Tango Request For Comments (RFC) Documentation

Release 9.3

Tango Controls Community

Jun 23, 2022

CONTENTS

CHAPTER

ONE

TANGO REQUEST FOR COMMENTS (RFC)

This repository is the home of all Tango Open Specification.

It is rendered on the readthedocs.io:

1.1 Mission

The goal of this RFC project is to provide a formal specification of the current (V9) Tango Controls system. This specification shall include:

1. Concepts,
2. Terminology,
3. Protocol behaviour,
4. Conventions,

each on a sufficient level for future evolution of Tango Controls and/or implementation in other languages. In that respect, concepts are more important than implementation details.

1.2 Contribution

The process to add or change an RFC is the following:

- An RFC is created and modified by pull requests according to the Collective Code Construction Contract (C4).
- The RFC life-cycle SHOULD follow the life-cycle defined in the Consensus-Oriented Specification System (COSS).

Read more here.

1.3 RFCs

The table below summarises all available or expected specifications.

Short Name	Title	Status	Editor
1/Tango	The Tango control system	Draft	Lorenzo Pivetta
2/Device	The device object model	Draft	Vincent Hardion
3/Command	The command model	Draft	Sergi Blanchi-Torné
4/Attribute	The attribute model	Draft	Sergi Blanchi-Torné
5/Property	The property model	Draft	Gwenaelle Abeillé
6/Database	The database system	Draft	Gwenaelle Abeillé
7/Pipe	The pipe model	Draft	Reynald Bourtembourg
8/Server	The server model	Draft	Lorenzo Pivetta
9/DataTypes	Data types	Draft	Gwenaelle Abeillé
10/RequestReply	The Request-Reply protocol	Draft	Reynald Bourtembourg
11/RequestReplyCORBA	The Request-Reply protocol - CORBA implementation	Postponed	
12/PubSub	The Publisher-Subscriber protocol	Draft	Vincent Hardion
13/PubSubZMQ	The Publisher-Subscriber protocol - ZeroMQ implementation	In progress	
14/Logging	Logging service	Draft	Sergi Blanchi-Torné
15/DynamicAttrCmd	The dynamic attribute and command	Draft	Reynald Bourtembourg
16/TangoResourceLocator	Tango Resource Locator	Draft	Thomas Braun
17/MemorisedAttr	Memorised attribute service	Postponed	
18/AccessCtrl	Authorisation system	Postponed	

CHAPTER TWO

1/TANGO

This document describes the philosophy of Tango, a control system framework. This RFC specifies a control system implementing the philosophy of Tango.

See also: All Tango RFC's

2.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

2.2 Tango Specification

The specification contained in the RFCs describes the preferred behaviour of the latest stable version (V9) of Tango. The specification is considered to be the desired behaviour and any implementation (in whichever language) differing from the specification is considered non-compliant. Anyone spotting such differences is encouraged to file an issue for the concerned implementation.

2.2.1 Goals

The *Tango toolkit* was initially developed for the needs of large research facilities, although the concept (philosophy) offers a framework for any control systems in general. It is based on the idea that devices represented as distributed objects are ideally suited to design and implement control systems which are flexible and capable of handling complexity. In this sense it aims to provide:

- an **Open Source Software** toolkit for building distributed object-oriented control systems. It should be freely available for all applications including industry. Tango Controls is sufficiently complete to control any installation small or large.

- a **Programmable way** which allows hardware to be interfaced and modelled in a device object model with a suited programming language.
- the tools for **configuring, managing, monitoring and archiving** large numbers of control points implemented as device attributes.
- the **naming, browsing and hierarchies** of devices and groups of devices.

It offers additionally the following qualities:

- **Scalable** for *building large and small control systems*. There should be no limit on the number of hardware and software devices the control system can control.
- **Usable** to replace proprietary industrial control systems.
- **Compatible** with a wide range of *communication protocols* and *industrial hardware standards*.

2.2.2 Concepts

The control system used in large facilities has to monitor and process large to huge amounts of data in real time and archive them for analysis and post processing.

Tango aims to interface almost any kind of hardware or software system, for which a well defined API exists, in order to represent these systems as Tango Devices within a control system (see *2/Device*).

Tango Devices are software components which have Commands (methods), Attributes (data), Properties (configuration) and Pipes (data streaming) and are accessible on the network or locally.

A **Tango control system** usually consists of a collection of Tango Devices, a Tango Database and a set of applications for configuring and controlling the Devices via well defined Application Programmers Interfaces (APIs).

The *Tango Database* provides a directory service to look up a Device's location in the control system for a client to directly establish a network connection with it (see *6/Database*). The *Tango Database* also store the configuration parameters for Devices and the relevant part of the control system.

2.2.3 Requirements

The critical requirements from the original design are [1] :

1. That the *Devices* are network objects with the minimum of functionality necessary for a distributed control object.
2. Use *CORBA* communication protocol while hiding details [2].
3. The support of the *synchronous, asynchronous and event driven* communication paradigms.
4. The support of *Device hierarchies* by inheritance and composition.
5. The support modern programming patterns like *multi-threading*.
6. Using a *naming database* which stores configuration information.
7. Using a generic, but finite, set of *Data Types* [3].
8. To be able to *monitors* Devices by polling.
9. To be able to cache Attribute.
10. The support for *scripting*
11. The support for *multiple platforms*

References:

[1] “TANGO - AN OBJECT ORIENTED CONTROL SYSTEM BASED ON CORBA” by Chaize et. al., ICALEPCS 1999 https://www.elettra.trieste.it/ICALEPCS99/proceedings/papers/wa2i01.pdf

[2] current implementations of Tango use CORBA/ZMQ as the communication protocol however the RFC’s do not impose this. Other communication protocols can be used to implement the Tango control system concepts and features.

[3] a rich set of over 20+ generic types which can be composed in *Pipes* are supported. Adding of new types is discouraged.

CHAPTER

THREE

2/DEVICE MODEL

This document describes the Tango Device model specification.

See also: *1/Tango*, *3/Command*, *4/Attribute*, *5/Property*, *6/Database*, *7/Pipe*, *8/Server*

3.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

3.2 Tango Device Specification

A Device is designed to represent any controlled object in Tango Controls System. This specification is inspired by the comment written by A. Götz and E. Taurel in the tango.idl file from the original implementation (https://gitlab.com/tango-controls/tango-idl)

3.2.1 Goals

The Tango Device represents the fundamental interface for all TANGO objects. Directly inspired by object-oriented programming, a Device object has data and actions. It allows users of the Tango Controls System to access (read and/or write) information (data) stored in or processed by hardware or virtual devices and call actions which may influence hardware state and behaviour. Typically, access to data is provided via Device's Attributes and Pipes and actions are initiated by calling Device's Commands.

Additionally, it aims to:

- Provide a standard way of translating different communication protocols to a communication protocol implemented by Tango Controls. For example, the same generic GUI application may be used to control a power supply interfaced with the SCPI protocol and to control a stepper motor driver interfaced with the CAN bus.

- Be usable as a logical abstraction of the hardware. For example, the same GUI application may be used to control power supplies which are powering magnets and powering heat coils.
- Expose, to Users and other Actors of the Tango Controls System, a semantic interface which is well understood, intuitive, and match the object-oriented paradigm. For example, a power supply Device may be implemented in Java, Python or C++ language. Irrespectively to the implementation language, Attributes *current* and *voltage* can be implemented as field members of an object, while *on* and *off* commands can be implemented as methods of the same object.

3.2.2 Use Cases

Typical use cases for Device are:

- It is common for particle accelerator systems to use a set of power supplies for powering a magnet system. Power supplies may come from different vendors. These power supplies may use different communication interfaces and protocols for remote control. However, there is a need to steer these power supplies from a remote location in a unified and consistent way. Steering includes setting and reading of voltages and currents as well as switching the power supplies on and off, each individually. Having them integrated into a system as Devices, each with an unique name, with Current and Voltage attributes and On() and Off() commands fulfills the requirement and assure consistency.
- Remote monitoring of a complex device like an electro-magnet, which is powered by power supplies and requires a cooling system may be provided by a Device with the following attributes: MagneticField, CoilTemperature, State and Status and with the commands On() and Off(). This Device, to serve its interface is using information coming (directly or through other devices) from power supplies and cooling systems.

3.3 Specification

A Tango Device is a strict definition of a distributed object.

- configuration is represented in the form of Properties.
- data are represented in the form of Attributes and Pipes.
- actions are represented in the form of Commands.

Its model can be represented as a defined tree where each element is from a defined type: Class, Device, Property, Attribute, Pipe, Command following the rules below:

- The Device is a distributed object which SHALL be accessible locally or via the network.
- The Device SHALL be an instance of one Class, see *Device Class section*.
- The Device MAY have one or several Property, called Device Property, see *5/Property*.
- The Device MAY have one or several Attribute, see *4/Attribute*.
- The Device MAY have one or several Pipe, see *7/Pipe*.
- The Device MAY have one or several Command, see *3/Command*.
- The Device SHALL have one Init Command.
- The Device SHALL have one State Attribute.
- The Device SHALL have one Status Attribute.
- The Device SHALL have one State Command.
- The Device SHALL have one Status Command.

Note: Future specification may remove 'SHALL have' requirements for State and Status Commands.

- The Device SHALL have one unique identifier which represents its Device Name.

3.3.1 Naming convention

- The Device Name SHALL use the following convention:

```
device-name = domain "/" family "/" member
domain = ALPHA *84acceptable-char
family = ALPHA *84acceptable-char
member = ALPHA *84acceptable-char
acceptable-char = ALPHA / DIGIT / underscore
underscore = %x5F
```

3.3.2 Device aliases

A Device MAY have an associated *alias*, a string which can be used in place of Device Name to address the Device.

See [16/TangoResourceLocator Alias](../16/TangoResourceLocator.md### Alias) for details.

3.3.3 Device Class

A Class is an association of a list of Class Properties, a list of Attributes, a list of Pipes, a list of Commands, a list of Device Properties with a Device Class Name.

- The Class Name SHOULD reflect its instances application context.
- The Class Name SHALL use the following convention:

```
class-name = ALPHA *254acceptable-char
acceptable-char = ALPHA / DIGIT / underscore
underscore = %x5F
```

- The Class MAY have one or more Device instance.
- The Class MAY have one or several Property, called Class Property.
- The Device SHALL have at least all Attributes, Pipes and Commands defined by its Class.
- The Device MAY have Attributes and/or Commands not defined by its Class. These are called Dynamic Attributes and/or Dynamic Commands respectively, see *15/Dynamic Attribute Command*.
- Dynamic Attributes and/or Dynamic Commands MAY be added to the Device during transition from the Not Exported phase to the Exported phase.
- In the Exported phase, Dynamic Attributes and/or Dynamic Commands MAY be added to the Device.
- In the Exported phase, Dynamic Attributes and/or Dynamic Commands MAY be removed from the Device.

3.3.4 Device lifecycle

The Device lifecycle is:

- Device MUST be in one of two phases: Exported or Not Exported.
- In the Exported phase, Device MUST be accessible to actors of the Tango Controls System.
- In the Not Exported phase, Device MUST NOT be accessible to actors of the Tango Controls System.
- At the transition from the Not Exported phase to the Exported phase the Device MUST be initialised as if its Init Command was executed.
- At the transition from the Exported phase to the Not Exported phase the Device SHOULD gracefully be uninitialised.
- The Device Server MUST manage the Device lifecycle.

3.3.5 Init Command

The Init Command purpose is to re-initialise the Device.

- The Init Command, when called, SHALL in a sequence:
  - Gracefully un-initialise the Device returning to a state before its first initialisation,
  - Initialise the Device using the values of its Properties.

3.3.6 Device State and Status

The State Attribute represents Device State.

- The *data type* of State Attribute MUST be `DevState`. The *read value* of Status Attribute MUST be of `DevState` data type.
- The State Attribute's *writable* metadata MUST be `READ`.
- The *data format* of State Attribute MUST be `SCALAR`.
- The *read value* of State Attribute SHOULD reflect the context of a running Device. If Device is related to hardware or a virtual entity or some process, the value of State Attribute SHOULD reflect the state of the entity or process.
- The *data type* of Status Attribute MUST be `DevString`. The *read value* of Status Attribute MUST be of `DevString` data type.
- The Status Attribute's `writable` metadata MUST be `READ`.
- The `data format` of the Status Attribute MUST be `SCALAR`.
- The Status Attribute SHOULD provide status information related to the Device.
- By default Status Attribute SHOULD indicate the Device State.
- The State Command MUST return the same value as would be read from the State Attribute at the time of the command execution.
- The Status Command MUST return the same value as would be read from the Status Attribute at the time of the command execution.

3.3.7 State Machine

- At startup the State Attribute value SHALL be initialised with the state of the object it represents. If not known it should be set to `UNKNOWN`.
- By default the Device State SHOULD be in ALARM if at a given time the Device State is ON and at least one of Device Attributes has the quality set to ALARM or WARNING.

Value of Device's State Attribute MAY impact how the Device responds to:

- Reading and writing of its Attribute or Pipe,
- Calling a Command.

The Device SHOULD interpret the State to allow or disallow execution of Commands, Attributes or Pipes. In case the Device decides to disallow execution it SHALL respond with throwing a `DevFailed` exception. The way how a Device allows/disallows execution of Commands, Attributes, Pipes is referred to as the State Machine.

- A Device SHOULD implement a State Machine.

3.3.8 Reserved Device Names

Although it is possible to use any Device Name matching the *Naming convention section*, some groups of names are reserved. One is Admin Device Name (see *8/Server*), defined as follows:

```
admin-device-name = domain "/" family "/" member
domain = %i"dserver"
```

See *Naming convention section* for the specification of `family` and `member`.

`admin-device-name` MUST not be used for Devices of other than `DServer` Device Class.

3.3.9 Reserved Class Names

Devices of some Classes are used to provide standard Tango Controls System services. These Classes' Names SHOULD not be used for other purposes. Below is a list of reserved Class Names:

- `%i"DataBase"`, see *6/Database*
- `%i"TangoAccessControl"`
- `%i"DServer"`

CHAPTER

FOUR

3/COMMAND

This document describes the specification of the Tango Command model.

See also: *2/Device*, *4/Attribute*

4.1 Preamble

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

4.2 Tango Command Specification

The specification of Command aims to define a standard structure and type to use in any implementation of Tango.

4.2.1 Goals

A Command in Tango Controls represents an action, a sequence or other procedure executed remotely by a Device. The meaning of commands is quite similar to the meaning of methods in object oriented design.

This specification aims to define a Command in order to be able to :

- Execute action considering an argument
- Execute an asynchronous action
- Expect a reply in any case within a certain timeout
- Be represented by meta data

Commands MAY be used for:

- Reading and writing device values, as an alternative to read and write Tango Attribute for legacy reason, however it is strongly RECOMMENDED to use Tango Attributes for device values.

4.2.2 Use Cases

The use of Command by the developer of Device allows to answer different questions:

- A command "Stop" can trigger the ending of a movement for a motor or an integration for a detector
- A command "Send" can take any SCPI command as argument and wait for the reply
- A command "Execute" can launch a macro which may take long time to be executed in the background. The reply is just an acknowledgement that the sequence will be executed. The current state of progress can be retrieved by the state of the Tango Device

4.3 Specification

A Tango Command is a strict definition of a function apply at the Device context:

- The Command SHALL have a unique identifier, called the Command Name
- The Command input or output are called Argument
- The Command SHALL have one input Argument, called ARGIN
- The Command SHALL have one output Argument, called ARGOUT

An Argument represents the input and output of a Command and its specification can only be applied to a Command as below:

- Argument SHALL have an Argument Type which set the type of the Argument Value
- Argument MAY have an Argument Description which describe the meaning of the Argument Value

An Argument Value only exists during the phase of execution of a Command. A Command MAY have 2 Arguments Values, one for ARGIN and one for ARGOUT.

The Argument Type, Argument Value, Argument Representation and Argument Description are specified below as literal representation. This SHOULD NOT constrain the implementation to use a binary format for the transport layer.

```
argument = dev-void | dev-double | dev-boolean | dev-float | dev-short | dev-long | dev-
↪long64 | dev-string | dev-uchar | dev-ushort | dev-ulong | dev-ulong64 | dev-boolean-
↪array | dev-double-array | dev-float-array | dev-short-array | dev-long-array | dev-
↪long64-array | dev-char-array | dev-string-array | dev-ushort-array | dev-ulong-array␣
↪| dev-ulong64-array | dev-long-string-array | dev-double-string-array | dev-boolean-
↪array | dev-encoded | dev-encoded-array

argument-desc = 0*CHAR

; Float Definition inspired by python float definition
; Some examples of floating point literals: 3.14 10. .001 1e100 3.14e-10
floatnumber = [ "-" ] pointfloat | exponentfloat
pointfloat = [intpart] fraction | intpart "."
exponentfloat:  ( %d1-9 DIGIT* | pointfloat) exponent
intpart:        %d1-9 DIGIT* | %d0
fraction:       "." DIGIT+
exponent:       ("e"|"E") ["+"|"-"] DIGIT+
```

(continues on next page)

(continued from previous page)

```
dev-void = dev-void-type WSP dev-void-value
dev-void-type = "DEVVOID" | "DevVoid"
dev-void-value = NULL

dev-double = dev-double-type WSP dev-double-value
dev-double-type = "DEVDOUBLE" | "DevDouble"
dev-double-value = floatnumber ; double-precision normalized numbers in IEC 60559

dev-boolean = dev-boolean-type WSP dev-boolean-value
dev-boolean-type = "DEVBOOLEAN" | "DevBoolean"
dev-boolean-value = BIT

dev-float = dev-float-type WSP dev-float-value
dev-float-type = "DEVFLOAT" | "DevFloat"
dev-float-value = floatnumber ; single-precision normalized numbers in IEC 60559

dev-short = dev-short-type WSP dev-short-value
dev-short-type = "DEVSHORT" | "DevShort"
dev-short-value = ["-"] DIGIT* ; In the range of "(2^15) - 1" to "-(2^15)".

dev-long = dev-long-type WSP dev-long-value
dev-long-type = "DEVLONG" | "DevLong"
dev-long-value = ["-"] DIGIT* ; In the range of "(2^31) - 1" to "-(2^31)".

dev-long64 = dev-long64-type WSP dev-long64-value
dev-long64-type = "DEVLONG64" | "DevLong64"
dev-long64-value = ["-"] DIGIT* ; In the range of "(2^63) - 1" to "-(2^63)".

dev-string = dev-string-type WSP dev-string-value
dev-string-type = "DEVSTRING" | "DevString"
dev-string-value = DQUOTE CHAR* DQUOTE

dev-uchar = dev-uchar-type WSP dev-uchar-value
dev-uchar-type = "DEVUCHAR" | "DevUChar"
dev-uchar-value = DIGIT* ; In the range of "0" to "(2^8) - 1". In abnf syntax it may␣
↪correspond to %d0-255

dev-ushort = dev-ushort-type WSP dev-ushort-value
dev-ushort-type = "DEVUSHORT" | "DevUShort"
dev-ushort-value = DIGIT* ; In the range of "0" to "(2^16) - 1".

dev-ulong = dev-ulong-type WSP dev-ulong-value
dev-ulong-type = "DEVULONG" | "DevULong"
dev-ulong-value = DIGIT* ; In the range of "0" to "(2^32) - 1".

dev-ulong64 = dev-ulong64-type WSP dev-ulong64-value
dev-ulong64-type = "DEVULONG64" | "DevULong64"
dev-ulong64-value = DIGIT* ; In the range of "0" to "(2^64) - 1".

dev-state = dev-state-type WSP dev-state-value
dev-state-type = "DEVSTATE" | "DevState"
```

(continues on next page)

(continued from previous page)

```
dev-state-value = "ALARM" | "INSERT" | "STANDBY" | "CLOSE" | "MOVING" | "UNKNOWN" |
↪"DISABLE" | "OFF" | "EXTRACT" | "ON" | "FAULT" | "OPEN" | "INIT" | "RUNNING"

dev-boolean-array = dev-boolean-array-type  WSP dev-boolean-array-value
dev-boolean-array-type = "DEVVARBOOLEANARRAY" | "DevVarBooleanArray"
dev-boolean-array-value = "[" dev-boolean [ ("," dev-boolean)* ] "]"

dev-double-array = dev-double-array-type  WSP dev-double-array-value
dev-double-array-type = "DEVVARDOUBLEARRAY" | "DevVarDoubleArray"
dev-double-array-value = "[" dev-double [ ("," dev-double)* ] "]"

dev-float-array = dev-float-array-type WSP dev-float-array-value
dev-float-array-type = "DEVVARFLOATARRAY" | "DevVarFloatArray"
dev-float-array-value = "[" dev-float [ ("," dev-float)* ] "]"

dev-short-array = dev-short-array-type WSP dev-short-array-value
dev-short-array-type = "DEVVARSHORTARRAY" | "DevVarShortArray"
dev-short-array-value = "[" dev-short [ ("," dev-short)* ] "]"

dev-long-array = dev-long-array-type WSP dev-long-array-value
dev-long-array-type = "DEVVARLONGARRAY" | "DevVarLongArray"
dev-long-array-value = "[" dev-long [ ("," dev-long)* ] "]"

dev-long64-array = dev-long64-array-type WSP dev-long64-array-value
dev-long64-array-type = "DEVVARLONG64ARRAY" | "DevVarLong64Array"
dev-long64-array-value = "[" dev-long64 [ ("," dev-long64)* ] "]"

dev-char-array = dev-char-array-type WSP dev-char-array-value
dev-char-array-type = "DEVVARCHARARRAY" | "DevVarCharArray"
dev-char-array-value = "[" dev-char [ ("," dev-char)* ] "]"

dev-string-array = dev-string-array-type WSP dev-string-array-value
dev-string-array-type = "DEVVARSTRINGARRAY" | "DevVarStringArray"
dev-string-array-value = "[" dev-string [ ("," dev-string)* ] "]"

dev-ushort-array = dev-ushort-array-type WSP dev-ushort-array-value
dev-ushort-array-type = "DEVVARUSHORTARRAY" | "DevVarUShortArray"
dev-ushort-array-value = "[" dev-ushort [ ("," dev-ushort)* ] "]"

dev-ulong-array = dev-ulong-array-type WSP dev-ulong-array-value
dev-ulong-array-type = "DEVVARULONGARRAY" | "DevVarULongArray"
dev-ulong-array-value = "[" dev-ulong [ ("," dev-ulong)* ] "]"

dev-ulong64-array = dev-ulong64-array-type WSP dev-ulong64-array-value
dev-ulong64-array-type = "DEVVARULONG64ARRAY" | "DevVarULong64Array"
dev-ulong64-array-value = "[" dev-ulong64 [ ("," dev-ulong64)* ] "]"

dev-long-string-array = dev-long-string-array-type WSP dev-long-string-array-value
dev-long-string-array-type = "DEVVARLONGSTRINGARRAY" | "DevVarLongStringArray"
dev-long-string-array-value =  dev-long-string-array-lvalue  dev-long-string-array-rvalue
dev-long-string-array-lvalue = dev-long-array
dev-long-string-array-rvalue = dev-string-array
```

(continues on next page)

(continued from previous page)

```
dev-double-string-array = dev-double-string-array-type WSP dev-double-string-array-value
dev-double-string-array-type = "DEVVARDOUBLESTRINGARRAY" | "DevVarDoubleStringArray"
dev-double-string-array-value =  dev-double-string-array-dvalue dev-double-string-array-
↪rvalue
dev-double-string-array-dvalue = dev-double-array
dev-double-string-array-rvalue = dev-string-array

dev-encoded = dev-encoded-type WSP dev-encoded-value
dev-encoded-type = "DEVENCODED | "DevEncoded"
dev-encoded-value =  dev-encoded-encoded-format  dev-encoded-encoded-data
dev-encoded-encoded-format = dev-string
dev-encoded-encoded-data = dev-char-array

dev-encoded-array = dev-encoded-array-type WSP dev-encoded-array-value
dev-encoded-array-type = DEVVARENCODEDARRAY | DevVarEncodedArray
dev-encoded-array-value = "[" dev-encoded [ ("," dev-encoded)* ] "]"
```

Additionally, the Command can be represented by meta data:

- A Command SHALL have a visibility level for the user called Display Level, which MAY be taken in consideration in the User Interface.

```
display-level = "OPERATOR" | "EXPERT"
```

Note: The description of a command is given by both the description of ARGIN and ARGOUT. There is no Command Description as such.

4.3.1 Reserved Commands

Although the user can define any kind of commands, some specific commands are essential to the Tango Device to work properly. The Reserved Command MUST be define as the Command Name, the ARGIN and the ARGOUT as describe below:

Reserved Command	Name	ARGIN	ARGOUT
State Command	“State”	DevVoid	DevState
Status Command	“Status”	DevVoid	DevString
Init Command	“Init”	DevVoid	DevVoid

4.3.2 Global Behaviour

- The Command SHALL always return a result when executed even when the ARGOUT is from the type DevVoid. The only exception is a “Fire-and-Forget” which does not return a result over the network after executing the Command (flag for asynchronous command execution).
- The execution of the Command SHALL be at most once
- The execution of the Command SHALL apply only in the context of a Device

4.3.3 Naming convention

- The Command Name SHALL use the following convention:

```
command-name = ALPHA *254acceptable-char
acceptable-char = ALPHA / DIGIT / underscore
underscore = %x5F
```

CHAPTER

FIVE

4/ATTRIBUTE

This document describes the Tango Attribute model specification in Tango V9.

5.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

5.2 Tango Attribute specification

This specification is intended to formally document the Tango Attribute static model. Runtime representations of Attribute in conforming Tango implementations MUST follow this specification.

5.2.1 Goals

An Attribute is a Tango concept representing in the most cases a physical quantity of a device. The main purpose of an Attribute is to provide read and (optionally) write access to this quantity.

In object oriented terminology, the Attribute corresponds to an instance variable (also called a field or a member) of a Device object. See *2/Device* for the definition of the Device.

5.2.2 Use Cases

Some example use cases of an Attribute are:

- an Attribute can represent the position of a motor device,
- an Attribute can represent the temperature reading of a thermocouple device.

5.3 Specification

An Attribute has:

- A set of static metadata that constitute Attribute's definition,
- A set of dynamically configurable properties, e.g. alarm min and max values,
- A set of runtime parameters describing the Attribute's value.

5.3.1 Attribute definition

The static metadata is part of the Attribute definition. It MUST be defined before an Attribute is created by any Tango implementation and MUST NOT change at runtime after the Attribute is initialized.

An Attribute MUST have associated following static metadata:

- *name*, a string identifying the Attribute. It MUST be unique[1] among all Attributes of a particular device. See `<attribute-name>` specification below,
- *data type*, an enumeration describing the type of the data (*9/DataTypes*),
- *data format*, an enumeration describing the dimension of the data (one of *SCALAR*, *SPECTRUM*, *IMAGE*),
- *writable*, an enumeration describing the write access to the Attribute (one of *READ*, *WRITE*, *READ_WRITE*, *READ_WITH_WRITE*).
  - An Attribute can be read from, if *writable* is one of *READ*, *READ_WRITE*, *READ_WITH_WRITE*.
  - An Attribute can be written to, if *writable* is one of *WRITE*, *READ_WRITE*.
- *display level*, an enumeration describing visibility level of the Attribute (one of *EXPERT*, *OPERATOR*).

If *data format* is *SPECTRUM*, an Attribute MUST have associated following additional static metadata:

- *max dim x*, an integer describing the maximum allowed number of data elements. The Attribute MUST be able to store up to *max dim x* data elements. It MUST be greater than 0.

If *data format* is *IMAGE*, an Attribute MUST have associated following additional static metadata:

- *max dim x*, an integer describing the maximum allowed number of data elements in dimension X. The Attribute MUST be able to store up to *max dim x* data elements in the X dimension. It MUST be greater than 0,
- *max dim y*, an integer describing the maximum allowed number of data elements in dimension Y. The Attribute MUST be able to store up to *max dim y* data elements in the Y dimension. It MUST be greater than 0.

If *writable* is *READ_WITH_WRITE*, an Attribute MUST have associated following additional static metadata:

- *writable attribute name*, a string describing the *name* of associated writable attribute. It MUST point to an attribute. The Attribute pointed to MUST have *writable* set to one of *WRITE*, *READ_WRITE*.

If *data type* is *ENUM*, an Attribute MAY have associated following additional static metadata:

[1] In current implementation it is possible to define more than one Attribute with the same name, but only one can be accessed externally.

- *enum labels*, a list of strings describing textual representation of enumerated values. It MAY be empty.

Memorized attribute

An Attribute MUST have associated following static metadata:

- *memorized*, a flag describing whether the *set value* is stored in the database and restored during Attribute initialization,
- *write hardware at init*, a flag describing whether the memorized *set value* is written to the Attribute during initialization or *init* call. It is effective only if *memorized* is also set.

A conforming implementation MUST allow to set the *memorized* flag and MUST support memorization for attributes which:

- *data format* is *SCALAR*,
- *writable* is *WRITE* or *READ_WRITE*,
- *data type* is not *STATE* or *ENCODED*.

If *memorized* is set and if database is being used, the *set value* MUST be persisted in the *__value* Attribute property upon each write to the Attribute.

If *memorized* is set and *__value* Attribute property is other than "Not used yet" (default):

- only during Attribute initialization (at startup):
  - *set value* MUST be set with a value stored in *__value* Attribute property,
- during Attribute initialization (at startup) or during an *init* command execution, if *write hardware at init* is also set:
  - value stored in *set value* MUST be written to the Attribute.

Forwarded attribute

An Attribute can be a Forwarded Attribute. A Forwarded Attribute MUST have associated a Root Attribute. This association is done with the *__root_att* Attribute property on the Forwarded Attribute and:

- It MUST be defined before attribute initialization at startup.
- It MUST NOT be changed at runtime.
- It MUST point to an existing attribute in another device.

Allowed syntax for *__root_att* is specified below as `<full-attribute-name>`.

Each read and write request to a Forwarded Attribute MUST be passed to the Root Attribute.

Each change to properties or to other metadata associated with a Forwarded Attribute MUST be reflected in the Root Attribute.

Each change to properties or to other metadata associated with the Root Attribute MUST be reflected in the Forwarded Attribute.

5.3.2 Attribute properties

An Attribute MAY have associated dynamic metadata in form of Properties (see *5/Property*).

The table below summarizes common attribute properties used by Tango. A detailed description is provided later in this section.

Property	Name in database	Type	Default value
description	`description`	string	Not specified / No description
label	`label`	string	Not specified / No label
unit	`unit`	string	Not specified / No unit / *empty*
standard unit	`standard_unit`	number	Not specified / No standard unit
display unit	`display_unit`	number	Not specified / No display unit
format	`format`	string	%6.2f (float), %d (integer), %s(string, enum), Not specified (state, encoded, boolean)
min value	`min_value`	number	Not specified
max value	`max_value`	number	Not specified
min alarm	`min_alarm`	number	Not specified
max alarm	`max_alarm`	number	Not specified
min warning	`min_warning`	number	Not specified
max warning	`max_warning`	number	Not specified
delta val	`delta_val`	number	Not specified
delta t	`delta_t`	number	Not specified / 0
rel change	`rel_change`	number	Not specified
abs change	`abs_change`	number	Not specified
archive rel change	`archive_rel_change`	number	Not specified
archive abs change	`archive_abs_change`	number	Not specified
period	`period`	number	1000
archive period	`archive_period`	number	Not specified
__value	`__value`	string	Not used yet
__root_att	`__root_att`	string	Not defined

Note: Both `delta_val` and `delta_t` MUST be set to non-default values in order for RDS (*read different than set*) alarms to work.

General properties:

- *description*, providing textual information about the Attribute,

- *label*, providing textual label to use as Attribute's name,
- *unit*, providing textual representation of Attribute's unit,
- *standard unit*, describing the conversion factor to transform Attribute's value into S.I units. It MUST be a valid numerical value,
- *display unit*, describing the conversion factor to transform Attribute's value into value usable in GUIs. It MUST be a valid numerical value,
- *format*, describing how to convert Attribute's data to a textual representation. It MUST comply to `printf` format string specification,
- *min value*, describing the minimum value of Attribute's *set value*. It MUST be defined only for Attributes with numerical *data type*. It MUST be defined only for writable Attributes. If defined: it MUST be a valid numerical value, it MUST be lower than *max value* (if specified),
- *max value*, describing the maximum value of Attribute's *set value*. It MUST be defined only for Attributes with numerical *data type*. It MUST be defined only for writable Attributes. If defined: it MUST be a valid numerical value, it MUST be greater than *min value* (if specified).

Properties for alarms:

- *min alarm*, describing the threshold value of Attribute's *read value* below which the attribute is considered as having *quality* set to ALARM. It MUST be defined only for Attributes with numerical *data type*. If defined: it MUST be a valid numerical value, it MUST be lower than *max alarm* (if specified),
- *max alarm*, describing the threshold value of Attribute's *read value* above which the attribute is considered as having *quality* set to ALARM. It MUST be defined only for Attributes with numerical *data type*. If defined: it MUST be a valid numerical value, it MUST be greater than *min alarm* (if specified),
- *min warning*, describing the threshold value of Attribute's *read value* below which the attribute is considered as having *quality* set to WARNING. It MUST be defined only for Attributes with numerical *data type*. If defined: it MUST be a valid numerical value, it MUST be lower than *max warning* (if specified),
- *max warning*, describing the threshold value of Attribute's *read value* above which the attribute is considered as having *quality* set to WARNING. It MUST be defined only for Attributes with numerical *data type*. If defined: it MUST be a valid numerical value, it MUST be greater than *min warning* (if specified),
- *delta val*, describing the maximum difference between Attribute's *read value* and *set value* after *delta t* time period, above which the attribute is considered as having *quality* set to ALARM. It MUST be defined only for writable Attributes,
- *delta t*, describing the time period (in milliseconds) after which the difference between *read value* and *set value* must be lower than *delta val*. Otherwise the attribute is considered as having *quality* set to ALARM. It MUST be defined only for writable Attributes, It MUST be defined if *delta val* is also defined.

The alarm *quality* MUST take precedence over the warning *quality*.

Properties for event reporting purposes:

- *rel change*, *abs change*, *archive rel change*, *archive abs change*, describing the threshold values for relative and absolute change in Attribute's *read value* above which a *CHANGE* event or *ARCHIVE* event MUST be reported. It MUST be either a valid numerical value or a pair of valid numerical values separated by a `,`. If one value is specified, it MUST be used for both negative and positive change. If two values are specified, the first MUST be used for negative change and the second MUST be used for positive change. It MUST be defined only for Attributes with numerical *data type*,
- *period*, *archive period*, describing the minimum time period in milliseconds after which a *PERIODIC* or *ARCHIVE* event MUST be reported, regardless of Attribute's *read value*. If *period* is not specified, a default value of 1000 ms is used,

If relative change is specified, it MUST be expressed as a percentage change relative to the value reported in the previous event, e.g. a relative change of 1 will generate an event when:

```
(current value - previous value) / previous value > 1%
```

See *Attribute events section* for more details.

```
rel-change = change
abs-change = change

archive-rel-change = change
archive-abs-change = change

number = [ "-" ] 1*DIGIT [ "." ] *DIGIT
change = number [ "," number ]

period = 1*DIGIT
archive-period = period
```

5.3.3 Attribute runtime parameters

At given point in time an Attribute MUST have associated:

- *quality*, an enumeration describing the state *read value* (one of *VALID*, *INVALID*, *ALARM*, *CHANGING*, *WARNING*),
- *read value*, an object representing the value of the Attribute. It MUST conform to *data format* and *data type*,
- *read dim x*, an integer describing the number of data elements in *read value* in X dimension. If *data format* is *SCALAR*, it MUST be either 0 or 1. If *data format* is *SPECTRUM* or *IMAGE*, it MUST be between 0 and *max dim x*.
- *read dim y*, an integer describing the number of data elements in *read value* in Y dimension. If *data format* is *SCALAR* or *SPECTRUM*, it MUST be 0. If *data format* is *IMAGE*, it MUST be between 0 and *max dim y*.

Additionally, if *writable* is *WRITE* or *READ_WRITE*, at a given point in time an Attribute MUST have associated:

- *set value*, an object representing the value set to the Attribute. It MUST conform to *data format* and *data type*,
- *write dim x*, an integer describing the number of data elements in *set value* in X dimension,
- *write dim y*, an integer describing the number of data elements in *set value* in Y dimension.

5.3.4 Attribute aliases

A *full attribute name* consists of *device name* and *attribute name* separated by a "/" character.

See *2/Device* for the definition of `<device-name>`.

An Attribute MAY have *alias* associated with it. An *alias* is a string which can be used in place of *full attribute name* to address the Attribute.

An *alias* MUST follow the same naming specification as an *attribute name*.

See [16/TangoResourceLocator/Alias](../16/TangoResourceLocator.md### Alias) for common Alias properties.

5.3.5 Attribute events

An Attribute can send following events:

- *PERIODIC*,
- *CHANGE*,
- *ARCHIVE*,
- *DATA READY*,
- *ATTR CONF*,
- *USER*,

Events can be sent automatically by a polling mechanism, automatically by the Tango system or manually from the device server code.

The polling mechanism can send PERIODIC, CHANGE and ARCHIVE events. In order to send events via the *polling mechanism*, the polling for the Attribute MUST be enabled. See *Attribute properties* section for conditions upon which the events are sent with the polling enabled. See *10/RequestReply* for more details on polling configuration.

The Tango system MUST send ATTR CONF event whenever the Attribute configuration is changed. See *Attribute properties* section for a list of modifiable attribute Properties.

The device server code can manually send *CHANGE*, *ARCHIVE*, *DATA READY* and *USER* events without the need to configure the polling for the Attribute.

5.3.6 Attribute naming schema

Formal specification of Attribute name is given below:

```
full-attribute-name = device-name "/" attribute-name
attribute-name = ALPHA *254acceptable-char
acceptable-char = ALPHA / DIGIT / underscore
underscore = %x5F
```

CHAPTER

SIX

5/PROPERTY

This document describes the Tango Property model.

See also: *1/Tango*, *2/Device*, *4/Attribute*, *6/Database*, *9/DataTypes*

6.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

6.2 Tango Property Specification

This specification is intended to formally document the Tango Property model. Runtime representation of Property in conforming Tango implementations MUST follow this specification.

6.2.1 Goals

A Property represents a configuration parameter in Tango.

6.2.2 Use Cases

Here are some Property use cases:

- To connect a real network equipment a Property can define the target IP address or hostname
- To change the representation of an information in order to be more user friendly a Property can define a new data format for the graphical user interface to convert the raw data
- To configure some alarms on Attribute Value (See *4/Attribute*)
- To store the Attribute value of a Memorized Attribute (See *4/Attribute*)
- To configure the labels associated to the possible values of a DevEnum Attribute

6.3 Specification

A Property is a strict definition of a pair of key/value:

- The Property SHALL have one key, called Property Name
- The Property SHALL have one value, which could be empty, called Property Value

There are 4 kinds of Properties:

- Class Property: a Property related to a Device Class (*2/Device*,).
- Device Property: a Property related to a Device (*2/Device*,).
- Attribute Property: a Property related to an Attribute (*4/Attribute*).
- Free Property: a Property defined for the entire Control System (*1/Tango*) i.e not related to any Class, Device or Attribute. A Free Property is attached to an entity called a Free Object.

A Property MAY be persisted in the Tango Database (See *1/Tango* and *6/Database*) or into a file if no Tango Database is used.

Device Properties and Class Properties MAY have a *Default Value* which is used if the Property Value has not been persisted.

A Tango Device Property MAY be defined as *Mandatory in Database*. This metadata can be used when the device server programmer requires this Property to be persisted. A Device Property defined as *Mandatory in Database* SHALL not have a Default Value.

Device Property and Class Property MAY have a Type metadata. It is RECOMMENDED to support the following Types for Device Properties and Class Properties (See *9/DataTypes*):

- DevBoolean
- DevShort
- DevUShort
- DevLong
- DevULong
- DevLong64
- DevULong64
- DevFloat
- DevDouble

- DevString
- DevVarShortArray
- DevVarLongArray
- DevVarLong64Array
- DevVarFloatArray
- DevVarDoubleArray
- DevVarStringArray

Device and Class Properties MUST share the same scope. This means that a Device Property will always take precedence over a Class Property of the same name.

It is RECOMMENDED to provide a way to retrieve a persisted Property Value and to convert it to the above types.

It is RECOMMENDED to support *Not a Number (NaN)* as well as *-infinity (-inf)* and *+infinity (+inf)* DevFloat and DevDouble values.

6.3.1 Naming convention

- The Property's Name SHALL use the following convention:

```
device_property_name = ALPHA *254acceptable-char
class_property_name = ALPHA *254acceptable-char
free_property_name = ALPHA *254acceptable-char
attribute_property_name = (ALPHA / underscore) *254acceptable-char
acceptable-char = ALPHA/DIGIT / underscore
underscore = %x5F
```

CHAPTER SEVEN

6/DATABASE

This document describes the Tango Database specification.

See also: *1/Tango*, *2/Device*, *4/Attribute*, *5/Property*, *8/Server*

7.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

7.2 Tango Database specification

This specification is intended to formally document the Tango Database. Runtime implementations of the Database in conforming Tango implementations MUST follow this specification.

7.2.1 Goals of the Database

- provide a means of defining and querying devices.
- provide a means for device servers to be configured from central control
- serve as nameserver such that the Devices are accessible by name

7.2.2 Features of the Database

- store configuration data (properties) used at startup of a device server
- act as nameserver by storing the ITR
- act as permanent store of dynamic settings which need to be memorized
- ensure the uniqueness of device name and of aliases
- keep list of controlled servers which are to be started and run on a particular host
- limited set of functionalities provided by file as alternative to network database

7.2.3 Use Cases of the Database

- register a new device server
- store the last user setting point of the current provided by a power supply
- define persistent configuration properties to be used by a camera every time it starts up
- allow a subscriber or client to get connected to a stepper motor
- define properties for all instances of a class of vaccuum pump
- provide the ITR of a device such as an electrometer, so a user can check whether it is online or not
- a user would like to rollback the configuration of a Device when its initialization is failing
- remove "empty" Devices from Database to clean it up
- get all Devices of a given class of temperature gauges
- user wants to see the history (up to a configurable depth) of a Property
- a monitor tool wants the Info of a not running detector Device (exported, host, server)
- a boot-up process wants to get Starter Level configuration for a list of Devices

7.3 Specification

The database SHALL provide a means of defining and querying devices.

As the Tango database service is a central service of the Tango Control System, it SHALL be designed to be highly available and support a high level of client requests (e.g the database must be able to treat thousands of requests a second).

The database service SHALL be implemented as a Tango Device Server.

The database service SHALL be implemented with:

- a frontend: as Tango Device Server
- a backend: a query and persistency engine (i.e. SQL or NoSQL engine), or a lighter (i.e. reduced set of functionalities) engine with a single file per device server.

The Database Device SHALL have no Class Properties or Device Properties.

In the Tango Control System, the database service SHALL be available on a pre-known network address.

Clients and subscribers SHOULD access the database service via TANGO commands requested on the database device.

As a Tango Device Server, the database SHALL provide State and Status.

A limited set of the database functionality to support Tango Device Server configuration SHALL be available as a read-only file. This is an option for use in the absence of a network service.

The database SHALL enforce uniqueness of device names and aliases.

The Database SHALL enforce the naming convention defined in all Tango RFCs for device name, the command name, the attribute name, the property name, the device alias name and the device server name.

The device name, its domain, member and family fields and its alias maximum sizes SHALL BE at least:

- device name: 255
- domain field: 85
- family field: 85
- member field: 85
- device alias name: 255

7.4 Device Export and Unexport protocols

The Device Export and Unexport protocols are used during the Device Export and Device Unexport sequences respectively (*8/Server*) when the Device transitions from the Not Exported phase to the Exported phase (*2/Device*) and vice versa.

These protocols describe the interactions between:

- when the Device Server in charge of the management and communication of the Device is starting/stopping and
- the Database

7.4.1 Device Export protocol

The Export protocol consists of signing up a Device in a Database so any Client of the Tango Control System could find the information necessary to access it.

The Export protocol MUST be initiated by a Device Server.

During the execution of this protocol the following information MUST reach the Database:

- Device name
- Interoperable Tango Reference (ITR) e.g. IOR in CORBA
- Device server process host name
- Device server process PID or string `null`
- Device server process version

Currently it is implemented by executing the `DbExportDevice` command on the Database device.

7.4.2 Device Unexport protocol

The Unexport protocol consists of marking a Device as inaccessible in the Database so any Client of the Tango Control System querying the Database knows that the Device is not accessible anymore.

The Unexport protocol SHALL be initiated by a Device Server.

In case a Device Server could not execute the protocol e.g. due to a crash of the Device Server process then another Client of the Tango Control System MAY execute it in order to maintain consistency.

During the execution of this protocol the following information MUST reach the Database:

- Device name

Currently, it is implemented by executing the `DbUnExportDevice` command and `DbUnExportServer` (for unexporting all the Devices previously exported by the Device Server) on the Database Device.

7.4.3 Getting information to access a Device

Any Client of the Tango Control System SHALL be able to ask a Database for information necessary to access a Device at any time, unless the Database service is not accessible.

The Database service MUST provide information on how to access a Device or that a Device is not accessible. This information includes:

- Device name
- Interoperable Tango Reference (ITR)
- Device version
- Device Server process name
- host name where the Device Server runs
- Device Class name
- exported flag
- Device Server process PID

Currently it is implemented by executing the `DbImportDevice` command on the Database Device.

7.5 List of Commands Grouped by Functionality

7.5.1 Devices

The database SHALL implement the following device-related commands:

- DBAddDevice
- DbDeleteDevice
- DbGetClassForDevice
- DbGetClassInheritanceForDevice
- DbGetDeviceClassList
- DbGetDeviceDomainList
- DbGetDeviceFamilyList

- DbGetDeviceInfo
- DbGetDeviceList
- DbGetDeviceWideList
- DbGetDeviceMemberList
- DbGetDeviceServerClassList

7.5.2 Classes

The database SHALL provide a means of querying device server classes.

The database SHALL implement the following class-related commands:

- DbGetClassList
- DbGetDeviceClassList
- DbGetDeviceServerClassList

7.5.3 Properties

The database SHALL provide a means of setting and getting properties for classes, devices, attributes, pipes and free objects.

The database SHALL provide a means to retrieve the history of property value settings.

The database SHALL implement the following property-related commands:

ClassAttributeProperty

- DbDeleteClassAttributeProperty
- DbGetClassAttributeProperty
- DbGetClassAttributeProperty2
- DbGetClassAttributePropertyHist
- DbPutClassAttributeProperty
- DbPutClassAttributeProperty2

ClassProperty

- DbDeleteClassProperty
- DbGetClassProperty
- DbGetClassPropertyHist
- DbGetClassPropertyList
- DbPutClassProperty

DeviceAttributeProperty

- DbDeleteDeviceAttributeProperty
- DbGetDeviceAttributeProperty
- DbGetDeviceAttributeProperty2
- DbGetDeviceAttributePropertyHist
- DbPutDeviceAttributeProperty
- DbPutDeviceAttributeProperty2
- DbDeleteAllDeviceAttributeProperty

DeviceProperty

- DbDeleteDeviceProperty
- DbGetDeviceProperty
- DbGetDevicePropertyHist
- DbGetDevicePropertyList
- DbPutDeviceProperty

Free Object Property

- DbDeleteProperty
- DbGetProperty
- DbGetPropertyHist
- DbGetPropertyList
- DbPutProperty

ClassPipeProperty

- DbGetClassPipeProperty
- DbDeleteClassPipeProperty
- DbPutClassPipeProperty
- DbGetClassPipePropertyHist

DevicePipeProperty

- DbGetDevicePipeProperty
- DbDeleteDevicePipeProperty
- DbDeleteAllDevicePipeProperty
- DbPutDevicePipeProperty
- DbGetDevicePipePropertyHist

7.5.4 Attributes

The database SHALL provide for the persistent storage of Attributes of classes and devices.

The database SHALL implement the following attribute-related commands:

ClassAttribute

- DbDeleteClassAttribute
- DbGetClassAttributeList

DeviceAttribute

- DbDeleteDeviceAttribute
- DbGetDeviceAttributeList

7.5.5 Aliases

The database SHALL provide a means of setting and getting a mapping from alias to device name.

The database SHALL provide a means of setting and getting a mapping from alias to device attribute name.

The database SHALL implement the following attribute-related commands:

AttributeAlias

- DbDeleteAttributeAlias
- DbGetAttributeAlias
- DbGetAttributeAliasList
- DbPutAttributeAlias
- DbGetAttributeAlias2
- DbGetAliasAttribute

DeviceAlias

- DbDeleteDeviceAlias
- DbGetAliasDevice
- DbGetDeviceAlias
- DbGetDeviceAliasList
- DbPutDeviceAlias

7.5.6 Hosts/Servers

The database SHALL provide a means of defining hosts and servers.

The database SHALL implement the following commands:

Host

- DbGetHostList
- DbGetHostServerList
- DbGetHostServersInfo

Server

- DBAddServer
- DbDeleteServer
- DbGetServerList
- DbGetServerNameList
- DbRenameServer

ServerInfo

- DbDeleteServerInfo
- DbGetServerInfo
- DbPutServerInfo

7.5.7 Pipes

The database SHALL provide a means of managing information about pipes.

The database SHALL implement the following pipe-related commands:

ClassPipe

- DbDeleteClassPipe
- DbGetClassPipeList

DeleteDevicePipe

- DbGetDevicePipeList

7.5.8 Exported Devices

The database SHALL provide a means to register the network address of a running Device Server and events.

The database SHALL implement the following free registry-related commands:

- DbExportDevice
- DbUnExportDevice
- DbGetDeviceExportedList
- DbGetExportedDeviceListForClass
- DbImportDevice
- DbExportEvent
- DbImportEvent
- DbUnExportEvent
- DbUnExportServer

7.5.9 Miscellaneous

The database SHALL provide the following miscellaneous commands.

- DbGetInstanceNameList
- DbGetForwardedAttributeListForDevice
- DbInfo
- ResetTimingValues
- DbGetDataForServerCache
- DbMySqlSelect
- DbGetCSDbServerList

7.6 List of Database commands and description

7.7 Database as a file

For Device servers not able to access the Tango database (most of the time due to network routing or for cyber-security reason), it is possible to start them using a file instead of a real database. This is done via the Device server command line option:
`-file=<file name>`

The file based database is OPTIONAL.

In this case, the Database SHALL handle the following functionality using the specified file:

- 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

The following set of limitations MAY exist:

- 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 ports

7.7.1 File syntax

The file is RECOMMENDED to be an ASCII file. The syntax of the file is defined below.

Generally, comments start with the '#' character and a blank line is ignored.

File syntax: Devices definition.

DEVICE is the keyword to declare a device(s) definition sequence. The general syntax SHALL be

```
<DS name>/<inst name>/DEVICE/<Class name>: dev1,dev2,dev3
```

Device(s) name can follow on next line if the last line character is \

File syntax: Device property definition.

The general device property SHALL be

```
<device name>-><property name>: <property value>
```

In case of array, the array element delimiter is the character `,`. Array definition can be split over several lines if the last line character is \. Allowed characters after the `:` delimiter are space, tabulation or nothing.

A device string property with special characters (spaces). A pair of `"` characters are used to delimit the string.

File syntax: Device attribute property definition.

The general device attribute property syntax SHALL be

```
<device name>/<attribute name>-><property name>: <property value>
```

Allowed characters after the `:` delimiter are space, tabulation or nothing.

File syntax: Class property definition.

The general class property syntax SHALL be

```
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. The " characters around the property value are mandatory due to the / character contains in the property value.

CHAPTER

EIGHT

7/THE TANGO PIPE SPECIFICATION

This specification describes the Tango Pipe feature of a Device.

8.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

8.2 Goals

A Tango Pipe is an Attribute *4/Attribute* with a flexible data structure, name and description. Unlike Commands or Attributes, a Pipe does not have a pre-defined data type. Tango Pipe data types may be a mixture of any of the basic Tango data types (or array of) and may change every time a pipe is written.

8.3 Use Case

1. To define complex data structures to act as a single Attribute
2. To pass data structures of arbitrary type and size

8.4 Specification

A Device MAY define zero, one or more Pipes.

Pipe MUST have static metadata:

- name
- display-level, an enumeration describing visibility level of the Pipe (one of EXPERT, OPERATOR)
- writable

```
name = attribute-name
display-level = "OPERATOR" / "EXPERT"
writable = "PIPE_READ" / "PIPE_WRITE"
```

Where `attribute-name` is defined in *4/Attribute*.

Pipe with writable="PIPE_READ" MUST be read only.

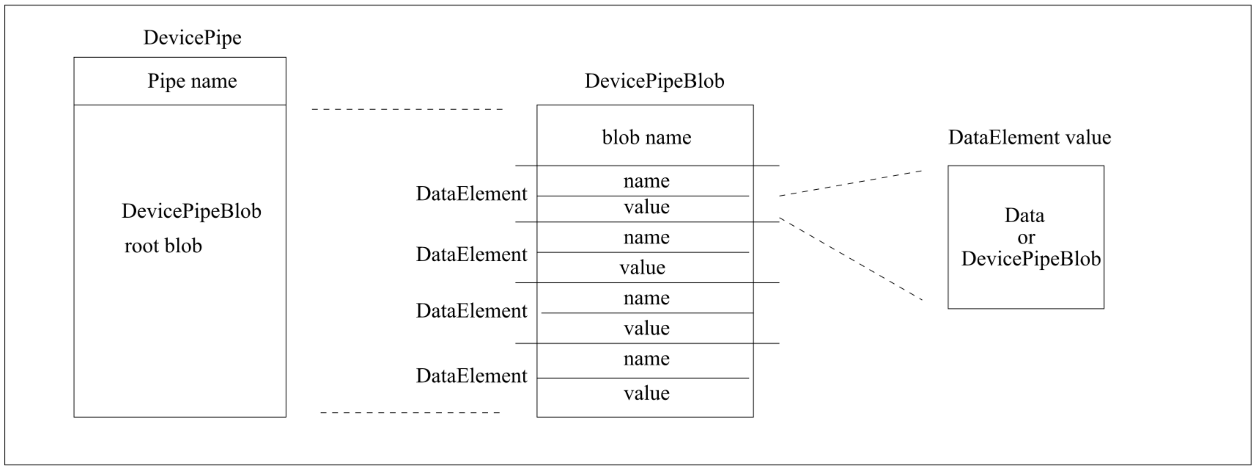
Pipe MUST have modifable metadata:

- description
- label

```
description = DevString
label = DevString
```

Where `DevString` is defined in *9/DataTypes*.

Pipe data structure MUST be of type `DevPipeBlob`, as specified in *9/DataTypes*. See image below.



V10 NOTE: Attribute can be considered as a derivative of a Pipe.

8.4.1 Pipe events

Pipe MUST NOT be polled.

Pipe MAY emit an event via Server API call.

8.4.2 Server/Client APIs

Server API MUST provide a set of methods for a Server Client code to define a Pipe.

Server API MUST provide a way to fill the Pipe with data.

Client API MAY provide a way to obtain Pipe schema.

Client API MUST provide methods for a client to fetch Pipe data from a Device.

Client API MUST provide methods to traverse Pipe structure.

CHAPTER

NINE

8/DEVICE SERVER MODEL

This document provides the specification of the Device Server.

See also: *2/Device*

9.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

9.2 Tango Device Server specification

This specification formally documents the Tango Device Server model. The runtime implementation of the Device Server model in a Tango compliant implementation MUST follow this specification.

9.2.1 Goals

In a Tango Control System a Device Server is in charge of managing the life cycle and the communication protocol of one or more Tango Devices. The Device Server can also offer different services to the clients as well as for the Device.

Additionally, it aims to:

- Provide a blackbox that registers the operations executed on every Device
- Be usable as a Device via the Administration Device referred to as DServer

9.2.2 Use Cases

The main use cases for DServer are:

- to manage several Devices in the same process.
- to share the same channel of communication
- to provide the connection protocols for accessing the managed Device
- to provide different services like logging
- to serialise the operations per Device
- to export or unexport Devices to the central registry

9.3 Specification

The Device Server manages the communication with one or several Devices. Additionally it enforces the compatibility of behaviour with the Tango Control System and offers different meta services.

The Device Server SHALL implement the following specifications to control and monitor the Device:

- A Device Server represents the process which manages one or several Devices
- A Device Server SHALL manage at least one Device.
- A Device Server MAY manage Devices belonging to different Classes
- A Device MUST NOT be managed by more than one Device Server
- A Device Server instance reference SHALL be unique in a Tango Control system
- The communication to the Device Server SHALL be done through a Tango Device DServer also-called Administration Device or Admin Device.

The specification defines the following concepts which are used by the Administration Device:

- A Device Server is a process which creates and serves devices
- Several instances of the same Device Server MAY exist in a Tango Control System
- Instance is an unique identifier which together with the Device Server name identifies the unique process in the Tango Control System
- A Device Server Instance CAN only have one DServer Device instance

The name of the DServer / Administration Device SHALL follow this convention:

```
administration-device-name = dserver "/" server "/" instance
dserver = "dserver" ; dserver is a reserved domain following the device-name convention.
server = family
instance = member
```

- The family part of the Administration Device Name SHALL be the Server name
- The member part of the Administration Device Name SHALL be the Instance

9.3.1 DServer interface

Property Name	description
polling_threads_pool_size	See *10/RequestReply Cache*
polling_threads_pool_conf	See *10/RequestReply Cache*
polling_before_9	See *10/RequestReply Cache*

Command Name	Argument input	Argument Output	description
AddLoggingTarget	DevVarStringArray	DevVoid	Define the level of Loggin (*14/Logging*
AddObjPolling	DevVarLongStringArray	DevVoid	Polling related command (*10/RequestReply*)
DevLockStatus	DevString	DevVarLongStringArray	Request Replay command (*10/RequestReply*
DevPollStatus	DevString	DevVarStringArray	Polling related command (*10/RequestReply*)
DevRestart	DevString	DevVoid	
EventConfirmSubscription	DevVarStringArray	DevVoid	
EventSubscriptionChange	DevVarStringArray	DevLong	
GetLoggingLevel	DevVarStringArray	DevVarLongStringArray	
GetLoggingTarget	DevString	DevVarStringArray	
Init	DevVoid	DevVoid	
Kill	DevVoid	DevVoid	
LockDevice	DevVarLongStringArray	DevVoid	
PolledDevice	DevVoid	DevVarStringArray	
QueryClass	DevVoid	DevVarStringArray	
QueryDevice	DevVoid	DevVarStringArray	
QuerySubDevice	DevVoid	DevVarStringArray	
QueryWizardClassProperty	DevString	DevVarStringArray	
QueryWizardDevProperty	DevString	DevVarStringArray	
ReLockDevices	DevVarStringArray	DevVoid	
RemObjPolling	DevVarStringArray	DevVoid	
RemoveLoggingTarget	DevVarStringArray	DevVoid	
RestartServer	DevVoid	DevVoid	
SetLoggingLevel	DevVarLongStringArray	DevVoid	
StartLogging	DevVoid	DevVoid	
StartPolling	DevVoid	DevVoid	
State	DevVoid	DevState	
Status	DevVoid	DevString	
StopLogging	DevVoid	DevVoid	
StopPolling	DevVoid	DevVoid	
UnLockDevice	DevVarLongStringArray	DevLong	
UpdObjPollingPeriod	DevVarLongStringArray	DevVoid	
ZmqEventSubscriptionChange	DevVarStringArray	DevVarLongStringArray	

9.3.2 The Device Export sequence

The Device Export sequence transitions a Device from the Not Exported phase to the Exported phase. It is intended to prepare a Device for operation and accessibility in the Tango Control System:

- The Device Server MUST take in charge of the Device Export sequence
- The Device Export sequence SHOULD bring the Device at the same State than after the Init Command.
- The Device Export sequence SHOULD register the Device to the Tango Control system by executing the Device Sign Up of the Database(*6/Database*), in order to be accessible locally and via network.

9.3.3 The Device Unexport sequence

The Device Unexport sequence transitions a Device from the Exported phase to the Not Exported phase. It is intended to prevent a Device from operation and accessibility in the Tango Control System.

- The Device Server MUST take in charge of the Device Unexport sequence
- The Device Unexport sequence SHALL gracefully stop the Device operation, even if there are requests in progress.
- The Device Unexport sequence SHALL gracefully stop the communication with any Tango Client
- The Device Unexport sequence SHOULD unregister from the Tango Control system by executing the Device unexport of the Database(*6/Database*), in order to notify that all managed Device are no longer accessible

CHAPTER

TEN

9/DATA TYPES

This document describes Tango Controls Data Types specification version Tango V9.

See also: *1/TANGO*, *3/Command*, *4/Attribute*, *5/Property*, *6/Database*, *7/Pipe*,

10.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

10.2 Tango Data Types Specification

Data Types specify data formats exchanged between Tango Controls components.

10.2.1 Goals

Data Types specification aims to standardise a way how different kinds of information sre entered, displayed and processed within a Tango Controls system irrespective of its encoding in a transport protocol or computer architecture. It also allows to distinguish between different kinds of numeric data, string data, enumerations or variable length arrays of basic types or stuctures.

10.2.2 Use Cases

There are many use cases for Data Types. Some of them are listed below:

- To specify expected data format for attributes, commands and pipes.
- To display in the same way a numeric value read from a device when some of them are using big-endian encoding and others use little-endian encoding.
- To limit information processed to a practical level by choosing a Data Type of certain precision. For example, a value of Data Type DevShort typically provides less precise information and consumes less computing resources than a value of Data Type DevLong.

10.2.3 DataType

- Any value of Properties, Attributes, Pipes' fields and Commands' input and output arguments SHALL have defined its DataType.
- DataType SHALL be one of the following:

```
DataType = DevVoid

DataType =/ DevBoolean

DataType =/ DevShort / DevLong / DevLong64
DataType =/ DevUChar / DevUShort / DevULong / DevULong64

DataType =/ DevFloat / DevDouble

DataType =/ DevString

DataType =/ DevVarBooleanArray / DevVarDoubleArray / DevVarFloatArray
DataType =/ DevVarShortArray / DevVarLongArray / DevVarLong64Array / DevVarCharArray
DataType =/ DevVarStringArray
DataType =/ DevVarUShortArray / DevVarULongArray / DevVarULong64Array

DataType =/ DevEncoded / DevVarEncodedArray

DataType =/ DevVarLongStringArray / DevVarDoubleStringArray

DataType =/ DevState / DevVarStateArray

DataType =/ DevEnum

DataType =/ DevPipeBlob

DataType =/ DevFailed
```

10.2.4 Numeric

DevBoolean

DevBoolean data type values represent logical state of True or False.

- DevBoolean SHALL use the following name:

```
DevBoolean = "DevBoolean"
```

- Values of DataType DevBoolean SHALL carry one bit of information.

Integer data types

DevShort, DevLong, DevLong64 represent signed integer values. DevUChar, DevUShort, DevULong and DevULong64 represent unsigned values.

- Integer DataType SHALL have the following names:

```
DevShort = "DevShort"
DevLong = "DevLong"
DevLong64 = "DevLong64"
DevUChar = "DevUChar"
DevUShort = "DevUShort"
DevULong = "DevULong"
DevULong64 = "DevULong64"
```

- Values of integer data types SHALL be in defined ranges and the information they carry SHALL have number of bits as specified in the table below:

DataType	Range	Bits of information
DevShort	-(2^15) to (2^15)-1	16
DevLong	-(2^31) to (2^31)-1	32
DevLong64	-(2^63) to (2^63)-1	64
DevUChar	0 to 255	8
DevUShort	0 to (2^16)-1	16
DevULong	0 to (2^32)-1	32
DevULong64	0 to (2^64)-1	64

Floating point data types

DevFloat and DevDouble represent floating point values.

- DevFloat and DevDouble SHALL use the following names:

```
DevFloat = "DevFloat"
DevDouble = "DevDouble"
```

- Values of DevFloat type SHALL be of IEEE 754 single precision floating-point format and MAY be encoded in 32 bits.
- Values of DevDouble type SHALL be of IEEE 754 double precision floating-point format and MAY be encoded in 64 bits.

String data type

DevString values represent strings (sequence of characters).

- DevString SHALL use the following names:

```
DevString = "DevString"
```

- Values of DevString SHALL be according to the string-value rule defined as follows:

```
string-value = *CHAR
```

Sequence data types

DevVarBooleanArray, DevVarShortArray, DevVarLongArray, DevVarLong64Array, DevVarCharArray, DevVarUShortArray, DevVarULongArray, DevVarULong64Array, DevVarFloatArray, DevVarDoubleArray, DevVarStringArray, DevVarEncodedArray, DevVarStateArray are Sequence DataTypes. Sequence data types are sequences (arrays) of values (elements) of matching DataType, which are respectively, DevBoolean, DevShort, DevLong, DevLong64, DevUChar, DevUShort, DevULong, DevULong64, DevFloat, DevDouble, DevString, DevEncoded, DevState.

- Names of the sequence data types SHALL be as follows:

```
DevVarBooleanArray = "DevVarBooleanArray"
DevVarShortArray = "DevVarShortArray"
DevVarLongArray = "DevVarLongArray"
DevVarLong64Array = "DevVarLong64Array"
DevVarCharArray = "DevVarCharArray"
DevVarUShortArray = "DevVarUShortArray"
DevVarULongArray = "DevVarULongArray"
DevVarULong64Array = "DevVarULong64Array"
DevVarFloatArray = "DevVarFloatArray"
DevVarDoubleArray = "DevVarDoubleArray"
DevVarStringArray = "DevVarStringArray"
DevVarEncodedArray = "DevVarEncodedArray"
DevVarStateArray = "DevVarStateArray"
```

- Value of sequence DataType SHALL follow rule:

```
sequence-value = *element
```

Where <element> follows specification of value of the matching <DataType>

- Sequence DataType SHALL provide an interface to determine number of its elements.
- Sequence DataType SHALL provide an interface to index its elements.

Structures

Value of the DevEncoded type is a structure to carry binary data with application and context specific meaning.

- Name of DevEncoded type SHALL be as follows:

```
DevEncoded = "DevEncoded"
```

- Value of <DevEncoded> SHALL be according to the <encoded-value> rule:

```
encoded-value = encoded_format encoded_data
                ; elements order implied by the above rule MAY be ignored
```

;where <encoded_format> has type DevString and <encoded_data> has type DevVarCharArray.

DevVarLongStringArray and DevVarDoubleStringArray values are structures to carry sequences of numbers (integer or double precision floating point, respectively) along with sequences of text/string data.

- Names DevVarLongStringArray and DevVarDoubleStringArray SHALL follow the following rules:

```
DevVarLongStringArray = "DevVarLongStringArray"
DevVarDoubleStringArray = "DevVarDoubleStringArray"
```

- Values of DevVarLongStringArray and DevVarDoubleStringArray SHALL follow rules <long-string-value> and <double-string-value>, respectively:

```
long-string-value = lvalue svalue
double-string-value = dvalue svalue
                      ; elements order implied by the above rules MAY be ignored
```

;where <lvalue> has type DevVarLongArray, <dvalue> has type DevVarDoubleArray and <svalue> has type DevVarStringArray.

State

DevState type standardizes the possible states of Device. Its values are from predefined set of 14 names.

- DevState name SHALL be according to the following rule:

```
DevState = "DevState"
```

- DevState type SHALL be an enumeration type with labels following the rule <dev-state-label>:

```
dev-state-label = "ALARM" / "INSERT" / "STANDBY" / "CLOSE" / "MOVING" / "UNKNOWN" /
↪"DISABLE" / "OFF"
dev-state-label =/ "EXTRACT" / "ON" / "FAULT" / "OPEN" / "INIT" / "RUNNING"
```

Enumeration

DevEnum data type allows to assign string Labels to DevShort values. It is valid only for Attributes.

- DevEnum name SHALL follow the following rule:

```
DevEnum = "DevEnum"
```

- Label values of DevEnum SHALL follow the rule <devenum-label>:

```
devenum-label = 1*VCHAR
```

- DevEnum labels SHALL be unique in the context of an Attribute.
- Each of DevEnum labels SHALL correspond to one unique DevShort value.
- Related DevShort values SHALL be consecutive and SHALL start with 0.
- Values of DevEnum DataType SHALL be transported as values of DevShort.
- For any Attribute of DevEnum data type, list of Labels SHALL be available as `enum_labels` Attribute property (see *4/Attribute*).
- Tango Controls SHALL allow a Tango Client to retrieve labels associated with the DevShort value.

Pipe

DevPipeBlob is a Data Type to transfer data related to Pipes (see *7/Pipe*).

- DevPipeBlob SHALL have the following name:

```
DevPipeBlob = "DevPipeBlob"
```

- A value of DevPipeBlob type SHALL follow the <pipe-blob-value> rule:

```
pipe-blob-value = name blob-data
                  ; elements order implied by the above rule MAY be ignored

name = 1*VCHAR

blob-data = *blob-data-element

blob-data-element = name (value / blob-data)
                    ; elements order implied by the above rule MAY be ignored
```

Where <value> SHALL be a value of one of the following data types: DevBoolean, DevShort, DevLong, DevLong64, DevFloat, DevDouble, DevChar, DevUShort, DevULong, DevULong64, DevString, DevState, DevState, DevEncoded, DevVarBooleanArray, DevVarShortArray, DevVarLongArray, DevVarLong64Array, DevVarFloatArray, DevVarDoubleArray, DevVarCharArray, DevVarUShortArray, DevVarULongArray, DevVarULong64Array, DevVarStringArray, DevVarStateArray, DevState, DevVarEncodedArray.

Exceptions

Results of failed operations (exceptions) within the Tango Controls are sent as values/messages of DevFailed or MultiDevFailed data type.

- DevFailed type name is as follows:

```
DevFailed = "DevFailed"
```

- Any value of DevFailed type SHALL follow <dev-failed-value> rule:

```
dev-failed-value = 1*error

error = reason severity desc origin

reason = *VCHAR ; SHOULD be a symbolic name, without whitespace, which points the␣
↪reason for the failure and is standardized for some errors thrown by the API.␣
↪Examples: "API_AttrOptProp", "API_AttrNotPolled","API_CmdNotPolled", "API_
↪CommandNotFound"
severity = "WARN" | "ERR" | "PANIC"
desc = *VCHAR ; SHOULD describe the failure in more details
origin = *VCHAR ; SHALL identify the operation or the tango object which caused the␣
↪failure, eg. function name
```

- MultiDevFailed type name is as follows:

```
MultiDevFailed = "MultiDevFailed"
```

- Any value of MultiDevFailed type SHALL follow <multi-dev-failed-value> rule:

```
multi-dev-failed-value = 1*named-dev-error

named-dev-error = name index-in-call 1*error

name = *VCHAR
index-in-call = num-val ; of long data type (32 bits) in the range of 0 to (2^31)-1

error = reason severity desc origin

reason = *VCHAR ; SHOULD be a symbolic name, without whitespace, which points the␣
↪reason for the failure and is standardized for some errors thrown by the API.␣
↪Examples: "API_AttrOptProp", "API_AttrNotPolled","API_CmdNotPolled", "API_
↪CommandNotFound"
severity = "WARN" | "ERR" | "PANIC"
desc = *VCHAR ; SHOULD describe the failure in more details
origin = *VCHAR ; SHALL identify the operation or the tango object which caused the␣
↪failure, eg. function name
```

CHAPTER

ELEVEN

10/REQUEST REPLY

This document describes The Request-Reply protocol which provides a two way communucation between the Tango Controls clients and servers.

See also: Y/OtherTemplate

11.1 Preamble

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

11.2 The Request-Reply protocol Specification

The Request-Reply protocol defines:

- types and content of messages sent between Tango Controls objects,
- communication sequences,
- rules for establishing and tearing-down network connections.

11.2.1 Goals

The Request-Reply protocol aims to standardise way of exchanging data between Tango Clients and Device Servers in a pattern where a client sends a request and the server reply with an answer. It SHALL allow for reading and writing Attributes and Pipes as well as for executing Commands.

Additionally, it aims to:

- Be independent of network transport/binary protocol,
- Define a rules for transparent reconnection,
- Allow for initiation and management of other communication protocols (like Publisher-Subscriber protocol) .

11.2.2 Use Cases

There are few main use cases for The Request-Reply protocol:

- Calling Commands on Devices,
- Reading Attributes' and Pipes' values and meta-data,
- Writing Attributes' and Pipe's data,
- Setting-up Publisher-Subscriber communication,

11.3 Specification

11.3.1 Client duty

All requests MUST be initiated by the client.

Reconnection

The client SHALL manage all connections to the Device Servers.

When sending any request to a Device Server the client:

- SHALL check if the connection to the Device Server is established and active,
- SHALL try to connect to the Device Server if the connection is not yet established or broken,
- SHALL inform the user by throwing a DevFailed exception if the connection cannot be made within a specified time defined by a user (timeout),
- SHALL use this connection to send the request if the connection is established and active,
- if the client gets no response within a certain time defined by a user (timeout) the connection SHALL be set to broken (teardown) and the client SHALL inform the user via throwing a DevFailed exception,
- SHALL respond adequately to any information on the connection state changes provided by the connection itself (e.g. os level exceptions).
- The Request-Reply protocol SHALL implement Transparent Reconnection mechanism on the client side. Transparent Reconnection is hiding temporary connection issues by re-trying of connection establishment.
- The Client MAY opt-out from using Transparent Reconnection.

11.3.2 Server duty

- The server MUST process all requests it receives.
- For every received request the server MUST send a reply to the client, in fire and forget fashion.

11.3.3 Version compatibility

The current network protocol version is 5. Devices and Clients MUST support at least network protocol version 5. Devices and Clients MAY support different network protocol versions. It is RECOMMENDED that Devices and Clients support network protocol version 3 and 4 too. When connecting to a Device, it is RECOMMENDED that a Client determines the highest possible common network protocol version. When communication with a Device, a Client SHALL use a network protocol version that is not higher than their highest common network protocol version. A Device MUST not use a network protocol version higher or lower than the network protocol version used by a Client when communicating with this Client.

network protocol Version Client uses	network protocol Version Device	Expected behaviour
C > D	D	The Client MUST support the complete specification of the Device's network protocol version D. The Client MUST not use any features from an network protocol version higher than D.
C < D	D	The Client MUST support network protocol version C. The Device MUST use network protocol version C to communicate with this Client.
C = D	D	The Client and the Device both fully support each other's network protocol version.

11.3.4 Protocol

Message

The Request-Reply protocol does not specify a binary structure of the message nor its encoding. The encoding and the binary format SHOULD be defined in an implementation RFC.

The Message implies that a Client has already made a connection to the Device. See [Connection management](#### Connection management)

However, this RFC specifies the information exchange between the Client and the Device. The communication always starts with a Request from a Client. The Device SHALL reply to any Client requests. The Request can be Synchronous or Asynchronous, see the specification below.

All Messages SHALL contain:

- request type (see table below)
- version of the protocol, since version 2.

It is RECOMMENDED for a Message to contain an id of the request.

Table of supported request type:

Request type	Version	Alternative
DEVICE NAME	3	
DEVICE DESCRIPTION	3	
DEVICE ADM_NAME	3	
DEVICE PING	3	
DEVICE BLACKBOX	3	
DEVICE INFO	3	
COMMAND EXECUTION	3	
COMMANDS LIST	3	
COMMAND INFO	3	
COMMAND HISTORY	3	
ATTRIBUTE READ	3	See ATTRIBUTES READ
ATTRIBUTE WRITE	3	See ATTRIBUTES WRITE
ATTRIBUTES READ	3	
ATTRIBUTES WRITE	3	
ATTRIBUTE WRITE READ	4	See ATTRIBUTES WRITE READ
ATTRIBUTES WRITE READ	5	
ATTRIBUTE SET CONFIG	3	
ATTRIBUTE GET CONFIG	3	
ATTRIBUTE HISTORY	3	
PIPE READ	5	
PIPE WRITE	5	
PIPE WRITE READ	5	
PIPE GET CONFIG	5	
PIPE SET CONFIG	5	

DEVICE NAME

A Device name request does not contain any additional argument.

A reply for Device name request SHALL contain:

- the Device Name

For reference see also *2/Device*.

DEVICE DESCRIPTION

A Device description request does not contain any additional argument.

A reply for Device description request SHALL contain:

- the Device description

For reference see also *2/Device*.

DEVICE ADM_NAME

A Device Admin Name request does not contain any additional argument.

A reply for Admin Device Name request SHALL contain:

- the name of Admin Device for the Device

For reference see also *2/Device*.

DEVICE PING

A Device Ping request does not contain any additional argument.

A device Ping reply consists of an acknowledgement that the command has been successfully executed.

The client implementation MAY return the elapsed time between the request and the reception of the reply.

DEVICE BLACKBOX

A Blackbox call request message SHALL contain:

- The number of requested records. If more records are requested than are available, then only the available records are returned,

A Blackbox call reply message SHALL contain:

- Array of at most nb black box entries as strings. It is RECOMMENDED that each string contains the following information:
  - time - the time when the operation was executed
  - operation - the type of operation executed on the Device e.g. ‘command_inout or read_attribute
  - source - source of operation (DEVICE or CACHE)
  - client_id - the host+pid or other identifier of the client
  - client_type - type of client (e.g. C++, Java or Python client)

DEVICE INFO

A DevInfo request SHALL contact the server and return information about the Device and Server.

A reply for DevInfo request SHALL contain the following information:

- Device class name
- Server identifier
- Server host
- Server version number
- Link to the documentation url
- Device type name

COMMAND EXECUTION

A Command call request message SHALL contain in addition:

- command name,
- argin, the specific structure is distinct to implementation,
- request type: CMD_INOUT
- source (cache or actual read), since version 2,
- client ID, since version 4.

A Command call reply message SHALL contain:

- type of the returned value,
- value returned by the command execution (so called argout).

For reference, see also *3/Command*.

COMMANDS LIST

The COMMANDS LIST request queries a Device and its reply informs a client of the Device about all of the Device Commands.

A COMMANDS LIST request does not have any additional parameters.

A COMMANDS LIST reply MUST include a sequence of a Device Command Information.

A Device Command Information MUST contain:

- The Command Name.
- The Command Display Level as defined in *3/Command*.
- An Input Data Type as defined in *9/DataTypes*.
- The Output Data Type as defined in *9/DataTypes*.
- The Input Data Type Description.
- The Output Data Type Description.

For reference, see also *3/Command* and *9/DataTypes*.

COMMAND INFO

The COMMAND INFO request queries a Device and its reply informs a client of the Device about one of the Device's Commands.

A COMMAND INFO request MUST have a one parameter:

- The Command Name.

A COMMAND INFO reply MUST include a Device Command Information.

The Device Command Information MUST contain:

- The Command Name.
- The Command Display Level as defined in *3/Command*.
- The Input Parameter Data Type as defined in *9/DataTypes*.
- The Output Parameter Data Type as defined in *9/DataTypes*.
- The Input Parameter Data Type Description.
- The Output Parameter Data Type Description.

For reference, see also *3/Command* and *9/DataTypes*.

COMMAND HISTORY

The command history allows to report of the latest Command Execution processed by the Device.

A command history request message SHALL contain:

- The Command Name.
- The number of requested entries

Until version 3 a collection of tuples with the following elements

A command history reply message SHALL contain:

- Time
- Command passed/failed state
- Command return value
- List of errors

The size of the returned collection is the minimum of the requested entries, the stored number of entries and configured depth.

Starting from verson 4:

A command history reply message SHALL contain:

- List of timestamps
- Command return value
- Tuple of x/y dimensions
- List of dimension ranges defined as index and length
- List of List of errors
- List of list of ranges into the errors array defined as index and length

- Command return type, see *3/Command*

The size of the returned first-level lists is the minimum of the requested entries, the stored number of entries and configured depth.

ATTRIBUTES READ

An Attributes read request message SHALL contain:

- list of attributes' names to read,
- source (cache or actual read), since version 2,
- client ID, since version 4.

Attributes read reply message SHALL contain:

- Sequence of attributes values. Each value SHALL provide:
  - one read value of Data Type of Attribute,
  - quality,
  - data_format,
  - data_type,
  - time,
  - name,
  - read dim x,
  - read dim y,
  - w_dim_x,
  - w_dim_y, For reference, see also *4/Attribute*.

ATTRIBUTES WRITE

An Attributes write request message SHALL contain:

- client ID, since version 4

A collection of the attributes to write where each of them contains: * value * data_type (one of Tango::AttributeDataType), * data_format (one of Tango::SCALAR/SPECTRUM/IMAGE), since version 4 * name, * quality, * time, * read dim x (until version 3), * read dim y (until version 3), * write dim x (since version 4), * write dim y (since version 4),

For reference, see also *4/Attribute*.

In case of success the reply message MUST contain nothing, on error it SHALL contain error information.

ATTRIBUTES WRITE READ

An Attributes write request message SHALL contain:

- client ID, since version 4

A collection of the attributes to write where each of them contains:

```
* value,
* data_type (one of Tango::AttributeDataType),
* data_format (one of Tango::SCALAR/SPECTRUM/IMAGE), since version 4
* name,
* quality,
* time,
* write dim x (since version 4),
* write dim y (since version 4),
```

An Attributes write read reply message SHALL provide the same information as the ATTRIBUTES READ reply.

For reference, see also *4/Attribute*.

ATTRIBUTES GET CONFIG

An Attributes get config request message SHALL contain:

- a list of attribute names
- Exception: The name "All attributes_3" can be used to get all attributes of the Device since version 3
- Exception: The name "All attributes" can be used to get all attributes of the Device until version 2

An Attribute get config reply message SHALL contain the requested list of Attribute Properties (as defined in *4/Attributes*).

Each Attribute Properties MUST include:

- the Attribute Name
- the writable Type
- the memorized mode
- the memorized initialisation mode
- the Data Format (as specified in *9/DataTypes*)
- the Data Type
- the maximum dimension x
- the maximum dimension y
- the description
- the label
- the unit
- the standard unit
- the display unit
- the format of the displayed value

- the minimum limit
- the maximum limit
- the minimum alarm level
- the maximum alarm level
- the minimum warning level
- the maximum warning level
- the delta t
- the delta val
- the Attribute Name to write when the writable mode is "READ_WITH_WRITE"
- the Root Attribute Name when the Attribute is of ForwardedAttribute type
- the Display level
- the enumeration labels for the Enum Type
- the Event relative change
- the Event absolute change
- the Archive relative change
- the Archive absolute change
- the Event period
- the Archive period
- any other configuration for future extensions
- any other system configuration for future extensions

ATTRIBUTES SET CONFIG

An Attributes set config request message SHALL contain:

- a list of Attribute Properties (as defined in *4/Attributes*).

Each Attribute Properties MUST include:

- the Attribute Name
- the writable Type
- the memorized mode
- the memorized initialisation mode
- the Data Format (as specified in *9/DataTypes*)
- the Data Type
- the maximum dimension x
- the maximum dimension y
- the description
- the label
- the unit

- the standard unit
- the display unit
- the format of the displayed value
- the minimum limit
- the maximum limit
- the minimum alarm level
- the maximum alarm level
- the minimum warning level
- the maximum warning level
- the delta t
- the delta val
- the Attribute Name to write when the writable mode is "READ_WITH_WRITE"
- the Root Attribute Name when the Attribute is of ForwardedAttribute type
- the Display level
- the enumeration labels for the Enum Type
- the Event relative change
- the Event absolute change
- the Archive relative change
- the Archive absolute change
- the Event period
- the Archive period
- any other configuration for future extensions
- any other system configuration for future extensions

No reply is required

ATTRIBUTE HISTORY

Allows to query the attribute history up to the available depth.

An Attribute History request message SHALL contain:

- Attribute name
- Number of requested entries

A command history reply message SHALL contain:

For version 3 a collection of tuples with the following elements:

- Attribute passed/failed state
- List of attribute values including:
  - value
  - quality

  - time
  - name
  - the dimension of the read part (x and y)
  - the dimension of the write part (x and y)
  - list of errors

The size of the returned collection is the minimum of the requested entries, the stored number of entries and configured depth.

For version 4 and above:

- Attribute name
- List of timestamps
- List of attribute qualities
- List of quality ranges defined as index and length
- List of x/y read dimensions
- List of read dimension ranges defined as index and length
- List of x/y write dimensions
- List of write dimension ranges defined as index and length
- List of list of errors
- List of list of ranges into the errors array defined as index and length
- data_type (one of Tango::AttributeDataType), since version 5
- data_format (one of Tango::SCALAR/SPECTRUM/IMAGE), since version 5

The size of the returned first-level lists is the minimum of the requested entries, the stored number of entries and configured depth.

PIPE GET CONFIG

The PIPE GET CONFIG request queries a Device and its reply informs a client of the Device about the Configurations of one or more Pipes in the Device.

A PIPE GET CONFIG request MUST be:

- A sequence of Pipe Names

A PIPE GET CONFIG reply MUST consist of a sequence of Pipe Information tuples.

A Pipe Information tuple MUST consist of:

- The Pipe Name.
- The Pipe Description.
- The Pipe Label.
- The Pipe's Display Level as defined in *7/Pipe*.
- The Output Data Type as defined in *7/Pipe*.
- The Pipe Write Type as defined in *7/Pipe*.

For reference, see also *7/Pipe*.

PIPE SET CONFIG

The PIPE SET CONFIG request allows a Device's client to instruct a Device to set the Configurations for one or more of its Pipes.

A PIPE SET CONFIG request MUST be:

- A sequence of Pipe Configuration tuples.
- A client ID.

A Pipe Configuration tuple MUST consist of:

- The Pipe Name.
- The Pipe Description.
- The Pipe Label.
- The Pipe's Display Level as defined in *7/Pipe*.
- The Pipe Write Type as defined in *7/Pipe*.

For reference, see also *7/Pipe*.

PIPE READ

A Pipe read request message SHALL contain:

- a name of pipe to read,
- a client ID.

A Pipe read reply message SHALL contain:

- a name,
- a time,
- a value with the Data Type of DevPipeBlob.

For reference, see also *7/Pipe*.

PIPE WRITE

A Pipe write request message SHALL contain:

- a name,
- a time,
- a value with Data Type of DevPipeBlob,
- a client ID.

In case of success the reply message MUST contain nothing, on error it SHALL contain error information.

For reference, see also *7/Pipe*.

PIPE WRITE READ

A Pipe write read request message SHALL contain:

- a name,
- a time,
- a value with Data Type of DevPipeBlob,
- a client ID.

A Pipe write read reply message SHALL contain:

- a name,
- a time,
- a value with Data Type of DevPipeBlob.

The data for the reply SHALL only be read after the write value has been written.

For reference, see also *7/Pipe*.

Exception

Certain conditions may prevent a Request to be handled properly.

- If a Request cannot be handled properly, the Client SHALL be notified by receiving of a DevFailed or MultiDevFailed exception (see *9/DataTypes*).
- The DevFailed exception MAY be generated by either a Client (API) or a Device Server.
- Exceptions MAY be raised either by API code or by a user provided code.
- The Request Reply Protocol SHALL provide mechanism of sending a DevFailed or MultiDevFaildeded exception from a Device Server to a Client.
- If a Device Server throws a (Multi)DevFailed exception upon request processing, the exception SHALL be sent to the Client which sent the request.
- If a Client receives a (Multi)DevFailed exception it SHALL be available (rethrown) to user (code).

Timeout

- On the client side, the Request-Reply protocol SHALL implement timeout mechanism for every type of request. Timeout mechanism is returning DevFailed exception if a request cannot be served in specified time called Timeout.
- For each request the client MAY setup Timeout value or opt-out from using timeout mechanism which means that:
  - if the request is Synchronous, the client SHALL be blocked as long as it takes the request to be served. A timeout realted exception SHALL not be returned.
  - if the request is Asynchronous, the Client SHALL NOT be blocked. The Client SHALL be able to access the request result after it is served by a Device Server without any time limits. A timeout realted exception SHALL NOT be returned.
- The default Timeout is 3 seconds.

Synchronous request

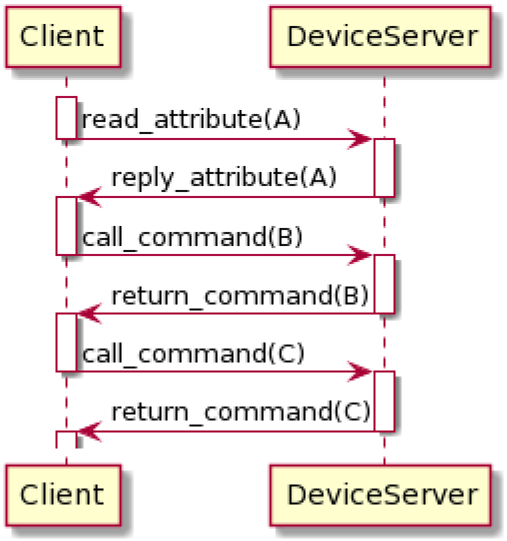
The client MAY send Synchronous Requests.

When the client sends a Synchronous Request it SHOULD wait for a request to be processed.

If one client thread sends multiple Synchronous Requests sequentially, these SHALL be processed in the same order as these have been sent.

The client SHALL handle the Synchronous Request in the way that it blocks the calling client thread until the request is fully processed (a Device Server reply to the request and the result is available to the client) or timeout or other error appear.

Below is a diagram showing an example seqence:



The client MAY allow multiple Synchronous Requests to be sent in parallel if these are sent by multiple client threads.

The Device Server MAY process multiple synchronous requests in parallel according to its [Serialization](#### Serialization).

Asynchronous request

The client MAY send Asynchronous Requests.

When the client sends an Asynchronous Request it MAY not wait for the request to be processed before sending another request.

The client thread which sends an Asynchronous Request MUST NOT be blocked for a time of the request being processed (except for a time needed for the request registration).

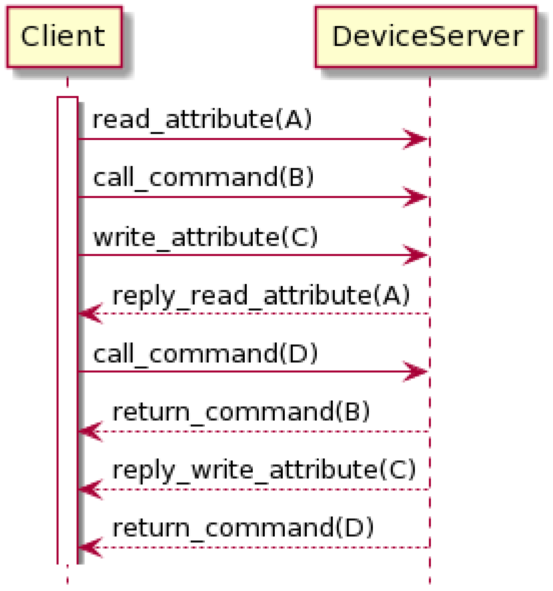
A result of the Asynchronous Request SHALL be available for the client.

The client SHALL be informed when the result of the Asynchronous Request is available.

The client MAY decide whether to process the result of Asynchronous Request:

- as soon as it is available (for example in the call-back pattern),
- or when it decides to process,
- or skip processing of the result.

Below is a diagram showing an example seqence of Asynchronous Requests:



A Device Server SHALL process any incoming requests in the order of their arrival and according to its [Serialization](#### Serialization).

Cache

The Cache is a circular buffer of recent responses for requests generated internally by the Device Server's Polling mechanism.

The Polling is periodically invoking a list of requests on Devices instantiated by a Device Server. The period between 2 same requests invocations is called a Polling Period.

- The Cache and Polling mechanism SHALL be implemented by a Device Server.
- The Polling SHALL support the following requests:
  - read an Attribute (with the exception of a Forwarded Attribute which MUST NOT be polled)
  - execution of a Command with DevVoid `argin`.
- An attempt to poll a forwarded Attribute MUST raise an exception.
- The Polling Period MAY be configured separately for each Attribute or Command of each Device.
- A Device Class MAY implement default settings of the Polling for Attributes and Commands.
- A Client MAY change the Polling Period of any Attribute or Command of any Device.
- A Client MAY switch off/on a Polling for any Attribute or Command of any Device.
- If the Polling Period is set to 0, the Device Server may fill the Cache with arbitrary values and arbitrary associated timestamps at any moment. This is called Externally Triggered Polling.
- The Polling SHALL be configurable by a Client via the following Admin Device commands:
  - `AddObjPolling`: 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 devices are polled
  - `DevPollStatus`: Allow a client to know the polling status for a device precisely.
- The runtime behaviour of the Polling MAY be configured via the following properties of the Device Server Administration Device:
  - `polling_threads_pool_size` to specify number of polling threads,
  - `polling_threads_pool_conf` to assign instances of Devices (within the Device Server) to specific polling threads,
  - `polling_before_9` to set a polling algorithm version to use,
  - If `polling_before_9` property is not set or is set to `False` the Device Server's polling thread SHALL poll all attributes of a Device with the same polling period using a single device call (read_attributes) and allow the polling thread to be late but only if number of late objects decreases. This is a default algorithm.
  - If `polling_before_9` is present and set to `True`, the Device Server SHALL use the old (pre 9) version of polling mechanism (algorithm). In this case the Device Server's polling thread SHALL poll one Attribute at a time even if the polling period is the same (use of read_attribute) and do not allow the polling thread to be late.
- The Polling configuration defined by a Client SHALL be stored in the Database and applied upon a Device initialization.
- For each request from the Polling list, the Device Server SHALL have at least the most recent response generated by the request in a Cache.
- The depth of the Cache (the circular buffer size) SHALL be configurable using `poll_ring_depth` device property from the database.
- For any Request, a Client MAY decide whether it wants a response from:
  - the Cache,
  - or execute the underlying request,
  - or the Cache with fallback to underlying request if the Cache is empty or old. The Cache for certain Attribute or Command is considered old if the most recent value is older than Polling Period multiplied by `poll_old_factor` property from the Database.
- The source of Attribute Read request on a Forwarded Attribute MUST be applied to the Root Attribute Read request.

  The Device Server SHALL respond adequately.
- A Client MAY read Attribute's or Command's history from Cache (read last n values).
- The Server SHALL provide interface to read Attribute's and Command's history.
- If the client asks for the Cache without fallback and the Cache is old, the Device Server SHALL respond with DevFailedException with <`reason`> field set to `API_NotUpdatedAnyMore`.
- If the client asks for the Cache without fallback and the Cache is empty, the Device Server SHALL respond with DevFailedExcpetion with <`reason`> field set to `API_NoDataYet`.
- The Request-Reply protocol SHALL allow a Client to read history from the Cache.

- If the Client asks for a history of not polled Attribute or Command, a Device Server SHALL return DevFailedException with `<reason>` field set to `API_AttrNotPolled` or `API_CmdNotPolled` respectively.

Serialization

The Serialization Mode defines how the Device Server handles multiple requests.

The Serialization SHALL apply to all requests independently of a request origin (any network host, localhost, the same Device Server process).

A Device Server MAY handle requests with the following Serialization Modes:

- Serialization by Device
- Serialization by Class
- Serialization by Process
- No Serialization

When a Device Server is set to Serialization by Device:

- it SHALL process requests to each Device in the order of requests arrival,
- for any Device in the Device Server, it MUST NOT start the processing of a new request until the processing of the previous request is not finished,
- it MAY process multiple requests in parallel, providing that there is maximum one processed request per a Device.

When a Device Server is set to Serialization by Class:

- it SHALL process requests to each set of Devices belonging to the same Class in the order of requests arrival,
- for each Device Class, it MUST NOT start the processing of a new request until the processing of the previous request is not finished,
- it MAY process multiple requests in parallel, providing that there is maximum one processed request per Device Class.

If Device Server is set to Serialization by Process:

- it SHALL process requests in the order of requests arrival,
- it MUST NOT start the processing of a new request until the processing of the previous request is not finished,

The Device Server SHOULD process requests as soon as possible. If an incoming request cannot be processed due to Device Server Serialization Mode it SHOULD be queued for later processing and served as soon as possible.

If Device Server is set to No Serialization it MAY process any requests in parallel. It is NOT RECOMMENDED to use No Serialization mode.

The default serialization Mode is “Serialization by Device”.

Blackbox

A blackbox system should record every REQUEST on the Device especially:

Attribute case

- Read Attribute request, Should register the attribute name, the id of the client (often the pid), the version of TANGO and the Source (DevSource)
- Write or WriteRead Attribute request, Should register the attribute name, attribute value (only for writing?) (AttributeValueList_4), the id of the client (often the pid) and the version of TANGO
- reading an Attribute configuration # (Op_Get_Attr_Config)
- changing an Attribute Configuration # (Op_Set_Attr_Config);
- Read the Attribute History

Pipe case

- Read Pipe request, Should register the pipe name, the id of the client (often the pid), the version of TANGO and the Source (DevSource)
- Write Pipe request, Should register the pipe name, attribute value, the id of the client (often the pid) and 0?, the version of TANGO_PIPE?
- WriteRead Pipe request, Should register the pipe name, attribute value, the id of the client (often the pid) and 1?, the version of TANGO_PIPE?
- reading an Pipe configuration # (Op_Get_Pipe_Config_5);
- changing an Pipe Configuration # (Op_Set_Pipe_Config_5);
- Read the Pipe History

Command case

- listing the list of Command Name [//]: # (Op_Command_list);
- execute a Command Name Should register the command name, the id of the client (often the pid), the version of TANGO and the Source (DevSource)
- any operation like reading the request of Command History [//]: # Op_Command_inout_history_4,

Operations case:

- reading the BlackBox itself # (Op_BlackBox);
- reading the device (Attribute?) name # (Attr_Name)
- request the adm_name attribute # (Attr_AdmName);
- request thedescription attribute # (Attr_Description);
- read the information of a device command # (Op_Command);
- read the information of a device # (Op_Info);
- executing Ping # (Op_Ping);

NOTE All GROUPED REQUESTS should be logged once with the same information as the unique request

Device Locking

A Client has the possibility to avoid any modification of a Device from other Clients. This system is so-called Device Lock. The Device Server has here the role and the responsibility to apply this mechanism.

The following specifies the Device Lock system:

- Only one Device Lock per Device SHALL be activated.
- A Device Lock SHALL be only owned by one Client.

When a Client activates a Device Lock, the Device Server MUST reject any of the following requests from any other Client:

- Command call except for the State Command, the Status Command and all commands listed in the Allowed Commands list (described further down).
- Write Attribute Request (including the atomic Write Read request)
- Write Pipe Request (including the atomic Write Read request)
- Setting Attribute Configuration
- Setting Pipe Configuration
- Setting Polling Configuration
- Setting Logging Configuration In general it is RECOMMENDED to block any request resulting in a mutation of a Device.

An Allowed Command is not affected by the Device Lock, even if this command execution implies a modification of the Device state. A list of Allowed Commands are defined per Device Class with the `AllowedAccessCmd` Class Property of type DevVarStringArray.

+Other Clients have the possibility to check a Device Lock is currently activated via the Device Server Command '"DevLockStatus"' specified by: +* Command Name: DevLockStatus +* Argument Input of DevString type corresponding to the Device Name +* Argument Output of DevVarLongStringArray type giving the Device Lock status with these mandatory values: +** String values in order: the Client hostname, +** Long values in order: the activation state, the Client IP
+

+The request of a Device Lock is done through the DeviceServer command '"LockDevice"' with the Input Argument of DevVarLongStringArray with the following value: +* as string value the Device Name to lock, +* as long value the validity time in second. +The protocol of activation SHALL be the responsibility of the Device Server, following this sequence: ```

- Check no other Client owns an active and valid DeviceLock on the requested Device
- Activate the Device Lock
- Increase the counter of Device Lock (See Below).
- Device Lock request SHALL be sent and validated by all Devices connected to the Forwarded Attributes of the targetted Device. In case of failure of the activation of a Device Lock the Device Server SHALL throw a related DevFailed exception with <reason> field set to "API_DeviceLocked".

A Client owning a Device Lock MAY:

- revoke the Device Lock by the DeviceServer's UnLockDevice Command,
- renew the Device Lock activation by the DeviceServer's ReLockDevices Command A Locking Tango Exception SHALL be thrown if the Device Lock is not active anymore.

A Device Lock is defined by:

- a Client Identification of the owner
- a timestamp corresponding of the locking activation
- a counter representing the number of valid activation from the same Client
- the activation time in second defining the locking period from the last activation timestamp

A Device Lock SHALL be deactivated in any of the following cases:

- The Client owning the Device Lock, calls the Device Server's UnLockDevice Command for the locked Device,
- Any Client calls the DeviceServer's UnLockDevice Command with a `force` flag set to true for the locked Device,
- The time of activation expired,
- The Connection to the Client is lost,
- The Device Server is reinitialised.

If the Device Lock has been disabled with the `force` flag set to true, the Server SHALL sent to the Client who originally owned the Device Lock a DevFailed exception with reason set to "API_DeviceUnlocked" upon next request from the Client.

A Client Identification SHOULD carry these information:

- Hostname where the Client process runs
- PID of the Client process
- IP of the Client connection It is recommended to include any other informations which may help the administrators of the system.

Connection management

- The Request-Reply protocol SHALL manage client connection.

Connection establishement

A Connection MUST consist of exactly two parties:

- Client: The Connection Initiator
- Server: The Connection Target

Interoperable Object Reference

An implementation SHALL specify a Connection Target by an Interoperable Tango Reference (ITR). An implementation MUST observe the following characteristics of an ITR:

- ITRs SHALL serve as a mapping between Tango Controls object addresses and an implementation's object addresses.
- It SHALL be possible to address every publicly exported Tango Controls object in a Server by an ITR.
- It MAY be possible to address not publicly exported Tango Controls objects in a Server by an ITR.
- Every ITR MUST be globally unique.
- Every generated ITR MUST be reproducible.

CHAPTER

TWELVE

12/PUBLISHER-SUBSCRIBER PROTOCOL

12.1 Preamble

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

12.2 Publisher-Subscriber protocol Specification

It is a relationship between Tango client (Client) and Tango server (Server). Based on criteria negotiated as part of a subscription, Tango client will receive updates. Beyond a set of basic requirements, various refinements are addressed. These refinements include: periodicity of updates, filtering out updates based on Tango server configuration and fault tolerance.

12.3 Basic Concepts

This document refers to other Tango Controls RFCs, corresponding link is always given in this case. Here is given a brief description of the main concepts. This document targets arbitrary software developer(s) who are to implement *Tango Client Library* and *Tango Server Library*.

Tango Server (Server) - a process that runs on a machine.

Tango Device (Device) - for more details see *2/Device*.

Tango Admin Device (Admin) - for more details see *8/Server*.

Tango Client (Client) - a process that instantiates communication with *Tango Server* using Request-Reply protocol *10/RequestReply*.

Tango Client Library (client kernel library) - library that encapsulates communication with *Tango Server* and provides *Tango Client API* to a user.

Tango Client API (client API) - an API that user uses to develop applications based on communication with *Tango Servers*.

Tango Server Library (server kernel library) - library that encapsulates *Tango Server* features and provides *Tango Server API* to a user.

Tango Server API (server API) - an API that user uses to develop *Tango Device* functionality.

Upstream Tango Server/Device (upstream server/device) - a *Tango Device* to which updates *client* subscribes for.

Polling thread (polling thread) - a thread within *Tango Server* that polls periodically defined *Tango Device*'s attributes/sometimes commands.

Tango Publisher-Subscriber protocol (pub-sub) - is a sub-set of *Tango Request-Reply protocol 10/RequestReply* that defines a relationship between *client* and *server* established via a negotiation phase and lasts till explicitly broken. During this relationship client receives updates published by *upstream server*. Updates are sourced from server's *polling thread* or directly through *server API*. Updates are filtered according to *device*'s configuration.

Tango Event (event) - an update sent by *server*.

Tango Event Callback (callback) - a piece of code that *client library* executes when an update is received.

Channel - virtual client-server point-to-point connection for transmitting events

12.4 Definitions

This section offers a number of ABNF formal definitions for entities repeatedly used below in this document.

> **NOTE**: in the below ABNF some basic definitions are used as rules, spaces are replaced with '_'
>
> **NOTE**: in the below ABNF data types are sometimes embedded into the rule e.g. bool:isExcept is equivalent to isExcept = true / false

```
endpoint = URL ; usually with port

SUBSCRIBER_INFO = upstream_device attribute_name action EVENT_TYPE client_idl_ver      ␣
↪                                                                                       ␣
↪

SUBSCRIPTION_INFO = server_library_release_ver upstream_device_idl_ver zmq_sub_event_hwm␣
↪rate ivl zmq_release_ver heartbeat_endpoint event_endpoint 1*alternate_heartbeat_
↪endpoint 1*alternate_event_endpoint HEARTBEAT_CHANNEL EVENT_CHANNEL

EVENT_INFO = EVENT_DATA/1*EVENT_ERROR bool:isExcept  ; see below

EVENT_DATA = ATT_CONF_DATA/PIPE_DATA/DATA_READY_DATA/INTERFACE_CHANGE_DATA/ATTRIBUTE_
↪DATA

ATT_CONF_DATA = ATTRIBUTE_PROPERTIES ; Attribute properties from [4/Attribute](../4/
↪Attribute.md)

PIPE_DATA =  int:size name timeVal PIPE_BLOB ; pipeBlob must be defined in [7/Pipe](../7/
↪Pipe.md)

DATA_READY_DATA = attribute_name data_type int:counter

INTERFACE_CHANGE_DATA = *ATTRIBUTE_PROPERTIES *COMMAND_META_INFO bool:deviceStarted
```

(continues on next page)

(continued from previous page)

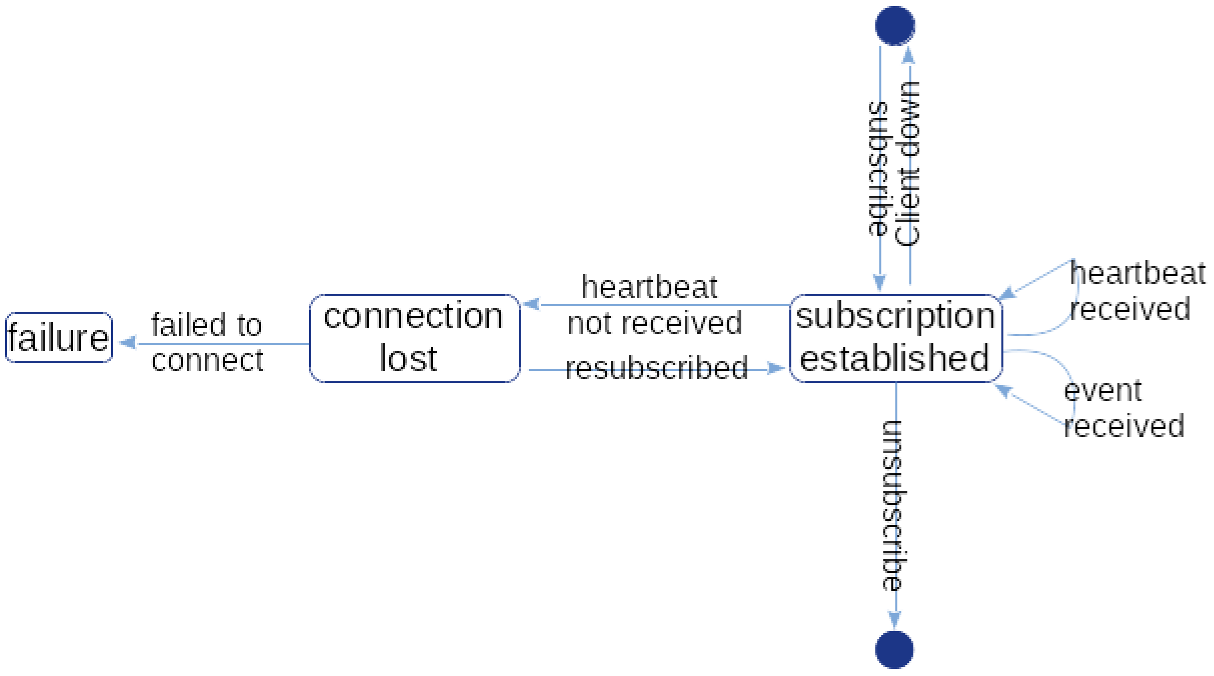
```
ATTRIBUTE_DATA = ATTRIBUTE_DEFINITION ; Attribute definition from [4/Attribute](../4/
↪Attribute.md)

EVENT_ERROR = reason severity desc origin

EVENT_TYPE = "INTERFACE_CHANGE"/"PIPE_EVENT"/"ATTR_CONF_EVENT"/"CHANGE_EVENT"/"PERIODIC_
↪EVENT"/"ARCHIVE_EVENT"/"USER_EVENT"

EVENT_CHANNEL = channel

HEARTBEAT_CHANNEL = channel

channel = string ; implementation specific
```

NOTE: In Tango V9 EVENT_DATA MAY include source idl version, event type

12.5 Runtime requirements

Client and server are up and running. Server is reachable from client i.e. may communicate using Request-Reply protocol *10/RequestReply*.

12.6 Publisher-Subscriber protocol

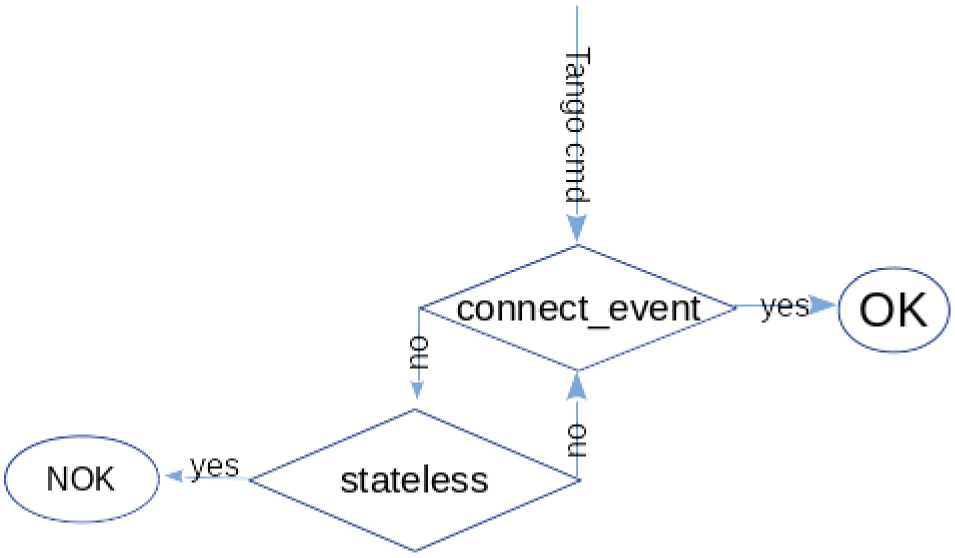


The main goal of this protocol implementation is to allow a Client receive events from a Server:

- Server MUST publish events of EVENT_TYPE
- Server library MUST provide an API to publish events of EVENT_TYPE

- Client SHOULD process events
- Client library SHOULD provide user an API to register user defined event callbacks

12.6.1 Negotiation



Negotiation phase implies that client and server exchange required data using Tango commands *3/Command*:

- Client MUST notify server providing SUBSCRIBER_INFO to establish Publisher-Subscriber relationship

  in Tango 9 the above is done by executing EventSubscriptionChange admin command

  Tango 9 forces client to explicitly execute ZmqEventSubscriptionChange to use ZMQ based implementation.

- Server MUST respond to the client providing SUBSCRIPTION_INFO
- Client MUST connect to upstream server using provided SUBSCRIPTION_INFO
- Client SHOULD postpone event subscription process in case of connection failure and try again later

Tango Event System uses concept of channels for delivering events. Channel is established between client and admin. There are two channels per subscription: heartbeat and event.

- Client MUST establish channel or multicast connection to heartbeat/event endpoints
- Server MAY send and client MAY iterate over alternative heartbeat/event endpoints to establish connection
- Client MAY NOT use provided heartbeat/event channels to identify event source

12.6.2 Subscription

- Server MUST send events via corresponding EVENT_CHANNEL
- Server MUST send HEARTBEAT via corresponding HEARTBEAT_CHANNEL event every 9 seconds
- Client SHOULD indicate its interest in subscription not later than every 600s

12.6.3 Cancel event subscription

- Client MAY cancel its subscription
- Server SHOULD NOT send events to client whose subscription is canceled

12.6.4 Fault tolerance

Tango Pub-Sub protocol implies a number of fault tolerance techniques:

- server heartbeat
- transparent client reconnection
- client subscription confirmation
- events client/server buffer
- Client SHOULD try to re-subscribe once HEARTBEAT event is missing
- Client MAY try to re-subscribe indefinitely
- Server MAY cancel client's subscription if it does not confirm its interest
- Client/Server SHOULD buffer the (in/out)coming events to prevent overload and overflow
- Clients SHOULD be notified in case of events overflow
- Client SHOULD be notified in case of missing events

  > e.g.: In the current C++ implementation at least, the client callback is executed with an API_MissedEvents error event when the client library detected that some events got lost. There is a mechanism with some counters associated with each event to help detecting this kind of problems

CHAPTER

THIRTEEN

14/THE TANGO LOGGING SERVICE

13.1 Introduction

This document describes the Tango Logging Service (TLS). The TLS allows Tango Devices to print messages to be:

- displayed on a console (the classical way)
- sent to a file
- sent to specific Tango device called log consumer
- or more than one of the above at the same time

See also: *2/Device*, *4/Attribute*

13.2 Preamble

13.3 Goals

This specification will describe the Tango Logging Service (TLS).

It will describe and specify such things as:

- how logging is statically configured for a Device, that is, configured from Device Properties for start up.
- some logging terms: level, target, rft and path.
- what are logging targets: console, file, device
- what is the logging environment variable TANGO_LOG_PATH
- logging levels: OFF, FATAL, ERROR, WARNING, INFO, DEBUG.

- logging commands available to all Devices
- the standard Tango Log Consumer Device and its interface (static mode, dynamic mode).

13.4 Use Cases

The Tango Logging Service (TLS) provides the programmer with a convenient way to print information which helps to:

- debug the software
- report on errors
- give regular information to user
- provide timely information of the activities of the Device
- report the version of the software
- provide clues to the internal changes happening in the Device

At run-time, the TLS supports such activities as:

- changing logging verbosity and destination of any Tango Device
- capture logs to files
- display log entries on the command line console
- interactively view log entries as they are produced by a remote Tango Device
- review logs already written to files
- make graphical dashboards with such tools as Kibana and Grafana

13.5 Specification

13.5.1 Logging Properties

TLS SHALL use four Device Properties to control Device Server logging at startup. The Properties may be set during runtime in the normal way.

logging_level property

SHALL control the initial logging level of a device. It SHALL have the set of possible values: OFF, FATAL, ERROR, WARN, INFO or DEBUG. This property SHALL be overwritten by the verbose command line option (-v).

logging_target property

SHALL give the name of one or more Logging Target (see below), separated by commas.

logging_rft property

SHALL cause the maximum file size written to be the given value. If a file grows past this size, SHALL be closed, renamed and a new file opened.

logging_path property

SHALL override the TANGO_LOG_PATH environment variable and SHALL only apply to classes derived from the DServer class. The value SHALL specify where log files are written for Logging Target “file” (with no filename).

13.5.2 Logging Targets

TLS SHALL allow Logging Targets, that is, destinations, for messages. A Device Server SHALL emit messages to all targets (comma-separated) to its logging_target Property.

console

output SHALL be delivered to stdout

file

output SHALL be written to a standard location in a directory specified by TANGO_LOG_PATH if nonblank, otherwise in a directory created under /tmp/tango-*.

file::filename

output SHALL be written to the filename given. File format SHALL be log4j compatible.

device::devicename

output SHALL be sent to the Tango Device given by the device name given. The The Device running as devicename SHALL implement the Log Consumer interface (see below).

13.5.3 Logging Levels

TLS SHALL support Logging Levels. The Logging Level is meant to act as a filter for the amount of inform. These are the logging levels, with typical meanings for when they are to be used.

- 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 SHALL be ordered the following way:
`DEBUG < INFO < WARN < ERROR < FATAL < OFF`
For a given device, a level SHALL be 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.

All the Device's Logging Targets SHALL share the same device logging level.

13.5.4 Log Consumer

When a Device Target is specified as "device", this SHALL designate a Device Server which is implementing the Log Consumer interface. To be a Log Consumer, a Tango Device SHALL implement just one Tango Command:
`void log (Tango::DevVarStringArray details)`.

The array of strings composing the argument SHALL be interpreted with the following meanings:

- 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)

13.5.5 GUI for viewing logs

TLS MAY include a GUI application called LogViewer for displaying logs pushed by the devices to the Tango logging system. LogViewer SHALL have the ability to fetch logs from different Tango devices and filter them by several properties. The LogViewer SHALL have the following abilities:

- show logging messages from multiple devices simultaneously
- filter logging messages by Logging Level
- be able to read log files from disk

LogViewer SHALL have two modes: static and dynamic.

- static: LogViewer SHALL be started with the name of the Log Consumer Device name and it SHALL receive all messages from Devices who have this name as their target.
- dynamic: LogViewer SHALL let the user use the GUI to specify which devices LogViewer will monitor.

CHAPTER

FOURTEEN

15/THE DYNAMIC ATTRIBUTE AND COMMAND

14.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

14.2 Goals

This specification describes the Tango kernel Dynamic Attribute and Command API.

14.3 Use Cases

The primary use case is to allow Tango Device Server programmers to create Attributes and/or Commands at runtime.

Reasons for that:

- create Attributes/Commands based on underlying library;
- create generic Tango device classes which interfaces are entirely/partly unknown at the moment of creation.

Consider a Tango Device Class that represents a series of detectors. Each detector differs a bit in terms of supported features, while the most part of the code could be the same for each detector. These small differentiations may be implemented as Dynamic Attributes or Commands.

Consider a Tango Device Class that represents a residual gaz analyzer. The device will provide measurement results for many different masses. The results can be presented as a Tango spectrum attribute showing the measurements for all the masses but it can be also very interesting to create an attribute for each different mass, then the user could monitor this specific mass or define some alarm thresholds or configure archiving events for this specific mass. Dynamic Attributes may be used to create easily these numerous attributes whose behaviour is almost identical.

Consider a Tango Device my/simu/device simulating the interface of another Tango Device my/original/device, for tests purposes. my/simu/device could query the interface of my/original/device at runtime and create dynamically the same attributes and commands as my/original/device. The simulated device could then be used to test high level client applications which would then talk to the simulated device instead of the original device.

14.4 Specification

- Dynamic Commands and Dynamic Attributes MAY be defined at the Device Class level.
- Dynamic Commands and Dynamic Attributes MAY be defined at the Device level.
- A Dynamic Attribute at the Device level MUST override a Dynamic Attribute at the Device Class level.
- A Dynamic Attribute at the Device level MUST NOT override a Static Attribute.

Note: current implementations of cppTango and JTango do not fully follow this specification.

14.4.1 API that allows to create Attributes and Commands in runtime

Tango kernel library MUST provide API to server client to add new Attributes, Commands.

This API MAY accept configuration for each Attribute or/and Command.

Accepted configuration MUST be the same as described in *3/Command* and *4/Attribute*.

As the result of the call, API creates corresponding Attribute/Command fully compliant to *3/Command* and *4/Attribute*.

An implementation MUST send an `INTERFACE_CHANGE` event, see *12/PubSub*, in response to added or removed Attributes and Commands.

CHAPTER

FIFTEEN

16/TANGO RESOURCE LOCATOR (TRL)

This document describes the Tango Resource Locator.

See also: *1/Tango*, *2/Device*, *4/Attribute*, *5/Property*, *6/Database*.

15.1 Preamble

This Specification is a free and open standard and is governed by the Digital Standards Organization's Consensus-Oriented Specification System.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

15.2 Tango Resource Locator (TRL) Specification

The distributed objects that make up a Tango control system need to be accessible on a network, as follows:

- Each distributed object MUST only be defined in a single Tango Database.
- Each distributed object MUST have a unique identifier within the network - this is termed the Tango Resource Locator.
- Each Tango Resource Locator MUST be sufficient for a client on the network to initiate communication with the specified object.
- All Tango Resource Locators MUST be case insensitive.
- Clients MAY use shortened forms of the Tango Resource Locator, but additional context MUST be used to disambiguate the name.
- Such additional context MAY include environment variables and data from configuration files.
- There MAY be additional names that refer to the same distributed object, such a name is termed an Alias.
- The Tango Resource Locator MUST use the following scheme:

```
[protocol://][host:port/]device-name[/attribute][->property][#dbase=xx]
```

Following the convention:

- `protocol`:

  ```
  protocol = %s"tango"
  ```

  - If omitted, the implementation MUST use `tango://`.
- `host`: MUST be either
  - an IP address (IPv4 or IPv6) or
  - a resolvable hostname

  as defined in RFC 1123 and RFC 952.
- `port`: MUST be an integer in the range of 1 to 65535 as defined in RFC 793. It is RECOMMENDED to not use privileged (1 to 1024) ports.
- If `host` and `port` are omitted:
  - The implementation MUST set it based on the client's local context (environment variables, config files, etc.):
    * The local context MAY include the environment variable `TANGO_HOST`, set to a string providing `host:port`.
    * If the local context does not provide sufficient detail then access MUST fail.
- `device-name`: as specified in *2/Device*.
- `attribute` and `property`:
  - `attribute`: as specified in *4/Attribute*.
  - `property`: as specified in *5/Property*.
  - If `attribute` is omitted, but `property` is included it MUST name a Device Property.
  - If both `attribute` and `property` are included they MUST name an Attribute Property.
- `dbase` and `xx`:

  ```
  dbase = "dbase"
  xx = "yes" / "no"
  ```

  - If `#dbase=xx` is omitted, the implementationr MUST use `#dbase=yes`.
  - If `#dbase=no`, then the implementation MUST not attempt to access a Tango Database when establishing the connection with a Device/Attribute/Property.

Note: The Tango Resource Locator looks similar to a uniform resource locator, as per RFC 3986, but the current implementation has minor deviations from RFC 3986. This specification will not refer to the Tango Resource Locator as a uniform resource locator (URL). Existing Tango documentation sometimes refers to the Tango Resource Locator as the Fully Qualified Domain Name (FQDN). The FQDN is also ambiguous, so it is not used in this specification.

15.2.1 Case sensitivity

Tango Controls does not distinguish between upper and lower case when interpreting the following names:

Device Server, Device, Device Alias, Command, Attribute, Property

Note: When storing any of the above values in the Tango database or declaring them in the code for readability then the case is preserved however all comparisons ignore case and all permutations of the case e.g. 'VOLTAGE'== 'voltage'.

All Property values, Class names and DServer (device server) names are exempt from this and are interpreted as case sensitive.

15.2.2 Alias

The aliasing mechanism is provided for convenience. It could provide a shorter or more memorable name for an object in the Tango control system.

- An Alias MAY refer to a Device.
- An Alias MAY refer to a Device Attribute.
- An Alias MUST be unique within a Tango control system (single Tango Database).
- An Alias MUST be stored in the Tango Database.
- The Device Alias specification is in *2/Device*.
- The Attribute Alias specification is in *4/Attribute*.
- Additional context MUST be used to find the Tango Database, in the same way that a shortened Tango Resource Locator is disambiguated.

15.2.3 Examples

```
tango://db.example.com:10000/lab/powersupply/01
```

This refers to a Device named `lab/powersupply/01`. The Device Server which is running the Device is running on an unknown host. Therefore a lookup will be performed by the Tango Database which runs on host `db.example.com` at port 10000 to make a client to Device connection possible.

```
LAB/POWERSUPPLY/01
```

This TRL refers to a Device named `lab/powersupply/01`. It must be defined in the Tango Database. The host and port for the Tango Database needs to be discovered in the local context. An example for a matching full TRL with a Tango Database at `db.example.com:10000` is `tango://db.example.com:10000/lab/powersupply/01#dbase=yes`.

Note that the case is ignored.

```
tango://db.example.com:10000/lab/powersupply/01/voltage
```

This TRL refers to the `voltage` Attribute on the `lab/powersupply/01` Device. The Device is defined in the Tango Database, which runs on `db.example.com` at port 10000.

```
tango://db.example.com:10000/lab/powersupply/01/voltage->unit
```

This refers to the `unit` Property of the `voltage` Attribute on the same Device.

```
tango://db.example.com:10000/lab/powersupply/01->address
```

This refers to the `address` Device Property on the same Device.

```
tango://lab.example.com:14555/lab/powersupply/01#dbase=no
```

This refers to a Device named `lab/powersupply/01`. Regardless if it is defined in a Tango Database or not, the Device will be directly accessed on the host `lab.example.com` at port 14555.

```
tango://lab.example.com:14555/lab/powersupply/01/voltage#dbase=no
```

This refers to the `voltage` Attribute on the same Device.