TANGO

ALBA

Tango Device Design

Highlights and Best Practices

Sergi Rubio Manrique, 40th Tango Community Meeting, 2026

Index

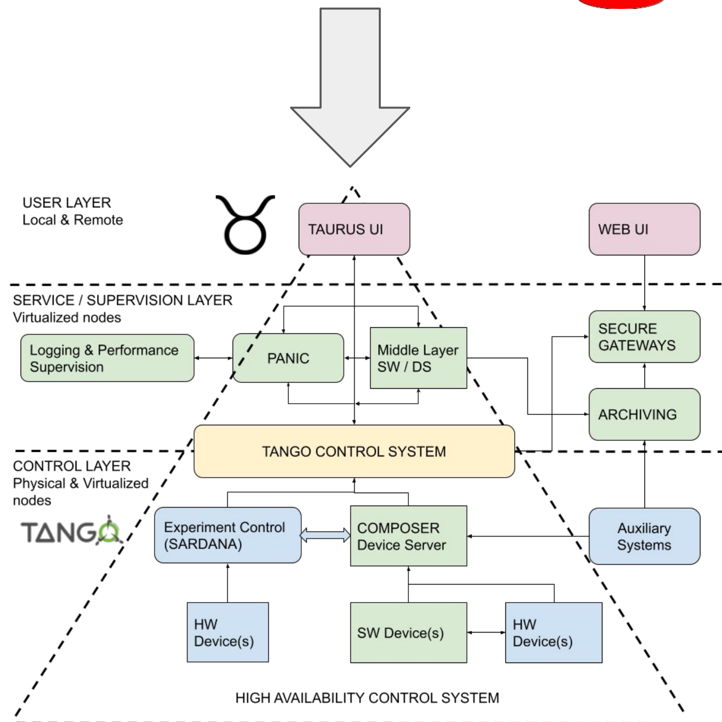
- What is a Tango Class/Device/Server?
- Device Server Architecture
- Commands, Attributes, States and Properties
- Polling, Serialization and Hooks
- Tango Device Patterns : Synchronous / Asynchronous / Threaded / Polled
- Scaling a Tango Control System
- Hints Summary

What is Tango?

We used to describe Tango as a **software bus** to communicate HW with UI's and services

But Tango is also a rich framework to develop **multi-layered control systems**, in many different ways.

Nevertheless, at the heart of Tango Control Systems and Services there is the same piece: a **Tango Device Class**



What is a Tango Device/Class/Server?

ALBA

It is a common believe that Servers have just 1 Class & 1 Device. So it is important to remark:

Tango Class* != Tango Device != Tango Server*

Tango Class => the code (c++/python/java) we write to control a given hardware/software component

Tango Device => an instantiated object (of a given class) within the **Tango Control System**

Tango Server => a process importing N classes and exporting X devices of such classes

Tango Database => the central point storing all these relationships and its configurations

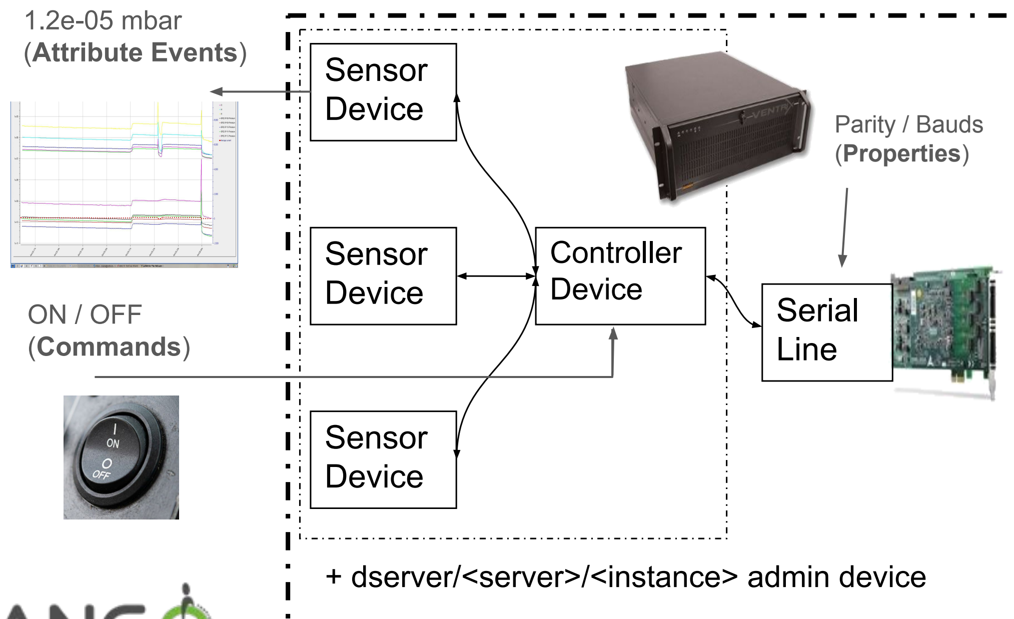
So, a server does not have to export only 1 device of 1 class.

** I deliberately omited "Device" on Tango Device Class and Server; to avoid mixing terms*

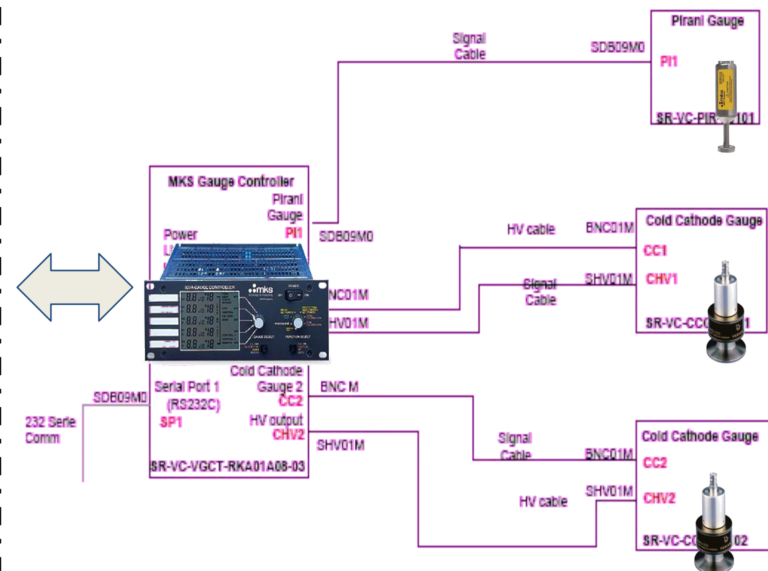
Tango Device Server Architecture

- A Tango Server with **3 classes**: SerialLine, Controller and Sensor
- **SerialLine** models a serial port, **Controller** class models the HW protocol; **Sensor** pushes values to UI
- Devices interact using **DeviceProxies**, **Commands** and **Attributes**
- Configured using **Properties ;** settings stored for each device or class in the **Tango Database**
- Tango Server have an internal **Admin device**, which controls Tango behavior (events/polling/buffers)

Software (1 Server / 3 Classes / 5+1 Devices)

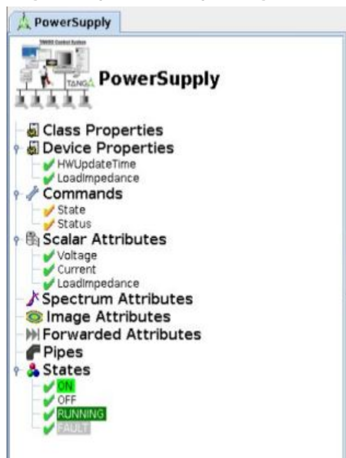


Hardware (1 Controller / Multiple HW equipments)



Commands, Attributes, States & Properties

from POGO, the Tango code generator
it gives you everything HL API doesn't!!

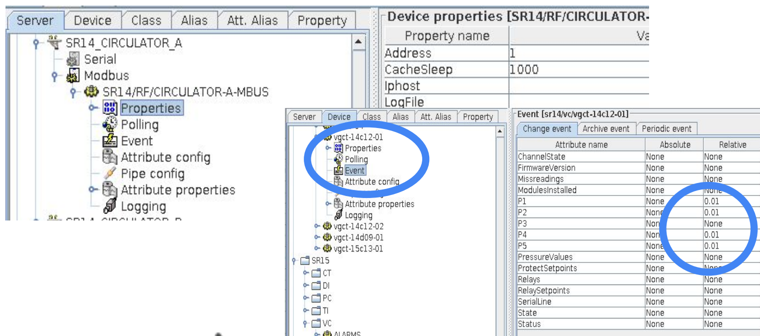


Commands : **Actions**, *A **command equals to** using a physical button in the hardware or using **a function** in your code.*

Attributes : Live Read or Read/Write values; pushing **events** or **alarms**. *An **attribute equals to** the values shown in an HW display, or the **variables** of your code.*

State/Status : attributes linked to the HW status, the result of a command **Open() => OPEN**) or the status of an attribute (**ALARM** if attribute > **MaxAlarm**)

from JIVE, the Tango Database navigator



Properties : Persistent settings stored in the database.

Loaded at startup or at **Init()** execution! (**like an /etc or .cfg** file)

They can exist at **Attribute, Device, Class or System (free) level.**

Attribute properties and configuration will be **memorized values**, **labels, alarm config, event filters, ...**

Polling, Serialization, Events, Alarms, ...

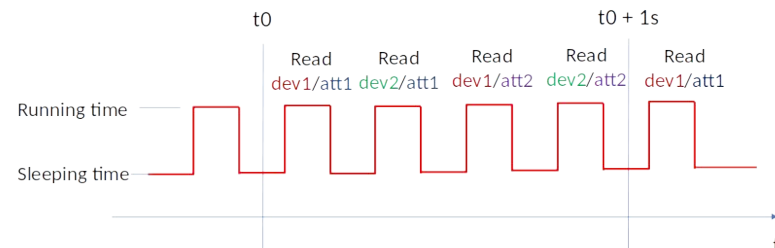
Serialization mode : manage concurrency between client/internal calls, at device/server level. Including asynchronous/non-serialized.

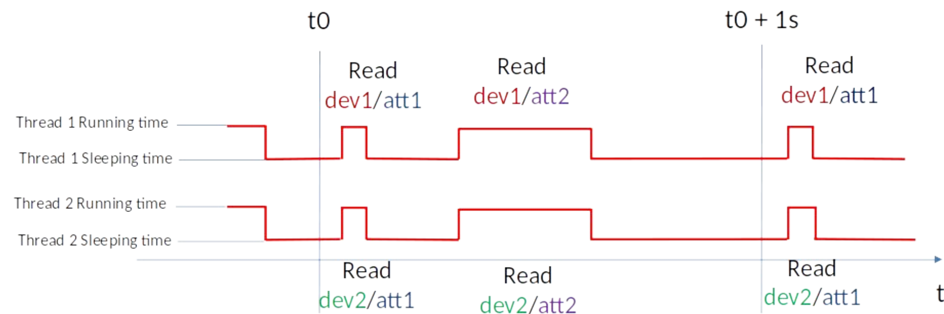
Polling mode, choose how often attributes will be read, and whether reads will be group or split.

Also **choose the number of threads**, the **size of the internal buffers** caching values, the **pushing of events** from the polled values, etc.

Polling implies **automatic Events/Alarms/State** based on Attribute configuration. You can rely on it or replace it with your own **push_<event> and set_state()** calls.

Most of these behaviours are code-less configured in the **Tango Database** and managed by the internal **Admin** device (so you can still restart/tune a hung device).





Special Device Methods: (beyond the HL API)

Must-write methods:

- **init_device** : once the object is already created, load properties from database, connect and initialize the device.
- **read_<attribute> / write_<attribute>**: methods to acquire/apply, either using hardware or internal caches.
- **<command>** : a method for each command, no overloading!, must match the specified input/ouput types for the class

Optional methods:

- **dev_state/dev_status** : by default calculates device state based on qualities, when called from client/polling thread
- **dyn_attr** : depending on the initial config, create new attributes if needed (e.g. Channels of a Data Acquisition Card)
- **always_executed_hook** : executed before each command / read / write call ; when reading multiple attributes is called only once.
- **read_attr_hardware** : executed before each read_attribute call ; when reading multiple attributes is called only once
- **is_<attribute>_allowed** : called for each attribute, it can be used to enable/disable some attributes depending on state (the StateMachine matrix in Pogo).
- **push_<eventType>_event** : push from your code events on specific conditions (change, archive, data ready, alarm, ...).
- **delete_device** : Ideally, creator (init_device) and destructor (delete_device) methods should be written in a way that **Init() method of the device can be safely called to return the device to a clean state**; or to reload configuration (properties).

Updating the Device State

ALBA

- Enabling Polling for your State attribute you will immediately get **State events** for all your clients.
- Some States should typically be linked to the result of specific "long" commands (Open() -> OPEN, Move() -> MOVING, Close() -> CLOSE, Init() -> INIT) . It is not automatic, but it is a good way of giving asynchronous feedback.
- Or the configuration of **attribute alarms** (changing device state to **ALARM** when attribute value > MaxAlarm configuration). This behavior is automatic by default.
- But, you can (and should) use **set_state()** and **set_status()** to provide verbose information on any error occurring in the device (e.g. FAULT State and Error code information on Status).
- For custom states, **always_executed_hook** will be called for every call, including State(), so it may be the right way to use **set_state()** and **set_status()**. But do not make it cpu heavy!! (**use cached data**)

Common Design Patterns

ALBA

There are as many ways as developers to write a Tango Device Class. But this would be the most typical patterns found in our control systems:

- Classical pattern : Every command / attribute is a hardware call.
- Grouped pattern : All hardware communications are done in a single method
- Callback/Threaded: Hardware values are updated out of Tango serialization
- Devices as Clients : The device processes data obtained by other devices

We may have polled/non-polled, synchronous/asynchronous, or custom event variants.

Implementing all logic in tango methods, or write a separate library is **developer's freedom of choice.**

Pattern 1 : Every method is a HW call

The classical pattern, just works for network hardware and few attributes:

- Every Command() or read_attribute() method **access HW and returns a value**
    - So, when not polled, every client read/command triggers a HW access!!
- Methods are serialized by the Tango Monitor, so multiple **clients will lock/wait on each other**!!
    - It is much more optimized by **configuring polling and using events**!

When using polling, client Attribute reads simply return the last polled value:

- Hardware refresh and events(change/archive/alarm) are managed by the Tango Polling Thread(s).
- Each command/attribute can have a different refresh period. Each device can have its own thread.
- **BUT!** This setup is prone to Timeout and OutOfSync errors, it only works if:
    - **HW acces time << (refresh_period / n_attributes)**

Pattern 2 : Grouped attribute readings

- Hardware communications are grouped in a single command or using read_attr_hardware.
  - optimal when hardware allows to retrieve all data in a single call
  - read_<attribute> simply returns last value from cache and pushes events

- **Since Tango 9, polling thread uses read_attributes() to group readings:**
  - When using read_attributes(), read_attr_hardware() is called with attr IDs
  - Use different polling periods to group attribute readings in bunches.
  - Typically, most of attributes do not need continuous refresh

Pattern 3 : Internal thread / callback

ALBA

- Callback/Threaded: Hardware values are updated by an internal thread or from a callback called by the hardware driver and then cached/pushed.
  - Ideal for fast hardware and drivers based on callbacks or streamed data
  - read_<attribute> methods can just read the cache and return the value
  - Use set_attribute_value_date_quality() to pass HW timestamp to clients
  - Events can be pushed from thread/callback in a not serialized method, use **EnsureOmniThread // omni_thread::ensure_self** to avoid issues.
- Can be combined with asynchronous / no-serialization mode. See how to deal with synchronous/asynchronous at :
    - https://tango-controls.readthedocs.io/en/latest/How-To/development/cpp/how-to-cpp-client-programmers-guide.html
- If HW is asynchronous, you can add a low level Tango Device to convert it to asynchronous (using events and command_inout_asynch + callback)

Pattern 4 : Device as Client

Tango Devices can create proxies to another Tango Devices, stored as **SubDevices** in the TangoDB.

- **Controller** Device : A device using another HW-based device (typically a SerialLine, Modbus, DAQ, ...) to perform a data acquisition process. Calls can be asynchronous, long queue may use dynamic attributes and data ready events.
- **Façade** Device : devices that collect data (events) from other devices in the control system and process their attributes to calculate new attribute values (e.g. PyAttributeProcessor)
- **Composer** Device : devices that summarize the information for an specific group of devices (belonging to the same class or system) and manages them as a whole (like a **Tango Group**).

These patterns allow to easily organize a Tango Control System in different logic layers, with constraints:

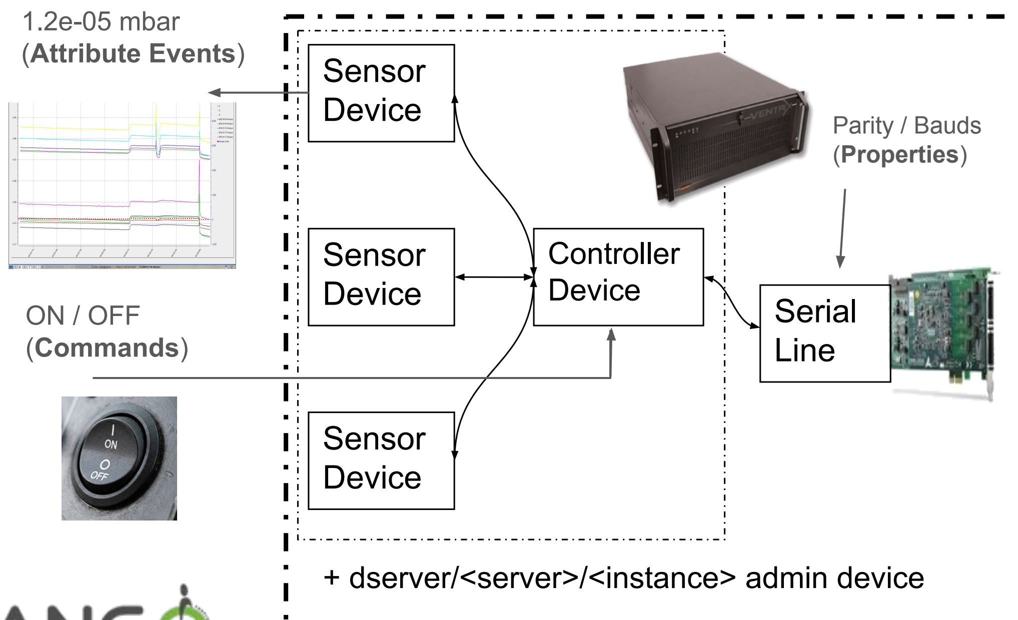
- If the number of SubDevices or connected attributes is high, use asynchronous methods or events
- When using callbacks, **callback execution time should be minimal** (avoid growing queues!)
- **Error management** must be taken into account, avoid black-box layers
- Using composers, you can summarize large attribute lists into array attributes

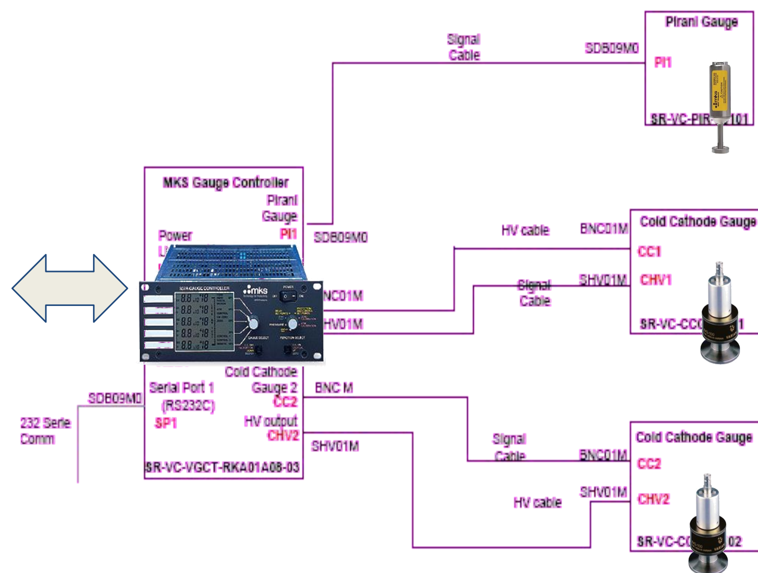
Tango Device Server Architecture (2)

- Communications between controller and sensors can be all based on events.
- **User interfaces will be updated by events** steadily with **no timeout.**
- Reading of hardware parameters can be grouped in **fast polling (sensor value) and slow polling (configuration)**
- Or the **serial line can be used asynchronously** using callbacks for returned values.
- Server load should be keep low enough (40% free time) so **commands can be executed directly into hardware.**

Software (1 Server / 3 Classes / 5+1 Devices)

Hardware (1 Controller / Multiple HW equipments)





Scaling Tango Systems

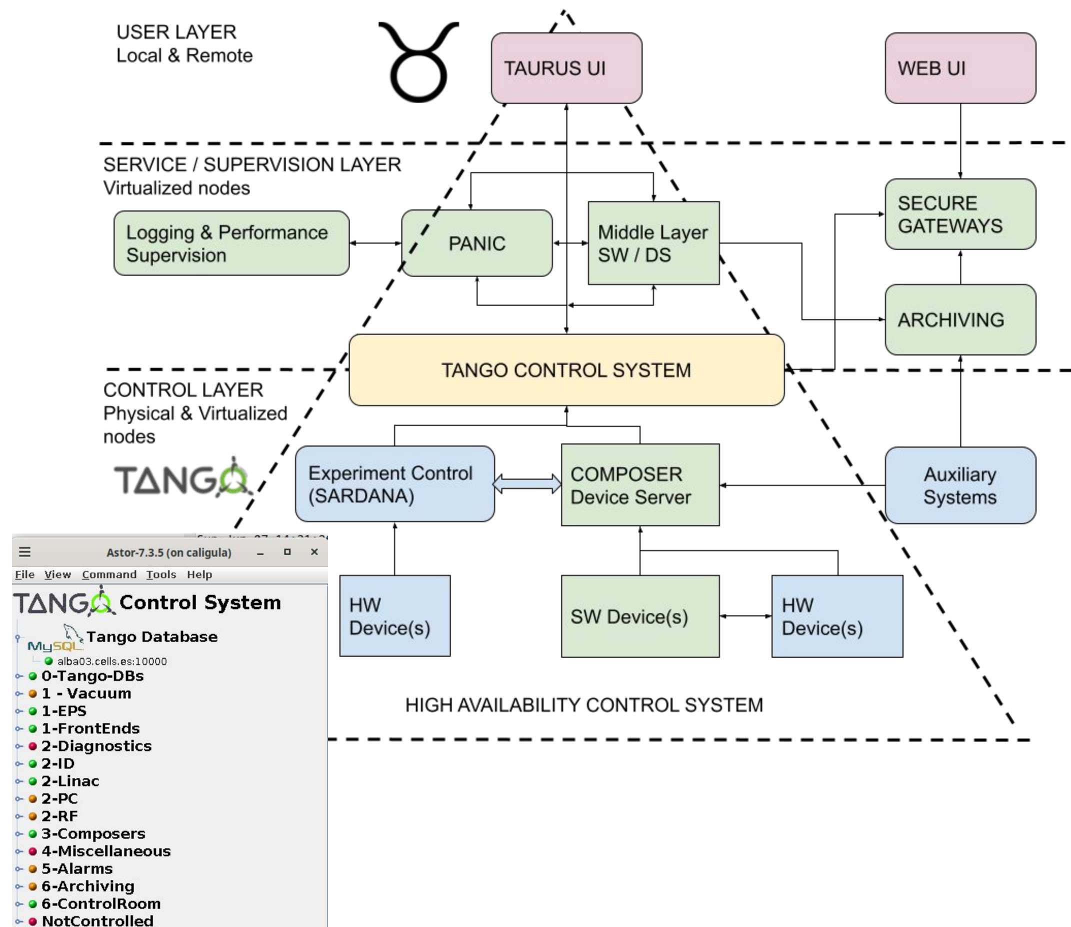
Split into dedicated devices for hardware and devices for system logic

Be asynchronous! Avoid timeout errors to propagate w/out diagnose, allow commands to propagate fast

Use Composers and arrays to **summarize** each layer

Each layer should **facilitate diagnose** on the layers below!

Use Astor/Starter **runlevels** to restart the system in the proper dependency order across hosts (or multivisor).



Device Profiling

Have tools in place to help you diagnose/detect cpu/memory problems in device servers. Using Tango Attributes will allow you to archive/alarm later!!

How to perform and stress test:

- Read all your attributes at **1 second period**. Monitor cpu & memory

- **Tune** polling periods fo find an stable refresh. Check polling status in Astor/Jive.

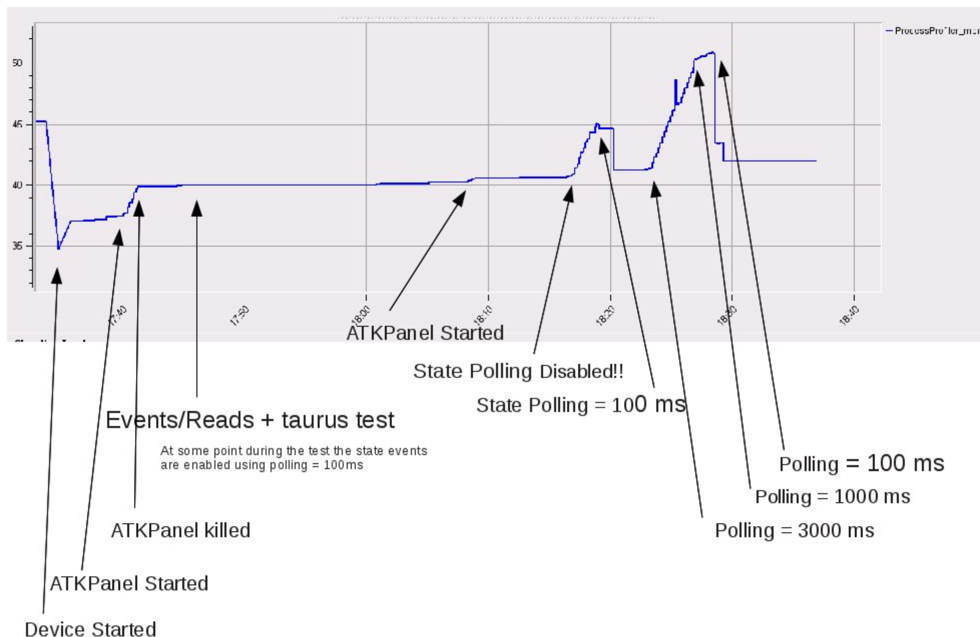
- Check if you can still **execute commands!**

- **Hint:** total reading period < 60% polling time

- Monitor stats and exceptions to detect **memleak, timeout or high cpu**

- **Things may and will fail!** But should **not hang!** Catch, log, notify and provide means to **diagnose**.

Process Profiler device, memory leak tests



Hints Summary

ALBA

Timeouts! => Avoid reading synchronously on slow hardware, read on the background (subdevice/thread/callback/...) and store in cache.

Slow hardware requires _asynch calls and high set_timeout_millis

Memleaks => limit python/java cpu usage!, ensure callbacks return fast (so queues don't grow)

High cpu => all hooks/callbacks should have only fast code, use caches for everything

Segfaults? => ensure you use EnsureOmniThread/omni_thread::ensure_self and **delete_device** cleans up threads

Too many processes => group multiple devices on each server, optimize resources

Too many attributes? => use events, group attribute reading/polling, use arrays for similar data

Too many events received? => replace callback by a buffer, then process using a polled command; use facade/composers to group/process the data in middle layer devices

Problems to test UI? => Move all the logic from UI's to easier to test high-level devices, split behaviors in logic layers

Last but not least, some Design Principles

- see RFC's (https://gitlab.com/tango-controls/rfc) and device-server-guidelines.html
  - A Tango Class should be **Reusable, Configurable, Extensible and Shareable**; use generic method names and use "*globish english*" for the interface and documentation.
  - Avoid non-alphanumeric characters, **only "_-." are allowed, and not in all cases!;** see the **RFC** for a full reference. For readability, use CamelCase for Servers, Commands, Properties and Classes.
  - Provide default units and alarm/event configuration when designing your device
  - Use standard logging_stream methods (**debug, info, warning, error**)
- and my personal advice:
  - Avoid non-standard types or encoded types, use them only when really justified.
  - At **init_device()**, update property values to TangoDB (so defaults become visible to everyone)
  - Provide cpu usage and memory usage as attributes, use the **new device_info** for SW versions!!
  - I now it's hard, but instead of reinventing the wheel, **collaborate and improve the existing classes!!**

Questions?

https://www.tango-controls.org,

https://tango-controls.readthedocs.io

https://pytango.readthedocs.io

https://www.youtube.com/@tango-controls

https://tango-controls.readthedocs.io/en/latest/Tutorials/ds-guideline/device-server-guidelines.html

https://developer.skao.int/projects/ska-tango-base/en/stable/tutorials/create-tango-device.html

Thanks to SKA, MaxIV, Soleil, ESRF, Elettra, Solaris, S2I, Observatory Sciences, Anton, Benjamin, the 3 Thomas (our Tango Muskeeters) and everyone that collaborated in the new Tango documentation

(sorry if I miss someone)