension" tab in the property window. For example in the screen shot below you can see all the bean properties available in Jdraw for the SimpleScalarViewer.



Dynamic Objects (Dynos)

Dynamic Objects also called Dynos in Jdraw are the graphic components for which the user has defined a dynamic (run-time) behavior.

For example a Dyno can be any graphic component associated to a numeric tango attribute and for which the user has defined a specific background color depending on the value of the attribute. You can see how to create and use the dynos in Jdraw in the following flash demo.

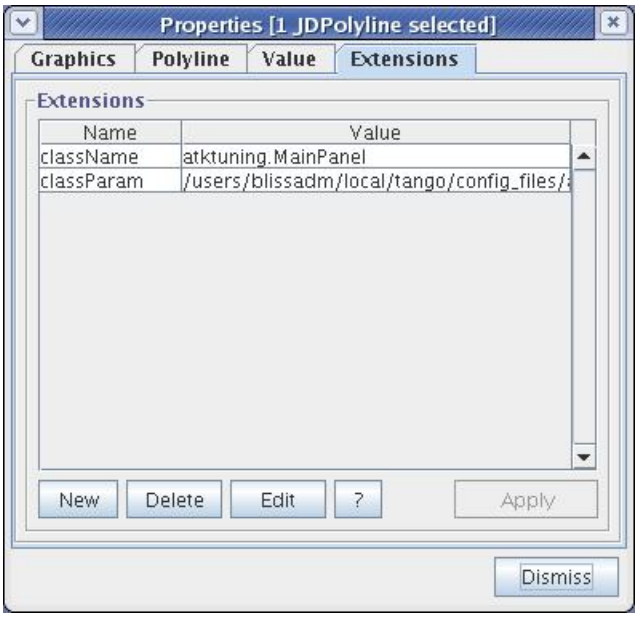
Dynamic Objects in Jdraw (Flash Demo)

Panel class definition

In a synoptic application when a graphic component is clicked by the user, in most cases, we need to launch a specific panel. In Jdraw you have the possibility to define the name of the class you want to start when the Jdraw object is clicked. To associated a Jdraw graphic component to a panel follow the steps below :

1. Double click the Jdraw graphic object to show the **Properties** window
2. Select the **Extension** Tab inside the Properties window
3. Click on the **"New"** button to add a new extension and give it the name **"className"**
4. Type in the fully defined class name of the panel you want to show, in the value field attached to **className** extension

5. Optionnally click on the **"New"** button to add another extension and give it the name **"classParam"**
6. Type in the string which is passed to the constructor of the panel class



The "panel class" defined with "className" extension :

- Should be a subclass of JFrame or Jdialog
- Must have a constructor with a String parameter (even if the parameter is ignored)
- Should not call system.exit() when it's window is closed

Conclusion

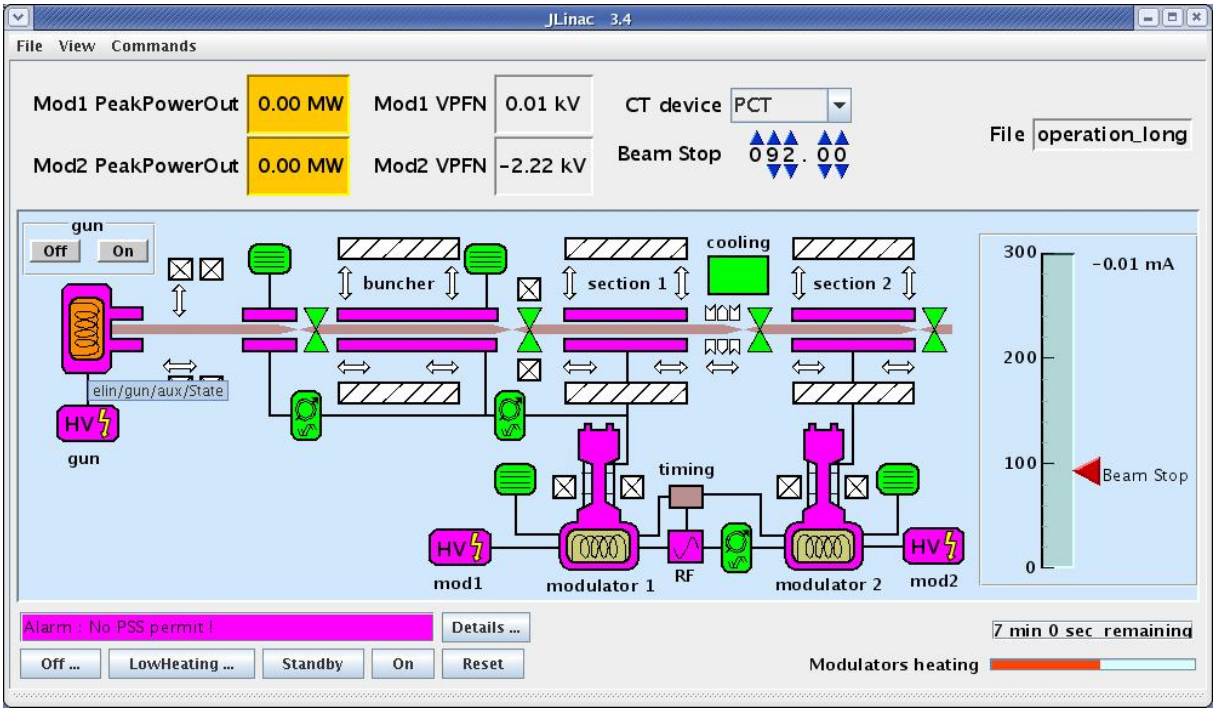
- The specific full qualified panel class name should be specified in the **"Extension"** tab of the property window under the name of **"className"**.
- The string parameter which will be passed to the panel class constructor can also be defined in the **"Extension"** tab of the property window under the name of **"classParam"**.
- If the **classParam** is not defined the constructor of the class is called with the name of the jdraw graphic object which has been clicked.
- If the **className** is not defined and the jdraw graphic object is associated to a Tango state attribute, **atkpanel** will be started in read only mode.

Include the Synoptic in an ATK application

Once the synoptic is drawn and well tested, it can be used through the generic application **SimpleSynopticAppli**. To launch the **SimpleSynopticAppli** start the following class :

fr.esrf.tangoatk.widget.jdraw.SimpleSynopticAppli

In most cases, the synoptic should be integrated inside a specific ATK application in the middle of other ATK viewers.



As you can see in the screen shot the synoptic is only part of the application's main window. There are other ATK attribute and command viewers outside of the synoptic area. Moreover there is also a specific menu bar with a lot of application specific commands.

SynopticFileViewer

ATK provides a viewer called **SynopticFileViewer** which belongs to the package : **fr.esrf.tangoatk.widget.jdraw**. This viewer can be used as any other ATK viewer. It can be added into any Swing container. It can also be added to a Java IDE palette (for example Netbeans palette) as a Java Bean.

Once the SynopticFileViewer is instantiated, the programmer should specify the synoptic file to be loaded by the viewer. There are two methods for synoptic file specification :

1. Load the synoptic from a file specified by a path name on the disk,
2. Load the synoptic from an Input Stream Reader.

Load the synoptic from a file

The application programmer will specify the file path name of the synoptic file to load. The drawback of this option is that the application programmer must know the absolute path name of the synoptic file and this path name is constant even if the application is deployed in different hosts and sites.

The following code sample shows how to use a SynopticFileViewer and specify the synoptic file to load:

```
SynopticFileViewer  sfv = new SynopticFileViewer();

sfv.setToolTipMode(TangoSynopticHandler.TOOL_TIP_NAME);
sfv.setAutoZoom(true);
Try
{
```

(continues on next page)

(continued from previous page)

```
    sfv. setJdrawFileName ("/my/root/dir/jdraw_file mySynoptic.jdw ");
}
Catch (Exception ex) {}
```

The call to **"setJdrawFileName"** will load the synoptic file if it can be found and opened, otherwise an exception is thrown.

Load the synoptic from an Input Stream Reader

The main advantage of this method is that the synoptic jdraw file can be included into the application jarfile. An input stream reader is created through the file resource by the application code. This input stream reader is passed to the SynopticFileViewer to load the synoptic.

This option allows that the synoptic file is packed inside the application Jar file and we don't make any assumption on the exact physical location of the synoptic file on the disk.The following code sample shows how to use a SynopticFileViewer and specify the synoptic file to load:

```
SynopticFileViewer  sfv = new SynopticFileViewer();
sfv.setToolTipMode(TangoSynopticHandler.TOOL_TIP_NAME);
sfv.setAutoZoom(true);
InputStreamReader   inStrReader=null;
InputStream   jdFileInStream = this.getClass().getResourceAsStream("/mypakcage/file.jdw
↪");
if (jdFileInStream!=null)
    inStrReader = new InputStreamReader(jdFileInStream);
if (inStrReader!=null)
{
    try
    {
       sfv .loadSynopticFromStream(inStrReader);
    }
    catch (Exception ex) {}
}
```

The call to **"loadSynopticFromStream"** will load the synoptic from the input stream if possible. In case of bad format or an empty stream (no component) an exception is thrown.

Predefined run time behavior

The synoptic file is loaded by ATK at run-time. All the run time animation / behavior is coded inside Atk class which loads the synoptic. All of the run time behavior is listed in this section.

Attribute

A Tango state attribute can be associated to any jdraw graphic object. From a simple drawing to a complex shape made of successive groups. A tango state attribute can also be associated to a Dyno.

Associated to a Jdraw Object (not a Dyno)

ATK will color the object according to the value of the state attribute. The state/color mapping is the same as the one used in all other parts / viewers of ATK.

- If the object is filled : the fill color is changed
- If the object is not filled : the line color is changed
- If the object is made of successive groups, the change is made recursively in each group util the basic graphic objects are reached. In this hierarchy of objects, the graphic objects whose name is **"IgnoreRepaint"** do not change their color at all.

Associated to a Dyno (Dynamic Object)

As described in the previous section a Dynamic Object (Dyno) has a specific dynamic behavior which has been defined during the drawing phase. In order to define your own behavior with a Dyno associated to a State attribute, you should define the mapping between each different tango state numeric values and the caracteristique affected by the value.

It's important to know that the Dyno will receive at run time a numeric value associated to the state attribute value. You can find the mapping between the numeric values and the tango state values in the Tango documentation:

User interaction

When the **mouse enters** the graphic component associated to the state attribute, the name of the state attribute is displayed inside a **tooltip**.

When an object associated to a state attribute is clicked by the user at run time, ATK tries to popup a panel.

- If the **"className"** extension is defined, the class is instantiated using a constructor with a String parameter.
- If the **"className"** extension is not defined the AtkPanel in read-only mode is instantiated.
- If the **"classParam"** extension is defined, the string is passed as the argument to the constructor of the panel class.
- If the **"classParam"** extension is not defined, the device name behind the state attribute is passed as the argument to the constructor of the panel class.

Tango Numeric Attribute

A Tango numeric attribute can be associated to a Dyno (Dynamic Object) or to an adapted Atk Viewer (for example SimpleScalarViewer).

Associated to a Dyno (Dynamic Object)

As described in the previous sections a Dynamic Object (Dyno) has a specific dynamic behavior which has been defined during the drawing phase. In order to define your own behavior with a Dyno associated to a tango numeric attribute, you should define the mapping between different values of the tango attribute and the caracteristique affected by the

value. You can for example associate "value intervals" to a caracteristique change. See the Dynamic Objects in Jdraw (Flash Demo).

It's important to know that the Dyno will receive at run time the numeric value of the tango attribute when it changes.

Associated to an Atk Viewer (for example SimpleScalarViewer)

The tango attribute will be set as the model of the AtkViewer (SimpleScalarViewer) and that's it. All the run-time behavior is defined by the AtkViewer which is used.

Some of the bean properties of the Atk Viewer are available in the extension Tab of the Jdraw properties window.

User interaction

When the **mouse enters** the graphic component associated to the tango numeric attribute, the name of the tango attribute is displayed inside a **tooltip**.

When an object associated to the tango attribute is selected by the user at run time, ATK tries to popup a panel :

- If the **"className"** extension is defined, the class is instantiated using a constructor with a String parameter.
- If the **"classParam"** extension is defined, the string is passed as the argument to the constructor of the panel class.
- If the **"classParam"** extension is not defined, the name of the Jdraw object (which is the name of the Tango numeric attribute) is passed as the argument to the constructor of the panel class.
- If the **"className"** extension is not defined nothing happens

Tango Boolean Attribute

A Tango boolean attribute can be associated to a Dyno (Dynamic Object) or to an adapted Atk Viewer (for example BooleanScalarCheckboxViewer).

Associated to a Dyno (Dynamic Object)

In order to define your own behavior with a Dyno associated to a tango boolean attribute, you should define the mapping between the two values of the boolean attribute (true and false) and the caracteristique affected by the value. See the Dynamic Objects in Jdraw (Flash Demo).

It's important to know that the Dyno will receive at run time the numeric value for the boolean attribute. It means that if the attribute value is false, the value 0 is sent to the Dyno and if the attribute value is true the value 1 is sent to the Dyno.

Associated to an Atk Viewer (for example BooleanScalarCheckboxViewer)

The tango attribute will be set as the model of the AtkViewer (*BooleanScalarCheckboxViewer*). All the run-time behavior is defined by the AtkViewer which is used.

Some of the bean properties of the Atk Viewer are available in the extension Tab of the Jdraw properties window.

User interaction

When the **mouse enters** the graphic component associated to the tango numeric attribute, the name of the tango attribute is displayed inside a **tooltip**.

When an object associated to the tango attribute is selected by the user at run time, ATK tries to popup a panel :

- If the **"className"** extension is defined, the class is instantiated using a constructor with a String parameter.
- If the **"classParam"** extension is defined, the string is passed as the argument to the constructor of the panel class.

- If the **"classParam"** extension is not defined, the name of the Jdraw object (which is the name of the Tango boolean attribute) is passed as the argument to the constructor of the panel class.
- If the **"className"** extension is not defined nothing happens

Tango DevState Spectrum Attribute

An element of a Tango DevState spectrum attribute can be associated to any jdraw graphic object. From a simple drawing to a complex shape made of successive groups. An element of a Tango DevState spectrum attribute can also be associated to a Dyno. To assign an element of a DevState spectrum attribute we use the brakets. So to associate the 10 th element of the state spectrum attribute sr/rf-tra/tra1/SubDevicesStates, the name of the graphic component should be **sr/rf-tra/tra1/SubDevicesStates[9]**.

Associated to a Jdraw Object (not a Dyno)

ATK will color the object according to the value of the element specified in the state spectrum attribute. The state/color mapping is the same as the one used in all other parts / viewers of ATK.

- If the object is filled : the fill color is changed according to the state value
- If the object is not filled : the line color is changed according to the state value
- If the object is made of successive groups, the change is made recursively in each group util the basic graphic objects are reached. In this hierarchy of objects, the graphic objects whose name is **"IgnoreRepaint"** do not change their color at all.

Associated to a Dyno (Dynamic Object)

As described in the previous sections a Dynamic Object (Dyno) has a specific dynamic behavior which has been defined during the drawing phase. In order to define your own behavior with a Dyno associated to an element of a State spectrum attribute, you should define the mapping between each different tango state numeric values and the caracteristique affected by the value.

It's important to know that the Dyno will receive at run time a numeric value associated to the state attribute value. You can find the mapping between the numeric values and the tango state values in the Tango documentation:

User interaction

When the **mouse enters** the graphic component associated to the state spectrum attribute, the name of the state spectrum attribute + index of the element in the spectrum is displayed inside a **tooltip**.

When an object associated to a state attribute is clicked by the user at run time, ATK tries to popup a panel.

- If the **"className"** extension is defined, the class is instantiated using a constructor with a String parameter.
- If the **"classParam"** extension is defined, the string is passed as the argument to the constructor of the panel class.
- If the **"classParam"** extension is not defined, the name of the Jdraw object (which is the name of the element of a tango DevState spectrum attribute) is passed as the argument to the constructor of the panel class.
- If the **"className"** extension is not defined nothing happens

Tango Command

A Tango Command can be associated to a Jdraw interactive component or to an adapted Atk Viewer (for example VoidVoidCommandViewer).

Associated to a Jdraw interactive component

When the interactive graphic component is clicked, the tango command is executed.

Associated to an Atk Viewer (for example VoidVoidCommandViewer)

The tango attribute will be set as the model of the AtkViewer (VoidVoidCommandViewer). All the run-time behavior is defined by the AtkViewer which is used.

Some of the bean properties of the Atk Viewer are available in the extension Tab of the Jdraw properties window.

User interaction

When the **mouse enters** the graphic component associated to the tango command, the name of the tango command is displayed inside a **tooltip**.

When the interactive object associated to the tango command is clicked by the user at run time, ATK sends the command to the associated Tango device.

Other types of Tango Attributes

Other type of Tango attributes can be associated only to an Atk viewer avialable in Jdraw editor under the "Atk Swing" button. They cannot be associated to a Jdraw graphic component. The use of an Atk viewer is mandatory.

The following tango attributes can be used in Jdraw and associated to their corresponding Atk viewers as listed below :

- **String Scalar** attribute should be associated to a **SimpleScalarViewer**
- **Numeric Spectrum** attribute should be associated to a **NumberSpectrumViewer**
- **Numeric Image** attribute should be associated to a **NumberImageViewer**

The run time behavior is the one provided by the Atk viewer.

7.2.6 Tango Application Toolkit "ATK"

Intended audience: developers, ***Programming language:*** java

Tango Application ToolKit, (ATK) is a TANGO client framework for building GUIs based on Java Swing.

Speeding up the development and standardizing look and feels ATK provides several swing based components to view and/or to interact with Tango device attributes and Tango device commands and also a complete synoptic viewing system.

Implementing the core of "any" Tango application ATK takes in charge the automatic update of device data either through Tango events or by polling the device attributes. ATK takes also in charge the error handling and display.

The ATK swing components are the Java Beans, so they can easily be added to a Java IDE (like NetBeans) to speed up the development of graphical control applications.

ATK is composed of two jarfiles ATKCore.jar and ATKWidget.jar.

You can get them from the ATK download page.

Contents:

Tango ATK Tutorial

Intended audience: developers, ***Programming language:*** java

This document is a practical guide for Tango ATK programmers and includes several trails with examples and demonstrations. Most of the examples and demonstrations are provided as Macromedia Flash documents.

In this document we assume that the reader has a good knowledge of the Java programming language, and a thorough understanding of object-oriented programming. Also, it is expected that the reader is fluent in all aspects regarding Tango devices, attributes, and commands.

Before going through the trails and examples, the Tango ATK architecture and key concepts are introduced. After this introduction the rest of the document is organized in a set of trails.

Introduction

Tango Application Toolkit also called "ATK" is a client framework for building applications based on Java Swing in a Tango control system.

Goals of Tango ATK

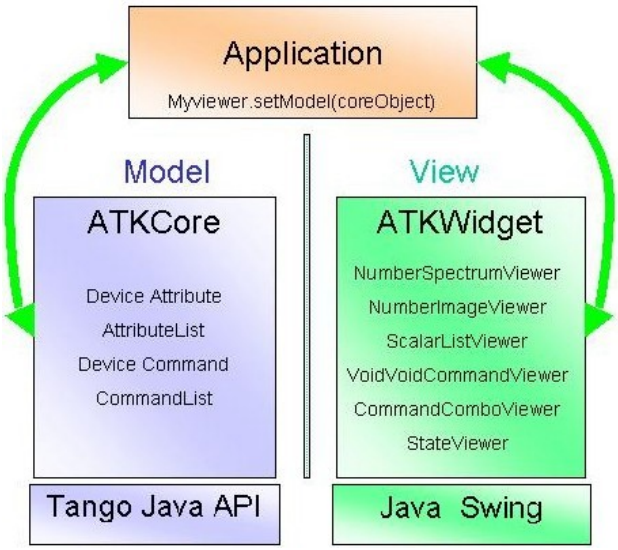
The main goals of ATK are the following:

1. Speeding up the development of Tango graphical clients.
2. Standardizing the look and feel of Tango applications.
3. Implementing the core of "any" Tango application.

To achieve the first and the second goals ATK provides several swing based components to view and/or to interact with Tango device attributes and Tango device commands and also a complete synoptic viewing system. To achieve the third goal ATK takes in charge the automatic update of device data either through Tango events or by polling the device attributes. ATK takes also in charge the error handling and display. The ATK swing components are the Java Beans, so they can easily be added to a Java IDE (like NetBeans) to speed up the development of graphical control applications.

The Software Architecture of Tango ATK

Tango ATK is developed using the Model-View-Controller design pattern also used in the Java Swing package. The Tango basic objects such as device attributes and device commands provide the model for the ATK Swing based components called viewers. The models and the viewers are regrouped in two separate packages respectively ATKCore and ATKWidget.



Note: ATK is based on Swing. Mixing the use of other "non swing" objects such as SWT (eclipse) with ATK is not recommended.

Note: ATK hides Tango Java API (TangORB). It is highly recommended not to use TangORB methods and objects directly in the application code. Always use the interface provided by ATK to access the control system

The key concepts of Tango ATK

Reminder: the central Tango component is the DEVICE. The Tango control system can be seen as a collection of devices distributed over the network. The tango devices provide attributes (for reading and setting data) and commands.

The central ATK components, to access the Tango control system, are: attributes and commands and not the devices. Through Tango Java API (TangORB) the control system is a collection of devices where through ATK the control system is a collection of attributes and commands.

In addition to ATK attributes and ATK commands ATK provides two components, which are ATK attribute lists and ATK command lists.

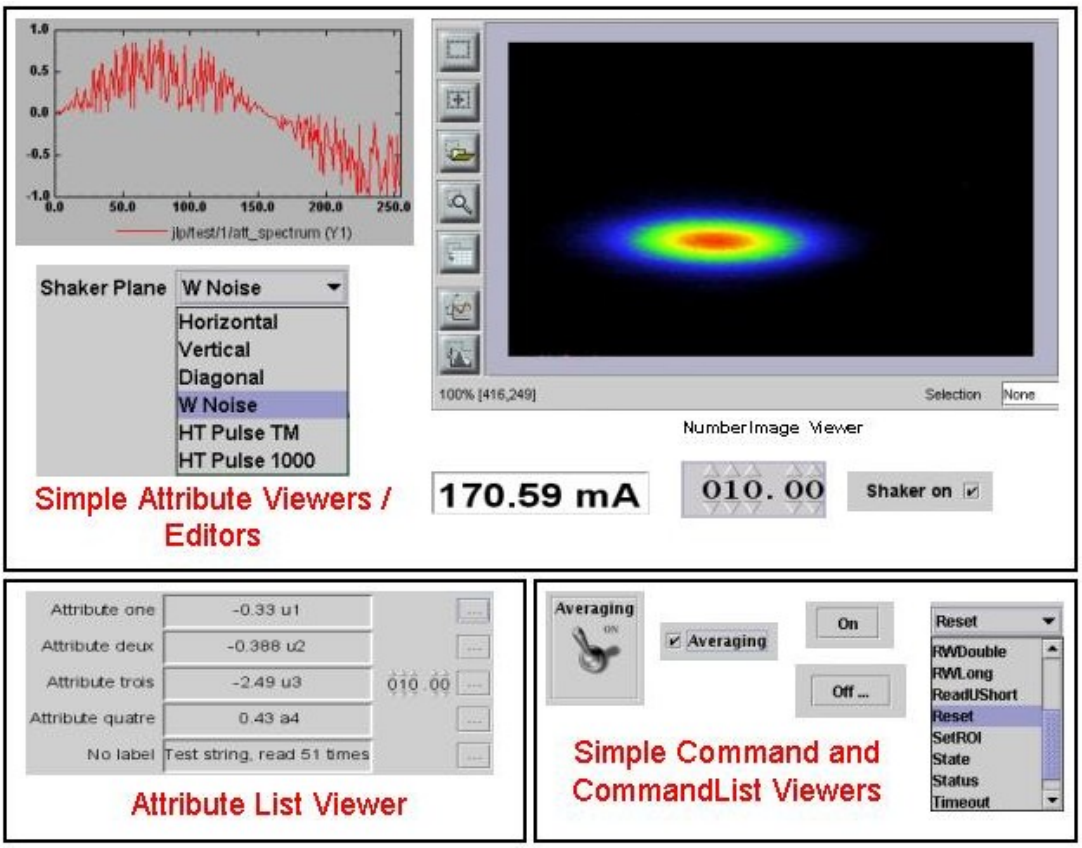
The central ATK components are:

1. ATK Attribute (interface to Tango device attribute),
2. ATK Command (interface to Tango Device command),
3. ATK AttributeList (collection of ATK Attributes)
4. ATK CommandList (collection of ATK Commands).

Tango ATK viewers

ATK viewers are provided as Java Beans and as such they can easily be added in a Java IDE (like NetBeans) to speed up the development of the graphical applications. This way the programmer can easily build up his (her) panels, mixing pure Swing objects and Tango ATK viewers.

ATK viewers are provided for different types of Tango Components. They can be divided into different categories such as: error history window, error popup window, simple attribute viewers / editors, attribute list viewers, simple command viewers, command list viewers, ...etc.



Synoptic drawing and viewing

A synoptic is a drawing in which each object can be linked to a Tango object. A part of the synoptic drawing can be linked to the state attribute of a Tango device where another part is associated to a numerical attribute of another Tango device. The main idea of synoptic drawing and viewing system is to provide the application designer with a simple and a flexible way to draw a synoptic and to animate it at runtime according to the values and states read from the control system. ATK provides two components for this purpose:

- A graphical editor called "Jdraw". This tool is used during the design phase to draw and to specify the synoptic. The synoptic is saved to a file.
- A synoptic viewer called "SynopticFileViewer". This viewer is used in the graphical user interface of the application. SynopticFileViewer loads and browses the synoptic drawing file and animates its elements at runtime according to their state or to their value.

Getting Started

The following short tutorial takes you through some of the basic steps necessary to develop a Tango Java application based on Tango ATK.

In this tutorial we don't use any Java IDE features. All the java code is entered manually using a java source editor. The NetBeans java source editor is used as any source editor.

Let's specify the application we want to build in terms of the Model-View-Controller design pattern described before.

Our "Getting Started" application will need to show two tango device attributes and one tango device command all related to the same device. The tango device name used in this tutorial is *"jlp/test/1"*. The application will show the "state" and the "att_spectrum" attributes of this device and will give access to its "Init" command.

1. The type of the "state" attribute (*jlp/test/1/state*) is "DevState" and its format is "Scalar"
2. The type of the "att_spectrum" attribute (*jlp/test/1/att_spectrum*) is "DevDouble" and its format is "Spectrum"
3. The "Init" command (*jlp/test/1/Init*) has no input and no output argument (input and output argument types are DevVoid)

The ATKCore components are used to create and initialise the "model" part of the design pattern:

1. One attribute list,
2. Two attributes (*jlp/test/1/state*, *jlp/test/1/att_spectrum*),
3. One command list,
4. One command (*jlp/test/1/Init*).

The ATKWidget components are used to create and initialise the *"view"* part of the design pattern. These components are the objects adapted to the type of the tango component we want to visualize. They are also called **"viewers"** (attribute viewers, command viewers, … etc.).

1. One State viewer (a viewer adapted to the DevState Scalar attributes)
2. One NumberSpectrum viewer (a viewer adapted to any numerical spectrum attribute)
3. One VoidVoidCommand viewer (a viewer adapted to the any command with no input and no output argument).

The controller part consists of making the relationship between the *"view"* components and the *"model"* components. Calling the *"setModel"* method of the view object makes this relationship. For example the call *"stateViewer.setModel(stateAtt)"* will make the relationship between these two objects.

Click on the following link to view a Flash demo of how to build the *"GettingStarted"* application.

Getting Started (Flash Demo)

The Structure of an ATK application

Any ATK application should perform a minimum set of operations. The following lists this minimum set of operations:

1. Declaration and initialization of ATKCore objects (AttributeLists, CommandLists, individual ATKCore attributes and individual ATKCore commands).
2. Declaration and instantiation of ATKWidget Error viewers to handle errors
3. Connection to attributes and commands by adding them to the appropriate list
4. Creation of the specific Attribute and command viewers, and add them to a swing window
5. Associate each viewer to an appropriate ATKCore attribute or command
6. Start the refresher thread associated to the attribute list

The following slide show will present in detail the skeleton of an ATK application :

ATK application skeleton (Flash Slide Show)

Using ATK inside a Java IDE (NetBeans)

Several Java IDEs (Integrated Design Environments) are available on the market and also as freeware. You can search the Internet to choose the most appropriate one for your usage. Here you can find some links to start with:

NetBeans (free download)

Eclipse (free download)

Intelligent Idea (commercial tool)

The use of the Java IDEs especially those including a good graphical user interface builder speed up the development of Tango ATK applications. From now on all the examples in this tutorial are made using the NetBeans 5.5 or 6.1, Java IDE. The present section presents the manner in which the ATK Java Beans can be integrated to the NetBeans Palette and used to build the user interface of the final ATK application.

If you are using another Java IDE please refer to its documentation to find out how to integrate and use the ATK Java Beans inside the IDE, to build a graphical user interface.

Learning NetBeans

You should first download and install the NetBeans IDE from NetBeans Web site. HIf you have never installed JDK on your computer or if the JDK on your computer is out of date, you may consider to install the bundle NetBeans+JDK depending on the version of NetBeans and JDK you wish to install. For example inside the download page of NetBeans you can find :

NetBeans IDE 6.1 with JDK 5.0 Bundle

This download will install JDK 1.5 and NetBeans 6.1 in a single operation.

Once the NetBeans is installed you can browse: NetBeans Tutorials, Guides, and Articles which can help you, learn more about NetBeans.

If you are a beginner with NetBeans we recommend you to go through the following quick start guides:

Guided Video Tour of NetBeans IDE

Create an ATK Application project in NetBeans

To create and ATK application project, you may go through the following steps:

1. Create the NetBeans Java Application Project,
2. Add the Tango and ATK jar files in the project 's class path,
3. Add several ATK Java Beans (ATK viewers) to the NetBeans palette.

ATK application project using NetBeans (Flash Demo)

First ATK Simple GUI application

Now we build a simple Tango GUI using ATK viewers, which have been added to the NetBeans palette. Here are the steps to follow to build this GUI:

1. Create a source package,
2. Create a new Jframe form in this package,
3. Add ATK beans from the palette into the form and place and resize them as you wish,
4. Add the necessary source code to create and to initialize the ATKCore (model) objects,
5. Associate each viewer with its model,
6. Start the ATK refresher(s),
7. Build and run this GUI.

First ATK Simple GUI appliction (Flash Demo)

ATK Quick Tour

This section includes the first list of tutorials, which give you a quick tour of the Tango ATK components by guiding you through the creation of a simple generic application very similar to AtkPanel. During this quick tour you will learn how to view device state and status attributes, and how to display a collection of tango scalar attributes all aligned with each other. You will also use a viewer to display a collection of tango device commands.

Device state and device status

The state and the status of the device are two attributes of any Tango device (IDL 3 and above). Atk provides two attribute viewers one called **StateViewer** and the other **StatusViewer** to display them. These viewers are included in the **fr.esrf.tangoatk.widget.attribute** package.

The model for the **StateViewer** is the state attribute (DevStateScalar) and the model for the **StatusViewer** is any scalar attribute of type String (StringScalarAttribute).

You can go through the following simple demo to see how to use these two viewers.

State and Status viewers (flash demo)

Display a list of scalar attributes

The ATK attribute list viewers / setters are provided to be able to display a collection of attributes all aligned together. In fact, the ATK attribute list viewers handle only scalar attributes. An attribute list **viewer's model is an attribute list**. This means the model for this type of viewers cannot be an individual attribute and should be an attribute list. The attribute list viewers are all included in the **fr.esrf.tangoatk.widget.attribute** package.

The ATK list viewers provide the application with three major advantages:

- The first advantage is that all the single attribute viewers are aligned in a coherent manner inside the attribute list viewer.
- The second advantage is that the application can be "generic". An application program with no knowledge of the exact names and types of the scalar attributes of a particular device, can display all of them easily with two lines of code.

- The third advantage is that the application programmer does not need to know which type of attribute viewer is adapted to which type of tango attribute. The ATK list viewers automatically select the adapted viewer and / or setter for each type of device attribute.

There are three classes for attribute list viewing:

- **ScalarListViewer**,
- **NumberScalarListViewer**,
- **ScalarListSetter**.

The ScalarListViewer and NumberScalarListViewer are almost the same. The only difference is that the NumberScalarList viewer will display only the scalar attributes which are numerical where ScalarListViewer will display also StringScalar attributes, BooleanScalar and EnumScalar attributes in addition to the numerical scalar attributes.

The attributes, members of the attribute list are displayed vertically. In each line an individual attribute is displayed in the following manner:

1. At the left the "label" property of the tango attribute,
2. Next to the label the "read" value of the attribute is displayed according to the "format" and the "unit" properties of the tango attribute,
3. In the third column the "setpoint" of the tango attribute is displayed inside a viewer (mostly called editor), which allows setting the attribute value,
4. The last (forth) column is used to display a pushbutton with three dots. A click on this pushbutton pops up a window called "SimplePropertyFrame". In this window the user can modify any property of the tango attribute such as: label, min alarm, max alarm, unit,..etc.

The application programmer can easily hide any three columns among four. There is always one column, which cannot be hidden.

- **ScalarListViewer**: three columns, which can be hidden, are label, setPoint editor (setter), and property button. The "read" value column cannot be hidden. All the attributes, members of the Attributelist model should be scalar attributes. All attributes with another format (Spectrum) will be ignored.
- **NumberScalarListViewer**: three columns, which can be hidden, are label, setPoint editor (setter), and property button. The "read" value column cannot be hidden. All the attributes, members of the Attributelist model should be scalar and numerical. All attributes with another type (String) and / or format (Spectrum) will be ignored.
- **ScalarListSetter**: three columns, which can be hidden, are label, "read" value, and property button. The setPoint editor (setter) column cannot be hidden. All the attributes, members of the attributeList model must be scalar and writable. The read-only attributes members of the attributeList model are ignored

ScalarListViewers and ScalarListSetters (Flash Demo)

View a list of device commands

There is only one class provided for the command list viewing: **CommandComboViewer.** This viewer is based on the Swing "JComboBox".The user can select any of the commands displayed in the list and send it to the device. The selection of an item in this list leads to the execution of the device command.

The viewers studied above (StateViewer, StatusViewer, ScalarListViewer and CommandComboViewer) can be used to build a generic tango device panel.

A generic tango device panel

The application we try to build in this tutorial is a generic tango device panel, which displays all the U **scalar** U attributes (no spectrum attribute, no image attribute) of a device and gives access to all commands of the same device. The application is generic because it has no knowledge of the attribute names and command names of the device.

The device name should be passed as a parameter through the class constructor so that this panel can be used for any Tango device.

The ATK viewers we will use for this exercise are:

1. **StateViewer** (fr.esrf.tangoatk.widget.attribute.StateViewer),
2. **StatusViewer** (fr.esrf.tangoatk.widget.attribute.StatusViewer),
3. **ScalarListViewer** (fr.esrf.tangoatk.widget.attribute.ScalarListViewer),
4. **CommandComboViewer** (fr.esrf.tangoatk.widget.command.CommandComboViewer).

The two last viewers are so-called "list viewers". It means that, their corresponding model should not be an individual attribute or an individual command. Their corresponding model should be respectively an attribute list and a command list.

Generic single device panel (Flash demo)

ATK Guided Tour

In this chapter you will study the essential components of the ATK starting with the simplest ones used to visualize individual tango attributes and / or tango commands. The final part of this chapter is dedicated to the synoptic system provided with ATK. You can study this chapter in any order.

Scalar attributes

A scalar attribute is a Tango attribute whose format is Scalar whatever the data type of the attribute. In this chapter we will see how to view and / or set a single scalar attribute. We will also see how to view a collection of scalar attributes.

One single scalar attribute

Use a generic scalar attribute viewer (used to view and / or to set)

This solution consists of using the same viewer for any type of scalar attributes (number, string, boolean). The attributeList viewers such as ScalarListViewer can be used to view a single scalar attribute. All you have to do is to build an attributeList in which you add only one single scalar attribute, which is the one you want to view. Create a ScalarListViewer and set it's model to this attributeList with one single attribute inside. See the code sample below:

```
AttributeList attl = new AttributeList;
try
{
    attl.add("my/test/device/onescalaratt");
    ScalarListViewer slv = new ScalarListViewer();
    slv.setModel(attl);
}
catch ()
{
```

(continues on next page)

(continued from previous page)

```
10
11 }
```

The use of ScalarListViewer even for an individual attribute allows that the attribute value is displayed and formatted with it's unit and eventually accompanied of it's label, a value setter, and a pushbutton to access and to edit the other attribute properties.

Moreover the ScalarListViewer automatically uses the appropriate viewer according to the type of the attribute. For example a BooleanCheckBoxViewer is used for the Boolean attributes and a SimpleScalarViewer is used for numerical and string attributes. For this reason the use of scalarListViewer makes the application code to be independent of the type of the scalar attribute to be displayed.

The ScalarListViewer is used to display the read value of the attribute and also to set the attribute if the attribute is writable.

By hiding one or the other part of the scalarListViewer (label, setter, propertyButton) you can adapt the display to what you really want to make available to the application's user. The screen shots below show the same scalar attribute displayed always with a ScalarListViewer. From left to right, the propertyButton, the setter and finally the label have been hidden.

Using a specific viewer / setter adapted to the attribute type

The use of specific viewers is dependent on the type of the scalar attribute to view and or to set. Normally a specific ATK viewer is designed either to display the read value of the attribute or to set the setPoint value of a writable attribute. But the specific ATK viewer generally does not do both of them. As we have seen before the list viewers (generic attribute viewers) can do both of these two functions read / write.

The specific viewer to use depends on the data type of the attribute and the fact that we want to use it for setting the attribute or only to display the read value. Therefore the source code also depends on the type of the attribute and the viewer. The code sample below is given for a NumberScalar attribute displayed by a SimpleScalarViewer. This code sample can be modified and adapted to other attribute types and viewers or setters.

```
1  AttributeList attl = new AttributeList;
2  try
3  {
4      INumberScalar ins = (InumberScalar) attl.add("my/test/device/oneNumberScalarAtt");
5      SimpleScalarViewer ssv = new SimpleScalarViewer();
6      slv.setModel(ins);
7  }
8  catch ()
9  {
10
11 }
```

Note: When using individual attribute viewers (instead of attribute list viewers) we need to keep a reference to the

scalar attribute ("ins" in the code sample) and use it to set the model of the scalar attribute viewer.

The code sample above has been adapted so that instead of viewing the read value of the attribute we want to set the setPoint value of it.

```
AttributeList attl = new AttributeList;
try
{
    INumberScalar ins = (InumberScalar) attl.add("my/test/device/oneNumberScalarAtt");
    NumberScalarWheelEditor nswe = new NumberScalarWheelEditor();
    nswe.setModel(ins);
}
catch ()
{

}
```

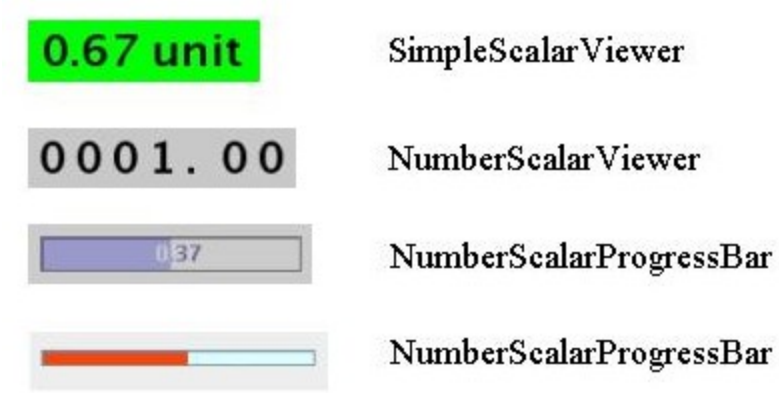
NumberScalar attributes

By number scalar attribute we mean any Tango Attribute whose format is "Scalar" and whose data type is one of the numerical types. No matter if it's a DevLong, DevDouble , or whatever numerical type.

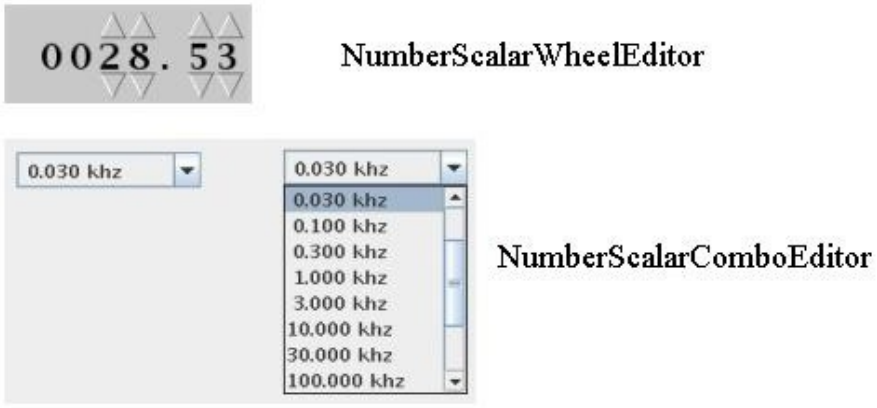
There are several viewers, which can be used to display the "read" value of a Number Scalar attribute. There are also several classes in ATK provided for setting the value of a number scalar attribute

1. **SimpleScalarViewer** : can be used to display the read value of a NumberScalar or a StringScalar attribute. The value of the NumberScalar attribute is formatted according to the "format" attribute property. The attribute value is displayed followed by it's unit (the attribute property unit). This viewer is actually the one used by ScalarListViewer to display the value of any Number or String scalar attribute.
2. **NumberScalarViewer** : can be used to display the read value of a NumberScalar. This viewer has a different character spacing and does not display the unit.This viewer should be used if you wish to align vertically the read value of a numberScalar attribute with it's setPoint value displayed with a NumberScalarWheelEditor.
3. **NumberScalarProgressBar** : gives a view of the attribute based on a progress bar.
4. **NumberScalarWheelEditor** : displays the setpoint value of a NumberScalar and the user can use the top and bottom arrow buttons to set the NumberScalar attribute value. The value of the NumberScalar attribute is formatted according to the "format" attribute property. The unit is not displayed. This component is the default component used for setting a NumberScalar attribute in ScalarListViewer.
5. **NumberScalarComboEditor** : allows to set the value of a number scalar attribute by selecting the value in a list of predefined possible values. The possible values are formatted according to the "format" attribute property and the unit property is displayed with these values. If a list of predefined possible values are defined for the attribute the ScalarListViewer will automatically use this component instead of the default one (NumberScalarWheelEditor) to set the attribute.

The figure below shows the screen shots of the viewers.



The figure below shows the screen shots for the "setter" classes.

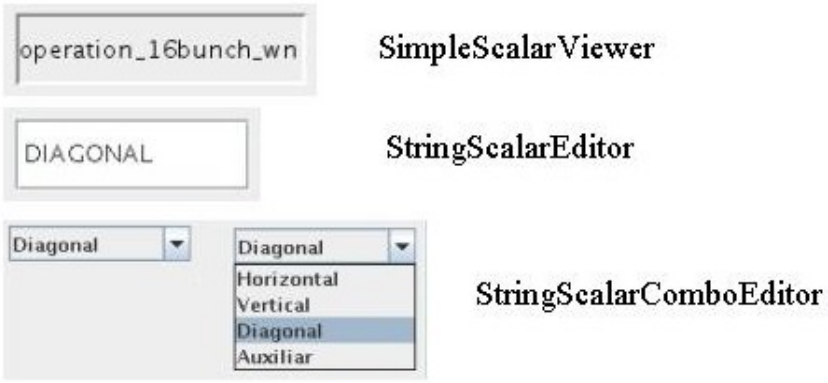


StringScalar attributes

By string scalar attribute we mean any Tango Attribute whose format is "Scalar" and whose data type is DevString.

1. The **SimpleScalarViewer** is used to display the value of a string scalar attribute. This viewer is the one used by ScalarListViewer to display the read value of a string scalar attribute.
2. **StringScalarEditor** : displays the set value of a StringScalar and the user can type inside the text field to set the value of the StringScalar attribute. This component is the default component used for setting a StringScalar attribute in ScalarListViewer.
3. **StringScalarComboEditor** : allows to set the value of a StringScalar attribute by selecting the value in a list of predefined possible values. If a list of predefined possible values are defined for the attribute the ScalarListViewer will automatically use this component instead of the default one (StringScalarEditor) to set the attribute.

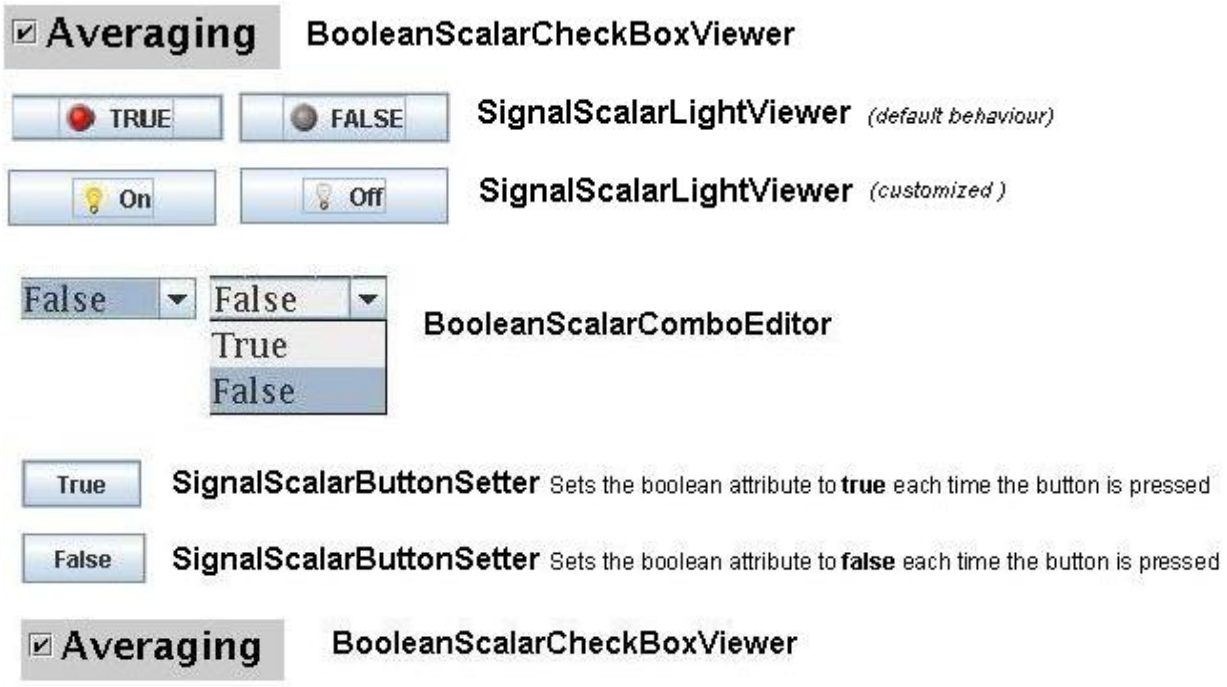
The figure below shows the screen shots for the "viewer" and "setter" components provided for StringScalar attributes.



BooleanScalar attributes

By boolean scalar attribute we mean any Tango Attribute whose format is "Scalar" and whose data type is DevBoolean.

1. **BooleanScalarCheckBoxViewer** is used to view and to set the value of a boolean scalar attribute. In fact the BooleanScalarCheckBoxViewer is a mixed component. It's a viewer and a setter. This component is used in ScalarListViewer to display the the read value of the Boolean attributes.
2. **SignalScalarLightViewer** is used to display the read value of a Boolean Scalar attribute.
3. **BooleanScalarComboEditor** : this component is the default component used in ScalarListViewer to set a boolean attribute. This component refreshes it's view according to the change in the "setpoint" value of the boolean attribute.
4. **SignalScalarButtonSetter** : this component is a pushbutton which is used to set the value of a boolean attribute always to the same value. The value (true or false) which is sent to the attribute at each click on the pushbutton is defined when the component is instantiated.



EnumScalar attributes

The Enumerated attributes will be available within the future releases of Tango but for the time being, Tango does not provide such a feature.

Nevertheless under some conditions ATK provides the possiblity to see some numeric and scalar attributes as enumerated attributes.

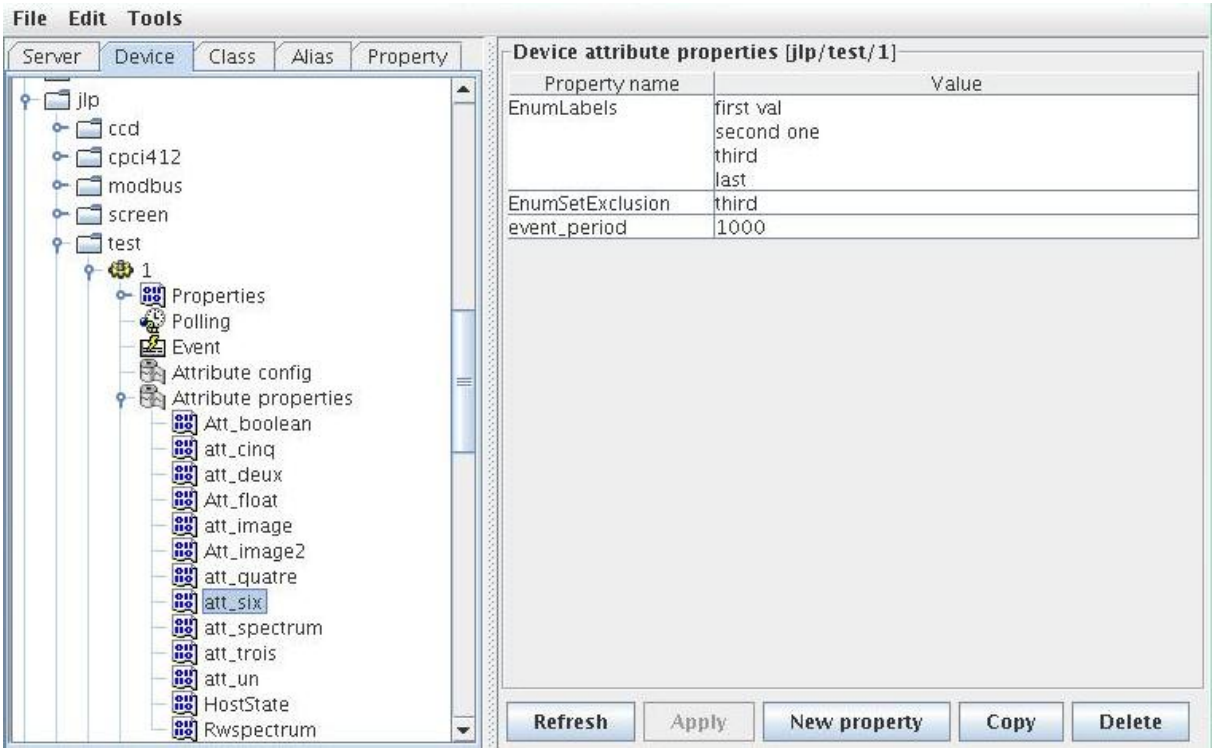
The condition for mumeric scalar attributes to be considered as enumerated scalar attributes (EnumScalar) is :

- The attribute data type should be *DevShort*.
- A property whose name is " *EnumLabels* should be defined for the attribute.
- Eventually (it is optionnal) another property whose name is U *EnumSetExclusion* U can also be defined for the attribute

The first property (*EnumLabels* U) specifies the list of all the possible values the attribute can have. This list is an ordered list. Each label in the list correponds to a numeric value. The first label is always associated to zero (0).

The second property (*EnumSetExclusion*) if specified, gives the list of labels, which can never be used to set the attribute. The labels specified by this property are possible values the attribute can have when we read it but they can not be used as possible set values. If this property is not specified, all the values / labels specified in U *EnumLabels* U, can be used to set the attribute value.

In the screen shot below, you can see how a DevShort scalar attribute (*jlp/test/1/att_six*) can be configured using JIVE such that ATK considers it as an enumerated attribute :



In this example the possible values for *jlp/test/1/att_six* are 0, 1, 2, and 3 respectively associated to "first val", "second one", "third" and "last". Note that the value "third"=2 can be read from the attribute but can never be used to set the attribute.

The **SimpleEnumScalarViewer** is used to display the read value of a enumerated scalar attribute. This component is used by the ScalarListViewer to view the enumerated attributes. **Do not forget that enumerated attribute is an ATK concept and in Tango the real type of the attribute is DevShort.**

The **SimpleEnumScalarViewer** reads the value of the attribute and displays the "label" corresponding to the read value. This label is one of those specified by the property U *EnumLabels* U associated to the attribute.

The **EnumScalarComboEditor** is used to set an EnumScalar attribute. This component is used by ScalarListViewer to set the enumerated attributes. This component displays the setPoint value of the attribute converting it to a label specified by the attribute property *UEnumLabelsU*. In the comboBox drop down list all the labels specified by *EnumLabels* property are displayed, excepted those defined in *UEnumSetExclusionU* property.

The picture above shows at the left side a SimpleEnumScalarViewer and at the right side an **EnumScalarComboEditor** both associated with the same attribute *jlp/test/1/att_six*. As you can see the label "third" is not proposed in the comboBox drop down list for setting since this label is included in the U *EnumSetExclusion* U property. But if this value (numerical value = 2) is read on the attribute the SimpleEnumScalarViewer on the left side will display "third".

DevState Scalar attributes

By DevState scalar attribute we mean any Tango attribute whose format is "Scalar" and whose data type is DevState. The " *StateViewer* " is one of the viewers used to view a DevState scalar attribute. The state is converted to a color by the ATK state viewers. The following color – state correpondance is used by all the ATK viewers:

State	Colour
ON, OPEN, EXTRACT	Green
OFF, CLOSE, INSERT	White
MOVING, RUNNING	Light Blue
STANDBY	Yellow
FAULT	Red
INIT	Beige
ALARM	
DISABLE	Magenta
UNKNOWN	Grey

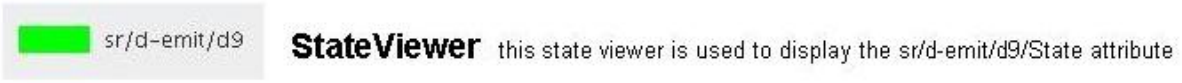
As you can see in the table above the **Open** and **Extract** states are represented by the **green** color. Green color represents a normal operational state. But the **Close** and **Insert** states are represented by the **white** color which means abnormal operational state. In practice, in some cases the green color should be associated to "Close" instead of Open, because close state is the normal operational state of a particular device. The inversion of the colors can also be acceptable for Extract and Insert states in some cases.

ATK allows to invert the color correspondance only for "Open" and "Close" states and for "Extract" and "Insert" states.

To invert the color correspondance for "Open" and "Close" states *the attribute property* **OpenCloseInverted** should be set to **True**.

To invert the color correspondance for "Extract" and "Insert" states *the attribute property* **InsertExtractInverted** should be set to **True**.

1. **StateViewer** is used to view the read value of a DevState Scalar attribute. The state is represented as a colored rectangle besides the name or the alias of the Tango Device.



ATK does not provide any component for setting a DevStateScalar attribute.

A Collection of scalar attributes

AttributeList viewers

As we have already studied them the attribute list viewers are the components which use an attribute list as their model (not an individual attribute). They display only the scalar attributes and ignore the non scalar attributes contained in the attribute list. They automatically choose the appropriate viewer depending on the type of the attribute. ATK proposes 3 attribute list viewers : NumberScalarListViewer, ScalarListViewer, ScalarListSetter. Please have a look into the section : *View a list of scalar attributes*.

A set of scalar attributes in a table (MultiScalarTableViewer)

The MultiScalarTableViewer is used to view a collection of scalar attributes inside a table. Each attribute is associated to a cell. The MultiScalarTableViewer will select the appropriate scalar attribute viewer according to the type of the attribute (NumberScalar, StringScalar, BooleanScalar or EnumScalar). The viewer is used inside the corresponding cell to display the read value of the attribute.

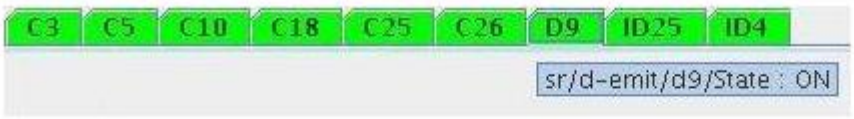
The user can also set the attribute value. To do so, (s)he should double click inside the cell. This will display a set panel adapted to the type of the scalar attribute. A double click on a read-only attribute has no effect.

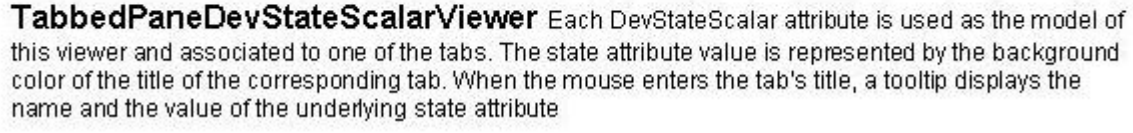
If the keyboard focus is on the table, when the mouse enters a cell a tooltip will display the precise tango name of the attribute.

Using the ATK MultiScalarTableViewer (Flash demo)

A set of DevStateScalar attributes (TabbedPaneDevStateScalarViewer)

The TabbedPaneDevStateScalarViewer is used to view a collection of state attributes in the titles of the panes of a tabbedPane. Each state attribute is added to the viewer by the call to *addDevStateScalarModel*. This method needs also the index of the tab to be associated to the state attribute. The screen shot below shows this viewer :





Trend of Scalar attributes

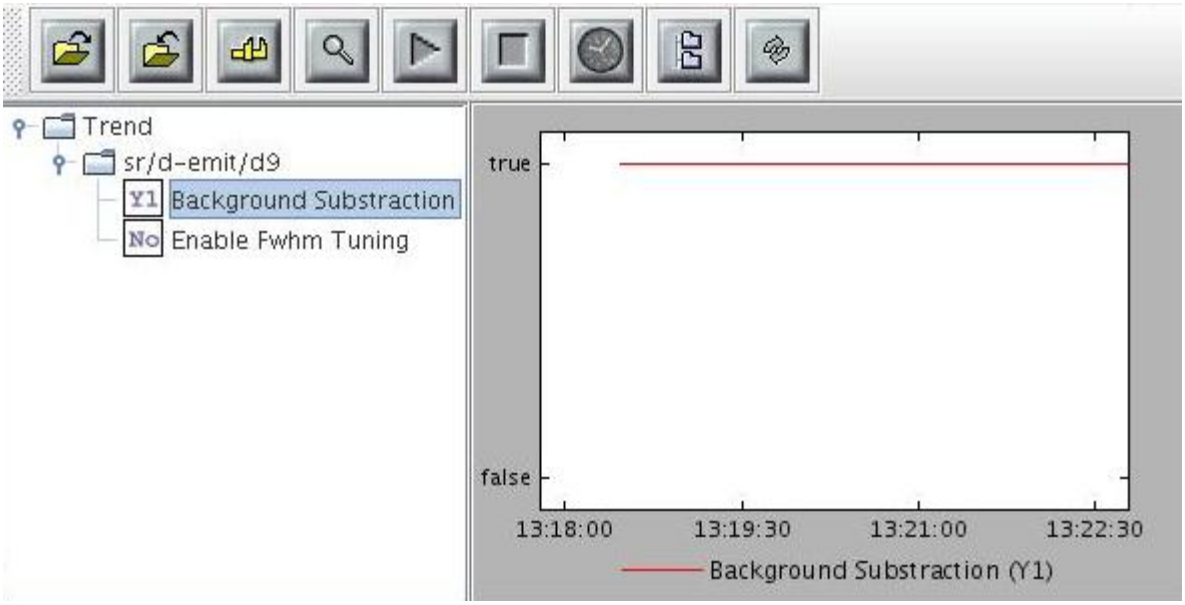
The trend of number scalar attributes

The ATK component *Trend* allows the user to follow the evolution of the value of one or more number scalar attributes during the time. Trend accepts an attribute list as model. The number scalar members of the attributeList can be plotted inside a chart during the time. Each NumberScalar attribute included in the attribute list will be read at the frequency of the refresh period and displayed as a separated plot.

Using the ATK Trend (Flash demo)

The trend of boolean scalar attributes

The ATK component *BooleanTrend* allows the user to follow the evolution of the value of one or more boolean scalar attributes during the time. BooleanTrend accepts an attribute list as model. The boolean scalar members of the attributeList can be plotted inside a chart during the time. Each BooleanScalar attribute included in the attribute list will be read at the frequency of the refresh period and displayed as a separated plot.



Spectrum attributes

A spectrum attribute is a Tango attribute whose format is Spectrum (one dimensional array) whatever the data type of the attribute. In this chapter we will see how to view and / or to set a single spectrum attribute. We will also see how to view a collection of spectrum attributes.

One single spectrum attribute

NumberSpectrum attributes

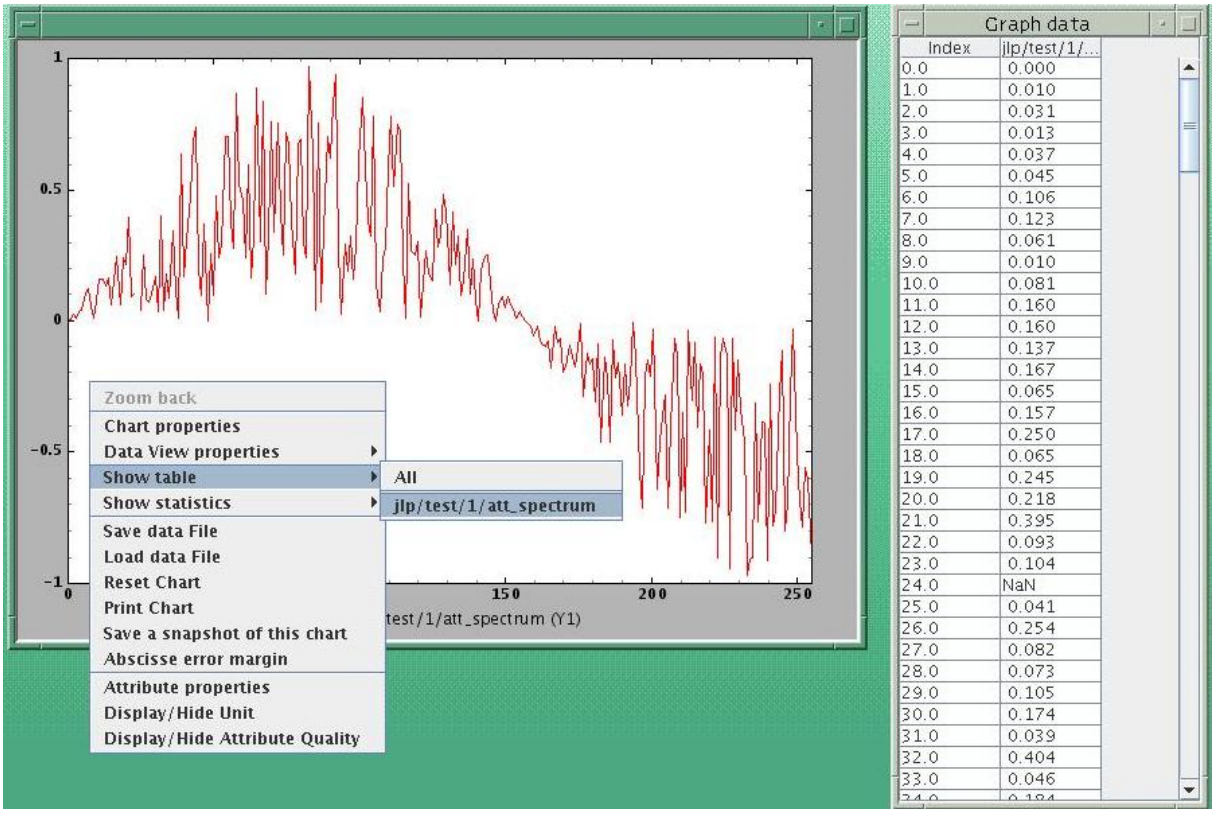
By number spectrum attribute we mean any Tango Attribute whose format is "Spectrum" and whose data type is one of the numerical types. No matter if it's a DevLong, DevDouble , or whatever numerical type.

The **NumberSpectrumViewer** is used to display the read value of a number spectrum attribute. This viewer displays the spectrum attribute as a plot in a chart. The user can display the values inside the spectrum in a table using the mouse right button menus. You can use this viewer following the code sample below:

```
AttributeList attl = new AttributeList;
try
{
    INumberSpectrum spect = (InumberSpectrum) attl.add("my/test/device/onespectrumatt");
    NumberSpectrumViewer nsv = new NumberSpectrumViewer();
    nsv.setModel(spect);
}
catch ()
{

}
```

The following screen shot shows a **numberSpectrumViewer**.

Note: The table on the right, has been displayed using the chart menus under the right mouse button.

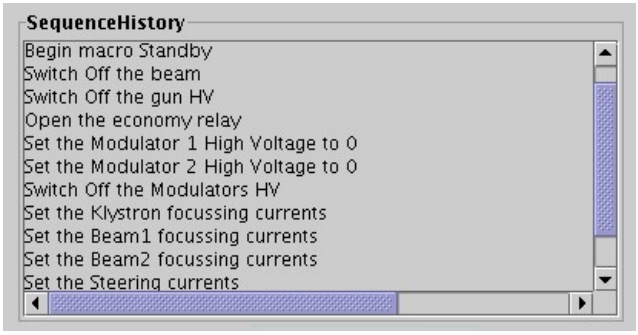


ATK does not provide any component for setting a NumberSpectrum attribute.

StringSpectrum attributes

By string spectrum attribute we mean any Tango Attribute whose format is "Spectrum" and whose data type is DevString.

The **SimpleStringSpectrumViewer** is used to display the value of a StringSpectrum attribute. The **SimpleStringSpectrumViewer** displays the spectrum attribute as a scrolled text. Each string element of the spectrum is displayed in a new line. The code sample is very similar to the one given in the previous section for the use of NumberSpectrumViewer. You just need to replace NumberSpectrumViewer by SimpleStringSpectrumViewer and replace INumberSpectrum by IStringSpectrum.



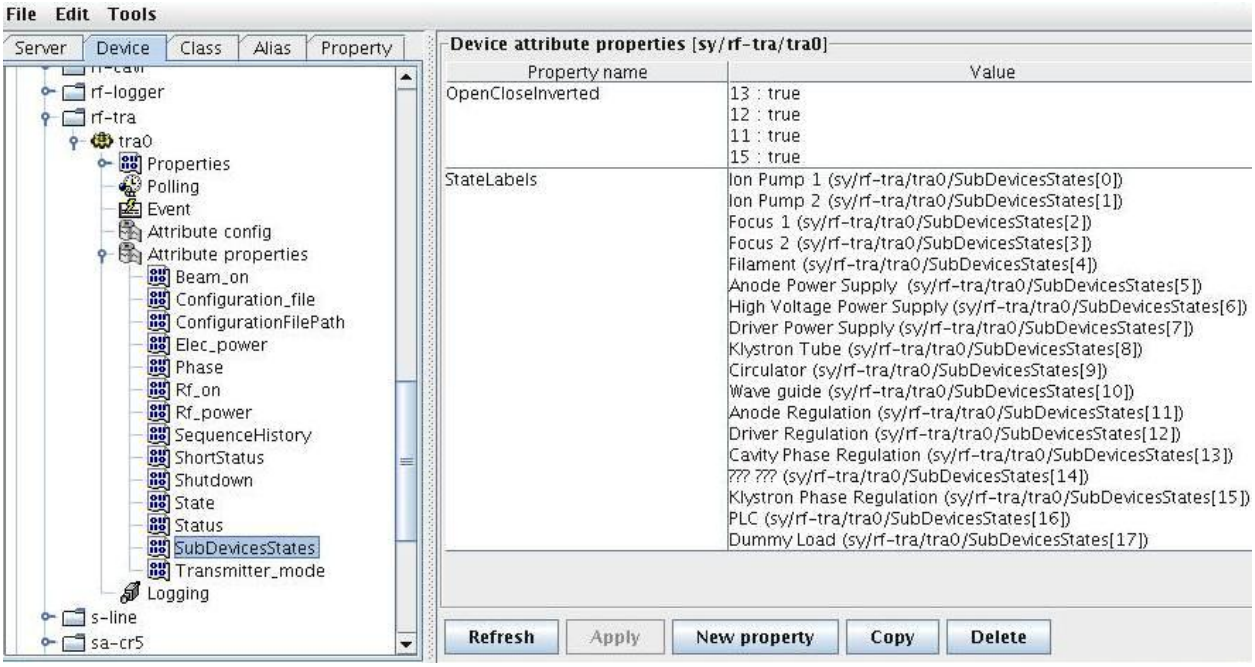
DevStateSpectrum attributes

By DevState spectrum attribute we mean any Tango Attribute whose format is "Spectrum" and whose data type is DevState.

The **DevStateSpectrumViewer** is used to display the value of a DevState Spectrum attribute. This viewer displays the elements of the state spectrum attribute vertically. Each elements is displayed in a line with three different areas: in the left a text label is displayed with the name of the attribute and the index of the element in the spectrum, in the middle a colored rectangle displays the state value and in the right side a text label displays the state value converted to a string.

DevStateSpectrumViewer The label on the left is the name of the attribute + index of the element in the spectrum. But the viewer can also display a customized label using the tango attribute properties. When the mouse enters one of the colored rectangles a tooltip displays the name of the attribute and the index of the corresponding element.

The label displayed on the left side of each element can be customized. By default this lable is the attribute name + [+ index +]. To define another label for the spectrum elements the tango attribute property **StateLabels** should be defined. In the example above, this attribute property has been defined using JIVE :



A collection of Spectrum attributes

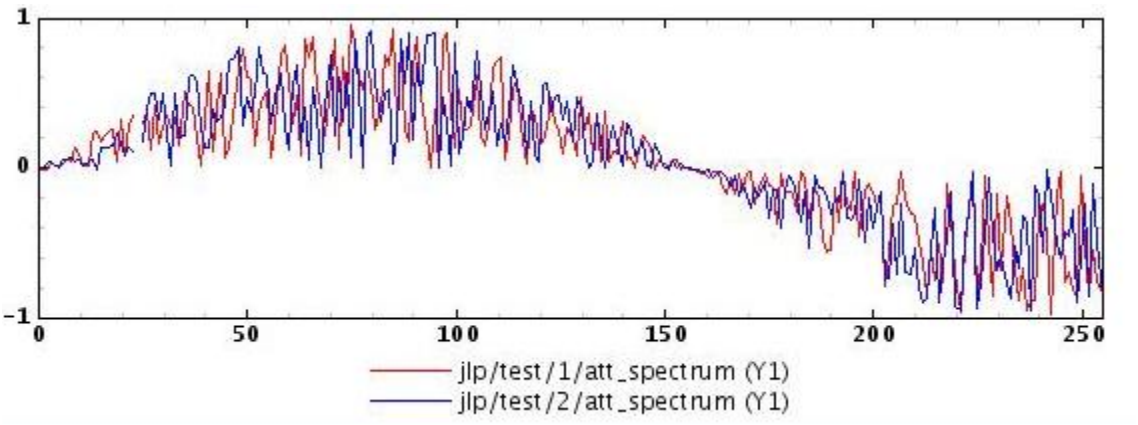
A set of NumberSpectrum attributes in one single chart

The MultiNumberSpectrumViewer is used to view a collection of number spectrum attributes inside a chart. Each number spectrum attribute is displayed as an individual plot. All plots are displayed inside the same.

The following code example uses the MultiNumberSpectrumViewer to view 2 NumberSpectrum attributes: "*jlp/test/1/att_spectrum*", "*jlp/test/2/att_spectrum*".

```
INumberSpectrum ins;
AttributeList attl = new AttributeList();
MultiNumberSpectrumViewer mnsv = new MultiNumberSpectrumViewer();
try{
    ins = (INumberSpectrum) attl.add("jlp/test/1/att_spectrum");
    mnsv.addNumberSpectrumModel(ins);
    ins = (INumberSpectrum) attl.add("jlp/test/2/att_spectrum");
    mnsv.addNumberSpectrumModel(ins);

// .... You can continue adding other spectrum attributes

}catch (Exception ex)
{
    System.out.println("Cannot connect device");
    ex.printStackTrace();
}
```

The following screen shot shows the result of the execution of this code example:



As you can see, this viewer associates each attribute plot to a colour in the order the attributes have been added by the call to "addNumberSpectrumModel" method. The user has the possibility to change the visual aspects (colour, line width, affine transform, marker, ...etc.) of each plot.

Trend of Spectrum attributes

The trend of number spectrum attributes

There are two ATK viewers which allow the user to follow the evolution of the values of the array elements of a NumberSpectrum attribute.

1. **NumberSpectrumTrendViewer**,
2. **NumberSpectrumItemTrend**.

The first component (NumberSpectrumTrendViewer) will display and follows the evolution of **ALL** elements of the spectrum.

The second component (NumberSpectrumItemTrend) is more flexible. It can display the trend of all elements of the spectrum as the first one does. But you can also specify which elements (items) of the spectrum you want to see in the trend.

The following code sample illustrates the use of the **NumberSpectrumItemTrend**.

```
NumberSpectrumItemTrend nsit = new NumberSpectrumItemTrend();
try
{
   ins = (INumberSpectrum) attList.add("fp/test/1/wave");
   nsit.setPlotAll(false);
   nsit.setModel(ins);
   nsit.plotItem(30, NumberSpectrumItemTrend.AXIS_Y1, "wave[30]");
   nsit.plotItem(1, NumberSpectrumItemTrend.AXIS_Y1, "wave[1]");
}
catch (Exception ex)
{
   System.out.println("caught exception : "+ ex.getMessage());
   System.exit(-1);
}
    mainFrame = new JFrame();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.getContentPane().add(nsit);
```

(continues on next page)

(continued from previous page)

```
 attList.startRefresher();
 mainFrame.setSize(800,600);
 mainFrame.pack();
 mainFrame.setVisible(true);

 // Test hide and show item!
 for (int i=0; i<10; i++)
 {
     try
     {
         Thread.sleep(5000);
     }
     catch(Exception ex)
     {

     }
     nsit.hideItem(7);
     try
     {
         Thread.sleep(5000);
     }
     catch(Exception ex)
     {

     }
     nsit.showItem(7);
 }
}
AttributeList attl = new AttributeList();
StringImageTableViewer sitv = new StringImageTableViewer();
Try
{
   isi = (IStringImage) attl.add("my/test/dev/att_str_image");
   sitv.setAttModel(isi);
}
catch (Exception ex)
{
    System.out.println("Cannot connect device");
    ex.printStackTrace();
}
```

The screenShot below show the NumberSpectrumItemTrend used for only two elements (index 1 and index 30) of a numberSpectrum attribute :

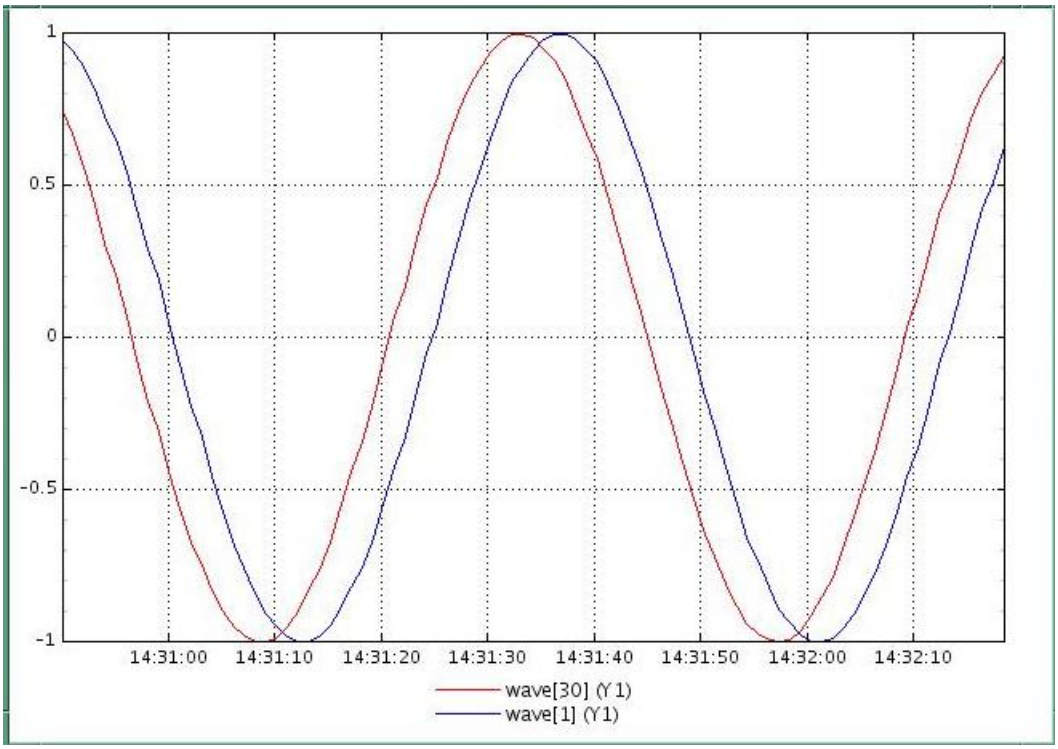


Image attributes

An image attribute is a Tango attribute whose format is Image (2 dimensional array) whatever the data type of the attribute. In this chapter we will see how to view and / or set a single image attribute.

One single image attribute

NumberImage attributes

By number image attribute we mean any Tango Attribute whose format is "Image" (2 dimensional array) and whose data type is one of the numerical types. No matter if it's a DevLong, DevDouble , or whatever numerical type. All the attributes which are not a video image such as a 2 dimensional array of numeric data, are considered to be NumberImage attributes.

The **NumberImageViewer** is used to display the value of a 2 dimensional array of numeric data (not a video image). The following code sample illustrates the use of the NumberImageViewer.

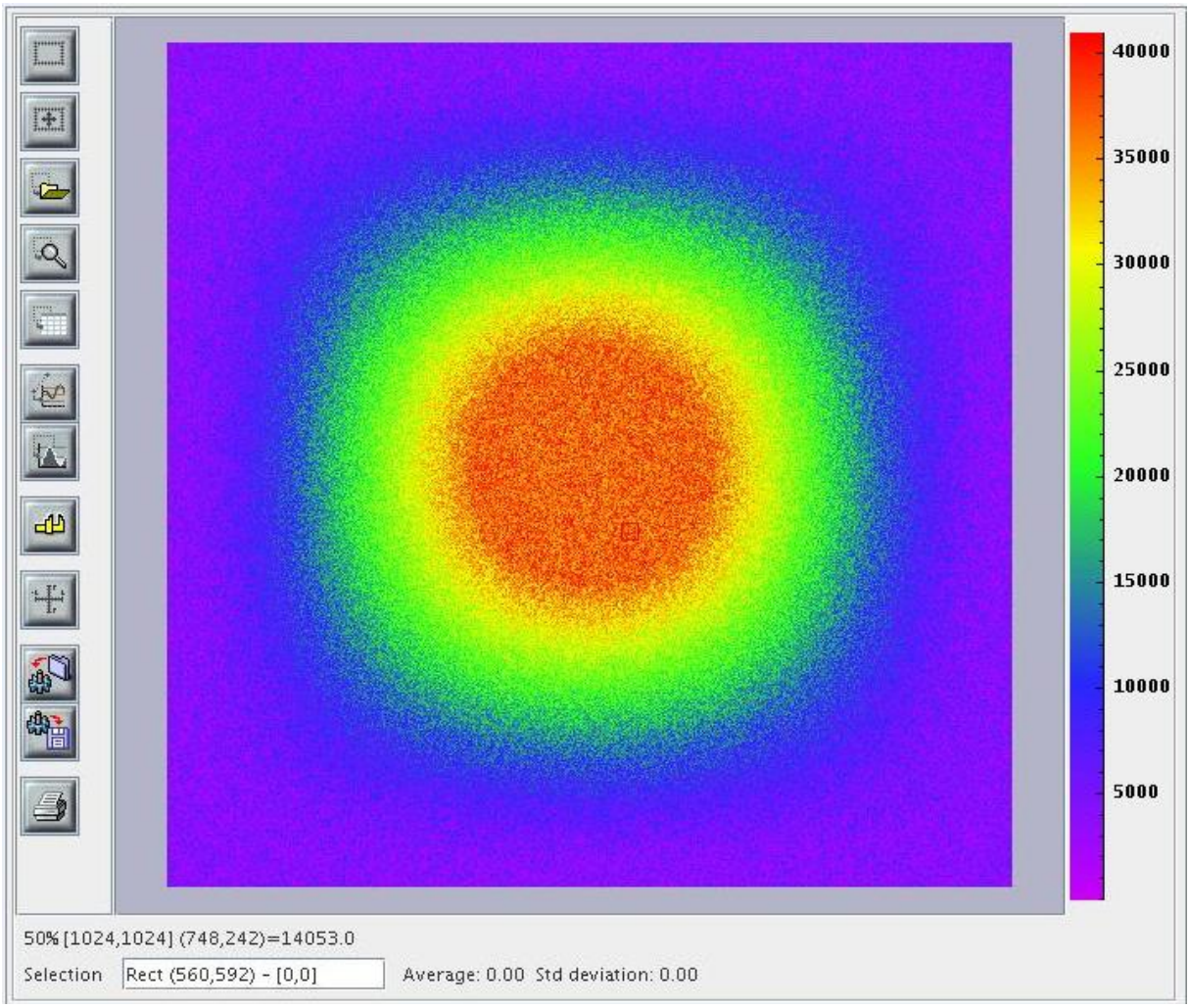
```
INumberImage ini;
AttributeList attl = new AttributeList();
NumberImageViewer niv = new NumberImageViewer();
Try
{
    ini = (INumberImage) attl.add("jlp/test/1/att_image");
    niv.setModel(ini);
}
    catch (Exception ex)
    {
```

(continues on next page)

(continued from previous page)

```
11          System.out.println("Cannot connect device");
12          ex.printStackTrace();
13      }
```

The following screen shot shows the result of the execution of the code sample above :



ATK does not provide any component for setting a NumberImage attribute.

RawImage attributes

RawImage attributes are used for the images coming from video camera, CCDs. By convention the Raw Image data (image coming from video camera, CCDs) should be sent as attributes with format = image and data type = DevUchar. The RawImage feature is not available for the moment in the standard ATK. We are waiting for a tango definition of CCD / vidéo camera images with different formats (jpeg, png, ...) in order to implement RawImages in standard ATK. The ATK RawImage viewer will be supported when the attribute data type "DevEncoded" will be available in Tango API.

ATK does not provide any component for setting a RawImage attribute .

StringImage attributes

By string image attribute we mean any Tango Attribute whose format is "Image" (2 dimensional array) and whose data type is DevString.

The **StringImageTableViewer** is used to view a StringImage attribute (a 2 dimensional array of string). Each element of the attribute array will be displayed in a cell in a swing JTable.

The following code sample illustrates the use of the **StringImageTableViewer**.

```
IStringImage isi;
AttributeList attl = new AttributeList();
StringImageTableViewer sitv = new StringImageTableViewer();
Try
{
    isi = (IStringImage) attl.add("my/test/dev/att_str_image");
    sitv.setAttModel(isi);
}
    catch (Exception ex)
    {
        System.out.println("Cannot connect device");
        ex.printStackTrace();
    }
```

ATK does not provide any component for setting a StringImage attribute.

Device Commands

Display a single tango device command

There are several viewers available to represent a Tango device command. The choice of the viewer depends on the type of the input and output argument of the command. For example the **VoidVoidCommandViewer** is used for all commands with no input argument and no output argument.

Commands with no input and no output argument (VoidVoidCommand)

The commands with no input and no output argument are called VoidVoid commands in ATK. The following list presents all the command viewers suitable for VoidVoidCommands:

1. **VoidVoidCommandViewer**: is a sub-classes of swing JButton. The label of the Jbutton is the name of the command. A click on a VoidVoidCommandViewer will immediately launch the execution of the corresponding command on the tango device. When the mouse enters the button a tooltip will display the name of the tango device on which the command will be executed.
2. **ConfirmCommandViewer**: is also a sub-classes of swing JButton.. The difference with previous viewer is that the click on the ConfirmCommandViewer button will just popup a confirmation dialog window. The device server's command is executed only if the user confirms the dialog window. As for the VoidVoidCommandViewer when the mouse enters the button a tooltip will display the name of the tango device on which the command will be executed.

VoidVoidCommandViewer the tango device name is displayed in the tooltip when the mouse enters the button. A click on the button will execute the "Standby" command of the elin/master/op device.

ConfirmCommandViewer the tango device name is displayed in the tooltip when the mouse enters the button. A click on the button will popup a confirmation dialog window.

The code sample below can be used for these two viewers indifferently:

```
ICommand ic;
CommandList cmdl = new CommandList();
VoidVoidCommandViewer vvcv = new VoidVoidCommandViewer();
Try
{
    ic = (ICommand)cmdl.add("elin/gun/beam/Off");
    vvcv.setAttModel(ic);
}
    catch (Exception ex)
    {
        System.out.println("Cannot connect device");
        ex.printStackTrace();
    }
```

Commands with DevBoolean input argument and no output argument (BooleanVoidCommand)

The commands with DevBoolean input argument and no output argument are called BooleanVoid commands in ATK. The following list presents all the command viewers suitable for BooleanVoidCommands:

1. **OnOffCheckBoxCommandViewer**: is a sub-classes of swing JCheckBox. A click on a OnOffCheckBoxCommandViewer will immediately execute the corresponding command on the tango device. The value of the input parameter passed to the device command depends on the state of the checkBox. If the checkBox is selected the device command is called with “true” parameter, otherwise the “false” parameter is sent to the command.
2. **OnOffSwitchCommandViewer**: A click on a OnOffSwitchCommandViewer will immediately execute the corresponding command on the tango device. The value of the input parameter passed to the device command depends on the state of the switch. The difference with the previous viewer is only in the graphical representation.

OnOffCheckboxCommandViewer a click on the checkbox will execute the "averaging" command of the tango device with a boolean parameter. The boolean parameter passed to the command is true if the checkbox is selected and false if the checkbox is not selected.

OnOffSwitchCommandViewer a click on the switch button will execute the "averaging" command of the tango device with a boolean parameter. The value of the boolean parameter passed to the command depends on the position of the switch button.

Commands with DevString input argument and no output argument

The following list presents all the command viewers suitable for the commands with DevString input argument and no output argument.

1. **OptionComboCommandViewer**: is a sub-class of Swing JCombobox. The limited possibilities for the input strings are displayed in the combobox drop down list. A click in this list will launch the execution of the Tango device command with the input parameter equal to the item selected in the combobox item list.

Commands with any type of input argument and any type of output argument

The following list presents all the command viewers suitable "any" tango command

1. **AnyCommandViewer**: is a sub-class of Swing JButton. This viewer is convenient for the tango device commands with input arguments and / or output arguments of **any type**. A click on the button will display a window (see the screen shot below) in which the user can enter the input argument, click on execute will execute the command with the specified input argument and if there is any output argument, it will be displayed in the lower area (scrolled text area) of this window.

Display a collection of tango device commands

ATK provides a viewer **CommandComboViewer** to display a collection of device commands in a Combo drop down list. Each element of this list acts as a "VoidVoid CommandViewer" if the command has no input and no output argument. The command list element acts as "AnyCommandViewer" if the command has an input and / or output argument.

The model for this viewer is a CommadList. All the members of the commandList will be displayed in the comboBox drop down list no matter what is the type of their input and / or output arguments.

When one of the iterms of the list is selected :

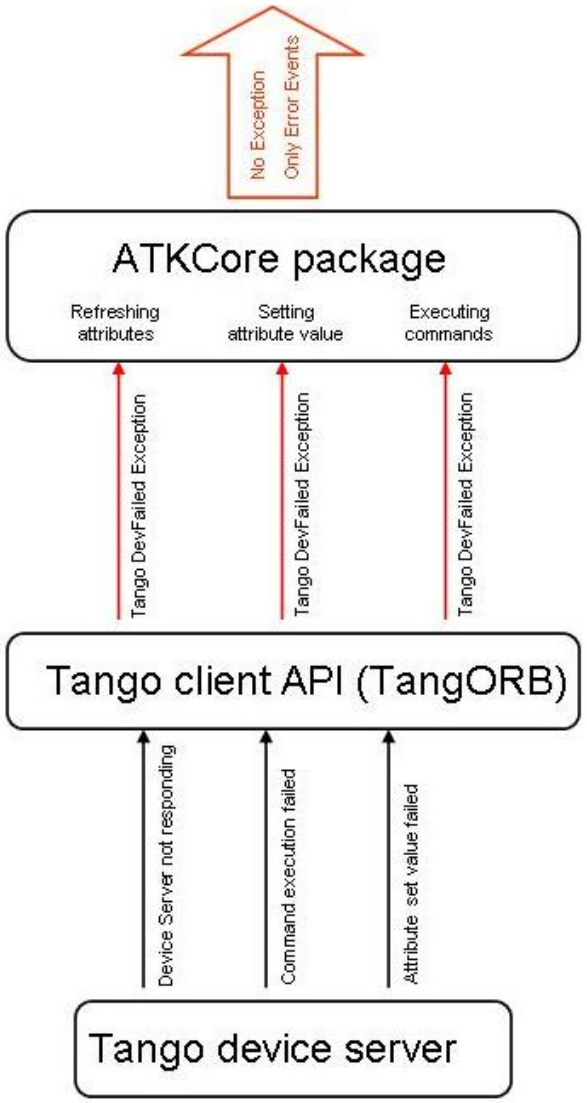
- If the command has no input argument it is immediately executed.
- If the command needs an input argument a "anyCommandViewer" window will be displayed asking for the argument to be entered.

Error Handling

All the exceptions thrown by Tango and caught by ATK are transformed into an ATK error event. Below is a list of some situations in which the exceptions are caught by ATK and tranformed into an ATK error event :

- Tango device access timeout during the refreshing of the attributes,
- Tango device access timeout during the actions like : setting the value of an attribute, execution of a command,
- Exceptions thrown by the device servers because of a non authorized action or value setting.

What is important to note is that normally all the exceptions thrown by the Tango API are caught inside ATK and transformed into error events. The only exception, which is not transformed to an ATK error, is the ConnectionException. This exception is thrown by ATK if and only if the initial connection to the tango device fails. So apart from the ConnectionException the ATK application programmer does not need to catch any tango related exception.



There are two kinds of errors in ATK. The first type of errors, called **"Error"**, is produced when the Tango DevFailed Exception occurs during the *reading* of an attribute or during the *execution* of a command. The second type of errors, called **"SetError"**, is produced when the Tango DevFailed Exception occurs during the *setting* value of an attribute. This is done to be able to make a clear separation between the errors which happen during the setting of an attribute and those which happen during the reading of the same attribute.

In addition to the ATK error events generated, ATK provides two classes of error viewers : **ErrorHistory** and **ErrorPopup**. They are the graphical viewer classes which listen to ATK error events and display the error to the application end user.

How to handle and display errors

The provided error viewer classes can be used to collect and to display ATK errors generated during the application session. Here are the steps to perform to handle errors :

- Create one or more ErrorViewer(s):

```
ErrorHistory errh = new ErrorHistory();
ErrorPopup errorpopup = ErrorPopup.getInstance();
```

- Add one or more error viewer(s) as error listeners to the empty attribute list just after it's instantiation:

```
AttributeList attl = new AttributeList();
attl.addErrorListener(errh);
attl.addSetErrorListener(errorpopup);
attl.addSetErrorListener(errh);
```

- Add one or more error viewer(s) as error listeners to the empty command list just after it's instantiation:

```
CommandList cmdl = new CommandList();
cmdl.addErrorListener(errh);
cmdl. addErrorListener(errorpopup);
```

- Connect to the attributes by adding them to the attribute list:

```
attl.add(att_one);
attl.add(att_two);
// ...
```

- Connect to the commands by adding them to the command list:

```
cmdl.add(cmd_one);
cmdll.add(cmt_two);
// ...
```

- Start the attribute list refresher:

```
attl.startRefresher();
// ...
```

The error viewers are registered as error listeners of the attribute list and the command list. This way they will be registered as the error listeners of all the members added to these lists. It is very important to *register them as error listeners of the list before the first adding* of the elements.

Error Viewers

There are two error viewer classes provided by ATK : **ErrorHistory** and **ErrorPopup**. To use them the application programmer should add them as error listeners to either attribute and command lists or to the attribute and command entities directly.

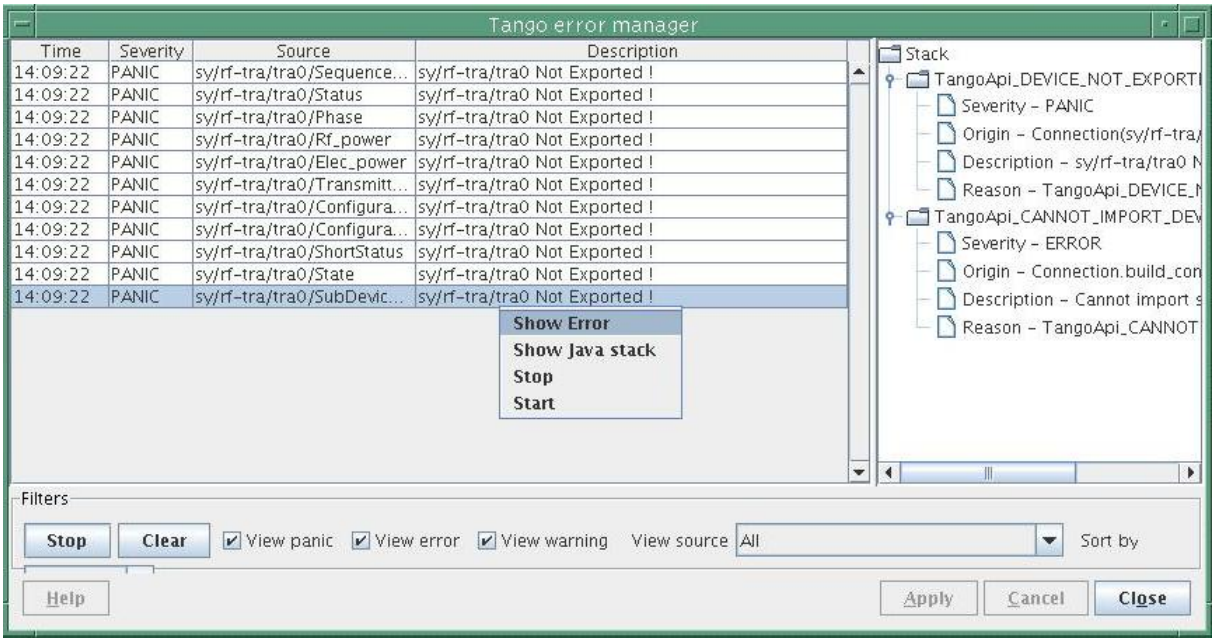
ErrorHistory

The ErrorHistory viewer is used to log all of the errors it receives and keep the history of all the errors received. It will display the list of these errors. If the same error occurs repeatedly, to save place in the window, only the timestamp of the error is changed. This way only the date and the time of the last time the error occurred is displayed.

The code sample below shows how to use ErrorHistory :

```
ErrorHistory eh = new ErrorHistory();
AttributeList attl = new AttributeList();
attl.addErrorListener(eh);
attl.addSetErrorListener(eh);
```

The call to **"addErrorListener"** will add the ErrorHistory as a listener for all errors excepted those happening during the attribute set value. If we want to log into the ErrorHistory the attribute setting errors we should call the **"addSetErrorListener"** in addition to **"addErrorListener"**.



A right click on one of the errors displayed in the list, will display detailed information about that particular error. "Show Error" will display on the right panel the Tango error stack.

ErrorPopup

The ErrorPopup viewer is a singleton class in ATK. This viewer is a dialog window which pops up as soon as it receives an error. The error description is displayed and the user can get the detailed description of the error. The ErrorPopup window waits for the user click to disappear.

Normally the ErrorPopup should NOT be used for the errors which occur during the attribute refreshing. It should be used for errors which occur rarely like the setting of an attribute or the execution of a command.

The code sample below shows how to use ErrorPopup:

```
ErrorPopup errpp = ErrorPopup.getInstance();
AttributeList attl = new AttributeList();
CommandList cmdl = new CommandList();
```

(continues on next page)

(continued from previous page)

```
4 attl.addSetErrorListener(errpp);
5 cmdl.addErrorListener(errpp);
```

Note: The ErrorPopup is only added as "SetErrorListener" to the attribute list.

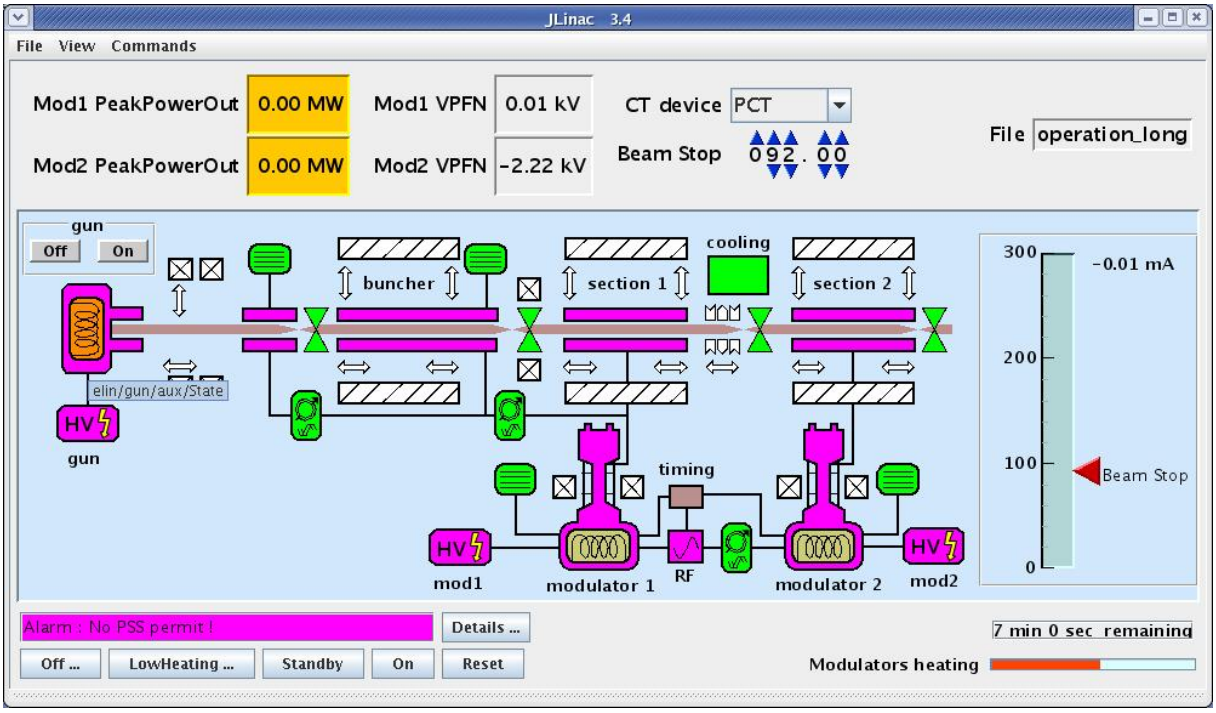
Synoptic drawing and programming

ATK provides a complete synoptic system. As already mentioned in the introduction, the main idea of the synoptic drawing and viewing system is to provide the application designer with a simple and a flexible way to draw a synoptic and to animate it at runtime according to the values and states read from the control system.

What is a synoptic application?

In an application based on a synoptic the user can see a "free style" drawing, in which different parts can report on the tango device states and/or the tango attribute values of the control system. We say that the drawing is "animated" at run-time according to the values / states of the control system objects.

The following picture is the snapshot of the ESRF Linac control application based on a synoptic. The synoptic is the drawing in the center with the background color in blue.



As you can see the drawing components in the synoptic have different colors according to the device state attribute to which they are linked. For example the drawing component linked to *"elin/gun/aux/State"* is colored in orange because the value of this state attribute is *Alarm*. Moreover you can see the red arrow (Beam Stop) on the slider pointing to the value of the tango attribute *"elin/master/op/SRCT_limit"* which is 92 as it is also represented outside of the synoptic on the top of the window.

What kind of animations are provided at run-time?

The run-time behavior of the synoptic is predefined in ATK and it depends on the type of the graphic object (free drawing, dynos, sliders, buttons, ...) on one hand and the tango control object to which it is linked (state attribute, numerical attribute, boolean attribute, tango command ...). The exact run-time behaviour in each case will be discussed in a further section.

Appendix 1 : attribute viewers / setters

Tango format and data type	View / Set	ATK class used as model	ATK viewer / setter	Tutorial section
Scalar Any type Single attribute	View and Set	AttributeList	*ScalarListViewer ScalarListSetter NumberScalarListViewer*	*Use a generic scalar attribute viewer*
Scalar Numeric type Single attribute	View	INumberScalar	*SimpleScalarViewer* *NumberScalarViewer* *NumberScalarProgressBar*	*Using specific viewers . . . *
Scalar DevString single attribute	View	IStringScalar	*SimpleScalarViewer* *StatusViewer*	*Using specific viewers . . . * *device status*
Scalar DevBoolean Single attribute	View	IBooleanScalar	*SignalScalarLightViewer*	*Using specific viewers . . . *
Scalar DevBoolean Single attribute	View and Set	IBooleanScalar	*BooleanScalarCheckBoxViewer*	*Using specific viewers . . . *
Scalar DevShort, DevUshort Single attribute	View	IEnumScalar	*SimpleEnumScalarViewer*	*Using specific viewers . . . *
Scalar DevState single attribute	View	IDevStateScalar	*StateViewer*	*device state* *Using specific viewers . . . *
Scalar Numeric type Single attribute	Set	INumberScalar	*NumberScalarWheelEditor* *NumberScalarComboEditor*	*Using specific viewers . . . *
Scalar DevString single attribute	Set	IStringScalar	*StringScalarEditor* *StringScalarComboEditor*	*Using specific viewers . . . *
Scalar DevBoolean Single attribute	Set	IBooleanScalar	*BooleanScalarCheckBoxViewer* *BooleanScalarComboEditor* *SignalScalarButtonSetter*	*Using specific viewers . . . *
Scalar DevShort, DevUshort Single attribute	Set	IEnumScalar	*EnumScalarComboEditor*	*Using specific viewers . . . *
Scalar Any type Collection of attributes	View and Set	AttributeList	*ScalarListViewer ScalarListSetter NumberScalarListViewer*	*AttributeList viewers* *AttListViewer Flash Demo*
Scalar Numeric type Collection of attributes	View	AttributePolledList	*Trend*	*The trend of numberScalar* *Trend Flash demo*
Scalar DevBoolean Collection of attributes	View	AttributePolledList	*BooleanTrend*	*The trend of boolean scalar attributes*
Scalar Any type Collection of attributes	View and Set	IAttribute	*MultiScalarTableViewer*	*A set of scalar att. . . * *Scalar Table Flash demo*
Scalar DevState Collection of attributes	View	IDevStateScalar	*TabbedPaneDevStateScalarViewer*	*A set of DevStateScalar attrributes*

Appendix 2 : command viewers

Input argument data type	Output argument data type	ATK class used as model	ATK Command Viewer	Tutorial section
DevVoid no input	DevVoid no output	ICommand	*VoidVoidCommandViewer* *ConfirmCommandViewer*	*Commands with no input and no output*
DevBoolean	DevVoid no output	ICommand	*OnOffCheckboxCommand-Viewer* *OnOffSwitchCommand-Viewer*	*Commands with DevBoolean input and no output*
DevString	DevVoid no output	ICommand	OptionComboCommand-Viewer	*Commands with DevString input and no output*
Any Type	Any Type	ICommand	*AnyCommandViewer*	*Commands with any type of input and any type of output*
Any Type	Any Type	*A collection of commads* CommandList	*CommandComboViewer.jpg*	*A collection of commands*

Appendix 3 : error viewers

Type of ATK error	ATK method to use	ATK Error Viewer	Tutorial section
Attribute read error during the attribute refreshing	addErrorListener	*ErrorHistory*	*ErrorHistory*
Attribute setting error during the attribute set value	addSetErrorListener	*ErrorHistory* *ErrorPopup*	*ErrorHistory* *ErrorPopup*
Command execution error	addErrorListener	*ErrorHistory* *ErrorPopup*	*ErrorHistory* *ErrorPopup*

ATK Java documentation

Intended audience: developers, ***Programming language:*** java

The most important information Dependencies, Connection of models and views and etc. you will find following this link.

7.2.7 LogViewer

Intended audience: all

Overview

LogViewer is an application for displaying logs pushed by the devices to the Tango logging system. You can fetch logs from different Tango devices and filter them by several properties to find what you need.

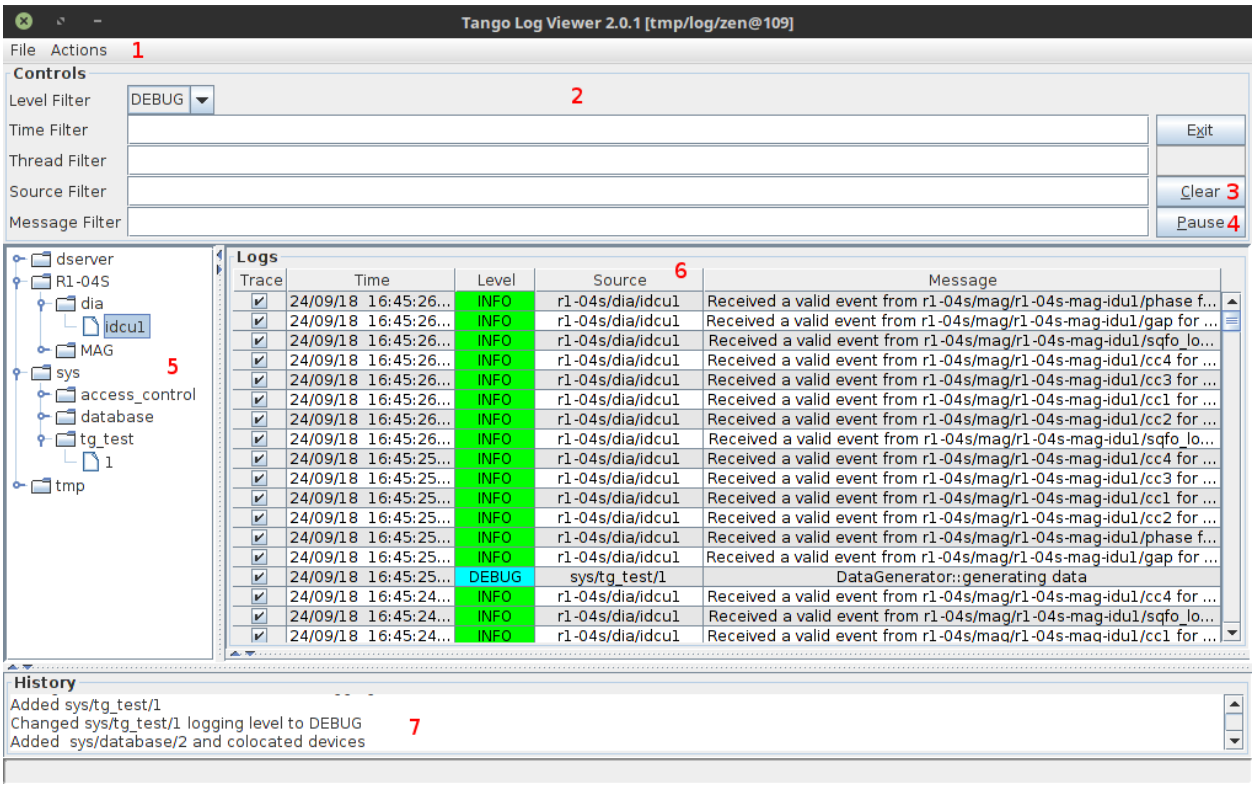


Fig. 21: LogViewer 2.0.1 main window

1 - Menu bar
2 - Filter settings
3 - Clear button - clears the logs window
4 - Pause button
5 - Device tree
6 - Logs
7 - History of actions

Tango Logging

Tango provides a logging service for devices. It is configured by several device properties, and support sending logs to various targets, such as files, log consumer devices or the console. For more information you can check out *more details about logging service*.

Adding devices

To add logs from device to the viewer you need to right click on the device in the device tree and click add. Then you need to set which level logs from that device should be visible. You can do this with *Set Logging Level* options in the device context menu.

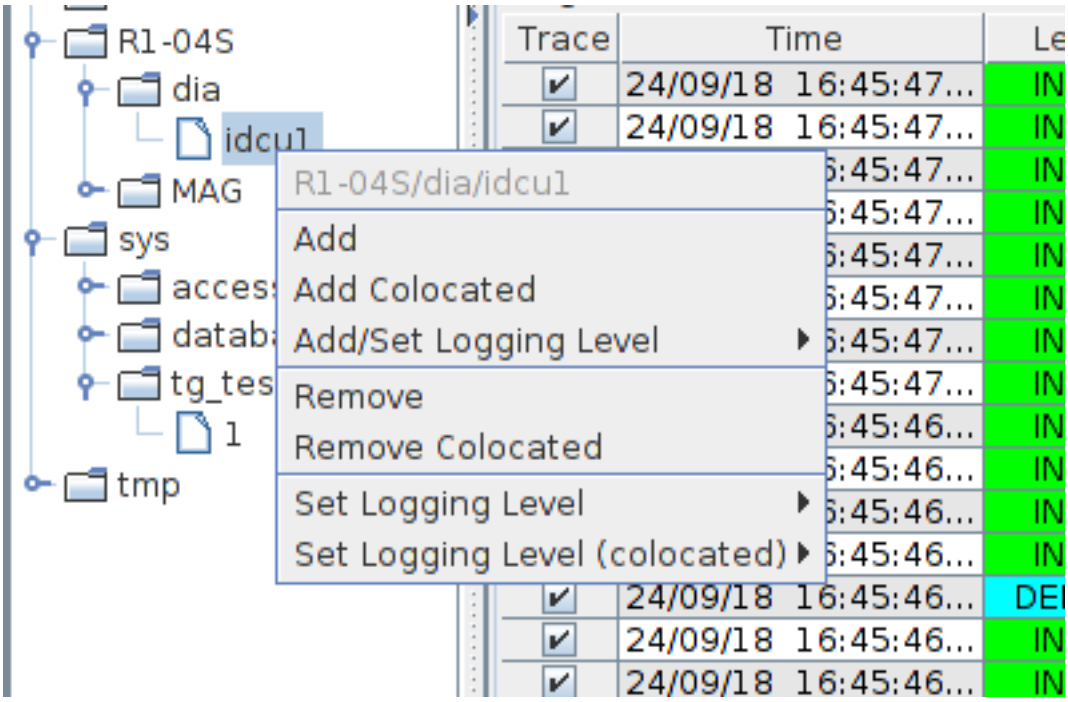


Fig. 22: Add logs from device

Alternatively, you can add device with log level already set with *Add/Set Logging Level* context menu option.

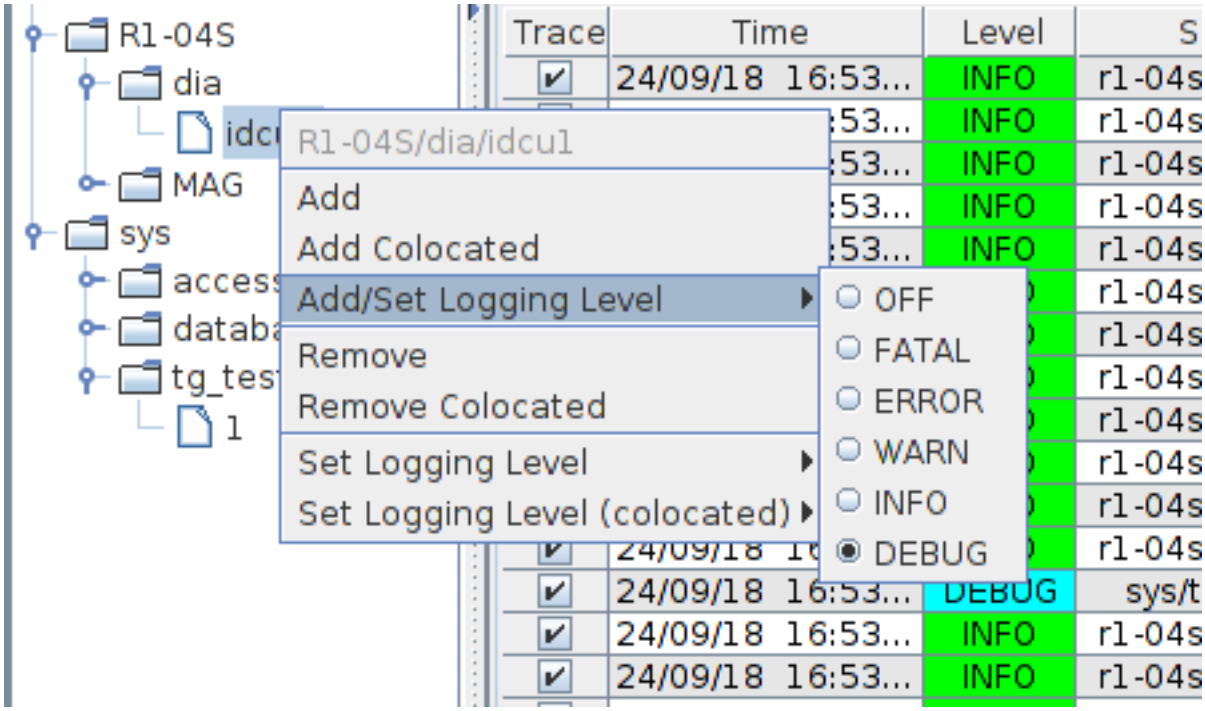


Fig. 23: Add and set logging level

Add colocated devices

The *Add Colocated* option adds all devices, running on the same device server instance, as logging sources. For example, if you have a `TangoTest/test` instance with `sys/tg_test/1` and `sys/tg_test/2` devices, choosing *Add Colocated* on any of these devices will add both of them. The same applies for *Remove Colocated* and *Set Logging Level (colocated)* options.

Actions menu

The actions menu provides access to some useful functions:

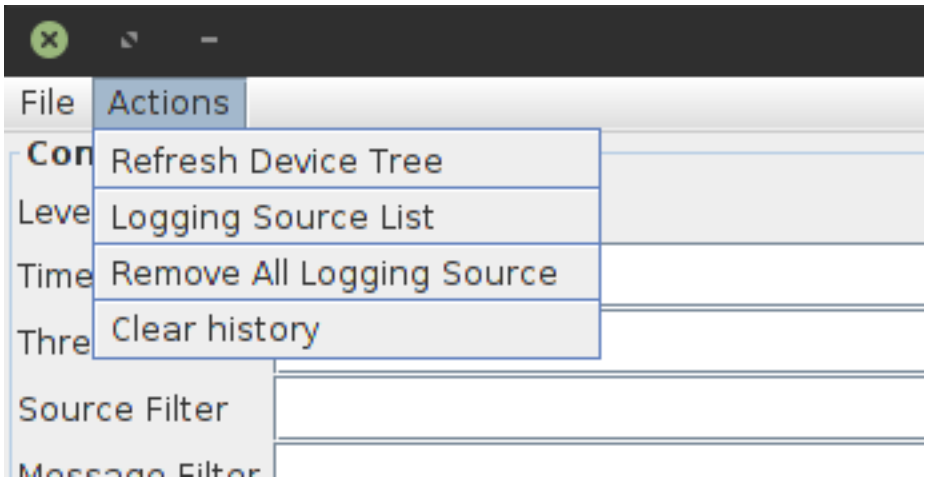


Fig. 24: Actions menu

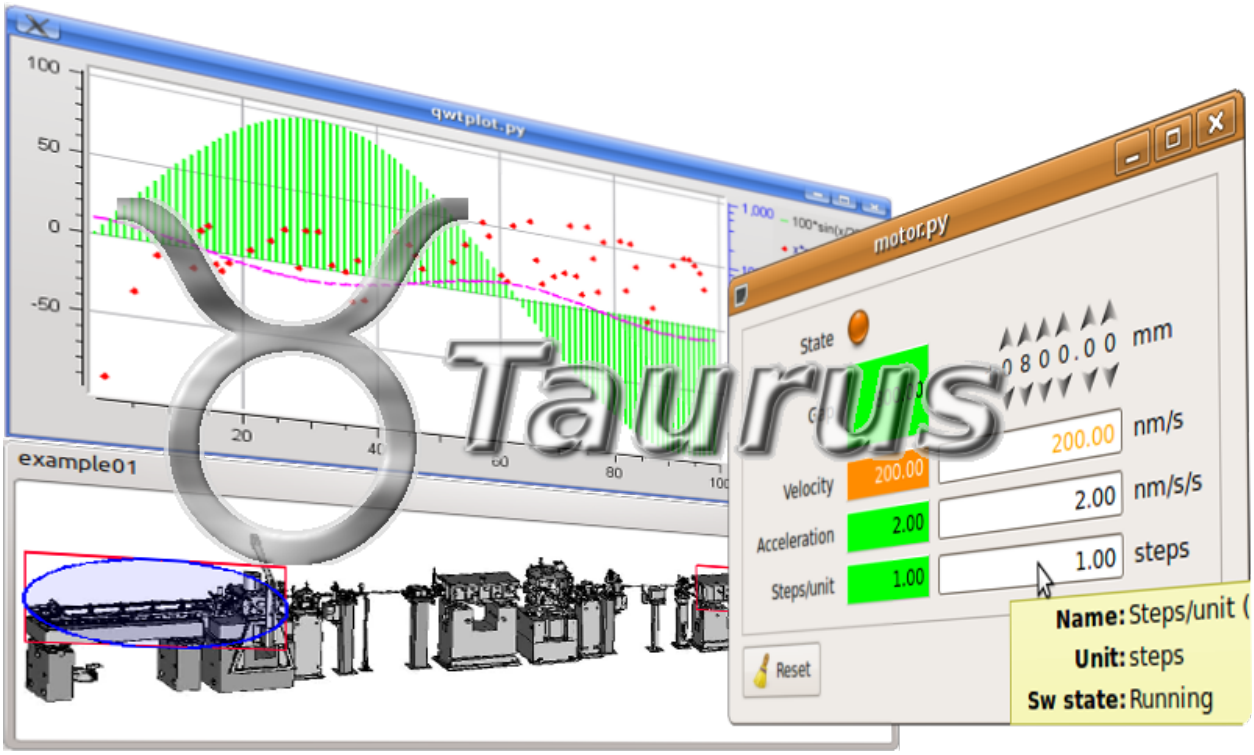
- *Refresh Device Tree* - refresh devices
- *Logging Source List* - list all added devices in the history window
- *Remove All Logging Source* - remove all devices from source list
- *Clear history* - clear history window

Loading logs from files

With the *File* > *Load file* menu option you can load a log file generated by the devices which logging target is set to file. This allows you to use the LogViewer features to filter and search through the logs offline. The log file format is XML.

7.3 GUI building

These are libraries for creating GUI applications.



7.3.1 Taurus (Python GUI library)

Intended audience: developers, ***Programming language:*** python

Taurus is a Python framework for control and data acquisition CLIs and GUIs in scientific/industrial environments. It supports multiple control systems or data sources: *Tango Controls*, EPICS, spec… New control system libraries can be integrated through plugins.

For non-programmers: Taurus allows the creation of fully-featured GUI (with forms, plots, synoptics, etc) from scratch in a few minutes using a "wizard", which can also be customized and expanded by drag-and-dropping elements around at execution time.

For programmers: Taurus gives full control to more advanced users to create and customize CLIs and GUIs programmatically using Python and a very simple and economical API which abstracts data sources as "models".

Taurus - based on Python and PyQt or PySide. Widely used by the Python and other communities.

Download taurus from PyPi. Source code is on GitHub.

You can find its full documentation here.

7.3.2 QTango

Intended audience: developers, ***Programming language:*** c++

QTango - based on C++ and Qt. Favourite amongst C++ developers.

QTango is a framework built on top of QTangoCore and QtControls. It consists of classes and widgets that interact with the Tango control system, while providing an easy API to the programmer and full integration with the Qt4 designer.

QTangoCore is a framework built on top of Trolltech's QT , version 4 and the Tango control system. It implements a multi threaded architecture, device centric and easy to use for the Qt programmer.

QtControls is a set of Qt-based widgets used to develop graphical user interfaces to set/display values. If you are going to use Tango don't use this library, take a look at QTango instead.

QTangoDBus is a QTango module introduced with the version 4.2 of the library. It provides an interface to the DBus message bus that can be used by the QTango applications that want to interoperate with each other. It is possible to be notified when a new application registers to the message bus, to connect to a running application and even impart commands to it by means of the TApplicationInterface. An example of the usage of the framework can be found under the dbus/utils/QTangoDBusIntrospector/ application.

The following presentations by Giacomo Strangolino are available:

- `Presentation QTangoCore` - a multi threaded framework to develop Tango application
- `QTWatcher and QTWriter classes` presentation
- The conference paper `"QTango: a library fir easy TANGO based GUIs development"`
- download qtango core
- download QGraphicsPlot a graphical library that uses QGraphicsView/QGraphicsScene to draw graphs on a plot.
- download TGraphicsPlot a library that uses QGraphicsPlot and QTango to provide scalar and spectrum curves on plots.
- download Mango a library that uses QGraphicsPlot and QTango to provide scalar and spectrum curves on plots.

7.4 Web tools

7.4.1 Canone

Intended audience: developers, administrators, users

Introduction

Canone is the PHP web interface of Tango. The external appearance is a set of "panels" from which a user can interact with Tango via web. Each panel is composed of a certain number of "widgets" and some HTML tags; a widget is a graphic element which represent a single attribute or command (or property, yet not implemented).

Requirements

The installation requires a web server with PHP; the PHP should be configured with Sockets, SQLite and GD2. It's also required a machine (possibly different from the web server) with PyTango. The client side requires only a web browser (Firefox is the best choice) with JavaScript and pop-ups enabled.

Download

You can download Canone here.

Panels

With Canone it's easy to build and customize control panels. A user endowed of enough privilege can save many different panels and chose one of them to be displayed from a selection box. A common panel is composed by a title and a simple table of widgets, but the page deployment may be as complex as allowed by HTML; sub-tables, external links are only a few examples of what can be added.

Panel Configuration

The configuration of a panel concerns two different aspects: the configuration of the page deployment and the configuration of each single widget. The page deployment is configured pressing the XML button which makes an easy pop-up window to appear; in this dialog there is the XML code corresponding to the configuration of the panel. Some XML tags are specific of Canone, but others are identical to HTML and can be used to modify the deployment of the panel. This XML code can be modified freely but a certain caution is recommended! Don't use double apices (“), it's better to use single apices (‘) The “modify” button allows the modification or the removal of any widget. Pressing this button each widget is surrounded by a frame in which are included also two buttons: a “modify” button (see head) and a “delete” button. Warning: in order to eliminate a widget it's better to use the XML dialog.

Widgets

A widget is a graphical element which displays one (or more) control system parameter. Now only read attributes and write commands are supported. Each widget is contained in its own class which extends the generic “widget” class. A fundamental attribute of this class is called “config” and contains all the parameters that can be customized from outside and for each parameter: an identifier, a sort of type definition, the default value, a short description and a long description (intended for the online help). The setParam() method allows to set all the parameters with a single string. Some widgets are implemented as PNG images others as tables, in both cases the HTML code necessary to visualize the widget is returned by the plot() method.

Widget Configuration

Each widget is associated with an attribute or a command and a panel. There is a table intended to customize each parameter of the widget. The colors are configurable also through color selection pop-up.

Users

Canone supports user access and user administration. The user autentication process in a web environment can't be too simple and can't assume at any time the loyalty of users. The access is validated using data stored in a database (SQLite).

User Login

Users must be registered in the system with Username and Password. Each user must be inserted in a user group, permissions are granted to user groups. Associated to a user group may be several IP numbers (and netmask) which entitle the rights of the user group without username and password. A user inserted through IP in a given group can at any time login with username and password and get the permission of a different user group.

Permissions

There are four levels of permissions:

- **read**: only reading operations are granted, any write operation is denied
- **operator**: both reading and writing operations are granted
- **expert**: both reading and writing operations are granted, panels may be modified
- **none**: all operations are denied

There is also a special group called admin which is the only with administration permissions. They can create or delete users, grant or revoke permissions to everybody.

Administration

The administration utility is coposed by three tools.

- **access control**: all groups (for users, devices and panels) are listed and privileges can be granted or revoked on the fly. Each user group must have a default permission which is overridden by specific permission given to the combination of user group and device group or user group and panel group (e.g. the host group has no default privilege but may have privilege to se panels in a specific group)
- **user**: you can search, create, modify or delete a user account
- **database**: a generic database graphical client allows to change any configuration. Should be used only by expert administrators

Communications

The communication to Tango is implemented through a raw text socket. There is a Python script which include the Python to Tango binding (PyTango) and act both as a Tango client and socket server. In an infinite loop it recognize the requests coming from the PHP through the socket, retrieves the data from the Tango host defined by the $TANGO_HOST environment variable and return the results as a single string. The first part of the string is the number of bytes of the rest of the string. The TANGO_HOST may be changed from the client socket on a per call basis. The socket is multi-channel i.e. many instances of the script may be launched each with a progressive integer parameter, each script will work on a port given by the port base plus the given parameter. In the first few lines of the script you must customize the base port and the IP address of the authorized clients, all other clients will be immediately disconnected.

A side-effect: SOAP

The PHP to Python socket interface which deals with the buffer database and un-serialize the data coming from the socket was an excellent starting point to create a web server powered by SOAP. It was so easy to create such a server that it can be considered as a side-effect of Canone. Along with the server (tango-serveer.php) there is also a 50 lines client for test purpose (tango-client.php). Of course also this server still doesn't implement command and property queries.

The TODO list

This are only a few possible extension/improvements, many more may be implemented in future releases:

- add quality factorof attributes
- extend the utilization of AJAX technology
- properties interfaces
- add LDAP support (only for password validation)
- export to PDF, XLS, e-mail
- add other generic and some specific widgets
- save and load from file
- improve customization of widgets
- use SOAP to replace raw sockets
- add graphical editor as a valid alternative to HTML text editor
- improve security checks
- add panels selection with tabs
- split this manual into user and administrator manual
- improve the messages and dialog of the online help

FAQ

- **What's Canone?**
  Canone is a web interface to Tango. With Canone you can build your own panels and interact with any installation of Tango.
- **Is Canone a graphical interface?**
  Canone is a fully graphical and automated interface. You can select and configure some graphical elements (widgets) and combine them in one or more panels.
- **What's a widget?**
  A widget is a pre-defined graphical objet which is associated with a single attribute and can take several configurations (e.g. background color, size, shape, etc).
- **What's a panel?**
  A panel is a collection of widget + some editable HTML tags.
- **Why HTML autorefresh isn't supported?**
  Autorefresh isn't supported becouse it would make uncomfortable any command sent to Tango from Canone. If an autorefresh event appens exactly when you are pressing a button, your click may be lost (a test gave aproximately 25% loss) and even worst you will not be notified if it has been lost or not. From version 2.0 some widgets support autorefresh using AJAX.

- **Why I can't see any panel?**
  You may be in a group of users which has no privilege to see any panel or the tango-host may be wrong. To check - modify the tango-host you should ask to the administrator who can modify it from the panel construction tool or he can modify a configuration file.

7.4.2 Waltz

Waltz - a general purpose Tango web application that provide the interface between the Tango control system and the scientific users who define and calibrate their experiments. It first had a name of TangoWebapp but on the Tango Users Meeting 32 during the voting it gained its name as Waltz. So if you find something about TangoWebapp just know that it is the same project.

Waltz was designed as a platform on top of which web application was built. The reason of developing the application as the idea to give some "freedom" to the beamline scientists, e.g. to give:

- Convenient remote access (possibility to track the experiment outside the control room)
- Integrated Tango tools
- Powerful monitoring
- Scripting

It is a general purpose application - you can browse devices, read and write attributes, pipes, monitor and configure (polling, adding, deleting).

Speaking about Waltz you should understand that

- Each Beamline may have its own instance of Waltz;
- Each Beamline may have its own custom Dashboard;
- Each Beamline may have 100% custom Web application ontop of Waltz platform.

In general, you or DevOps install Waltz in the Beamline (which obviously should have Tango Controls), get all the credentials from sysandmins and can check what is going on in you Beamline accessing it in browser from any part of the world. The configuration options (e.g. hosts, devise filters, monitored attributes) and scripts are saved per user bas.

A full information about Waltz can be found by visiting its documentation page.

7.5 Archiving

7.5.1 Overview

Intended audience: beginners, developers, administrators, users

Tango has two main archiving solutions - the original one (HDB) and a new one (HDB++). The difference between the two is in the features. HDB supports Oracle and MySQL databases. HDB++ supports higher time resolution, multiple database backends (MySQL and Cassandra), and is based on events. HDB++ is designed to have a higher throughput and a number of improvements like better error management, lower footprint etc.

This map shows you how the archiving systems and the related tools are layered.

7.5.2 HDB

Intended audience: administrators, developers

Archiving - the TANGO archiving system implemented in Java.

This is a previous version of HDB++.

You can download source and documentation here.

Mambo User Manual

Intended audience: users, administrators Contents:

Authors:

L.Claisse, R.Girardot, M.Ounsy, A.Buteau, M.Thiam, A. Fourneau, G.Abeillé - Soleil

Introduction

This document is an end-user guide to using the Mambo application, and a brief developer-oriented presentation of the application's architecture.

Application's context: Historic and Temporary archiving

The goal is the automatic archiving of the values of a set of Tango Attributes, in a given time range.

In this time range, the value of an attribute is archived:

- Every [*given time interval*]

  AND

- When the attribute's value meets [*given condition on attribute's value*],

where [*given time interval*] and [*given condition on attribute's value*] are user-defined for each attribute (individually or for a group of attributes).

Each attribute value is associated with a time stamp, representing the time when the attribute value was read. This time stamp is archived as well.

Attributes can be archived in two distinct databases:

- The Historical Data Base (*HDB* henceforth): attributes values are stored forever.
- The Temporary Data Base (*TDB* henceforth): every [*given deletion time*], the oldest attributes values records are deleted. The TDB is regularly purged. The Interval of this purge is defined by the data base administrator and the system administrator.

When an attribute is "being **archived**", it means that every [*base time unit*], the archiving device tests whether the current value of the attribute has to be stored in HDB (resp. TDB).

If the test is positive, a record is added in the table for this attribute.

Later, those values can be **exploited**. It can be selected from HDB/TDB to be displayed and saved in a file

Application's description and goals

Application's goals

Mambo allows the user to define *configurations* that describe the archiving and exploitation for a group of attributes. Those *configurations* can be saved as files and modified.

Mambo is thus naturally divided (both in functionalities and display) in two parts:

- The archiving control part.
- The archived data exploitation part.

A first look at Mambo

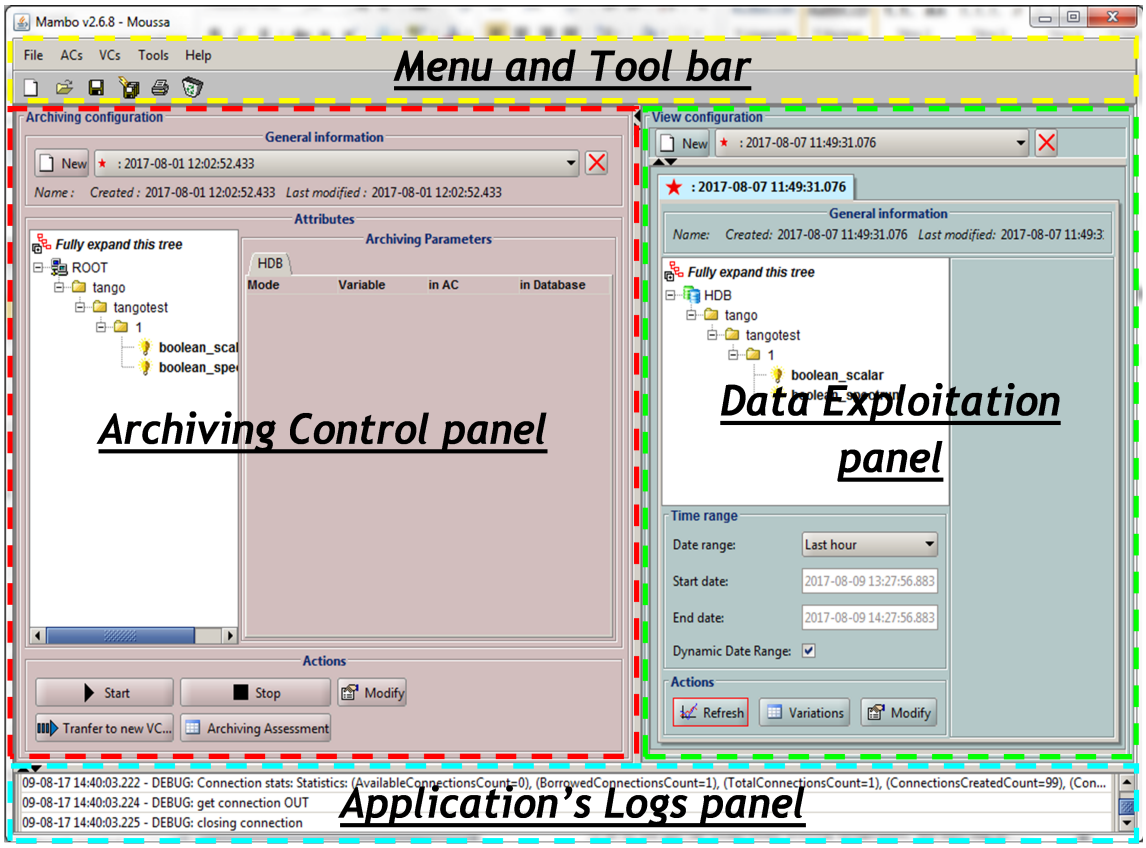


Figure 1: General view of Mambo

- The Archiving Control panel is where the user controls the archiving of Tango attributes in HDB/TDB.

  To do so, the user can create/modify archiving configurations, and Start/Stop those configurations.

  An **Archiving Configuration** (*AC*) is a list of attributes to be archived, with a list of archiving parameters for each attribute.

  Users can also start/stop the archiving of some attributes in the configuration, (see 5.3.2 *Starting/Stopping archiving on an AC*).

- The Data Exploitation panel is where the user exploits the previously stored data, mainly displaying curves and extracting raw data from HDB/TDB.

He does so by creating/modifying **View Configurations** (*VC*), and calling **View Stop** on those configurations.

A **View Configuration** is a list of attributes to be displayed, with a list of displaying parameters for each attribute.

- The Menu and the Tool bar are for actions shortcuts and application's options.

Each panel will be detailed in its own subsection.

Account manager

The Mambo Account Manager is here to manage accounts, which means:

- Creating a new account.
- Deleting an existing account.
- Launching application with an account chosen in a list.

An account is associated with a directory, in which are stored application setup and history. Having various accounts allows different users to have their own application configuration and history.

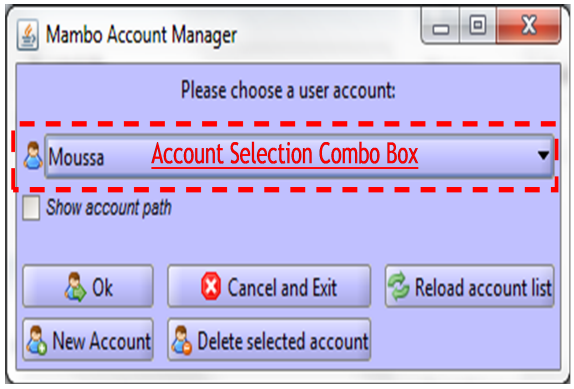
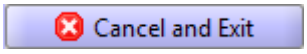


Figure 2: Mambo Account Manager

The above figure presents the account Manager Interface, at application start. With this manager, user can create a new account, or delete or use an existing one.

To quit the application, simply click on 

or button.

Existing accounts are listed in the account Selection Combo Box, which you can be reloaded by clicking on *Reload account list* button (if you think that someone could have modified it by creating a new account or deleting an existing one, for example).

Creating a new account

To create a new account, click on the *New Account* button (at the bottom left of the panel). A new dialog will appear, as following.

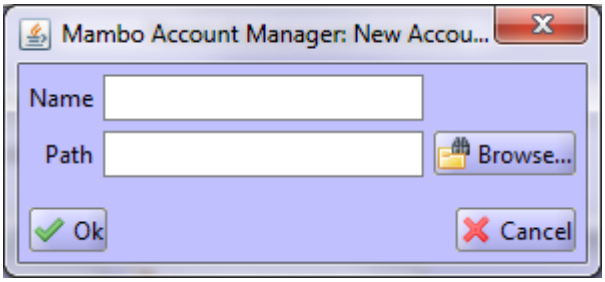
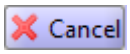


Figure 3: Creating a new account

In this new dialog, you will have to enter the name of your new account and the path of the application working directory for this account. If you prefer, you can browse for the path by clicking on the *Browse...* button. Then, a classic browsing dialog will be displayed, in which you can choose the directory. When both fields ("Name" and "Path") are fulfilled, click on *Ok* button to validate your new account, which will be automatically added in the list of existing accounts. If you click on 

or button, you go back to the first dialog, as presented in, and nothing is done.

Deleting an existing Account

To delete an existing account, first select the account in the account Selection Combo Box, as following:

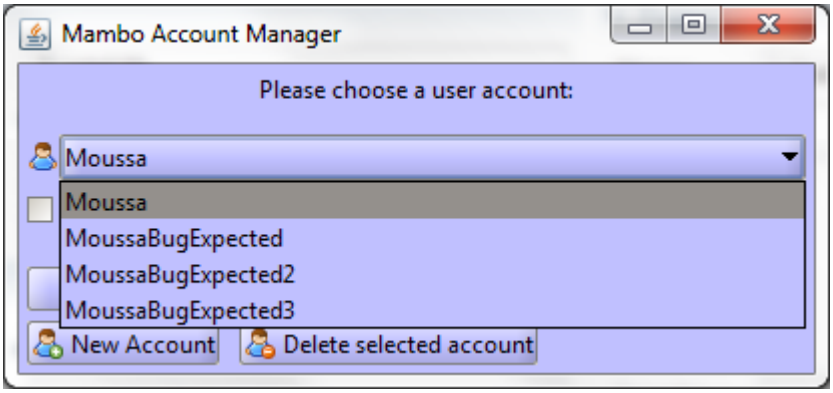
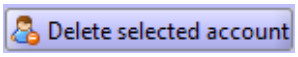


Figure 4: Account selection

When the account is selected, click on 

button to delete it. If you do it, you won't be able to use this account any more (and no other user either), because the account is definitely removed from list. The account deletion doesn't involve the corresponding directory (neither its content) deletion.

If you want to see your account path, you can check *Show account path*.

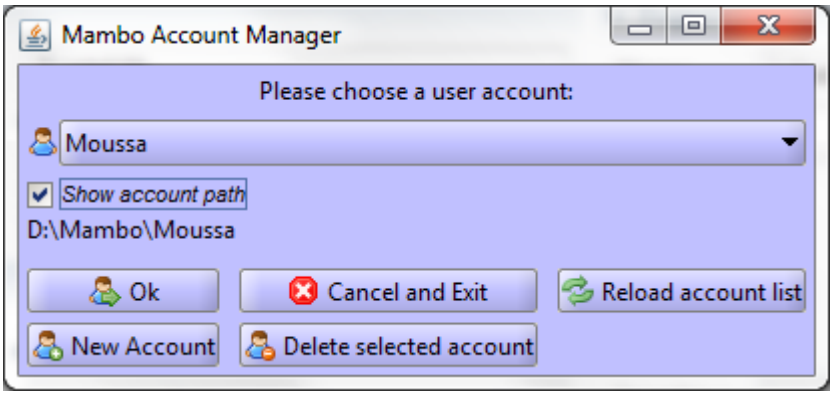


Figure :Show account path

Launching application with an existing account

To launch application with an existing account, first select the account in the account Selection Combo Box, as presented in *Figure 4: Account selection*.

Then, click on *Ok* button, and you will reach the application main panel configured with this account (the account name is displayed in frame title).

Control archiving

This section deals with controlling archiving of Tango attributes through **Archiving Configurations** (henceforth **AC**).

An AC describes which attributes to archive and how, and consists of:

- General information (when the AC was created . . .).
- A list of attributes to archive.
- For each attribute, a set of HDB/TDB archiving modes.

ACs is accessed/used in the AC panel:

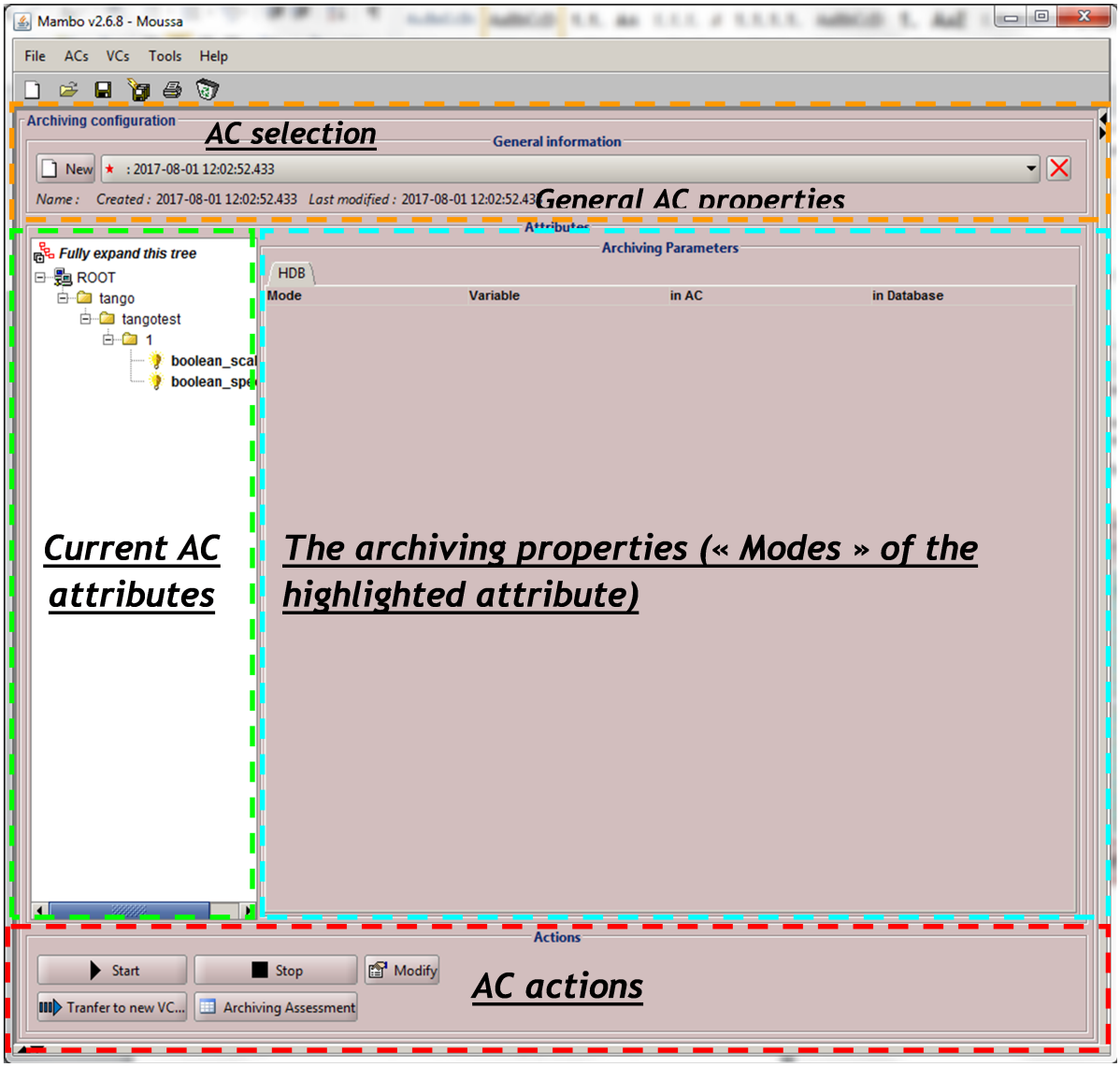


Figure 6: The AC panel

Creating/modifying an AC

The creation or modification of an AC is done the same way, using the *New*/*Modify* button respectively. We will see how to load an AC later.

Step 1: AC general properties

- Click the New/Modify button in the AC actions panel
- The following dialog appears

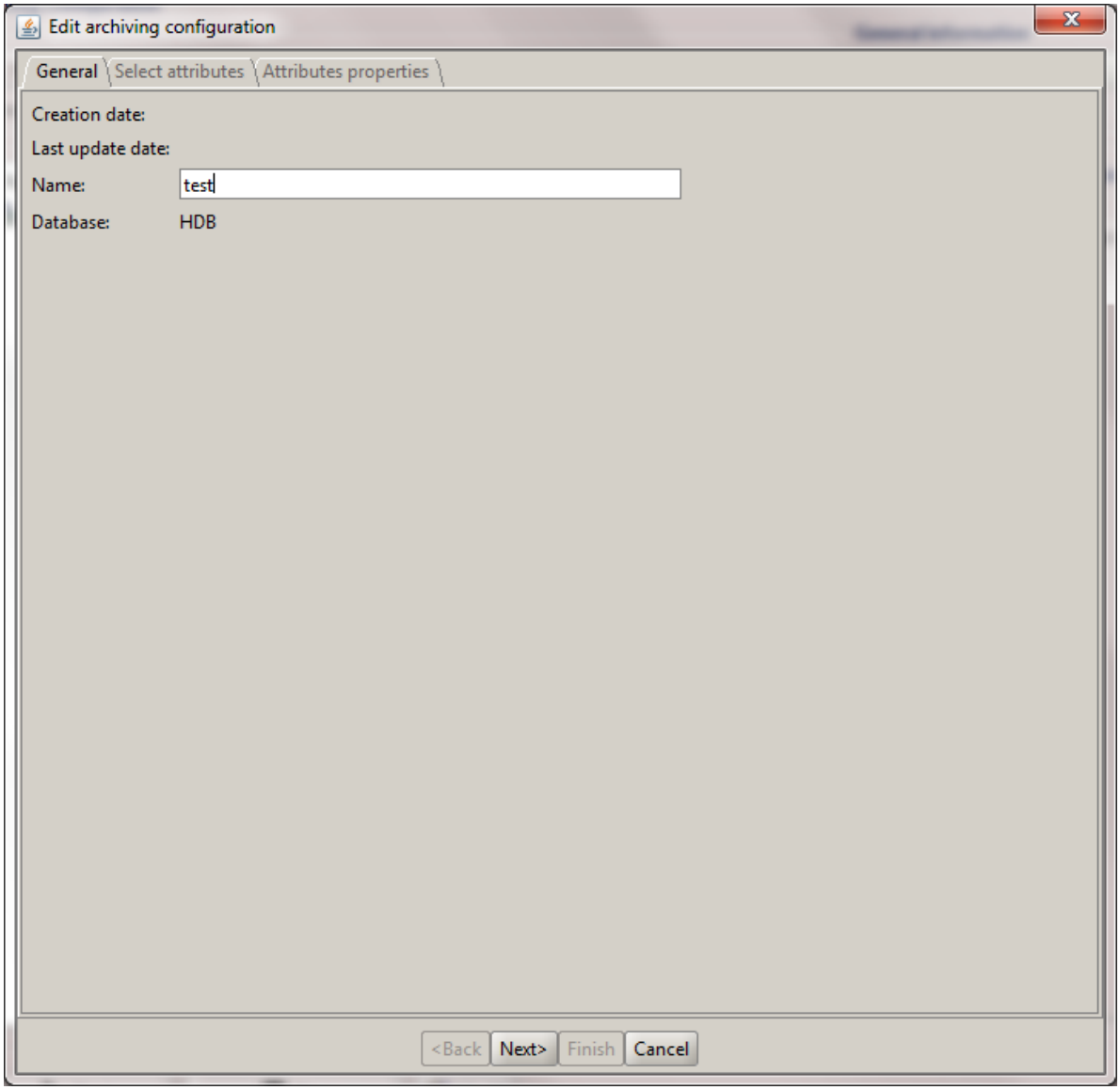


Figure 7: Step1, general properties

- You can choose a name for the AC. This field isn't mandatory but it is recommended to fill it, to better identify your different ACs later.
- The creation and update dates are filled or modified automatically upon saving an AC. Click next

Step 2: Select the AC attributes

There are 2 ways to select the AC attributes. The user can choose a selection method in the Options.

Step 2, method 1:

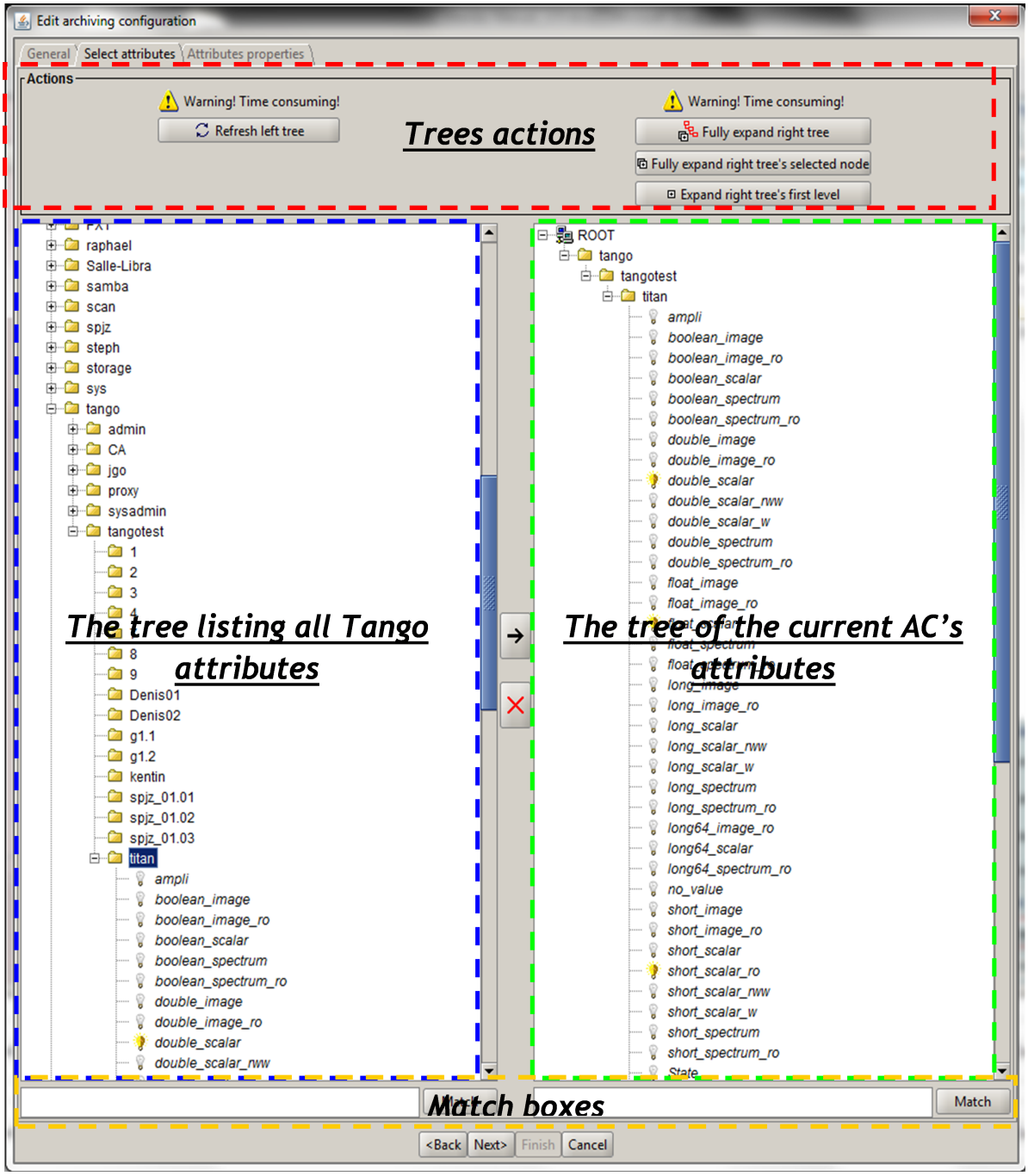


Figure 8: Step 2, select AC attributes (standard method)

- The left tree lists the Tango attributes. The right tree represents the current attributes of the edited AC. Both trees have the same structure of TANGO_HOST/Domains/Families/Members/Attributes.

Warning: The left tree listing the Tango attributes is initially only loaded up to Members level to avoid overloading. The attributes are loaded upon user selection of a Member (device) node.

- Adding attributes to the current AC:
  - Select a tree node to expand it.
  - Select attributes in the left tree then press the right arrow to add them to the current AC.
  - Select a node in the left tree then press the right arrow to add all attributes under this node to the current AC (only works if those attributes were loaded).
- Removing attributes from the current AC:
  - Select attributes in the right tree then press the red X to remove them from the current AC.
  - Select a node in the right tree then press the red X to remove all attributes under this node from the current AC.
- Using "Match" to filter attributes:

  The user can enter an expression of the form D/F/M and click match to filter attributes in the left and right trees.

  D represents the Domains; F represents the Families, and M the Members. All trees can contain the joker character '*'.

  Since the tree loading is initially only up to Member level, it's not possible to filter up to Attributes level.

Step 2, method 2:

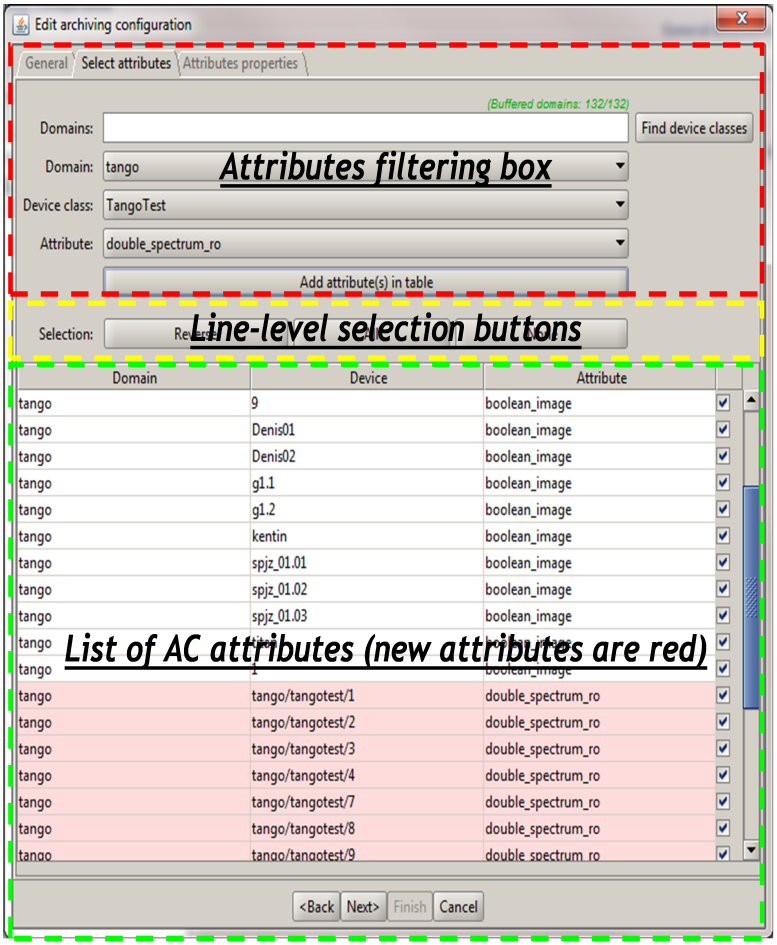


Figure 9: Step 2, select AC attributes (alternate method)

- Attribute selection and automatic attributes adding:
  - Choose a Domain. This refreshes the list of possible Device classes for this Domain.
  - Choose a Device class. This refreshes the list of possible Attributes for this Domain and Device class.
  - Choose an Attribute and press Add attribute:
  - All Attributes
    - with the selected name

    AND

    - belonging to any Device of the selected Class and Domain are added to the current AC's list of attributes.

  All new attributes are red until the AC is saved.
- Line level sub-selection of loaded attributes:

  Each attribute is initially checked, but this check can be removed by the user. When the user clicks on next, all unchecked attributes will be removed from the current AC.
- Click *Select All* to select all lines.
- Click *Select None* to select no line.

- Select lines in the list (CTRL and SHIFT are usable), then click *Reverse for selected lines* to reverse the checked/unchecked status of all selected lines.

Step 3: Set the AC attributes properties

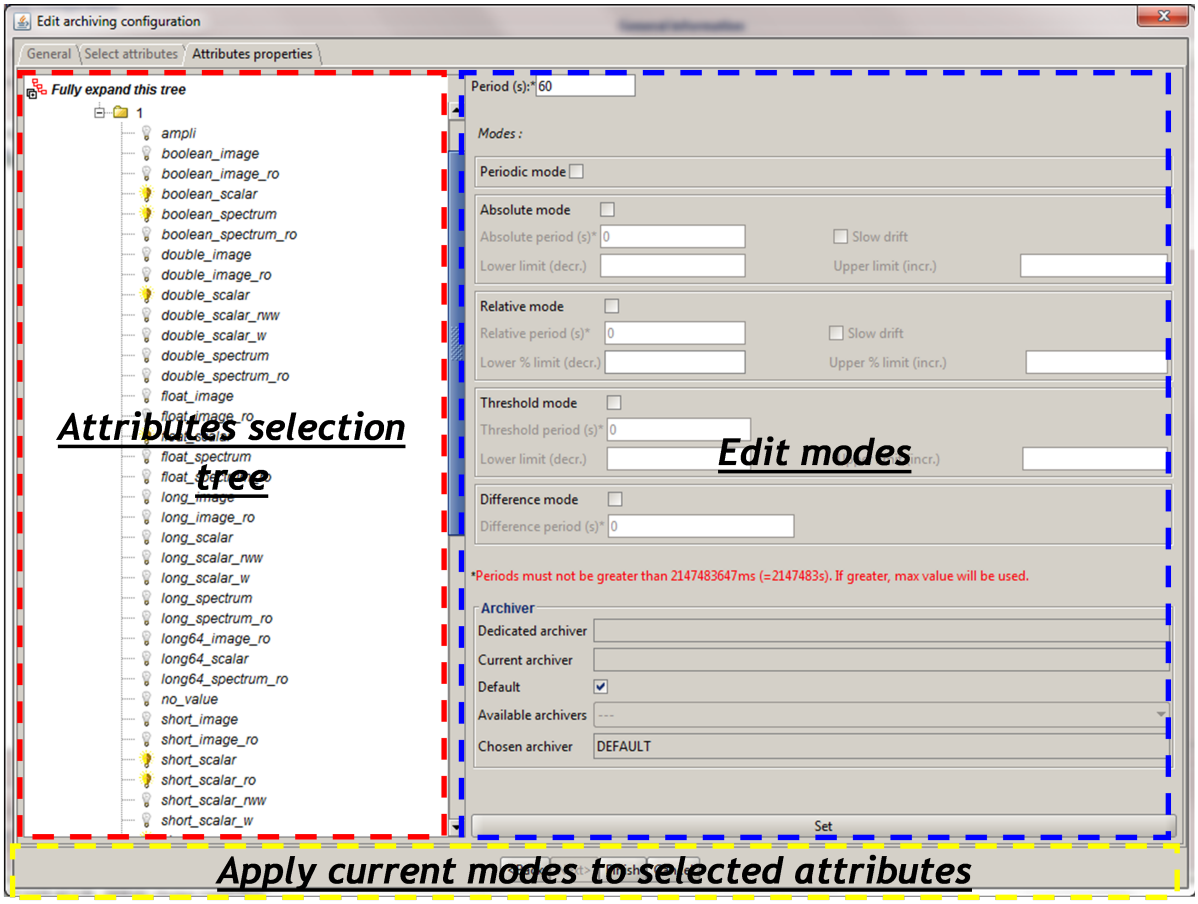


Figure 10: Step 3, selecting archiving modes

- A general description of the properties setting process:

  The way you set up archiving modes for each attribute is as follows:

  - Select a group of attributes.
  - Edit HDB/TDB modes.
  - Call *Set* for the current group of attributes: the currently displayed modes are applied to all attributes of the selection.
  - Repeat with next group of attributes.
  - End edition by clicking *Finish*.

- Attributes selection

  The setting up of archiving modes can be "factorized" for a selection of attributes. All attributes of the selection will be set up with the currently displayed properties when the user presses *Set*.

  The multiple selections can consist of:

  - A manual select at Attributes level (CTRL and SHIFT are usable).
  - Selection of an upper node level: all Attributes nodes under this node will be set up.
  - A combination of the two.

- Unset attributes and default values

  Attributes which haven't received any Mode yet (unset) are displayed in *Italic*. Attributes which have (set) are displayed in **Bold.**

  Clicking on a set attribute displays its archiving modes.

  Clicking on an unset attribute displays default archiving modes.

- Controls upon AC validation.

  The user can not choose any and all combination of modes, nor any and all numeric values for each mode. Thus, controls are performed upon call to "Finish":

- If any HDB (resp. TDB) mode is chosen for a given attribute, it must also have the basic HDB (resp. TDB) Periodic mode.
- Any unset attribute will be removed from the AC; the user is prompted to continue editing the AC, or ignore them.
- Numeric values are controlled for each mode.

  When the validation is over, the AC is displayed in the application's AC panel.

The opened ACs menu

Mambo can have several opened ACs at once, even if only one is displayed on screen at a time. A drop-down menu allows the user to choose an AC in the list of opened ACs:

- Each time the user loads an AC, it's added to the top of the opened ACs list (the older ACs are shifted downwards in the opened ACs list)
- The list identifies ACs by their name and date of last update. When the user selects an AC, it becomes the current AC, and its general information and attributes are displayed.
- To remove an AC from the list, push the red X button (this will do nothing if the list is empty or only has 1 element). The next AC in the reduced list (i.e. the one that was added to the list the most recently) is automatically displayed.
- The list can hold no more than *[MAX_NUMBER]* ACs, this number can be defined in the AC tab of the options panel (default=5). If the list's length reaches *MAX_NUMBER,* the oldest AC will be removed from the list when needed.
- ACs that have unsaved modifications are identified by a red star.

  If Mambo has the *History save* option turned on, the list of opened ACs will be saved at shutdown, and loaded at startup.

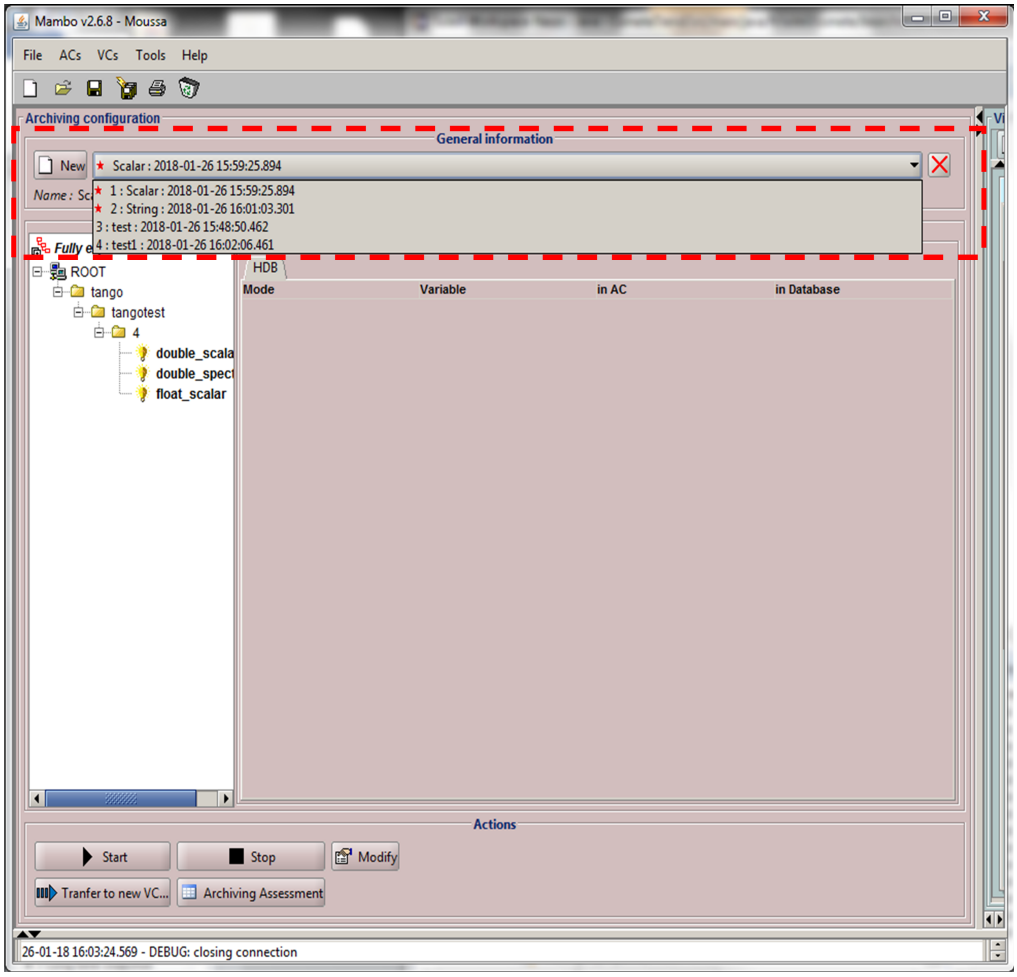


Figure 11: Acs menu

Starting/Stopping archiving

The Current Archiving Configuration detail panel

Attributes are displayed differently in selection trees, with respect to their current archiving status. The display shows their current status in DB, not their archiving modes in the current AC.

- Attributes which aren't being archived are displayed as off bulbs:
- Attributes which are being archived only in HDB are displayed as on, yellow bulbs:
- Attributes which are being archived only in TDB are displayed as on, brown bulbs:
- Attributes which are being archived in HDB and TDB are displayed as on, bicolor bulbs:

Starting archiving

- Create/Load an AC or just use the current AC. The AC that will be used is the one displayed in the application's AC panel.
- Call Start to start archiving each attribute by all its modes.

Stopping archiving

- Create/Load an AC or just use the current AC. The AC that will be used is the one displayed in the application's AC panel.
- Call Stop to stop archiving of each attribute. All archiving modes will be stopped.

In both cases, a success/failure message confirms the action.

Displaying an AC

The Current Archiving Configuration

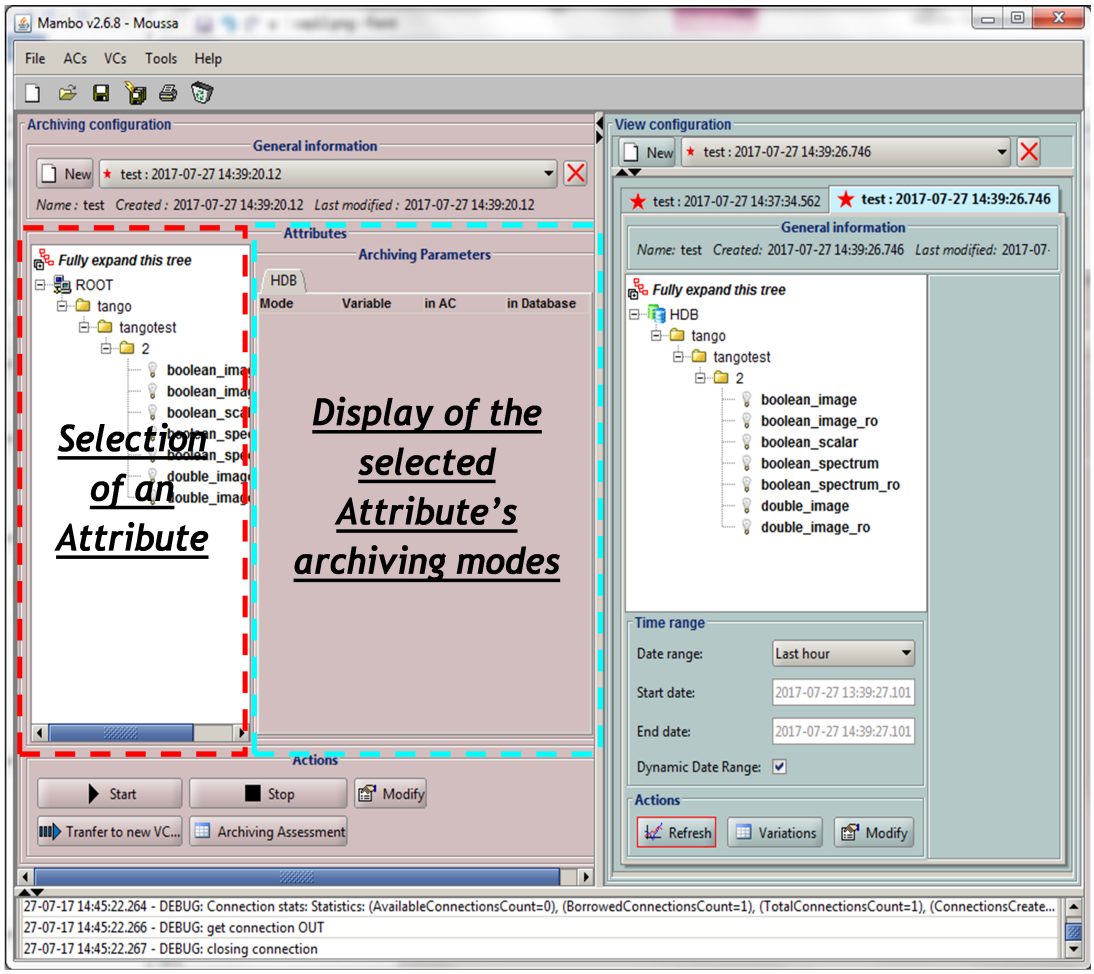


Figure 12: Detail of the current AC

This panel represents the current Archiving Configuration.

On the left, a tree lists all of the AC's attributes.

When the user selects a given attribute on this tree, its HDB and TDB modes are displayed on the right sub-panel.

This displays the attributes modes individually. To get a global view of the AC's Archiving Modes, use the "Archiving assessment" command.

The Archiving assessment window

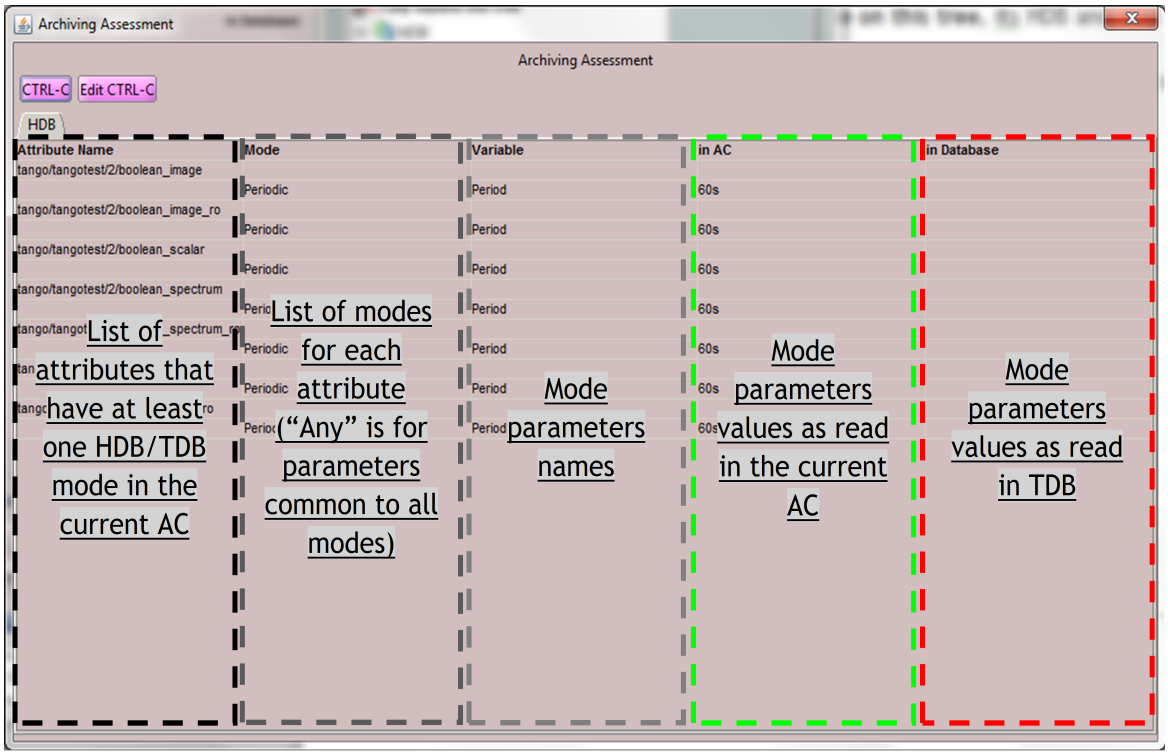


Figure 13: Global view of all the modes of an AC

This window sums up the current Archiving Configuration in two tabs (one for HDB, one for TDB).

For each attribute contained in the current AC, its archiving modes are detailed, and if the attribute is being archived, can be compared to the modes values found in HDB (resp. TDB).

Saving/Loading an AC

Saving an individual AC

Archiving Configurations are saved /loaded as XML files, with the .ac file extension.

In the menu, select *ACs* → *Save* or *File* → *Save* → *Archiving configuration*:

Saved operations work like they do with Word (for example):

- The first time a file is saved, the user is prompted to choose a path and file name. Initially the file chooser dialog is in the "ac" subdirectory of the Mambo working directory. The ".ac" file extension is automatically added if the user doesn't.

- If a file was already saved, it will automatically be saved in the same file the next time the "Save" menu is selected.
- The user can still specify a different file, by using the *ACs* → *Save as* menu (or *File* → *Save As* → *Archiving configuration*).

When an AC is loaded, it becomes the current Archiving Configuration and it is added to the list of opened ACs.

In both cases, a success/failure message confirms the action in the log panel.

Saving all opened ACs

It is possible to save all opened ACs. Only modified ACs will be saved (i.e. the ACs that has a "red star").

In the menu, select *ACs* → *Save All*:

For each modified AC, the save operation will follow the same rules as an individual save:

- If the file has already been saved before, it will be saved silently.
- Otherwise, the user will be prompted to choose a directory and file.

  In this case, the ACs that are being saved are successively selected before each file chooser prompt (so that the user knows which AC he's choosing a path for).

The generic *Save All* menu item in *File* → *Save All* does this and the same thing for opened VCs.

Transfer to VC

This functionality is used to quickly create a VC on the same attributes as a given AC (for example, when the user wants to monitor the values of attributes after starting archiving on an AC).

It will create a VC automatically, with the following properties:

- Its attributes list is the same as the AC the "Transfer to VC" functionality is used on
- The attributes values are all extracted from HDB, even if some of the AC's attributes were only archived in TDB (obviously, the display for such an attribute will be empty).
- The date range of a one-hour range ends when the VC is created
- Every Attribute has an automatically determined color (if there are more attributes than colors to choose from, different attributes will have the same color).

Exploit archived data

This section deals with controlling exploitation of archived attributes (HDB/TDB) through **View Configurations** (henceforth **VC**).

A VC describes which attributes to display and how, and consists of:

- **General information (when the VC was created, the date range to** extract . . .).
- A list of attributes to display.
- For each attribute, a set of display properties.

The archived attributes of a VC either all come from HDB or all come from TDB, since it wouldn't make sense to display HDB and TDB attributes in a common plot,

VCs are accessed and used in the VC panel:

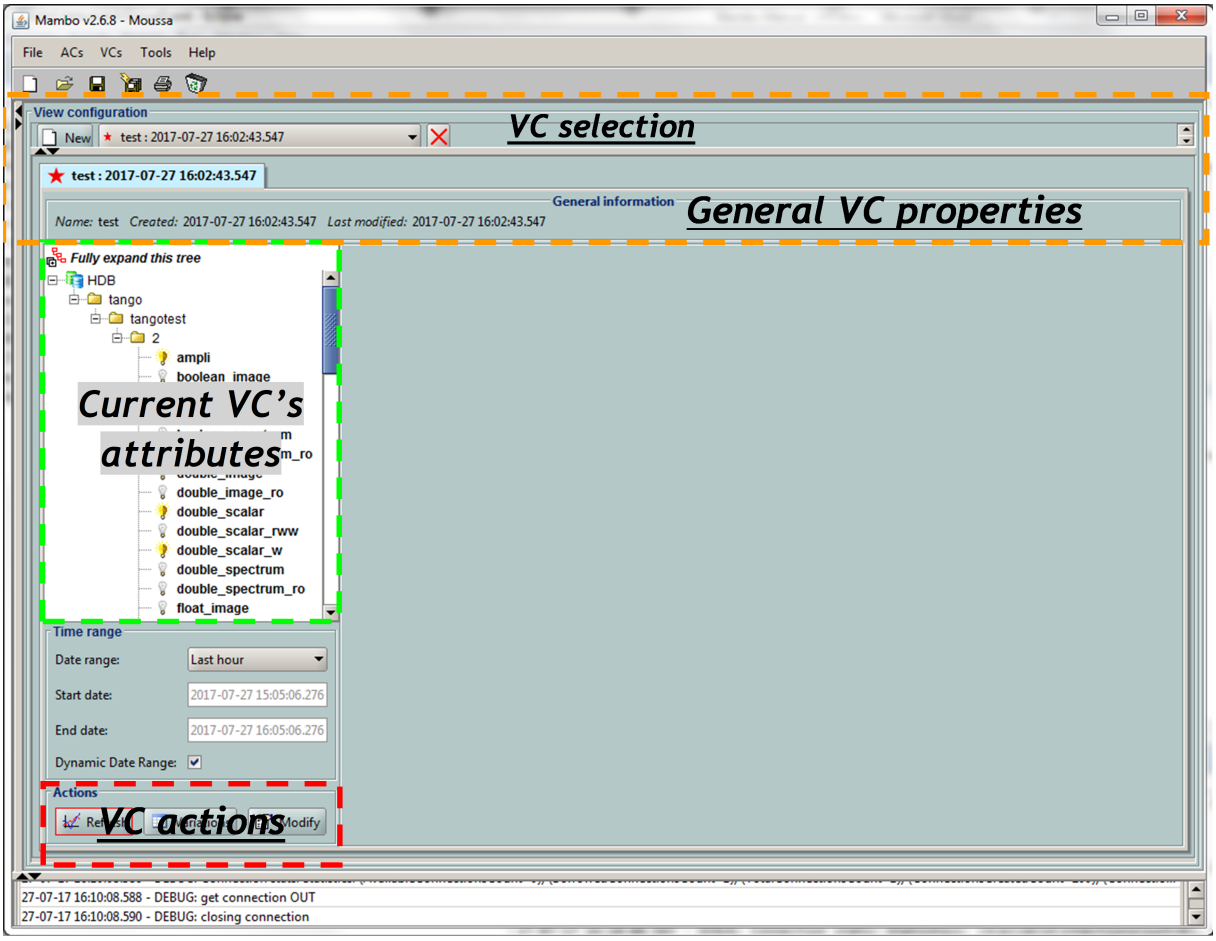


Figure 14: The VC panel

Creating/modifying a VC

Creation or modification of a VC is done the same way, using the *New*/*Modify* button respectively. We will see how to load a VC later.

Step 1: VC general properties

Those properties don't depend on a specific attribute. They are common to all VC attributes or describe the VC.

Click the *New*/*Modify* button in the VC actions panel, the following dialog appears:

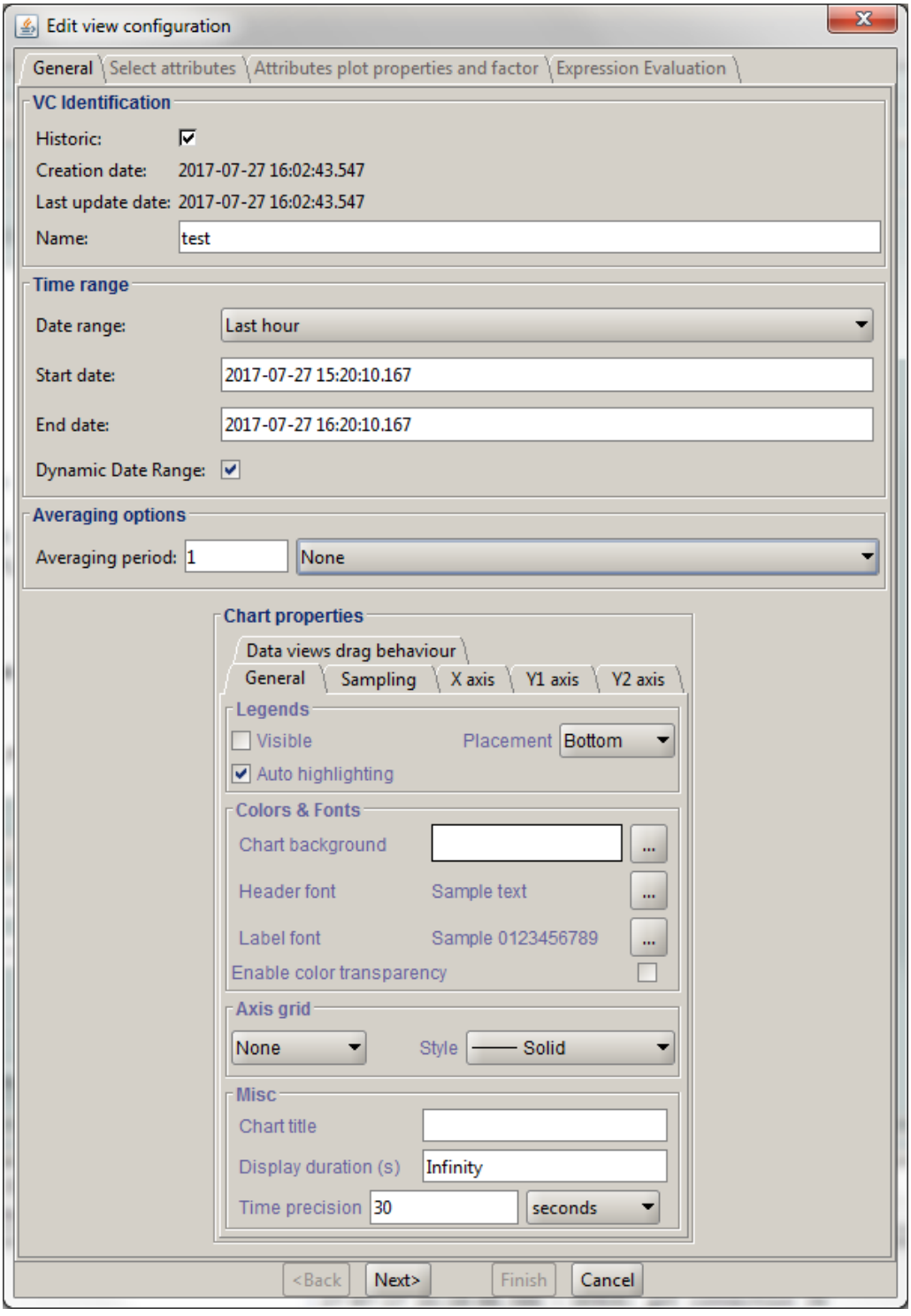


Figure 15: Step 1, general properties

Step 1.1: VC properties

- You can choose a name for the VC. This field isn't mandatory but it is recommended to fill it.
- The creation and update dates are filled automatically upon saving the VC.
- The user must define a date *range* (Start/End dates) to extract data from. This can be done either manually, or by

selecting a *Since xxx* item in the "Date range" drop-down menu.

In the latter case, the user can check the "Dynamic date range". What this option does is recalculate the Start/End dates at each edition or refreshing the VC (see 6.3 *Displaying a VC*).

Example: At 09:00 the user chose "Last 1 hour" as the "since option".

The date range will be [08:00-09:00].

If the VC is then edited at 11:00, and *Dynamic date range* was checked, the date range will be [10:00-11:00]; otherwise it will still be [08:00-09:00].

- If the *Historic* checkbox is checked, VC attributes will be chosen among HDB attributes (otherwise, TDB).
- If the *History* checkbox is not checked, you are in TDB mode so you can import long term snapshot. In this case you can't modify start and end date because the snapshot configures it himself.

Step 1.2: General chart properties

- The user can define global chart properties (title, background . . .).

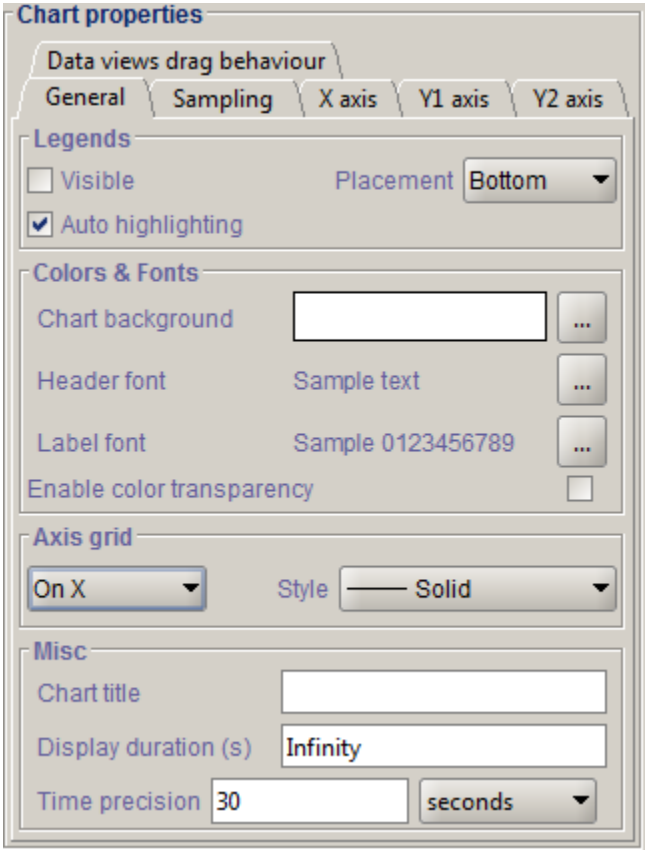


Figure 16: Step 1, general chart properties

Step 1.3: Y1 axis properties

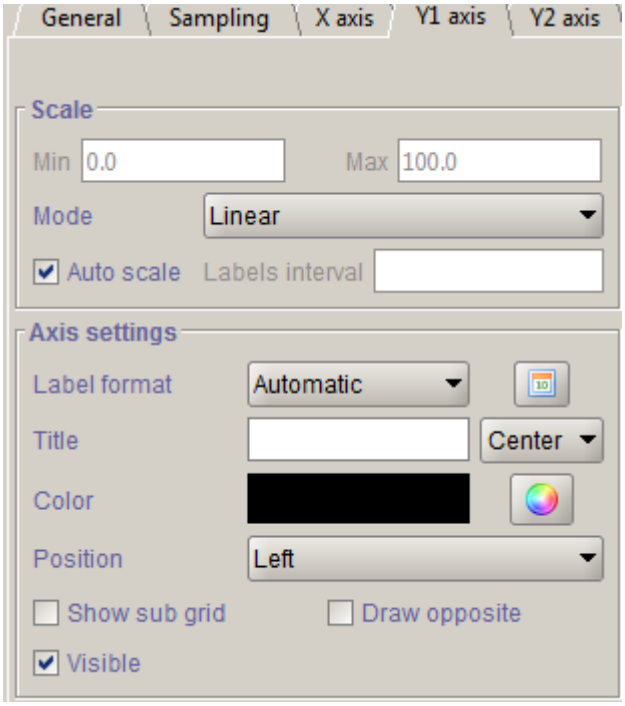


Figure 17: Step 1, Y1 axis properties

- The user can define the left vertical axis properties (title, scale, ...).

Step 1.4: Y2 axis properties

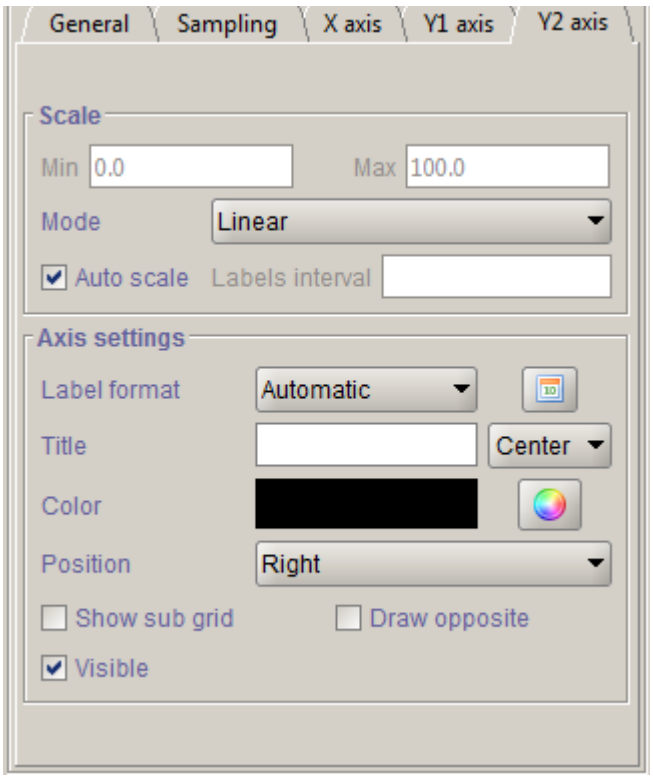


Figure 18: Step 1, Y2 axis properties

- The user can define the right vertical axis properties (title, scale, ...).

Step 2: Select the VC attributes

The attributes selection works the same way as an AC attributes selection, except for two differences:

- The only available selection mechanism is the tree selection

  The trees are initially loaded to the attributes level.

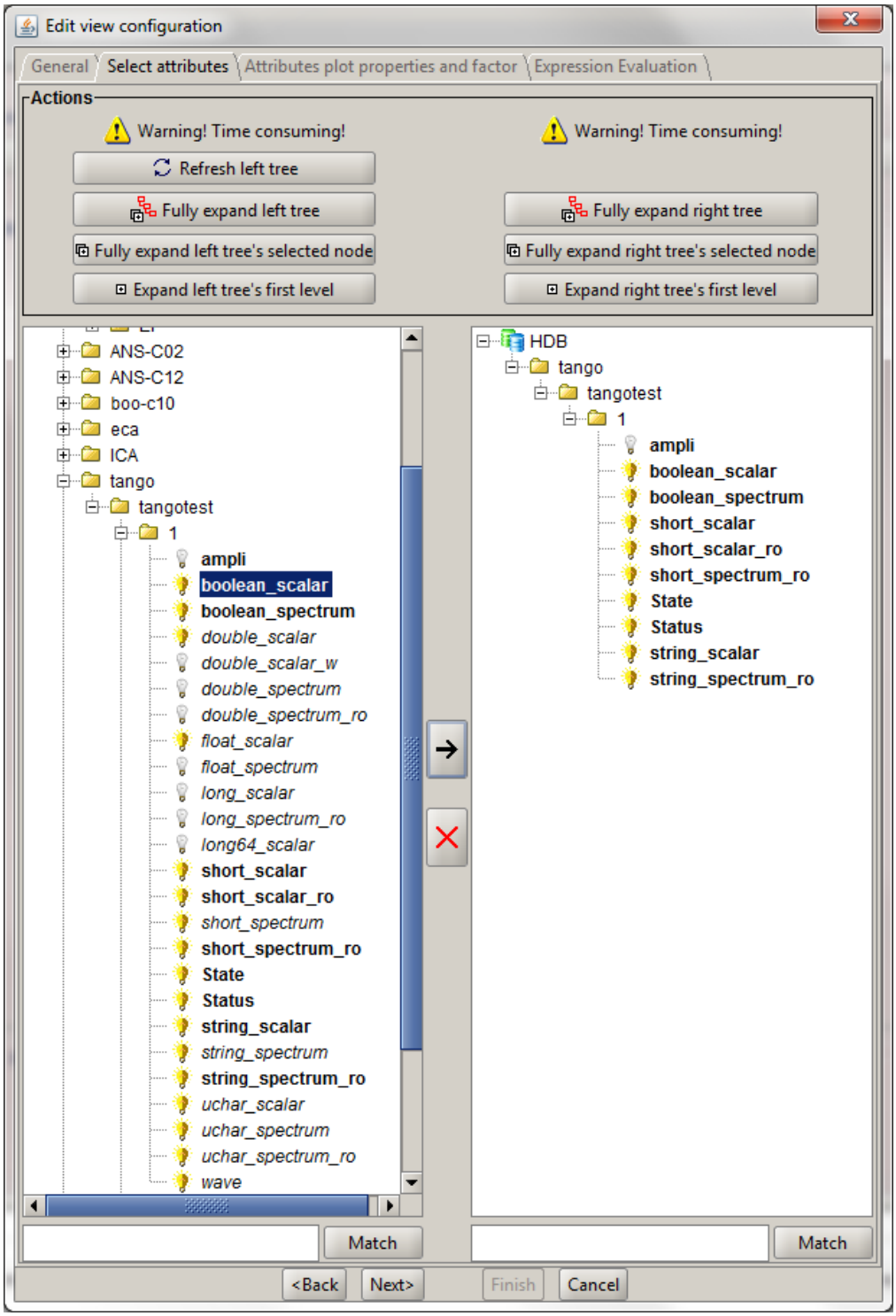


Figure 19: Step 2, selecting attributes

- The left tree lists the HDB (resp. TDB) attributes. The right tree represents the current attributes of the edited VC. Both trees have the same structure of HDB (resp. TDB)/Domains/Families/Members/Attributes.

> **Warning:** Unlike AC trees, VC trees are initially loaded to the Attributes level since archived attributes are a small subset of all Tango attributes.

- Consequently, the match functionality works up to attributes names: one can use a D/F/M/A criterion.

Step 3: Set the VC attributes properties

General flow

To set the VC attributes, user can refer to the ACs, it's the same process.

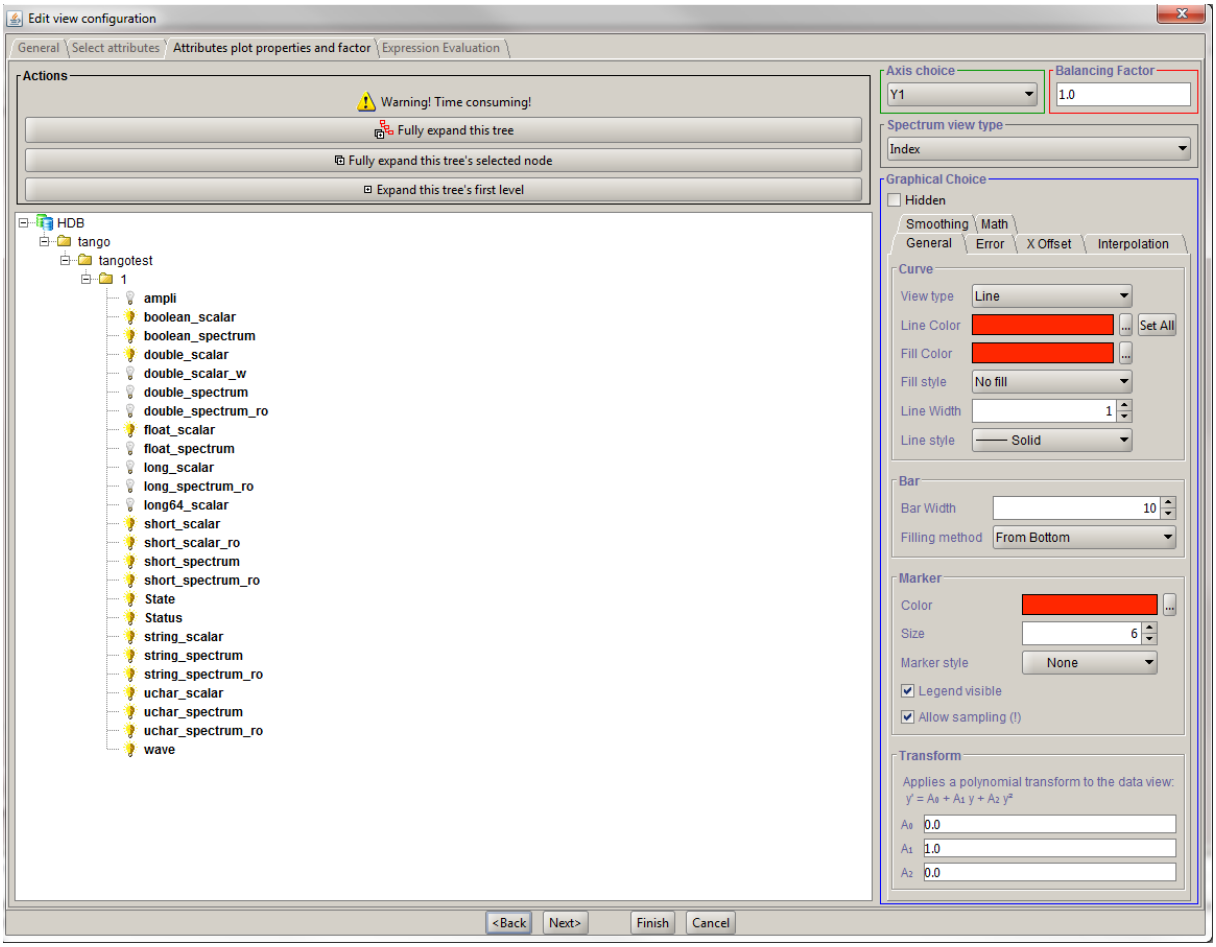


Figure 20: Set the vc attributes

Step 4: Controls upon VC validation

- Any unset attribute will be removed from the VC; the user is prompted to continue editing the VC, or ignore them.
- The VC's date range is controlled

When the validation is over, the VC is displayed in the application's VC panel.

The opened VCs menu

The VCs menu opens in the same way as the ACs menu.

See *The opened ACs menu*

Displaying a VC

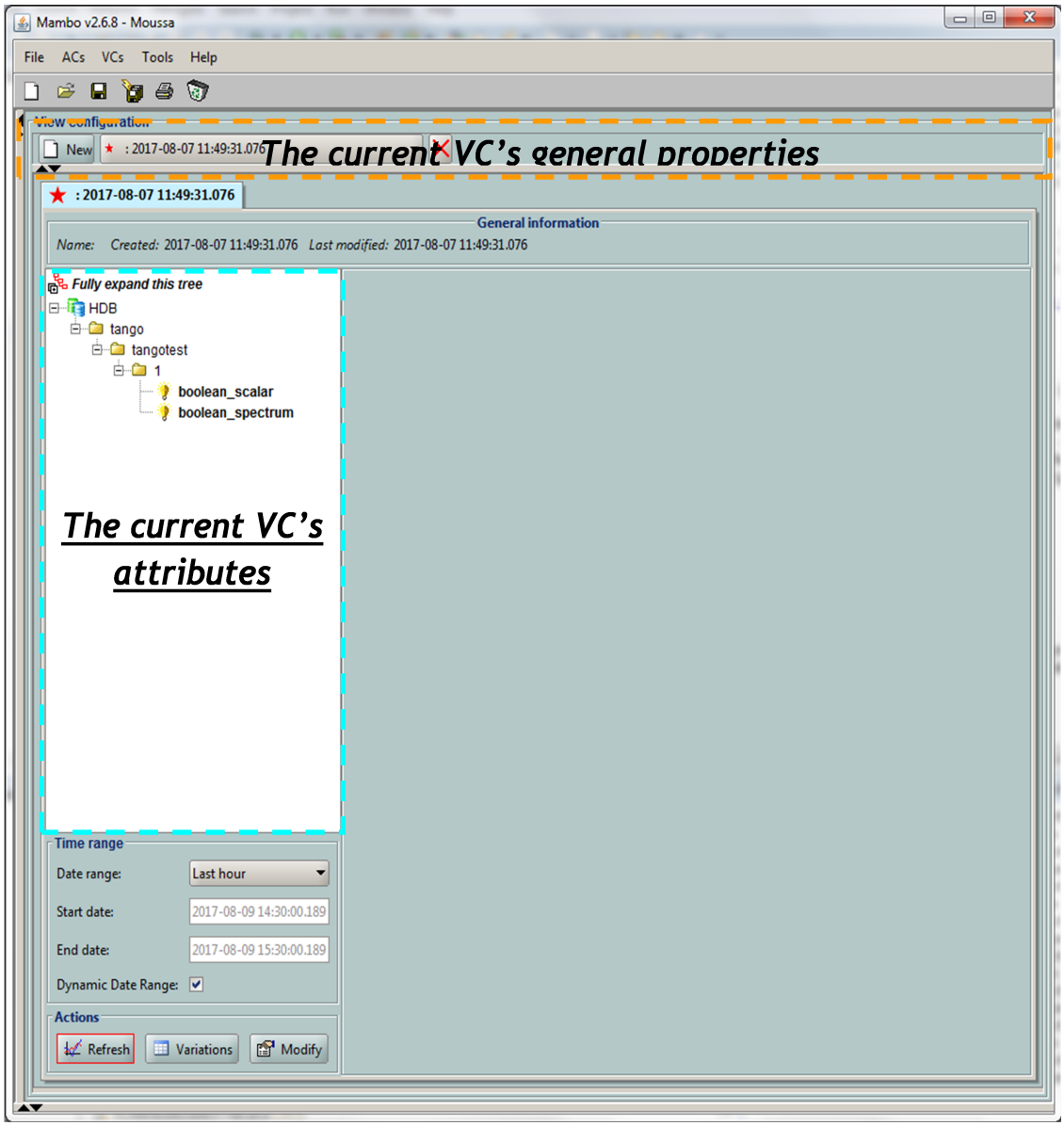


Figure 21: The current VC panel

This panel represents the current View Configuration; a tree lists all the VC's attributes.

To view specific parameters, the user must use the "Modify" button.

Thanks to the docking, users can see two generals information.

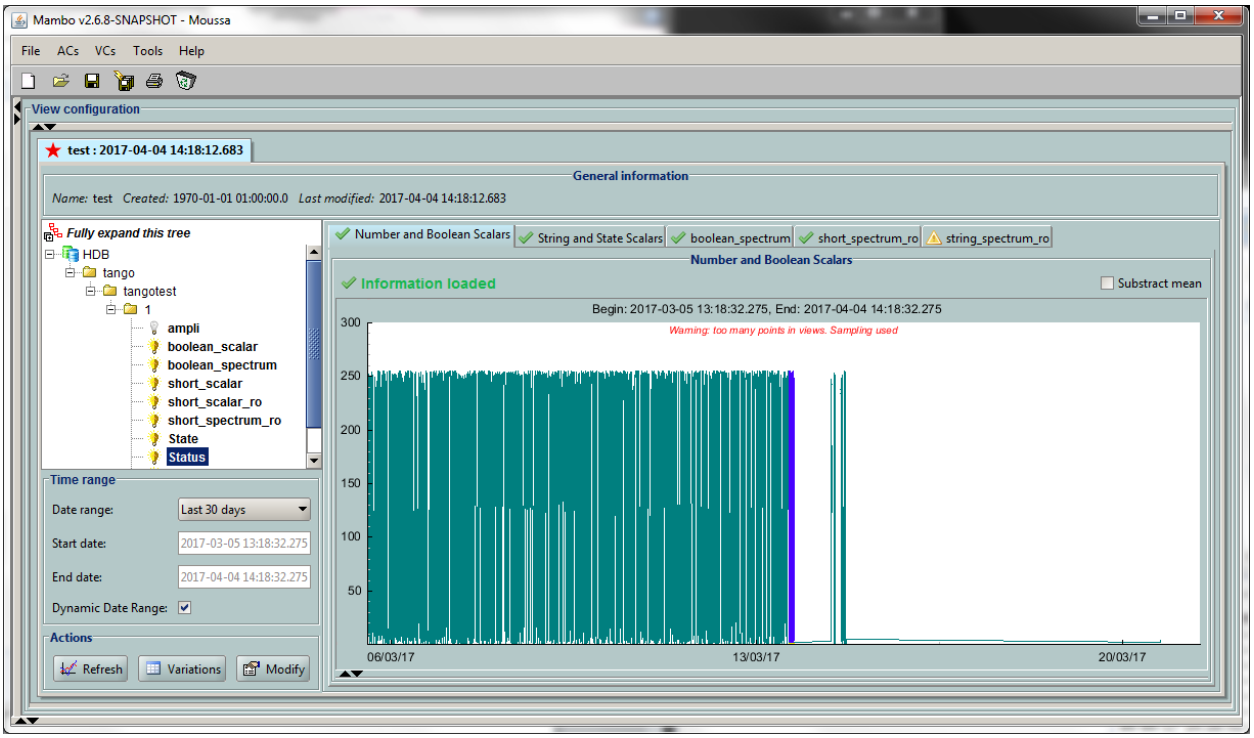
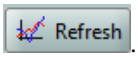
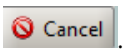


Figure 22: Extraction of Number and Boolean Scalars

This panel shows the attribute's extractions.

If the user wants to extract attributes, he must select his time range then press to the button refresh 

.

If he wants to stop the refreshing he will press to the button cancel 

.

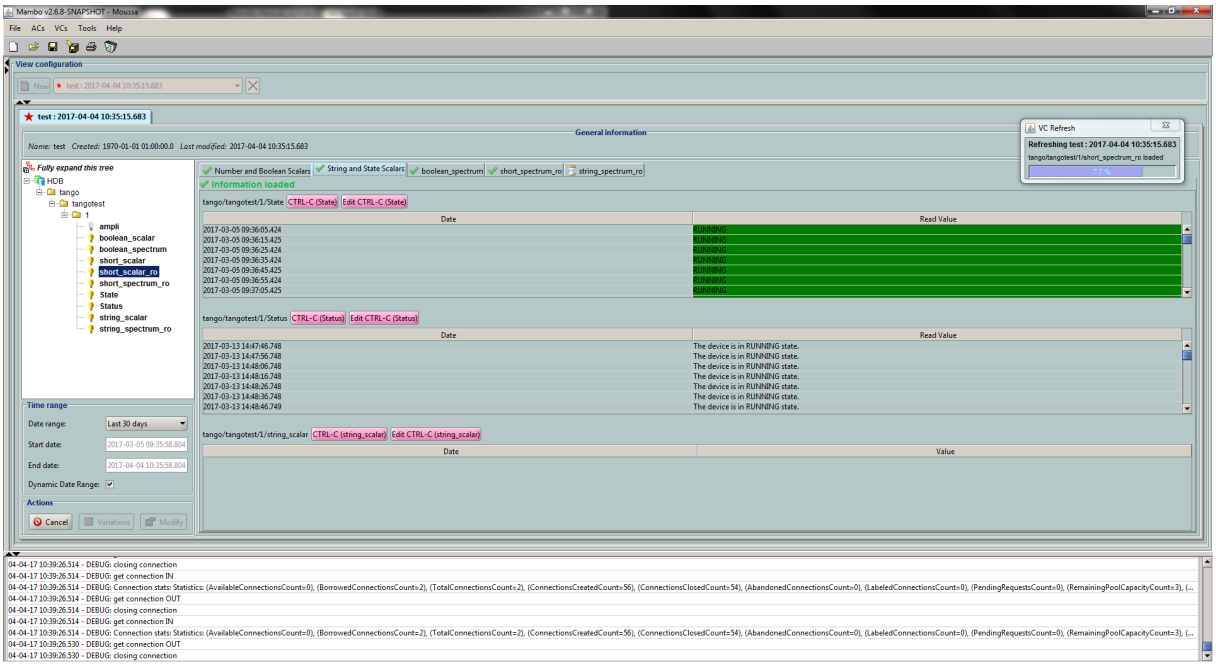


Figure 23: Extraction of String and State Scalars

This panel gives Extraction of the string and state scalars.

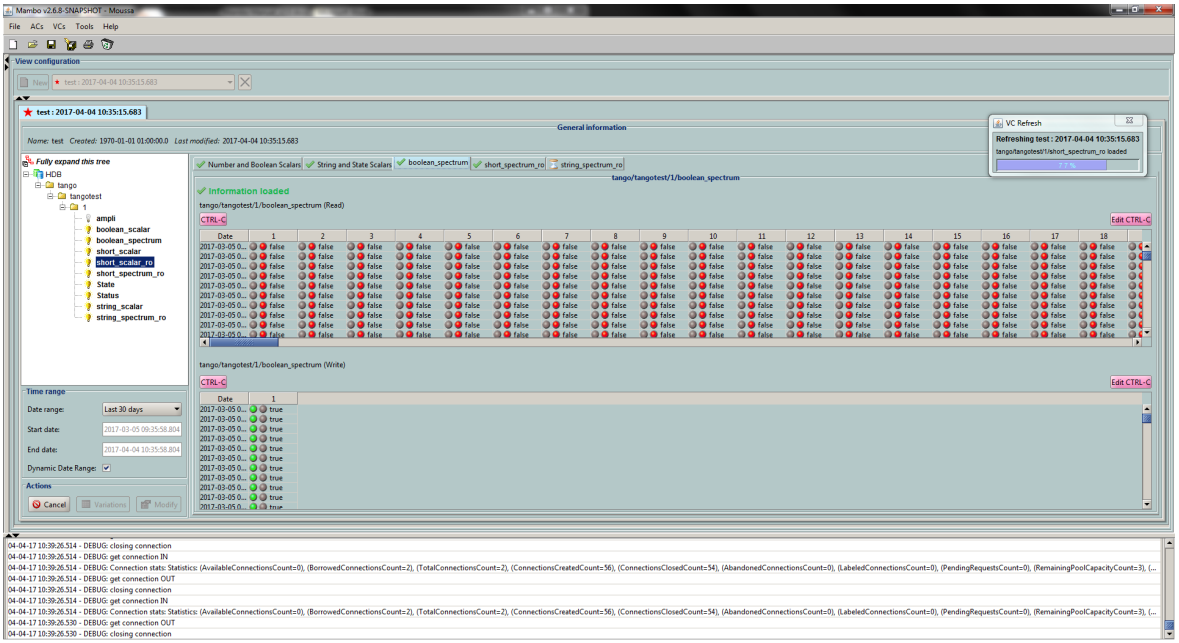


Figure 24: Extraction of Boolean Spectra

This panel shows the results of the Extraction of Boolean spectra.

Saving/Loading a VC

The user does the same thing as in ACs to save or load VC.

See *Saving/Loading an AC*.

Using the Variations functionality

The Variations functionality is a filter which comes between a VC and the final plot display.

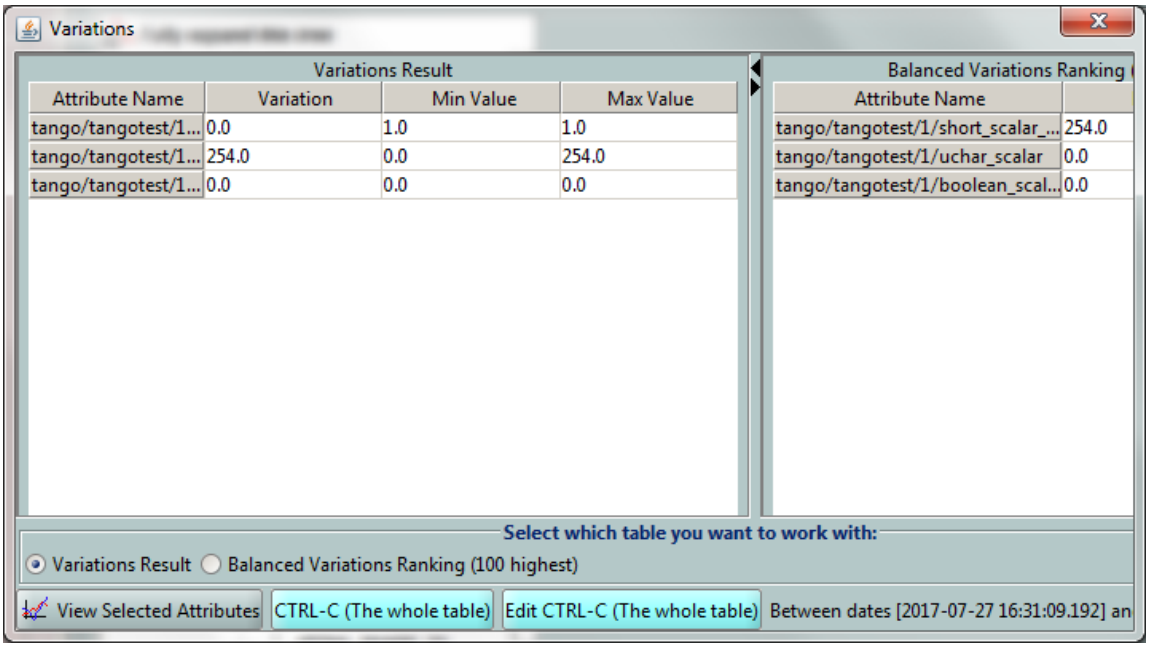


Figure 25: The "variations" attributes selection window

The Variations window lists the VC attributes and their "variation", which is defined as the difference between the min and max values in the VC's date range.

The user can then select attributes which variations that seem abnormal or interesting, and only plot those attributes.

The Variations window is made of two different tables:

- The first one shows for every attribute its minimum and maximum values, and its variation.
- The second one gives every attribute and its balanced variation.

Select lines in the left table, and press *View Selected Attributes* to display the filtered plot.

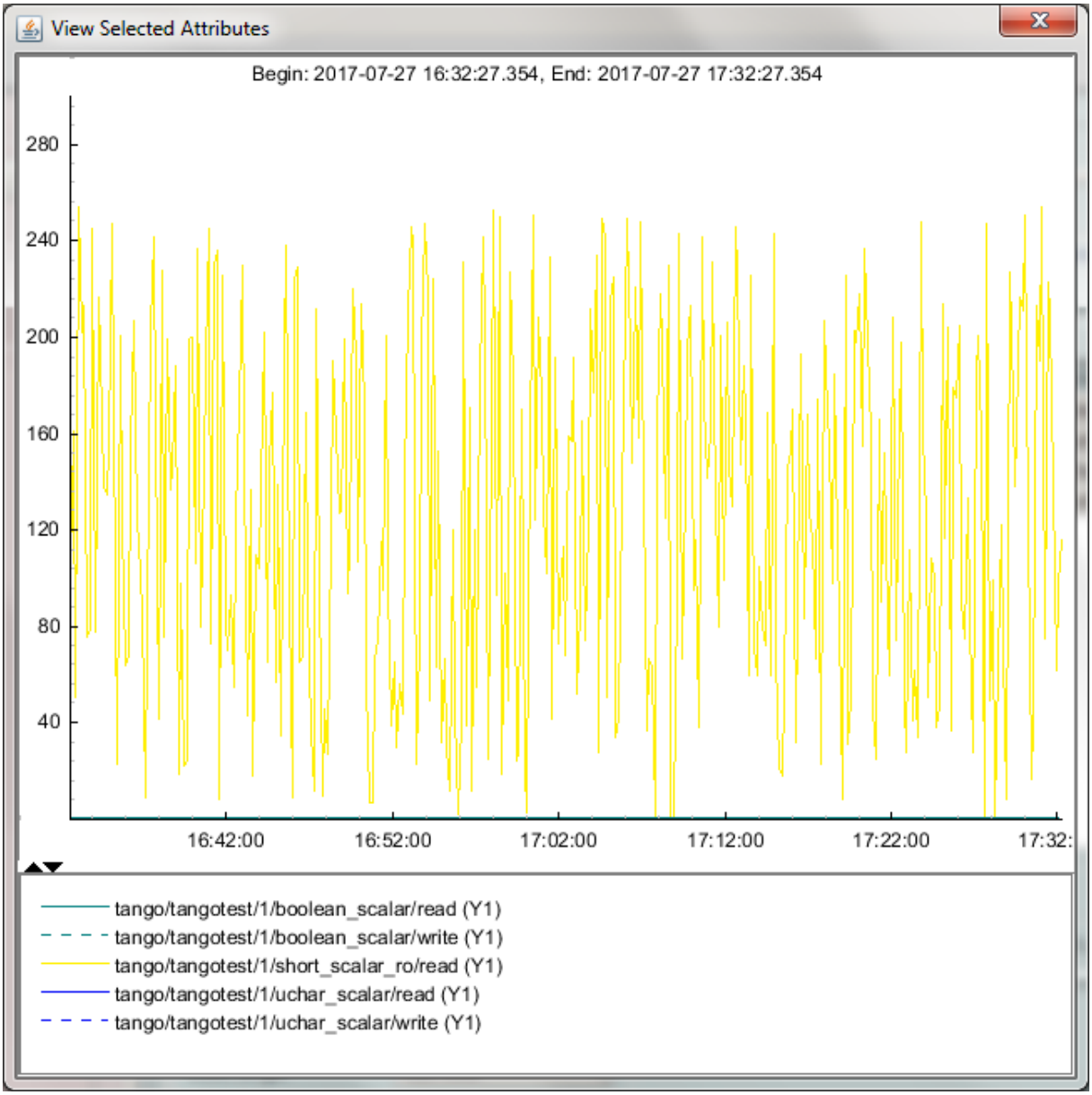


Figure 26: The View selected attributes

Options

Mambo manages global options. Those options are saved on application shutdown, and loaded on startup.

The Options menu is located in the Menu bar: *ToolsOptions*.

Application's history save/load options

These options define whether Mambo has a history, i.e. a persistent state when closed/reopened.

If *Yes* is checked, a XML History file will be saved in Mambo's workspace, and on next start up the current AC and VC will be loaded.

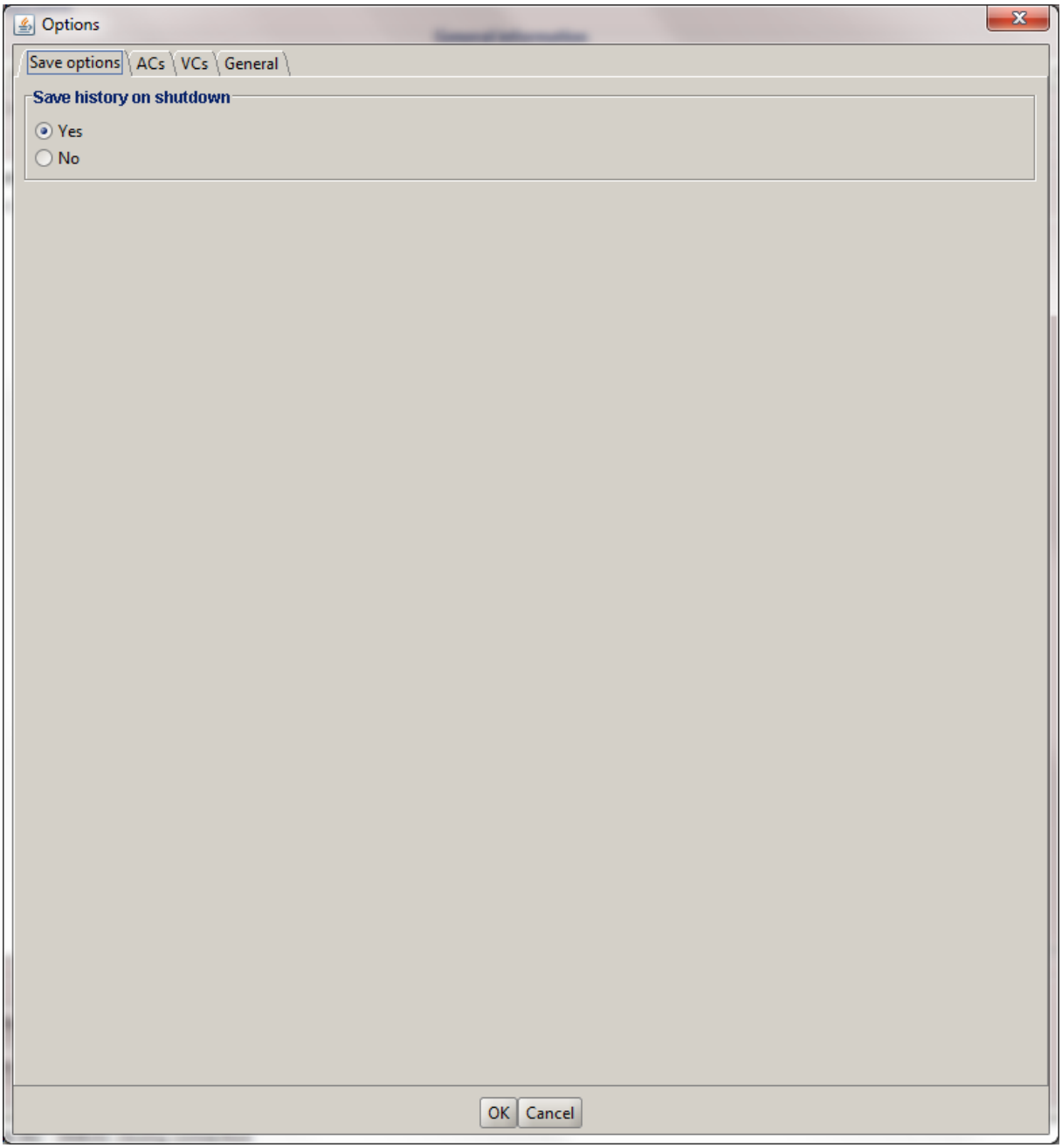


Figure : The history options

AC options

Options for all Archiving Configurations, the user can define:

- An attribute selection mechanism for AC edition (see 5 *Control archiving*).
- Default modes selection and values for HDB and TDB archiving (see *Creating/modifying an AC*).
  - The selected modes and values will be preset for all unset attributes.
  - The default values can be saved to /loaded from an Archiving Configurations Defaults file (.acd extension).
  - The "Restore defaults" button reloads the defaults with non user-modifiable predefined values.
- The size of the "AC stack", that is the maximum number of opened ACs.

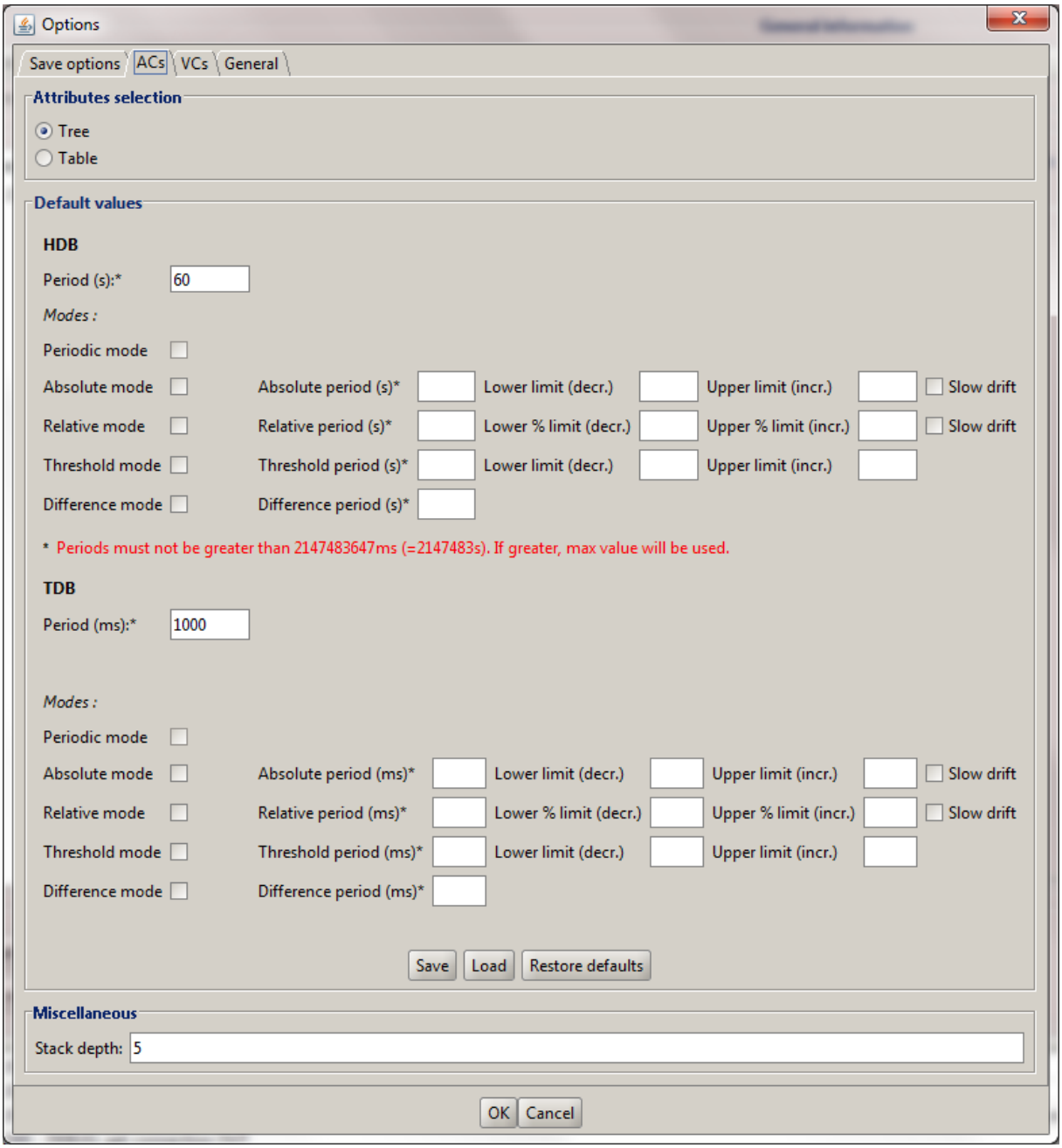


Figure : The AC options

VC options

These are the options for all View Configurations. The user can define:

- Whether he wants to display the Read value of attributes only, the Write value only, or both.
- The size of the "VC stack", that is the maximum number of opened VCs.
- The chart properties.
- The spectrum view (spectrum view type).

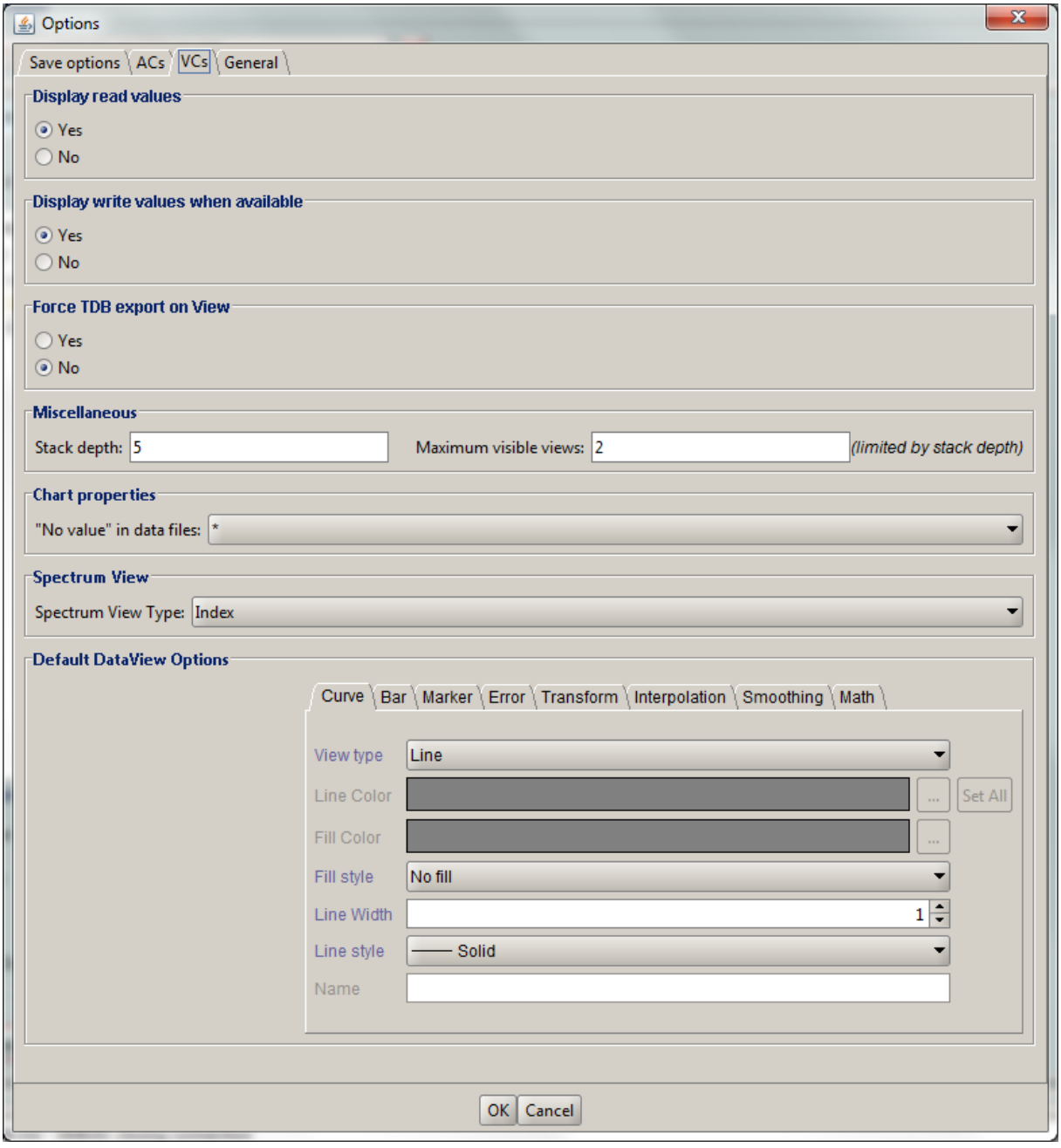


Figure : The VC options

General options

The user can define:

- The column separator for the tables
- The buffering of Tango attributes.

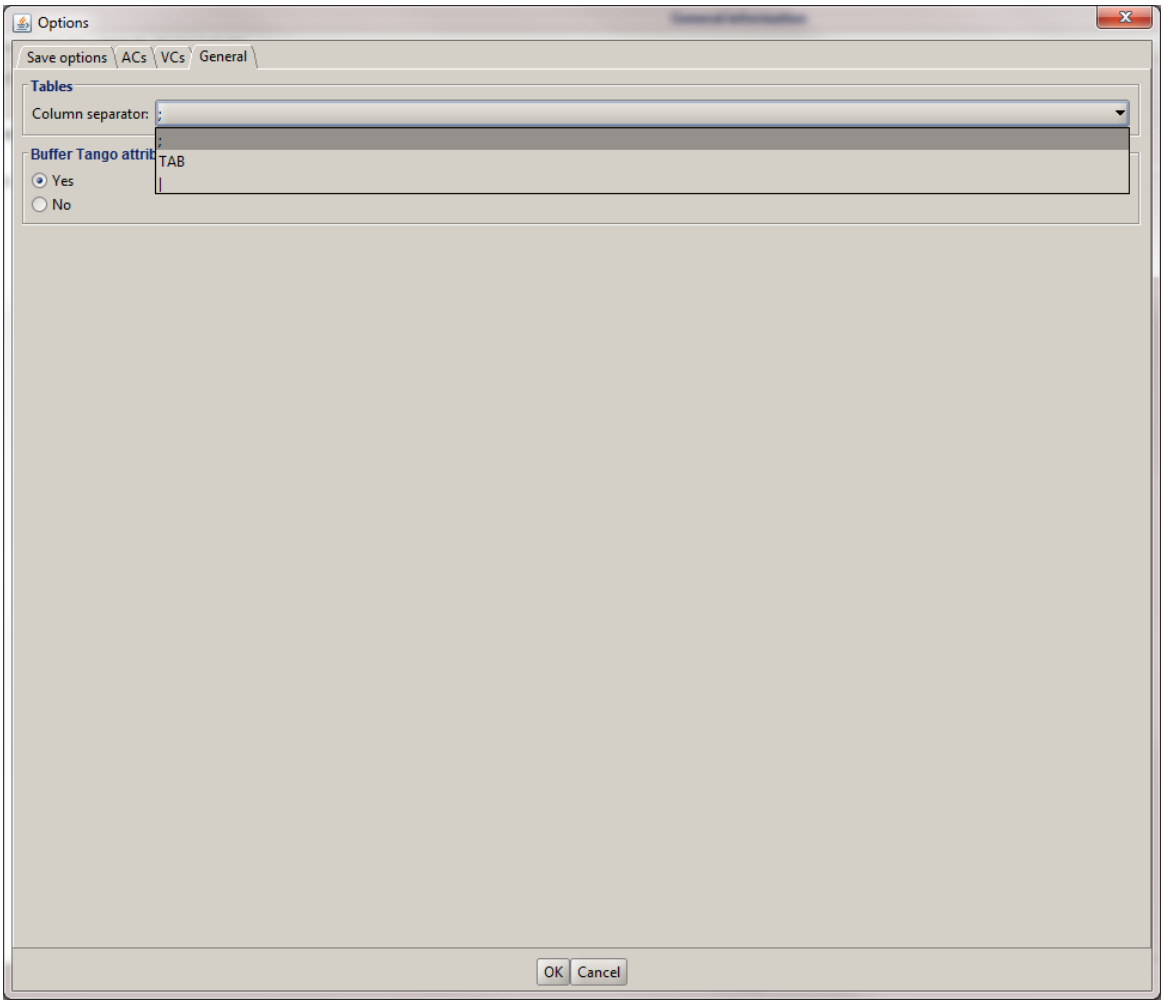


Figure : General options

The Mambo toolbar

The toolbar is located under the menu bar, and consists mainly of a set of shortcuts to often used functionalities.

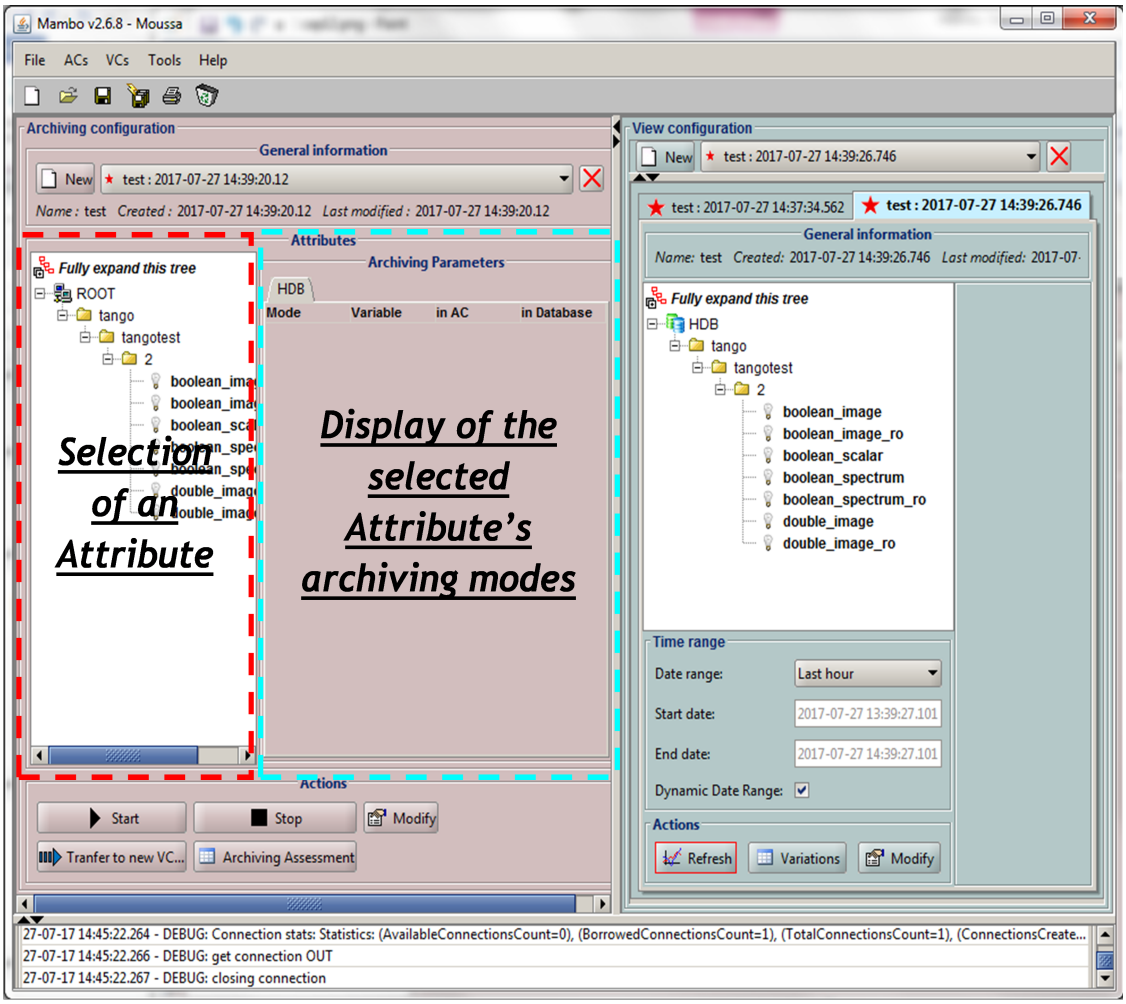


Figure 31: The Mambo toolbar

- is a shortcut to create a new Archiving configuration or a new View configuration.
- is a shortcut to load an Archiving configuration or a View configuration.
- is a shortcut to save an Archiving configuration or a View configuration.
- is a shortcut to save quickly all Archiving configurations or View configurations.
- is a shortcut to print Archiving configuration or View configuration.
- is a shortcut to reset Archiving configuration or View configuration.

Appendices

Lexicon

Archiving
The action of storing in database the values of Tango attributes in a time range, each value being associated with a time stamp.

Archiving mode
An archiving mode is a rule defining when an attribute has to be archived.

The most basic archiving mode is the Periodic archiving mode, where an attribute is archived every N milliseconds.

More elaborate archiving modes like the Threshold archiving mode ask for archiving when the attribute's value meets certain conditions, but they can only be used in association with the Periodic archiving mode.

AC
Short for an Archiving Configuration

Describes archiving for a set of attributes, by associating a set of archiving modes to each of its attributes.

VC
Short for a View Configuration.

Describes plot properties for a set of attributes, by associating a set of plot properties to each of its attributes.

HDB
Short for the Historic Database.

Attributes values archived in HDB are stored forever.

TDB
Short for the Temporary Database.

Attributes values archived in TDB are deleted every [given deletion time].

The TDB is regularly purged and the Interval of the purge is defined by the data base administrator and the system administrator.

Quick Save/Quick load
The action of saving (resp. loading) to a default file is Called "quick" because no user interaction to choose a directory/file name is required.

7.5.3 HDB++

Intended audience: developers, administrators

Contents

- *HDB++*
  - *Target*
  - *Primary Presentation*
  - *Elements*
    - *Backend support*

        * *Configuration Manager Device Server*
        * *Event Subscriber Device Server (Archiver)*
        * *HdbConfigurator Server*
        * *HdbViewer*
    - *Source code*
    - *Documentation for building and installing*
    - *Rationale*
    - *Presentations and papers*

Target

This document is directed to beginner developer.

Primary Presentation

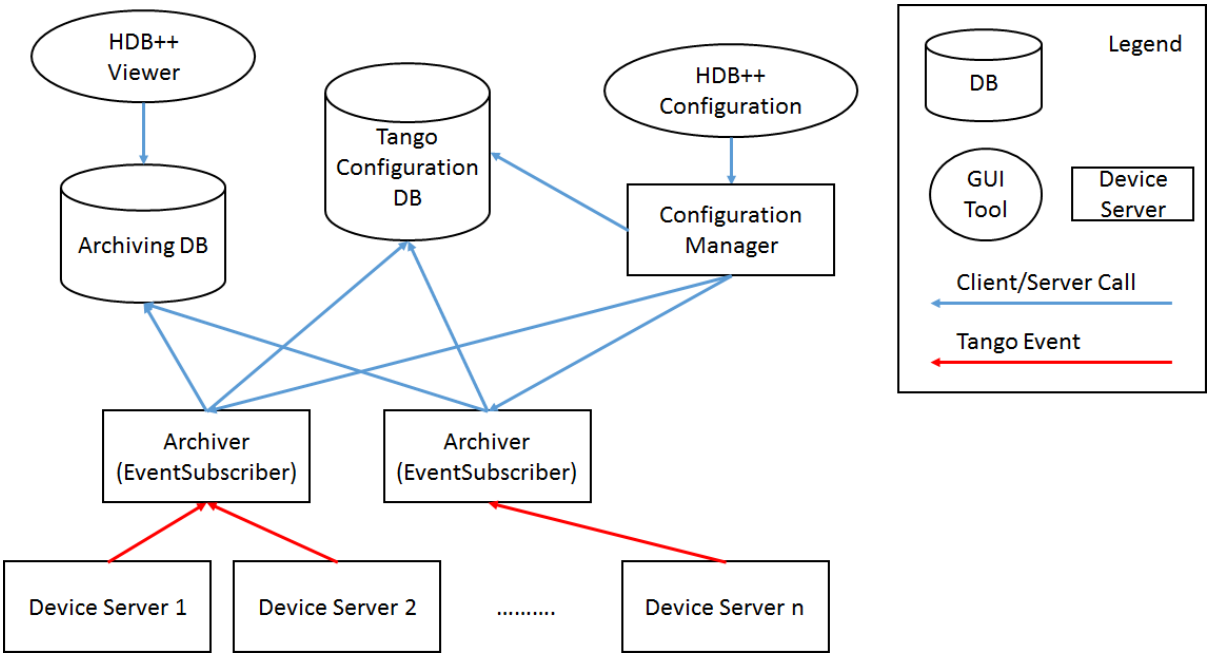


Figure 1: HDB++ Runtime View (part one)

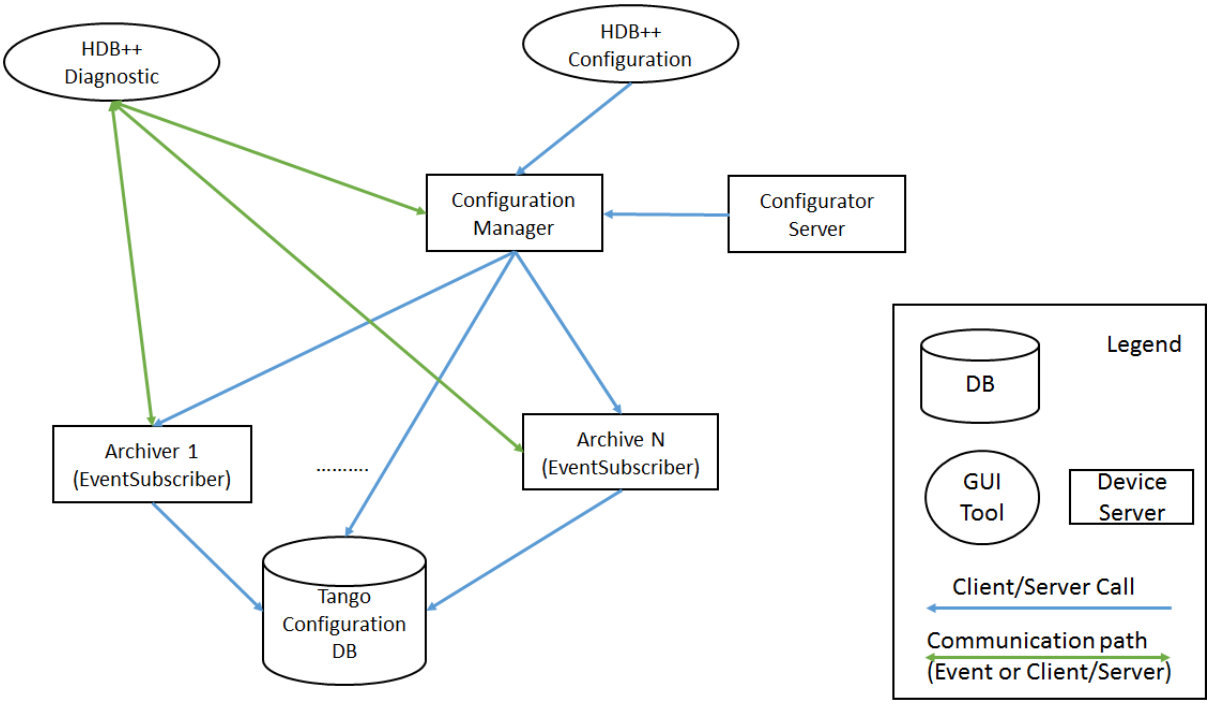


Figure 2: HDB++ Runtime View (part two)

Elements

Block	Description
HDB++ Viewer	Standalone JAVA application designed to monitor signals coming from HDB++. It has been written using Swing and needs a JVM higher than 1.7.0. *More information*.
HDB++ Configuration	Standalone JAVA application that allows interaction with the configuration manager in order to add, modify, move or delete an attribute from the archiving system
HDB++ Diagnostic	Standalone JAVA application that visualizes both the status of all the archiver device servers and the overall archiving system
Archiving DB	Specific Database devoted to storing attribute values. The currently supported backend are Mysql, Cassandra (support has been dropped), PostgreSQL, ElasticSearch or Timescaledb.
Tango Configuration DB	Tango database where every property and configuration of the Tango control framework is stored
Archiver	The EventSubscriber TANGO device server, or Archiver, is the archiving system engine. On typical usage, it will subscribe to archive events on request by the ConfigurationManager device. The EventSubscriber is designed to start archiving all the already configured Attributes, even if the ConfigurationManager is not running. Moreover, being a TANGO device, the EventSubscriber configuration can be managed with Jive. The list of Attributes to be gathered by each EventSubscriber is stored in the AttributeList Property of the EventSubscriber device.
Device Server i	Generic Device server that contains one or more devices that needs to archive one or more attributes
Configurator Server	Device server that assists in adding, modifying, moving, deleting multiple attributes in the archiving system using the Configuration Manager.
Configuration Manager	Device server that assists in adding, modifying, moving, deleting an Attribute to/from the archiving system

HDB++ inherits the database structure from the existing Tango Historical Data Base (*HDB*) and introduces new storage architecture possibilities, better internal diagnostic capabilities and an optimized API. Different backends to store the data can be implemented through an unified interface, currently Timescaledb, MySQL, Postgresql, ElasticSearch and Apache Cassandra (support for Cassandra has been dropped) are supported.

The HDB++ archiving system must fully comply to the Tango device server model, with two immediate benefits. First, all the required configuration parameters are stored to and retrieved from the Tango database; some of these parameters are, for user convenience, duplicated into a dedicated table of the HDB++ schema by a mechanism that guarantees the

consistency of the copy. Second, the HDB++ archiving system inherits the Tango scaling capability: any number of EventSubscriber instances can be deployed according to the desired architecture and overall performance.

The HDB++ architecture is fully event based; therefore, a part of HDB++ setup consists of conveniently configure Tango device servers to send events as required.

Backend support

A C++ interface *libhdb++* offers an interface for any backend to implement for data storage. These libraries, written in C++, are addressed to the EventSubscriber Tango device server and their main purpose is to provide *an abstraction layer*. Actually, some shared objects are available implementing the abstraction layer and the specific interface:

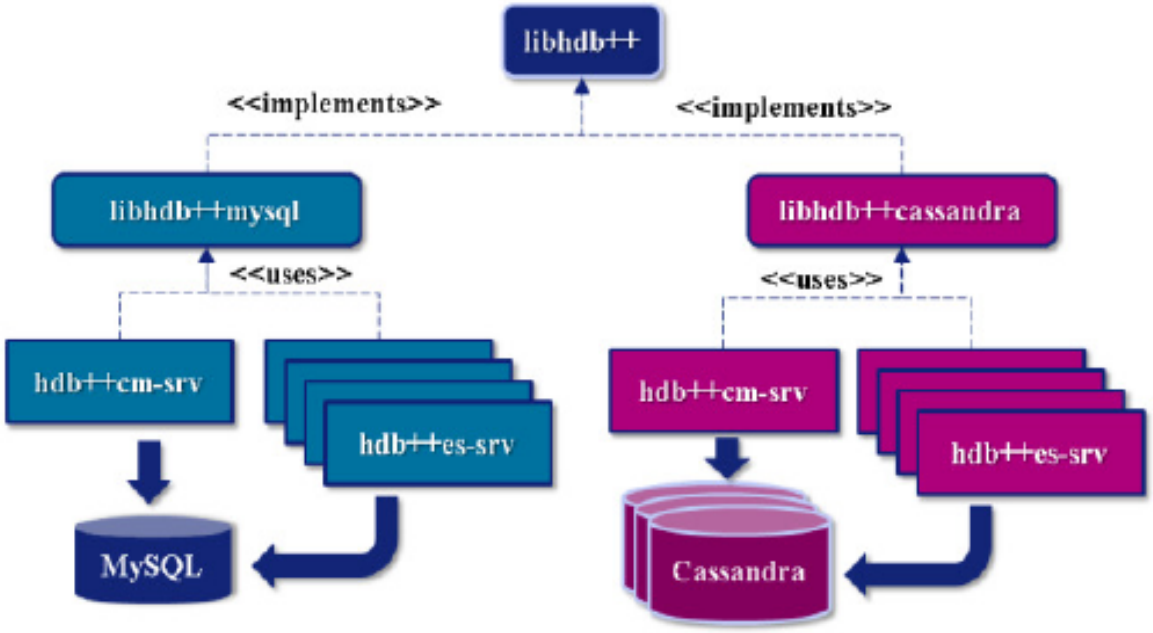


Figure 4: HDB++ Device Servers design

- **libhdb++**: database abstraction layer, decouples the interface to the database back-end from the implementation.
- **libhdbmysql**: legacy HDB schema support for MySQL back-end
- **libhdb++mysql**: HDB++ schema support for MySQL back-end
- **libhdb++cassandra**: Cassandra back-end implementation of libhdb++
- **libhdb++timescale**: Timescaledb back-end implementation of libhdb++
- **libhdb++postgres**: Postgresql back-end implementation of libhdb++
- **hdb++ELK**: ElasticSearch back-end implementation of libhdb++

These libraries allow reusing the EventSubscriber, the ConfigurationManager and the GUIs without changes.

Configuration Manager Device Server

It configures the attributes to be archived and defines which Event Subscriber is responsible for a set of Tango attributes to be archived. It provides diagnostics data as well. Configuration Manager Tango device server will assist in the operations of adding, editing, moving and deleting an Attribute to/from the HDB++ archiving system. A specific library, exposing a suitable API, addresses the historical data extraction from the archive. The ConfigurationManager device server is able to perform the following operations on the managed EventSubscriber pool:

- handle the request of archiving a new Attribute;
- setup the Attribute archive event configuration;
- assign the new Attribute to one of the archivers;
- move an Attribute from one archiver to another;
- show the Attribute/archiver coupling;
- start/stop the archiving of an Attribute;
- remove an attribute from archiving.

The Configuration Manager also exposes some Attributes to keep trace of the global statistics:

- total number of EventSubscribers;
- total number of working/faulty attributes;
- total number of events per second;
- overall minimum and maximum processing and storing time.

These attributes could be themselves archived to enable a follow up versus time.

Event Subscriber Device Server (Archiver)

Also referenced as archiver, is in charge of gathering the values from the Tango devices and storing them into the historical database. To address the requirements coming from large systems the need to distribute the workload over a number of archivers shows up. The EventSubscriber Tango device server is the core of the HDB++ archiving system. It subscribes to archive events for the specified Attributes list, stored into a Property in the Tango database, as well as a number of additional parameters, such as the hostname and port number where the back-end is running, the name of the database and the username and password to be used. It subscribes to Tango archive events, which are ZeroMQ events in the latest Tango releases, and stores the received events in the historical database. It provides diagnostics data as well.

The EventSubscriber device server allows to perform the following operations:

- add/remove an Attribute to/from archiving;
- start/stop the archiving for all Attributes;
- start/stop the archiving for one Attribute;
- read the status of an Attribute;
- read the list of Attributes currently archived (started);
- read the list of Attributes currently not archived (stopped);
- read the number/list of Attributes in charge;
- read the configuration parameters of each Attribute;
- read the number/list of working Attributes;

- read the number/list of faulty Attributes;
- read the number/list of Attributes pending in the FIFO.

Working at the EventSubscriber level implies that the database entry and the archive event parameters have to be already configured. Besides, no action is performed on the archived data when removing an Attribute, which means that the data remain available in the historical database. The EventSubscriber Tango device server also exposes some additional figures of merit, such as:

- for each instance, total number of records per time;
- for each instance, total number of failures per time;
- for each Attribute, number of records per time;
- for each Attribute, number of failures per time;
- for each Attribute, time stamp of last record.

These numbers can sum up in a counter, which can be reset every hours/days/weeks, to rank each Attribute in term of data rate, error rate etc. This allows preventive maintenance and fine tuning, detecting, for instance, when an Attribute configuration is wrong because the variation threshold is lower than the noise level. These statistics are a key element for qualifying the health of the system. All these Attributes are archived themselves to enable a follow-up versus time. For each Attribute, the EventSubscriber Tango device server also computes the minimum and maximum processing and storing times, which helps to discover possible bottlenecks. There may be several EventSubscriber device servers.

HdbConfigurator Server

The HdbConfigurator device server is a Java device server. It was developed just to simplify the task of configurating dynamically several new attributes to be added in HDB++.

At the ESRF, it is used to configure/add automatically some dynamic attributes into HDB++. There are some cases where the same device will create some dynamic attributes at startup depending on its configuration (device properties), if the configuration changes, different attributes might be created. In this specific case, we have the requirement to ensure these dynamic attributes are archived. So the device server will configure them via the HdbConfigurator device server.

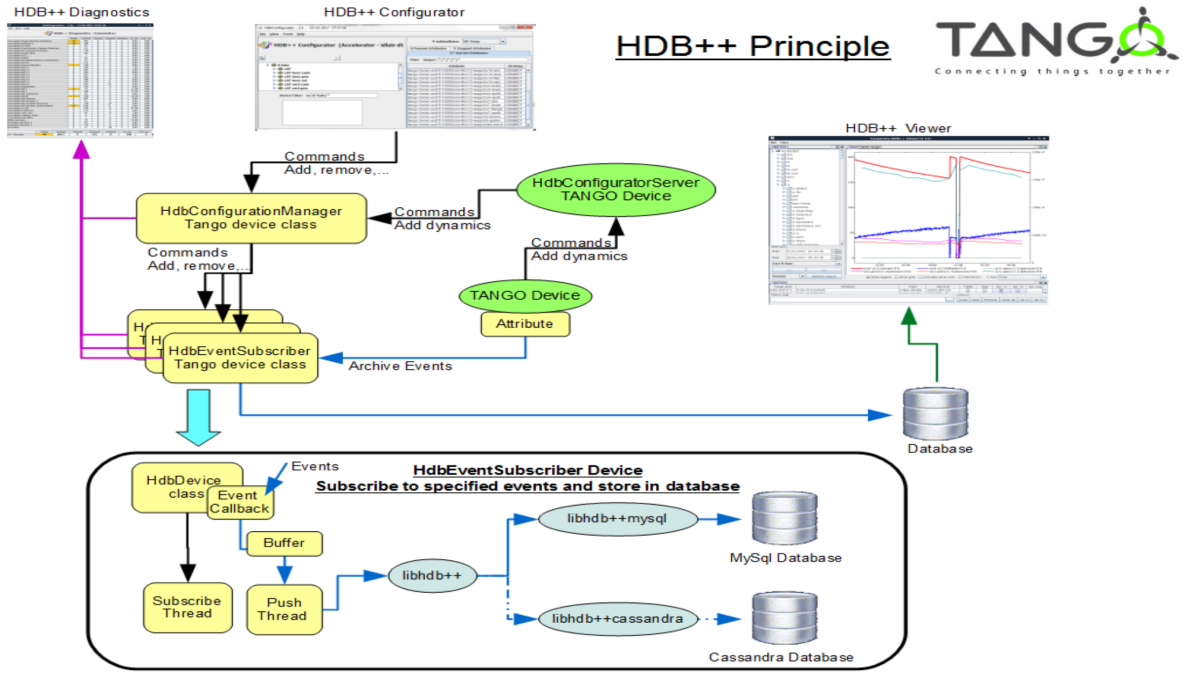


Figure 3: HDB++ and HdbConfiguratorServer

To be more concrete, for the people working in a synchrotron, we have a device, controlling the insertion devices used by a given beamline. Dynamic attributes are created for each undulator which can be controlled by this beamline. During a shutdown period, the undulator might have been replaced, so there will be new attributes created for the new undulators and the attributes related to the undulators which have been removed will no longer exist.

The HdbConfigurator server will handle a queue and coordinate the requests sent to the HdbConfiguration Manager device. It will help to configure several attributes in parallel. If one would like to do it with only the HdbConfiguration Manager device, one would need to lock the device, set many attributes, send the addAttribute command and finally release the lock of the device. The HdbConfigurator Server is doing that for you in the same way as the HDB++ Configuration Manager GUI is doing it.

The HdbConfigurator Server is indeed using the same classes as the HDB++ Configuration GUI but its source code is actually currently available on tango-ds Sourceforge. It is listed in the Tango classes catalog: http://www.tango-controls.org/developers/dsc/ds/274/

HdbViewer

It visualizes the data stored in the historical database.

Two libraries have been developed to the *historical data extraction*:

- A java implementation, *libhdbpp-extraction-java*, has been used for the *HdbViewer* GUI and is a native choice for Java device servers. The HdbViewer Java framework, supports both legacy HDB archiving system, and the new HDB++ design.
- A C++ implementation, dedicated *Qt/Qtango* based GUIs or to C++ Tango device servers. The *HdbExtractor++ multithread library* allows fetching the data from the legacy HDB and the new HDB++ MySQL schema in a simple Object Oriented way. An additional module provides a Qt interface to the HdbExtractor++ and a dedicated GUI, exploiting the MathGL framework, aimed at displaying mono and bidimensional data over time.

Note: The C++ extraction library currently supports only the MySQL back-end.

Source code

The source code is available on Gitlab in the following repositories:

- hdbpp-timescale-project: A project to centralize and build all the needed components of a full HDB++ setup with the Timescaledb backend.
- hdbpp-mysql-project: A project to centralize and build all the needed components of a full HDB++ setup with the Mysql backend.
- hdbpp-cm : the HDB++ Configuration Manager device server
- hdbpp-es: the HDB++ Event Subscriber device server
- hdbpp-cm-es: Device server able to export HDB++ Event Subscriber and Configuration Manager devices in the same device server. It is a multiclass Tango device server capable of both CongigurationManager and EventSubscriber devices. Can be used in place of hdbpp-cm and hdbpp-es devices.
- libhdbpp: the HDB++ insertion (abstract) library
- libhdbpp-mysql: the HDB++ insertion library for MySQL backend
- libhdbpp-mysql-legacy: the HDB++ insertion library for MySQL backend using the old Tango HDB database schema
- libhdbpp-cassandra: the HDB++ insertion library for Cassandra backend
- libhdbpp-timescale: the HDB++ insertion library for Timescaledb backend
- libhdbpp-postgresql: the HDB++ insertion library for Postgrsql backend
- libhdbpp-elk: the HDB++ insertion library for ElasticSearch backend
- hdbpp-benchmark: A project to compare performances of different HDB++ backends using docker images
- CassandraMonitor: A Java client/server to monitor cassandra nodes using jmx calls.
- hdbpp-configurator: the HDB++ Configuration GUI (in Java).
- hdbpp-viewer: the HDB++ Viewer GUI (in Java)
- libhdbpp-extraction-java: HDB++ Java extraction library
- libhdbpp-extraction-cpp: HDB++ C++ extraction library
- libhdbpp-python: HDB++ python extraction library
- eGiga2m: Web graphic data viewer able to show HDB++ data

Note: Please, find README file in each repository. It contains a lot of necessary information.

Documentation for building and installing

The documentation for building and installing many of the components is available on the different git repositories from https://gitlab.com/tango-controls/hdbpp Gitlab organization.

Please, have a look at the README files from the different git repositories (For instance: https://gitlab.com/tango-controls/hdbpp/hdbpp-timescale-project). They explain how to install the HDB++ libraries as well as the device servers.

Note: -v5 can be used to see the DEBUG messages coming from the Tango library itself too. It also sets the logging level of the device to DEBUG. -v4 can be used to see the DEBUG messages coming from the Tango devices themselves. By doing this, DEBUG level logs will be printed on your terminal. For example, command on terminal: ./hdb++es-srv 01 -v5

Note: In recent versions of *libhdb++cassandra* library, there are some configuration parameters to adjust libhdb++cassandra library log level and the cassandra driver log level. These configuration parameters are parameters of LibConfiguration Tango class or device property. These parameters are named differently depending on the version of the library used (logging_enabled or logging_level and cassandra_driver_log_level)

Rationale

The HBD++ archiving system is built on top of the Tango Event model which provides a specific event for archiving, this is the **archive event**. The archive events are configured with three attributes properties:

- **archive_abs_change:**
  a Property of up to 2 values, positive and negative delta, that specifies the absolute change with respect to the previous Attribute value, which triggered the event. If only one value is specified it is used for both positive and negative change. If no thresholds are specified then the relative change is used.
- **archive_rel_change:**
  a Property of up to 2 values, positive and negative delta, that specifies the relative change with respect to the previous Attribute value, which triggered the event. If only one value is specified it is used for both positive and negative change. If no thresholds are specified archive events are not sent on value change.
- **archive_period:**
  the time between which periodic archive events are sent, in milliseconds. If no period is specified no periodic archive events are sent.

Usually it is composed of several TANGO device servers (Archiver aka EventSubscriber), but there must be at least one device server. Each EventSubscriber device is in charge of archiving a number of attributes from a number of devices. The number of EventSubscriber TANGO devices to deploy and the number of TANGO devices/Attributes in charge of each subscriber is not bounded and depends on the desired performance.

The ConfigurationManager device server manages a pool of EventSubscribers; the list is stored in the ArchiverList property of each ConfigurationManager device, and is updated via the ArchiverAdd, ArchiverRemove and AttributeSetArchiver commands. The list is stored in the ArchiverList device Property of the ConfigurationManager device using the FQDN syntax. This tells the ConfigurationManager everything which is needed to connect to the managed EventSubscribers: protocol, host, port and device name.

Presentations and papers

- HDB++ Meeting minute 2016-05-10
- HDB++: a new archiving system for TANGO (Proceedings of ICALEPCS2015)
- How Cassandra improves performances and availability of HDB++ Tango Archiving System (Proceedings of ICALEPCS2015)
- HDB++: High availability with Cassandra - presentation from the 29th TANGO collaboration meeting
- HDB++ Cassandra presentation at the ICALEPCS 2015 Tango workshop

More information is available in Administration -> Services section of this documentation under *HDB++ - an archiving/historian service*.

7.5.4 SNAP

Intended audience: administrators, users, developers

The SNAP is set of device servers and an GUI application (**Bensikin**) providing so called SNAPshot functionality.

A snapshot is, as said in the name, a "picture" of a list of equipment's "settings" (*more precisely of their Tango attributes values*) taken at a precise instant.

Snapshot are stored in a database (**MySQL** or **Oracle**). These can be retrieved and restored to the equipment (devices). This kind of functionality is often called a *recipe management*. It allows to create set of configuration settings (*recipes*) used for particular purposes (like selected mode of accelerator operation).

SNAPshot (Archiving) installation and configuration

Intended audience: administrators, users, developers

Overview

The *SNAPshot* system is distributed as a part of the *Archiving* solution.

The server side consists of a database (either MySQL or Oracle) and a set of device servers. On the client side, the application *Bensikin* is used.

To use SNAPshot system, at lease one device must be created for each class:

- *SnapManager*, responsible for creating snapshot contexts and applying snapshots to the devices,
- *SnapArchiver*, responsible for creating snapshots and storing them in the database,
- *SnapExtractor*, responsible for extracting snapshots from the database.

If more that one device of a particular class is defined, Archiving API will randomly assign a device for each request.

Prerequisites

Following components are required to set-up and use Archiving:

- Tango Controls,
- SQL database (this guide assumes MariaDB 10),
- Java Runtime Environment (this guide assumes OpenJDK 8).

Note: This guide assumes that the installation is performed on the Ubuntu system but it should be applicable also to the other GNU/Linux-based systems.

To install SNAPshot on Windows, minor changes need to be applied, e.g. the applications from the *win32* directory should be used instead of the applications from the *linux* directory.

Installation

Installation steps:

1. download and extract the *ArchivingRoot* package,
2. add the executable flag to the scripts located in `bin/linux`,
3. add the executable flat to the scripts located in `device/linux`,
4. replace the `sh` interpreter with `bash` in shebang in scripts located in `bin/linux` and in `device/linux`. The scripts are not POSIX compliant,
5. run SQL script `db/create-SNAPDB-InnoDB.sql` to create the database and users,
6. set `ARCHIVING_ROOT` environment variable to point to the directory with the contents of the *ArchivingRoot* package.

Configuration

Configuration steps:

1. **define following devices:**

Server/instance	Class	Device
SnapManager/1	SnapManager	archiving/snap/manager.1
SnapExtractor/1	SnapExtractor	archiving/snap/extractor.1
SnapArchiver/1	SnapArchiver	archiving/snap/archiver.1

2. **configure SnapManager class properties:**
   - *DbUser*: snapmanager,
   - *DbPassword*: snapmanager,
3. **configure SnapExtractor device properties:**
   - *DbUser*: snapbrowser,
   - *DbPassword*: snapbrowser,
4. **configure SnapArchiver device properties:**

- *DbUser*: snaparchiver,
- *DbPassword*: snaparchiver,
- *BeansFileName*: beansBeamline.xml.

Note: If *BeansFileName* is configured as a device property but the server is still looking for the default *beans.xml* during startup, the *init* command must be executed as a workaround.

```
----- Hibernate Resources -----
configure hibernate resource: beans.xml
Exception in thread "main" org.springframework.beans.factory.
↪BeanDefinitionStoreException: IOException parsing XML document from class path␣
↪resource [beans.xml]; nested exception is java.io.FileNotFoundException: class path␣
↪resource [beans.xml] cannot be opened because it does not exist
...
```

Alternatively, *BeansFileName* can be configured as a class property.

Usage

To use SNAPshot system, all device servers must be started.

The application `bin/linux/bensikin-rw` can be used to configure and trigger snapshots. See *Bensikin manual* for more details.

Note: The machine where *Bensikin* is running must have access to the snapshot database. *Bensikin* will use credentials specified in *SnapManager* class configuration.

Properties reference

Following tables summarize configuration properties. The *Class* column indicates that this is the class property. Otherwise it is a device property.

Note: The database connection properties are defined in device classes:

- *SnapManager* for Snap

Every device and application that has to connect to a database will read the properties of these classes. However, every device can have its database connection properties redefined in the device properties.

SnapManager properties

Name	Description	Default	Mand.	Class
Description	Device description			✓
ProjectTitle	Project description			✓
DbUser	User name used to connect to the database	archiver	✓	
DbPassword	Password used to connect ot the database	archiver	✓	
DbHost	Database Host name	localhost	✓	✓
DbName	Database name		✓	✓
DbSchema	Schema name		✓	✓
isRac	Oracle database is in Rac Mode	false	✓	✓

SnapArchiver properties

Name	Description	Default	Mand.	Class
Description	Device description			✓
ProjectTitle	Project description			✓
DbUser	User name used to connect to the database	archiver		
DbPassword	Password used to connect ot the database	archiver		
DbHost	Database Host name	localhost		
DbName	Database name			
DbSchema	Schema name			
beansFileName	Name of the beans file (on CLASSPATH)	beans.xml		

Note: This device will check *SnapManager* class properties to discover how to connect to the database.

SnapExtractor properties

Name	Description	Default	Mand.	Class
Description	Device description			✓
ProjectTitle	Project description			✓
DbUser	User name used to connect to the database	snap	✓	
DbPassword	Password used to connect ot the database	snap	✓	

Note: This device will check *SnapManager* class properties to discover how to connect to the database.

Bensikin User Manual

Intended audience: users, administrators

Authors:

R.Girardot, M.Ounsy, A.Buteau, M.Thiam - Soleil

Introduction

This document is an end-user guide to using the **Bensikin** application, and a brief developer-oriented presentation of the application's architecture.

Application's context: Contexts and Snapshots

A snapshot is, as said in the name, a "picture" of a list of equipment's "settings" (*more precisely of their Tango attributes values*) taken at a precise instant.

A snapshot is based on a context, which is a group of Tango devices and their attributes that must be part of the snapshots. A context is described by meta-data (author, description, etc.), so that users know which context is used what for.

A typical use case of a snapshot is to memorize a particular configuration of a set of equipment's, to be able in the future to reset them to the values of this snapshot (*for example, reposition all Insertion devices to their parking position after a beam loss*).

Application's description and goals

Application's goals

Bensikin allows the user to define contexts and take snapshots. Snapshots can be saved as files and modified.

Bensikin is ready for multi-user use, which has for consequence the need to define accounts. An account is a way to map a user with a working directory. An important consequence of this is that **2 different users must not use the same working directory**, or you may encounter application misbehaviors. An account has a name and a path to a working directory.

Bensikin is thus naturally divided (both in functionalities and display) in three parts:

- **The account part, which is an introduction to the rest of the** application
- The context part
- The snapshot part

A first look to Bensikin

The Bensikin Account Manager is here to manage accounts, which means:

- Creating a new account
- Deleting an existing account
- Launching application with an account chosen in a list

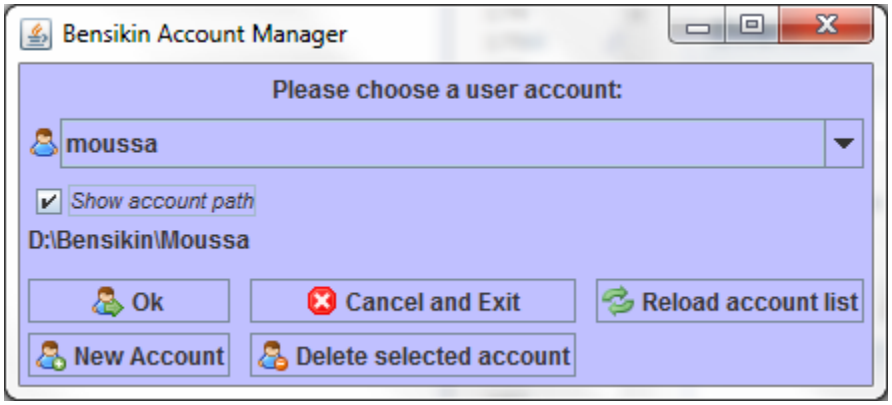


Fig. 25: Figure 1: Bensikin Account Manager

The *Context Control Panel* is where user can manage contexts, which means creating, loading and modifying contexts, and launching snapshots based on the defined contexts.

The *Snapshot Control* Panel is where user can manage snapshots, which means saving snapshots in files, loading snapshots from database and files, temporary modify snapshots attributes values and set equipment with defined snapshots (with or without modifying snapshots) or a subpart of them.

The application's *logs panel* displays the application information and error messages (like database interaction, encountered problems, etc.)

The *Menu* and the *Tool bar* are for actions shortcuts and application's options.

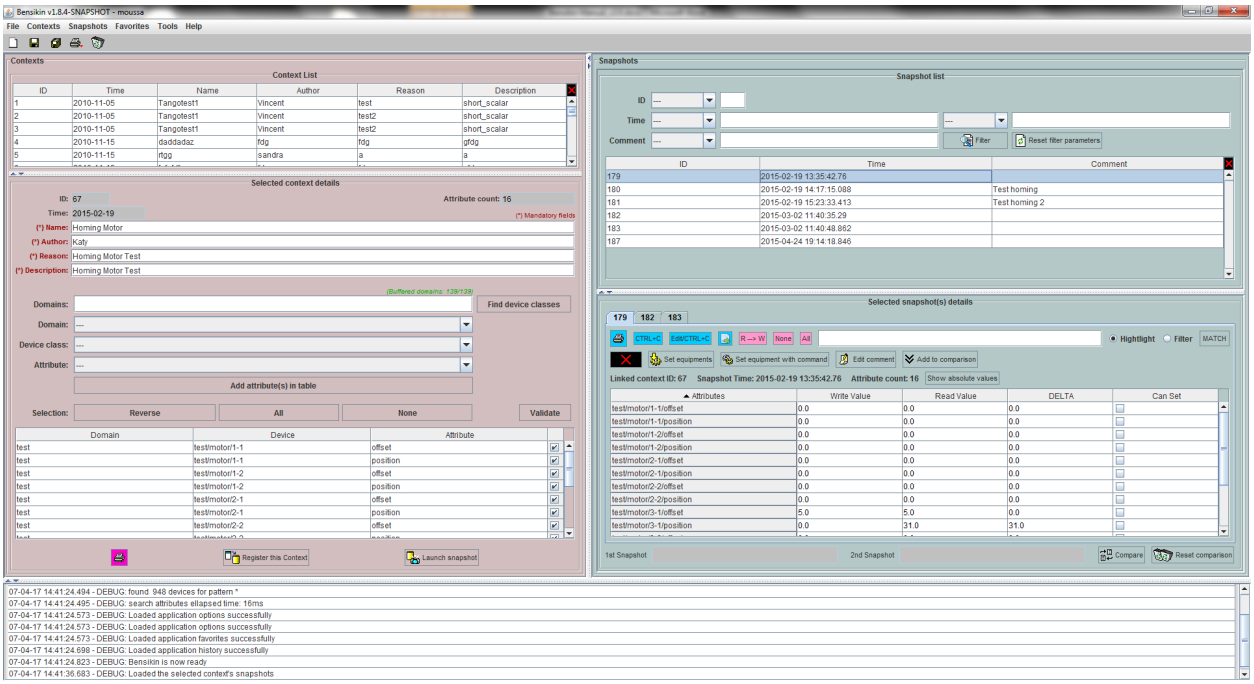


Fig. 26: Figure 2: Bensikin main panel

Account Manager

The *Figure 1: Bensikin Account Manager* presents the account Manager Interface, on application start. With this manager, you can create a new account, or delete or use an existing one.

To quit the application, simply click on

or button.

Existing accounts are listed in the account *Selection Combo Box*, which you can reload by clicking on

button (if you think that someone could have modified it by creating a new account or deleting an existing one, for example).

Creating a new account

To create a new account, click on the button

(at the bottom left of the panel). A new dialog will appear, as following.

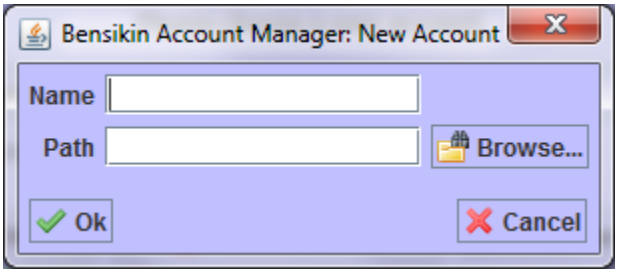


Fig. 27: Figure 3: Creating a new account

In this new dialog, you will have to enter the name of your new account and the path of the application working directory for this account. If you prefer, you can browse for the path by clicking on the Browse... button. Then, a classic browsing dialog will be displayed, in which you can choose the directory. When both fields (*Name* and *Path*) are fulfilled, click on Ok button to validate your new account, which will be automatically added in the list of existing accounts. If you click on Cancel or button, you go back to the first dialog, as presented in *Figure 1: Bensikin Account Manager*, and nothing is done.

Deleting an existing Account

To delete an existing account, first select the account in the account selection combo box, as following:

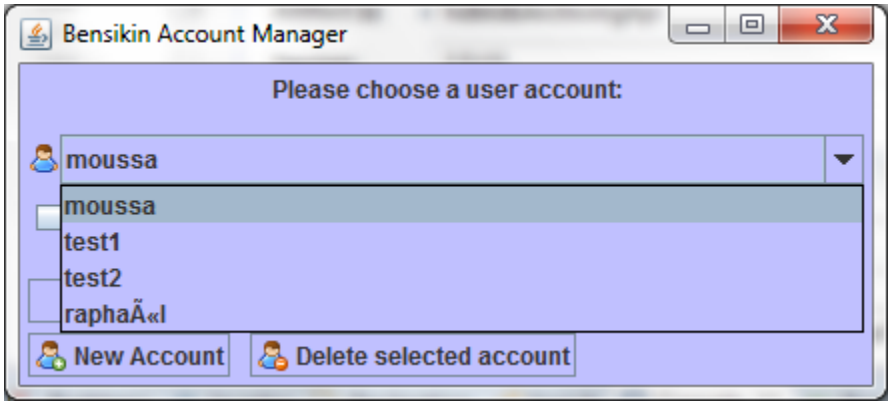


Fig. 28: Figure 4: Account selection

When the account is selected, click on *Delete* button to delete it. If you do it, you won't be able to use this account any more (and no other user either), because the account is definitely removed from list. The account deletion doesn't involve the corresponding directory (neither its content) deletion.

If you want to see your account path, you can check *Show account path*.

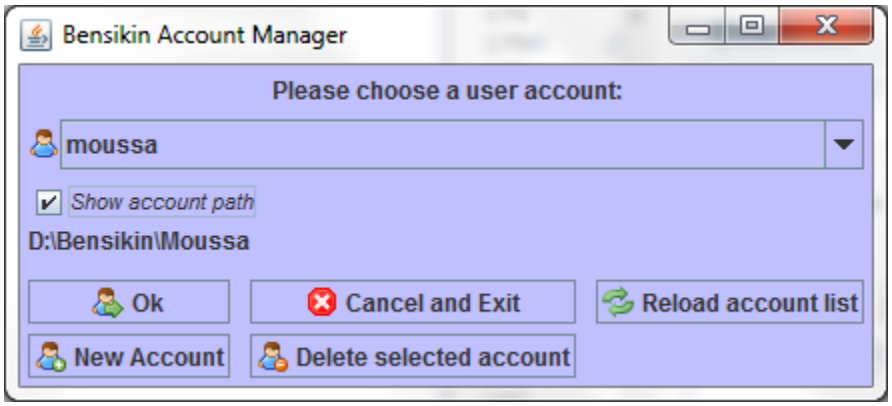


Fig. 29: Figure 5: Show account path

Launching application with an existing account

To launch application with an existing account, first select the account in the account selection combo box, as presented in *Figure 4: Account selection*.

Then, click on *Ok* button, and you will reach the application main panel configured with this account (the account name is displayed in frame title).

Contexts Management

This section describes how to control contexts with Bensikin. A context is a list of attributes for which you can make a snapshot. A context has an ID and a creation date, both defined by the database. A context also has a name, an author, a reason and a description. The reason usually describes why the context was created (example: because of an incident or in order to set some equipment), whereas the description is here to have an idea of what kind of attributes you will find in this context.

Contexts are managed in the context control panel:

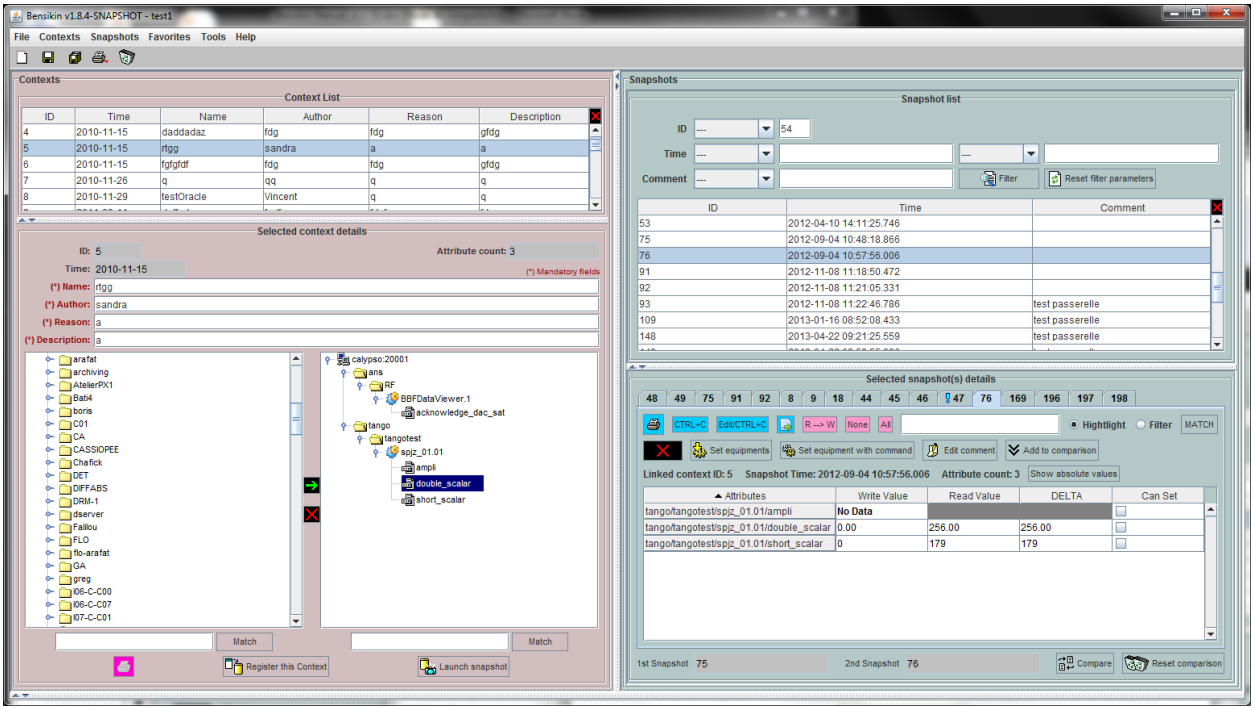
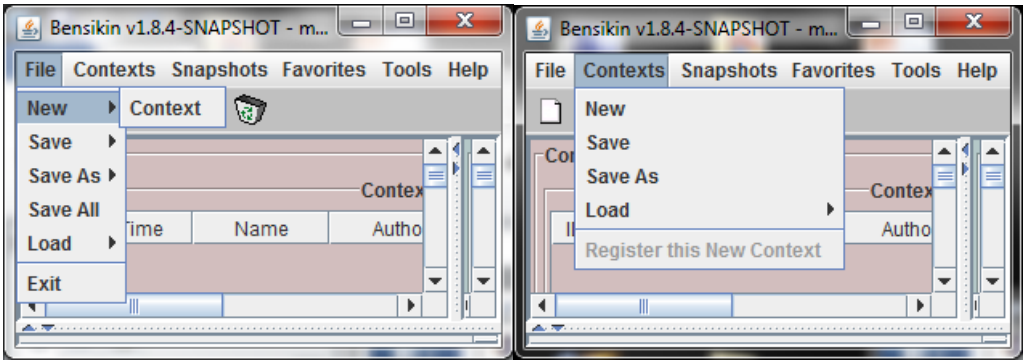


Fig. 30: Figure 6: Context control panel

Creating a new context

To create a new context, click on the *new* icon in toolbar ( ), or choose option to make a new context from *File* menu or *Contexts* menu:



You also are ready to make a new context at application first start or by clicking on the *reset* icon ( ):

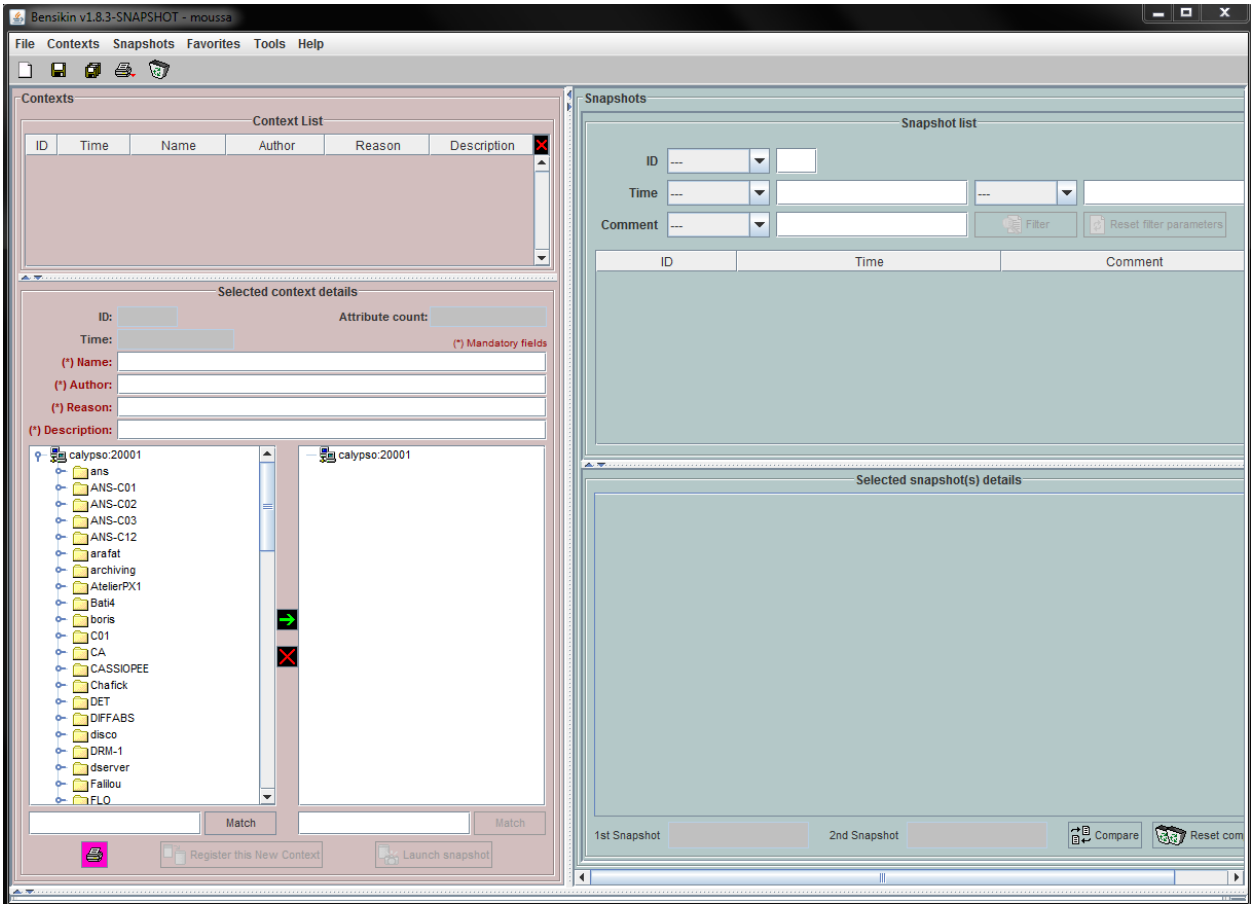


Fig. 31: Figure 7: Application first start

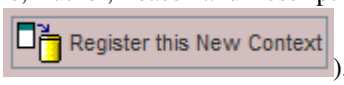
The difference between the *reset* icon( ) and the *new* icon( ) is, that the "reset" icon will clear every panel, whereas the "new" icon will only clear the snapshot list and the Context Details sub panel.

Classic way (tree)

The tree on the left side of the *Context Details* sub panel allows you to check for available attributes. The one on the right side represents your context attributes.

To add attributes in your context browse the left tree, select attributes (represented by the icon), and click on the arrow to transfer them to the right tree.

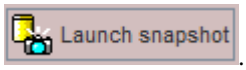
To remove attributes from your context, select them in the right tree and click on the cross .

Finally, fill the context Meta data (Name, Author, Reason and Description) in the corresponding fields (Note that filling the fields activates the *register* button 

).

Then, you can save your context in database by clicking on the *register* button

.

Doing so will deactivate the *register* button and activate the *launch snapshot* button 

.

You can save your context in a file using the *save* icon .

Alternate way (table)

To select this alternate way, go to *tools* menu and select *options*.

Then select the *context* tab and click on the *table* radio button.

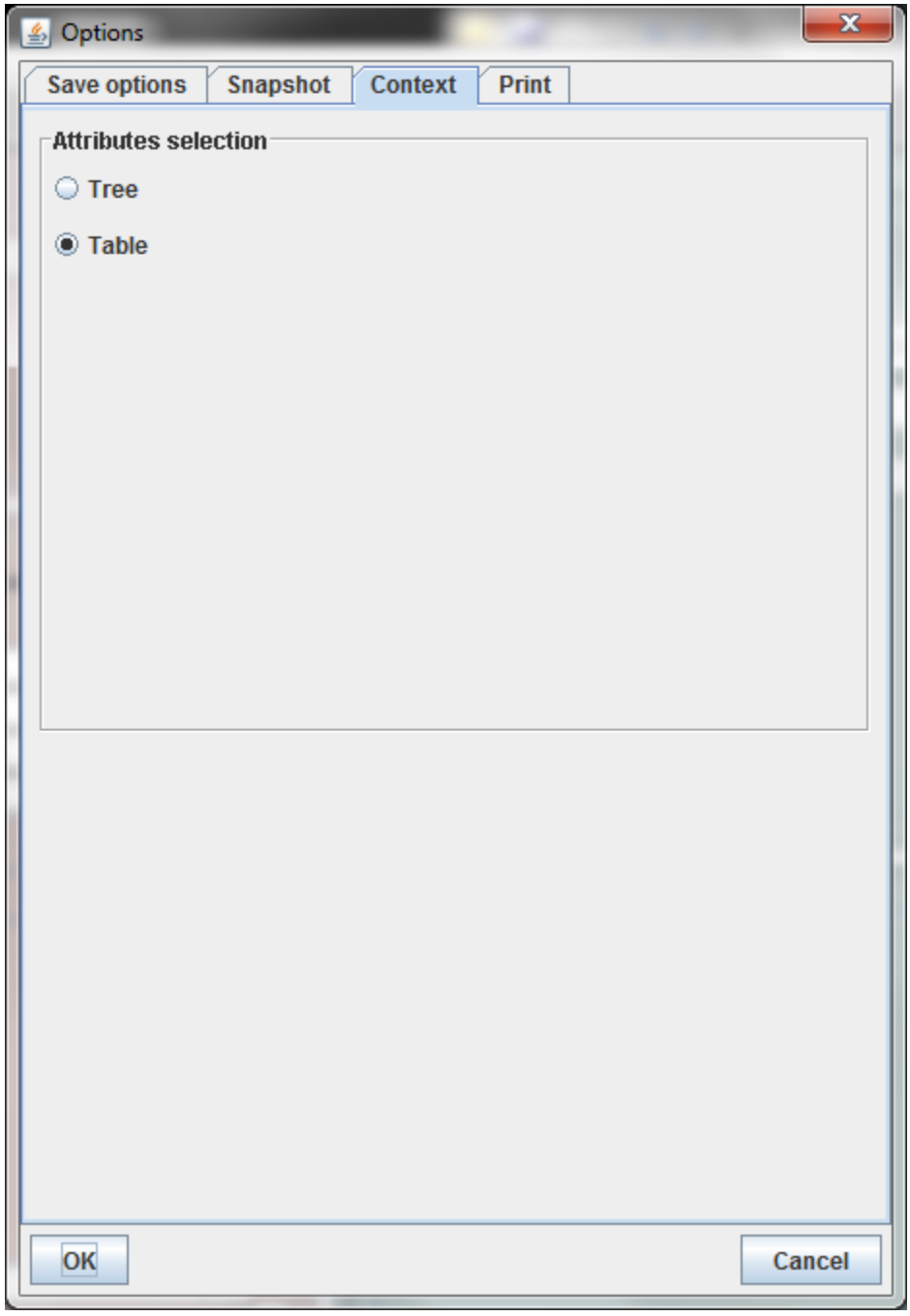


Fig. 32: Figure 8: Option –context tab

Click on the *ok* button. The context panel now has the “table selection mode”.

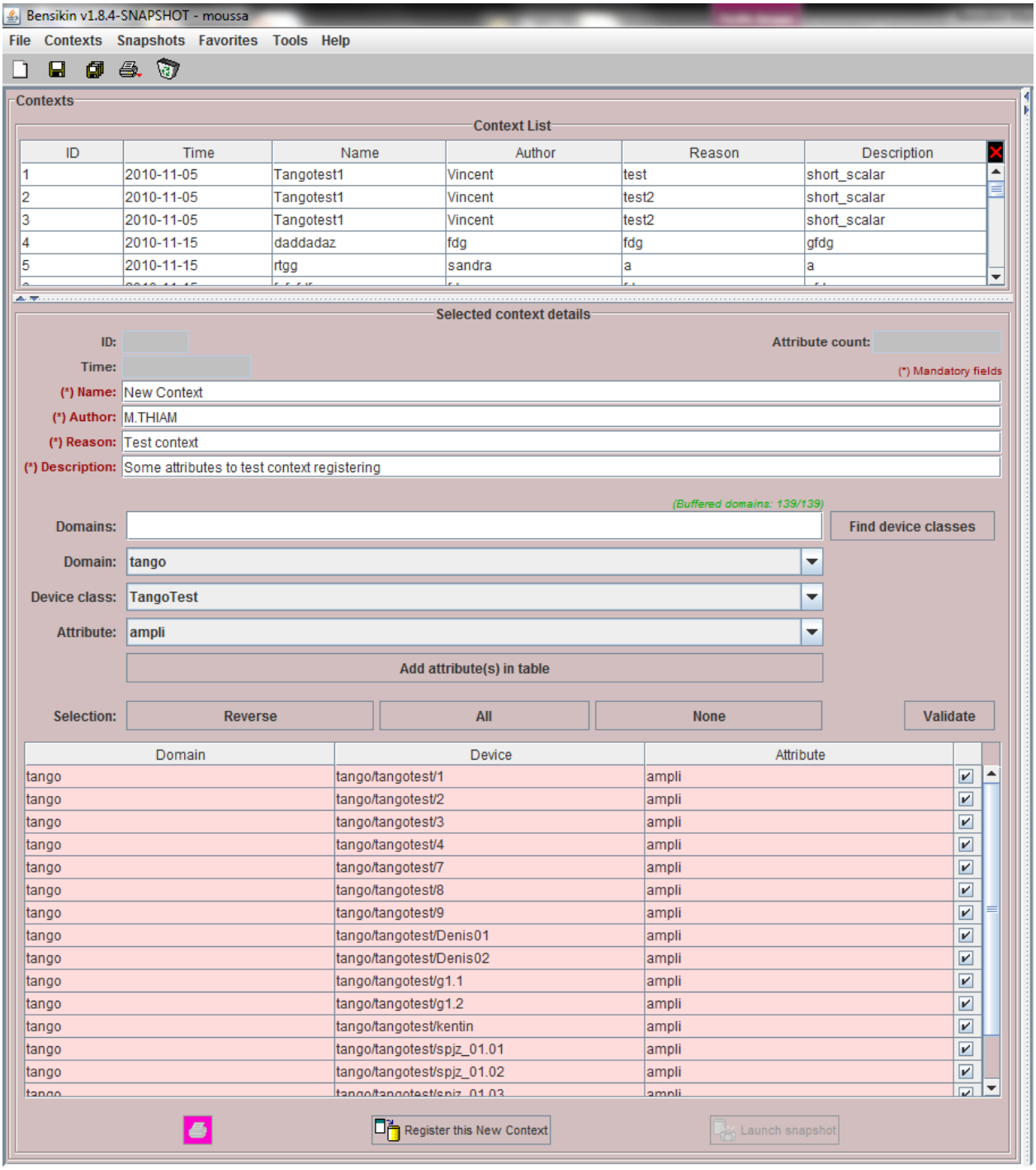


Fig. 33: Figure 9: Bensikin with context table selection mode (new context)

- Attribute selection and automatic attributes adding:
  - Choose a Domain. This refreshes the list of possible Device classes for this Domain.
  - Choose a Device class. This refreshes the list of possible Attributes for this Domain and Device class.
  - Choose an Attribute and press *OK*:

    All Attributes with the selected name **AND** belonging to any Device of the selected Class and Domain are

added to the current Context's list of attributes.

All new attributes are light red until the Context is registered.

- Line level sub-selection of loaded attributes:

  Each attributes are initially checked, but this check can be removed by the user. When the user clicks on *validate*, all unchecked attributes will be removed from the current Context.

  - Click *All* to select all lines
  - Click *None* to select no lines
  - Highlight lines in the list (CTRL and SHIFT are usable), then click *Reverse highlighted* to reverse the checked/unchecked status of all highlighted lines.

As for the classic way, you will have to fill the Meta data fields and register your context in database by clicking on the *register* button

Modifying an existing context

As a matter of fact, you can't really "modify" a context. What you can do is to create a new context with its information (attributes and Meta data) based on another one.

The very difference is in alternate mode, where former attributes are in white and new ones in light red:

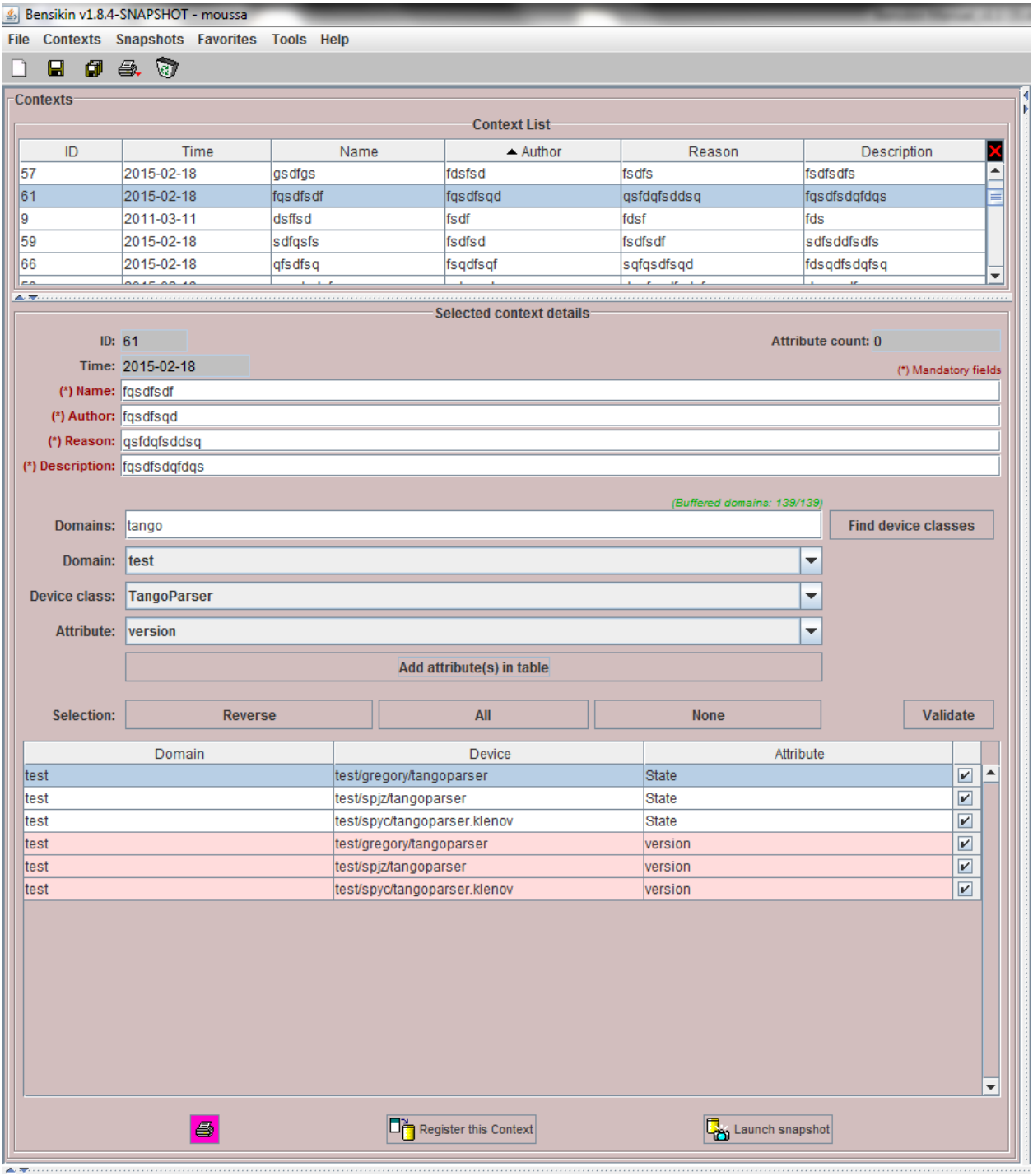


Fig. 34: Figure 10: Bensikin with context table selection mode (modified context)

The “register” button changed a little too: its text is *Register this context* instead of “Register this new context”, as you can see on the figure above.

Loading a context

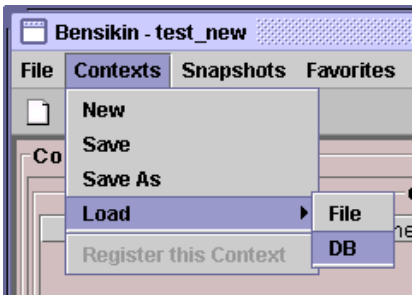
There are 2 ways to load a context:

- Load it from the database
- Load it from a file

In both cases, loading a context will apply a quick filter on the snapshot list, so you can see the snapshots about this context that have been created this day (the day when you load the context).

Loading a context from database

In the *Contexts* menu, choose *load* then select *DB*:



A dialog will then appear to allow you to filter the list of contexts in database following different criteria:

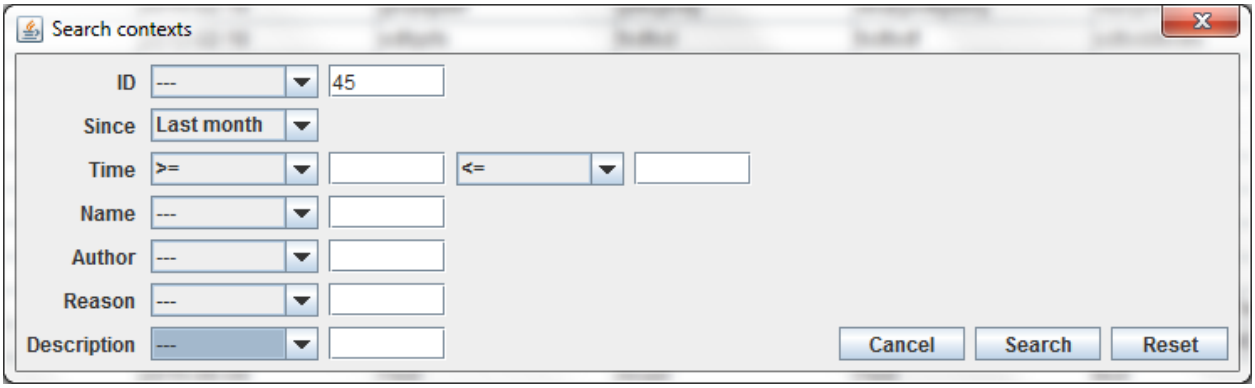


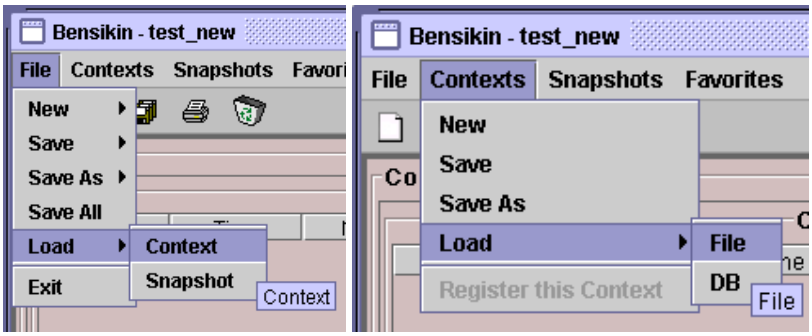
Fig. 35: Figure 11: Data base Context filter dialog

Select no criterion to search for all contexts present in database. Click on the Search button to apply the filter. The list of corresponding contexts will then appear in the Context List sub panel, as shown in *Figure 6: Context control panel*. Double click on a context in table to load it and see its details in the Context Details sub panel (See *Figure 6: Context control panel*).

If there are too many contexts in the list, you can remove some contexts from list (not from database) by selecting them in list and clicking on the cross on the top right corner of the list ( ).

Loading a context from file

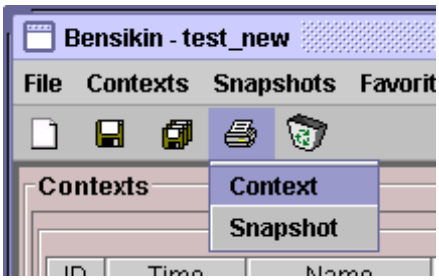
In the *Contexts* menu, choose *load* then select *File*, or in *File* menu choose *load* then select *Context*:



A classic file browser will appear. Search for your ".ctx" file and select it to load the corresponding context in the *Context Details* sub panel (See *Figure 6: Context control panel*).

Printing a context

Once you have context ready, click on the *print* icon ( ) and select *context*:

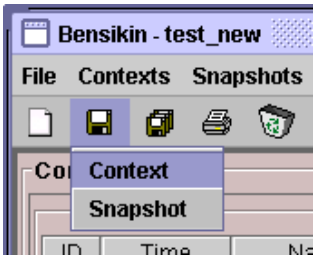


The classic print dialog will then appear. Validate your print configuration to print an xml representation of your context.

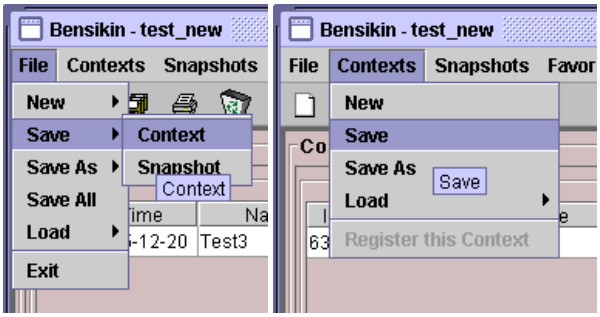
User can also print context by pressing the button

Saving a context

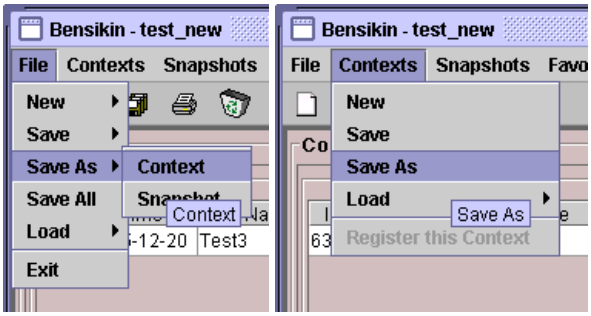
Once you have context ready, click on the *save* icon ( ) and select *context*:



You can also go to menu *Contexts* and click on *save*, or go to menu *File*, select *Save* and click on *Context*.



Then, the behavior is "Word-like". This means that if this is the first time you save this context, you will see the classic file browser to choose where to save your context, with file name. However, else, it will automatically save in the corresponding file. If you want to save in another file, you have to go to *File* menu, select *Save As* and click on *Context* or go to *Contexts* menu and click on *Save As*



Snapshot Management

This section describes how to control snapshots with Bensikin. A Snapshot is a view of your equipment at a precise date, view based on a context. A Snapshot has an ID, a creation date (Time), and a comment to describe it (which can be left empty).

Snapshots are managed in the snapshot control panel:

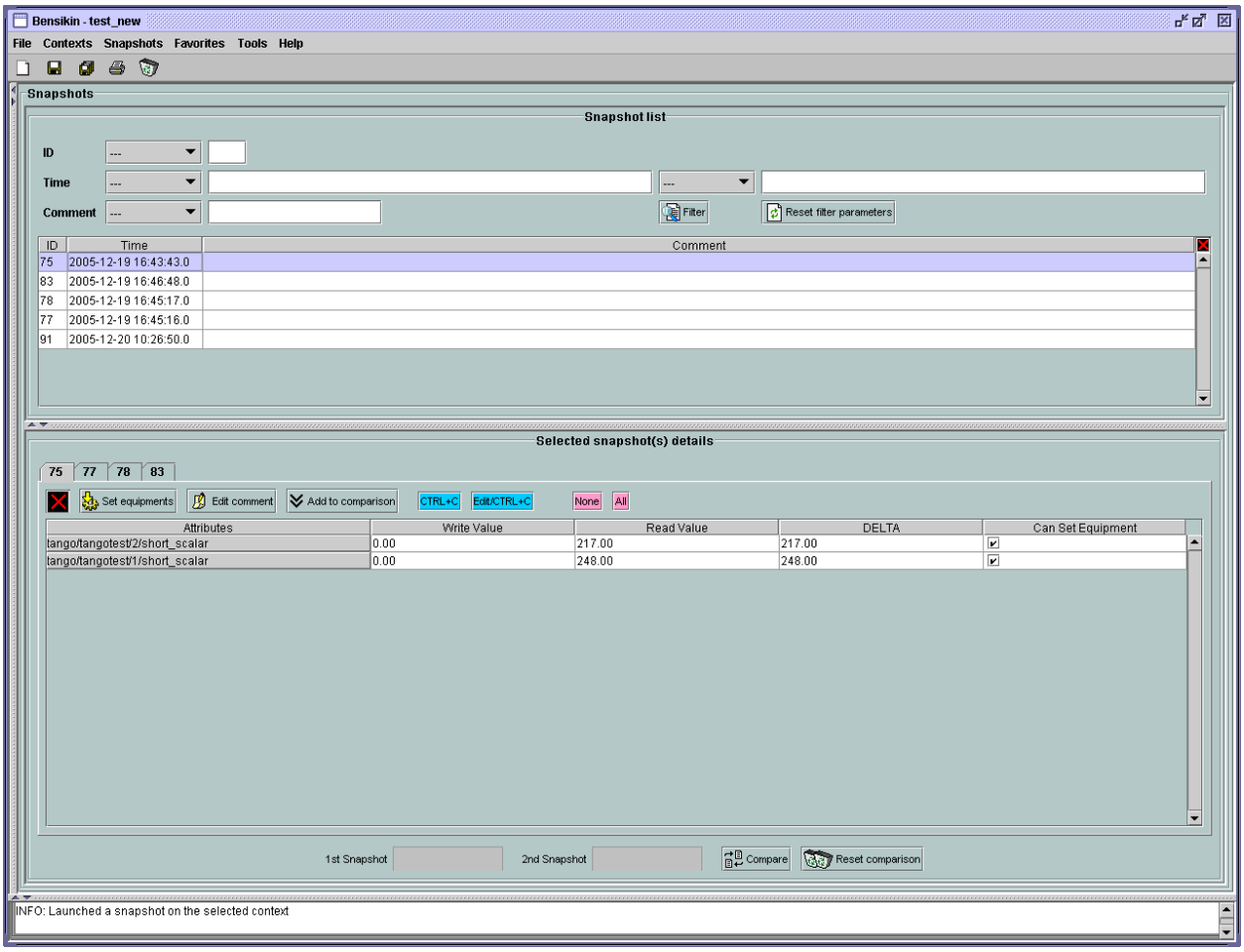


Fig. 36: Figure 12: Snapshot control panel

Creating a new snapshot

To create a new snapshot, first select a valid context in the context control panel (see *Figure 6: Context control panel*). Then click on the button

. The corresponding snapshot is added in the list of snapshots in the Snapshot List sub panel.

Loading a snapshot

There are 2 ways to load a snapshot:

- Load it from the database
- Load it from a file

Loading a snapshot from database

Loading a snapshot from database consists in adding this snapshot in the list of snapshots in the Snapshot List sub panel.

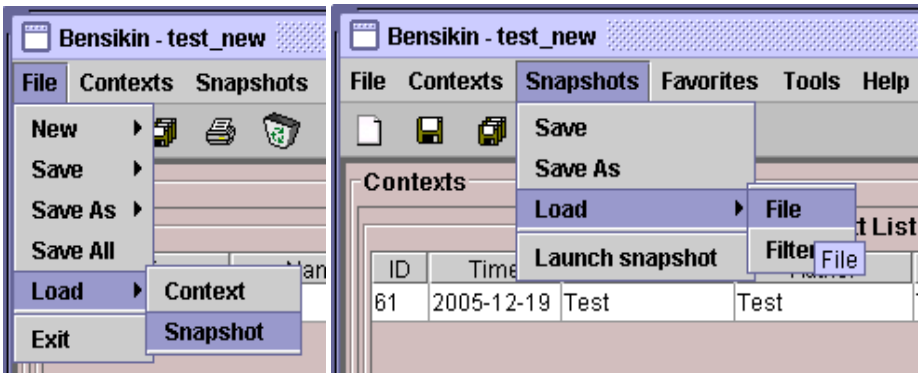
As you can see in *Figure 12: Snapshot control panel*, the *Snapshot List* sub panel allows you to filter snapshots from database to find the snapshot you want to load. However, have in mind that this filter is "context dependent", which means that the snapshots which will appear in the list by clicking on the *filter* button (

) are the one that correspond to your filter criteria **AND** the selected context in the *Context Control Panel*. If the filter is cleared (which you can obtain by clicking on the button

), you will search for all the snapshots in database that correspond to the selected context.

Loading a snapshot from file

In the *Snapshots* menu, choose *load* then select *File*, or in *File* menu choose *load* then select *Snapshot*:

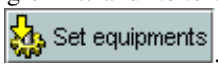


A classic file browser will appear. Search for your ".snap" file and select it to load the corresponding snapshot in the Snapshot Details sub panel (See *Figure 12: Snapshot control panel*)

Editing a snapshot

To edit a snapshot, double click on the snapshot you want to edit in the snapshot list (in the *Snapshot List* sub panel). This will open a new tab about this snapshot in the Snapshot Details sub panel, tab named by this snapshot ID. If you load a snapshot from file, the name of the tab is the name of the file. To differentiate snapshots loaded from file and the ones loaded from database, the snapshot loaded from file tabs have the icon .

Setting equipment with a snapshot

A snapshot allows you to set equipment with its attributes write values. You can choose which attributes will set equipment, and which not, by selecting or unselecting the corresponding check box in the column *Can Set Equipment* (See *Figure 12: Snapshot control panel*). By default, every attribute is selected. If you unselect some attributes, an icon will appear in tab title to notify you that these attributes will not set equipments. You can quick select/unselect all the attributes by clicking on *All* and *None* buttons. When you are ready to set equipment with the selected write values, click on the button 

.

You can also modify the write value before setting equipment by editing it in the table. If you do so, the value becomes red and an ★ icon appears to warn you about the fact that you made modifications in this snapshot (these modifications will not be saved in database, they are just here to set equipment).

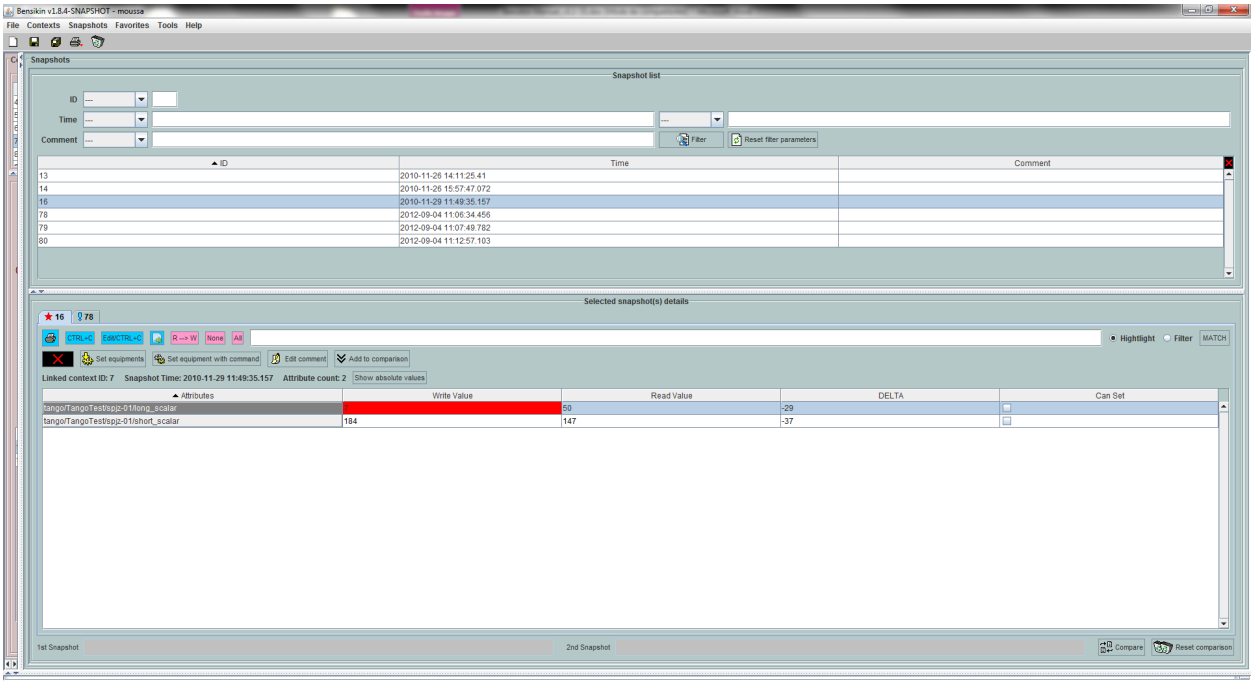


Fig. 37: Figure 13: Modified snapshot

Snapshot comparison

There are 2 ways to compare snapshots:

- Compare a snapshot with another one:

  To do so, select a tab in Snapshot Details sub panel (*Figure 12: Snapshot control panel*). Click on button Add to comparison. You will see the tab title of this attribute appear in the field "1 st snapshot". Select another tab and click again on Add to comparison button to put this attribute tab title in the field "2 nd snapshot". Click then on Compare button to see the comparison between these 2 snapshots.

  If user wants to see only the first line of comparison, he must check filter Filter

  Else if he/she wants to see all the details of the comparison, he/she must check Hightlight

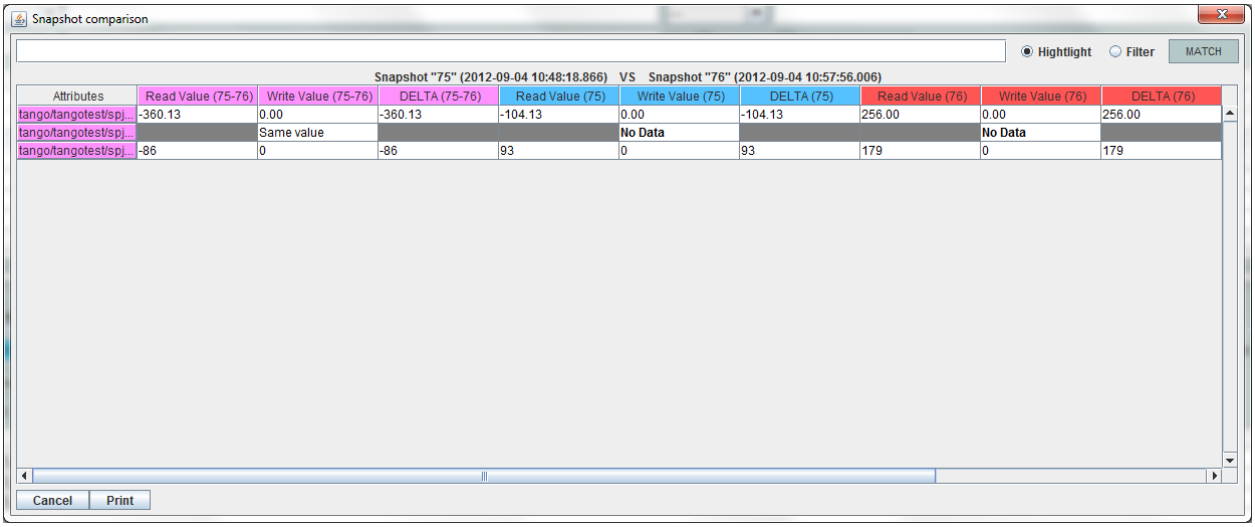


Fig. 38: Figure 14: Snapshot comparison - full table

To print this comparison table, click on *Print* button.

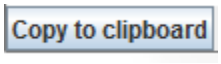
- Compare a snapshot with current state:

  To compare a snapshot with current state, set this snapshot as "1 st snapshot", as explained above, and leave the "2 nd snapshot" empty. Note that once the "1 st snapshot is selected, you only can update the "2 nd snapshot or clear the comparison selection. To do so, click on the button

. What is hidden behind this is a creation of a snapshot, named "BENSIKIN_AUTOMATIC_SNAPSHOT", and you compare this snapshot with your snapshot. Have in mind that this automatic snapshot is registered in database. So, in the comparison table, the current state will appear as the second snapshot with the name "Current state" (red block in the comparison table).

Snapshot Details copy

As you can see in *Figure 12: Snapshot control panel*, snapshots are detailed in a table. You can copy this table to clipboard as a text-CSV formatted table by clicking on

or 

button. If you want to see this text result and may be filter it (like removing lines), click on

button. You will see the text appear in a dialog.

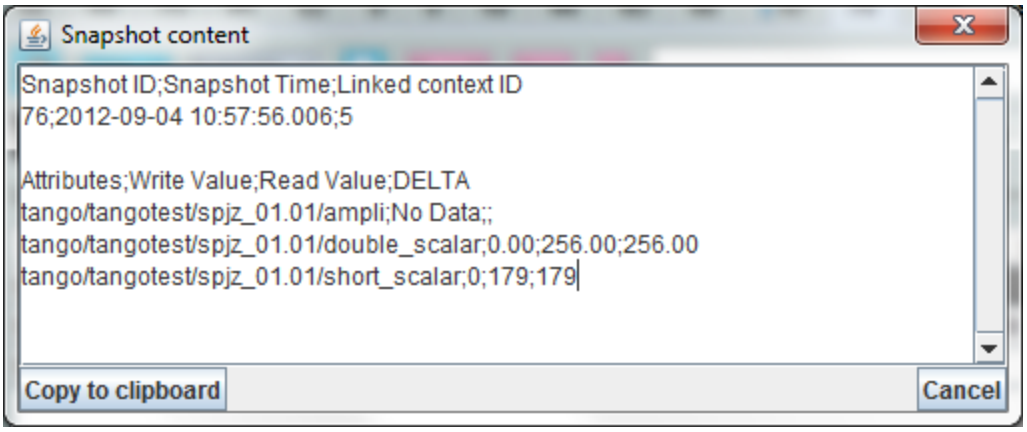


Fig. 39: Figure 15: Snapshot edit clipboard dialog

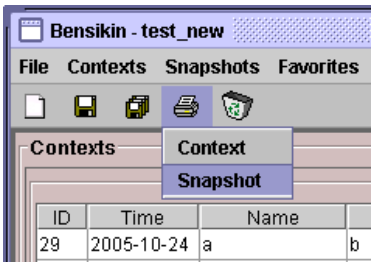
Modifying a snapshot comment

Once your snapshot details are loaded, click on

button to modify its comment (and save it in database or file).

Printing a snapshot

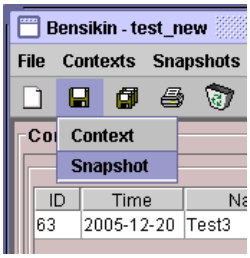
Once you have context ready, click on the *print* icon ( ) and select *snapshot*:



The classic print dialog will then appear. Validate your print configuration to print an xml representation of your snapshot.

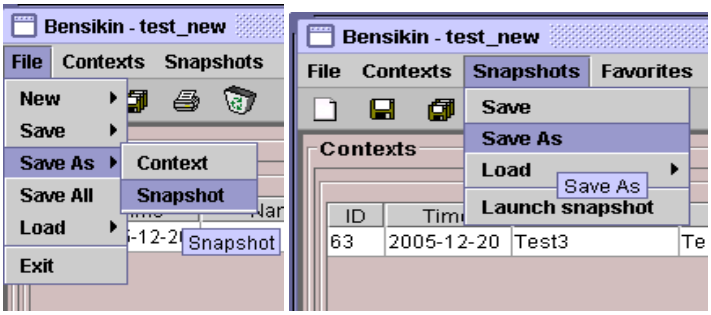
Saving a snapshot

Once you have context ready, click on the *save* icon ( ) and select *snapshot*:



You can also go to menu *Context* and click on *Save*, or go to menu *File -> Save -> Snapshot*.

Then, the behavior is "Word-like". This means that if this is the first time you save this snapshot, you will see the classic file browser to choose where to save your snapshot, with file name. However, if not, it will automatically save in the corresponding file. If you want to save in another file, you have to go to *File* menu, select *Save As* and click on *Snapshot*, or go to *Snapshots* menu and click on *Save As*.

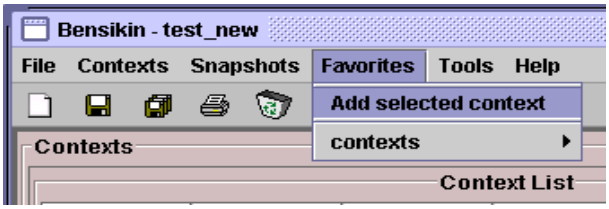


Favorites

Bensikin manages a list of favorite context, so you can quickly switch to anyone of them. Those favorites are saved at application shutdown and loaded on startup.

Adding a context to favorites

To add a context to your favorites, have your context ready by creating or loading it. Then go to *Favorites* menu and click on *Add selected context*.



Switching to a context in favorites

To switch to a context in favorites, which means to load it from favorites, go to "Favorites" menu, select "contexts", and click on the context you want to load.

Options

Bensikin manages global options. Those options are saved at application shutdown, and loaded on startup. The Options menu is located in the Menu bar: *Tools -> Options*.

Application's history save/load Options

Define whether Bensikin has a history, i.e. a persistent state when closed/reopened.

If yes is checked, a XML History file will be saved in Bensikin's workspace, and on next startup the current Context and Snapshot will be loaded.

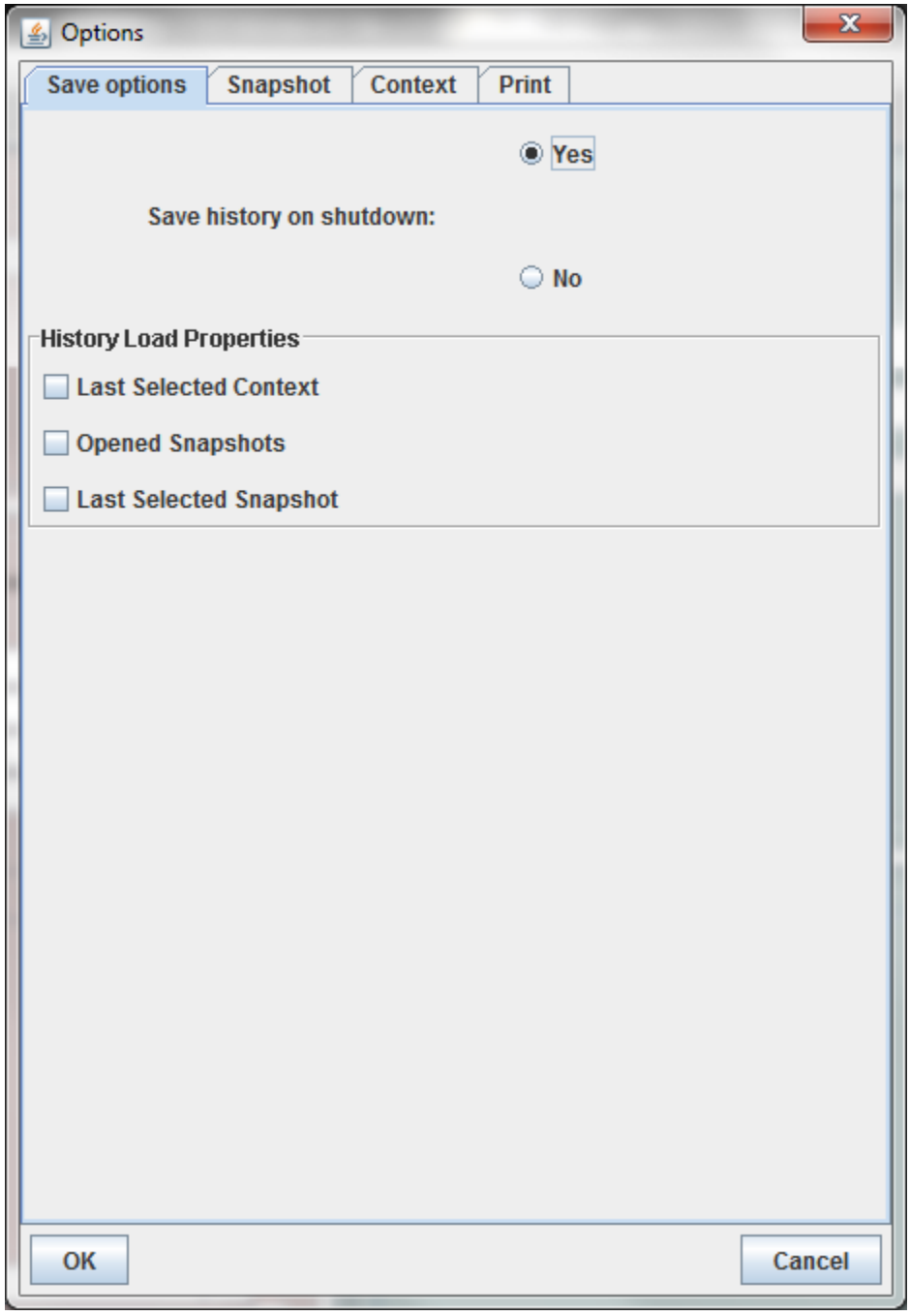


Fig. 40: Figure 16: Save option

Snapshot Options

These are the Bensikin Snapshot Options:

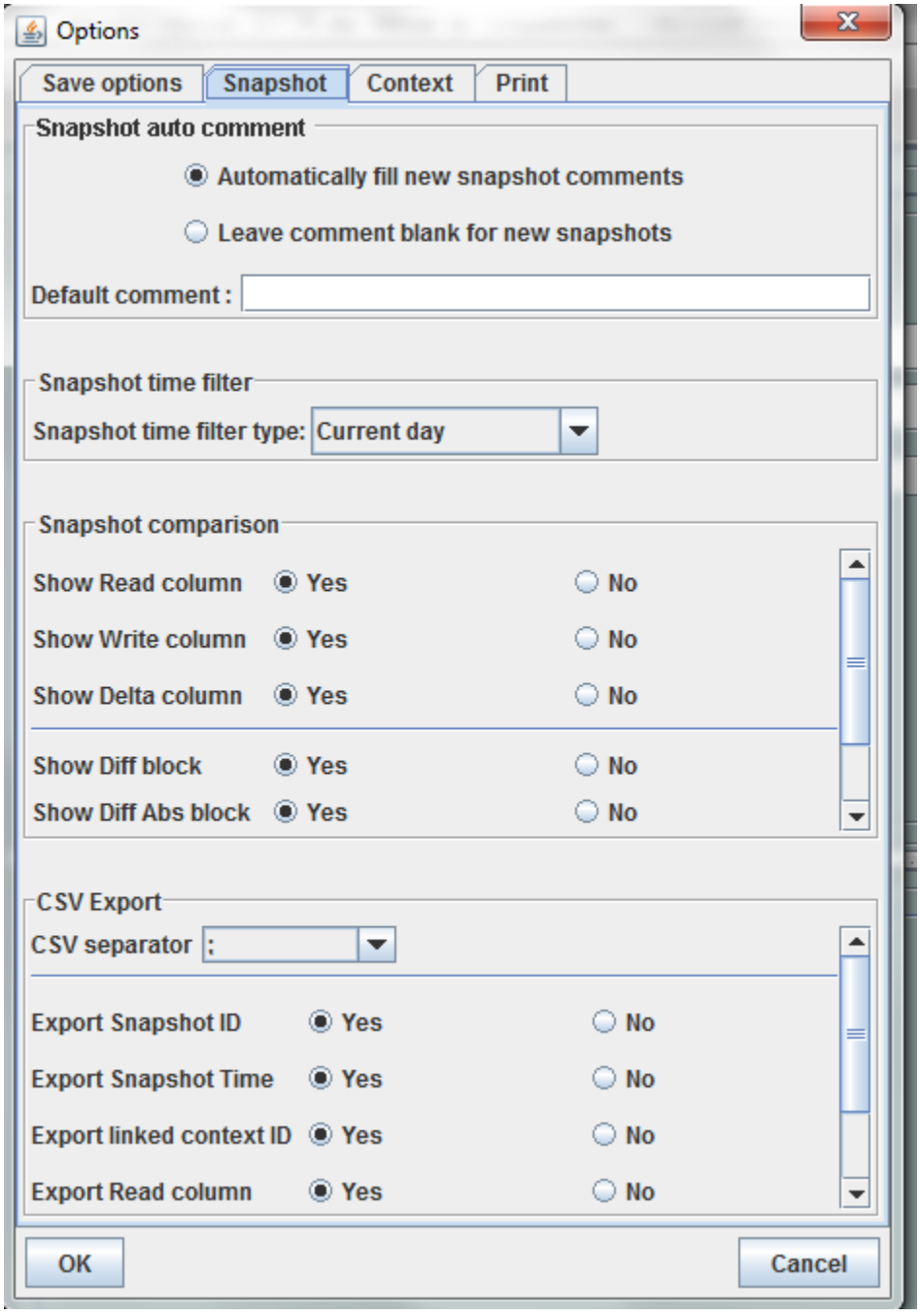


Fig. 41: Figure 17: Snapshot options

- In the Comment Panel, you can choose to automatically set or not a value to a new snapshot comment. This means, when you click on

button, the newly created snapshot will or will not have a predefined comment.

- In the *Comparison Panel*, you can choose which columns you want to show/hide for every block in the Snapshot Comparison table. You can choose to show/hide the Difference block too (See *Figure 14: Snapshot comparison - full table*)
- In the *Export Panel*, you can choose the column separator for your text-CSV formatted tables (See *Figure 15: Snapshot edit clipboard dialog*), and which columns to export.

Context Options

Context options allow you to select which way you want to edit your contexts, see *Figure 8: Option –context tab* and the *Creating a new snapshot* section.

Print Options

The Print option allows you to print text or table in the Snapshots or in the Contexts.

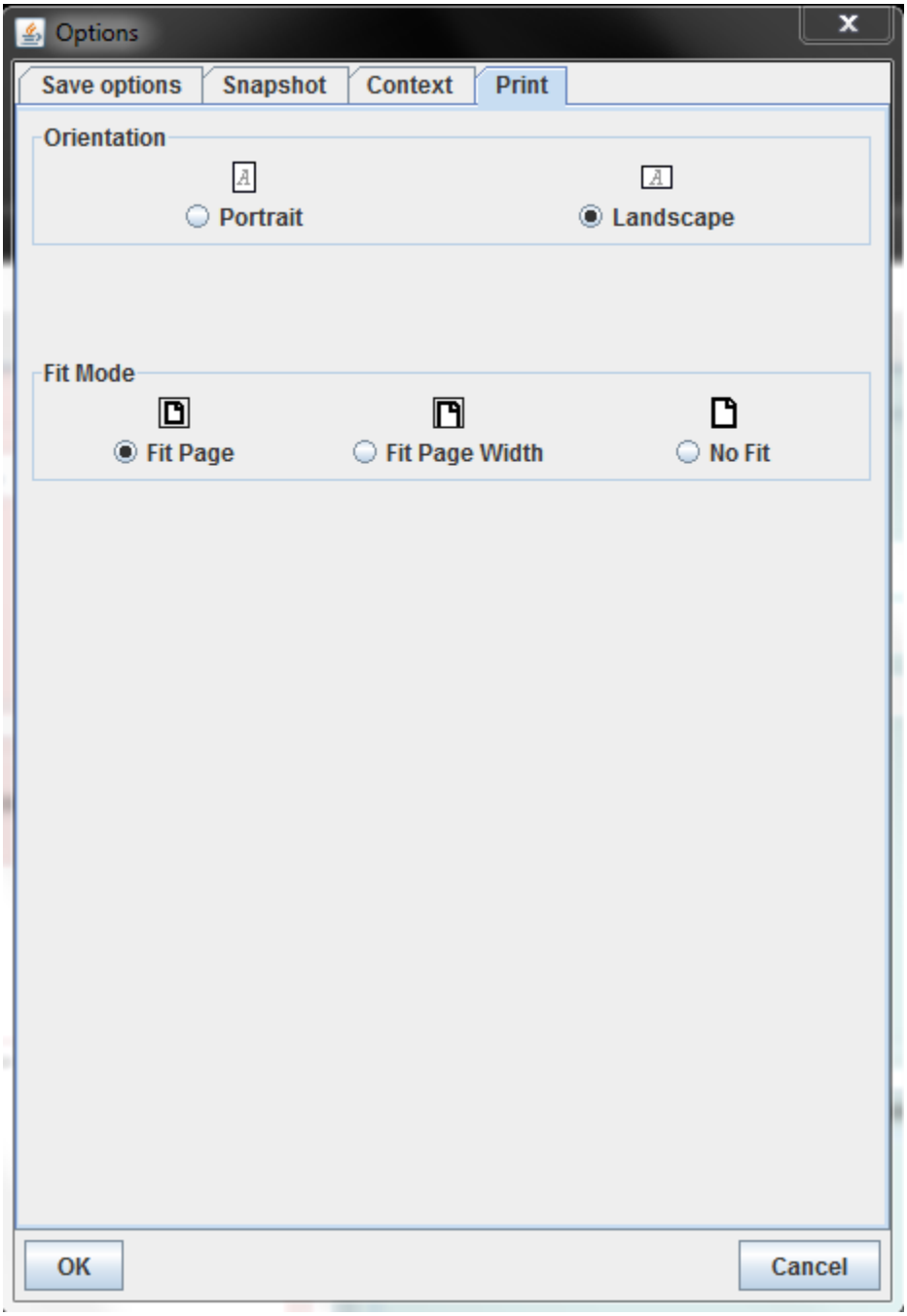
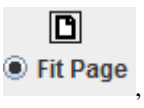
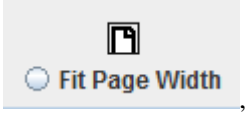


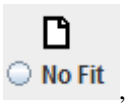
Fig. 42: Figure 18: Print option

When you check 

, you adapt the size of your print to the size of your page.

When you check 

, you cut the length of your print on several parts and the width of your print takes the width of your page.

When you check 

, the length and the width of your print are cut on several parts.

The Bensikin toolbar

The toolbar is located under the menu bar, and consists mainly of a set of shortcuts to often used functionalities.

Fig. 43: Figure 19: Bensikin toolbar

- is a shortcut to creating a new Context
- is a shortcut to saving the selected Context/Snapshot into a Context/Snapshot file
- is a shortcut to doing a saving all opened Contexts and Snapshots
- is a shortcut to printing the xml representation of the current Context/Snapshot
- is a shortcut to removing all opened Contexts and Snapshots from display

7.6 Bindings

7.6.1 Overview

Intended audience: developers, users

Tango has a number of bindings to other languages and tools. This clickable map shows you how the known bindings.

There may be others - ask on the forum if you need a binding which is not listed here or would like to publish a binding you have written.

7.6.2 C Language

Intended audience: developers

C is supported for clients written in old style C (like SPEC). The C binding does not support all features of TANGO. Clients are encouraged to use the C++ api if they need access to all features of TANGO.

- **C language**
  - A minimal client binding for the good old C language
  - Release 3.0.2 including features of Tango 8 but no events
  - Source code distribution

7.6.3 Matlab & Octave

Intended audience: developers, users

- Client API for Matlab and Octave
- Release 2.0.6 for Matlab >= R2009b or Octave >= 3.6.2
- Runs on Windows and Linux
- **Provides 64 bits support on both platforms**
  - the 32 bits mode is still available on Linux but could be abandoned in a near future
- **Binary distribution 2.0.6 for Windows x86**
  - tested with Matlab R2009b
- **Binary distribution 3.1.0 for Windows x64**
  - tested with Matlab R2016b
  - this release contains a major change - see the README file for details
- **Source code available on GitHub**
  - please visit the MathWorks web site in order to identify the official gcc version associated with your Matlab version

7.6.4 LabVIEW

Intended audience: developers

- Client and server API for LabVIEW
- Runs on Windows and Linux
- Provides 32 and 64 bits support on both platforms
- **Release 3.0.0 for LabVIEW 2015**
  - Binary distribution for Windows [LabVIEW 32 bits]
- **Release 3.0.0 for LabVIEW 2014**
  - Binary distribution for LabView 2014 on Windows [LabVIEW 64 bits]
  - Binary distribution for Linux [LabVIEW 32 bits]
- **Patch for release 3.0.0**
  - requires LabVIEW >= 2014
  - this Vi library contains some bug fixes, simply replace the original one in the /vis directory
- Source code available on GitHub

7.6.5 Igor Pro

Intended audience: developers, users

- Client API for Igor Pro
- **Release 3.0.0 for Igor Pro 7.x**
  - Runs on Windows x64 [no more official support for x86]
  - Binary distribution for 7.x for Windows x64
- **Release 2.5.0 for Igor Pro 6.x**
  - Runs on Windows x86 & x64
  - Binary distribution for 6.x for Windows x86
- **Source code available on GitHub**
  - compiling this binding requires the WaveMetrics' XOP Toolkit

7.6.6 Panorama

Intended audience: developers

The Panorama binding is available thanks to CEA

- Release 1.2 for Tango 7.2.1 and Panorama E2 4.0 SP1 (or above)
- Runs on Windows 32 and 64 bits.
- Binary and source distribution for Windows 32 and 64 bits. This distribution consists of an ISO file of the full source and binary CD distribution. You can use this iso file to burn a CD or to initialize a USB memory stick (using for example isotousb). In this distribution, you will find:
  - A windows installer
  - Documentation (in French)
  - Tango 7.2.1
  - A set of libraries allowing a Panorama E2 application to control Tango device(s)
  - A binary application (TangoExplorer) which automatically generates a Panorama E2 application from all available Tango devices.

7.7 Other tools

7.7.1 PANIC Alarm System

PANIC is a set of tools (api, Tango device server, user interface) that provides:

- Periodic evaluation of a set of conditions.
- Notification (email, sms, pop-up, speakers)
- Keep a log of what happened. (files, Tango Snapshots)
- Taking automated actions (Tango commands / attributes)
- Tools for configuration/visualization

- The Panic package contains the python AlarmAPI for managing the PyAlarm device servers from a client application or a python shell. The panic module is used by PyAlarm, Panic Toolbar and Panic GUI.

All the information connected to the project can be found visiting the following links:

- PANIC on github
- PANIC presentation at PCAPAC'14
- PANIC documentation

CHAPTER

EIGHT

ADMINISTRATION

Intended audience: administrators

In addtion to the articles listed below you may find *Tutorials for administrators* and *HOW-TOs for administrators* interesting. Please also look to *a list of articles in all sections which may relate to system administration*.

8.1 Overview

Intended audience: all, administrators

As soon as a Tango system grows to over a few devices and servers it needs to be administered. A number of tools have been developed to administer a Tango control system.

This clickable map gives and overview of the administrative tools.

Intended audience: administrators

8.2 Deployment

8.2.1 Starting a Tango control system

Intended audience: all

Without database

When used without database, there is no additional process to start. Simply starts device server using the -nodb option (and eventually the -dlist option) on specific port. See *Running a device server without SQL database* to find information on how to start/write Tango device server not using the database.

With database

Starting the Tango control system simply means starting its database device server on a well defined host using a well defined port. Use the host name and the port number to build the TANGO_HOST environment variable. See *environment variable* to find how starting a device server on a specific host. Obviously, the underlying database software (MySQL) must be started before the Tango database device server. The Tango database server connects to MySQL using a default logging name set to root. You can change this behaviour with the MYSQL_USER and MYSQL_PASSWORD environment variables. Define them before starting the database server.

If you are using the Tango administration graphical tool called **Astor**, you also need to start a specific Tango device server called *Starter* on each host where Tango device server(s) are running. See [ASTOR] for Astor documentation. This starter device server is able to start even before the Tango database device server is started. In this case, it will enter a loop in which it periodically tries to access the Tango database device. The loop exits and the server starts only if the database device access succeed.

With database and event

For Tango releases lower than 8

On top of what is described in the previous chapter, using event means using CORBA Notification service. Start one Notification Service daemon on each host where device server(s) used via events are running. The Notification Service daemon event channel factory IOR has to be registered in the Tango database. This is done with the **notifd2db** command. The notification daemon is a process with a high thread number. By default, Unix like operating systems reserve a big amount of memory for each thread stack (8 MByte for Linux/Ubuntu, 10 MByte for Linux/RedHat 4). If your process has several hundreds of threads, this could generate a too high memory requirement on virtual memory and even exceed the maximun allowed memory per process (3 GBytes on Linux for 32 bits computer). The notification service daemon works very well with a value of only 2 Mybtes for thread stack. The Unix command line ulimit -s 2048 asks the operating system to give 2 Mbytes for each thread stack. Example of starting and registering a Notification Service daemon on a UNIX like operating system

```
1 ulimit -s 2048
2 notifd -n -DDeadFilterInterval=300 &
3 notifd2db
```

The Notification Service daemon is started at line 2. Its -DDeadFilterInterval option is used to specify some internal cleaning of dead objects within the notification service. The -n option is used to disable the use of the CORBA Naming Service for registering the default event channel factory. The registration of the Notification Service daemon in the Tango database is done at line 2.

It differs on a Windows computer

```
1 notifd -n -DDeadFilterInterval=300 -DFactoryIORFileName=C:\Temp\evfact.ior &
2 notifd2db C:\Temp\evfact.ior
```

For release 8 and above

A new event system has been implemented starting with Tango release 8. With this new event system, the CORBA Notification service is not needed any more. This means that **as soon** as all Tango device server processes running on a host and all clients using events from their devices used Tango 8, it is not required to start any process other than the device servers and the clients. You can forget the previous sub-chapter!

With file used as database

When used with database on file, there is no additional process to start. Simply starts device server using the -file option specifying file name port. See *Device server using file as database* to find information on how to start Tango device server using database on file.

With file used as database and event

For Tango releases lower than 8

Using event means using CORBA Notification service. Start one Notification Service daemon on the host where device server(s) using events are running. The Notification Service daemon event channel factory IOR has to be registered in the file(s) use as database. This is done with the **notifd2db** command. Example of starting and registering a Notification Service daemon on a UNIX like operating system

```
1  notifd -n -DDeadFilterInterval=300 &
2  notifd2db -o /var/myfile.res
```

The Notification Service daemon is started at line 1. Its -n option is used to disable the use of the CORBA Naming Service for registering the default event channel factory. The registration of the Notification Service daemon in the file used as database is done at line 2 with its **-o** command line option.

It differs on a Windows computer because the name of the file used by the CORBA notification service to store its channel factory IOR must be specified using its -D command line option. This file name has also to be passed to the notifd2db command.

```
1 notifd -n -DDeadFilterInterval=300 -DFactoryIORFileName=C:\Temp\evfact.ior &
2 notifd2db C:\Temp\evfact.ior -o C:\Temp\myfile.res
```

For release 8 and above

A new event system has been implemented starting with Tango release 8. With this new event system, the CORBA Notification service is not needed any more. This means that **as soon** as all clients using events from devices embedded in the device server use Tango 8, it is not required to start any process other than the device server and its clients.

With the controlled access

Using the Tango controlled access means starting a specific device server called TangoAccessControl. By default, this server has to be started with the instance name set to 1 and its device name is sys/access_control/1. The command line to start this device server is:

```
TangoAccessControl 1
```

This server connects to MySQL using a default logging name set to root. You can change this behaviour with the MYSQL_USER and MYSQL_PASSWORD environment variables. Define them before starting the controlled access device server. This server also uses the MYSQL_HOST environment variable if you need to connect it to some MySQL server running on another host. The syntax of this environment varaible is host:port. Port is optional and if it is not defined, the MySQL default port is used (3306). If it is not defined at all, a connection to the localhost is made. This controlled access system uses the Tango database to retrieve user rights and it is not possible to run it in a Tango control system running without database.

8.2.2 The Starter device

Intended audience: administrators, developers, users

Introduction

The Starter device can be used to control other device servers running on the same host.

Possible use cases are:

- start all device servers on system startup,
- get list of all started device servers,
- start or stop a device server,
- get logs from a device server.

Astor is a graphical client for Starter devices.

Installation

There are several ways to install the Starter device server:

- *as a part of Tango source distribution*,
- from Starter source,
- *using a package manager* (e.g. tango-starter for debian),
- *from binary package (Windows)*.

Configuration

General recommendations for Starter configuration:

- there should be only one instance of a Starter device server running on a host,
- the instance name can be arbitrary,
- the Starter device server provides a *Starter* class,
- the *member* part of a Starter device name must match the short name of the host.
- the *domain/family* part of a Starter device name should be "tango/admin" in order for the Starter device to be detected by Astor.

An example how to define Starter device using *tango_admin* tool:

```
host=$(hostname -s)
tango_admin --add-server Starter/$host Starter tango/admin/$host
Starter $host
```

Note: The requirement for *member* part of the name to match the hostname can be disabled by setting environment variable `DEBUG` to `true`. The starter will be visible in Astor under the name specified in *member*.

Usage

The Astor GUI can be used to *start a new Tango device server* using the Starter device.

Alternatively, the Starter interface can be used directly, e.g. to start a device server:

```
import tango
starter = tango.DeviceProxy("tango/admin/tangobox")
starter.command_inout("DevStart", "TangoTest/test")
```

The Starter device will use the paths from the `StartDsPath` property when looking for an executable of a device server to start. The paths are searched in the order in which they appear in the property. If the property is empty, Starter device server's working directory is used when searching for executables.

Autostarting Starter

The Starter device server can be automatically started during system startup using a service manager of choice. It can then start any other device servers.

Following are example configuration files for different service managers.

systemd

Follow *instructions for systemd integration*.

System V init

Follow *instructions in the installation guide*.

NSSM (Windows)

Follow *NSSM configuration instructions in the installation guide for Windows*.

References

- Starter documentation,
- Starter in the Device Servers Catalogue,
- Starter source code repository.

8.2.3 Running a device server without SQL database

Intended audience: administrators, developers

Device server using file as database

For device servers not able to access the Tango database (most of the time due to network route or security reason), it is possible to start them using file instead of a real database. This is done via the device server

-file=<file name>

command line option. In this case,

- Getting, setting and deleting class properties
- Getting, setting and deleting device properties
- Getting, setting and deleting class attribute properties
- Getting, setting and deleting device attribute properties

are handled using the specified file instead of the Tango database. The file is an ASCII file and follows a well-defined syntax with predefined keywords. The simplest way to generate the file for a specific device server is to use the Jive application. See [Jive] to get Jive documentation. The Tango database is not only used to store device configuration parameters, it is also used to store device network access parameter (the CORBA IOR). To allow an application to connect to a device hosted by a device server using file instead of database, you need to start it on a pre-defined port, and you must use one of the underlying ORB option called *endPoint* like

myserver myinstance_name -file=/tmp/MyServerFile -ORBendPoint giop:tcp::<port number>

to start your device server. The device name passed to the client application must also be modified in order to refect the non-database usage. See *device naming* to learn about Tango device name syntax. Nevertheless, using this Tango feature prevents some other features to be used :

- No check that the same device server is running twice.

- No device or attribute alias name.
- In case of several device servers running on the same host, the user must manually manage a list of already used network port.

Device server without database

In some very specific cases (Running a device server within a lab during hardware development...), it could be very useful to have a device server able to run even if there is no database in the control system. Obviously, running a Tango device server without a database means loosing Tango features. The lost features are :

- No check that the same device server is running twice.
- No device configuration via properties.
- No event generated by the server.
- No memorized attributes
- No device attribute configuration via the database.
- No check that the same device name is used twice within the same control system.
- In case of several device servers running on the same host, the user must manually manage a list of already used network port.

To run a device server without a database, the **-nodb** command line option must be used. One problem when running a device server without the database is to pass device name(s) to the device server. Within Tango, it is possible to define these device names at two different levels :

1. At the command line with the **-dlist** option: In case of device server with several device pattern implementation, the device name list given at command line is only for the last device pattern created in the *class_factory()* method. In the device name list, the device name separator is the comma character.
2. At the device pattern implementation level: In the class inherited from the Tango::DeviceClass class via the re-definition of a well defined method called *device_name_factory()*

If none of these two possibilities is used, the tango core classes defined one default device name for each device pattern implementation. This default device name is *NoName*. Device definition at the command line has the highest priority.

Example of device server started without database usage

Without database, you need to start a Tango device server on a pre-defined port, and you must use one of the underlying ORB option called *endPoint* like

```
myserver myinstance_name -ORBendPoint giop:tcp::<port number> -nodb -dlist a/b/c
```

The following is two examples of starting a device server not using the database when the *device_name_factory()* method is not re-defined.

- StepperMotor et -nodb -dlist id11/motor/1,id11/motor/2
  This command line starts the device server with two devices named *id11/motor/1* and *id11/motor/2*
- StepperMotor et -nodb
  This command line starts a device server with one device named *NoName*

When the *device_name_factory()* method is re-defined within the StepperMotorClass class.

```
void StepperMotorClass::device_name_factory(vector<string> &list)
{
    list.push_back("sr/cav-tuner/1");
    list.push_back("sr/cav-tuner/2");
}
```

- StepperMotor et -nodb
  This commands starts a device server with two devices named *sr/cav-tuner/1* and *sr/cav-tuner/2*.
- StepperMotor et -nodb -dlist id12/motor/1
  Starts a device server with only one device named id12/motor/1

Connecting client to device within a device server started without database

In this case, the host and port on which the device server is running are part of the device name. If the device name is *a/b/c*, the host is *mycomputer* and the port *1234*, the device name to be used by client is

mycomputer:1234/a/b/c#dbase=no

Some clients like atkpanel require *tango://* prefix:

tango://mycomputer:1234/a/b/c#dbase=no

See *device naming* for all details about Tango object naming.

Intended audience: administrators

8.2.4 Multiple database servers within a Tango control system

Tango uses MySQL as database and allows access to this database via a specific Tango device server. It is possible for the same Tango control system to have several Tango database servers. The host name and port number of the database server is known via the TANGO_HOST environment variable. If you want to start several database servers in order to prevent server crash, use the following TANGO_HOST syntax

TANGO_HOST=<host_1>:<port_1>,<host_2>:<port_2>,<host_3>:<port_3>

All calls to the database server will automatically switch to a running servers in the given list if the one used dies.

8.2.5 The property file syntax

Intended audience: administrators, developers

Property file usage

A property file is a file where you store all the property(ies) related to device(s) belonging to a specific device server process. In this file, one can find:

- Which device(s) has to be created for each Tango class embedded in the device server process
- Device(s) properties
- Device(s) attribute properties

This type of file is not required by a Tango control system. These informations are stored in the Tango database and having them also in a file could generate some data duplication issues. Nevertheless, in some cases, it could very very helpful to generate this type of file. These cases are:

1. If you want to run a device server process on a host which does not have access to the Tango control system database. In such a case, the user can generate the file from the database content and run the device server process using this file as database (-file option of device server process)
2. In case of massive property changes where no tool will be more adapted than your favorite text editor. In such a case, the user can generate a file from the database content, change/add/modify file contents using his favorite tool and then reload file content into the database.

Jive (JIVE home page) is the tool provided to generate and load a property file. To generate a device server process properties file, select your device server process in the Server tab, right click and select Save Server Data. A file selection window pops up allowing you to choose your file name and path. To reload a file in the Tango database, click on File then Load Property File.

Property file syntax

```
#-------------------------------------------------------------
# SERVER TimeoutTest/manu, TimeoutTest device declaration
#-------------------------------------------------------------

TimeoutTest/manu/DEVICE/TimeoutTest: "et/to/01",\
                                     "et/to/02",\
                                     "et/to/03"


# --- et/to/01 properties

et/to/01->StringProp: Property
et/to/01->ArrayProp: 1,\
                     2,\
                     3
et/to/01->attr_min_poll_period: TheAttr,\
                                1000
et/to/01->AnotherStringProp: "A long string"
et/to/01->ArrayStringProp: "the first prop",\
                           "the second prop"

```

(continues on next page)

(continued from previous page)

```
22 # --- et/to/01 attribute properties
23
24 et/to/01/TheAttr->display_unit: 1.0
25 et/to/01/TheAttr->event_period: 1000
26 et/to/01/TheAttr->format: %4d
27 et/to/01/TheAttr->min_alarm: -2.0
28 et/to/01/TheAttr->min_value: -5.0
29 et/to/01/TheAttr->standard_unit: 1.0
30 et/to/01/TheAttr->__value: 111
31 et/to/01/BooAttr->event_period: 1000doc_url
32 et/to/01/TestAttr->display_unit: 1.0
33 et/to/01/TestAttr->event_period: 1000
34 et/to/01/TestAttr->format: %4d
35 et/to/01/TestAttr->standard_unit: 1.0
36 et/to/01/DbAttr->abs_change: 1.1
37 et/to/01/DbAttr->event_period: 1000
38
39 CLASS/TimeoutTest->InheritedFrom:   Device_4Impl
40 CLASS/TimeoutTest->doc_url:   "http://www.esrf.fr/some/path"
```

Line 1 - 3: Comments. Comment starts with the '#' character

Line 4: Blank line

Line 5 - 7: Devices definition. DEVICE is the keyword to declare a device(s) definition sequence. The general syntax is:

<DS name>/<inst name>/DEVICE/<Class name>: dev1,dev2,dev3

Device(s) name can follow on next line if the last line character is '\' (see line 5,6). The " characters around device name are generated by the Jive tool and are not mandatory.

Line 12: Device property definition. The general device property syntax is

<device name>**->**<property name>: <property value>

In case of array, the array element delimiter is the character ','. Array definition can be splitted on several lines if the last line character is '\'. Allowed characters after the ':' delimiter are space, tabulation or nothing.

Line 13 - 15 and 16 - 17: Device property (array)

Line 18: A device string property with special characters (spaces). The " character is used to delimit the string

Line 24 - 37: Device attribute property definition. The general device attribute property syntax is

<device name>/<attribute name>**->**<property name>: <property value>

Allowed characters after the ':' delimiter are space, tabulation or nothing.

Line 39 - 40: Class property definition. The general class property syntax is

CLASS/<class name>**->**<property name>: <property value>

CLASS is the keyword to declare a class property definition. Allowed characters after the ':' delimiter are space, tabulation or nothing. On line 40, the " characters around the property value are mandatory due to the '/' character contains in the property value.

8.2.6 High Availability

Intended audience: administrators

Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specific time interval. Improve availability means minimize service outage time by mitigating faults.

Tactics to improve the availability

In order to improve the availability of Tango, a list of tactics could be used. Some of these are Tango functionality, others are best practices or suggested solutions. Using or not the different tactics depends on use cases.

Fault Detection

Before any system can take action regarding a fault, the presence of the fault must be detected. The tactic of Tango system to detect a fault are:

- **Ping**: refers to an asynchronous exchange of message between nodes, in order to determine if the pinged component is alive and responding properly. Tango Ping is performed by Astor, that contacts all the starters defined for given tango host, and each starter is pinging regularly (sending ping command) all the device servers of its host.
- **Monitor** is a component that is used to monitor the health status of a host (monitoring processor, process and so on). Astor is used to monitor the status of device servers (running/starting/not running). The device servers are grouped by host in a tree. Several hosts can be listed in a branch of the tree. It is a good practice to put the not critical hosts in branches listed at the end of the tree. The last branches to be listed by Astor can be configured from "File → Ctrl System Preferences → Last Collections" in Astor. The status of a group of device servers is represented by different colours:
  - *Red*: Starter is not running or cannot be reached by Astor
  - *Green*: all device servers necessary (or critical) for the operation are running
  - *Orange*: at least one of the necessary (or critical) device server for the operation is not running
  - *White*: all starters are running and all device servers from this host are stopped (useful for a full restart of a big control system. "File→ Ctrl System Preferences → Starter starts servers at startup" option in Astor is very useful if one needs to restart all the computers of a big control system in a very short period of time without overloading the Tango database).
  - *Blue:* At least one device server is starting or stopping.

  One best practice is to use a different software system monitor (like Nagios) in order to monitor the status of the applications or resources (e.g. processor usage, disk load) and the network (e.g. congestion of a switch).

  It is also possible to develop a device server that monitors network and resources, as the network collisions, network errors, operating system and so on. If an anomaly is detected, a Tango alarm is sent.
- **Heartbeat**: is a fault mechanism that sends a periodic message to ensure that the communication between the components works properly. Every Tango device server sends periodic messages (Tango heartbeat events) to all the clients interested in Tango events sent from this device server. Please refer to *Device server process (Please note that the document has to be updated because it refers to the previous Tango event implementation using the Corba notification daemon).*
- **Timestamp**: a timestamp is associated with every Tango attribute value. For example, in order to help evaluating the latency in the storing of events, HDB++ stores the timestamp of the Tango attribute data, the timestamp when

the HDB++ Event Subscriber received the event and the timestamp when the HDB++ Event Subscriber tried to store the data into the database.

- **Sanity checking**: check the correctness and the validity of a specific operation or outputs of a component. Tango device server can change the attribute quality factor to invalid if the attribute value is not making sense in the current state
- **Condition Monitoring**: is a best practice to change the state and status of the device or quality factor of an attribute to alarm when behaviour of the system is different than the expected conditions
- **Self-test**: components can perform procedures to test themselves. By default Tango doesn't implement self test, but a device server programmer could easily implements a self-test command or add a self-test during the init sequence of device.

Recovery from fault

Recovery from a fault is composed by a series of tactics that are concerned with reintroducing a failed component back into normal operation:

- **Replication**: It is a good practice to replicate the Tango Database. One can use MySQL/MariaDb replication feature to automatically replicate the data to a spare MySQL/MariaDb server. Recovery is possible manually by exchanging the faulty MySQL/MariaDb server with its slave.
- **Redundancy**: a best practice is to use the native redundancy for the database server, using MySQL NDB Cluster MySQL engine in order to remove the single point of failure.

Note: Tango provides the possibility to run several Tango Database Servers on the same control system. See: *Multiple database servers...*.

It is also a good solution that the redundancy of the device servers is managed by the programmer based on the use cases.

- **Exception Handling/Retry**: Once a fault has been detected, the system has to handle it. Tango uses the tactic of retry. (AutoRestartDuration property of the starter device). The Starter can automatically restart a device server in case of failure.
- **Ignore Faulty Behaviour/Degradation**: In case of database failure, all the clients and the device servers already connected together continue to communicate until they need to access to the database (in case of reconnection, if the server needs to write a new property or to memorize an attribute)

Prevent Fault

- **Exception prevention**: A good solution is using a software system monitor to collect monitoring data in order to analyse the trend of the normal behaviour and extract information that helps the prevention of fault.

Note: Tango will automatically change the attribute quality factor to WARNING or ALARM and eventually change the state of the device to ALARM if the attribute value is reaching the warning or alarm levels defined via min_alarm, max_alarm, min_warning and max_warning attribute properties. This feature can be used to raise alarms for instance or some specific software could be developed to monitor the state of the device or the quality factor of a specific attribute and eventually automatically intent a recovery procedure.

8.2.7 Tango Controls versions in use

Intended audience: all

ALBA Synchrotron:

- Tango Controls versions: 7.2.x/8.1.x/9.2.x/9.3.3
- PyTango versions: 7.2.x/8.1.x/9.2.x
- JTango versions: Released with 7.x and 8.1.x c++ version/8.4/9.2.x/9.3.x/**
- OS: *nix/Windows

DESY:

- Tango Controls versions: 9.2.x
- PyTango versions: 9.2.x
- JTango versions: 9.1.x (Released with c++ 9.2.5)
- OS: *nix/Windows

Elettra:

- Tango Controls versions: 7.2.x/8.1.x/9.2.x/9.3.3
- PyTango versions: 7.2.x/9.2.x/9.3.2
- JTango versions: 9.1.x (Released with c++ 9.2.5)
- OS: *nix

ESRF accelerator:

- cppTango versions: 9.3.6-rc2
- PyTango versions: 9.3.6
- JTango versions: 9.6.6
- OS: *nix/Windows

MAX IV Laboratory:

- Tango Controls versions: 9.3.5
- PyTango versions: 9.3.3
- JTango versions: 9.3.x
- OS: *nix

NSRC Solaris:

- Tango Controls versions: 8.1.x/9.2.x/9.3.3
- PyTango versions: 9.2.x
- OS: *nix/Windows

SKA Organisation:

- Tango Controls versions: 9.3.4-rc4
- PyTango versions: 9.3.2
- OS: *nix

SOLEIL:

- Tango Controls versions: 6.x/7.2.x/9.2.x
- PyTango versions: 8.1.x/9.2.x
- JTango versions: 9.5.x
- OS: *nix/Windows

Square Kilometre Array:

- Tango Controls versions: 9.3.3
- PyTango versions: 9.3.1
- JTango versions: 9.5.x
- OS: *nix

IJCLab (ThomX):

- Tango Controls versions: 9.2.5
- PyTango versions: 9.2.0
- JTango versions: 9.2.5
- OS: *nix

8.3 Services

Intended audience: administrators

Intended audience: administrators, developers

8.3.1 Event system

Using multicast protocol to transfer events

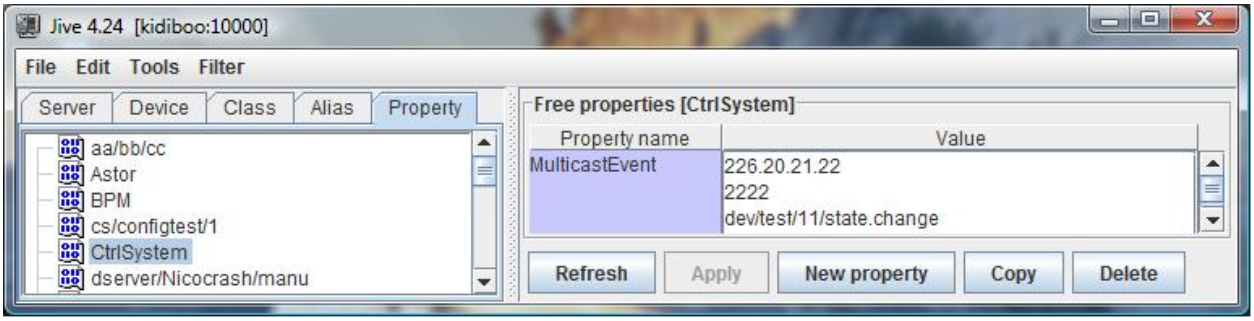
This feature is available starting with Tango 8.1. Transferring events using a multicast protocol means delivering the events to a group of clients simultaneously in a single transmission from the event source. Tango, through ZMQ, uses the OpenPGM multicating protocol. This is one implementation of the PGM protocol defined by the RFC 3208 (Reliable multicasting protocol). Nevertheless, the default event communication mode is unicast and propagating events via multicasting requires some specific configuration.

Configuring events to use multicast transport

Before using multicasting transport for event(s), you have to choose which address and port have to be used. In a IP V4 network, only a limited set of addresses are associated with multicasting. These are the IP V4 addresses between

224.0.1.0 and 238.255.255.255

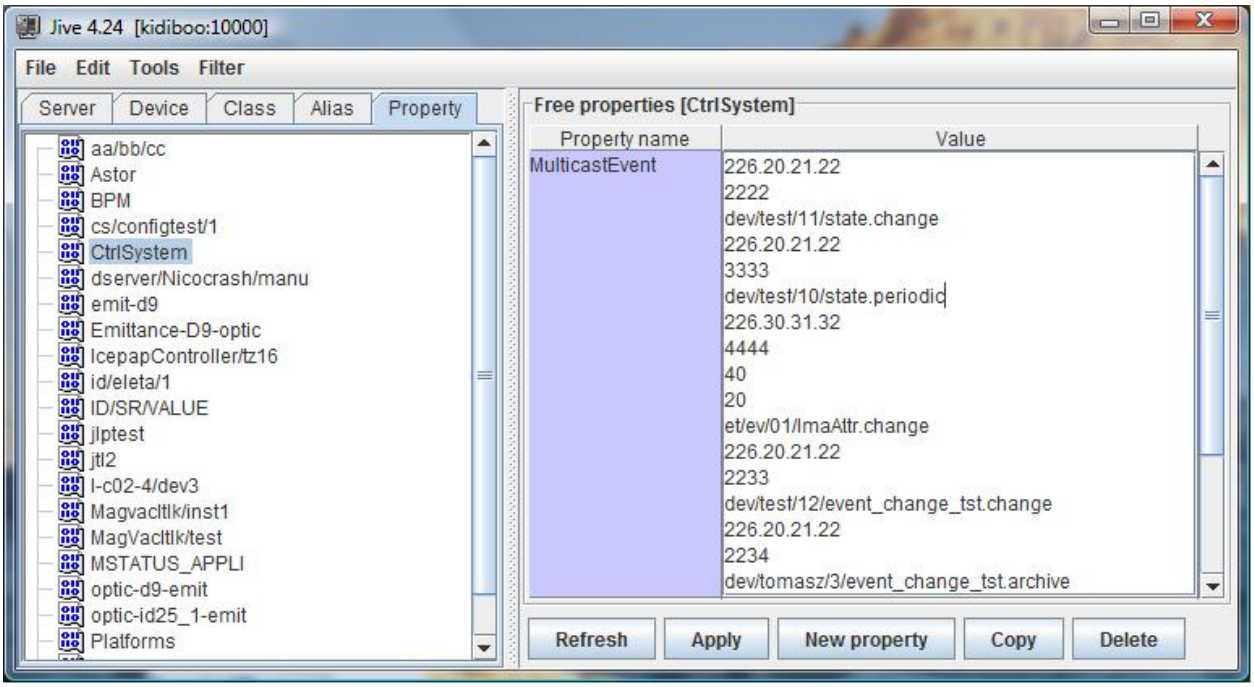
Once the address is selected, you have to choose a port number. Together with the event name, these are the two minimum configuration informations which have to be provided to Tango to get multicast transport. This configuration is done using the **MulticastEvent** free property associated to the **CtrlSystem** object.



In the above window dump of the Jive tool, the *change* event on the *state* attribute of the *dev/test/11* device has to be transferred using multicasting with the address *226.20.21.22* and the port number *2222*. The exact definition of this CtrlSystem/MulticastEvent property for one event propagated using multicast is

```
CtrlSystem->MulticastEvent:   Multicast address,
                              port number,
                              [rate in Mbit/sec],
                              [ivl in seconds],
                              event name
```

Rate and Ivl are optional properties. In case several events have to be transferred using multicasting, simply extend the MulicastEvent property with the configuration parameters related to the other events. There is only one MultiCastEvent property per Tango control system. The underlying multicast protocol (PGM) is rate limited. This means that it limits its network bandwidth usage to a user defined value. The optional third configuration parameter is the maximum rate (in Mbit/sec) that the protocol will use to transfert this event. Because PGM is a reliable protocol, data has to be buffered for re-transmission in case a receiver signal some lost data. The optional forth configuration parameter specify the maximum amount of time (in seconds) that a receiver can be absent for a multicast group before unrecoverable data loss will occur. Exercise care when setting large recovery interval as the data needed for recovery will be held in memory. For example, a 60 seconds (1 minute) recovery interval at a data rate of 1 Gbit/sec requires a 7 GBytes in-memory buffer. Whan any of these two optional parameters are not set, the default value (defined in next sub-chapter) are used. Here is another example of events using multicasting configuration



In this example, there are 5 events which are transmitted using multicasting:

1. Event *change* for attribute *state* on device *dev/test/11* which uses multicasting address *226.20.21.22* and port number *2222*
2. Event *periodic* for attribute *state* on device *dev/test/10* which uses multicasting address *226.20.21.22* and port number *3333*
3. Event *change* for attribute *ImaAttr* on device *et/ev/01* which uses multicasting address *226.30.31.32* and port number *4444*. Note that this event uses a rate set to *40 Mbit/sec* and a ivl set to *20 seconds*.
4. Event *change* for attribute *event_change_tst* on device *dev/test/12* which uses multicasting address *226.20.21.22* and port number *2233*
5. Event *archive* for attribute *event_change_tst* on device *dev/tomasz/3* which uses multicasting address *226.20.21.22* and port number *2234*

Default multicast related properties

On top of the MulticastEvent property previously described, Tango supports three properties to defined default value for multicast transport tuning. These properties are:

- **MulticastRate** associated to the CtrlSystem object. This defines the maximum rate will will be used by the multicast protocol when transferring event. The unit is Mbit/sec. In case this property is not defined, the Tango library used a value of 80 Mbit/sec.
- **MulticastIvl** associated to the CtrlSystem object. It specifies the maximusm time (in sec) during which data has to be buffered for re-transmision in case a receiver signals some lost data. The unit is seconds. In case this property is not defined, the Tango library takes a value of 20 seconds.
- **MulticastHops** associated to the CtrlSystem object. This property defines the maximum number of element (router), the multicast packet is able to cross. Each time one element is crossed, the value is decremented. When it reaches 0, the packet is not transferred any more. In case this property is not defined, the Tango library uses a value of 5.

Events in Tango releases lower than 8

The notifd2db utility

Intended audience: administrators, developers

The notifd2db utility usage (For Tango releases lower than 8)

The notifd2db utility is used to pass to Tango the necessary information for the Tango servers or clients to build connection with the CORBA notification service. Its usage is:

```
notifd2db [notifd2db_IOR_file] [host] [-o Device_server_database_file_name] [-h]
```

The [notifd2db_IOR_file] parameter is used to specify the file name used by the notification service to store its main IOR. This parameter is not mandatoty. Its default value is /tmp/rdfact.ior. The [host] parameter is ued to specify on which host the notification service should be exported. The default value is the host on which the command is run. The [-o Device_server_database_file_name] is used in case of event and device server started with the file as database (the -file device server command line option). The file name used here must be the file name used by the device server in its -file option. The [-h] option is just to display an help message. Notifd2db utility usage example:

```
notifd2db
```

to register notification service on the current host using the default notifictaion service IOR file name.

```
notifd C:\Temp\nd.ior
```

to register a notification service with IOR file named C:\Temp\nd.ior.

```
notifd -o /var/my_ds_file.res
```

to register notification service in the /var/my_ds_file.res file used by a device server started with the device server -file command line option.

Intended audience: administrators, users

8.3.2 The Tango controlled access system

User rights definition

Within the Tango controlled system, you give rights to a user. User is the name of the user used to log-in the computer where the application trying to access a device is running. Two kind of users are defined:

1. Users with defined rights
2. Users without any rights defined in the controlled system. These users will have the rights associated with the pseudo-user called All Users

The controlled system manages two kind of rights:

- Write access meaning that all type of requests are allowed on the device
- Read access meaning that only read-like access are allowed (write_attribute, write_read_attribute and set_attribute_config network calls are forbidden). Executing a command is also forbidden except for commands defined as **Allowed commands**. Getting a device state or status using the command_inout call is always allowed. The definition of the allowed commands is done at the device class level. Therefore, all devices belonging to the same class will have the allowed commands set.

The rights given to a user is the check result splitted in two levels:

1. At the host level: You define from which hosts the user may have write access to the control system by specifying the host name. If the request comes from a host which is not defined, the right will be Read access. If nothing is defined at this level for the user, the rights of the All Users user will be used. It is also possible to specify the

host by its IP address. You can define a host family using wide-card in the IP address (eg. 160.103.11.* meaning any host with IP address starting with 160.103.11). Only IP V4 is supported.

2. At the device level: You define on which device(s) request are allowed using device name. Device family can be used using widecard in device name like domin/family/*

Therefore, the controlled system is doing the following checks when a client try to access a device:

- Get the user name
- Get the host IP address
- If rights defined at host level for this specific user and this IP address, gives user temporary write acccess to the control system
- If nothing is specified for this specific user on this host, gives to the user a temporary access right equal to the host access rights of the All User user.
- If the temporary right given to the user is write access to the control system
  - If something defined at device level for this specific user
    - If there is a right defined for the device to be accessed (or for the device family), give user the defined right
    - Else
      - If rights defined for the All Users user for this device, give this right to the user
      - Else, give user the Read Access for this device
  - Else
    - If there is a right defined for the device to be accessed (or for the device family) for the All User user, give user this right
    - Else, give user the Read Access right for this device
- Else, access right will be Read Access

Then, when the client tries to access the device, the following algorithm is used:

- If right is Read Access
  - If the call is a write type call, refuse the call
  - If the call is a command execution
    - If the command is one of the command defined in the Allowed commands for the device class, send the call
    - Else, refuse the call

All these checks are done during the DeviceProxy instance constructor except those related to the device class allowed commands which are checked during the command_inout call.

To simplify the rights management, give the All Users user host access right to all hosts (.*.*.*) and read access to all devices (/*/*). With such a set-up for this user, each new user without any rights defined in the controlled access will have only Read Access to all devices on the control system but from any hosts. Then, on request, gives Write Access to specific user on specific host (or family) and on specific device (or family).

The rights managements are done using the Tango Astor [ASTOR] tool which has some graphical windows allowing to grant/revoke user rights and to define device class allowed commands set. The following window dump shows this Astor window.

In this example, the user taurel has Write Access to the device sr/d-ct/1 and to all devices belonging to the domain fe but only from the host pcantares He has read access to all other devices but always only from the host pcantares. The user verdier has write access to the device sys/dev/01 from any host on the network 160.103.5 and Read Access to all the remaining devices from the same network. All the other users has only Read Access but from any host.

Running a Tango control system with the controlled access

All the users rights are stored in two tables of the Tango database. A dedicated device server called **TangoAccessControl** access these tables without using the classical Tango database server. This TangoAccessControl device server must be configured with only one device. The property **Services** belonging to the free object **CtrlSystem** is used to run a Tango control system with its controlled access. This property is an array of string with each string describing the service(s) running in the control system. For controlled access, the service name is AccessControl. The service instance name has to be defined as tango. The device name associated with this service must be the name of the TangoAccessControl server device. For instance, if the TangoAccessControl device server device is named *sys/access_control/1*, one element of the Services property of the CtrlSystem object has to be set to

AccessControl/tango:sys/access_control/1

If the service is defined but without a valid device name corresponding to the TangoAccessControl device server, all

users from any host will have write access (simulating a Tango control system without controlled access). Note that this device server connects to the MySQL database and therefore may need the MySQL connection related environment variables MYSQL_USER and MYSQL_PASSWORD described in [sub:Db-Env-Variables]

Even if a controlled access system is running, it is possible to by-pass it if, in the environment of the client application, the environment variable SUPER_TANGO is defined to true. If for one reason or another, the controlled access server is defined but not accessible, the device right checked at that time will be Read Access.

Intended audience: all

8.3.3 HDB++ - an archiving/historian service

Below, details on deployment and configuration of the *HDB++* service are provided. For overview, see *HDB++* of *Tools and Extensions* section.

Intended audience: administrators, developers, ***Programming language:*** C++

HDB++ Design and implementation

Author: *Lorenzo Pivetta*

Below, details on deployment and configuration of the *HDB++* service are provided. For overview, see *HDB++* of *Tools and Extensions* section.

Contents

- *HDB++ Design and implementation*
  - *HDB++ Contributions*
  - *Historical Database*
  - *HDB++ TANGO Device Server*
  - *Configuration and diagnostic tools*
  - *Database interface*
    - *database structure*
  - *Deployment best practices*
  - *General remarks*
  - *Project references and source code*
  - *Event Subscriber full documentation*
  - *Configuration Manager full documentation*

HDB++ Contributions

Summary

The is a novel device server for Historical Data Base (HDB) archiving. It's written in C++ and is fully event-driven.

Contributions

R. Bourtembourg, J.M. Chaize, F.Poncet, J.L. Pons, P.Verdier, D.Lacoste - ESRF
C.Scafuri, G.Scalamera, G.Strangolino, L.Zambon - ELETTRA

Revisions

Date	Rev.	Author	
2012-12-04	1.0	L.Pivetta	First release
2013-01-29	1.1	L.Pivetta	Merged suggestions from ESRF
2013-01-31	1.2	L.Pivetta	Cleanup
2013-05-10	1.3	L.Pivetta	Revision after HDB++ meeting on 2013.03.14
2014-01-30	1.4	L.Pivetta	details + Extraction library
2014-03-07	1.5	L.Pivetta	Database interface
2014-05-05	1.6	L.Pivetta	Cleanup, full ES and CM doc
2014-07-28	1.7	L.Pivetta	Revision after HDB++ meeting on 2014.06.25
2016-08-12	1.8	L.Pivetta	Revision after HDB++ meeting on 2016.05.10

Historical Database

The Historical Database is a tool that allows to store the values of Attributes into a database. The core implements an event-based interface to allow device servers to publish the data to be archived. The **archive** event can be triggered by two mechanisms:

- delta_change: the attribute value changed *significantly*
- periodic: at a fixed periodic interval

The configuration parameters of each attribute, i.e. polling period, delta change thresholds, archiving period, are defined as properties in the database. In addition the archive event can be manually pushed from the device server code.

For additional information concerning the event subsystem please refer to *The Control System Manual* Version 9.2.

HDB++ TANGO Device Server

The architecture is composed by several TANGO devices. More in detail, at least one, but actually many, EventSubscriber TANGO device jointly with one ConfigurationManager device TANGO and one or more DataExtraction TANGO device are foreseen in a TANGO facility.

The EventSubscriber device, also called archiver device, will subscribe to archive events on request by the ConfigurationManager. The EventSubscriber will be able to start archiving all the already configured events even if the ConfigurationManager is not running. The EventSubscriber device must have the following characteristics:

1. the archiving mechanism is event-based, thus the EventSubscriber device tries to subscribe to the event; an error means a fault. A transparent re-subscription to the faulty event is required.

2. one additional thread is in charge of events subscription and call-back execution; the call back, acting as producer, must put the complete data of the received events in a FIFO queue; the thread and the callback must be able to handle an *arbitrary* number of events, possibly limited just by the available memory and/or the required performance; moreover, a high-mark threshold must be setup on the FIFO in order to alert for an overloaded Event Subscriber
3. one additional thread, acting as consumer of the FIFO, is in charge of pushing the data into the database back-end, preserving the event data time-stamp; the code to access the database engine shall be structured to allow the use of different back-ends (Timescaledb, MySQL, Oracle, etc…)
4. the device server methods, commands and attributes, must allow to perform the following operations:
    - start the archiving for all attributes
    - stop the archiving for all attributes
    - start the archiving for one attribute
    - stop the archiving for one attribute
    - read the number of attributes in charge
    - read the list of attributes in charge
    - read the configuration parameters of each attribute
    - read the number of working attributes
    - read the list of working attributes
    - read the number of faulty attributes
    - read the list of faulty attributes with diagnostics
    - read the size of the FIFO queue
    - read the number of attributes pending in the FIFO
    - read the list of attributes pending in the FIFO

The list of attributes in charge of each EventSubscriber is stored in the database as property of the device.

The EventSubscriber device must be able to run and report on the working/faulty attributes/events by means of the standard API (commands and/or attributes) without the need of a graphical interface.

The diagnostics of faults could also be stored in the general info about each attribute; the diagnostics are used by the EventSubscriber device itself to detect that some data is not being stored as requested. Moreover, whenever the archive event period for a given Attribute has been configured, the device server checks that at least the one archive event/period is received; if not, a error is raised and the Attribute marked as faulty (NOK).

Stopping the archiving of an attribute does not persist after a restart, i.e. restarting an EventSubscriber device instance triggers the archiving of *all* configured attributes. A property can be setup not to start archiving at startup.

Note: A new Context based mechanism has been introduced to label and automatically start/stop archiving for groups of attributes depending on the specified archiving strategy. See Context description below.

One NULL value with time stamp is inserted whenever the archiving of an attribute is stopped, due to error or by a specific stop command. Moreover, if an error occurred, the corresponding attribute is marked as faulty in the archiving engine and the error description stored. In case the archiving was suspended due to error, it is automatically resumed when good data is available again. The quality factor of the attribute is also stored into the historical database. One or more alarms can be configured in the Alarm System to asynchronously inform about the status of the archiving devices.

Some of the attribute configuration parameters, such as *display-unit*, *format-string* and *label* will also be available in the archive back-end and updated by means of the attribute configuration change event.

A mechanism to specify per-attribute archiving strategies, called context, has been defined and added to the EventSubscriber. The syntax of the AttributeList Property has been modified to support a *name=value* syntax for the context, except for the Attribute name; fields are separated by semicolon. Keeping the current syntax for the attribute field allows for unchanged backwards compatibility:

```
$ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/eos/climate/18b20 eos.01/State;
↪strategy=RUN|SHUTDOWN
```

The labels for the context, are defined in a free property, and/or in the class property and/or in the device property, with increasing priority. The defaults values, as well as the default context, are pre-defined but can be modified by the user. The default values are shown in table.

label
ALWAYS
RUN
SHUTDOWN
SERVICE

Table 1: Context default labels.

Each context is exclusive, except for the ALWAYS context. An attribute configured to be run in the ALWAYS context will be archived at any time. This is why the ALWAYS context must be defined, it it is not it will be automatically added to the list of context. A new memorized attribute, named **Context**, written by upper layer logic, tells the archiver about the current context status or rather the required context transition. Being a memorized Attribute, the **Context** attribute needs to be written at least once. If not set, a warning will be displayed in the device status, and only the attributes with the ALWAYS context will be archived. This means that, once the device has been deployed, and the AttributeList pupulated with the relevant attributes to be archived, complete with the strategy, the appropriate label has to be written in **Context**.

The device server shall also expose some additional figures of merit such as:

- for each instance, total number of records per time
- for each instance, total number of failures per time
- for each attribute, number of records per time
- for each attribute, number of failures per time
- for each attribute, time stamp of last record

The system can sum these numbers in a counter which can be reset every period to rank each attribute in term of data rate, error rate etc. This allows preventive maintenance and fine tuning, detecting, for instance, when an attribute is too verbose (e.g. variation threshold below the noise level). These statistics are a key element for qualifying the health of the system. All these attributes will be themselves archived to enable a follow-up versus time.

The device server must maintain at least the following operating states:

- **ON**: archiving running, everything works
- **ALARM**: one or more attributes faulty or the FIFO size grows above high-mark threshold
- **FAULT**: all attributes faulty
- **OFF**: archiving stopped

Configuration and diagnostic tools

With all the statistics kept in the device servers and the device server, the diagnostic tool can be straightforward to develop as a simple QTango or ATK GUI. This GUI will also give read access to the configuration data stored as attribute properties in the database to display the attribute polling frequency of the involved device servers, whenever available, and the archive event configuration. The HDB++ Configurator GUI is available for archiving configuration, management and diagnostics. It is written in Java. Refer to the documentation page for any additional information:

HDB++ Configuration GUI documentation Download GUI jar file

Database interface

A C++ API will be developed to address the writing and reading operations on the database and made availabile as a library. This library will provide the *essential* methods for accessing the database. The device servers, library and tools will eventually take advantage of the library. Actually a number of libraries are already available to encapsulate database access:

libhdb++	abstraction layer
libhdb++timescale	table support, Timescaledb
libhdb++mysql	table support, MySQL
libhdb++cassandra	table support, Cassandra
libhdbmysql	legacy HDB table support, MySQL

Table 2: Available database interfacement libraries.

Additional libraries are foreseen to support different database engines, such as Oracle, Postgres or possibly noSQL implementations.

Note: The Cassandra Error: "All connections on all I/O threads are busy" is connected with incorrect name of Data Center. For example, the correct name is "datacenter1" but libhdbpp-cassandra have default value "DC1". To change this value you should add to LibConfiguration property: *local_dc* =datacenter1

---

database structure

The structure of the legacy HDB is based on three tables, (*adt*, *amt*, *apt*) shown in appendix. In addition, one table, named att_xxxxx is created for each attribute or command to be archived. Many of the columns in the lagacy tables are used for storing HDB archiving engine configuration parameters and are no more required.

The new database structure, whose tables have been designed for the archiver, provides just the necessary columns and takes advantage of microsecond resolution support for daytime. Three SQL scripts are provided to create the necessary database structure for MySQL or Cassandra backend:

create_hdb_mysql.sql	legacy HDB MySQL schema
create_hdb++_mysql.sql	MySQL schema
create_hdb_cassandra.sql	Cassandra schema

Table 3: Database setup scripts.

The *att_conf* table associates the attribute name with a unique id and selects the data type; it's worth notice that the *att_name* raw always contains the complete FQDN, e.g. with the hostname and the domainname.

```
mysql> desc att_conf;
+-----------------------+------------------+------+-----+---------+----------------+
| Field                 | Type             | Null | Key | Default | Extra          |
+-----------------------+------------------+------+-----+---------+----------------+
| att_conf_id           | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| att_name              | varchar(255)     | NO   | UNI | NULL    |                |
| att_conf_data_type_id | int(10) unsigned | NO   | MUL | NULL    |                |
| att_ttl               | int(10) unsigned | YES  |     | NULL    |                |
| facility              | varchar(255)     | NO   |     |         |                |
| domain                | varchar(255)     | NO   |     |         |                |
| family                | varchar(255)     | NO   |     |         |                |
| member                | varchar(255)     | NO   |     |         |                |
| name                  | varchar(255)     | NO   |     |         |                |
+-----------------------+------------------+------+-----+---------+----------------+
```

The *att_conf_data_type* table creates an unique ID for each data type.

```
mysql> desc att_conf_data_type;
+-----------------------+------------------+------+-----+---------+----------------+
| Field                 | Type             | Null | Key | Default | Extra          |
+-----------------------+------------------+------+-----+---------+----------------+
| att_conf_data_type_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| data_type             | varchar(255)     | NO   |     | NULL    |                |
| tango_data_type       | tinyint(1)       | NO   |     | NULL    |                |
+-----------------------+------------------+------+-----+---------+----------------+
```

The *att_history* table stores the timestamps relevant for archiving diagnostics together with the *att_history_event*. The copmplete list of supported TANGO data types is shown in table [db:datatypes]. As an example the table *att_scalar_devlong_rw*, for archiving one value, is also shown below. Three timestamp rows are currently supported: the datum timestamp, the receive time timestamp and the database insertion timestamp.

```
mysql> desc att_history;
+-----------------------+------------------+------+-----+---------+-------+
| Field                 | Type             | Null | Key | Default | Extra |
+-----------------------+------------------+------+-----+---------+-------+
| att_conf_id           | int(10) unsigned | NO   | MUL | NULL    |       |
| time                  | datetime(6)      | NO   |     | NULL    |       |
| att_history_event_id  | int(10) unsigned | NO   | MUL | NULL    |       |
+-----------------------+------------------+------+-----+---------+-------+
```

```
mysql> desc att_history_event;
+-----------------------+------------------+------+-----+---------+----------------+
| Field                 | Type             | Null | Key | Default | Extra          |
+-----------------------+------------------+------+-----+---------+----------------+
| att_history_event_id  | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| event                 | varchar(255)     | NO   |     | NULL    |                |
+-----------------------+------------------+------+-----+---------+----------------+
```

```
mysql> desc att_scalar_devlong_rw;
+-------------------+------------------+------+-----+-----------------------------+-------
↪+
| Field             | Type             | Null | Key | Default                     | Extra␣
↪|
```

(continues on next page)

(continued from previous page)

```
+------------------+------------------+------+-----+---------------------------+-------
↪+
| att_conf_id      | int(10) unsigned | NO   | MUL | NULL                      |       ␣
↪|
| data_time        | timestamp(6)     | NO   |     | 0000-00-00 00:00:00.000000 |       ␣
↪|
| recv_time        | timestamp(6)     | NO   |     | 0000-00-00 00:00:00.000000 |       ␣
↪|
| insert_time      | timestamp(6)     | NO   |     | 0000-00-00 00:00:00.000000 |       ␣
↪|
| value_r          | int(11)          | YES  |     | NULL                      |       ␣
↪|
| value_w          | int(11)          | YES  |     | NULL                      |       ␣
↪|
| quality          | tinyint(1)       | YES  |     | NULL                      |       ␣
↪|
| att_error_desc_id | int(10) unsigned | YES  | MUL | NULL                      |       ␣
↪|
+------------------+------------------+------+-----+---------------------------+-------
↪+
```

scalar	vector
att_scalar_devboolean_ro	att_array_devboolean_ro
att_scalar_devboolean_rw	att_array_devboolean_rw
att_scalar_devdouble_ro	att_array_devdouble_ro
att_scalar_devdouble_rw	att_array_devdouble_rw
att_scalar_devencoded_ro	att_array_devencoded_ro
att_scalar_devencoded_rw	att_array_devencoded_rw
att_scalar_devfloat_ro	att_array_devfloat_ro
att_scalar_devfloat_rw	att_array_devfloat_rw
att_scalar_devlong64_ro	att_array_devlong64_ro
att_scalar_devlong64_rw	att_array_devlong64_rw
att_scalar_devlong_ro	att_array_devlong_ro
att_scalar_devlong_rw	att_array_devlong_rw
att_scalar_devshort_ro	att_array_devshort_ro
att_scalar_devshort_rw	att_array_devshort_rw
att_scalar_devstate_ro	att_array_devstate_ro
att_scalar_devstate_rw	att_array_devstate_rw
att_scalar_devstring_ro	att_array_devstring_ro
att_scalar_devstring_rw	att_array_devstring_rw
att_scalar_devuchar_ro	att_array_devuchar_ro
att_scalar_devuchar_rw	att_array_devuchar_rw
att_scalar_devulong64_ro	att_array_devulong64_ro
att_scalar_devulong64_rw	att_array_devulong64_rw
att_scalar_devulong_ro	att_array_devulong_ro
att_scalar_devulong_rw	att_array_devulong_rw
att_scalar_devushort_ro	att_array_devushort_ro
att_scalar_devushort_rw	att_array_devushort_rw
att_scalar_double_ro	att_array_double_ro
att_scalar_double_rw	att_array_double_rw

continues on next page

Table 1 – continued from previous page

scalar	vector
att_scalar_string_ro	att_array_string_ro
att_scalar_string_rw	att_array_string_rw

Table 4: Supported data types.

To support temporary storage of historical data the att_ttl column has to be added to the att_conf table. The att_ttl defines the time-to-live in hours on a per-attribute basis. Deleting expired data is delegated to the SQL backend; the basic machanism foreseen is a SQL script run by cron.

The complete SQL source for all the tables is reported in appendix. The main points can be summarized as:

- microsecond timestamp resolution
- no per-attribute additional tables; the number of tables used is fixed and does not depend on the number of archived attributes
- specific data type support
- temporary storage support

Note: There are some special OS settings to tune for Cassandra to work as expected, in particular, it is recommended to disable the SWAP and to change the resource limits on Linux, as described in this documentation page: Recommended production settings for Linux.

Deployment best practices

To take full advantage of the high performance and scaling capability of the device server some constraints have to be taken into account. Though a single instance of the device server is capable of handling thousands of events per second, the following setup is preferrable:

- setup per-subsystem instances of the device server (homogeneous dedicated archiving)
- possibly separate attributes that have to be archived all the time, e.g. also during maintenance periods, from attributes that are run-centric

A native tool, available to be run locally, as well as a reworked web interface (E-Giga) are foreseen. A specific library with a dedicated API could be developed to address the extraction and the be used into whatever tool may be provided: a device server, a web interface, a native graphical panel, etc. The library shall be able to deal with event based archived data. The possible lack of data inside the requested time window shall be properly managed:

- returning some *no-data-available* error: in this case the reply contains no data and a *no-data-available* error is triggered. Care must be taken whenever the requirement of getting multiple data simultanously is foreseen.
- enlarging the time window itself to include some archived data: the requested time interval is enlarged in order to incorporate some archived data. A mechanism shall be provided to notify the client of the modified data set. No fake samples have to be introduced to fill the values in correspondence of the requested timestamps.
- returning the value of the last archived data anyhow: the requested time interval is kept and the last available data sample is returned. The validity of the data is guaranteed when the archiving mechanism is based on archive event on change; care must be taken when using the data in case of periodic event.

Moreover, whenever extracting multiple rows, the library shall allow to select one of the following behaviours:

- return variable length data arrays for each row
- return equal length data arrays for all rows, filling the gaps with the previous data value

A C++ native implementation, as well as a Java implementation, exposing the same API, are foreseen and are currently available. Please refer to the *hdbextractor* reference manual for the C++ implementation

and the *HDB++ Java Extraction Library* for Java HDB++ java-extraction-api

General remarks

Care must be taken to avoid introducing dependencies from libraries not already needed by the core.

Project references and source code

The HDB++ project page is available on GitHub.

The HDB++ source code for the archiving engine as well as the configuration tools, extraction libraries and GUI are available on

Sourceforge

Event Subscriber full documentation

Please, refer to the `HDB++ Design and implementation .pdf file (page 42)`

Configuration Manager full documentation

Please, refer to the `HDB++ Design and implementation .pdf file (page 72)`

Intended audience: administrators, developers, ***Programming language:*** C++

Event Subscriber interface

Contents

- *Event Subscriber interface*
  - *Commands*
  - *Attributes*
  - *Class properties*
  - *Device properties*

More in detail the device server interface is summarized in table 1 and table 2.

Commands

AttributeAdd	add an attribute to archiving; the complete FQDN has to be specified otherwise it is completed by the using getaddrinfo()
AttributeContext	read the specified attribute current context
AttributePause	pause archiving specified attribute but do not unsubscribe archive event
AttributeRemove	remove an attribute from archiving; the archived data and the attribute archive event configuration are left untouched
AttributeStatus	read attribute status
AttributeStart	start archiving specified attribute
AttributeStop	stop archiving specified attribute, unsubscribe archive event
AttributeUpdate	update context of an already archived attribute
Pause	pause archiving all attributes but do not unsubscribe archive events
Start	start archiving
Stop	stop archiving, usubscribe all archive events
ResetStatistics	reset statistics

Table 1: Event Subscriber Command.

Attributes

AttributeContextList	return the list of attribute contexts
AttributeErrorList	return the list of attribute errors
AttributeEventNumberList	number of events received for each attribute
AttributeFailureFreq	total number of failures per time
AttributeFailureFreqList	per-attribute number of failures per time
AttributeList	return configured attribute list
AttributeMaxPendingNumber	maximum number of attributes waiting to be archived
AttributeMaxProcessingTime	max processing time
AttributeMaxStoreTime	max storing time
AttributeMinProcessingTime	min processing time
AttributeMinStoreTime	min storing time
AttributeNokList	return the list of attribute in error
AttributeNokNumber	number of archived attribute in error
AttributeNumber	number of attributes configured for archiving
AttributeOkList	return the list of attributes not in error
AttributeOkNumber	number of archived attributes not in error
AttributePausedList	list of paused attributes
AttributePausedNumber	number of paused attributes
AttributePendingList	list of attributes waiting to be archived
AttributePendingNumber	number of attributes waiting to be archived
AttributeRecordFreq	total number of records per time
AttributeRecordFreqList	per-attribute number of records per time
AttributeStartedList	list of started attributes
AttributeStartedNumber	number of started attributes
AttributeStoppedList	list of stopped attributes
AttributeStoppedNumber	number of stopped attributes
Context	archiver current context (r/w)
StatisticsResetTime	seconds elapsed since last statistics reset

Table 2: Event Subscriber Attributes.

The class and device properties availabile for configuration are shown in table 3. According to TANGO device server design guidelines, Device Properties, when defined, override Class properties. Please note that class and device Properties have changed since release of the TANGO device server.

Class properties

CheckPeriodicTimeoutDelay	delay before timeout when checking periodic events, in seconds
PollingThreadPeriod	default period for polling thread, in seconds
LibConfiguration	configuration parameters for backend support library
ContextsList	definition of possible archiver operating contexts
DefaultStrategy	default context for a new attribute (Default to RUN)
StatisticsTimeWindow	timeslot for statistics in seconds
SubscribeRetryPeriod	retry period for subscribe event, in seconds

Table 3: Event Subscriber Class properties.

The **LibConfiguration** property contains the following multi-line configuration parameters *host*, *user*, *password*, *dbname*, *libname*, *port*. Table shows example configuration parameters for backend:

host=srv-log-srf.fcs.elettra.trieste.it
user=hdbarchiver
password=myownpassword
dbname=hdbpp
libname=dependOnDatabase (see below)
port=3306

Table 4: LibConfiguration parameters for database.

Note: The event subscriber can be built directly against a specific backend or the dynamic libhdbpp. In case of direct linking, this information is not relevant.

libname should be set to one of the following values:

libname=libhdb++mysql.so if you intend to use HDB++ with the MySQL backend libname=libhdbmysql.so if you intend to use HDB++ with the MySQL Legacy backend libname=libhdb++cassandra.so if you intend to use HDB++ with the Cassandra backend libname=libhdb++timescale.so if you intend to use HDB++ with the TimescaleDB backend

The library specified in LibConfiguration->libname is loaded dynamically by the EventSubscriber device (e.g. *hdb++-es-srv*). You need to have your LD_LIBRARY_PATH environment variable correctly set (including the directory where the library you intend to use is located).

libhdb++mysql and libhdb++cassandra are just implementations of the classes defined in libhdb++ library. The user can decide which implementation to use by specifying this LibConfiguration -> libname device property config parameter.

The device dynamically loads the configured library configured (using dlopen()) during the device initialization. See *Database interface* section for more information.

Device properties

AttributeList	list of configured attributes
CheckPeriodicTimeoutDelay	delay before timeout when checking periodic events, in seconds
PollingThreadPeriod	default period for polling thread, in seconds
LibConfiguration	configuration parameters for backend support library
ContextsList	definition of possible archiver operating contexts
DefaultStrategy	default context for a new attribute (Default to RUN)
StatisticsTimeWindow	timeslot for statistics
SubscribeRetryPeriod	retry period for subscribe event, in seconds

Table 5: Event Subscriber Device properties.

In addition to the already described Class properties, device Properties comprehend the AttributeList property which contains the list of attributes in charge of the current device. The syntax is *fully-qualified-attribute-name;context=CONTEXT* where *CONTEXT* can be one or a combination of the defined contexts (logic OR). Whenever not specified the DefaultContext specified in the Class property or in the Device Property applies. Table shows some examples:

```
1 $ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/eos/climate/18b20 eos.01/State;
  ↪context=RUN|SHUTDOWN
2 $ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/eos/climate/18b20 eos.01/
  ↪Temperature;context=RUN|SHUTDOWN
```

(continues on next page)

(continued from previous page)

```
$ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/ctf/diagnostics/ccd_ctf.01/State;context=RUN
$ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/ctf/diagnostics/ccd_ctf.01/HorProfile;context=RUN
$ tango://srv-tango-srf.fcs.elettra.trieste.it:20000/ctf/diagnostics/ccd_ctf.01/VerProfile;context=RUN
```

Table 6: AttributeList example

The first two attributes will be archived in both RUN and SHUTDOWN contexts; the last three only when in RUN.

Intended audience: administrators, developers, ***Programming language:*** C++

Configuration Manager interface

Contents

- *Configuration Manager interface*
  - *Commands*
  - *Attributes*
  - *Class properties*
  - *Device properties*

In order to address large archiving systems the need to distribute the workload over a large number of event subscriber shows up. The configuration manager device server will assist in the operations of adding, editing, moving, deleting an attribute from the archiving system. All the configuration parameters, such as polling period, variation thresholds etc., are kept in the database as properties of the archived attribute. In order to be managed by the device server each instance has to added to the pool using the ArchiverAdd command.

The configuration manager shall be able to perform the following operations on the managed pool:

1. manage the request of archiving a new attribute
   - setup the attribute's archive event configuration
   - assign the new attribute to one of the device servers
     - following some rules of load balancing
     - to the specified device server
2. move an attribute from a device server to another one
3. keep trace of which attribute is assigned to which
4. start/stop the archiving of an attribute at runtime
5. remove an attribute from archiving

The configuration shall be possible via the device server API as well as via a dedicated GUI interface; the GUI just use the provided API.

The ConfigurationManager may also expose a certain number of attributes to give the status of what is going on:

- total number of Archivers

- total number of working attributes
- total number of faulty attributes
- total number of calls per second

These attributes could be themselves archived to enable a follow up versus time.

More in detail the device server exposes the following interface.

Commands

The available commands are summarized in commands-table.

ArchiverAdd	add a new archiver instance to the archivers list; the instance must have been already created and configured via jive/astor and the device shall be running
ArchiverRemove	remove an archiver from the list; neither the device instance nor the attributes configured are removed from the database
AttributeAdd	add an attribute to archiving
AttributeAssign	assign attribute to an archiver
AttributeGetArchiver	return the archiver in charge of attribute
AttributePause	pause archiving specified attribute
AttributeRemove	remove an attribute from archiving; the archived data and the attribute archive event configuration are left untouched
AttributeSearch	return list of attributes containing input pattern
AttributeStart	start archiving an attribute
AttributeStatus	read attribute archiving status
AttributeStop	stop archiving an attribute
AttributeUpdate	update context of an already archived attribute
Context	set context to all managed archivers
ResetStatistics	reset statistics the configuration manager of and all archivers

Table 1: Configuration Manager Commands.

Note that the list of managed archivers is stored into the ArchiverList device property that is maintained via the ArchiverAdd, ArchiverRemove and AttributeSetArchiver commands. Therefore in the archiving system the device server instances can also be configured by hand, if required, an run independently.

Attributes

The attributes of the configuration manager are summarized in attributes-table.

ArchiverContext	return archiver context
ArchiverList	return list of managed archivers
ArchiverStatisticsResetTime	seconds elapsed since last statistics reset
ArchiverStatus	return archiver status information
AttributeFailureFreq	total number of failures per time
AttributeMaxPendingNumber	max number of attributes waiting to be archived (all archivers)
AttributeMaxProcessingTime	max processing time (all archivers)
AttributeMaxStoreTime	max storing time (all archivers)
AttributeMinProcessingTime	min processing time (all archivers)
AttributeMinStoreTime	min storing time (all archivers)
AttributeNokNumber	total number of archived attribute in error
AttributeNumber	total number of attributes configured for archiving
AttributeOkNumber	total number of archived attribute not in error
AttributePausedNumber	total number of paused attributes
AttributePendingNumber	total number of attributes waiting to be archived
AttributeRecordFreq	total number of records per time
AttributeStartedNumber	total number of started attributes
AttributeStoppedNumber	total number of stopped attributes
SetAbsoluteEvent	set archive absolute thresholds; for archiving setup
SetArchiver	support attribute for setup
SetAttributeName	support attribute for setup
SetCodePushedEvent	specify event pushed in the code
SetContext	set archiving context; for archiving setup
SetPeriodEvent	set archive period; for archiving setup
SetPollingPeriod	set polling period; for archiving setup
SetRelativeEvent	set archive relative thresholds; for archiving setup
SetTTL	set time-to-live for temporary storage; for archiving setup

Table 2: Configuration Manager Attributes.

The SetXxxYyy attributes are used for archive event and archiver instance configuration setup and must be filled before calling the AttributeAdd command. The AttributeAdd checks the consistency of the desired event configuration and then adds the new attribute to the archiver instance specified with SetArchiver. Then the AttributeAdd command creates the required entries into the historical database.

Class properties

LibConfiguration	configuration parameters for backend support library
MaxSearchSize	max size for AttributeSearch result

Table 3: Event Subscriber Class properties.

Device properties

ArchiverList	list of existing archivers
LibConfiguration	configuration parameters for backend support library
MaxSearchSize	max size for AttributeSearch result

Table 4: Configuration Manager device properties.

Intended audience: administrators, developers, ***Programming language:*** SQL

Legacy HDB tables structure

Contents

- *Legacy HDB tables structure*


```
mysql> describe adt;
+-------------+--------------------------------+------+-----+---------+----------------+
| Field       | Type                           | Null | Key | Default | Extra          |
+-------------+--------------------------------+------+-----+---------+----------------+
| ID          | smallint(5) unsigned zerofill  | NO   | PRI | NULL    | auto_increment |
| time        | datetime                       | YES  |     | NULL    |                |
| full_name   | varchar(200)                   | NO   | PRI |         |                |
| device      | varchar(150)                   | NO   |     |         |                |
| domain      | varchar(35)                    | NO   |     |         |                |
| family      | varchar(35)                    | NO   |     |         |                |
| member      | varchar(35)                    | NO   |     |         |                |
| att_name    | varchar(50)                    | NO   |     |         |                |
| data_type   | tinyint(1)                     | NO   |     | 0       |                |
| data_format | tinyint(1)                     | NO   |     | 0       |                |
| writable    | tinyint(1)                     | NO   |     | 0       |                |
| max_dim_x   | smallint(6) unsigned           | NO   |     | 0       |                |
| max_dim_y   | smallint(6) unsigned           | NO   |     | 0       |                |
| levelg      | tinyint(1)                     | NO   |     | 0       |                |
| facility    | varchar(45)                    | NO   |     |         |                |
| archivable  | tinyint(1)                     | NO   |     | 0       |                |
| substitute  | smallint(9)                    | NO   |     | 0       |                |
+-------------+--------------------------------+------+-----+---------+----------------+
```

```
mysql> describe amt;
+-----------------+--------------------------------+------+-----+---------+-------+
| Field           | Type                           | Null | Key | Default | Extra |
+-----------------+--------------------------------+------+-----+---------+-------+
| ID              | smallint(5) unsigned zerofill  | NO   |     | 00000   |       |
| archiver        | varchar(255)                   | NO   |     |         |       |
| start_date      | datetime                       | YES  |     | NULL    |       |
| stop_date       | datetime                       | YES  |     | NULL    |       |
| per_mod         | int(1)                         | NO   |     | 0       |       |
| per_per_mod     | int(5)                         | YES  |     | NULL    |       |
| abs_mod         | int(1)                         | NO   |     | 0       |       |
```

(continues on next page)

(continued from previous page)

```
| per_abs_mod       | int(5)                          | YES  |     | NULL    |       |
| dec_del_abs_mod   | double                          | YES  |     | NULL    |       |
| gro_del_abs_mod   | double                          | YES  |     | NULL    |       |
| rel_mod           | int(1)                          | NO   |     | 0       |       |
| per_rel_mod       | int(5)                          | YES  |     | NULL    |       |
| n_percent_rel_mod | double                          | YES  |     | NULL    |       |
| p_percent_rel_mod | double                          | YES  |     | NULL    |       |
| thr_mod           | int(1)                          | NO   |     | 0       |       |
| per_thr_mod       | int(5)                          | YES  |     | NULL    |       |
| min_val_thr_mod   | double                          | YES  |     | NULL    |       |
| max_val_thr_mod   | double                          | YES  |     | NULL    |       |
| cal_mod           | int(1)                          | NO   |     | 0       |       |
| per_cal_mod       | int(5)                          | YES  |     | NULL    |       |
| val_cal_mod       | int(3)                          | YES  |     | NULL    |       |
| type_cal_mod      | int(2)                          | YES  |     | NULL    |       |
| algo_cal_mod      | varchar(20)                     | YES  |     | NULL    |       |
| dif_mod           | int(1)                          | NO   |     | 0       |       |
| per_dif_mod       | int(5)                          | YES  |     | NULL    |       |
| ext_mod           | int(1)                          | NO   |     | 0       |       |
| refresh_mode      | tinyint(4)                      | YES  |     | 0       |       |
+-------------------+---------------------------------+------+-----+---------+-------+
```

```
mysql> describe apt;
+---------------+--------------------------+------+-----+---------+-------+
| Field         | Type                     | Null | Key | Default | Extra |
+---------------+--------------------------+------+-----+---------+-------+
| ID            | int(5) unsigned zerofill | NO   | PRI | 00000   |       |
| time          | datetime                 | YES  |     | NULL    |       |
| description   | varchar(255)             | NO   |     |         |       |
| label         | varchar(64)              | NO   |     |         |       |
| unit          | varchar(64)              | NO   |     | 1       |       |
| standard_unit | varchar(64)              | NO   |     | 1       |       |
| display_unit  | varchar(64)              | NO   |     |         |       |
| format        | varchar(64)              | NO   |     |         |       |
| min_value     | varchar(64)              | NO   |     | 0       |       |
| max_value     | varchar(64)              | NO   |     | 0       |       |
| min_alarm     | varchar(64)              | NO   |     | 0       |       |
| max_alarm     | varchar(64)              | NO   |     | 0       |       |
+---------------+--------------------------+------+-----+---------+-------+
```

Intended audience: administrators, developers, ***Programming language:*** SQL

schema SQL source (TimescaleDb)

Contents

- *schema SQL source (TimescaleDb)*

All the sql resources for TimescaleDb can be found on the github project https://github.com/tango-controls-hdbpp/hdbpp-timescale-project. Under resource/schema. On top of the general architecture you can find extensions schemas for reordering, aggregates and any other features.

Intended audience: administrators, developers, ***Programming language:*** SQL

schema SQL source (MySQL)

Contents

- *schema SQL source (MySQL)*


```
CREATE TABLE IF NOT EXISTS att_conf
(
att_conf_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
att_name VARCHAR(255) UNIQUE NOT NULL,
att_conf_data_type_id INT UNSIGNED NOT NULL,
att_ttl INT UNSIGNED NULL DEFAULT NULL,
facility VARCHAR(255) NOT NULL DEFAULT '',
domain VARCHAR(255) NOT NULL DEFAULT '',
family VARCHAR(255) NOT NULL DEFAULT '',
member VARCHAR(255) NOT NULL DEFAULT '',
name VARCHAR(255) NOT NULL DEFAULT '',
INDEX(att_conf_data_type_id)
) ENGINE=MyISAM COMMENT='Attribute Configuration Table';

DROP TABLE att_conf_data_type;
CREATE TABLE IF NOT EXISTS att_conf_data_type
(
att_conf_data_type_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
data_type VARCHAR(255) NOT NULL,
tango_data_type TINYINT(1) NOT NULL
) ENGINE=MyISAM COMMENT='Attribute types description';

INSERT INTO att_conf_data_type (data_type, tango_data_type) VALUES
('scalar_devboolean_ro', 1),('scalar_devboolean_rw', 1),('array_devboolean_ro', 1),
('array_devboolean_rw', 1),('scalar_devuchar_ro', 22),('scalar_devuchar_rw', 22),
('array_devuchar_ro', 22),('array_devuchar_rw', 22),('scalar_devshort_ro', 2),
('scalar_devshort_rw', 2),('array_devshort_ro', 2),('array_devshort_rw', 2),
('scalar_devushort_ro', 6),('scalar_devushort_rw', 6),('array_devushort_ro', 6),
('array_devushort_rw', 6),('scalar_devlong_ro', 3),('scalar_devlong_rw', 3),
('array_devlong_ro', 3),('array_devlong_rw', 3),('scalar_devulong_ro', 7),
('scalar_devulong_rw', 7),('array_devulong_ro', 7),('array_devulong_rw', 7),
('scalar_devlong64_ro', 23),('scalar_devlong64_rw', 23),('array_devlong64_ro', 23),
```

(continues on next page)

(continued from previous page)

```
('array_devlong64_rw', 23),('scalar_devulong64_ro', 24),('scalar_devulong64_rw', 24),
('array_devulong64_ro', 24),('array_devulong64_rw', 24),('scalar_devfloat_ro', 4),
('scalar_devfloat_rw', 4),('array_devfloat_ro', 4),('array_devfloat_rw', 4),
('scalar_devdouble_ro', 5),('scalar_devdouble_rw', 5),('array_devdouble_ro', 5),
('array_devdouble_rw', 5),('scalar_devstring_ro', 8),('scalar_devstring_rw', 8),
('array_devstring_ro', 8),('array_devstring_rw', 8),('scalar_devstate_ro', 19),
('scalar_devstate_rw', 19),('array_devstate_ro', 19),('array_devstate_rw', 19),
('scalar_devencoded_ro', 28),('scalar_devencoded_rw', 28),('array_devencoded_ro', 28),
('array_devencoded_rw', 28);

CREATE TABLE IF NOT EXISTS att_history
(
att_conf_id INT UNSIGNED NOT NULL,
time TIMESTAMP(6) DEFAULT 0,
att_history_event_id INT UNSIGNED NOT NULL,
INDEX(att_conf_id),
INDEX(att_history_event_id)
) ENGINE=MyISAM COMMENT='Attribute Configuration Events History Table';

DROP TABLE att_history_event;
CREATE TABLE IF NOT EXISTS att_history_event
(
att_history_event_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
event VARCHAR(255) NOT NULL
) ENGINE=MyISAM COMMENT='Attribute history events description';

INSERT INTO att_history_event (event) VALUES
('add'),('remove'),('start'),('stop'),('crash'),('pause');

CREATE TABLE IF NOT EXISTS att_parameter
(
att_conf_id INT UNSIGNED NOT NULL,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
label VARCHAR(255) NOT NULL DEFAULT '',
unit VARCHAR(64) NOT NULL DEFAULT '',
standard_unit VARCHAR(64) NOT NULL DEFAULT '1',
display_unit VARCHAR(64) NOT NULL DEFAULT '',
format VARCHAR(64) NOT NULL DEFAULT '',
archive_rel_change VARCHAR(64) NOT NULL DEFAULT '',
archive_abs_change VARCHAR(64) NOT NULL DEFAULT '',
archive_period VARCHAR(64) NOT NULL DEFAULT '',
description VARCHAR(1024) NOT NULL DEFAULT '',
INDEX(recv_time),
INDEX(att_conf_id)
) ENGINE=MyISAM COMMENT='Attribute configuration parameters';

CREATE TABLE IF NOT EXISTS att_error_desc
(
att_error_desc_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
error_desc VARCHAR(255) UNIQUE NOT NULL
) ENGINE=MyISAM COMMENT='Error Description Table';
```

(continues on next page)

(continued from previous page)

```
CREATE TABLE IF NOT EXISTS att_scalar_devboolean_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT(1) UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Boolean ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devboolean_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT(1) UNSIGNED DEFAULT NULL,
value_w TINYINT(1) UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Boolean ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devboolean_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT(1) UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Boolean ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devboolean_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT(1) UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
```

(continues on next page)

(continued from previous page)

```
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w TINYINT(1) UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Boolean ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devuchar_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar UChar ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devuchar_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
value_w TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar UChar ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devuchar_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array UChar ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devuchar_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
```

(continues on next page)

(continued from previous page)

```
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array UChar ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devshort_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r SMALLINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Short ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devshort_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r SMALLINT DEFAULT NULL,
value_w SMALLINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Short ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devshort_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r SMALLINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Short ReadOnly Values Table';
```

(continues on next page)

(continued from previous page)

```
CREATE TABLE IF NOT EXISTS att_array_devshort_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r SMALLINT DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w SMALLINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Short ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devushort_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r SMALLINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar UShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devushort_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r SMALLINT UNSIGNED DEFAULT NULL,
value_w SMALLINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar UShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devushort_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
```

(continues on next page)

(continued from previous page)

```
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r SMALLINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array UShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devushort_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r SMALLINT UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w SMALLINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array UShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r INT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Long ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r INT DEFAULT NULL,
value_w INT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Long ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong_ro
(
```

(continues on next page)

(continued from previous page)

```
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r INT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Long ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r INT DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w INT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Long ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong_ro
(
att_conf_id INT UNSIGNED NOT NULL,
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r INT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar ULong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r INT UNSIGNED DEFAULT NULL,
value_w INT UNSIGNED DEFAULT NULL,
```

(continues on next page)

(continued from previous page)

```
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar ULong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r INT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array ULong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r INT UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w INT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array ULong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong64_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BIGINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Long64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong64_rw
(
```

(continues on next page)

(continued from previous page)

```
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BIGINT DEFAULT NULL,
value_w BIGINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Long64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong64_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BIGINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Long64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong64_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BIGINT DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w BIGINT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Long64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong64_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BIGINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
```

(continues on next page)

(continued from previous page)

```
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar ULong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong64_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BIGINT UNSIGNED DEFAULT NULL,
value_w BIGINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar ULong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong64_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BIGINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array ULong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong64_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BIGINT UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w BIGINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array ULong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devfloat_ro
(
```

(continues on next page)

(continued from previous page)

```
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r FLOAT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Float ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devfloat_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r FLOAT DEFAULT NULL,
value_w FLOAT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Float ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devfloat_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r FLOAT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Float ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devfloat_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r FLOAT DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w FLOAT DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
```

(continues on next page)

(continued from previous page)

```
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Float ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devdouble_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r DOUBLE DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Double ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devdouble_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r DOUBLE DEFAULT NULL,
value_w DOUBLE DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Double ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devdouble_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r DOUBLE DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Double ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devdouble_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
```

(continues on next page)

(continued from previous page)

```
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r DOUBLE DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w DOUBLE DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Double ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstring_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r VARCHAR(16384) DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar String ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstring_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r VARCHAR(16384) DEFAULT NULL,
value_w VARCHAR(16384) DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar String ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstring_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r VARCHAR(16384) DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array String ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstring_rw
(
```

(continues on next page)

(continued from previous page)

```
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r VARCHAR(16384) DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w VARCHAR(16384) DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array String ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstate_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar State ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstate_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
value_w TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar State ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstate_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
```

(continues on next page)

(continued from previous page)

```
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array State ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstate_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r TINYINT UNSIGNED DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w TINYINT UNSIGNED DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array State ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devencoded_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BLOB DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Encoded ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devencoded_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
value_r BLOB DEFAULT NULL,
value_w BLOB DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Scalar Encoded ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devencoded_ro
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
```

(continues on next page)

(continued from previous page)

```
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BLOB DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Encoded ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devencoded_rw
(
att_conf_id INT UNSIGNED NOT NULL,
data_time TIMESTAMP(6) DEFAULT 0,
recv_time TIMESTAMP(6) DEFAULT 0,
insert_time TIMESTAMP(6) DEFAULT 0,
idx INT UNSIGNED NOT NULL,
dim_x_r INT UNSIGNED NOT NULL,
dim_y_r INT UNSIGNED NOT NULL DEFAULT 0,
value_r BLOB DEFAULT NULL,
dim_x_w INT UNSIGNED NOT NULL,
dim_y_w INT UNSIGNED NOT NULL DEFAULT 0,
value_w BLOB DEFAULT NULL,
quality TINYINT(1) DEFAULT NULL,
att_error_desc_id INT UNSIGNED NULL DEFAULT NULL,
INDEX att_conf_id_data_time (att_conf_id,data_time)
) ENGINE=MyISAM COMMENT='Array Encoded ReadWrite Values Table';
```

Intended audience: administrators, developers, ***Programming language:*** CQL

schema CQL source (Cassandra)

Contents

- *schema CQL source (Cassandra)*


```
-- Create hdb keyspace
-- Please adapt the replication factor (3 by default here) to your use case
CREATE KEYSPACE IF NOT EXISTS hdb WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy
↪', 'DC1' : 3 };

USE hdb;

CREATE TYPE IF NOT EXISTS devencoded (
 encoded_format text,
 encoded_data blob
);

CREATE TABLE IF NOT EXISTS att_conf (
```

(continues on next page)

(continued from previous page)

```
cs_name text,
att_name text,
att_conf_id timeuuid,
data_type text,   -- data_types set<text> in the future?
PRIMARY KEY (cs_name, att_name)
)
WITH comment='Attribute Configuration Table'
AND caching = {'keys' : 'NONE', 'rows_per_partition': 'ALL' };

CREATE INDEX on att_conf(data_type);
CREATE INDEX on att_conf(att_conf_id);

CREATE TABLE IF NOT EXISTS att_history
(
att_conf_id timeuuid,
time timestamp,
time_us int,
event text, -- 'add','remove','start','stop' or 'crash'
PRIMARY KEY(att_conf_id, time, time_us)
)
WITH comment='Attribute Configuration Events History Table';


CREATE TABLE IF NOT EXISTS att_scalar_devboolean_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r boolean,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevBoolean ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devboolean_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r boolean,
value_w boolean,
quality int,
error_desc text,
```

(continues on next page)

(continued from previous page)

```
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevBoolean ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devuchar_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
) WITH comment='Scalar DevUChar ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devuchar_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
value_w int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevUChar ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devshort_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
) WITH comment='Scalar DevShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devshort_rw (
```

(continues on next page)

(continued from previous page)

```
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
value_w int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devushort_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevUShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devushort_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
value_w int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevUShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong_ro (
att_conf_id timeuuid,
period text,
```

(continues on next page)

(continued from previous page)

```
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevLong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
value_w int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevLong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r bigint,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevULong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
```

(continues on next page)

(continued from previous page)

```
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r bigint,
value_w bigint,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevULong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong64_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r bigint,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevLong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devlong64_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r bigint,
value_w bigint,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevLong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong64_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
```

(continues on next page)

(continued from previous page)

```
insert_time_us int,
value_r bigint,              // issue here with very big numbers
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevULong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devulong64_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r bigint, // issue here with very big numbers
value_w bigint, // issue here with very big numbers
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevLong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devfloat_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r float,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevFloat ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devfloat_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r float,
value_w float,
```

(continues on next page)

(continued from previous page)

```
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevFloat ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devdouble_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r double,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevDouble ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devdouble_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r double,
value_w double,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevDouble ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstring_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r text,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
```

(continues on next page)

(continued from previous page)

```
)
WITH comment='Scalar DevString ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstring_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r text,
value_w text,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevString ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstate_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevState ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devstate_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r int,
value_w int,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevState ReadWrite Values Table';
```

(continues on next page)

(continued from previous page)

```
CREATE TABLE IF NOT EXISTS att_scalar_devencoded_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r frozen<devencoded>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevEncoded ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_scalar_devencoded_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r frozen<devencoded>,
value_w frozen<devencoded>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Scalar DevEncoded ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devboolean_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<boolean>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevBoolean ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devboolean_rw (
att_conf_id timeuuid,
```

(continues on next page)

(continued from previous page)

```
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<boolean>,
value_w list<boolean>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevBoolean ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devuchar_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevUChar ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devuchar_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
value_w list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevUChar ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devshort_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
```

(continues on next page)

(continued from previous page)

```
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devshort_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
value_w list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devushort_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevUShort ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devushort_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
```

(continues on next page)

(continued from previous page)

```
insert_time timestamp,
insert_time_us int,
value_r list<int>,
value_w list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevUShort ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevLong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,
value_w list<int>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevLong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
```

(continues on next page)

(continued from previous page)

```
value_r list<bigint>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevULong ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<bigint>,
value_w list<bigint>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevULong ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong64_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<bigint>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevLong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devlong64_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<bigint>,
value_w list<bigint>,
quality int,
```

(continues on next page)

(continued from previous page)

```
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevLong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong64_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<bigint>,  // issue with very big numbers
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevULong64 ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devulong64_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<bigint>, // issue with very big numbers
value_w list<bigint>, // issue with very big numbers
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevULong64 ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devfloat_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<float>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
```

(continues on next page)

(continued from previous page)

```
WITH comment='Array DevFloat ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devfloat_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<float>,
value_w list<float>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevFloat ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devdouble_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<double>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevDouble ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devdouble_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<double>,
value_w list<double>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevDouble ReadWrite Values Table';
```

(continues on next page)

(continued from previous page)

```
CREATE TABLE IF NOT EXISTS att_array_devstring_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<text>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevString ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstring_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<text>,
value_w list<text>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevString ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstate_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>, // Store a special type here
                   // where we could store an int and a string?
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevState ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devstate_rw (
att_conf_id timeuuid,
```

(continues on next page)

(continued from previous page)

```
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<int>,// Store a special type here
value_w list<int>,// where we could store an int and a string?
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevState ReadWrite Values Table';

CREATE TABLE IF NOT EXISTS att_array_devencoded_ro (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<frozen<devencoded>>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevEncoded ReadOnly Values Table';

CREATE TABLE IF NOT EXISTS att_array_devencoded_rw (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
value_r list<frozen<devencoded>>,
value_w list<frozen<devencoded>>,
quality int,
error_desc text,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Array DevEncoded ReadWrite Values Table';

-------------------------
-- att_parameter table --
-------------------------
CREATE TABLE IF NOT EXISTS att_parameter (
```

(continues on next page)

(continued from previous page)

```
att_conf_id timeuuid,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
label text,
unit text,
standard_unit text,
display_unit text,
format text,
archive_rel_change text,
archive_abs_change text,
archive_period text,
description text,
PRIMARY KEY((att_conf_id), recv_time, recv_time_us)
) WITH COMMENT='Attribute configuration parameters';

--------------------------------
-- Attributes browsing tables --
--------------------------------
CREATE TABLE IF NOT EXISTS domains (
cs_name text,
domain text,
PRIMARY KEY (cs_name, domain)
)
WITH CLUSTERING ORDER BY (domain ASC)
AND comment='Domains Table'
AND caching = {'keys' : 'NONE', 'rows_per_partition': 'ALL' };

CREATE TABLE IF NOT EXISTS families (
cs_name text,
domain text,
family text,
PRIMARY KEY ((cs_name, domain),family)
)
WITH CLUSTERING ORDER BY (family ASC)
AND comment='Families Table'
AND caching = {'keys' : 'NONE', 'rows_per_partition': 'ALL' };

CREATE TABLE IF NOT EXISTS members (
cs_name text,
domain text,
family text,
member text,
PRIMARY KEY ((cs_name, domain,family),member)
)
WITH CLUSTERING ORDER BY (member ASC)
AND comment='Members Table'
AND caching = {'keys' : 'NONE', 'rows_per_partition': 'ALL' };

CREATE TABLE IF NOT EXISTS att_names (
cs_name text,
```

(continues on next page)

(continued from previous page)

```
domain text,
family text,
member text,
name text,
PRIMARY KEY ((cs_name, domain,family,member),name)
)
WITH CLUSTERING ORDER BY (name ASC)
AND comment='Attributes Names Table'
AND caching = {'keys' : 'NONE', 'rows_per_partition': 'ALL' };

CREATE TABLE IF NOT EXISTS time_stats (
att_conf_id timeuuid,
period text,
data_time timestamp,
data_time_us int,
recv_time timestamp,
recv_time_us int,
insert_time timestamp,
insert_time_us int,
PRIMARY KEY ((att_conf_id ,period),data_time,data_time_us)
)
WITH comment='Time Statistics Table';
```

Intended audience: administrators

8.4 Maintenance

8.4.1 Tango Admin utility

Intended audience: administrators

The Tango Database can be maintained using a command-line interface with **tango_admin** tool. The following features are available:

- ping the database server
- check if a device is defined in DB
- check if a server is defined in DB
- create a server in DB
- delete a server from the DB
- create a property in DB
- delete a property from DB

Using tango_admin

```
$ /usr/lib/tango/tango_admin --help

Usage:
--help              Prints this help
--ping-database     [max_time (s)] Ping database
--check-device <dev>    Check if the device is defined in DB
--add-server <exec/inst> <class> <dev list (comma separated)>   Add a server in DB
--delete-server <exec/inst> [--with-properties]   Delete a server from DB
--check-server <exec/inst>   Check if a device server is defined in DB
--server-list  Display list of server names
--server-instance-list <exec>   Display list of server instances for the given server␣
↪name
--add-property <dev> <prop_name> <prop_value (comma separated for array)>    Add a␣
↪device property in DB
--delete-property <dev> <prop_name>   Delete a device property from DB
--tac-enabled Check if the TAC (Tango Access Control) is enabled
--ping-device <dev> [max_time (s)] Check if the device is running
--ping-network [max_time (s)] [-v] Ping network
```

CHAPTER

NINE

TUTORIALS AND HOW-TOS

Intended audience: all

9.1 Tutorials

Intended audience: all, developers, administrators

9.1.1 Documentation, how to contribute

Intended audience: developers

Prerequisites

To work with documentation, first you need to have the following programs installed on your system:

- **`Python`** >= 2.7 (as Sphinx is a Python tool),
- **`Git`** (since the sources are kept in a git repository).

Sphinx installation

Once installed the prerequisites, you can use Sphinx through two options:

- using Python3 venv way, or
- using virtualenv way.

Using Python3 venv way

If you have prerequisites installed, you need to install Sphinx tools:

1. Install virtual environments support for Python:

   ```
   python3 -m venv doc-env
   ```

2. Activate the environment:

   - On GNU/Linux:

   ```
   source doc-env/bin/activate
   ```

   - On Windows:

```
doc-envScriptsactivate.bat
```

3. Install Sphinx:

```
pip install sphinx
```

Using virtualenv way

1. Install virtual environments support for Python:

```
pip install virtualenv
```

Note: If you're using a Unix-based system, you might need to use `sudo pip install virtualenv`.

---

2. Create an environment for sphinx tools:

```
virtualenv doc-env
```

3. Activate the environment:
   - On GNU/Linux:

```
source doc-env/bin/activate
```

   - On Windows:

```
doc-env\Scripts\activate.bat
```

4. Install Sphinx:

```
pip install sphinx
```

Get documentation sources

1. Go to a folder where you keep sources:

```
cd src
```

2. Clone documentation from the repository:

```
git clone https://github.com/tango-controls/tango-doc.git
```

3. Change current folder to the documentation folder:

```
cd tango-doc
```

4. Install extra requirements (like sphinx theme):

```
pip install -r requirements.txt
```

5. Try to build the documentation:

```
sphinx-build source build
```

6. Open build/index.html with your favorite browser to see if it has been built correctly.

Updating documentation

1. Create your local working branch:

   Note: The following command creates a branch based on *origin/master*. If you would like to contribute to another branch, e.g. directly to *9.2.5*, you need to use: **git checkout -b "TD-66-step-by-step-demo" origin/9.2.5**

   To see what what branch is the current one use: **git branch -a**. The current branch is marked with an asterisk (*).

   ```
   git checkout -b "TD-66-step-by-step-demo" origin/master
   ```

2. Edit a file (or create it if it doesn't exist) you would like to change. If you are following this tutorial for learning please use this file: `source/tutorials-and-howtos/tutorials/example.rst`
3. Make sure that the file appears in a relevant toc-tree (in some `index.rst` file or in `source/contents.rst`). If you are now learning please check `source/tutorials/index.rst`
4. Check if your changes have built correctly:

   ```
   sphinx-build source build
   ```

5. Check results with a browser. If you've edited the example, open `build/tutorials/index.html`

If everything is OK, you may commit changes and send a pull request (ask to review and merge into an on-line branch).

Committing changes

1. Add modifications to a commit list. For example:

   ```
   git add source/tutorials-and-howtos/tutorials/example.rst
   ```

   ```
   git add source/tutorials-and-howtos/tutorials/index.rst
   ```

2. Commit the changes providing some meaningful message. For example:

   ```
   git commit -m "doing tutorial"
   ```

   Note: The changes are now committed to your local repository. To share them, you need to push. You may repeat editing, checking and commit steps several times without pushing util you are happy with your work. This way you may track the history of changes.

3. If your work took a long time it is good to do rebasing with recent changes done by someone else. For example:

   ```
   git fetch origin
   ```

   ```
   git rebase origin/master
   ```

   Note: If you are contributing to other branch than *master*, for example directly to the *9.5.2*, you need to call **git rebase 9.5.2**

Pushing (to the GitHub repository)

1. Push your changes to the origin repository. For example:

   ```
   git push -u origin TD-66-step-by-step-demo
   ```

Now you are ready to ask for merging by sending a pull request on GitHub.

Pull request (asking for merge)

1. Go to https://github.com/tango-controls/tango-doc
2. Click the button *New pull request*.
3. On the *base* selector select the branch you want to update (usually *master* or some *#.#.#*).
4. On the *compare* selector select your branch.
5. Provide a relevant comment and click *Create pull request*.

Now, someone will review your contribution, merge into selected branch and publish. If he/she finds some issues, he/she will get back to you.

Continuing the contribution

If you would like to come up with some other contribution, you do not need to clone sources again. Follow the following steps:

1. Fetch changes from the origin repository:

   ```
   git fetch origin
   ```

2. Switch to the main branch you are going to update (for example 9.2.5):

   ```
   git checkout origin/9.2.5
   ```

3. Pull the changes:

   ```
   git pull
   ```

4. Follow steps from *Updating documentation*

9.1.2 Waltz (ex. TangoWebapp) installation tutorial (Windows)

Intended audience: developers, administrators, ***Programming language:*** java, javascript

Contributed by Florian Pourchayre

1. Download Apache Tomcat 8 and install it to *C:/apache-tomcat-8*. To verify that it got installed successfully, try to go to `http://127.0.0.1:8080`
2. Download the latest rest-server.jar unzip and put it in *C:/apache-tomcat-8/webapps*
3. Follow the .jar installation on rest-server Wiki (it is assumed below tango host is localhost:20000 i.e. 127.0.0.1:20000)

```
TANGO_ACCESS -> tango://127.0.0.1:20000/sys/access_control/1
TANGO_DB -> tango://127.0.0.1:20000/sys/database/2
TOMCAT_PORT ->8844
```

4. Export DatabaseDS, and load it to create a TangoAccessControl server (For this step, you will need to update "database" by "tangoAccessControl").

```
TangoAccessControl
1
TangoAccessControl
sys/access_control/1
```

5. On jive, click on edit, then create free property named "CtrlSystem". Then go to property (last tab), click on CtrlSystem, add a new property named "Services" with the value **AccessControl/tango:sys/access_control/1**
6. Download the latest TangoWebapp.war on then put it to *C://apache-tomcat-8/webapp* (When TomCat starts it creates a TangoWebapp folder).
7. define a tomcat user in *C://apache-tomcat-8/conf*:

```
<role rolename="mtango-user"/>
<role rolename="mtango-rest"/>
<user password="tango" roles="mtango-rest, root, manager-script,admin" username="tango-cs
↪"/>
```

8. Go to $Tango_home/bin and launch with a command line `TangoAccessControl.exe 1`
9. Go to *C:/apache-tomcat-8/webapp* and launch mtangorest.server with a command line e.g. `java -jar -DTANGO_HOST=127.0.0.1:20000 rest-server-1.8.jar test`
10. Open Jive, then open an atk panel on TangoControlAccess DS, then test device, and execute the AddDeviceForUser command with **["tango-cs","sys/database/2", "write"]** as parameter. Also, add a new environment variable name SUPER_TANGO with the value true. Then go on Astor, tool and open AccessControl HMI. Add ..*.* as allowed address for the user tango-cs.
11. Try your rest-server instance by the URL : `http://127.0.0.1:8844/tango/rest` (enter username: tango-cs, password : tango if necessary).
12. Time to try the tango webapp. try to go to the URL : `http://127.0.0.1:8080/TangoWebapp`. Get logged, then at the top enter as rest host : `http://127.0.0.1:8844` (`http://` is important), and as host : $Tango_host. Then refresh the page via *CTRL+F5*. You should have an access to the devices tree.

9.1.3 Scada introduction with TANGO examples

Intended audience: all

Scada introduction with TANGO examples from Jean-Charles Tournier. He works at CERN Geneva and teach at EPFL (Ecole Polytechnique Fédérale de Lausanne). Important notes from an expert! He uses Tango as an example in his course at EPFL.

Download .pdf file `here`.

9.1.4 Snapshots from TANGO labs at the ESRF

Intended audience: all

Some snapshots and short notes from labs (made by Paul Deglo de Besses) at the ESRF, which were held by Andy Götz and Jean Michel Chaize personally.

The document was done during the licence professionnelle "embedded systems" 2014-2015 at Grenoble UJF (Université Joseph Fourier).

Download .pdf file `here`.

9.1.5 TANGO as CS course in Uni

Intended audience: all

This is a technical course based upon Tango in order to share "best practices" in software development ... First session took place in Aïn Oussera - Algeria in Octobre 2014.

Couse is made by Raphaël Ponsard who teaches CS at LGM (Lycée du Grésivaudan Meylan) and at UJF (Université Joseph Fourier). He uses Tango in his course since many years.

Some of his material can be found here. He was also involved in a course on behalf of the IAEA (UN International Atomic Energy Agency in Vienna).

9.1.6 Write a TANGO device class for an Arduino temperature sensor

Intended audience: developers, ***Programming language:*** c++

> In `this presentation` you will find how to write a Tango Device Server for an Arduino temperature sensor.

9.1.7 General TANGO training for C++

Intended audience: developers, ***Programming language:*** c++

The support document `TANGO training for C++` for the Tango training course which covers most of the Tango features. The source code for all exercises is also available as zipped tar archives.

9.1.8 General TANGO training for Python

Intended audience: developers, ***Programming language:*** python

`TANGO training for Python` - the last Tango training given at MaxLab in Sweden. The training code examples and exercises are in Python. The source code for all exercises (in Python) is available as a zipped archive.

9.2 HOW-TOs

Intended audience: all

9.2.1 How to cross compile omniORB

Intended audience: advanced developers, ***Programming language:*** c++

This HowTo gives details on how you can cross compile omniORB.

When used with C++ or Python, Tango requires the omniORB librairies. Depending on the hardware on which you want to run Tango device server or client, it is sometimes needed to cross-compile Tango and therefore to cross compile omniORB. omniORB is distributed as a set of CORBA idl and cpp files. When you build omniORB, the idl files are compiled by a CORBA IDL to C++ compiler before they are sent to a classical C++ compiler with the set of cpp files in order to build the libraries. This is represented in the following figure

From this figure it is easay to understand that the first thing which is done when you build omniORB is to build the IDL to C++ compler which will be used in the following build steps. When you cross-compile omniORB, you provide to the configure command line the path to the C++ and C compilers for the target hardware (CXX=path_to_target_cpp_compiler and CC=path_to_target_C_compiler). If you don't take special care, omniORB build system will create the IDL to C++ compiler using these tools and it will generate an executable for your target host which will obviously not run on your compilation host. If for instance, you cross compile omniORB on a x64 host for a ARM CPU, the IDL to C++ compiler will be created for the ARM architecture and will certainly not run on your x64 host.

Here are what you have to type to correctly cross-compile omniORB:

```
$ TOP=/path_to_where_omniORB_has_been_downloaded
$ tar xjf $(TOP)/omniORB-4.1.7.tar.bz2
$ cd omniORB-4.1.7
$ mkdir build
$ cd build
$ ../configure --prefix=/opt/tango/orb/omniORB-4.1.7/arm
               CC=/opt/Xilinx/SDK/2013.2/gnu/arm/lin/bin/arm-xilinx-linux-gnueabi-gcc
               CXX=/opt/Xilinx/SDK/2013.2/gnu/arm/lin/bin/arm-xilinx-linux-gnueabi-g++
               --host=arm-xilinx-linux-gnueabi
               --build=x86_64-unknown-linux-gnu
```

When the configure command is run, the **CC** and **CXX** variables define the target architecture C and Cpp compilers. the configure **host** option defines the target architecture while the **build** option defines the architecture on which the cross compilation is done.

We now need to build the IDL to Cpp compiler

with the local compilers

```
$ make CC=gcc-4.7 -C src/tool/omniidl/cxx/cccp
$ make CXX=g++-4.7 -C src/tool/omniidl/cxx
$ make CC=gcc-4.7 -C src/tool/omkdepend
```

There are some small changes required in the omniORB build system: Edit dir.mk file in following directories:

- omniORB-4.1.7/src/appl/omniMapper/dir.mk
- omniORB-4.1.7/src/appl/omniNames/dir.mk
- omniORB-4.1.7/src/appl/utils/cation/dir.mk

- omniORB-4.1.7/src/appl/utils/convertior/dir.mk
- omniORB-4.1.7/src/appl/utils/genior/dir.mk
- omniORB-4.1.7/src/appl/utils/nameclt/dir.mk

to find the definition **@(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))** and replace it by **@(libs="$(CORBA_LIB_NODYN) -lstdc++"; $(CXXExecutable))**

Then finally build the omniORB libs

```
$ make
$ sudo make install
```

The omniORB Wiki and experiments done by ESRF colleagues have been used to write this HowTo.

9.2.2 How to create inheritance link between Tango classes (C++)

Intended audience: developers, ***Programming language:*** c++

This HowTo explains how it is possible to create a new Tango class which inherits from an already existing Tango class.

In Tango's vocabulary, a Tango class is not a computer language class. A Tango class has at least 2 computer language classes and very often even more than 2. This HowTo explains how you can create a new Tango class which "inherits" from an already existing class. In this case, inheritance means:

1. The high level class supports all the commands defined in the low level class on top of its own commands (if any),
2. The high level class supports all the attributes defined by the low leval class on top of its own attributes (if any),
3. The device(s) available for the external world is(are) an instance(s) of the high level class.

Warning: Once modified, Pogo will not be able to understand the high level Tango class and therefore can't be used for further changes in this class

The example case

Let's say that we already have a Tango class called **LowLevel** with one command called *LowLevelCmd* and one attribute called *LowLevelAttr*. We want to create a new Tango class called **HighLevel** with its own command called *HighLevelCmd* and its own attribute called *HighLevelAttr*. Both classes has been generated using Pogo. The following is a description of what has to be modified in the code generated by Pogo for the HighLevel Tango class in order to create this "inheritance" link.

All modified code in the following code snippets is in bold font.

The HighLevelClass.h file

In this file, we will modify the HighLevelClass declaration in order to inherit from the LowLevelClass class instead of inheriting from the Tango::DeviceClass class but we first need to add an extra include file at the file beginning:

```
#ifndef _HIGHLEVELCLASS_H
#define _HIGHLEVELCLASS_H

#include <tango.h>
#include <HighLevel.h>

#include <LowLevelClass.h>
```

Then, we change the HighLevelClass inheritance declaration:

```
//
// The HighLevelClass singleton definition
//

class
#ifdef WIN32
    __declspec(dllexport)
#endif
HighLevelClass : public LowLevel_ns::LowLevelClass
{
public:
//  properties member data

//  add your own data members here
//------------------------------------
```

The HighLevelClass.cpp file

In this file, we have to modify:

1. The HighLevelClass definition (due to the change in its inheritance),
2. The HighLevelClass command_factory() method,
3. The HighLevelClass attribute_factory() method.

```
//+------------------------------------------------------------------------------
//
// method :         HighLevelClass::HighLevelClass(string &s)
//
// description :    constructor for the HighLevelClass
//
// in : - s : The class name
//
//-------------------------------------------------------------------------------
HighLevelClass::HighLevelClass(string &s):LowLevel_ns::LowLevelClass(s)
{

    cout2 << "Entering HighLevelClass constructor" << endl;
```

(continues on next page)

(continued from previous page)

```
    set_default_property();
    get_class_property();
    write_class_property();

    cout2 << "Leaving HighLevelClass constructor" << endl;

}
```

Then, the changes in the command_factory() method which needs to call the LowLevelClass command_factory() method:

```
//+------------------------------------------------------------------
//
// method :         HighLevelClass::command_factory
//
// description :    Create the command object(s) and store them in the
//          command list
//
//-------------------------------------------------------------------
void HighLevelClass::command_factory()
{
    LowLevel_ns::LowLevelClass::command_factory();

    command_list.push_back(new HighLevelCmdClass("HighLevelCmd",
        Tango::DEV_VOID, Tango::DEV_VOID,
        "",
        "",
        Tango::OPERATOR));

    //  add polling if any
    for (unsigned int i=0 ; i<command_list.size(); i++)
    {
    }
}
```

Finally, the changes in the attribute_factory() method which needs to call the LowLevelClass attribute_factory() method:

```
//+------------------------------------------------------------------
//  Method: HighLevelClass::attribute_factory(vector<Tango::Attr *> &att_list)
//-------------------------------------------------------------------
void HighLevelClass::attribute_factory(vector<Tango::Attr *> &att_list)
{
    LowLevel_ns::LowLevelClass::attribute_factory(att_list);

    //  Attribute : HighLevelAttr
    HighLevelAttrAttrib *high_level_attr = new HighLevelAttrAttrib();
    att_list.push_back(high_level_attr);

    //  End of Automatic code generation
    //-------------------------------------------------------------
}
```

The HighLevel.h file

This file has to be modified in order to:

1. **Change the HighLevel class inheritance from Tango::Device_3Impl to** LowLevel_ns::LowLevel
2. **Add a new data member in the HighLevel class in order to correctly** implement the device Init command (a boolean is enough)
3. **Modify the class destructor for a correct management of the device** Init command

First, we have to add a new include file:

```
#ifndef _HIGHLEVEL_H
#define _HIGHLEVEL_H

#include <tango.h>
#include <LowLevel.h>
```

Then, the change in the HighLevel class inheritance:

```
class HighLevel: public LowLevel_ns::LowLevel
{
public :
    //  Add your own data members here
    //-----------------------------------------
```

The addition of the new data member at the end of the HighLevel class declaration:

```
protected :
    //  Add your own data members here
    //-----------------------------------------
    bool device_constructed;
}
```

And finally, the change in the HighLevel class destructor:

```
/**
 * The object desctructor.
 */
    ~HighLevel() {device_constructed=false;delete_device();}
```

The HighLevel.cpp file

In this file, we have to modify

1. The HighLevel class constructors to reflect the change in its inheritance and to initialize the new data member (device_constructed)
2. The HighLevel class delete_device() and init_device() to correctly handle the device Init command
3. The HighLevel class always_executed_hook() and read_attr_hardware() methods in order that they call the corresponding LowLevel class method

Let's start with the changes in the HighLevel class constructors:

```
//+------------------------------------------------------------------
//
// method :         HighLevel::HighLevel(string &s)
//
// description :    constructor for simulated HighLevel
//
// in : - cl : Pointer to the DeviceClass object
//      - s : Device name
//
//------------------------------------------------------------------
HighLevel::HighLevel(Tango::DeviceClass *cl,string &s)
:LowLevel_ns::LowLevel(cl,s.c_str()),device_constructed(false)
{
    init_device();
    device_constructed = true;
}

HighLevel::HighLevel(Tango::DeviceClass *cl,const char *s)
:LowLevel_ns::LowLevel(cl,s),device_constructed(false)
{
    init_device();
    device_constructed = true;
}

HighLevel::HighLevel(Tango::DeviceClass *cl,const char *s,const char *d)
:LowLevel_ns::LowLevel(cl,s,d),device_constructed(false)
{
    init_device();
    device_constructed = true;
}
```

Now, the modified HighLevel class init_device() and delete_device() methods:

```
//+------------------------------------------------------------------
//
// method :         HighLevel::delete_device()
//
// description :    will be called at device destruction or at init command.
//
//------------------------------------------------------------------
void HighLevel::delete_device()
{
    INFO_STREAM << "In HighLevel::delete_device()" << endl;

    //  Delete device's allocated object

//    Your specific code (if any)
if (device_constructed == true)
        LowLevel_ns::LowLevel::delete_device();
}

//+------------------------------------------------------------------
//
```

(continues on next page)

(continued from previous page)

```
// method :         HighLevel::init_device()
//
// description :    will be called at device initialization.
//
//--------------------------------------------------------------------------
void HighLevel::init_device()
{
    if (device_constructed == true)
        LowLevel_ns::LowLevel::init_device();

    INFO_STREAM << "HighLevel::HighLevel() create device " << device_name << endl;

    // Initialise variables to default values
    //--------------------------------------------
```

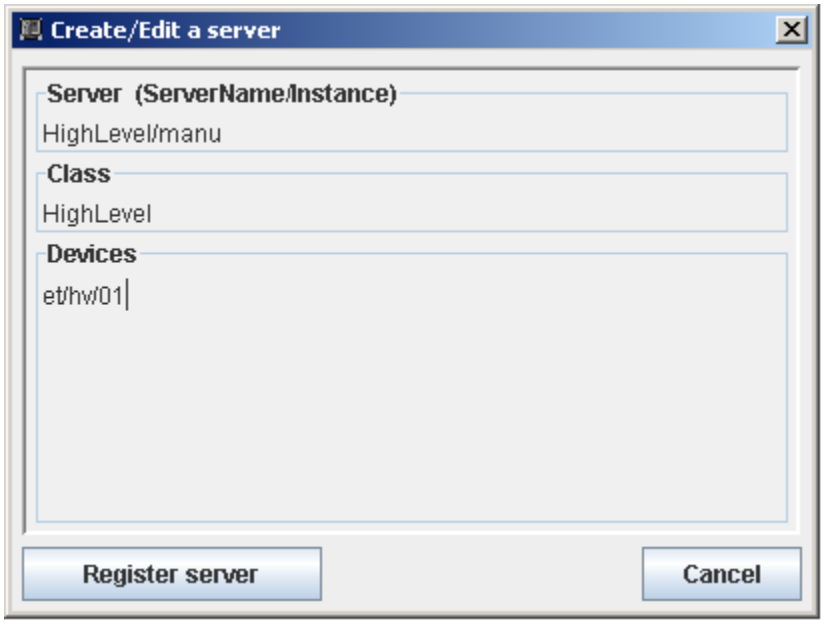
And finally, the HighLevel class always_executed_hook() and read_attr_hardware() methods:

```
//+--------------------------------------------------------------------------
//
// method :         HighLevel::always_executed_hook()
//
// description :    method always executed before any command is executed
//
//--------------------------------------------------------------------------
void HighLevel::always_executed_hook()
{
    LowLevel_ns::LowLevel::always_executed_hook();
    INFO_STREAM << "In HighLevel::always_executed_hook()" << endl;
//     Your code here (if any)
}

//+--------------------------------------------------------------------------
//
// method :         HighLevel::read_attr_hardware
//
// description :    Hardware acquisition for attributes.
//
//--------------------------------------------------------------------------
void HighLevel::read_attr_hardware(vector<long> &attr_list)
{
    LowLevel_ns::LowLevel::read_attr_hardware(attr_list);
        DEBUG_STREAM << "HighLevel::read_attr_hardware(vector<long> &attr_list) entering.
↪.. "<< endl;
    //  Add your own code here
}
```

Don't forget to also modify the **Makefile** in order to link the three LowLevel Tango class object files (Lowlevel.o, LowLevelClass.o and LowLevelStateMachine.o) to your executable.

Defining the class in the Tango control system

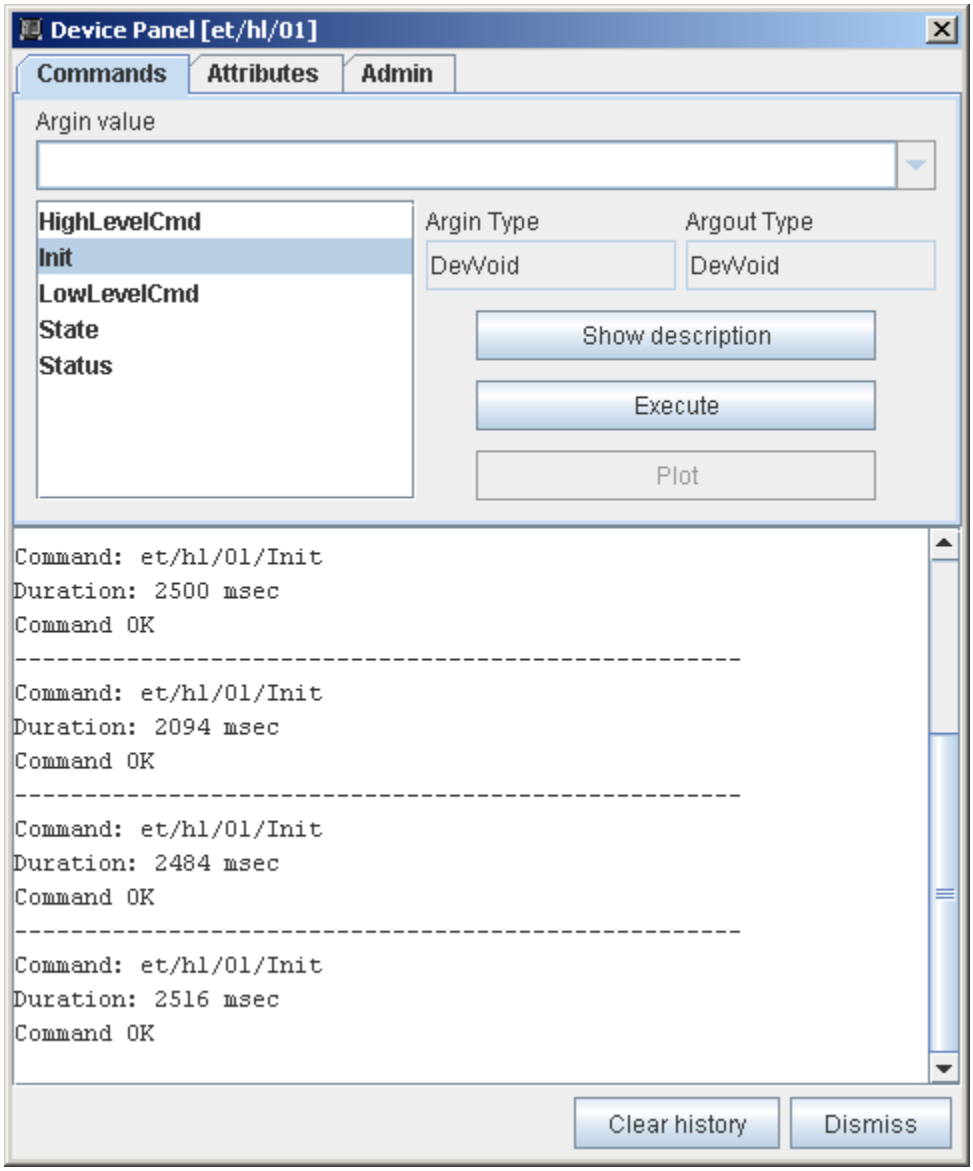
Once the executable is linked, the device server process instance has to be created in the Tango database (using Jive for instance).

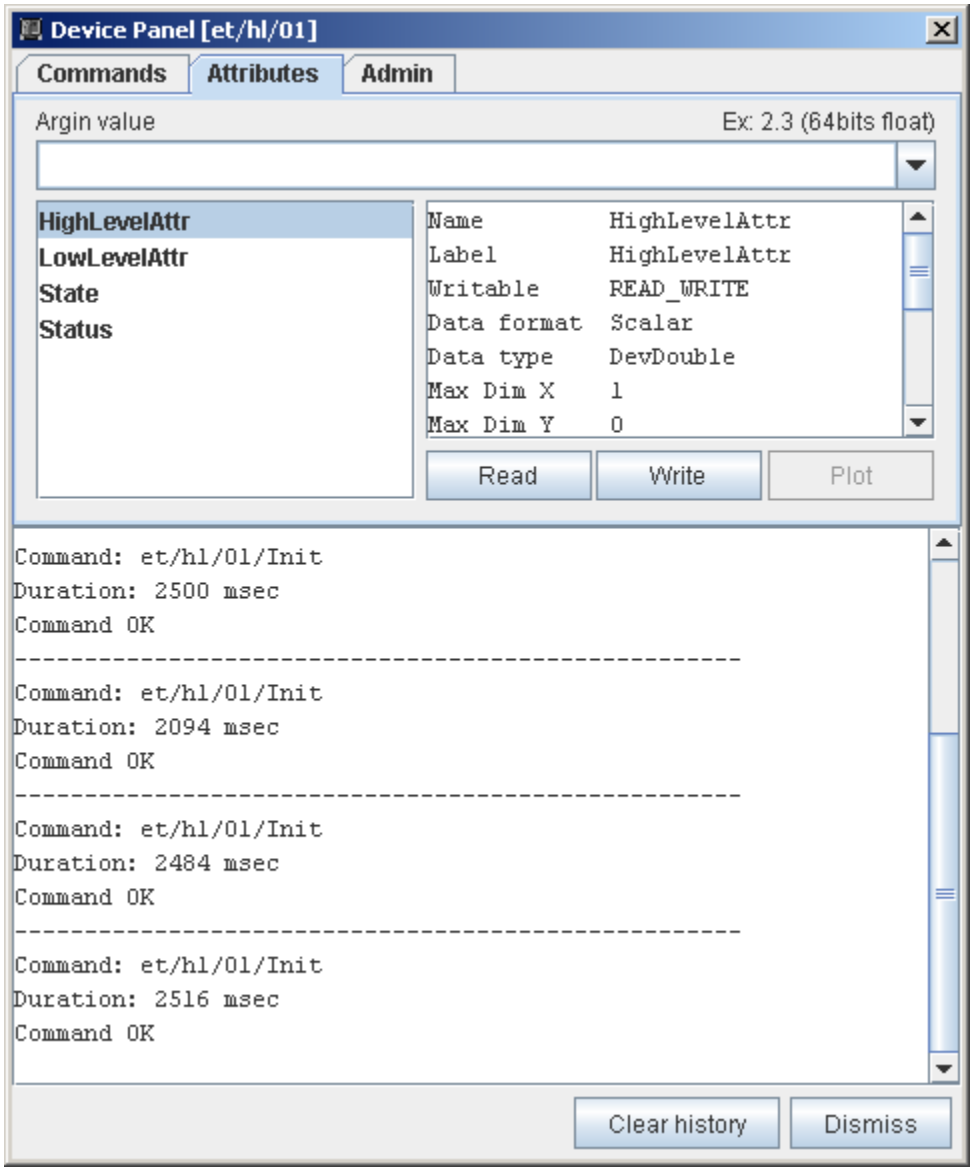


Running your class

Nothing special here. Simply start your device server as usual. Connecting to the device using a "Test Panel" shows that the device now has:

- two commands (one from the LowLevel class and the other from the HighLevel class),
- two attributes (from LowLevel class and from HighLevel class).


Device Panel [et/hl/01]
Commands
Attributes
Admin
Argin value
HighLevelCmd
Init
LowLevelCmd
State
Status
Argin Type
Argout Type
DevVoid
DevVoid
Show description
Execute
Plot
Command: et/hl/01/Init
Duration: 2500 msec
Command OK
Command: et/hl/01/Init
Duration: 2094 msec
Command OK
Command: et/hl/01/Init
Duration: 2484 msec
Command OK
Command: et/hl/01/Init
Duration: 2516 msec
Command OK
Clear history
Dismiss



Conclusion

With these relatively simple changes in the HighLevel class, we now have a device instance of a Tango class which "inherits" from another Tango class. The drawback of this method is that once the file has been modified, **Pogo will not be able to understand the HighLevel class** any more and should not be used for further changes in this class!

With a couple of "virtual" methods, it is also possible in the HighLevel class to overwrite a command or an attribute defined in the Lowlevel class.

9.2.3 How to deal with string Tango attribute (C++)

Intended audience: developers, ***Programming language:*** c++

This HowTo gives example on howthe code for string attribute could be written. This is only for C++ Tango device class.

Unfortunately, CORBA has been standardized before the standardization of the C++ string class. In the CORBA IDL to C++ mapping standard, the IDL string data type maps to classical C string. This means that you have to deal with

1. The pointer to the memory where the string is stored (char *)
2. The memory where the the string character are stored

This adds one level of complexity and you have to take care about memory allocation for these entities.

In a Tango class with string attribute (scalar, spectrum or image), you have to deal with this level of complexity. The aim of this HowTo is to give examples on how the code related to these string attributes has to be written

The first question you have to answer is: Do I want static or dynamic memory allocation for my string attribute?

Static memory allocation means that the memory is allocated once. Dynamic allocation means that the memory used for the attribute is allocated and freed each time the attribute is read. Once one method is chosen, both the pointer and the characters array memory has to follow the same rule (all static or all dynamic but not a mix of them)

Scalar attribute - static allocation

Within Pogo, for a correct management of this type of attribute, do not click the "allocate" toggle button when you define the attribute. The pointer to the character area is defined as one of the device data member in the file MyDev.h. The Tango data type DevString is simply a typedef for a good old "char *" pointer.

```
class MyDev : public Tango::Device_4Impl
{
    /*----- PROTECTED REGION ID(MyDev::Data Members) ENABLED START -----*/

    // Add your own data members
public:
    Tango::DevString the_str;
    /*----- PROTECTED REGION END -----*/ // MyDev::Data Members
}
```

In the init_device method (file MyDev.cpp), you have to initialize the attribute data member created for you by Pogo

```
void MyDev::init_device()
{
    DEBUG_STREAM << "MyDev::init_device() create device " << device_name << endl;

    /*----- PROTECTED REGION ID(StringAttr::init_device) ENABLED START -----*/

    attr_StringAttr_read = &the_str;

    /*----- PROTECTED REGION END -----*/    // MyDev::init_device
}
```

The attribute related code in the file MyDev.cpp looks like

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;

    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    the_str = const_cast<char *>("Hola Barcelona");
    attr.set_value(attr_StringAttr_read);

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

The pointer *the_str* defined as a device data member is initialized to a statically allocated string. The argument of the Attribute::set_value() method is of type "char **" which is coherent with the definition of the Tango::DevString type. Nevertheless, the definition of statically allocated string in C / C++ is a "const char *". This is why we need a const_cast during the pointer initialization.

Note that the use of the Pogo generated data member (named attr_StringAttr_read in our case) is not mandatory. You can directly give the address of the *the_str* pointer to the Attribute::set_value() method and do not need any additional code in the init_device() method.

Scalar attribute - dynamic allocation

Memory freeing done by Tango layer

Within Pogo, for a correct management of this type of attribute, do not click the "allocate" toggle button when you define the attribute. In this case, we do not need to define anything as device data member.

The attribute related code in the file MyDev.cpp looks like

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;

    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    attr_StringAttr_read = new Tango::DevString;
    *attr_StringAttr_read = Tango::string_dup("Bonjour Paris");
    attr.set_value(attr_StringAttr_read,1,0,true);

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

As explained in the introduction, both the pointer and the char array memory are dynamically allocated. The pointer is allocated first, then it is is initialized with the result of a Tango::string_dup() method which allocates memory and copy the string given as argument (It's the same call than CORBA::string_dup). The Tango attribute value is set with the classical set_value() method but requiring Tango to free all the memory previously allocated.

Memory freeing done by device class

This example is in the case where within Pogo, the "allocate" toggle button was active when the attribute was defined.

The init_device() and delete_device() method looks like:

```
void MyDev::init_device()
{
    DEBUG_STREAM << "MyDev::init_device() create device " << device_name << endl;

    attr_StringAttr_read = new Tango::DevString[1];

    /*----- PROTECTED REGION ID(StringAttr::init_device) ENABLED START -----*/

    *attr_StringAttr_read = NULL;

    /*----- PROTECTED REGION END -----*/    // MyDev::init_device
}

void MyDev::delete_device()
{
    /*----- PROTECTED REGION ID(MyDev::delete_device) ENABLED START -----*/

    Tango::string_free(*attr_StringAttr_read);

    /*----- PROTECTED REGION END -----*/    // MyDev::delete_device
    delete[] attr_StringAttr_read;

}
```

The pointer for the characters array is allocated in the init_device() and initialized to NULL. In the delete_device() method, the character array memory is freed with the Tango::string_free() method. If you are using a Tango C++ library version older than v9.3.3, you have to use CORBA::string_free() method instead of Tango::string_free().

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;

    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    Tango::string_free(*attr_StringAttr_read);
    *attr_StringAttr_read = Tango::string_dup("Bonjour Paris");
    attr.set_value(attr_StringAttr_read);

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

The Tango::DevString pointer created by Pogo (named attr_StringAttr_read) is allocated in the init_device() method (Pogo generated code) and freed in the delete_device() method (Pogo generated code). Nevertheless, nothing is done for the memory used to store the characters array. This is done in this code snippet in the first line of the protected region. Then the memory is allocated for the new characters array and used to set to the Tango Attribute instance value.

Note that only the memory allocatd for the characters array is allocated / freed at each attribute reading. The pointer is allocated once in the init_device() method and freed in the delete_device() method.

Spectrum / Image attribute - static allocation

The code needed in this case is very similar to the scalar case. We also need pointers to the character areas. They are defined as device data member in the file MyDev.h.

```
class MyDev : public Tango::Device_4Impl
{
   /*----- PROTECTED REGION ID(MyDev::Data Members) ENABLED START -----*/

  //        Add your own data members
public:
   Tango::DevString  the_str_array[2];

   /*----- PROTECTED REGION END -----*/ // MyDev::Data Members
```

In the init_device method (file MyDev.cpp), you have to initialize the attribute data member created for you by Pogo

```
void MyDev::init_device()
{
    DEBUG_STREAM << "MyDev::init_device() create device " << device_name << endl;

    /*----- PROTECTED REGION ID(StringAttr::init_device) ENABLED START -----*/

attr_StringAttr_read = the_str_array;

    /*----- PROTECTED REGION END -----*/    // MyDev::init_device
}
```

The attribute related code in the file MyDev.cpp looks like

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;
    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    the_str_array[0] = const_cast<char *>("Hola Barcelona");
    the_str_array[1] = const_cast<char *>("Tchao Trieste");
    attr.set_value(attr_StringAttr_read,2);

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

The array *the_str_array* defined as a device data member is initialized to statically allocated strings. The argument of the Attribute::set_value() method is of type “char **” which is coherent with the definition of the Tango::DevString type. Nevertheless, the definition of statically allocated string in C / C++ is a “const char *”. This is why we need a const_cast during the pointer initialization.

Note that the use of the Pogo generated data member (named attr_StringAttr_read in our case) is not mandatory. You can directly give the name of the *the_str_array* data member to the Attribute::set_value() method and do not need any additional code in the init_device() method.

Something similar can be done using a vector of C++ strings if:

1. The vector is initialized somewhere in your Tango class
2. The vector is declared as a device data member (in MyDev.h)

3. The vector size is less or equal to the attribute maximum dimension

The code looks like

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;
    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    for (unsigned int i = 0;i < vs.size();i++)
       the_str_array[i] = const_cast<char *>(vs[i].c_str());
    attr.set_value(attr_StringAttr_read,vs.size());

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

Spectrum / Image attribute - dynamic allocation

Memory freeing done by Tango layer

Within Pogo, for a correct management of this type of attribute, do not click the "allocate" toggle button when you define the attribute. In this case, we do not need to define anything as device data member.

The attribute related code in the file MyDev.cpp looks like

```
void MyDev::read_StringAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyDev::read_StringAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;
    /*----- PROTECTED REGION ID(MyDev::read_StringAttr) ENABLED START -----*/
    //  Set the attribute value
    Tango::DevString *ptr_array = new Tango::DevString [2];
    ptr_array[0] = Tango::string_dup("Bonjour Paris");
    ptr_array[1] = Tango::string_dup("Salut Grenoble");
    attr.set_value(ptr_array,2,0,true);

    /*----- PROTECTED REGION END -----*/    // MyDev::read_StringAttr
}
```

The Tango::DevString pointer array is allocated first, then it is is initialized with the results of a Tango::string_dup() method which allocates memory and copy the string given as argument (It's the same call than CORBA::string_dup). The Tango attribute value is set with the classical set_value() method but requiring Tango to free all the memory previously allocated.

Conclusion

To conclude this HowTo, the important point to remember:

Note: Do not mix solution. Use dynamic or static allocation but for the 2 levels (pointer and character array)

Warning: If you do not follow this rule, the penalty will be fatal !!

9.2.4 How to reconnect Database at device server startup time

Intended audience: advanced developers, ***Programming language:*** c++

This HowTo is a CPP example of how you can program a Tango DS in order that it can be started before the Tango's database and which will wait for the Tango database to start.

Add the following lines at the beginning of the main method (File *main.c*):

```
//  Set an automatic retry on database connection
   //-------------------------------------------------
   Tango::Util::_daemon = true;
   Tango::Util::_sleep_between_connect = 5;
```

The device server will retry to connect database in case of failure periodicaly (every 5 seconds in example).

9.2.5 How to distinguish clients on the server side

Intended audience: advanced developers, ***Programming language:*** c++

Problem overview

Some times Tango device is not just about talking with the hardware. Now-days a Tango device could be a sophisticated thing that performs complex work and serves numbers of clients simultaneously. And it might be required that the context of these clients may vary (context - a set of parameters of the task or result configuration etc). This is especially true due to multithreading reality we have.

Detailed cases

Client wants to specify in which format it receives an output from the server. Lets say it can be plain text or JSON or XML etc.

Server runs several tasks in parallel. One task per client. And each client wants to know its task ID, let's say to control or monitor the task's execution.

Solution overview

CORBA provides clnt_idnt, i.e. client identity. This can be used to create a concurrent map: ClientIdentity -> Context. And then each requests from the client can be served within the specified context.

Java:

```
//Server.java
//ConcurrentMap to hold each client's context
private final ConcurrentMap<String, RequestContext> ctxs =  Maps.newConcurrentMap();

//see Tango Java API for deviceManagment
@DeviceManagement
private DeviceManager deviceManager;

//this will be an attribute, each client will see its own value
@Attribute
//this method returns client identity from CORBA as String
public String getClientId() throws Exception {
    //deviceManager exports CORBA's client identity feature
    return ClientIDUtil.toString(deviceManager.getClientIdentity());
}

//command that returns its result in the format defined in the client's context
@Command
public String doJob() throws Exception {
    String clientId = getClientId();
    RequestContext ctx = ctxs.get(clientId);
    switch(ctx.outputType){
        case OutputType.PLAIN:
            return plainResult();
        case OutputType.JSON:
            return jsonResult();
    }
}
```

```
//RequestContext.java
//NOTE this class must be thread safe
@Immutale
public class RequestContext {
    //OutputType is an enum
    public final OutputType outputType;

    public RequestContext(OutputType outputType) {
        this.outputType = outputType;
    }

    /**
     * Creates default context
     */
    public RequestContext() {
        this(OutputType.PLAIN);
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

CPP:

TODO

Python:

TODO

9.2.6 How to add dynamic attributes to a device class

Intended audience: advanced developers, ***Programming language:*** c++

Creating new attributes at Startup

by Sergi Rubio Manrique, srubio@cells.es, with help from R.Bourtembourg and E.Taurel.

Do you need to create more or less Attributes depending on a certain device property? Are you interested in changing the name/type/dimensions of your attributes depending on different applications?

That is common while working with Input/Output devices with a variable number of channels (PLCs, DaqBoards, Timers/Counters), when depending of the operation mode more or less scalar or spectrum attributes could be needed.

This brief **HowTo** implements this feature in a Tango class base on code generated by **Pogo 8**.

Generating your class with Pogo

DynAttr

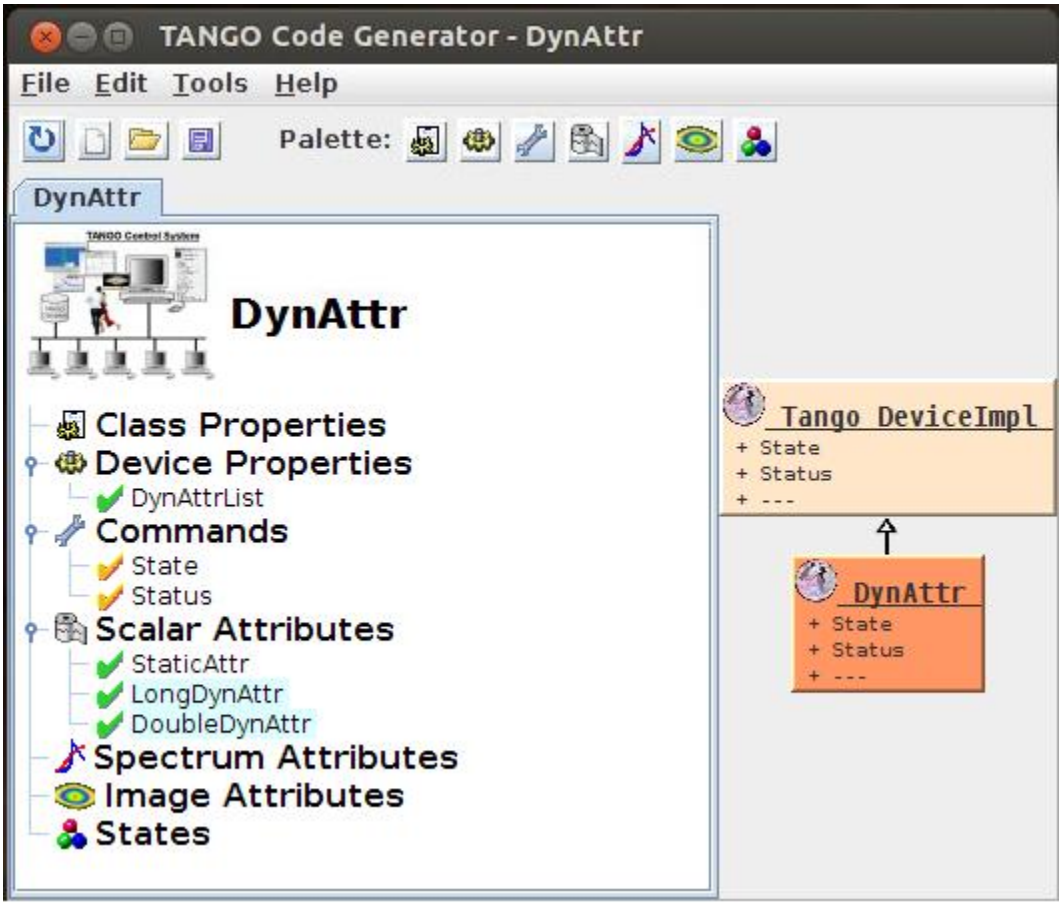
with three types of attributes:

1. StaticAttr: A scalar, Tango::DevShort, READ attribute,
2. LongDynAttr: A scalar, Tango::DevLong, READ_WRITE attribute,
3. DoubleDynAttr: A scalar, Tango::DevDouble, READ_WRITE attribute.

Each device belonging to this class has one StaticAttr and a dynamic list of LongDynAttr or DoubleDynAttr attributes. This list of dynamic attributes is defined with a device property named **DynAttrList**

Therefore, using Pogo, we define a Tango class with three scalar attributes with the definition given above. We also create the DynAttrList property for defining dynamic attribute. This is a vector of strings with a couple of strings for each attribute. The first string is the dynamic attribute type (LongDynAttr or DoubleDynAttr) and the second string is the attribute name.

Note: The dynamic attributes have a light green background color on the main Pogo window.



Implementing the Dynamic Attributes

Pogo 8 has been modified to help user to create Tango class with dynamic attribute(s).

Dynamic attribute registration

To register device dynamic attribute, you need to:

1. call the Tango::DeviceImpl::add_attribute() method for each device dynamic attribute.
2. create the data used with the attribute (for the Attribute::set_value() method).

Pogo generates method(s) named **DynAttr::add_<Dyn attr class name>_dynamic_attribute(string att_name)** which do this job for you. You have to call these methods according to your needs in another Pogo generated method named **DynAttr::add_dynamic_attributes()** which is executed at device creation time. In our example, in this method we have to:

- analyze the content of the device DynAttrList property and create the necessary attributes using the helper method also generated by Pogo

The code of the DynAttr::add_dynamic_attributes() method looks like

```
1 void DynAttr::add_dynamic_attributes()
2 {
3     //  Example to add dynamic attribute:
```

(continues on next page)

(continued from previous page)

```
    //  add_LongDynAttr_dynamic_attribute("MyAttribute");
    //  add_DoubleDynAttr_dynamic_attribute("MyAttribute");

    /*----- PROTECTED REGION ID(DynAttr::add_dynamic_attributes) ENABLED START -----*/

    if (dynAttrList.empty() == false)
    {
    if ((dynAttrList.size() % 2) != 0)
    {
// Throw exception
    }

    for (unsigned int i = 0;i < dynAttrList.size();i = i + 2)
    {
        if (dynAttrList[i] == "LongDynAttr")
        {
        add_LongDynAttr_dynamic_attribute(dynAttrList[i + 1]);
        }
        else if (dynAttrList[i] == "DoubleDynAttr")
        {
        add_DoubleDynamicAttr_dynamic_attribute(dynAttrList[i + 1]);
        }
        else
        {
// Throw exception
        }
    }
    }

    /*----- PROTECTED REGION END -----*/    //  DynAttr::add_dynamic_attributes
```

The code to throw exception has been removed.

Note: The data associated with all LongDynAttr dynamic attributes are initialized to 0 and the data associated to all DoubleDynAttr dynamic attributes are initialized with 0.0

The definition of the DoubleDynAttr attribute is simply to return when read, the last value which has been written. The code for the DoubleDynAttr reading/writing is the following

```
void DynAttr::read_DoubleDynAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "DynAttr::read_DoubleDynAttr(Tango::Attribute &attr) entering... " <
↪< endl;
    Tango::DevDouble *att_value = get_DoubleDynAttr_data_ptr(attr.get_name());

    /*----- PROTECTED REGION ID(DynAttr::read_DoubleDynAttr) ENABLED START -----*/

    //  Set the attribute value
    attr.set_value(att_value);

    /*----- PROTECTED REGION END -----*/    //  DynAttr::read_DoubleDynAttr
```

(continues on next page)

(continued from previous page)

```
}

void DynAttr::write_DoubleDynAttr(Tango::WAttribute &attr)
{
    DEBUG_STREAM << "DynAttr::write_DoubleDynAttr(Tango::Attribute &attr) entering... " <
↪< endl;

    //  Retrieve write value
    Tango::DevDouble    w_val;
    attr.get_write_value(w_val);

    /*----- PROTECTED REGION ID(DynAttr::write_DoubleDynAttr) ENABLED START -----*/

    Tango::DevDouble *att_value = get_DoubleDynAttr_data_ptr(attr.get_name());
    *att_value = w_val;

    /*----- PROTECTED REGION END -----*/    //  DynAttr::write_DoubleDynAttr
}
```

The code of the read method in it's Pogo generated part retrieves a pointer to the data associated with this attribute with the helper method named **DynAttr::get_<Dyn attr class name>_data_ptr(sting att_name)**. The user code simply pass this pointer to the Tango Attribute::set_value() method.

The user code of the write method also uses the Pogo generated helper method to get the attribute data pointer and set this data to the value sent by the caller.

The definition of the LongDynAttr is a bit more sophisticated. For one device of this Tango class, we have several dynamic attributes of this LongDynAttr type. According to which attribute is read or written, we have to call different method accessing the hardware.

The code for reading/writing the LongDynAttr attribute is given below:

```
void DynAttr::read_LongDynAttr(Tango::Attribute &attr)
{
    DEBUG_STREAM << "DynAttr::read_LongDynAttr(Tango::Attribute &attr) entering... " <<␣
↪endl;
    Tango::DevLong *att_value = get_LongDynAttr_data_ptr(attr.get_name());

    /*----- PROTECTED REGION ID(DynAttr::read_LongDynAttr) ENABLED START -----*/

    string &att_name = attr.get_name();
    if (att_name == dynAttrList[1])
    *att_value = read_hardware_channel1();  // Access hardware for channel 1 which is␣
↪the first attribute in the list
    else if (att_name == dynAttrList[3])
    *att_value = read_hardware_channel2();  // Access hardware for channel 2 which is␣
↪the second attribute in the list
    else
    {
// Throw exception
    }

    //  Set the attribute value
    attr.set_value(att_value);
```

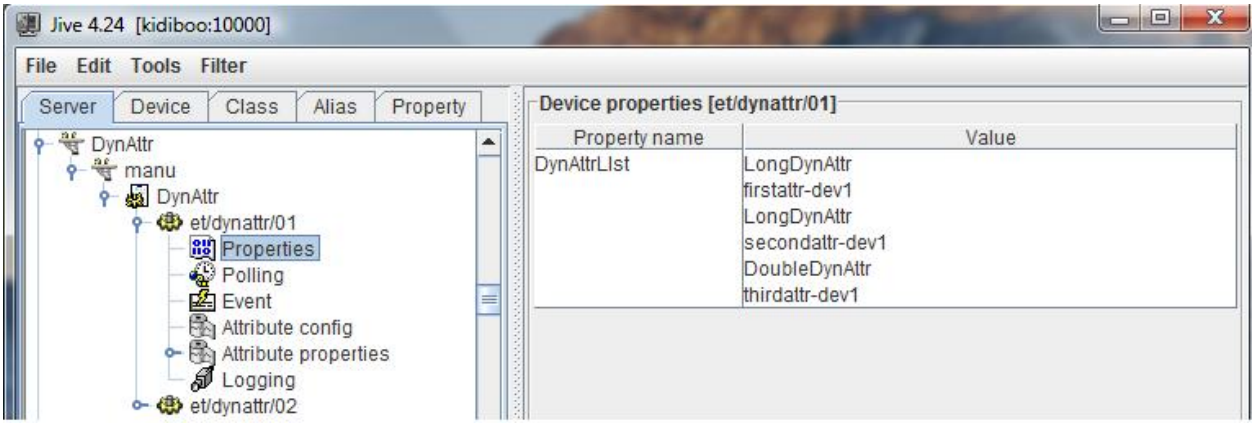
(continues on next page)

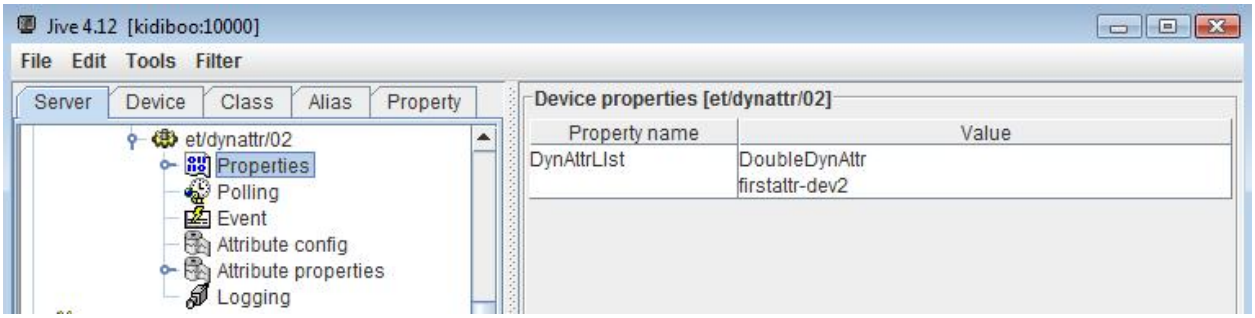
(continued from previous page)

```
    /*----- PROTECTED REGION END -----*/     //  DynAttr::read_LongDynAttr
}

void DynAttr::write_LongDynAttr(Tango::WAttribute &attr)
{
    DEBUG_STREAM << "DynAttr::write_LongDynAttr(Tango::Attribute &attr) entering... " << endl;

    //  Retrieve write value
    Tango::DevLong  w_val;
    attr.get_write_value(w_val);

    /*----- PROTECTED REGION ID(DynAttr::write_LongDynAttr) ENABLED START -----*/

    string &att_name = attr.get_name();
    if (att_name == dynAttrList[1])
    write_hardware_channel1(w_val); // Access hardware for channel 1 which is the first attribute in the list
    else if (att_name == dynAttrList[3])
    write_hardware_channel2(w_val); // Access hardware for channel 2 which is the second attribute in the list
    else
    {
// Throw exception
    }

    /*----- PROTECTED REGION END -----*/     //  DynAttr::write_LongDynAttr
}
```

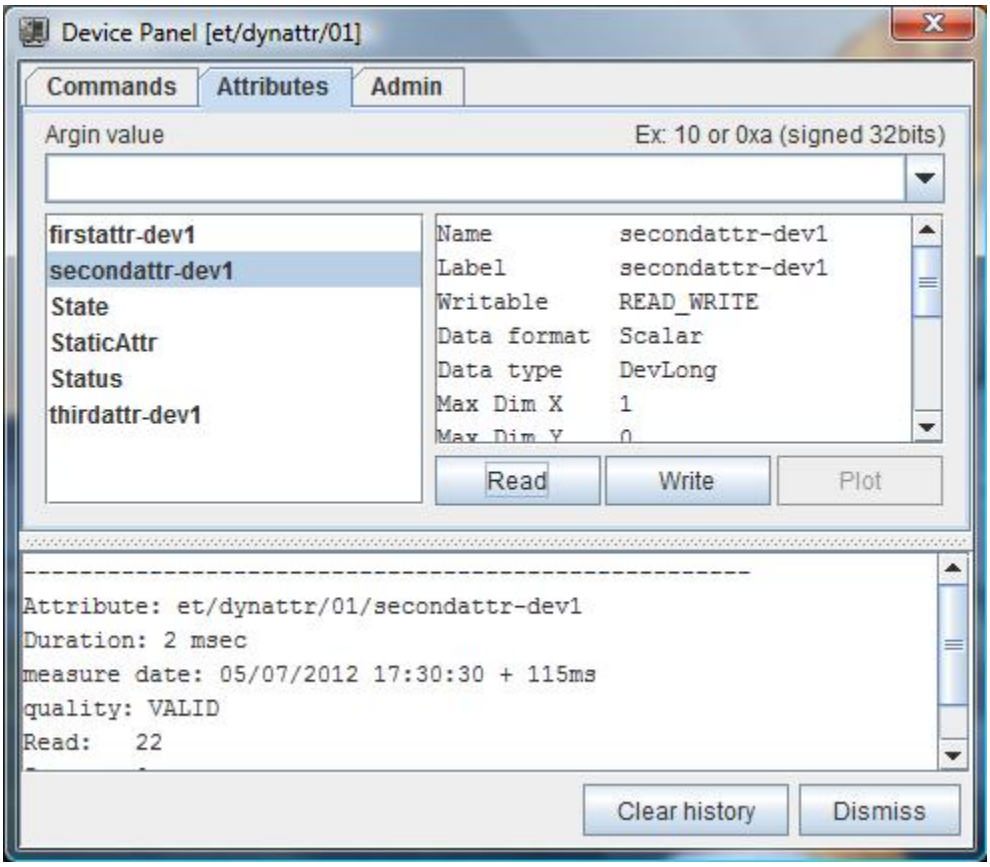
Running the server

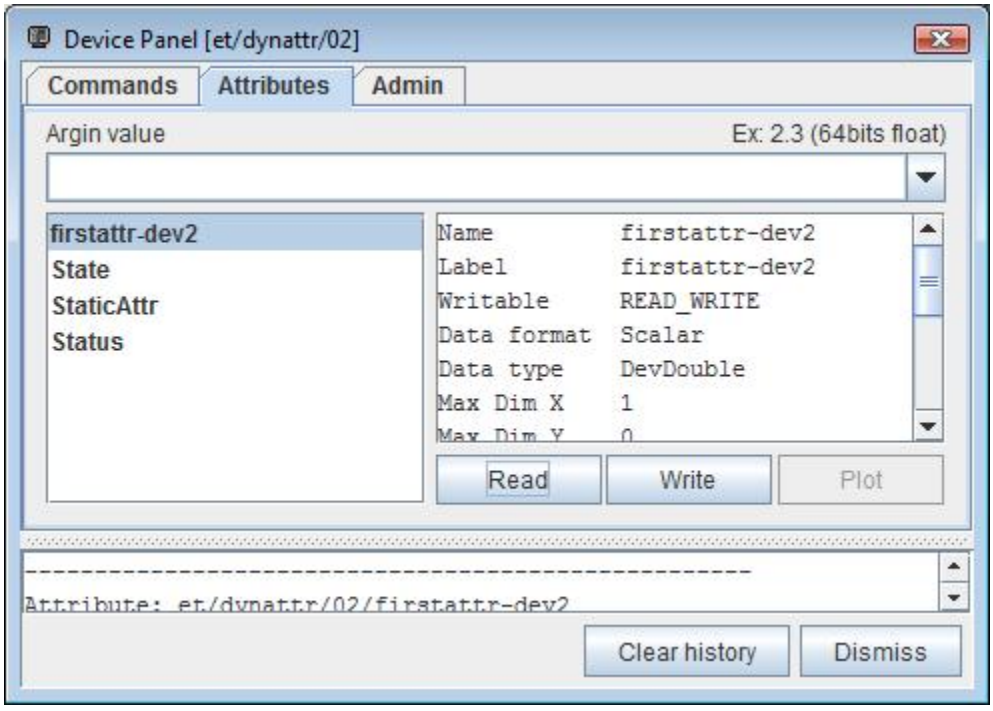
A Tango device server process hosting this DynAttr class has been defined in the database with two device. The dynamic attributes for these two devices are:





As shown by the Pogo screen-shot in the beginning of this HowTo, the Tango class also defines a static attribute for each device named StaticAttr. Running the device server and opening TestDevice panels on each device displays device attribute list:





This method fully supports restarting device(s) or server using the device server process admin device.

Conclusion

Pogo 8 simplifies a lot the code needed to write Tango class with dynamic attributes. Restarting device(s) from the admin device is fully supported.

9.2.7 How to extract read and set values from a scalar attribute

Intended audience: developers, ***Programming language:*** c++

The following example will show how you can extract the read and set values from a scalar attribute. In the example, the attribute is a *Tango::DevDouble*.

```
Tango::DevVarDoubleArray * attr_val;
Tango::DevDouble read_value, set_value;
try {
    Tango::DeviceAttribute dev_attr =  tg_device_proxy->read_attribute("attr_name");
    dev_attr >> attr_val;
    read_value = (*attr_val)[0];
    set_value = (*attr_val)[1];
    delete attr_val;
}
catch(Tango::DevFailed &e){
    Tango::Except::print_exception(e);
    throw;
}
```

Note: The operator ** >>** will allocate the memory for the **Tango::DevVarDoubleArray** but the programmer should free this memory when the work is done with this variable.

9.2.8 How to Fandango

Intended audience: developers, ***Programming language:*** python

by *Sergi Rubio*

fun4tango, functional programming for Tango

Warning: Fandango package is a refactored version of the old PyTango_utils package, which is now deprecated

Note: Fandango is now on github

Several recipes available at github

Description

Fandango (previously called PyTango_utils) is a Python module created to simplify the configuration of big control systems; implementing the behavior of Jive (configuration) and/or Astor (deployment) tools in methods that could be called from scripts using regexp and wildcards.

It has been later extended with methods commonly used in some of our python API's (archiving, CCDB, alarms, vacca) or generic devices (composers, simulators, facades).

Downloading

Fandango module is now on github:

```
$ git clone https://github.com/tango-controls/fandango
```

For Tango 9 you can still get sources from sourceforge:

```
$ svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/share/fandango/trunk/
↪fandango fandango
```

Features

Most of submodules provide some usage recipes, see github:

This library provides submodules with utilities for PyTango device servers and applications written in python:

1. functional: functional programming, data format conversions, caseless regular expressions,
2. tango: tango api helper methods, search/modify using regular expressions,
3. dynamic attributes/states/commands and online python code evaluation, see the github,
4. server: Astor-like python API,
5. device: some templates for Tango device servers, TangoEval for fast "tango code" evaluation,
6. interface: device server inheritance,
7. db: MySQL access,
8. dicts,arrays: advanced containers, sorted/caseless list/dictionaries, .csv parsing,
9. log: logging,
10. objects: object templates, singletones, structs,
11. threads: serialized hardware access, multiprocessing,
12. linos: some linux tricks,
13. web: html parsing,
14. qt: some custom Qt classes, including worker-like threads.

Main Classes

- DynamicDS / DynamicAttributes,
- ServersDict,
- TangoEval,
- ThreadDict / SingletoneWorker,
- TangoInterfaces (FullTangoInheritance).

Where it us used

Several PyTango APIs and device servers use Fandango modules:

1. PyTangoArchiving,
2. PyPLC,
3. SplitterBoxDS,
4. PyStateComposer,
5. SimulatorDS / PySignalSimulator,
6. PANIC / PyAlarm,
7. CSVReader.

Requirements

1. It requires PyTango to access Tango,
2. It requires Taurus to use Tango Events,
3. Some submodules have its own dependencies (Qt,MySQL), so they are always imported within try, except clauses.

Recipes

Get devices or attributes matching a regular expression

Using fandango.tango.get_matching_devices or get_matching_attributes:

```
from fandango import tango
tango.get_matching_devices('sr[0-9]+/vc/(ipct|vgct)*')
    ['SR01/VC/IPCT-01A08-01',
     'SR01/VC/IPCT-01A08-02',
     'SR01/VC/IPCT-02A01-01',
     'SR01/VC/VGCT-01A08-01',
     'SR01/VC/VGCT-02A01-01',
     'SR02/VC/IPCT-02A02-01',
    ]
```

Search for device attribute/properties matching a regular expression

```
fandango.tango.get_matching_device_properties('s01/*/*ct*','serial*')
{'S01/VC/IPCT-01': {'SerialLine': 'S01/VC/SERIAL-01'},
 'S01/VC/IPCT-02': {'SerialLine': 'S01/VC/SERIAL-02'},
 'S01/VC/VGCT-01': {'SerialLine': 'S01/VC/SERIAL-10'}}
```

Obtain all information from a device

```
In [59]: fandango.tango.get_device_info('sr/vc/gll')
Out[59]: fandango.Struct({
        'name': sr/vc/gll,
        'level': 4,
        'started': 11th February 2013 at 13:07:37,
        'PID': 11024,
        'ior': ...,
        'server': PyStateComposer/SR_VC,
        'host': nanana01,
        'stopped': 11th February 2013 at 12:49:49,
        'exported': 1,        })
```

servers.ServersDict: the Astor-like python API

fandango.ServersDict is a dictionary of TServer classes indexed by server/instance names and loaded using wildcard expressions.

Provides Jive/Astor functionality to a list of servers and allows to **select/start/stop** them by host, class or devices It's purpose is to allow generic start/stop of lists of Tango DeviceServers. This methods of selection provide new ways of search apart of Jive-like selection.

```
from fandango import Astor
astor = Astor()
astor.load_by_name('snap*')
astor.keys()
    ['snapmanager/1', 'snaparchiver/1', 'snapextractor/1']

server = astor['snaparchiver/1']
server.get_device_list()
    ['dserver/snaparchiver/1', 'archiving/snaparchiver/1']

astor.states()
server.get_all_states()
    dserver/snaparchiver/1: ON
    archiving/snaparchiver/1: ON

astor.get_device_host('archiving/snaparchiver/1')
    palantir01

astor.stop_servers('snaparchiver/1')
astor.stop_all_servers()
astor.start_servers('snaparchiver/1','palantir01',wait=1000)
astor.set_server_level('snaparchiver/1','palantir01',4)

#Setting the polling of a device:
server = astor['PySignalSimulator/bl11']
for dev_name in server.get_device_list():
    dev = server.get_device(dev_name)
    attrs = dev.get_attribute_list()
    [dev.poll_attribute(attr,3000) for attr in attrs]
```

Start/Stop all device servers in a machine (like Astor -> Stop All)

Stopping

```
import fandango
fandango.Astor(hosts=['my.host']).stop_all_servers()
```

and the other way round ...

```
astor = fandango.Astor(hosts=['my.host'])
astor.start_all_servers()
```

if you just want to see if things are effectively running or not:

```
astor.states()
```

Implement full (attibutes+properties) inheritance between PyTango classes

Just inheriting from a Device Server does not automatically updates all properties and attributes from the parent. The fandango.interface module enables that functionality using FullTangoInheritance function.

To use it you have to **add 3 lines in the "__main__"** part of your python file (and at the end of the file, if you want to further continue inheriting between classes):

```
#Replace <YourDevice> and <ParentDevice> with your Device classes names

if __name__ == '__main__':
    try:
        py = PyTango.Util(sys.argv)

        # Adding DeviceServer Inheritance, added here to be not overwritten by Pogo
        from fandango.interface import FullTangoInheritance
        from <ParentDevice> import <ParentDevice>,<ParentDevice>Class
        <YourDevice>,<YourDevice>Class = \
            FullTangoInheritance('<YourDevice>',<YourDevice>,<YourDevice>Class, \
                                 <ParentDevice>,<ParentDevice>Class,ForceDevImpl=True)

        py.add_TgClass(<YourDevice>Class,<YourDevice>,'<YourDevice>')
        U = PyTango.Util.instance()
        U.server_init()
        U.server_run()

    except PyTango.DevFailed,e:
        print '-------> Received a DevFailed exception:',e
    except Exception,e:
        print '-------> An unforeseen exception occured....',e

# Adding DeviceServer Inheritance (to be visible by subclasses)
from fandango.interface import FullTangoInheritance
from <ParentDevice> import <ParentDevice>,<ParentDevice>Class
<YourDevice>,<YourDevice>Class = FullTangoInheritance('<YourDevice>', <YourDevice>,
↪<YourDevice>Class, <ParentDevice>, <ParentDevice>Class, ForceDevImpl=True)
```

dynamic.DynamicDS: template for Dynamic Attributes

DynamicAttributes are using DynamicDS template.

Use TangoEval to evaluate strings containing Tango Attributes

TangoEval class provides PyAlarm-like evaluation of strings containing attribute names (replacing them by its values). It is part of **fandango.device** module. The result of each evaluation is stored in te.result.

```
In [14]: from fandango import TangoEval
In [15]: te = TangoEval('(s01/vc/gauge-01/pressure + s01/vc/gauge-01/pressure) / 2.')
Out[15]: TangoEval: result = 7.2e-10
```

Use CSVArray to turn a .csv into a dictionary

```
cat tmp/tree_test.csv
A       B       2
        C       3
```

```
In [16]: csv = fandango.arrays.CSVArray('tmp/tree_test.csv')
In [17]: csv.expandAll()
In [18]: csv.getAsTree(lastbranch=1)
Out[18]: {'A': {'B': ['2'], 'C': ['3']}}
```

Fast property update

```
import fandango.functional as fun
servers = fandango.Astor('PyAlarm/*')
8 : devs = [d for d in fun.chain(*[servers[s].get_device_list() for s,v in servers.
↪states().items() if v is not None]) if not d.startswith('dserver')]
for d in devs:
    prop = servers.proxies[d].get_property(['AlarmReceivers'])['AlarmReceivers']
    servers.proxies[d].put_property({'AlarmReceivers':[s.replace('%SRUBIO','%DFERNANDEZ
↪') for s in prop]})
for d in devs: servers.proxies[d].ReloadFromDB()
```

ReversibleDict

```
In [133]: ch = fandango.dicts.ReversibleDict()

In [134]: ch.update([(unichr(ord('a')+i),i,unichr(ord('A')+i)) for i in range(26)])

In [135]: ch
Out[135]:
(u'a', 0, u'A')
(u'b', 1, u'B')
```

(continues on next page)

(continued from previous page)

```
(u'c', 2, u'C')
(u'd', 3, u'D')
...

In [136]: ch['a']
Out[136]: (0, u'A')

In [137]: ch['A']
Out[137]: (0, u'a')

In [138]: ch['a'].keys()
Out[138]: set([0])

In [139]: ch['A'].keys()
```

ThreadDict

from PyPLC

```
def initThreadDict(self):
    def read_method(args,comm=self.Regs,log=self.debug): #It takes a key with commas and splits it to have a list of arguments
        try:
            log('>'*20 + ' In ThreadDict.read_method(%s)' % args)
            args = [int(s) for s in args.split(',')[:2]]
            return comm(args,asynch=True)
        except PyTango.DevFailed,e:
            print 'Exception in ThreadDict.read_method!!!'
            print str(e).replace('\n','')[:100]
        except Exception,e:
            print '#'*80
            print 'Exception in ThreadDict.read_method!!!'
            print traceback.format_exc()
            print '#'*80
            return [] ## Arrays must not be readable if communication doesn't work!!!!

    self.threadDict = fandango.ThreadDict(
        read_method = read_method,
        trace=True)
    self.threadDict.set_timewait(max(0.1,self.ModbusTimeWait/1000.))

    self.info('Mapped Arrays are: %s' % self.MapDict)

    for var,maps in self.MapDict.items():
        regs = self.GetCommands4Map(maps)
        for reg in regs:
            vals = ','.join(str(r) for r in reg)
            self.debug('Adding %s(%s) as ThreadDict[%s]' % (var,reg,vals))
            self.threadDict.append(vals,[])#period=[]) #append(key,value='',period=3000)

    self.threadDict.start()
```

(continues on next page)

(continued from previous page)

```
    self.info('out of PyPLC.initThreadDict()')

Pa leeer .........

    for reg in regs:
        key = ','.join(str(r) for r in reg)
        val = self.threadDict[key]
```

Piped, iPiped, zPiped interfaces

Fandango will have now a new set of operators to use regular-or operator (**'|'**) like a linux pipe between operators (inspired by Maxim Krikun - Shell-like data processing).**

```
cat('filename') | grep('myname') | printlines
```

Using fandango:

```
from fandango.functional import *

v | iPiped(rd.get_attribute_values,start_date='2012-07-10',stop_date='2012-07-17') |␣
↪iPiped(PyTangoArchiving.utils.decimate) | zPiped(time2str) | plist

#equals to:

[(time2str(v[0]),v[1]) for v in PyTangoArchiving.utils.decimate(rd.get_Attribute_
↪values(v,start_date='2012-07-10',stop_date='2012-07-17'))]
```

Available interfaces are:

```
class Piped:
    """This class gives a "Pipeable" interface to a python method:
        cat | Piped(method,args) | Piped(list)
        list(method(args,cat))
    """
    ...

class iPiped:
    """ Used to pipe methods that already return iterators
    e.g.: hdb.keys() | iPiped(filter,partial(fandango.inCl,'elotech')) | plist
    """
    ...

class zPiped:
    """
    Returns a callable that applies elements of a list of tuples to a set of functions
    e.g. [(1,2),(3,0)] | zPiped(str,bool) | plist => [('1',True),('3',False)]
    """
    ...
```

Available operators are:

```
pgrep = lambda exp: iPiped(lambda input: (x for x in input if inCl(exp,x)))
pmatch = lambda exp: iPiped(lambda input: (x for x in input if matchCl(exp,str(x))))
pfilter = lambda meth=bool,*args: iPiped(filter,partial(meth,*args))
ppass = Piped(lambda x:x)
plist = iPiped(list)
psorted = iPiped(sorted)
pdict = iPiped(dict)
ptuple = iPiped(tuple)
pindex = lambda i: Piped(lambda x:x[i])
pslice = lambda i,j: Piped(lambda x:x[i,j])
penum = iPiped(lambda input: izip(count(),input) )
pzip = iPiped(lambda i:izip(*i))
ptext = iPiped(lambda input: '\n'.join(imap(str,input)))
```

9.2.9 How to write your first Device Class

Intended audience: developers, ***Programming language:*** all

This HowTo explains how to create your first Device Class regardless of the language used.

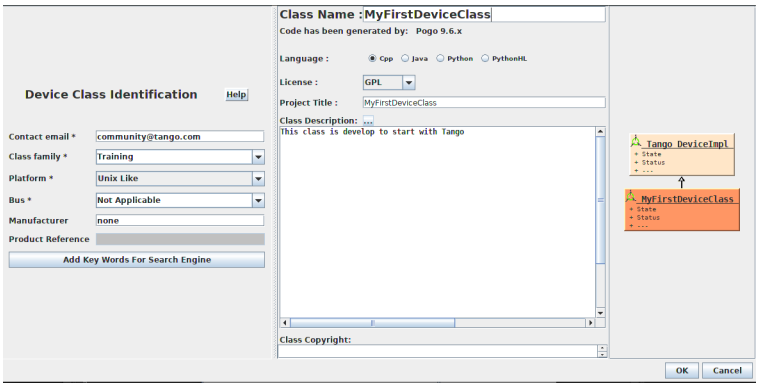
Prerequisite : Tango Environment.

Start Pogo Generator

Now you can create a new Class. Click on File and New.

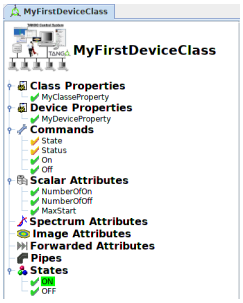
Fill in your Class

Fill in required informations. The Device Class identification on the left part and the description on the right.



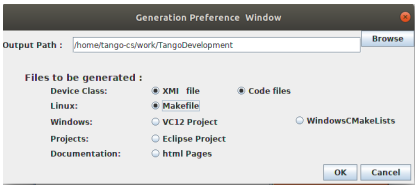
You will see an empty Pogo interface.

You can add some Properties, Commands and Attributes, by double-clicking on each one. There are different possible configuraton.

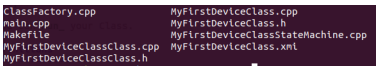


Generate

Generate your files Choose your folder's path and files you want to create. For example, on linux OS with Cpp language, the minimum is XMI File, Code files and Makefile like you can see behind.



You can now see **Pogo**'s files in your folder. **Pogo** had create skeleton files with your Properties, Commands and Attributes.



Now you have a basic server who's make nothing. You can provide contents or fill it with different needs.

You can have some informations about how to start *here*

You can have more informations about device server usage *here*.

How to fill your device in *Cpp*, *Java* and *Python*

Compile And Run

Now you can compile and *Run* your Class.

9.2.10 How to import multiple device classes to the Catalogue

Intended audience: administrators

The Device Classes Catalogue is available on the Tango Controls web page: http://www.tango-controls.org/.

To import multiple device classes to the catalogue from a repository please use an import script.

Requirements

- Python2.7
- Subversion
- SVN python library: **pip install svn**

> **Warning:** The library will not work if any of files in the SVN has no author defined which is the case for tango-ds repository on the Sourceforge. To avoid problems one can edit `svn/common.py` around line 358 to have something like the following:
>
> ```
> author = ''
> if commit_node.find('author') is not None:
>     author = commit_node.find('author').text
> ```

- urllib2 library if it is not installed with Python itself: **pip install urllib2**
- Requests library version >= 2.12, you may need to run pip with –upgrade option: **pip install --upgrade Requests**

How-to import multiple classes

1. Install packages listed in *Requirements* if not yet installed
2. **Create a folder for a local copy of a repository:**
   - example: **mkdir ~/tmp/local-repo**
3. **Clone the import utility with git:**
   - **git clone https://github.com/piogor/dsc-import.git**
4. **Get into cloned sources:**
   - **cd dsc-import**
5. **Make your local branch to be sure your settings will not be overwritten by someone else.**
   - **git checkout -b my_local_branch**
6. Update variables in a file `dsc_import_utility.py` to reflect your environment:

```
FORCE_UPDATE = False  # when True no timestamps are checked and updates are␣
↪performed
TEST_SERVER_AUTH = False  # Set true if script is run against test server␣
↪with additional authentication (webu test)
VERIFY_CERT = False  # set this to false if running aginst test server␣
↪without a valid certificate
USE_DOC_FOR_NON_XMI = True # when True, parse documentation to get xmi␣
↪conntent for device servers without XMI
ADD_LINK_TO_DOCUMENTATION = True # when True it provides a link to␣
↪documentation

# set the following variables to point to the repositories
LOCAL_REPO_PATH = '/home/piotr/tmp/tango-ds-repo/'  # local copy of the␣
↪repository will be synced there
```

(continues on next page)

(continued from previous page)

```
LOG_PATH = '/home/piotr/tmp'  # where to log some information about import␣
↪process, not used now.

REMOTE_REPO_HOST = 'svn.code.sf.net'  # host of the SVN repository
REMOTE_REPO_PATH = 'p/tango-ds/code'  # path within the server where the␣
↪repository is located

# if one would like to limit a search tree (useful for one device server␣
↪update and/or tests)
REPO_START_PATH = 'DeviceClasses'  # do not provide start nor end slashes

# Tango Controls or test server address
SERVER_BASE_URL = 'http://www.tango-controls.org/'
```

7. run with a command: **python dsc_import_utility.py**

> **Note:** It will ask you for your credentials for *tango-controls.org* and import/update device classes using provided account.

How the script works

It does import in the following way:

- It makes a local copy (in path defined by *LOCAL_REPO_PATH*) of a SVN repository to speed up search procedure.
- Then it searches the local copy for folders containing .XMI files. It takes into account the standard *branches/tags/trunk* structure. The folders where it findes .xmi files or a proper structure are listed as candidates to be device servers.
- Then, the list of candidates then is processed and compared (by repository URL) with content in the Device Classes Catalogue.
  - If there are changes or *FORCE_UPDATE* is True the catalogue is updated
    - For device server without .XMI file it looks for documentation server and tries to parse html documentation generated by **Pogo**.
  - If there ara no changes the device server is skipped

These information you will find in a `README.rst` (https://github.com/piogor/dsc-import/blob/master/README.rst).

9.2.11 Integrate Java Tango servers with Astor

Intended audience: developers, administrators, ***Programming language:*** java

Problem overview

One wants to control Java Tango servers using Astor after deployment.

Detailed cases

Simplify deployment and integration of new Java Tango servers with existing Tango environment. Give users ability to use well known tools for controlling and monitoring Java Tango servers.

Solution overview

The following describes solutions for linux. Windows users may use the same strategy except bash files must be replaced with corresponding batch files.

There are two possible ways we can prepare this receipt:

1. Generate fat jar, i.e. executable jar that contains everything.

```
<!-pom.xml ->
...
<build>
        <plugins>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <!-assemble executable jar with all dependencies ->
                <configuration>
                    <appendAssemblyId>false</appendAssemblyId>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <mainClass>hzg.wpn.tango.TestServer</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
...
```

Executing mvn assembly:single produces one big jar file in the target folder, i.e. TestServer-1.4.jar.

To use our server we just copy it to some location on the target machine and use the following bash script:

```
#!/bin/bash

/usr/bin/java -jar /absolute/path/to/our/jar $1 hzg.wpn.tango.TestServer $1
```

Super easy, isn't it? (Do not mind the last two parameters they are here to workaround this issue)

This script is saved into /usr/lib/tango/server/TestServer. /usr/lib/tango/server can be replaced with any other location where Starter can find the script, i.e. defined in StartDsPath property.

We need to specify an absolute path to the jar file as Astor runs servers from /var/tmp/ds.log folder

PROS

- it is much easier to deal with immutable artifacts

CONS

- fat jar… imagine 1000 servers each requires 17MB

2. Use exploded assembly, i.e. first package everything into tar.gz

```
<!-pom.xml ->
<build>
        <plugins>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <appendAssemblyId>false</appendAssemblyId>
                    <descriptors>
                        <descriptor>src/main/assembly.xml</descriptor>
                    </descriptors>
                </configuration>
            </plugin>
        </plugins>
    </build>
<!-src/main/assembly.xml ->
<assembly schemaLocation="http://maven.apache.org/xsd/assembly-1.0.0.xsd">
    <id>distr</id>
    <formats>
        <format>tar.gz</format>
    </formats>
    <dependencySets>
        <dependencySet>
            <outputDirectory>/lib</outputDirectory>
            <fileMode>0777</fileMode>
        </dependencySet>
    </dependencySets>
    <fileSets>
        <fileSet>
            <directory>conf</directory>
            <outputDirectory>/conf</outputDirectory>
        </fileSet>
    </fileSets>
</assembly>
```

When deployed it is extracted from the archive and copied to some placed aka SERVER_ROOT. Exploded archive has the following structure:

```
SERVER_ROOT\
           |- lib\
           |      |- *.jar
           |
           |- conf\
                  |- test.xml
```

Startup bash script may look like this:

```
#!/bin/bash

# import essential environmental variables like absolute path to our server root aka SERVER_ROOT
. /etc/tangorc

/usr/bin/java -cp $SERVER_ROOT/lib/* -Dconf=$SERVER_ROOT/conf/test.xml hzg.wpn.tango.TestServer
```

Again the script is in /usr/lib/tango/server/TestServer. /usr/lib/tango/server can be replaced with any other location where Starter can find the script, i.e. defined in StartDsPath property.

We need to specify an absolute path to the lib and conf folders as Astor runs servers from /var/tmp/ds.log folder

PROS

- if there are several servers common dependencies can be placed into a single location, hence safe some hdd space - server may use external resources (like conf in the example above), just make sure to use absolute pathes

CONS

- dealing with exploded assemblies quickly becomes messy

Both solutions assume that maven is used to handle project's lifecycle.

9.2.12 Tango with systemd integration

Intended audience: administrators

This recipe shows how to set up Tango environment using systemd, for instance on a developer's box

At first systemd units must be defined for common Tango device servers (db, tango-accesscontrol, tango-test etc):

```
# /etc/systemd/system/tango-db.service
[Unit]
Description = Tango DB
Requires=mysql.service
After=mysql.service
StopWhenUnneeded=true

[Service]
User=tango
Environment=TANGO_HOST=localhost:10000
ExecStart=/opt/tango-9.2.2/bin/DataBaseds 2 -ORBendPoint giop:tcp::10000

[Install]
WantedBy=tango.target
```

```
# /ect/systemd/system/tango-accesscontrol.timer
[Timer]
OnActiveSec=3

[Install]
WantedBy=tango.target
```

timer is needed because we have to wait before database is ready to accept requests

```
# /etc/systemd/system/tango-accesscontrol.service
[Unit]
Description=TangoAccessControl device server
Wants=tango-db
After=tango-db
StopWhenUnneeded=true

[Service]
Environment=TANGO_HOST=localhost:10000
ExecStart=/opt/tango-9.2.2/bin/TangoAccessControl 1
```

A service unit for Starter device server can also be defined:

```
# /etc/systemd/system/tango-starter.service
[Unit]
Description=Starter device server
After=tango-accesscontrol.service
Requires=tango-db.service
Requires=tango-accesscontrol.service

[Service]
Restart=always
RestartSec=10
User=tango
Environment=TANGO_HOST=localhost:10000
ExecStart=/usr/local/bin/Starter tangobox
```

Finally combine everything into a single target:

```
# /etc/systemd/system/tango.target
[Unit]
Description=Tango development environment target
Requires=tango-db.service
Requires=tango-starter.service
Requires=tango-accesscontrol.timer
Requires=tango-test.timer

[Install]
#WantedBy=multi-user.target
```

Then these entities must be enabled and started:

```
$> sudo systemctl enable {each timer and service}
$> sudo systemctl start tango.target
```

systemd unit files may be generated when the device server is built. For instance as it is done in our tango-maven-archetype

Defining Tango servers as systemd units can be also very useful for production.

9.2.13 How to PyTango

Intended audience: developers, ***Programming language:*** python

My list of short recipes for common tasks ... please check official documentation first!

Before anything else

```
import PyTango
```

Installation notes

The new PyTango, is now available to download from the Tango download page or PiPy

If you have already installed PyTango with *pip* you can simply update your PyTango version by doing:

```
$ pip -U PyTango
```

```
$ sudo apt-get update python-pytango
```

The documentation is available at official readthedocs website.

If you encounter problems installing or running this release, please report them back to the tango mailing list.

You can check out this version:

```
$ svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/PyTango/tags/Release_7_1_
↪3 PyTango-7.1.3`
```

You can check out the latest version:

```
$ svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/PyTango/trunk PyTango-
↪latest`
```

Nice PyTango dancing

Using the DeviceProxy object

Getting the polling buffer values

Only for polled attributes we can get the last N read values. the polling buffer depth is managed by the admin device.

```
dp = PyTango.DeviceProxy('some/tango/device')
dp.attribute_history('cpustatus',10)
```

Get/Set polled attributes

```
def get_polled_attributes(dev_name):
    dp = PyTango.DeviceProxy(dev_name)
    attrs = dp.get_attribute_list()
    periods = [(a,dp.get_attribute_poll_period(a)) for a in attrs]
    return dict((a,p) for a,p in periods if p)

[plc4.poll_attribute(a,5000) for k,v in periods if v]
```

Modify the polling of attributes

```
 import re,PyTango
 period = 10000
 devs = PyTango.Database().get_device_exported('some/tango/devices*')
 for dev in devs:
     dp = PyTango.DeviceProxy(dev)
     attrs = sorted([a for a in dp.get_attribute_list() if re.match(
↪'(Output|Temperature)_[0-9]$',a)])
     [dp.poll_attribute(a,period) for a in attrs]
     print('\n'.join(dp.polling_status()))
```

Events

Creating an event callback

```
# The callback must be a callable or an object with a push_event(self,event) method
```

Configuring an event

```
#From the client side
#subscribe_event(attr_name, event_type, cb_or_queuesize, filters=[], stateless=False,␣
↪extract_as=PyTango._PyTango.ExtractAs.Numpy)
event_id = PyTango.DeviceProxy.subscribe_event(attributeName,PyTango.EventType.CHANGE_
↪EVENT,callback_function,[],True)

#From inside the device server
self.set_change_event('State',True,True)
```

Device Server Internal Objects

Forcing in which host the device is exported

This environment variable must be set before launching the device:

```
$ export OMNIORB_USEHOSTNAME=10.0.0.10
```

Creating a Device Server from ipython

Having defined your device in MyDS.py:

```
from MyDS import *
py = PyTango.PyUtil(['MyDS.py','InstanceName'])
py.add_TgClass(MyDSClass,MyDS,'MyDS')
U = PyTango.Util.instance()
U.server_init()
U.server_run()
```

Get the device server admin

NOT TESTED

```
U = PyTango.Util.instance()
U.get_dserver_device()
```

Modify internal polling

Note: It doesn't work at *init_device()*; must be done later on in a *hook* method.

```
U = PyTango.Util.instance()
admin = U.get_dserver_device()
dir(admin)
    [
        StartPolling
        StopPolling
        AddObjPolling
        RemObjPolling
        UpdObjPollingPeriod
        DevPollStatus
        PolledDevice
    ]

polled_attrs = {}
for st in admin.DevPollStatus(name):
    lines = st.split('\n')
    try: polled_attrs[lines[0].split()[-1]]=lines[1].split()[-1]
    except: pass

type_ = 'command' or 'attribute'
for aname in args:
if aname in polled_attrs:
    admin.UpdObjPollingPeriod([[200],[name,type_,aname]])
else:
    admin.AddObjPolling([[3000],[name,type_,aname]])
```

Get all polling attributes

The polling of the attributes is recorded in the **property_device** table of the tango database in the format of a list like *[ATTR1,PERIOD1,ATTR2,PERIOD2,...]*

The list of polled attributes can be accessed using this method of admin device:

```
dp = PyTango.DeviceProxy('dserver/myServerClass/id22')
polled_attrs = [a.split('\n')[0].split(' ')[-1] for a in dp.DevPollStatus('domain/family/
↪member-01')]
```

Get the device class object from the device itself

```
self.get_device_class()
```

Get the devices inside a Device Server

```
def get_devs_in_server(self,MyClass=None):
    """
    Method for getting a dictionary with all the devices running in this server
    """
    MyClass = MyClass or type(self) or DynamicDS
    if not hasattr(MyClass,'_devs_in_server'):
        MyClass._devs_in_server = {} #This dict will keep an access to the class␣
↪objects instantiated in this Tango server
    if not MyClass._devs_in_server:
        U = PyTango.Util.instance()
        for klass in U.get_class_list():
            for dev in U.get_device_list_by_class(klass.get_name()):
                if isinstance(dev,DynamicDS):
                    MyClass._devs_in_server[dev.get_name()]=dev
    return MyClass._devs_in_server
```

Identify each attribute inside read_attr_hardware()

```
def read_attr_hardware(self,data):
    self.debug("In DynDS::read_attr_hardware()")
    try:
        attrs = self.get_device_attr()
        for d in data:
            a_name = attrs.get_attr_by_ind(d).get_name()
            if a_name in self.dyn_attrs:
                self.lock.acquire() #This lock will be released at the end of read_dyn_
↪attr
                self.myClass.DynDev=self #VITAL: It tells the admin class which device␣
↪attributes are going to be read
                self.lock_acquired += 1
        self.debug('DynamicDS::read_attr_hardware(): lock acquired %d times'%self.lock_
```

(continues on next page)

(continued from previous page)

```
↪acquired)
    except Exception,e:
        self.last_state_exception = 'Exception in read_attr_hardware: %s'%str(e)
        self.error('Exception in read_attr_hardware: %s'%str(e))
```

Device server logging (using Tango logs)

```
$ Device_4Impl.
$
$ debug_stream ( str )
$ info_stream ( str )
$ warning_stream ( str )
$ error_stream ( str )
$ fatal_stream ( str )
```

Or use fandango.Logger object instead

Adding dynamic attributes to a device

```
self.add_attribute(
    PyTango.Attr( #or PyTango.SpectrumAttr
        new_attr_name,PyTango.DevArg.DevState,PyTango.AttrWriteType.READ, #or READ_WRITE
        #max_size or dyntype.dimx #If Spectrum
        ),
    self.read_new_attribute, #(attr)
    None, #self.write_new_attribute #(attr)
    self.is_new_attribute_allowed, #(request_type)
    )
```

Using Database Object

```
import PyTango
db = PyTango.Database()
```

Register a new device server

```
dev = 'SR%02d/VC/ALL'%sector
klass = 'PyStateComposer'
server = klass+'/'+dev.replace('/','_')

di = PyTango.DbDevInfo()
di.name,di._class,di.server = device,klass,server
db.add_device(di)
```

Remove "empty" servers from database

```
tango = PyTango.Database()
[tango.delete_server(s)
    for s in tango.get_server_list()
    if all(d.lower().startswith('dserver') for d in tango.get_device_class_list(s))
]
```

Force unexport of a failing server

You can check using db object if a device is still exported after killed

```
$ bool(db.import_device('dserver/HdbArchiver/11').exported)
$ True
```

You can unexport this device or server with the following call:

```
db.unexport_server('HdbArchiver/11')
```

It would normally allow you to restart the server again.

Get all servers of a given class

```
class_name = 'Modbus'
list_of_names = ['/'.join((class_name,name)) for name in db.get_instance_name_
↪list(class_name)]
```

Differences between DB methods:

```
get_instance_name_list(exec_name): return names of **instances**
get_server_list(): returns list of all **executable/instance**
get_server_name_list(): return names of all **executables**
```

Get all devices of a server or a given class

The command is:

```
db.get_device_class_list(server_name): return
['device/name/family','device_class']*num_of_devs_in_server
```

The list returned includes the admin server (*dserver/exec_name/instance*) that must be pruned from the result:

```
list_of_devs = [dev for dev in db.get_device_class_list(server_name) if '/' in dev and␣
↪not dev.startswith('dserver')]
```

Get all devices of a given class from the database

```
import operator
list_of_devs = reduce(operator.add,(list(dev for dev in db.get_device_class_list(n) \
    if '/' in dev and not dev.startswith('dserver')) for n in \
    ('/'.join((class_name,instance)) for instance in db.get_instance_name_list(class_
↪name)) \
    ))
```

Get property values for a list of devices

```
db.get_device_property_list(device_name,'*') : returns list of
available properties
db.get_device_property(device_name,[property_name]) : return
{property_name : value}
```

```
prop_names = db.get_device_property_list(device_name)
    ['property1','property2']
dev_props = db.get_device_property(device_name,prop_names)
    {'property1':'first_value' , 'property2':'second_value' }
```

Get the history (last ten values) of a property

```
[ph.get_value().value_string for ph in tango.get_device_property_history('some/alarms/
↪device','AlarmsList')]

[['MyAlarm:a/gauge/controller/Pressure>1e-05', 'TempAlarm:a/nice/device/Temperature_Max␣
↪> 130'],
```

Get the server for a given device

```
>>> print db.get_server_list('Databaseds/*')
['DataBaseds/2']
>>> print db.get_device_name('DataBaseds/2','DataBase')
['sys/database/2']
>>> db_dev=PyTango.DeviceProxy('sys/database/2')
>>> print db_dev.command_inout('DbImportDevice','et/wintest/01')
([0, 2052], ['et/wintest/01', 'IOR:010000017000xxxxxx', '4',
'WinTest/manu', 'PCTAUREL.esrf.fr', 'WinTest'])
```

Get the Info of a not running device (exported, host, server)

```
def get_device_info(dev):
    vals = PyTango.DeviceProxy('sys/database/2').DbGetDeviceInfo(dev)
    di = dict((k,v) for k,v in zip(('name','ior','level','server','host','started',
↪'stopped'),vals[1]))
    di['exported'],di['PID'] = vals[0]
    return di
```

Set property values for a list of devices

Attention , Tango property values are always inserted as lists! {property_name : **[** property_value **]**}

```
prop_name,prop_value = 'Prop1','Value1'
[db.put_device_property(dev,{prop_name:[prop_value]}) for dev in list_of_devs]
```

Get Starter Level configuration for a list of servers

```
[(si.name,si.mode,si.level) for si in [db.get_server_info(s) for s in list_of_servers]]
```

Set Memorized Value for an Attribute

```
db.get_device_attribute_property('tcoutinho/serial/01/Baudrate',['__value'])
db.put_device_attribute_property('tcoutinho/serial/01/Baudrate',{'__value':VALUE})
```

Useful constants and enums

```
In [31]:PyTango.ArgType.values
Out[31]:
{0: PyTango._PyTango.ArgType.DevVoid,
 1: PyTango._PyTango.ArgType.DevBoolean,
 2: PyTango._PyTango.ArgType.DevShort,
 3: PyTango._PyTango.ArgType.DevLong,
 4: PyTango._PyTango.ArgType.DevFloat,
 5: PyTango._PyTango.ArgType.DevDouble,
 6: PyTango._PyTango.ArgType.DevUShort,
 7: PyTango._PyTango.ArgType.DevULong,
 8: PyTango._PyTango.ArgType.DevString,
 9: PyTango._PyTango.ArgType.DevVarCharArray,
 10: PyTango._PyTango.ArgType.DevVarShortArray,
 11: PyTango._PyTango.ArgType.DevVarLongArray,
 12: PyTango._PyTango.ArgType.DevVarFloatArray,
 13: PyTango._PyTango.ArgType.DevVarDoubleArray,
 14: PyTango._PyTango.ArgType.DevVarUShortArray,
 15: PyTango._PyTango.ArgType.DevVarULongArray,
 16: PyTango._PyTango.ArgType.DevVarStringArray,
```

(continues on next page)

(continued from previous page)

```
 17: PyTango._PyTango.ArgType.DevVarLongStringArray,
 18: PyTango._PyTango.ArgType.DevVarDoubleStringArray,
 19: PyTango._PyTango.ArgType.DevState,
 20: PyTango._PyTango.ArgType.ConstDevString,
 21: PyTango._PyTango.ArgType.DevVarBooleanArray,
 22: PyTango._PyTango.ArgType.DevUChar,
 23: PyTango._PyTango.ArgType.DevLong64,
 24: PyTango._PyTango.ArgType.DevULong64,
 25: PyTango._PyTango.ArgType.DevVarLong64Array,
 26: PyTango._PyTango.ArgType.DevVarULong64Array}

In [30]:PyTango.AttrWriteType.values
Out[30]:
{0: PyTango._PyTango.AttrWriteType.READ,
 1: PyTango._PyTango.AttrWriteType.READ_WITH_WRITE,
 2: PyTango._PyTango.AttrWriteType.WRITE,
 3: PyTango._PyTango.AttrWriteType.READ_WRITE}

In [29]:PyTango.AttrWriteType.values[3] is PyTango.READ_WRITE
Out[29]:True
```

Using Tango Groups

This example uses PyTangoGroup to read the status of all devices in a Device Server

```
import PyTango

server_name = 'VacuumController/AssemblyArea'
group = PyTango.Group(server_name)
devs = [d for d in PyTango.Database().get_device_class_list(server_name) if '/' in d␣
↪and 'dserver' not in d]
for d in devs:
    group.add(d)

answers = group.command_inout('Status',[])
for reply in answers:
    print 'Device %s Status is:' % reply.dev_name()
    print reply.get_data()
```

About Exceptions

Be aware that I'm not sure about all of this:

```
try:
    #reason,desc(ription),origin
    PyTango.Except.throw_exception("TimeWAITBetweenRetries",
                "Last communication failed at %s, waiting %s millis"%(time.ctime(self.
↪last_failed),self.ErrorTimeWait),
                inspect.currentframe().f_code.co_name)
except PyTango.DevFailed,e:
```

(continues on next page)

(continued from previous page)

```
    if e.args[0]['reason']!='API_AsynReplyNotArrived':
        PyTango.Except.re_throw_exception(e,"DevFailed Exception",str(e),inspect.
↪currentframe().f_code.co_name)
```

Passing Arguments to Device command_inout

When type of Arguments is *special* like **DevVarLongStringArray** the introduction of arguments is something like:

```
api.manager.command_inout('UpdateSnapComment',[[40],['provant,provant...']])
```

Using asynchronous commands

```
cid = self.modbus.command_inout_asynch(command,arr_argin)
while True:
    self.debug('Waiting for asynchronous answer ...')
    threading.Event().wait(0.1)
    #time.sleep(0.1)
    try:
        result = self.modbus.command_inout_reply(cid)
        self.debug('Received: %s' % result)
        break
    except PyTango.DevFailed,e:
        self.debug('Received DevFailed: %s' %e)
        if e.args[0]['reason'] != 'API_AsynReplyNotArrived':
           raise Exception,'Weird exception received!: %s' % e
```

Setting Attribute Config

```
for server in astor.values():
    for dev in server.get_device_list():
        dp = server.get_proxy(dev)
        attrs = dp.get_attribute_list()
        if dev.rsplit('/')[-1].lower() not in [a.lower() for a in attrs]: continue
        conf = dp.get_attribute_config(dev.rsplit('/')[-1])
        conf.format = "%1.1e"
        conf.unit = "mbar"
        conf.label = "%s-Pressure"%dev
        print 'setting config for %s/%s' % (dev,conf.name)
        dp.set_attribute_config(conf)
```

Porting device servers to PyTango

The changes to easily port PyTango devices are:

- C++ : Replace Device_3Impl with Device_4Impl
- Python : Replace Device_3Impl with Device_4Impl, PyDeviceClass with DeviceClass and PyUtil with Util.

If you are quite lazy you can add this at the beginning of your *$Class.py* file (and be still parseable by Pogo):

```
import PyTango
if 'PyUtil' not in dir(PyTango):
    PyTango.Device_3Impl = PyTango.Device_4Impl
    PyTango.PyDeviceClass = PyTango.DeviceClass
    PyTango.PyUtil = PyTango.Util
```

Simplify changes by adding this line

```
if 'PyUtil' not in dir(PyTango):
PyTango.PyDeviceClass = PyTango.DeviceClass
PyTango.PyUtil = PyTango.Util
```

9.2.14 PyTangoArchiving Recipes

Intended audience: developers, ***Programming language:*** python

by Sergi Rubio

PyTangoArchiving is the python API for Tango Archiving.

This package allows to:

- Integrate Hdb and Snap archiving with other python/PyTango tools.
- Start/Stop Archiving devices in the appropiated order.
- Increase the capabilities of configuration and diagnostic.
- Import/Export .csv and .xml files between the archiving and the database.

Don't edit this wiki directly, the source for this documentation is available at PyTangoArchiving UserGuide

Installing PyTangoArchiving:

Repository is available on sourceforge:

```
$ svn co https://svn.code.sf.net/p/tango-cs/code/archiving/tool/PyTangoArchiving/trunk
```

Dependencies:

1. Tango Java Archiving, ArchivingRoot from sourceforge,
2. PyTango
3. python-mysql
4. Taurus (optional)
5. fandango:

```
$ svn co https://svn.code.sf.net/p/tango-cs/code/share/fandango/trunk/fandango␣
↪fandango
```

Setup:

- Follow Tango Java Archiving installation document to setup Java Archivers and Extractors.
- Some of the most common installation issues are solved in several topics in Tango forums (search for Tdb/Hdb/Snap Archivers):
  - tango-archiving,
  - installation-of-hdb-and-tdb-in-linux.
- Install PyTango and MySQL-python using their own setup.py scripts.
- fandango, and PyTangoArchiving parent folders must be added to your *PYTHONPATH* environment variable.
- Although Java Extractors may be used, it is recommended to configure direct MySQL access for PyTangoArchiving

Accessing MySQL:

Although not needed, I recommend you to create a new MySQL user for data querying:

```
$ mysql -u hdbmanager -p hdb

$ GRANT USAGE ON hdb.* TO 'user'@'localhost' IDENTIFIED BY '**********';
$ GRANT USAGE ON hdb.* TO 'user'@'%' IDENTIFIED BY '**********';
$ GRANT SELECT ON hdb.* TO 'user'@'localhost';
$ GRANT SELECT ON hdb.* TO 'user'@'%';

$ mysql -u tdbmanager -p tdb

$ GRANT USAGE ON tdb.* TO 'user'@'localhost' IDENTIFIED BY '**********';
$ GRANT USAGE ON tdb.* TO 'user'@'%' IDENTIFIED BY '**********';
$ GRANT SELECT ON tdb.* TO 'user'@'localhost';
$ GRANT SELECT ON tdb.* TO 'user'@'%';
```

Check in a python shell that your able to access the database:

```
import PyTangoArchiving

PyTangoArchiving.Reader(db='hdb',config='user:password@hostname')
```

Then configure the Hdb/Tdb Extractor class properties to use this user/password for querying:

```
import PyTango

PyTango.Database().put_class_property('HdbExtractor',{'DbConfig':'user:password@hostname
↪'})

PyTango.Database().put_class_property('TdbExtractor',{'DbConfig':'user:password@hostname
↪'})
```

You can test now access from a Reader (see recipes below) object or from a taurustrend/ArchivingBrowser UI (Taurus required):

```
python PyTangoArchiving/widget/ArchivingBrowser.py
```

Download

Download PyTangoArchiving from sourceforge:

```
svn co https://svn.code.sf.net/p/tango-cs/code/archiving/tool/PyTangoArchiving/trunk
```

Submodules

- api,
  - getting servers/devices/instances implied in the archiving system and allowing
- historic,
  - configuration and reading of historic data
- snap,
  - configuration and reading of snapshot data,
- xml,
  - conversion between xml and csv files
- scripts,
  - configuration scripts
- reader,
  - providing the useful Reader and ReaderProcess objects to retrieve archived data

General usage

In all these examples you can use hdb or tdb just replacing one by the other

Get archived values for an attribute

The reader object provides a fast access to archived values

```
In [9]: import PyTangoArchiving
In [10]: rd = PyTangoArchiving.Reader('hdb')
In [11]: rd.get_attribute_values('expchan/eh_emet02_ctrl/3/value','2013-03-20 10:00',
↪'2013-03-20 11:00')
Out[11]:
[(1363770788.0, 5.79643e-14),
 (1363770848.0, 5.72968e-14),
 (1363770908.0, 5.7621e-14),
 (1363770968.0, 6.46782e-14),
 ...
```

Start/Stop/Check attributes

You must create an Archiving api object and pass to it the list of attributes with its archiving config:

```
import PyTangoArchiving
hdb = PyTangoArchiving.ArchivingAPI('hdb')
attrs = ['expchan/eh_emet03_ctrl/3/value','expchan/eh_emet03_ctrl/4/value']

#Archive every 15 seconds if change> +/-1.0, else every 300 seconds
modes = {'MODE_A': [15000.0, 1.0, 1.0], 'MODE_P': [300000.0]}

#If you omit the modes argument then archiving will be every 60s
hdb.start_archiving(attrs, modes)

hdb.load_last_values(attrs)
{'expchan/eh_emet02_ctrl/3/value': [[datetime.datetime(2013, 3, 20, 11, 38, 9),
    7.27081e-14]],
    'expchan/eh_emet02_ctrl/4/value': [[datetime.datetime(2013, 3, 20, 11, 39),
    -3.78655e-08]]
}

hdb.stop_archiving(attrs)
```

Loading a .CSV file into Archiving

The .csv file must have a shape like this one (any row starting with '#' is ignored):

```
Host  Device  Attribute   Type    ArchivingMode   Periode >15  MinRange    MaxRange

#This header lines are mandatory!!!
@LABEL  Unique ID
@AUTHOR Who?
@DATE   When?
@DESCRIPTION    What?

```

(continues on next page)

(continued from previous page)

```
#host   domain/family/member    attribute   HDB/TDB/STOP    periodic/absolute/relative

cdi0404 LI/DI/BPM-ACQ-01    @DEFAULT        periodic    300
                            ADCChannelAPeak HDB absolute    15  1   1
                                            TDB absolute    5   1   1
                            ADCChannelBPeak HDB absolute    15  1   1
                                            TDB absolute    5   1   1
                            ADCChannelCPeak HDB absolute    15  1   1
                                            TDB absolute    5   1   1
                            ADCChannelDPeak HDB absolute    15  1   1
                                            TDB absolute    5   1   1
```

The command to insert it is:

```
import PyTangoArchiving
PyTangoArchiving.LoadArchivingConfiguration('/...fbecheri_20130319.csv','hdb',
↪launch=True)
```

There are some arguments to modify Loading behavior.

launch:

```
if not explicitly True then archiving is not triggered, it just verifies that format of␣
↪the file is Ok and attributes are available
```

force:

```
if False the loading will stop at first error, if True then it tries all attributes even␣
↪if some failed
```

overwrite:

```
if False attributes already archived will be skipped.
```

Checking the status of the archiving

```
hdb = PyTangoArchiving.ArchivingAPI('hdb')
hdb.load_last_values()
filter = "/" #Put here whatever you want to filter the attribute names
lates = [a for a in hdb if filter in a and hdb[a].archiver and hdb[a].modes.get('MODE_P
↪') and hdb[a].last_date<(time.time()-(3600+1e-3*hdb[a].modes['MODE_P'][0]))]

#Get the list of attributes that cannot be read from the control system (ask system␣
↪responsibles)
unav = [a for a in lates if not fandango.device.check_attribute(a,timeout=6*3600)]
#Get the list of attributes that are not being archived
lates = sorted(l for l in lates if l not in unav)
#Get the list of archivers not running properly
bad_archs = [a for a,v in hdb.check_archivers().items() if not v]

#Restarting the archivers/attributes that failed
```

(continues on next page)

(continued from previous page)

```
bads = [l for l in lates if hdb[l] not in bad_archs]
astor = fandango.Astor()
astor.load_from_devs_list(bad_archs)
astor.restart_servers()
hdb.restart_archiving(bads)
```

Restart of the whole archiving system

```
admin@archiving:> archiving_service.py stop-all
...
admin@archiving:> archiving_service.py start-all
...
admin@archiving:> archiving_service.py status

#see archiving_service.py help for other usages
```

Using the Python API

Start/Stop of an small (<10) list of attributes

```
#Stopping ...
api.stop_archiving(['bo/va/dac/input','bo/va/dac/settings'])

#Starting with periodic=60s ; relative=15s if +/-1% change
api.start_archiving(['bo/va/dac/input','bo/va/dac/settings'],{'MODE_P':[60000],'MODE_R
↪':[15000,1,1]})

#Restarting and keeping actual configuration

attr_name = 'bo/va/dac/input'
api.start_archiving([attr_name],api.attributes[attr_name].extractModeString())
```

Checking if a list of attributes is archived

```
In [16]: hdb = PyTangoArchiving.api('hdb')
In [17]: sorted([(a,hdb.load_last_values(a)) for a in hdb if a.startswith('bl04')])
Out[17]:
[('bl/va/elotech-01/output_1',
  [[datetime.datetime(2010, 7, 2, 15, 53), 6.0]]),
 ('bl/va/elotech-01/output_2',
  [[datetime.datetime(2010, 7, 2, 15, 53, 11), 0.0]]),
 ('bl/va/elotech-01/output_3',
  [[datetime.datetime(2010, 7, 2, 15, 53, 23), 14.0]]),
 ('bl/va/elotech-01/output_4',
  [[datetime.datetime(2010, 7, 2, 15, 52, 40), 20.0]]),
...
```

Getting information about attributes archived

Getting the total number of attributes:

```
import PyTangoArchiving
api = PyTangoArchiving.ArchivingAPI('hdb')
len(api.attributes) #All the attributes in history
len([a for a in api.attributes.values() if a.archiving_mode]) #Attributes configured
```

Getting the configuration of attribute(s):

```
#Getting as string
modes = api.attributes['rs/da/bpm-07/CompensateTune'].archiving_mode

#Getting it as a dict
api.attributes['sr/da/bpm-07/CompensateTune'].extractModeString()

#OR
PyTangoArchiving.utils.modes_to_dict(modes)
```

Getting the list of attributes not updated in the last hour:

```
failed = sorted(api.get_attribute_failed(3600).keys())
```

Getting values for an attribute:

```
import PyTangoArchiving,time

reader = PyTangoArchiving.Reader() #An HDB Reader object using HdbExtractors
#OR
reader = PyTangoArchiving.Reader(db='hdb',config='pim:pam@pum') #An HDB reader accessing␣
↪to MySQL

attr = 'bo04/va/ipct-05/state'
dates = time.time()-5*24*3600,time.time() #5days
values = reader.get_attribute_values(attr,*dates) #it returns a list of (epoch,value)␣
↪tuples
```

Exporting values from a list of attributes as a text (csv / ascii) file

```
from PyTangoArchiving import Reader
rd = Reader(db='hdb') #If HdbExtractor.DbConfig property is set one argument is enough
attrs = [
         'bl11-ncd/vc/eps-plc-01/pt100_1',
         'bl11-ncd/vc/eps-plc-01/pt100_2',
        ]

#If you ignore text argument you will get lists of values, if text=True then you get a␣
↪tabulated file.
ascii_values = rd.get_attributes_values(attrs,
                      start_date='2010-10-22',stop_date='2010-10-23',
                      correlate=True,text=True)

print ascii_values

#Save it as .csv if you want ...
open('myfile.csv','w').write(ascii_values)
```

Filtering State changes for a device

```
import PyTangoArchiving as pta
rd = pta.Reader('hdb','...:...@...')
vals = rd.get_attribute_values('bo02/va/ipct-02/state','2010-05-01 00:00:00','2010-07-13␣
↪00:00:00')
bads = []
for i,v in enumerate(vals[1:]):
    if v[1]!=vals[i-1][1]:
        bads.append((v[0],vals[i-1][1],v[1]))
report = [(time.ctime(v[0]),str(PyTango.DevState.values[int(v[1])] if v[1] is not None␣
↪else 'None'),str(PyTango.DevState.values[int(v[2])] if v[2] is not None else 'None'))␣
↪for v in bads]

report =
[('Sat May  1 00:07:03 2010', 'UNKNOWN', 'ON'),
...
```

Getting a table with last values for all attributes of a same device

```
hours = 1
device = 'bo/va/ipct-05'
attrs = [a for a in reader.get_attributes() if a.lower().startswith(device)]
vars = dict([(attr,reader.get_attribute_values(attr,time.time()-hours*3600)) for attr in␣
↪attrs])
table = [[time.ctime(t0)]+
         [([v for t,v in var if t<=t0] or [None])[-1] for attr,var in sorted(vars.
↪items())]
        for t0,v0 in vars.values()[0]]
```

(continues on next page)

(continued from previous page)

```
print('\n'.join(
      ['\t'.join(['date','time']+[k.lower().replace(device,'') for k in sorted(vars.
↪keys())])]+
      ['\t'.join([str(s) for s in t]) for t in table]))
```

Using CSV files

Loading an HDB/TDB configuration file

Create dedicated archivers first

If you want to use this option it will require some RAM resources in the host machine (64MbRAM/250Attributes) and installing the ALBA-Archiving bliss package.

```
from PyTangoArchiving.files import DedicateArchiversFromConfiguration
DedicateArchiversFromConfiguration('LX_I_Archiving.csv','hdb',launch=True)
```

TDB Archiving works different as it shouldn't be working on diskless machines, using instead a centralized host for all archiver devices.

```
DedicateArchiversFromConfiguration('LX_I_Archiving.csv','tdb',centralized='archiving01',
↪launch=True)
```

Loading the .csv files

All the needed code to do it is:

```
import PyTangoArchiving

#With launch=False this function will do a full check of the attributes and print the␣
↪results
PyTangoArchiving.LoadArchivingConfiguration('/data/Archiving//LX_I_Archiving_.csv','hdb',
↪launch=False)

#With launch=True configuration will be recorded and archiving started
PyTangoArchiving.LoadArchivingConfiguration('/data/Archiving//LX_I_Archiving_.csv','hdb',
↪launch=True)

#To force archiving of all not-failed attributes
PyTangoArchiving.LoadArchivingConfiguration('/data/Archiving//LX_I_Archiving_.csv','hdb',
↪launch=True,force=True)

#Starting archiving in TDB mode (kept 5 days only)
PyTangoArchiving.LoadArchivingConfiguration('/data/Archiving//LX_I_Archiving_.csv','tdb',
↪launch=True,force=True)
```

Note: You must take in account the following conditions:

- Names of attributes must match the NAME, not the LABEL! (that's a common mistake)

- Devices providing the attributes must be running when you setup archiving.
- Regular expressions are **NOT ALLOWED** (I know previous releases allowed it, but never worked really well)

filtering a list of CSV configurations / attributes to load

You can use GetConfigFiles and filters/exclude to select a predefined list of attributes

```
import PyTangoArchiving as pta

filters = {'name':".*"}
exclude = {'name':"(s.*bpm.*)|(s10.*rf.*)|(s14.*rf.*)"}

#TDB
confs = pta.GetConfigFiles(mask='.*(RF|VC).*')
for target in confs:
    pta.LoadArchivingConfiguration(target,launch=True,force=True,overwrite=True,
↪dedicated=False,schema='tdb',filters=filters,exclude=exclude)

#HDB
confs = pta.GetConfigFiles(mask='.*BO.*(RF|VC).*')
for target in confs:
    pta.LoadArchivingConfiguration(target,launch=True,force=True,overwrite=True,
↪dedicated=True,schema='hdb',filters=filters,exclude=exclude)
```

Comparing a CSV file with the actual configuration

```
import PyTangoArchiving
api = PyTangoArchiving.ArchivingAPI('hdb')
config = PyTangoArchiving.ParseCSV('Archiving_RF_.csv')

for attr,conf in config.items():
    if attr not in api.attributes or not api.attributes[attr].archiving_mode:
        print '%s not archived!' % attr
    elif PyTangoArchiving.utils.modes_to_string(api.check_modes(conf['modes']))!=api.
↪attributes[attr].archiving_mode:
        print '%s: %s != %s' %(attr,PyTangoArchiving.utils.modes_to_string(api.check_
↪modes(conf['modes'])),api.attributes[attr].archiving_mode)
```

Checking and restarting a known system from a .csv

```
import PyTangoArchiving.files as ptaf
borf = '/data/Archiving/BO_20100603_v2.csv'
config = ptaf.ParseCSV(borf)
import PyTangoArchiving.utils as ptau
hdb = PyTangoArchiving.ArchivingAPI('hdb')

missing = [
    'bo/ra/fim-01/remotealarm',
```

(continues on next page)

(continued from previous page)

```
    'bo/ra/fim-01/rfdet1',
    'bo/ra/fim-01/rfdet2',
    'bo/ra/fim-01/arcdet5',
    'bo/ra/fim-01/rfdet3',
    'bo/ra/fim-01/arcdet3',
    'bo/ra/fim-01/arcdet2',
    'bo/ra/fim-01/vacuum']

ptau.check_attribute('bo/ra/fim-01/remotealarm')
missing = 'bo/ra/fim-01/arcdet4|bo/ra/fim-01/remotealarm|bo/ra/fim-01/rfdet1|bo/ra/fim-
↪01/rfdet2|bo/ra/fim-01/arcdet5|bo/ra/fim-01/rfdet3|bo/ra/fim-01/arcdet3|bo/ra/fim-01/
↪arcdet2|bo/ra/fim-01/vacuum'

ptaf.LoadArchivingConfiguration(borf,filters={'name':missing},launch=True)
ptaf.LoadArchivingConfiguration(borf,filters={'name':'bo/ra/eps-plc.*'},stop=True,
↪force=True)
ptaf.LoadArchivingConfiguration(borf,filters={'name':'bo/ra/eps-plc.*'},launch=True,
↪force=True)

rfplc = ptaf.ParseCSV(borf,filters={'name':'bo/ra/eps-.*'})
stats = ptaf.CheckArchivingConfiguration(borf,period=300)
```

9.2.15 How to run a device server with an active Windows firewall

Intended audience: administrators, developers

When running Tango device servers on a Windows PC with active firewall you might find some problems when trying to reconnect to a restarted server. In some cases the client will never reconnect.

This behavior is due to the automatic closure of the used device server port when stopping the process. The client will only receive a timeout exception on the blocked port instead of an expected "connection failed" exception which triggers the reconnection.

When restarting a device server it will dynamically open another port. New clients can connect to the new port. But old clients might not reconnect.

To overcome the problem, start your device server with a fixed port (11000 in this example) as

```
$ TangoTest win -ORBendPoint giop:tcp11000
```

and open the fixed port in the firewall as

```
$ Netsh firewall add portopening TCP 11000 TangoTest
```

You can verify the open ports for the firewall with

```
$ Netsh firewall show portopening
```

The result should look like

```
Port configuration for Domaine profile:

Port     Protocol    Mode      Name
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------

11000    TCP         Enable    TangoTest
```

9.2.16 How to run device servers on Windows XP with IPv6

Intended audience: administrators, developers

This HowTo gives some advices about running Tango device server on Windows XP when IPv6 protocol stack is installed.

The ORB (CORBA Object Request Broker) used by Tango is omniORB. On platforms where it is available, omniORB supports IPv6. On most Unix platform, Ipv6 sockets accept both IPv6 and IPv4 connections. On Windows, each socket type only accepts incoming connections of the same type, so an IPv6 socket cannot be used with IPv4 clients and vice-versa. Since Windows XP SP2, it is easily possible to install the IPv6 stack. Running the ipconfig command in a cmd window will rapidly show you if IPv6 is installed.

The TANGO_HOST environment variable

Classically, the TANGO_HOST environment variable is set to the name of the computer where the database is running followed by a semi-colon and the database server port number (MyWindowsXp:20000). If the host where the database server is running is a Windows XP with IPv6 installed, the host name part in the environment variable has to be replaced by the host IPv4 address. If the host where is running the database server has a name "MyWindowsXp" and a IPv4 address set to 160.103.81.168, set the TANGO_HOST to:

```
TANGO_HOST=160.103.81.168:20000
```

Running a device server

To run a device server on this Windows host with IPv6 installed, we have to explicitly ask the underlying ORB to create an IPv4 address and to publish in the IOR only the IPv4 address. Two omniORB command line options allow us to do this:

```
MyDeviceServer inst_name -ORBendPoint giop:tcp:0.0.0.0: -ORBendPointPublish ipv4
```

Running the database server

The database server creates one device which is the anchor of a Tango control system. To achieve this, this device is registered in the ORB in a specific manner. To start the database server on a Windows host with IPv6 installed, it is neceesary to change its command line and to use the following command line arguments (Server listenning only on IPv4 port 20000)

```
database dbds -ORBendPoint giop:tcp:0.0.0.0:20000
```

Conclusion

To know the differences between IPv6 vs IPv4 and which is the most secure, please, follow the link.

9.2.17 How to start a device server

Intended audience: all

While reading this how-to please refer to manuals of *Jive* and *Astor* tools.

Starting device servers with Jive

In Jive select *Tools* > *Server Wizard* from the menu. In the wizard, fill the server and the instance name and click *Next*. The server name should be the same as the server executable name, and instance name can be any name you provide.

Fig. 1: Wizard - step 1 - device server instance

The wizard will now wait until you start the device server. You can do this from command line, device servers usually take the instance name as an argument. For this example you can use `TangoTest test2` command. When done click *Next*.

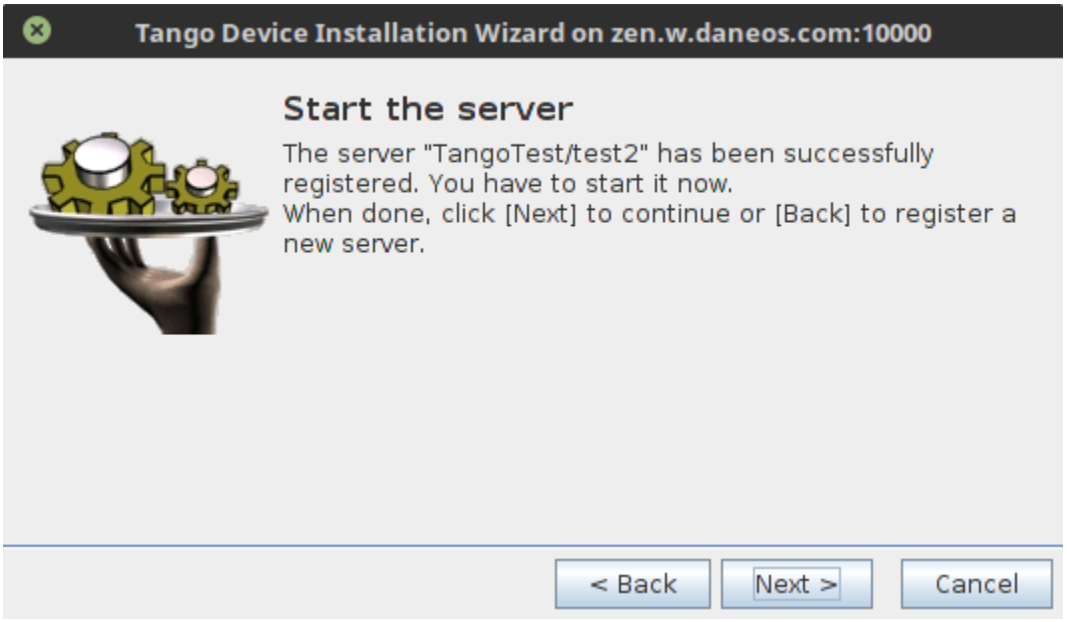


Fig. 2: Wizard - step 2 - starting the device server

In the next steps you will be asked to configure the devices that your device server instance will host. For this you need to pick a class and name for your device. A list of supported classes is provided by the device server, and the name of the device can be any name unique in the system in the `<domain>/<family>/<member>` format.

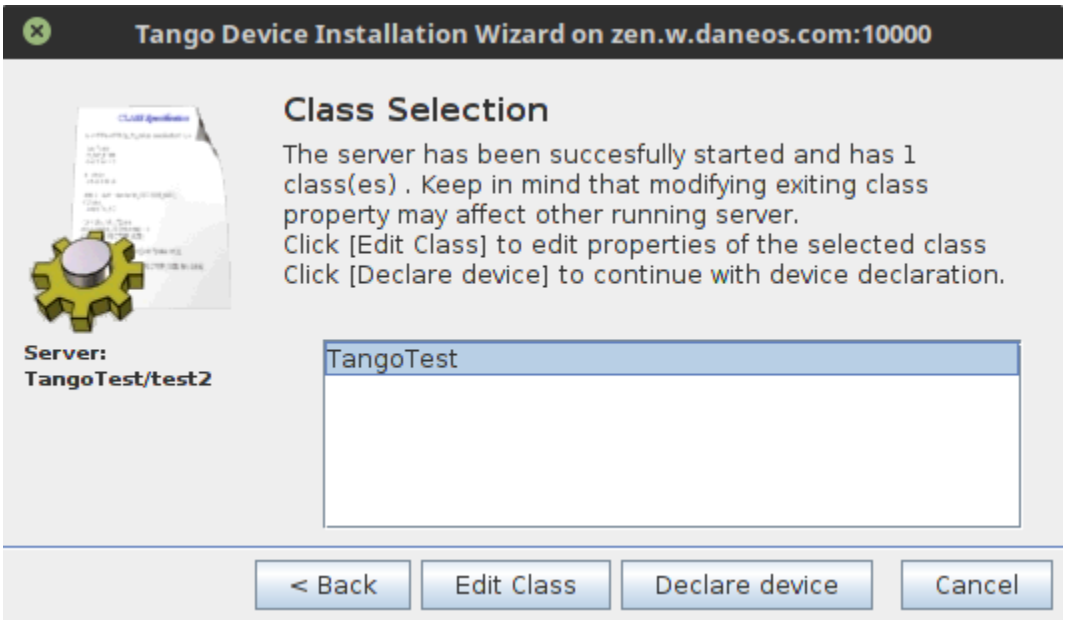


Fig. 3: Wizard - step 3 - device class

Fig. 4: Wizard - step 4 - device name

Next, you can configure properties for the newly created devices.

Fig. 5: Wizard - step 5 - device properties

After that you can create another device of the same class by clicking *New Device* or pick another class by clicking *New Class*. You can also finish the configuration and exit the wizard with the *Finish* button.

Fig. 6: Wizard - step 6 - finish

After the configuration, the device server must be restarted to load and start configured devices.

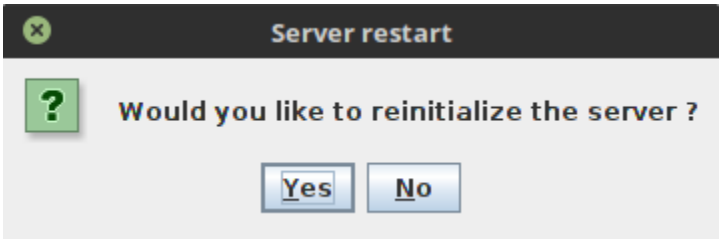


Fig. 7: Wizard - step 7 - device server restart

Starting device servers with Astor

Note: To configure device servers with Astor, you will need *Starter* installed and running on the host you want to start your device server on. Device server should be available in paths configured for the Starter.

In Astor go to the host control panel and click *Start New* button in the top left corner. If you want just to start an existing device server instance, pick it from the list and click *Start Server*.

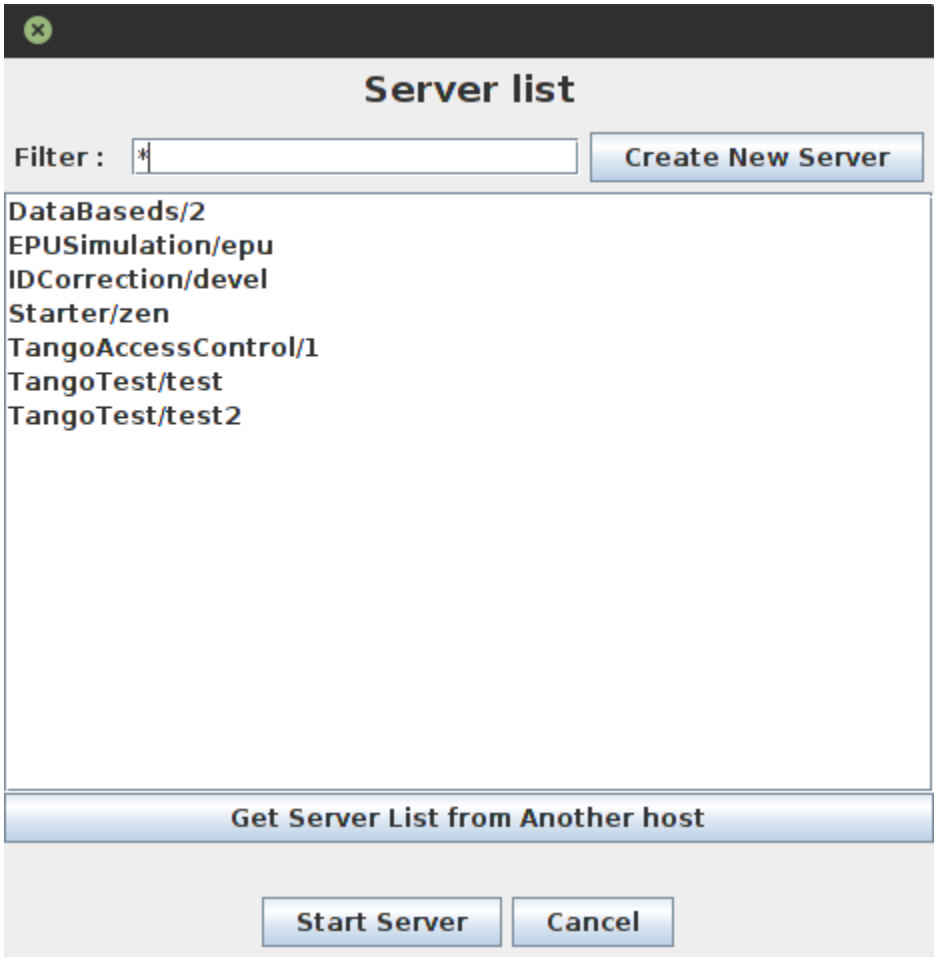


Fig. 8: Server list in Astor

If you want to add new server, click *Create New Server*. This will open the device installation wizard from Jive. Astor will handle starting the device server for you in step 2 of the wizard. After finishing the wizard, you will be asked to select the startup level of the new server.

Fig. 9: Startup level configuration

Other options

You can use the tango_admin utility to register server from scripts or command line. There are also some device servers (e.g. Sardana) that can register themselves upon starting.

9.2.18 How to try Tango Controls

Intended audience: all, developers

There are several ways to try the Tango Controls System.

For the first quick look, you can download and run the *TangoBox Virtual Machine*. Also, it is a possibility to build the system from *source code*.

If you have installed and configured Tango, you can skip to: *Play with Tango Controls*.

Installation notes on Linux base distribution OS

SHORTCUTS

1. Using Ansible
2. Installation on Debian
3. *Installation on RedHat(CentOS)*

If you want to use Debian/Ubuntu or Docker solution to try Tango you can skip this article and go to:

- *How to install Tango on Debian/Ubuntu*
- *Using Tango docker containers*

Tango requires:

- omniORB - CORBA implementation
- zeroMQ - event system

You can install this library from source: omniORB, zeroMQ or by your package-management tools like yum or apt-get.

Note: Remember that Tango needs the header files (.h) in the compilation process so is a need to install developers version of these libraries.

Also, many elements of the Tango ecosystem and the Control System itself requires the **MariaDB** or **MySQL** database. You can easily install it using apt or yum:

```
apt install mysql-server/mariadb-server
```

or

```
yum install mysql-server/mariadb-server
```

Download the source of the Tango Control System

```
wget -L https://gitlab.com/api/v4/projects/24125890/packages/generic/
↪TangoSourceDistribution/9.3.5/tango-9.3.5.tar.gz
tar xzf tango-9.3.5.tar.gz
```

In this directory, a good practice is to create the build folder, to don't mix a configuration/compile file with the source.

```
cd tango-9.3.5
mkdir build
```

Compile and installation notes

To fully configure TANGO installation you can tell where is the omniORB, ZMQ library and where you want to install Tango Controls System. Also, you need to configure the connection with the database and show where is the mysqlclient-lib.

```
../configure                                    \
     --prefix=/usr/install_dir_linux            \
     --with-omni=/usr/update                    \
     --with-zmq=/usr/update                     \
     --with-mysqlclient-lib=/usr/lib64/mysql
```

If the configuration of the system ends successfully, you should see this output:

If the value of the **database server** and **database schema create** is **no**, this means that during the configuration process the connection with the database can't be established. In this case, check if the path to the mysqlclient-lib is correct. The second reason can be the database credentials. Because the default value for the mysql-user and mysql-password is blank. For setup the proper database configuration can be used the **–with-mysql-admin** and **–with-mysql-admin-pass** flag adding to the configure command. Also is a possibility to add the **–with-mysql-ho** flag to set up the different host with the database to connect.

Now compile and install TANGO by the command:

```
make all
make install
```

Running the Tango System

It is essential to correctly run the Tango elements, to better maintain and automate the process starting the necessary part of the Tango Control System. For this purpose, it is recommended to create the system services.

In the case when the system is installed from a .deb package or is running in a docker container the package or the image provides services. However, in the approach described in the above paragraph creating the services and daemon have to be done manually.

So the process of starting the two main server-side elements of the tango environment (*Tango Database* and *Tango Starter*) can be optimized.

For creating the **Tango DB services** make in your */lib/systemd/system directory*, file named tango-db.service, containing *tango_db.service*.

In this service system start the mariadb database process, so for the mysql database this requirement must be changed:

```
Requires=mysqld.service
After=mysqld.service
```

There are two main environment files used in this service. One containing the *Tango Host* address in file */etc/tangorc*:

```
TANGO_HOST=address:port
```

The second file is providing the database credentials. Tango automatically creates it in the */etc/sysconfig/tango-db*. This file contains the database setting e.q:

```
MYSQL_USER=tango
MYSQL_PASSWORD=tango
```

To proper setup the **Tango Starter daemon**, create the file in the */etc/init.d/tango-starter*, containing *tango_starter*. Starter daemon similar like the Tango DB service uses the TANGO_HOST variables to create a connection with a database. The second setting equals the system user, used to start the daemon. The variables informing about this user are configured in the */etc/sysconfig/tango-starter* file:

```
TANGO_USER=tangosys
```

When all configuration is done, finally system is ready to start the Tango Control System main elements:

```
systemctl start tango-db
systemctl start tango-starter
```

The **`systemctl start`** command run the process once. If these services should automatically start on the boot of the machine, it needs to run the enable command for these processes:

```
systemctl enable tango-db
systemctl enable tango-starter
```

See more: *Tango with systemd integration*

Installation on CentOS

CentOS base on RPM Package Manager and used YUM as the primary tool to packages management and update the libraries by the system command line interfaces. Thanks **MAX IV Laboratory** that provides the public RPM repository with Tango packages now process of installation the Tango Control System can also be simplified. It is enough that it is added to the system as a new repository:

```
yum-config-manager --add-repo http://pubrepo.maxiv.lu.se/rpm/el7/x86_64/
```

This command creates a new repository and file in /etc/yum.repos.d containing all necessary information about it. Now you can install Tango Starter or Tango Database by the command:

```
yum install tango-starter/tango-db
```

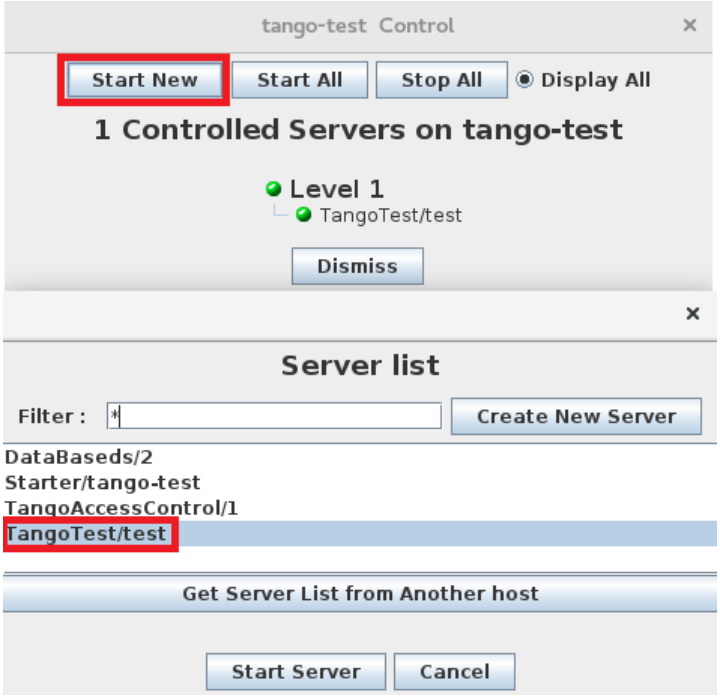
Play with Tango Controls

Tango eco-system provides a lot of management application and framework to visualization the data. This chapter provides a quick overview of a basic use case for Tango Control.

The new host is adding automatically by tango-starter daemon, but we can do this manually using the **Astor**. This application is used to configuring the Control System and its components. Moreover, it provides a quick view of the statuses of all *device servers* in the Tango. If you want to add the new host manually see: *Add a new controlled host*. For more information about Astor application see: *Astor (TANGO Manager)*.

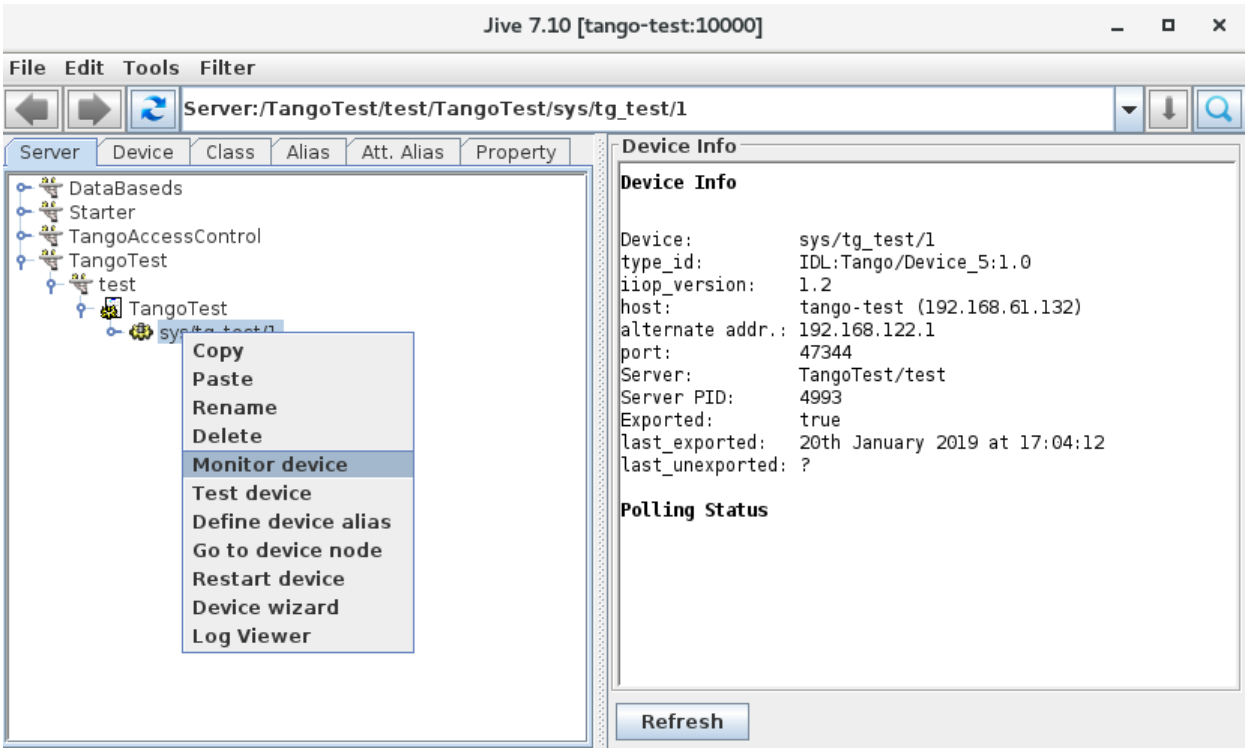
TangoTest this is a *device class* provides all types of attribute available in Tango Devices which can be used for the testing process. In help with Astor the process of starting this device server is simplified. After opening the control

panel for specific hosts (in Astor application), we can start a new device server which will be automatically run. Like in this screenshot:



See more: *Start a new server with Astor*.

When the TangoTest device is working, we can run the *atkpanel in jive* application and see the attribute, properties, all configuration of the selected device. For this purpose, we can run the jive application (by typing the command **`jive`**) and chose in the GUI options *Monitor Device* on the *Tango Device*, like in this screenshot:



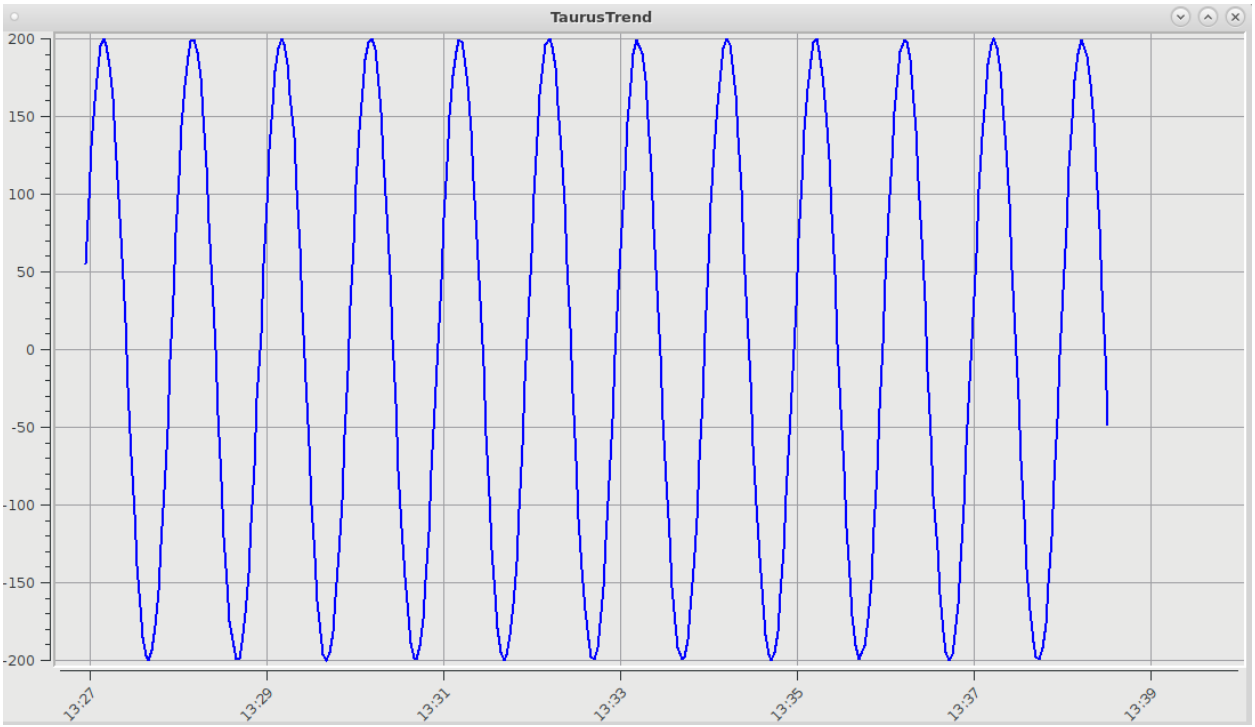
In atkpanel users can *execute the command* directly to the selected TangoTest device. List of useful command for this device:

- **`SwitchStates`** - changes states of the device (form RUN states to FAULT or FAULT to RUN)
- **`DevType`** - this is a DevType command example
- **`State`** - return the states of the device
- **`CrashFromX`** - simulate the crash of the device

Jive is an application design to browse and edit the static TANGO database, configure event and test devices. Full documentation for this application you can see here: *Jive*

In the atkpanel the attribute shows mainly the real-time values. So to better visualization of changes the selected values, the user can use *Taurus framework* widgets.

TangoTest attribute is generate using trigonometric functions, so it is easy to check if the device works correctly. The TaurusTrend for TangoTest attribute should look like in the screen below:



To run TaurusTrend uses a command:

```
taurustrend sys/tg_test/1/double_scalar_rww
```

To see custom Taurus device panel (similar like atkpanel) run the different command:

```
taurusdevicepanel sys/tg_test/1/double_scalar_rww
```

To storage the longtime history of changes of the attributes, you can use the *Tango Archiving System*.

Tango database service source

```
# If you want to customize, the
# best way is to create a file "/etc/systemd/system/tango-db.service",
# containing
#   .include /lib/systemd/system/tango-db.service
#   ...make your changes here...
# or create a file "/etc/systemd/system/tango-db.service.d/foo.conf",
# which doesn't need to include ".include" call and which will be parsed
# after the file tango-db.service itself is parsed.
#
# For more info about custom unit files, see systemd.unit(5) or
# http://fedoraproject.org/wiki/Systemd#How_do_I_customize_a_unit_file.2F_add_a_custom_
↪unit_file.3F

# Note: /usr/lib/... is recommended in the .include line though /lib/...
# still works.
# Don't forget to reload systemd daemon after you change unit configuration:
```

(continues on next page)

(continued from previous page)

```
# root> systemctl --system daemon-reload

[Unit]
Description=Tango database server
Requires=mariadb.service
After=mariadb.service

[Service]
Type=simple
User=tangosys
Group=tangosys

# find the TANGO_HOST setting
EnvironmentFile=/etc/tangorc
# read db credentials
EnvironmentFile=/etc/sysconfig/tango-db

ExecStart=/usr/bin/DataBaseds 2 -ORBendPoint giop:tcp:${TANGO_HOST}

# Give a reasonable amount of time for the server to start up/shut down
TimeoutSec=10

# Place temp files in a secure directory, not /tmp
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

Tango starter daemon source

```
#!/bin/sh
#
# tango-starter
#
# description:       Tango control system starter daemon
# chkconfig:         2345 90 10
#
. /etc/rc.d/init.d/functions

PREFIX=/usr
TANGO_CONFIG=/etc/tangorc
STARTER_CONFIG=/etc/sysconfig/tango-starter

PID_FILE=/var/run/tango-starter.pid
LOCK_FILE=/var/lock/subsys/tango-starter

# set up environment
. /etc/profile

[ -f $TANGO_CONFIG ] && . $TANGO_CONFIG
```

(continues on next page)

(continued from previous page)

```
[ -f $STARTER_CONFIG ] && . $STARTER_CONFIG

if [ -z $TANGO_HOST ]; then
    echo $"error: TANGO_HOST not set"
    exit 1
fi
export TANGO_HOST

DAEMON=Starter
DAEMON_ARGS=`hostname -s`

CLASS=Starter
SERVER=$CLASS/`hostname -s`
DEVICE=tango/admin/`hostname -s`

RETVAL=0
start()
{
    echo -n $"Starting Tango starter daemon"

    status -p $PID_FILE tango-starter >/dev/null
    if [ $? -eq 0 ]; then
    echo
    echo -n $"Starter server is running"
    RETVAL=0
    else
    tango_admin --ping-database >/dev/null
    if [ $? -ne 0 ]; then
        echo
        echo -n "Tango database connection failed"
        RETVAL=1
    else
        tango_admin --check-device $DEVICE
        if [ $? -ne 0 ]; then
        echo
        echo -n "Adding $SERVER to the Tango database"
        tango_admin --add-server $SERVER $CLASS $DEVICE >> /dev/null
        tango_admin --add-property $DEVICE startDsPath $PREFIX/bin >> /dev/null
        fi

        runuser -m $TANGO_USER -c \
        "$DAEMON $DAEMON_ARGS >/dev/null 2>&1 &"
        sleep 2

        # check if daemon survived..
        pidofproc $DAEMON > $PID_FILE
        RETVAL=$?
        if [ $RETVAL -eq 0 ]; then
        touch $LOCK_FILE
        else
        rm -f $PID_FILE
        fi
```

(continues on next page)

(continued from previous page)

```
    fi
    fi

    [ $RETVAL -eq 0 ] && success || failure
    echo
    return $RETVAL

}

stop()
{
    echo -n $"Stopping Tango starter daemon"

    status -p $PID_FILE tango-starter >/dev/null
    if [ $? -eq 3 ]; then
    echo
    echo -n $"Starter server is not running"
    RETVAL=0
    else
    killproc -p $PID_FILE $DAEMON
    RETVAL=$?
    [ $RETVAL -eq 0 ] && rm -f $PID_FILE $LOCK_FILE
    fi

    [ $RETVAL -eq 0 ] && success || failure
    echo
    return $RETVAL
}

restart()
{
    stop
    start
}

# See how we were called.
case "$1" in
    start )
    start
    ;;
    stop )
    stop
    ;;
    status )
    status -p $PID_FILE tango-starter
    ;;
    restart | reload )
    restart
    ;;
    condrestart )
    [ -f $LOCK_FILE ] && restart || :
    ;;
```

(continues on next page)

(continued from previous page)

```
    *)
    echo $"Usage: $0 {start|stop|status|restart|reload|condrestart}"
    exit 1
esac

exit 0
```

9.2.19 Using Tango servers without DB for unit testing

Intended audience: advanced developers, ***Programming language:*** java

Problem overview

One wants to test own server using full tango stack. This test must not be related to any environment and be performed together with other tests.

Detailed cases

Useful for benchmarking. Testing without setting up the whole Tango infrastructure.

Solution overview

It is possible to start Tango server without db: see *Device server without database*

Java:

Below is how one can integrate it into JUnit framework

```
1   public class ServerTest {
2
3       public static final String NO_DB_GIOP_PORT = "12345";//any random non-occupied␣
    ↪port
4       public static final String NO_DB_INSTANCE = "test_server";
5       public static final String NO_DB_DEVICE_NAME = "test/junit/" +NO_DB_INSTANCE;
6       //our server will run in a dedicated thread
7       private final ExecutorService server = Executors.newSingleThreadExecutor(new␣
    ↪ThreadFactory() {
8           @Override
9           public Thread newThread(Runnable r) {
10              Thread thread = new Thread(r);
11              thread.setName("My fancy Tango server");
12              thread.setDaemon(true);
13              return thread;
14          }
15      });
16
17      @Before
18      public void before() throws InterruptedException {
```

(continues on next page)

(continued from previous page)

```
            System.setProperty("OAPort", NO_DB_GIOP_PORT);

            final CountDownLatch latch = new CountDownLatch(1);
            server.submit(new Runnable() {
                @Override
                public void run() {
                    ServerManager.getInstance().start(new String[]{NO_DB_INSTANCE, "-nodb
↪", "-dlist", NO_DB_DEVICE_NAME}, TestServer.class);//TestServer - is our Tango server␣
↪we want to test against
                    latch.countDown();
                }
            });
            //make sure server has been started before leaving method
            latch.await();
        }

        @After
        public void after() {
            server.shutdownNow();
        }


        @Test
        public void testGetClientId() throws Exception {
            TangoProxy proxy = TangoProxies.newDeviceProxyWrapper("tango://localhost:" +␣
↪NO_DB_GIOP_PORT + "/" + NO_DB_DEVICE_NAME + "#dbase=no");

            assertSame(DeviceState.ON,proxy.readAttribute("State"));
        }
    }
```

It is also possible to use 3rd party Tango server to test against. Suppose our Tango server uses other Tango server to perform its tasks (Data aggregation, state monitoring etc). In this case 3rd party binary can be added to the project. This binary can be then launched from within JUnit test:

```
    //this test cases uses precompiled TangoTest stored in {PRJ_ROOT}/exec/tango/
↪<win64|debian> folder
    public class TangoTestProxyWrapperTest {
        public static final String TANGO_DEV_NAME = "test/local/0";
        public static final int TANGO_PORT = 16547;
        public static final String TEST_TANGO = "tango://localhost:" + TANGO_PORT + "/"␣
↪+ TANGO_DEV_NAME + "#dbase=no";
        public static final String X64 = "x64";
        public static final String LINUX = "linux";
        public static final String WINDOWS_7 = "windows 7";
        public static final String AMD64 = "amd64";

        private static Process PRC;

        @BeforeClass
        public static void beforeClass() throws Exception {
            String crtDir = System.getProperty("user.dir");
```

(continues on next page)

(continued from previous page)

```
            //TODO define executable according to current OS
            String os = System.getProperty("os.name");
            String arch = System.getProperty("os.arch");
            StringBuilder bld = new StringBuilder(crtDir);
            //TODO other platforms or rely on the environmet
            if (LINUX.equalsIgnoreCase(os) && AMD64.equals(arch))
                bld.append("/exec/tango/debian/").append("TangoTest");
            else if (WINDOWS_7.equalsIgnoreCase(os) && AMD64.equals(arch))
                bld.append("\\exec\\tango\\win64\\").append("TangoTest");
            else
                throw new RuntimeException(String.format("Unsupported platform: name=%s␣
↪arch=%s", os, arch));

            PRC = new ProcessBuilder(bld.toString(), "test", "-ORBendPoint", "giop:tcp::
↪" + TANGO_PORT, "-nodb", "-dlist", TANGO_DEV_NAME)
                    .start();

            //drain slave's out stream
            new Thread(new Runnable() {
                @Override
                public void run() {
                    char bite;
                    try {
                        while ((bite = (char) PRC.getInputStream().read()) > -1) {
                            System.out.print(bite);
                        }
                    } catch (IOException ignore) {
                    }
                }
            }).start();

            //drains slave's err stream
            new Thread(new Runnable() {
                @Override
                public void run() {
                    char bite;
                    try {
                        while ((bite = (char) PRC.getErrorStream().read()) > -1) {
                            System.err.print(bite);
                        }
                    } catch (IOException ignore) {
                    }
                }
            }).start();
        }

        @AfterClass
        public static void afterClass() throws Exception {
            PRC.destroy();
        }

        //this test directly writes/reads  to/from TangoTest double_scalar_w
```

(continues on next page)

(continued from previous page)

```
        @Test
        public void testWriteReadAttribute_Double() throws Exception {
            TangoProxy instance = TangoProxies.newDeviceProxyWrapper(TEST_TANGO);

            instance.writeAttribute("double_scalar_w", 0.1984D);

            double result = instance.<Double>readAttribute("double_scalar_w");

            assertEquals(0.1984D, result);
        }

        //in other test case one can create instance of his own server (see previous␣
↪code snippet)

    }
```

CPP:

//TODO

Python:

//TODO

9.2.20 How to tune polling by code in a TANGO class

Intended audience: advanced developers,

This HowTo explains how it is easily possible to tune attribute or command polling parameters in the code of a Tango class

Since Tango 8, it is possible to configure command or attribute polling within a Tango class code. A new set of polling related methods has been added to the base class **Tango::DeviceImpl.** These methods are similar to those you find in the Tango::DeviceProxy class when you write a Tango client. With them, you can

- Check if a command or attribute is polled
- Start/Stop polling for a command or a attribute
- Get or update polling period for a polled attribute or command

In C++

Programming language: c++

To display some information related to polling of the attribute named *TheAtt*:

```
string att_name("TheAtt");
cout << "Attribute " << att_name;

if (is_attribute_polled(att_name) == true)
   cout << " is polled with period " << get_attribute_poll_period(att_name) << " mS" <<␣
```

(continues on next page)

(continued from previous page)

```
    ↪endl;
else
   cout << " is not polled" << endl;
```

To poll a command simply type:

```
poll_command("TheCmd",250);
```

If the command is already polled, this method will update its polling period to 250 mS. Finally, to stop polling the same command, type:

```
stop_poll_command("TheCmd");
```

All these DeviceImpl polling related methods are documented in the DeviceImpl class documentation page

In Python

Programming language: python

To display some information related to polling of the attribute named *TheAtt,* in a **DeviceImpl** context type:

```
att_name = "TheAtt"

if self.is_attribute_polled(att_name):
    print("{0} is polled with period {1} ms".format(att_name, self.get_attribute_poll_
    ↪period(att_name))
else:
    print("{0} is not polled".format(att_name))
```

To poll a command, in a **DeviceImpl** context, simply type:

```
self.poll_command("TheCmd", 250)
```

If the command is already polled, this method will update its polling period to 250 mS. Finally, to stop polling the same command, in a **DeviceImpl** context type:

```
self.stop_poll_command("TheCmd")
```

All these DeviceImpl polling related methods are documented in the PyTango DeviceImpl class documentation page.

In Java

Programming language: java

The polling can be retrieved and modified from the DeviceManager class. Here is an example:

```
import org.tango.server.annotation.Device;
import org.tango.server.annotation.DeviceManagement;
import org.tango.server.device.DeviceManager;
import fr.esrf.Tango.DevFailed;
@Device
public class Test {
    @DeviceManagement
```

(continues on next page)

(continued from previous page)

```
    private DeviceManager deviceManager;
     ...
        final String attName = "TheAttr";
        if (deviceManager.isPolled(attName)) {
            System.out.println(attName + " is polled with period " + deviceManager.
↪getPollingPeriod(attName) + " mS");
        } else {
            System.out.println(attName + " is not polled");
        }
        deviceManager.startPolling("TheCmd", 250);
        deviceManager.stopPolling("TheCmd")
        ...

   public void setDeviceManager(final DeviceManager deviceManager) {
        this.deviceManager = deviceManager;
    }
}
```

9.2.21 How to use vectors to set attributes

Intended audience: developers, ***Programming language:*** c++

This page contains examples on how to use C++ vectors to set attribute values on the servers side.

> **Warning:** Tango is optimized not to copy data. For this reason all the attribute set_value() methods only take pointers as input. If you are going to use C++ vectors, you should be aware of the fact that you are going to copy the data! This might slow down execution time when working with large amount of data.

Examples for a vector of short and a vector of string:

```
void MyClass::read_Spectrum(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyClass::read_Spectrum() entering... "<< endl;

    vector<Tango::DevShort> val;
    val.push_back(1);
    val.push_back(2);
    val.push_back(3);

    // data copy !!
    Tango::DevVarShortArray tmp_seq;
    tmp_seq << val;

    attr.set_value (tmp_seq.get_buffer(), tmp_seq.length());
}
```

```
void MyClass::read_StringSpectrum(Tango::Attribute &attr)
{
    DEBUG_STREAM << "MyClass::read_StringSpectrum() entering... "<< endl;

```

(continues on next page)

(continued from previous page)

```
    vector<string> val;
    val.push_back("Hello");
    val.push_back("cruel");
    val.push_back("world!");

    // data copy !!
    Tango::DevVarStringArray tmp_seq;
    tmp_seq << val;

    attr.set_value (tmp_seq.get_buffer(), tmp_seq.length());
}
```

9.3 Tutorials for administrators

Intended audience: administrators, all

9.4 HOW-TOs for administrators

Intended audience: administrators

9.5 HOW-TOs for developers

Intended audience: developers

CHAPTER

TEN

REFERENCE

Intended audience: all

10.1 Map Key

Intended audience: beginners, developers, administrators, users

The color coded key for the clickable maps of the Tango ecosystem.

10.1.1 Acknowledgements

The maps were originally produced by Lajos Fülöp (ELI-ALPS) and have been updated and formatted for this document by Stuart James (ESRF).

10.1.2 Making Modifications

The layer map graph is maintained and modified using yEd Graph Editor. yEd exports the graph as a HTML/PNG file combo. This tool can be found at <https://www.yworks.com/products/yed>.

A simplified workflow can be described as follows:

- Open the desired graph to be edited. This is a .graphml file. Files pertaining to this project are located with the static data.
- Edit the visual graph as required. For example, updating html links in the various object properties.
- Save the changes, then select File->Export to export the changes. Export the graph as a "HTML ImageMap".
- You will be prompted with the "HTML Map Export" dialog. Ensure the image format is PNG on the Image tab. There should be no need to change any other settings.

The result of the above will be both a HTML file and a PNG file that can be embedded in the documentation. The HTML file is embedded directly using the raw keyword, and it will load the PNG file when rendered.

10.2 Glossary

Intended audience: all

Tango Controls
: Tango Controls is an object oriented, distributed control system framework which defines communication protocol and API as well as provides a set of tools and libraries to build software for control systems, especially *SCADA*.

Core
Tango Core
: Tango Core is a set of main tools, libraries and specifications of the Tango Controls framework. It consists of libraries and API definitions for C++, Java and Python as well as tools to manage the system: *Astor*, *Jive*, etc.

SCADA
: It is an abbreviation standing for Supervisory Control and Data Acquisition.

device server
: A Device Server is a program (executable) which is able to create *devices* of certain classes. A Device Server may implement one or multiple classes and instantiate one or more devices. A running device server is called a *device server instance*.

instance
device server instance
: A running device server is called a device server instance. So it means, it is a process. Every device server instance has an unique name in Tango Controls by which it can be referenced. The name is built as *{DeviceServerName}/{instanceName}*. For each running device server the system creates a special device of *DServer class*: *dserver/{DeviceServerName}/{instanceName}*. This device provides a management facility for the corresponding device server instance (see *DServer class device commands*).

class
device class
: A Device Class is an abstraction of a device's interface. It defines *attributes*, *pipes*, *commands* and *properties* which a device of the class provides to users and to other components of a Tango system. A device class ofter relates to a specific kind of equipment it allows to interface with like a *SerialLine* class defines interface to communicate with serial line equipment.

device
: A device is a key concept of Tango Controls. It is an object providing access to its *attributes*, *pipes* and *commands*. The list of attributes, pipes and commands available for a certain device is defined by its *device class*. The device may be related to a hardware device it interfaces with or it may be a kind of a logical device providing some functionalities not directly related to hardware.

attribute
: An attribute represents a process value (or values) in the system. It may have different formats or dimensions like scalar(0D), spectrum(1D) or image(2D). The attribute allows to read and/or write these values depending on programmer-defined access. The values may have different data types. In addition, an attribute provides some metadata like *attribute quality*, timestamp or configuration properties. For a complete list please refer to the manual. A list of attributes available for a certain device is defined by its *class*.

attribute quality
: A value returned by an *attribute* has a runtime quality facotr which is an enumeration describing the state of the read value (one of VALID, INVALID, ALARM, CHANGING, WARNING).

dynamic attribute
: A *device* may create attributes which configration is determined during device initialization or even at runtime. This kind of attributes is called *dynamic*.

command
A command is an operation a user may invoke on a device (eg. *SwitchOn*, *SwitchOff*). It also relates to a specific method in OOP (Object-Oriented Programming). Tango Controls allows a command to get input argument (argin) and to return a value (argout). List of available commands for a certain device is defined by its *device class*.

pipe
A pipe allows to read and/or write a structured data from and/or to a *device*. The data may be built out of several basic Tango datatypes. The structure of data is defined by a *device class* and is not fixed. It may be changed at runtime by the *device* itself or modified upon request from a client according to *set_pipe_config* operation provided by pipe. List of pipes available for a *device* is defined by its *device class*.

state
device state
A *device* may be in a certain state determined at runtime. State of a *device* may reflect state of a piece of equipment it interfaces with or be determined in other way. The behaviour is defined by the *device class* which implements a *state machine*. The state may define attributes', commands' and pipes' operations available at the moment. Tango Controls limits a set of states the device may be in to 14: ON, OFF, CLOSE, OPEN, INSERT, EXTRACT, MOVING, STANDBY, FAULT, INIT, RUNNING, ALARM, DISABLE, and UNKNOWN.

state machine
A state machine for a *device class* defines operations (commands', attributes' and pipes' access) available in different *states* of a *device*.

Tango Host
Each Tango Controls system/deployment has to have at least one running DataBaseds *device server*. The machine on which DataBaseds *device server* is running has a role of a so called *Tango Host*. *DataBaseds* is a device server providing configuration information to all other components of the system as well as a runtime catalog of the components/devices. It allows (among other things) client applications to find devices in distributed environment.

Tango Database
A database providing configuration and some runtime information about Tango Controls components in so called Tango System instance or deployment. It is used by *Databaseds* device server and constitutes *Tango Host*.

property
A configuration parameter stored in the *Tango Database*. Properties can be assigned to a *device class*, *device* or elements of device interface (*attributes*, *commands*, *pipes*). Properties can be also not related to *device* - such properties are called *free properties*. Property values are often used by elements of *Tango Controls* system during its startup. These usually provides information required to configure things like connections to hardware or to adjust to user preferences.

10.3 Bibliography

Intended audience: all

- genindex

CHAPTER

ELEVEN

CONTENTS FOR ADMINISTRATORS

Intended audience: administrators

CHAPTER

TWELVE

CONTENTS FOR DEVELOPERS

Intended audience: developers

CHAPTER

THIRTEEN

CONTENTS FOR USERS

Intended audience: user

BIBLIOGRAPHY

[OMG] OMG home page

[Henning] Advanced CORBA programming with C++ by M.Henning and S.Vinosky (Addison-Wesley 1999)

[MySQLbook] MySQL and mSQL by Randy Jay Yarger, George Reese and Tim King (O'Reilly 1999)

[Patterns] Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley 1995)

[omniORB] omniORB home page

[NotificationService] The Notification Service specification available from OMG home page - http://www.omg.org

[ASTOR] ASTOR home page

[Jive] JIVE home page

[ZMQ] ZMQ home page

[TangoRefMan] Tango class development reference documentation

INDEX

Symbols

$CLASSPATH, 393

A

AC, **543**
Archiving, **543**
Archiving mode, **543**
attribute, **752**
attribute quality, **752**

C

class, **752**
command, **753**
Core, **752**

D

device, **752**
device class, **752**
device server, **752**
device server instance, **752**
device state, **753**
dynamic attribute, **752**

E

environment variable
    $CLASSPATH, 393
    MYSQL_PASSWORD, 23
    MYSQL_USER, 23
    PKG_CONFIG_PATH, 451
    TANGO_HOST, 13, 15, 16, 22, 393
    TANGO_HOST=HOSTNAME:PORT, 15
    TANGO_HOST=productionserver:10000, 15, 16
    TANGO_HOST=testserver:10000, 15
    TANGO_ROOT, 444

H

HDB, **543**

I

instance, **752**

M

MYSQL_PASSWORD, 23
MYSQL_USER, 23

P

pipe, **753**
PKG_CONFIG_PATH, 451
property, **753**

Q

Quick Save/Quick load, **543**

S

SCADA, **752**
state, **753**
state machine, **753**

T

Tango Controls, **752**
Tango Core, **752**
Tango Database, **753**
Tango Host, **753**
TANGO_HOST, 13, 15, 16, 22, 393
TANGO_HOST=HOSTNAME:PORT, 15
TANGO_HOST=productionserver:10000, 15, 16
TANGO_HOST=testserver:10000, 15
TANGO_ROOT, 444
TDB, **543**

V

VC, **543**