panic Documentation

Author

Apr 14, 2020

Contents

PANIC is a set of tools (api, Tango device server, user interface) that provides:

- Periodic evaluation of a set of conditions.
- Notification (email, sms, pop-up, speakers)
- Keep a log of what happened. (files, Tango Snapshots)
- Taking automated actions (Tango commands / attributes)
- Tools for configuration/visualization

Contents:

CHAPTER 1

PANIC Description

CHAPTER 2

Changelog

CHAPTER 3

Installing PANIC on a New System

3.1 Dependencies

PANIC is available from Github, PyPI and as Debian or SuSE packages.

If you install from SuSE or Debian packages dependencies will be automatically installed.

If not, then you'll need Tango, PyTango and Fandango for the server side (including its dependencies, ZMQ, numpy, ...).

For the client side you'll also need Taurus library and PyQt4.

You should be able to get all these packages also from www.tango-controls.org

3.2 Run the GUI and create a PyAlarm

Running "setup.py install" should install the panic-gui script in your system.

But if you don't want to install the application you can just run python panic/gui/gui.py to launch the client.

In your first run it will apply completely empty. Just create your first PyAlarm instance going to the "Config" icon in the toolbar and pushing "Create New" button.

Now you can create your first PyAlarm pushing "New" in the main widget. You'll be prompted to fill the gaps, for a first installation I recommend this alarm:

> TAG: TEST_LOG Description: just testing Severity: WARNING Receivers: your_mail@your_domain.com Formula: True

This simple alarm will allow you to check if email sending works properly.

3.3 Run the PyAlarm Server

Use Astor or the shell to start your newly created PyAlarm:

```
python ds/PyAlarm.py TEST -v4
```

After ~45 seconds (if you didn't modified the default configuration) you'll receive your first email from PANIC.

Now head to the configuration docs to know all the options you have for tuning the behaviour.

CHAPTER 4

PyAlarm Device Server User Guide

Contents

4.1 Description

This device server is used as a alarm logger, it connects to the list of attributes provided and verifies its values.

Its focused on notifying Alarms by log files, Mail, SMS and (some day in the future) electronic logbook.

You can acknowledge these alarms by a proper command.

4.2 Internal Structure

The device server behaviour relies on three python objects: AlarmAPI, updateAlarms thread and TangoEval.

Each alarm is independent in terms of formula and receivers; but all alarms within the same PyAlarm device will share a common evaluation environment determined by PyAlarm properties.

4.2.1 The AlarmAPI

This object encapsulates the access to the alarm configurations database. Tango Database is used by default, all alarm configurations are stored as device properties of each declared PyAlarm device (AlarmList, AlarmReceivers, AlarmSeverities).

The api object allows to load alarms, reconfigure them and transparently move Alarms between PyAlarm devices.

4.2.2 The updateAlarms thread

This thread will be executed periodically at a rate specified by the PollingPeriod. All Enabled alarms will be evaluated at each cycle; and if evaluated to a True value (understood as any value not in (0,"",None,False,[],{})).

Once an Alarm has been active by a number of cycles equal to the device AlarmThreshold it will become Active. Then the PyAlarm will process all elements of the AlarmReceivers list.

4.2.3 The TangoEval engine

This engine will automatically replace each Tango attribute name in the formula by its value. It will also provide several methods for searching attribute names in the tango database.

Amongst other features, all values are kept in a cache with a depth equal to the AlarmThreshold+1. This cache allows to create alarms using .delta or inspecting the cache for specific behaviors.

4.3 Alarm Syntax Recipes

Alarms are parsed and evaluated using *fandango.TangoEval* class.

4.3.1 Sending a Test Message at Startup

This alarm formula is just "True" ; therefore will be enabled immediately sendin an email message to test@tester.com

```
AlarmList -> DEBUG:True
AlarmDescriptions -> DEBUG:The PyAlarm Device $NAME has been restarted
AlarmReceivers -> DEBUG: test@tester.com
```

4.3.2 Testing a device availability

It is done if you put directly the name of the device or its State as a condition by itself. In the second case and alarm will be triggered either if the Pressure is above threshold or the device is not reachable.

```
PRESSURE:SR/VC/VGCT/Pressure > 1e-4
STATE_AND_PRESSURE:?SR/VC/VGCT and SR/VC/VGCT/Pressure > 1e-4
```

4.3.3 Getting Tango state/attribute/value/quality/time/delta in formulas

The Alarm syntax allows to add the following clauses to the attribute name (value returned by default):

```
some/device/name{/attribute}{.value/all/time/quality/delta/exception}
```

attribute: if no attribute name is given, then device state is read.

```
PLC_Alarm: BL22/CT/EPS-PLC-01 == FAULT
```

value: default, returns the value of the attribute

```
Pressure_Alarm: BL22/CT/EPS-PLC-01/CC1_AF.value > 1e-5
```

time: returns the epoch in seconds of the last value read

```
Not_Updated: BL22/CT/EPS-PLC-01/CPU_Status.time < (now-60)
```

quality : returns the tango quality value (ATTR_VALID, ATTR_INVALID, ATTR_WARNING, ATTR_ALARM).

```
Temperature_Alarm: BL22/CT/EPS-PLC-01/OP_WBAT_OH01_01_TC11.quality == ATTR_ALARM
```

delta : returns the variation of the value in the last N=AlarmThreshold reads (stored in TangoEval.cache array of size AlarmThreshold+1)

```
Valve_Just_Closed: BL22/CT/EPS-PLC-01/VALVE_11.delta == -1
```

exception : True if the attribute is unreadable, False otherwise

```
Not_Found: BL22/CT/EPS-PLC-01/I_Dont_Exist.exception
```

all : returns the raw attribute object as returned by PyTango.DeviceProxy.read_attribute method.

4.3.4 Creating a periodic self-reset alarm

A simple clock alarm would use the current time and will set AlarmThreshold, PollingPeriod and AutoReset properties. See this example:

> https://github.com/tango-controls/PANIC/blob/documentation/doc/recipes/CustomAlarms.rst#clock-alarm-triggered-by-time

A single formula clock would be more hackish; this alarm will execute a command on its own formula

```
PERIODIC:(FrontEnds/VC/Elotech-01/Temperature and FrontEnds/VC/VGCT-01/P1 \
and (1920<(now%3600)<3200)) or (ResetAlarm('PERIODIC') and False)
```

4.3.5 Enabling search, expression matching and list comprehensions

Having the syntax `dom/fam/mem/attr.quality` whould allow us to call attrs like:

```
any([ATTR_ALARM==s+'.quality' for s in FIND('dom/fam/*/pressure')])
```

One way may be using QUALITY, VALUE, TIME key functions:

```
any([ATTR_ALARM==QUALITY(s) for s in FIND('dom/fam/*/pressure')])
```

The use of FIND allows PyAlarm to prepare a list Taurus models that can be redirected from an <pre>event_received(...)</pre> hook.

4.3.6 Some list comprehension examples

```
any([s for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

equals to

```
any(FIND(SR/ID/SCW01/Cooler*Err*))
```

The negate:

```
any([s==0 for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

is equivalent to

```
any(not s for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

is equivalent to

```
not all(FIND(SR/ID/SCW01/Cooler*Err*))
```

is equivalent to

```
[s for s in FIND(SR/ID/SCW01/Cooler*Err*) if not s]
```

4.3.7 Grouping Alarms in Formulas

The proper way is (for readability I use upper case letters for alarms):

```
ALARM_1: just/my/tango/attribute_1
ALARM_2: just/my/tango/attribute_2
```

then:

```
ALARM_1_OR_2: ALARM_1 or ALARM_2
```

or:

```
ALARM_1_OR_2: any(( ALARM_1 , ALARM_2 ))
```

or:

```
ALARM_ANY: any( FIND(my/alarm/device/ALARM_*) )
```

Any alarm you declare becomes both a PyAlarm attribute and a variable that you can anywhere (also in other PyAlarm devices). You don't trigger any new read because you just use the result of the formula already evaluated.

The GROUP is used to tell you that a set of conditions has changed from its previous state. GROUP instead will be triggered not if any is True, but if any of them toggles to True. It forces you to put the whole path to the alarm:

```
GROUP(my/alarm/device/ALARM_[12])
```

4.4 PyAlarm Device Properties

4.4.1 Distributing Alarms between servers

Alarms can be distributed between PyAlarm servers using the PyAlarm/AlarmsList property. A Panic system works well with 1200+ alarms distributed in 75 devices, with loads between 5 and 70 attrs/device. But instead of thinking in terms of N attrs/pyalarm you must distribute load trying to group all attributes from the same host or subsystem.

There are two reasons to do that (and also apply to Archiving):

- When a host is down you'll have a lot of proxy threads in background trying to reconnect to lost devices. If alarms are distributed on rough numbers it becomes a lot of timeouts spreading through the system. When alarms are grouped by host you isolate the problems.
- Same applies for very event-intensive devices. Devices that generate a lot of information will need lower attrs/pyalarm ratio than devices that do not change so much.

But, it is a good advice to keep the overall number of alarms in the system below 10K alarms. For manageability of the log system and avoid avalanches of useless information the logical number of alarms should be around or below 1000.

4.4.2 Alarm Declaration Properties

AlarmList

Format of alarms will be:

```
TAG1:LT/VC/Dev1
TAG2:LT/VC/Dev1/State
TAG3:LT/VC/Dev1/Pressure > 1e-4
```

NOTE: This property was previously called AlarmsList; it is still loaded if AlarmList is empty for backward compatibility

AlarmDescriptions

Description to be included in emails for each alarm. The format is:

```
TAG:AlarmDescriptions...
```

NOTE: Special Tags like $NAME (for name of PyAlarm device) or $TAG (for name of the Alarm) will be automatically replaced in description.

AlarmReceivers

```
TAG1:vacuum@accelerator.es,SMS:+34935924381,file:/tmp/err.log
vacuum@accelerator.es:TAG1,TAG2,TAG3
```

Other options are SNAP or ACTION:

```
user@cells.es,
SMS:+34666777888, #If SMS sending available
SNAP, #Alarm changes will be recorded in SNAP database.
ACTION(alarm:command,mach/alarm/beep/play_sequence,$DESCRIPTION)
```

Or Telegram messages, see:

https://github.com/tango-controls/PANIC/blob/documentation/doc/recipes/TelegramSetup.rst

Adding ACTION as receiver

Executing a command on alarm/disable/reset/acknowledge:

```
ACTION(alarm:command,mach/alarm/beep/play_sequence,$DESCRIPTION)
```

The syntax allow both attribute/command execution and the usage of multiple typed arguments:

```
ACTION(alarm:command,mach/dummy/motor/move,int(1),int(10))
ACTION(reset:attribute,mach/dummy/motor/position,int(0))
```

Also commands added to the Class property @AllowedCommands@ can be executed:

```
ACTION(alarm:system:beep&)
```

PhoneBook (not implemented yet)

File where alarm receivers aliases are declared; e.g.

```
User:user@accelerator.es;SMS:+34666555666
```

Default location is: ‘‘ *$HOME/var/alarm_phone_book.log* ‘‘

If User and Operator are defined in phonebook, AlarmsReceivers can be:

```
TAG2:User,Operator
```

4.4.3 REMINDER / RECOVERED / AUTORESET messages

Reminder

If a number of seconds is set, a reminder mail will be sent while the alarm is still active, if 0 no Reminder will be sent.

AlertOnRecovery

A message is sent if an alarm is active but the conditions of the attributes return to a safe value. To enable the message the content of this property must contain ‘email’, ‘sms’ or both. If disabled no RECOVERY/AUTO-RESET messages are sent.

AutoReset

If a number of seconds is set, the alarm will reset if the conditions are no longer active after the given interval.

4.4.4 Snapshot properties

UseSnap

If false no snapshots will be trigered (unless specifically added to receivers using “SNAP”),

CreateNewContexts

It enables PyAlarm to create new contexts for alarms if no matching context exists in the database.

4.4.5 Alarm Configuration Properties

(In future releases these properties could be individually configurable for each alarm)

Enable : If False forces the device to Disabled state and avoids messaging.

LogFile : File where alarms are logged Default: *"/tmp/alarm_$NAME.log"*

FlagFile : File where a 1 or 0 value will be written depending if theres active alarms or not.n
This file can be used by other notification systems. Default: *"/tmp/alarm_ds.nagios"*

PollingPeriod : Periode in seconds. in which all attributes not event-driven will be polled. Default: *60000*

MaxAlarmsPerDay : Max Number of Alarms to be sent each day to the same receiver. Default: *3*

AlarmThreshold : Min number of consecutive Events/Pollings that must trigger an Alarm. Default: *3*

FromAddress : Address that will appear as Sender in mail and SMS Default: *"controls"*

SMSConfig : Arguments for sendSMS command Default: ":"

MaxMessagesPerAlarm : To avoid the previous property to send a lot of messages continuously this property has been added to limit the maximum number of messages to be sent each time that an alarm is enabled/recovered/reset.

StartupDelay : Time that PyAlarm waits before starting the Alarm evaluation threads.

EvalTimeout : Timeout for read_attribute calls, in milliseconds .

UseProcess : To create new OS processes instead of threads.

4.5 Device Server Example

These will be the typical properties of a PyAlarm device

```
#--------------------------------------------------------
# SERVER PyAlarm/AssemblyArea, PyAlarm device declaration
#--------------------------------------------------------
PyAlarm/AssemblyArea/DEVICE/PyAlarm: "LAB/VC/Alarms"
# --- LAB/VC/Alarms properties
LAB/VC/Alarms->AlarmDescriptions: "OVENPRESSURE:The pressure in the Oven exceeds Range
↪",\
                                  "ADIXENPRESSURE:The pressure in the Roughing Station␣
↪exceeds Range",\
                                  "OVENTEMPERATURE:The Temperature of the Oven exceeds␣
↪Range",\
                                  "DEBUG:Just for debugging purposes"
LAB/VC/Alarms->AlarmReceivers: OVENPRESSURE:somebody@cells.es,someone_else@cells.es,
↪SMS:+34999666333,\
                               ADIXENPRESSURE:somebody@cells.es,someone_else@cells.es,
↪SMS:+34999666333,\
                               OVENTEMPERATURE:somebody@cells.es,someone_else@cells.es,
↪SMS:+34999666333,\
```

(continues on next page)

(continued from previous page)

```
                            DEBUG:somebody@cells.es
LAB/VC/Alarms->AlarmsList: "OVENPRESSURE:LAB/VC/BestecOven-1/Pressure_mbar > 5e-4",\
                           "OVENRUNNING:LAB/VC/BestecOven-1/MaxValue > 70",\
                           "ADIXENPRESSURE:LAB/VC/Adixen-01/P1 > 1e-4 and OVENRUNNING",\
                           "OVENTEMPERATURE:LAB/VC/BestecOven-1/MaxValue > 220",\
                           "DEBUG:OVENRUNNING and not PCISDOWN"
LAB/VC/Alarms->PollingPeriod: 30
LAB/VC/Alarms->SMSConfig: ...
```

4.6 Mail Messages

PyAlarm allows to send mail notifications. Each alarm may be configured with AlarmReceivers property to provide notification list. There is also a *GobalReceivers* property which allows to define notification for all alarms.

PyAlarm supports two ways of sending mails configured with the *MailMethod* class property:

- using *mail* shell command, when *MailMethod* is set to *mail*, which is default,
- or using *smtplib* python library when *MailMethod* is set to *smtp[:host[:port]]*.

When using *mail* method it setup *from* variable as '-S' option (see: https://linux.die.net/man/1/mail). However, some setups may require to use *-r* option additionally. To enable it set *MailDashRoption* class property with a proper mail address.

As it is now, mail messages are formatted as the following:

4.6.1 Format of Alarm message

```
Subject:     LAB/VC/Alarms: Alarm RECOVERED (OVENTEMPERATURE)
Date:     Wed, 12 Nov 2008 11:52:39 +0100

TAG: OVENTEMPERATURE
          LAB/VC/BestecOven-1/MaxValue > 220 was RECOVERED at Wed Nov 12 11:52:39 2008

Alarm receivers are:
          somebody@cells.es
          someone_else@cells.es
Other Active Alarms are:
          DEBUG:Fri Nov  7 18:37:35 2008:OVENRUNNING and not PCISDOWN
          OVENRUNNING:Fri Nov  7 18:37:17 2008:LAB/VC/BestecOven-1/MaxValue > 70
Past Alarms were:
          OVENTEMPERATURE:Fri Nov  7 20:49:46 2008
```

4.6.2 Format of Recovered message

```
Subject:     LAB/VC/Alarms: Alarm RECOVERED (OVENTEMPERATURE)
Date:     Wed, 12 Nov 2008 11:52:39 +0100

TAG: OVENTEMPERATURE
          LAB/VC/BestecOven-1/MaxValue > 220 was RECOVERED at Wed Nov 12 11:52:39 2008
```

(continues on next page)

(continued from previous page)

```
Alarm receivers are:
          somebody@cells.es
          someone_else@cells.es
Other Active Alarms are:
          DEBUG:Fri Nov  7 18:37:35 2008:OVENRUNNING and not PCISDOWN
          OVENRUNNING:Fri Nov  7 18:37:17 2008:LAB/VC/BestecOven-1/MaxValue > 70
Past Alarms were:
          OVENTEMPERATURE:Fri Nov  7 20:49:46 2008
```

CHAPTER 5

PANIC Recipes

5.1 Alarms Distribution

5.1.1 About distributing load (answer to paul bell, 2014)

We have 1200+ alarms and system works quite well with it. But regarding distribution of PyAlarm devices and servers the rules must be more intelligent.

Instead of thinking in terms of N attrs/pyalarm you must distribute load trying to group all attributes from the same host or subsystem.

There are two reasons to do that (and also apply to Archiving):

- When a host is down you'll have a lot of proxy threads in background trying to reconnect to lost devices. If alarms are distributed on rough numbers it becomes a lot of timeouts spreading through the system. When alarms are grouped by host you isolate the problems.
- Same applies for very event-intensive devices. Devices that generate a lot of information will need lower attrs/pyalarm ratio than devices that do not change so much.

Apart of that ... if you have 1000 alarms just for the linac then you may have a wrong specification. I use to say than "all" should be in the order of 10K ; by experience any number about that is too much. If you need more than 10K of a kind what you really need is to add a level of abstraction (do not check all gauges of a vacuum section, just had an attribute where you can read the max value).

It applies to all Tango systems I've seen (alarms, archiving, save/restore, pool, device tree, ...); if you reach a number above 10K then you must add an abstraction layer. It's not only that you reach a performance limit, also your users will feel too dazed and confused when searching for things.

e.g. Our accelerator group requested 1200 alarms ... and after some months they asked for a filter to show only the 240 they really care about.

5.2 Alarm Formulas Examples

Contents

Alarms are parsed and evaluated using *fandango.TangoEval* class.

5.2.1 Sending a Test Message at Startup

This alarm formula is just "True" ; therefore will be enabled immediately sendin an email message to test@tester.com

```
AlarmList -> DEBUG:True
AlarmDescriptions -> DEBUG:The PyAlarm Device $NAME has been restarted
AlarmReceivers -> DEBUG: test@tester.com
```

5.2.2 Testing a device availability

It is done if you put directly the name of the device or its State as a condition by itself. In the second case and alarm will be triggered either if the Pressure is above threshold or the device is not reachable.

```
PRESSURE:SR/VC/VGCT/Pressure > 1e-4
STATE_AND_PRESSURE:?SR/VC/VGCT and SR/VC/VGCT/Pressure > 1e-4
```

5.2.3 Getting Tango state/attribute/value/quality/time/delta in formulas

The Alarm syntax allows to add the following clauses to the attribute name (value returned by default):

```
some/device/name{/attribute}{.value/all/time/quality/delta/exception}
```

attribute: if no attribute name is given, then device state is read.

```
PLC_Alarm: BL22/CT/EPS-PLC-01 == FAULT
```

value: default, returns the value of the attribute

```
Pressure_Alarm: BL22/CT/EPS-PLC-01/CC1_AF.value > 1e-5
```

time: returns the epoch in seconds of the last value read

```
Not_Updated: BL22/CT/EPS-PLC-01/CPU_Status.time < (now-60)
```

quality : returns the tango quality value (ATTR_VALID, ATTR_INVALID, ATTR_WARNING, ATTR_ALARM).

```
Temperature_Alarm: BL22/CT/EPS-PLC-01/OP_WBAT_OH01_01_TC11.quality == ATTR_ALARM
```

delta : returns the variation of the value in the last N=AlarmThreshold reads (stored in TangoEval.cache array of size AlarmThreshold+1)

```
Valve_Just_Closed: BL22/CT/EPS-PLC-01/VALVE_11.delta == -1
```

exception : True if the attribute is unreadable, False otherwise

```
Not_Found: BL22/CT/EPS-PLC-01/I_Dont_Exist.exception
```

all : returns the raw attribute object as returned by PyTango.DeviceProxy.read_attribute method.

5.2.4 Creating a periodic self-reset alarm

A simple clock alarm would use the current time and will set AlarmThreshold, PollingPeriod and AutoReset properties. See this example:

> https://github.com/tango-controls/PANIC/blob/documentation/doc/recipes/CustomAlarms.rst#clock-alarm-triggered-by-time

A single formula clock would be more hackish; this alarm will execute a command on its own formula

```
PERIODIC:(FrontEnds/VC/Elotech-01/Temperature and FrontEnds/VC/VGCT-01/P1 \
and (1920<(now%3600)<3200)) or (ResetAlarm('PERIODIC') and False)
```

5.2.5 Enabling search, expression matching and list comprehensions

Having the syntax `dom/fam/mem/attr.quality` whould allow us to call attrs like:

```
any([ATTR_ALARM==s+'.quality' for s in FIND('dom/fam/*/pressure')])
```

One way may be using QUALITY, VALUE, TIME key functions:

```
any([ATTR_ALARM==QUALITY(s) for s in FIND('dom/fam/*/pressure')])
```

The use of FIND allows PyAlarm to prepare a list Taurus models that can be redirected from an <pre>event_received(...)</pre> hook.

5.2.6 Some list comprehension examples

```
any([s for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

equals to

```
any(FIND(SR/ID/SCW01/Cooler*Err*))
```

The negate:

```
any([s==0 for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

is equivalent to

```
any(not s for s in FIND(SR/ID/SCW01/Cooler*Err*)])
```

is equivalent to

```
not all(FIND(SR/ID/SCW01/Cooler*Err*))
```

is equivalent to

```
[s for s in FIND(SR/ID/SCW01/Cooler*Err*) if not s]
```

5.2.7 Grouping Alarms in Formulas

The proper way is (for readability I use upper case letters for alarms):

```
ALARM_1: just/my/tango/attribute_1
ALARM_2: just/my/tango/attribute_2
```

then:

```
ALARM_1_OR_2: ALARM_1 or ALARM_2
```

or:

```
ALARM_1_OR_2: any(( ALARM_1 , ALARM_2 ))
```

or:

```
ALARM_ANY: any( FIND(my/alarm/device/ALARM_*) )
```

Any alarm you declare becomes both a PyAlarm attribute and a variable that you can anywhere (also in other PyAlarm devices). You don't trigger any new read because you just use the result of the formula already evaluated.

The GROUP is used to tell you that a set of conditions has changed from its previous state. GROUP instead will be triggered not if any is True, but if any of them toggles to True. It forces you to put the whole path to the alarm:

```
GROUP(my/alarm/device/ALARM_[12])
```

5.2.8 Alarm on delta and value

This alarm will be triggered whenever a channel (HV*Code attributes) changes its value (delta!=0) and the new value is OFF (value=0)

```
any([(changed and value==0) for changed,value in

zip( FIND(bl*/vc/ipct*/hv*code.delta) ,

FIND(bl*/vc/ipct*/hv*code.value) )])
```

5.2.9 Generating Clock Signals

Playing with PollingPeriod, AlarmThreshold and AutoReset properties is possible to achieve an square signal that keeps the alarm active/inactive at regular intervals.

CLOCK=NOT CLOCK

The AlarmThreshold applies to both activation and reset of the alarm, so it has to be added to the AutoReset period to regulate the duty cycle. Keeping the PollingPeriod and AutoReset values very small will generate an accurate frequency (do not expect high accuracy, that's a trick for testing but not a proper signal generator).

My values for a 10 seconds alarm cycle are:

```
.. code-block:: python
```

PollingPeriod = 0.1 AlarmThreshold = 50 AutoReset = 0.0001

If you want a more accurate alarm, you can also use the NOW() function. This example generates a switch every second

```
CLOCK = NOW()%2<1
PollingPeriod=1
AlarmThreshold-1
```

5.3 AlarmStates

Contents

5.3.1 State transitions

Alarm States and Severities are defined in panic.properties module.

With PyAlarm > 6.1; GUI will read the current Alarm state from the AlarmList attribute.

For compatibility with older versions, the events of ActiveAlarms will be used instead:

- If ActiveAlarms doesn't cotain tag, alarm.active will be 0, state = NORM
- Activealarms contains tag, alarm.active = activealarms timestamp, state = ACTIVE
- ActiveAlarms is None or Exception, alarm.active will be set to -1. state = ERROR

5.3.2 Disabled States

Their meanings are:

- OOSRV = Device server is Off (not exported), no process running

- DSUPR = Enabled property is False
- SHLVD = Alarm is listed in DisabledAlarms attribute (temporary disabled)
- ERROR = Device is alive but the alarm is not being evaluated (exported=1 and thread dead or exception).

5.3.3 IEC 62682: AlarmStates Definition and related Actions

Different annunciators can be setup for each State change

Reset() can be automatic or forced to be manual

Reminder() : Alarm still ACTIVE, additional action can be configured

RTNUN : Condition recovered (but not Reset) Alarm ACTIVE : (UNACKED) Alarm ACKED : (action taken by operator) RTNUN: return to NORM NORM: after Reset() or not triggered

First peaks ignored if (t < polling*AlarmThreshold)

SHLVD, DSUPR, OOSRV: Unactive states.

SHELVED for temporary disabling,

DSUPR by process condition,

OOSRV is permanent (device disabled).

All of them are controlled by the Enable/Disable states/commands of PyAlarm.

In addition, PANIC adds ERROR State to raise problems with Tango devices.

5.4 Hierarchies In Alarms

Contents

5.4.1 TOP/BOTTOM

The TOP/BOTTOM just provides a filter for finding alarms where the value of another alarm is used directly in the formula. It is case sensitive, so you can use lower/upper case to show/hide alarms in these filters.

To use hierarchies, alarms shall be written using the result of previous ones:

```
GAB1 = any([t >5 for t in FIND(tc1:10000/LMC/C01/GAB/*)])
GAB2 = any([t >5 for t in FIND(tc1:10000/LMC/C02/GAB/*)])
GAB_ALL= GAB1 or GAB2
OTHER = tc1:10000/LMC/C02/Other/State != ON
CAPITAL = GAB_ALL or OTHER
```

Then, the filter by hierarchy will return:

```
TOP (alarms that depend on others): CAPITAL, GAB12
BOTTOM (alarms isolated or referenced from others): OTHER, GAB_ALL, GAB1, GAB2
```

In this case GAB_ALL appears in both lists; to avoid that just rewrite it using lower case attribute names:

```
GAB_ALL = any(FIND('lmc1:10000/lmc/alarms/01/gab*'))
```

Now you should have only "CAPITAL" as TOP Alarm.

You can reproduce this behaviour from the api calling:

```
panic.AlarmAPI().filter_hierarchy('TOP')
```

5.4.2 Alarm GROUP

For an expression matching multiple alarms or attributes, GROUP returns a new formula that will evaluate to True if any of the alarm changes to active state (.delta) or matches a given condition:

```
GROUP(ALARM1, ALARM2, ALARM3)
```

Thus, GROUP will be activated when any of the three alarms switches to active; and immediately reset to wait for the next change. In this way you get a notification for any new activation of the three alarms.

NOTE: BY DEFAULT IS NOT LIKE any(FIND(*)); it will react only on change, not taking in account previous states!

NOTE2: you must tune your PyAlarm properties to have AlarmThreshold = 1 and AutoReset <= 3 to take profit of this feature.

NOTE3: The GROUP activation will be just a peak when using .delta (default); take this in account when setting up several levels of alarms as fast peaks may not be noticed if higher level alarms have long thresholds.

It uses the read_attribute schema from TangoEval, thus using .delta to keep track of which values has changed. For example, GROUP(test/alarms/*/TEST_[ABC]) will be replaced by:

```
any([t.delta>0 for d in FIND(test/alarms/*/TEST_[ABC].all)])
```

But, as regular expressions may trigger unexpected results, the syntax with explicit ALARM names is prefered.

The GROUP macro can be called with one or several expressions separated by commas and a condition separated by semicolon:

```
GROUP(expression1[,expression2;condition)
```

Expressions may contain a device name or not. If no device name is passed then it will search for it in the alarm list:

```
expression=[a/dev/name*/]attribute*
```

Thus, a valid GROUP expression is:

```
GROUP(LOCAL_ALARM1,t01:10000/an/alarm/dev/ALARM2)
```

Or

```
GROUP(LOCAL_ALARM1,t01:10000/an/alarm/dev/ALARM2;x>=1)
```

In the first case you'll get a peak when any of them changes from 0 to 1; in the second case you'll get if any of them is already on 1 (so a change in the second alarm will not trigger a second peak).

Future Releases

In future releases the GROUP macro will be capable of evaluating any tango attribute and not only alarms. **As of 6.0 this feature is not yet supported**

If the condition is empty then PyAlarm checks any .delta != 0. It can be modified if the formula contains a semicolon ";" and a condition using 'x' as variable; in this case it will be used instead of delta to check for alarm:

```
GROUP(bl09/vc/vgct-*/p[12];x>1e-5) => [x>1e-5 for x in FIND(bl09/vc/vgct-*/p[12])]
```

5.5 Special Alarm Recipes

5.5.1 Special keys used in Alarm formulas

- DEVICE: PyAlarm device name
- DOMAIN,FAMILY,MEMBER: Parts of the device name
- ALARMS: Alarms managed by this device
- PANIC: API containing all declared alarms
- t: time since the device was started
- T(...): string to time
- str2time(...): string to time
- now, NOW(): current timestamp
- DEVICES: instantiated devices
- DEV(device): DeviceProxy(device)
- NAMES(expression'): Finds all attributes matching the expression and return its names.
- CACHE: Saved values
- PREV: Previous values
- READ(attr): TangoEval.read_attribute(attr)
- FIND('expression'): Finds all attributes matching the expression and return its values.

5.5.2 Expiration Date

Disabling or re-enabling after a given date

A temporal condition can be achieved using the T() macro in the formula.

To disable an Alarm after a given date:

```
T() < T('2013-04-23') and D/F/M.A > V1
```

To re-enable it after a maintenance period:

```
T() > T('2013-04-23') and D/F/M.A > V1
```

5.5.3 Accessing PyAlarm Values CACHE

The PyAlarm CACHE dictionary contains the last values stored for each tango attribute that appeared in the formulas. The size of cache is AlarmTrheshold + 1

Usage:

```
PASS_BY_0=[(k,v.time.tv_sec,str(v.value)) for k,t in CACHE.items() for v in t if v.
↪value==0]
```

This will trigger alarm if ALL values in the cache are equal, it is NOT the same as Delta because it checks only the first and last values:

```
not (lambda l:max(l)-min(l))([v.value for v in CACHE['wr/rf/circ-1/heartbeat']])
```

5.5.4 Clock: Alarm triggered by time

This alarm will be enabled/disabled every 5 seconds.

First, create a new PyAlarm device:

```
import fandango as fn
fn.tango.add_new_device('PyAlarm/Clock','PyAlarm','test/pyalarm/clock')
```

Add the new alarm (formula will use current time to switch True/False very 5 seconds)

```
from panic import AlarmAPI
alarms = AlarmAPI()
alarms.add(device='test/pyalarm/clock',tag='CLOCK',formula='NOW()%10<5')
```

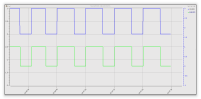
Start your device server using Astor, fandango or manually

```
import fandango as fn
fn.Astor('test/pyalarm/clock').start_servers(host='your_hostname')
```

Then, configure the device properties to react every second for both activation and reset:

```
dtest = alarms.devices['test/pyalarm/clock']
dtest.get_config()
dtest.config['Enabled'] = 1
dtest.config['AutoReset'] = 1
dtest.config['AlarmThreshold'] = 1
dtest.config['PollingPeriod'] = 1
alarms.put_db_properties(dtest.name,dtest.config)
dtest.init()
```

This is the result you can expect when plotting test/pyalarm/clock/CLOCK in a taurustrend:



5.6 Exception Management

Alarm properties that control if exceptions trigger alarms or not . . .

'RethrowState': [PyTango.DevBoolean, "Whether exceptions in State reading will be rethrown.", [True]],#Overriden by panic.DefaultPyAlarmProperties

'RethrowAttribute': [PyTango.DevBoolean, "Whether exceptions in Attribute reading will be rethrown.", [False]],#Overriden by panic.DefaultPyAlarmProperties

'IgnoreExceptions': [PyTango.DevBoolean, "If True unreadable values will be replaced by None instead of Exception.", [True]],#Overriden by panic.DefaultPyAlarmProperties

5.7 Grouping Alarms

The proper way is (for readability I use upper case letters for alarms):

ALARM_1: just/my/tango/attribute_1 ALARM_2: just/my/tango/attribute_2

then:

ALARM_1_OR_2: ALARM_1 or ALARM_2

or:

ALARM_1_OR_2: any((ALARM_1 , ALARM_2))

or:

ALARM_ANY: any(FIND(my/alarm/device/ALARM_*))

Any alarm you declare becomes both a PyAlarm attribute and a variable that you can anywhere (also in other PyAlarm devices). You don't trigger any new read because you just use the result of the formula already evaluated.

The GROUP is used to tell you that a set of conditions has changed from its previous state. GROUP instead will be triggered not if any is True, but if any of them toggles to True. It forces you to put the whole path to the alarm:

GROUP(my/alarm/device/ALARM_[12])

5.8 How PyAlarm Device Server Works

This document tries to summarize how PyAlarm processes alarms and executes its actions. A full explanation of alarm syntax and each property is available in the PyAlarm user guide, but here I provide a summary for convenience.

The device server behaviour relies on three python objects: AlarmAPI, updateAlarms thread and TangoEval.

Each alarm is independent in terms of formula and receivers; but all alarms within the same PyAlarm device will share a common evaluation environment determined by PyAlarm properties.

Contents

5.8.1 The AlarmAPI

This object encapsulates the access to the alarm configurations database. Tango Database is used by default, all alarm configurations are stored as device properties of each declared PyAlarm device (AlarmList, AlarmReceivers, AlarmSeverities).

The api object allows to load alarms, reconfigure them and transparently move Alarms between PyAlarm devices.

5.8.2 The updateAlarms thread

This thread will be executed periodically at a rate specified by the PollingPeriod. All Enabled alarms will be evaluated at each cycle; and if evaluated to a True value (understood as any value not in (0,"",None,False,[],{})).

Once an Alarm has been active by a number of cycles equal to the device AlarmThreshold it will become Active. Then the PyAlarm will process all elements of the AlarmReceivers list.

AlertOnRecovery and AlarmReset

Whenever an alarm formula becomes True; a counter starts to increase until it reaches the AlarmThreshold value, becoming an active alarm.

This counter is kept at AlarmThreshold value and starts decreasing once the formula is no longer True. If the counter reaches 0 (its minimum value) the alarm will be still active but its new state will be RECOVERED, an email will be sent to receivers if AlertOnRecovery property is True.

Then, if the AlarmReset value (in seconds) is distinct from 0, a time count starts from the point of RECOVERY. If there's no change in the alarm state during this time count, the alarm will be automatically RESET (notifying receivers or not depending on configuration).

So, if you need an alarm to have a fast recovery keep in mind that you'll have to apply a delay equal to AlarmThreshold+PollingPeriod to the value that you have set as AutoReset.

5.8.3 The TangoEval engine

This engine will automatically replace each Tango attribute name in the formula by its value. It will also provide several methods for searching attribute names in the tango database.

Amongst other features, all values are kept in a cache with a depth equal to the AlarmThreshold+1. This cache allows to create alarms using .delta or inspecting the cache for specific behaviors.

5.9 PANIC Setup

by Sergi Rubio — 2006, 2016

Contents

5.9.1 Description

The Package for Alarms and Notification of Incidents from Controls

PANIC Alarm System is a set of tools (api, Tango device server, user interface) that provides:

- Periodic evaluation of a set of conditions.
- Notification (email, sms, pop-up, speakers)
- Keep a log of what happened. (files, Tango Snapshots)
- Taking automated actions (Tango commands / attributes)
- Tools for configuration/visualization.

Other Documentation in this same repository

- PANIC presentation at PCAPAC'14: Panic Talk at PCAPAC'14
- The Panic python API: PanicAPI.rst
- The PyAlarm User Guide: PyAlarmUserGuide.rst
- The Panic UI manual: panicdoc.html

5.9.2 Launch your PANIC System in few steps

Dependencies

You must have PyTango + Tango + MySQL up and running and your TANGO_HOST and PYTHONPATH environment variables properly set.

PyTango is available at PyPI: https://pypi.python.org/pypi/PyTango

Get the code

ALL OF THIS IS DEPRECATED; GET THE PACKAGES FROM https://github.com/tango-controls INSTEAD

Fandango library (functional tools for tango) is required to be in your PYTHONPATH:

```
svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/share/fandango/trunk/
↪fandango fandango
```

You can download PyAlarm and the panic api from tango-ds at sourceforge:

```
svn co https://svn.code.sf.net/p/tango-ds/code/DeviceClasses/SoftwareSystem/PyAlarm/
↪trunk
```

The PANIC User Interface is available in the /clients branch:

```
svn co  https://svn.code.sf.net/p/tango-ds/code/Clients/python/Panic/trunk
```

Setup your Tango database

Create your devices from a python console (or Jive):

```
import PyTango
db = PyTango.Database()


def add_new_device(server,klass,device):
dev_info = PyTango.DbDevInfo()
dev_info.name = device
dev_info.klass = klass
dev_info.server = server
get_database().add_device(dev_info)


#Create a PyAlarm device
add_new_device('PyAlarm/1','PyAlarm','test/alarms/1')


#I'll add a simulator, but you can't use TangoTest or whatever device you want:
add_new_device('PySignalSimulator/1','PySignalSimulator','test/sim/1')
db.put_device_property('test/sim/1',{'DynamicAttributes':['A=t%100']})
```

From shell, launch your PyAlarm and Simulator devices:

```
# python PyAlarm/PyAlarm.py 1 &
# python PySignalSimulator/PySignalSimulator.py 1 &
```

Create a TEST_ALARM using the API:

```
import panic
alarms = panic.api()
alarms.add('TEST_ALARM',formula='(test/sim/1/A%15 > 5)',description='test',receivers=
↪'your@mail')
```

Run the panic application and configure your Alarms

```
python Panic/gui.py
```

See the application manual: http://plone.tango-controls.org/tools/panic/panic-ui/

If you want to see faster changes in the alarm cycle try to set the following configuration values (Tools->Adv.Config):

```
PollingPeriod = 1
AlarmThreshold = 1
AutoReset = 5
Notification Services
```

The syntax for sending an email (from linux, you'll need the "mail" command available in the system, from windows you'll have to set as receiver a command from a device running in a linux machine):

```
DeviceProxy("your/alarm/device").command_inout("SendMail",["Bonjour,\n\nthis is a␣
↪test message\n\nau revoire","RE: testing","your-name@tango-controls.org"])
```

The other command we have for notification is SendSMS; but it requires our smslib.py file that is specific to our SMS provider (it uses http transactions to send the messages). If you're interested on it you'll have to write your own smslib.py file to use it.

FestivalDS, Speech and pop-ups

There's another notification device you can use, the FestivalDS. It provides speech synthesizing and pop-ups in a linux environment (it requires "festival" and "libnotify-bin" linux packages):

```
https://svn.code.sf.net/p/tango-ds/code/DeviceClasses/InputOutput/FestivalDS/trunk
```

The commands are:

```
Play(string): speech to speakers
Beep(): beep!
Play_sequence(string):  it just makes some beeps before and after the speech
PopUp(title,text,[seconds]): shows a pop-up with title/text for the given time
```

And that's all regarding our current notifiers, for database we don't have anything yet, as we use the device properties to store all the data. You'll find more information in the PyAlarm user guide.

5.10 Exception Management in Panic Alarms

The exception management will be done using the _raise=RAISE argument of the TangoEval.eval method.

Three properties control if exceptions will enable the alarm or will be simply ignored.

IgnoreExceptions if False then all exceptions will be registered as FailedAlarms and the PyAlarm will change to FAULT whenever an exception is encountered. If no rethrow option is active, FailedAlarms will be displayed in grey in AlarmGUI as "disabled".

RethrowAttribute if True, any exception in the formula will set the alarm as active. PyAlarm state will change to ALARM or FAULT if IgnoreExceptions is False and all alarms are in failed state.

RethrowState if True, only alarms reading State attributes will be activated by exception. PyAlarm state will change to ALARM or FAULT if IgnoreExceptions is False and all alarms are in failed state.

So, in case of having an alarm reading a faulty attribute, the status of the alarm will be:

DISABLED If IgnoreExceptions=False and RethrowAttribute=False

NOT ACTIVE If IgnoreExceptions=True and RethrowAttribute=False

ACTIVE If IgnoreExceptions=False and RethrowAttribute=True

ACTIVE If IgnoreExceptions=True and RethrowAttribute=True

5.11 Using the PANIC python API

Contents

5.11.1 The Panic Module

Panic contains the python AlarmAPI for managing the PyAlarm device servers from a client application or a python shell. The panic module is part of the Panic bliss package.:

```
import panic
alarms = panic.api()
```

5.11.2 Browsing existing alarms

The AlarmAPI is a dictionary-like object containing Alarm objects for each registered Alarm tag. In addition the AlarmAPI.get method allows caseless search by tag, device, attribute or receiver:

```
alarms.get(self, tag='', device='', attribute='', receiver='')

alarms.get(device='boreas')
Out[232]:
 [Alarm(BL29-BOREAS_STOP:The BakeOut controller has been stop),
 Alarm(BL29-BOREAS_PRESSURE_1:),
 Alarm(BL29-BOREAS_PRESSURE_2:),
 Alarm(BL29-BOREAS_START: BL29-BOREAS bakeout started
 ...]

alarms.get(receiver='eshraq')
Out[234]:
 [Alarm(RF_LOST_EUROTHERM:),
 Alarm(OVEN_COMMS_FAILED:Oven temperatures not updated in the last 5 minutes),
 Alarm(RF_PRESSURE:The pressure in the cavity exceeds Range),
 Alarm(OVEN_TEMPERATURE:The Temperature of the Oven exceeds Range),
 Alarm(RF_EUROTHERM:),
 Alarm(RF_LOST_MKS:),
 Alarm(RF_TEMPERATURE_MAX2:),
 ...]

alarms['RF_LOST_MKS'].receivers
Out[237]: '%SRUBIO,%ESHRAQ,%VACUUM,%LOTHAR,%JNAVARRO'
```

5.11.3 Adding / Removing alarms

The add/remove methods take care of properties modification:

```
alarms.add('RF_ON_FIRE','rf/ct/alarms',formula='rf/ct/plc-01/temperature>1000.',
↪message='FIRE!',receivers='rf@cells.es,plc@cells.es')

alarms.remove('RF_ON_FIRE')
```

5.11.4 Modifying alarms

Each Alarm object contains strings with its configuration, if you modify it you must call Alarm.write() method to update the alarm device. An Alarm.rename() method is also available.

In [235]: alarms['RF_LOST_MKS'].device Out[235]: 'sr/rf/alarms'

In [236]: alarms['RF_LOST_MKS'].formula Out[236]: 'SR/RF/VGCT-01/State==UNKNOWN or SR/RF/VGCT-02/State==UNKNOWN'

In [237]: alarms['RF_LOST_MKS'].receivers Out[237]: '%SRUBIO,%ESHRAQ,%VACUUM,%LOTHAR,%JNAVARRO'

In [238]: alarms['RF_LOST_MKS'].write()

5.11.5 Modifying a receiver in all alarms

And a fast way for updating alarm receivers:

```
[a.replace_receiver('%DFERNANDEZ','%SRUBIO') for a in alarms.get(receiver='fernandez
↪')]
```

5.12 PanicAdminUsers property

Contents

The PanicAdminUsers property will contain all users enabled to modify an alarm.

Although, any user identified as an email receiver of an alarm will be allowed to change it.

The propery is check from the get_admins_for_alarm() method in AlarmAPI.

The method will be used to call the setAllowedUsers() of a validator plugin.

The methods that the i*ValidatedWidget decorator requires of a validator are:

- setLogging()
- setAllowedUsers()
- setLogMessage()
- exec_()

User validation in the GUI will be kept for consecutive actions as long as the allowed users list for each action doesn't change. If a new action is required on an Alarm with different receivers, the login will be asked again.

The login will be kept for a time defined by *PyAlarm.PanicUserTimeout* property. This time is 60 seconds by default.

5.13 PyAlarm Startup Modes

The PyAlarm Startup is controlled by **StartupDelay** and **Enabled** properties.

StartupDelay will put the PyAlarm in *PAUSED* state after a restart; to not start to evaluate formulas immediately but after some seconds, thus giving time to other devices to start.

The **Enabled** property will instead control the notification actions:

- If *False*, no notification will be triggered.
- If *True*, all notifications can be sent once **StartupDelay** has passed.
- If a *Number* is given, all notifications triggered between startup and `t+Enabled` will be ignored.
- `Enabled>(AlarmThreshold*PollingPeriod)`: "*Silent restart*", activates the Alarms that were presumably active before a restart; but do not retriggers the notifications.

`Enabled = 120` is the typical case; not triggering notifications until the device has been running for at least 3 minutes.

If `Enabled = False` or while `t < Start+Enabled` the PyAlarm State will be **DISABLED**.

5.14 PyAlarm timing configuration

- StartupDelay: the device will wait before starting to evaluate the alarms (e.g. giving some time to the system to recover from a powercut).
- Enabled: if False or 0 the PyAlarm it equals to disabling all alarm actions of the device; if it is True the behavior will be the normal expected; if it has a numeric value (e.g. 120) it means that the device will evaluate the alarms but not execute actions during the first 120 seconds (thus alarms can be activated but no action executed). It is used to prevent a restart of the device to re-execute all alarms that were already active.
- EvalTimeout: The proxy timeout used when evaluating the attributes (any read attribute slower than timeout will raise exception).
- AlarmThreshold: number of cycles that an alarm must evaluate to True to be considered active (to avoid alarms on "glitches").
- RethrowAttribute/RethrowState: Whether exceptions on reading attributes or states should be rethrown to higher levels, thus causing the alarm to be triggered. By default alarms are enabled if an State attribute is not readable (RethrowState=True), but when a numeric attribute is not readable its value is just replaced by None (RethowAttribute=False) and the formula evaluated normally.
- Reminder: A new email will be sent every XX seconds if the alarm remains active. When AlertOnRecovery is True an email will be sent also every time when the formula result oscillates from True to False.
- UseProcess: This is an experimental feature, like UseTaurus and others. In general, I advice you to not modify any parameter that is not detailed in the PyAlarm user guide as you may obtain unexpected results. Some parameters are used to test new features still under development and their behavior may vary between commits.

Regarding actions on recovery . . . this option is planned but not yet fully available. Actually just emails are sent when AlertOnRecovery is True. This feature may be implemented in the next 6 months or so but the syntax is still to be decided.

5.15 Testing your PyAlarm installation

This script will check the current performance of your PyAlarm devices:

```
> TANGO_HOST=your_hostname:10000 python panic/extra/report.py check
```

5.16 PANIC Receivers, Logging and Actions

Contents

5.16.1 Alarm Receivers

Allowed receivers are email, sms, action and shell commands.

5.16.2 SMS / Mail Config

These CLASS properties will control how SMS and Mail is configured:

SMSConfig

SMSMaxLength

SMSMaxPerDay

FromAddress

MailMethod

5.16.3 Global Receivers

The PyAlarm class property "GlobalReceivers" allows to set receivers that will be applied to all Alarms; independently of the device that is managing them.

The syntax is:

```
GlobalReceivers
  {regexp}:{receivers}
  .*:oncall@facility.dom
```

5.16.4 Logging

Alarm logging can be managed in three ways: local logs, remote logs via FolderDS or Snapshoting.

All the logging methods support defined variables ($ALARM, $DATE, $DEVICE, $MESSAGE, $VALUES, $...)

Local LogFile

Simply set the LogFile property to your preferred local file path:

```
LogFile = /tmp/pyalarm/$NAME_$DATE_$MESSAGE.log
```

Remote LogFile

You can use the fandango.FolderDS device to specify a remote logfile destination on the LogFile property:

```
# LogFile = tango://[folderds/device/name]/[logfile_name]
LogFile = tango://sys/folder/panic-logs/$NAME_$DATE_$MESSAGE.log
```

You can have both local and remote logging by setting LogFile to a local file and adding an ACTION receiver:

```
LogFile = /tmp/pyalarm/$NAME_$DATE_$MESSAGE.log

AlarmReceivers = ACTION(alarm:command,controls02:10000/test/folder/tmp-folderds/
↪SaveText,
                            '$NAME_$DATE_$MESSAGE.txt','$REPORT')
```

FolderDS documentation: https://github.com/tango-controls/fandango/blob/documentation/doc/devices/FolderDS.rst

Using SNAP database

This database logging will save the alarm state and all associated attributes every time that the alarm is activated/reset.

You should have configured previously an Snapshoting Database (java/mysql service by Soleil).

Then you have to:

- Set the CreateNewContexts property of PyAlarm to True (it will automatically create a new context on alarm triggering)
- Or create manually a new context in the database using Bensikin.
- Set UseSnap=True to trigger snapshots for all alarms
- Or simply add the SNAP receiver.

Creating a context manually instead of doing it with PyAlarm may allow you to store Tango attributes that do not appear in the formula, thus enabling a sort of alarm-triggered archiving mode.

5.16.5 Triggering Actions from PyAlarm

See basic details on the user guide:

https://github.com/tango-controls/PANIC/blob/documentation/doc/PyAlarmUserGuide.rst#id20

Here you have some more examples:

```
# Send an email (equivalent to just %MAIL:address@mail.com)
%SENDMAIL:ACTION(alarm:command,lab/ct/alarms/SendMail,$DESCRIPTION,$ALARM,
↪address@mail.com)

# Reset another alarm, DONT USE [] TO CONTAIN ARGUMENTS!
%RESET:ACTION(alarm:command,test/pyalarm/logfile/resetalarm,'TEST','$NAME_$DATE_
↪$DESCRIPTION')

# Reload another device
%INITLOG:ACTION(alarm:command,test/pyalarm/logfile/init)

# Write a tango attribute
%WRITE:ACTION(alarm:attribute,sys/tg_test/1/string_scalar,'$NAME_$DATE_$VALUES')

# Execute a command in another tango host
# in this example a FolderDS saves the alarm log
%LOG:ACTION(alarm:command,controls02:10000/test/folder/tmp-folderds/SaveText,'$NAME_
↪$DATE_$MESSAGE.txt','$REPORT')
```

Then declare the AlarmReceivers like:

```
ACTION(alarm:command,mach/dummy/motor/move,int(1),int(10))
ACTION(reset:attribute,mach/dummy/motor/position,int(0))
```

The first field is one of each PyAlarm.MESSAGE_TYPES:

```
ALARM
ACKNOWLEDGED
RECOVERED
REMINDER
AUTORESET
RESET
DISABLED
```

Available keywords (managed by PyAlarm.parse_devices()) in ACTION are:

```
$TAG / $NAME / $ALARM
$DEVICE
$DATE / $DATETIME
$MESSAGE
$VALUES
$REPORT
$DESCRIPTION
```

5.17 PyAlarm Using Events With Taurus

5.17.1 Setting up a PyAlarm getting Tango events from Taurus

We will test events using the CLOCK alarm created in the previous recipe (polling should be enabled, this example uses polling on CLOCK attribute at 10 ms):

> https://github.com/tango-controls/panic/blob/documentation/doc/recipes/CustomAlarms.rst#clock-alarm-triggered-by-time

Then, create a new PyAlarm device and the event-based alarm:

```
import fandango as fn
fn.tango.add_new_device('PyAlarm/events','PyAlarm','test/pyalarm/events')

from panic import AlarmAPI
alarms = AlarmAPI()
alarms.add(device='test/pyalarm/events',tag='EVENTS',formula='test/pyalarm/clock/clock
↪')
```

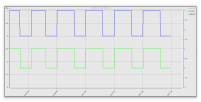
Start your device server using Astor, fandango or manually

```
import fandango as fn
fn.Astor('test/pyalarm/events').start_servers(host='your_hostname')
```

Then, configure the device properties to read attributes using Taurus and react as fast as possible Taurus will take care of subscribing to events and update cached values.

```
dtest = alarms.devices['test/pyalarm/events']
dtest.config['UseTaurus'] = True
dtest.config['AutoReset'] = 0.05
dtest.config['Enabled'] = 10
dtest.config['AlarmThreshold'] = 1
dtest.config['PollingPeriod'] = 0.05
alarms.put_db_properties(dtest.name,dtest.config)
dtest.init()
```

This is the result you can expect when showing both alarm attributes (test/pyalarm/clock/clock and test/pyalarm/events/events) in a taurustrend:

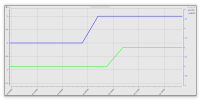


5.17.2 Is this approach really Event-Based?

Yes, but not asynchronously. PyAlarm will use Taurus to catch Tango Events and buffer them; but alarms are still triggered by the internal polling thread of PyAlarm. It means that the PyAlarm.PollingPeriod property effectively filters how often incoming events are processed.

But, delegating event collection to Taurus allows to not execute read_attribute in the polling thread; allowing to very small PollingPeriod values (10-20 ms)

As seen in this picture, it allows to have a very fast reaction from the Alarm attributes respect to the trigger:



This approach, however, is costly in terms of cpu usage if using polling periods below 100 ms. A pure-asynchronous event implementation of the PyAlarm is still pending.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search