Skippy Device Server
sources version 1.4.2-alpha

Sergi Blanch-Torné
Controls Software Engineer - Alba Synchrotron
sblanch@cells.es

December 22, 2017

Abstract

Following the wikipedia's definition: the SCPI "*defines a standard for syntax and commands to use in controlling programmable test and measurement devices*". Many scientific instrumentation uses this schema for the configuration and operation of the instrument's purpose.

Since 2007, the Alba Synchrotron has developed a set of tango Device Servers to access instruments that provides this protocol. Development experience has guided us until this Skippy device server. In parallel, due to our own needs over *in-house* development of instrumentation, it has been developed also a python module that provides some of our instruments this interface to the upper layers, called python-scpilib.

The original development of a tango Device Server provides only access from this Distributed Control System (DCS), but it is also good to provide a simple python access in between the instrument and the Control System.

Then with this two developments together, the scpilib and the skippy, the circle has been closed supporting all the points of view over this protocol.

Contents

1 Introduction

The Alba Synchrotron has collected some experience controlling instruments that supports the SCPI protocol from one or many of the interfaces. Previous to this skippy Device Server we have implemented the *PyVisaInstrWrapper*, that uses lower level tango devices (like it is the *PyVisa* or the *Serial*, to connect to instrument like *oscilloscopes*, *radio frequency generators*, *signal and function generators* and even *spectrum analyzers*.

The *Wrapper* was in fact a generalization of an even previous *PyScope* that was confined to be one of the Device Classes supported. Even before that, there is the first of our instrument control, made in `C++`, that was only controlling one series of Tektronix scopes. The name *TekDPO7000* says everything.

2 Design

The main user case of this project is to access the instruments from `tango` control system. The design starts from this very above view. A distributed system agent (aka `device`) in charge of promoting information of a single instrument to the control system has an state machine, common for all kind of instrument below. It normalizes in a common upper view of what would be down there.

But this `device` does not have the functionalities implemented within itself, the project provides a `python` module with a main class (called `Skippy` as one can expect) that in the constructor the `device` properties are setup to then have a pure python interface to the instrument. This has been made to allow future uses without a `tango device` (but having a dependency on `PyTango`) or, and the important case, to migrate the `device server` code to the High level server API.

2.1 Machine state

When a device of an instance of the `Skippy` device server is launched, the first state it has is `INIT`[1], on the left side of the picture 1. When all the internal elements are well initialised, then it changes its state to `OFF`. At this state, the device has an internal object with the responsibility of the communications with an instrument (but no communication has been started).

Then, if the configuration properties do not say the contrary (details about them will be extended in section 2.2.1), the device will try to establish communication with the instrument. This is reported with the state change to `STANDBY`. The instrument will receive a request to identify itself, and the answer will allow the device to know which instructions set is necessary to talk further with the instrument.

Again, and still the flow is controlled by the configuration properties, the device create the specific set of (dynamic) attributes for the identified instrument. Reporting this succeed with a state change to `ON`. If there are attributes to have their value monitored and pushing events, and for the last time, the configuration properties will allow or inhibit the next state change, the *Start()* process will be launched to have an specific thread in charge of this monitoring. This state corresponds with `RUNNING` state, and more details will be explained in 3.5.

The figure 1 show other state-change transition, the edges, with the command name that produces the change. In any case, if something goes wrong, its state will decay to `FAULT`. To recover from fault, the *Init()* command should be called, apart from cases where the device itself can recover the situation to then do the necessary steps to recover normal operation.

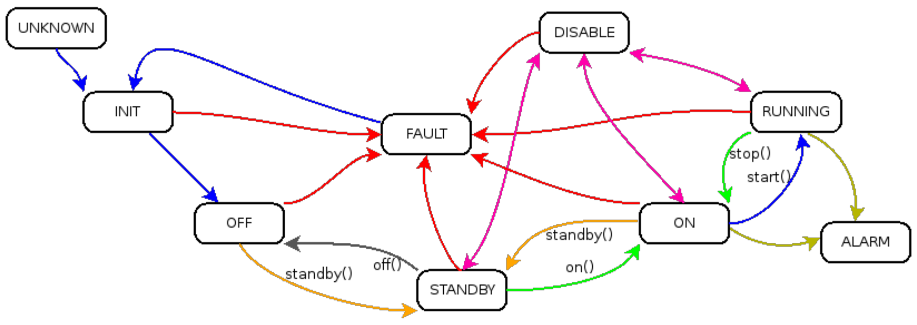

Figure 1: State machine diagram of the Device in the Skippy Device Server

There are two more states that the `device` may report sporadically. When the queries to the instrument takes longer than expected, then the state will be reported as `ALARM` until the access to the instrument goes back to a normal timing. When this happens the `device` would try to reduce the load of the instrument by reducing the frequency of queries send to the instrument or the number of queries sent in parallel.

The last of the states is the `DISABLE`. One of the internal features of this `device` is to check periodically if the instrument is still available. This is specially important to report a communications fault when the instrument has been shutdown, as well as to recover a communications glitch. `TODO:` "*More details about that in section 3.6 about the* `WatchDog`".

[1]There is even a previous state, when it has not been initialised, called `UNKNOWN`. But immediately in the `tango device` construction it is set to `INIT`.

2.2 Device as an Object

As `Tango` is an *Object oriented* (OO paradigm) DCS, each of the agents have a set of construction parameters, what is called `Properties` (section 2.2.1. Once the *object/agent* is build in the system, it offers a set of `Attribute` (section 2.2.3) and `Commands` (the *methods* in the OO paradigm, section 2.2.2).

2.2.1 Device Properties

- `Instrument`: mandatory property to configure the instrument that will be introduced to the distributed system via the device.
  - If the content string is a host name, use sockets to communicate with the instrument
  - If the content string is a device name if the class of this subdevice is `PyVisa` or even `[Py]Serial`, it will be used as a bridge to the instrument.
  - If the content string is a serial line file descriptor (for example a `/dev/ttyS`*n* or a `/dev/ttyr`*NN*), it will be the way the instrument will be accessed.
  - else, decay to fault state with the appropriate status message
- `Port`: optional property to specify a port, in the socket communication, when it should be different than the default 5025.
- `SerialBaudrate`, `SerialBytesize`, `SerialParity`, `SerialStopbits`, `SerialTimeout`, `SerialXonXoff`: optional properties to define the serial line communications when the instrument is accessed by a serial line in the machine where it runs.
- `NumChannels`: optional property to define, in case the instrument has channels, the number of them available.
- `NumFunctions`: optional property to define, in case the instrument has functions, the number of them available.
- `NumMultiple`: more generic than the previous two, and has been made because of the limitations they had. It is a generalisation for channels and functions. It must be composed by a list of pairs with the *scpiPrefix* (equivalent to the 'ch' or 'fn') followed by the number of them that shall be build. `TODO:` "*Further information about how to setup correctly those dynamic attributes. Add an example*"
- `MonitoredAttributes`: When the device is in RUNNING state, the attributes listed here will be monitored (having events) with a period said in the attribute TimeStampsThreashold (or different if specified with a : separator after the attrName). `TODO:` "*reference to an example*"
- `AutoStandby`: When device startup, try an standby() to connect to the instrument authomatically. Default True.
- `AutoOn`: When device startup, try an on() begin communication with the instrument.
  Default True.
- `AutoStart`: When device startup, try an Start() to launch the necessary monitor threads (if `MonitoredAttributes` is configured), authomatically.
  Default True.
- `TxTerminator`: by default the instruments uses '`\n`' as a Transmission terminator, but there are instruments that requires to use '`\r`', or both in a certain order like '`\r\n`'.

2.2.2 Device Commands

- `IDN()`: Request identification to the instrument.
- `Off()`: Release the communication with the instrument.
- `Standby()`: Open the communications with the instrument and do the identification and attribute builder, but do not allow yet any other query.
- `On()`: Stablish communication with the instrument.
- `Start()`: Start an active monitoring.

- `Stop()`: Stop the active monitoring.
- `AddMonitoring(AttrName)`: Add an attribute to the list of monitored attributes
- `RemoveMonitoring(AttrName)`: Remove an attribute from the list of monitored attributes
- `SetMonitoringPeriod([AttrName,AttrPeriod])`: From the list of already monitored attributes, stablish (or change) the period that it is checked. With this command the monitored attribute will become one of the splitted thread monitor, even if the period in the same thant the normal periodic. To force to place in this normal monitor list, use period value 0.
- `GetMonitoringPeriod(AttrName)`: Get the period that is checked an attribute monitored. `NaN` if it is not monitored.
- *`Exec():`* Expert attribute to look inside the device during execution.
- *`CMD():`* Expert command for a direct send of a SCPI command and read the answer.
- *`CMDfloat():`* Expert command for a direct send of a SCPI command and read the answer converted to a float list.
- `TODO:` "*`DumpAttr([file,time]):` Once received, dump the raw data readed from the read of an attribute during the specified seconds*"
- `TODO:` "`DynamicCommands`"

2.2.3 Device Attributes

- `idn`: read attribute with the identification of the instrument linked.
- `QueryWindow`: Expert attribute to configure the number of request sent in parallel to the instrument. Bigger queries will be splitted in subqueries o of this size.
- `TimeStampsThreshold`: This value sets the threshold time to use a cached value or hardware read it.

Other attributes can be built using a Builder pattern. In fact, when the device starts and the instrument is identified, with the information provided in this identification, the device can find how to build the instructions set.

`TODO:` "*List all the keywords in the `Attribute(...)` definition.*"

2.2.3.1 Attribute definition examples

An example of an attribute definition:

```
Attribute('State',
          {'type':PyTango.CmdArgType.DevBoolean,
           'dim':[0],
           'readCmd':lambda ch,num:":%s%d:DISPlay?"%(ch,num),
           'writeCmd':lambda ch,num:(lambda value:":%s%d:DISPlay␣%s"%(ch,num,value)),
           'channels':True, 'functions':True})
```

This example, would correspond with an *oscilloscope* with 4 channels and 4 functions. The `device` will setup 8 `READ_WRITE` boolean attributes, called with the pattern: `State{Ch,Fn}[1..4]`. Relevant to mention is the `readCmd` and the `writeCmd` uses those `lambdas` because part of the command to be send to the instrument will be build in different moments. In the *read* case, and because this is an attribute description to build more than one, the `lambda` function is set to build the final command for each of the channels and functions. A bit more complicated is the *write* case, where there are two nested `lambdas`, one used in the attribute build and the other used when write the attribute because is when the value is known.

Another example of an attribute definition:

```
Attribute('Frequency',
          {'type':PyTango.CmdArgType.DevDouble,
           'dim':[0],
           'readCmd':":FREQ?",
           'writeCmd':lambda value:":FREQ␣%s"%(str(value)),
           'rampeable':True})
```

That example, would correspond with a *Radio Frequency* generator, will setup 3 READ_WRITE double attributes, called Frequency, FrequencyStep, FrequencyStepSpeed. This describes a behabiour where, when a *frequency* is set, a thread will be launched to each *StepSpeed* seconds, the current value will be increased/decreased by the *Step* in the direction of the setpoint.

Here an example of an spectrum attribute definition:

```
Attribute('Waveform',
          {'type':PyTango.CmdArgType.DevDouble,
           'dim':[1,40000000],
           'readCmd':lambda ch,num:":WAVeform:SOURce␣%s%d;:WAVeform:DATA?"%(ch,num),
           'channels':True, 'functions':True})
```

With this attribute is broken a backward compatibility with *PyVisaInstrWrapper* because there those spectrum attributes did not follow the naming of name. There is a naming convention in attribute definitions with channels and functions, to the name given in the definition is concatenated at the end two characters (Ch or Fn) followed by the number of the channel.

There is an alternative that allow to stablish the naming with a bigger flexibility (but not full-free).

```
Attribute('IO',
          {'type': PyTango.CmdArgType.DevString,
           'dim': [0],
           'readCmd': lambda mult, num: "%s%.2d:VALU?" % (mult, num),
           'writeCmd': lambda mult, num: (lambda value: "%s%.2d:VALU␣%s"
                                          % (mult, num, value)),
           'multiple': {'scpiPrefix': 'IOPOrt', 'attrSuffix': 'Port'}
           })
```

This attribute description will generate some attributes under the name patter '*IOPort*NN', and there will be as many as specified in the NumMultiple attribute where one of the pairs shall have as first element 'IOPOrt'.

2.3 Class Diagram

In figure 2 can be found a UML draw with the class diagram.

3 Device features

3.1 Dynamic commands

This feature is not yet implemented

3.2 Dynamic attributes

Device is capable to dynamically build attributes based on the instrument identification (described in the the particular instruction set).

- Supported Scalar and Spectrum (1 dimension arrays) attribute definition. Image (2 dimension arrays) are not currently developed, neither in the schedule because they are not needed by any of the current supported instruments. Although the device is prepared to sopport them if need be. (Perhaps the frequencies spectra in SpectrumAnalyser).

- The dynamic attributes build after the instrument identification can be Memorized if this has been configured this way in the file with the instruction set.

3.3 Multiple requests per query

Multiple request of data to the instrument: As figure 3 describes, when the device receives a request for reading to the instrument it can manage to cut those requests in subsets to avoid stress in the instrument. The communication itself supports a direct socket connection to the instrument and the use of the *PyVisa* device as a bridge in the communication.

3.4 Write attributes ramping

Write process to some attributes would require a smooth ramp. As an example, to change the frequency in the *RFGenerator*, the change would not be set directly (or the timing system will suffer). Then in the build process of an specific attribute with a certain flag of "movability" (name to be determined) two extra

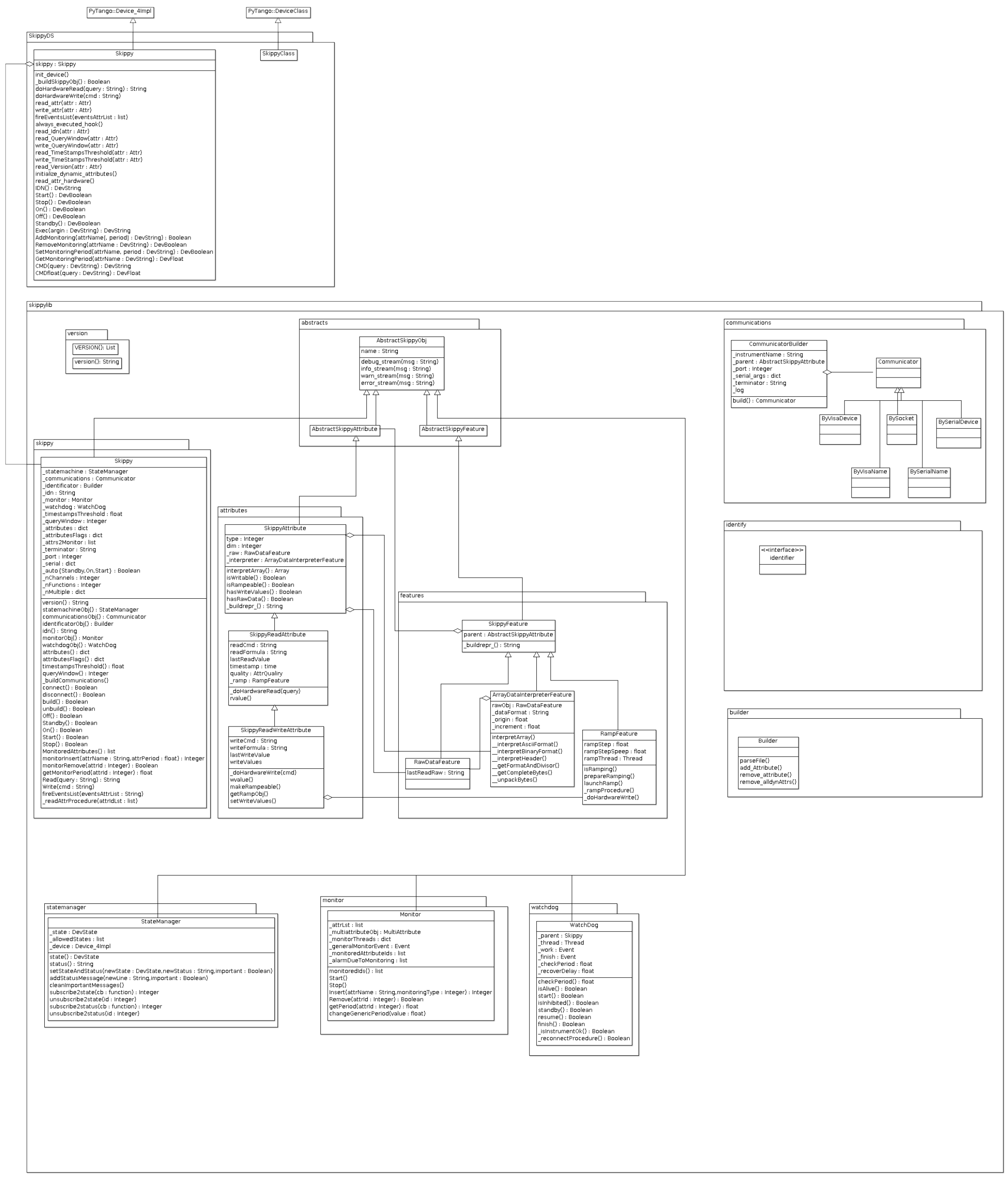

Figure 2: Class diagram of the Skippy device.

attributes, internals in the device, must be built also: a `Step` and a `StepSpeed`. On the write operation a thread is launched and every "StepSpeed" seconds change the value by the content in "Step" in the direction to the final setpoint. This was already implemented in *PyRfSignalGenerator* Class. `FIXME:` "*This will require a refactoring in order to encapsulate the functionality in a* responsible object".

3.5 Monitoring values

Monitor attributes. Having a property with a list of attributes, if they exist in the instrument definition, configure them in events and do a separated thread polling that will emit events on this attributes. This feature is different that setting up a "polling" from tango to the attribute because this fails to use the multiple reading.

- In the definition of the attribute to be monitored, the *MonitoredAttributes* property, using an specific notation (a ':' followed by a number of seconds) can set up an specific monitoring period

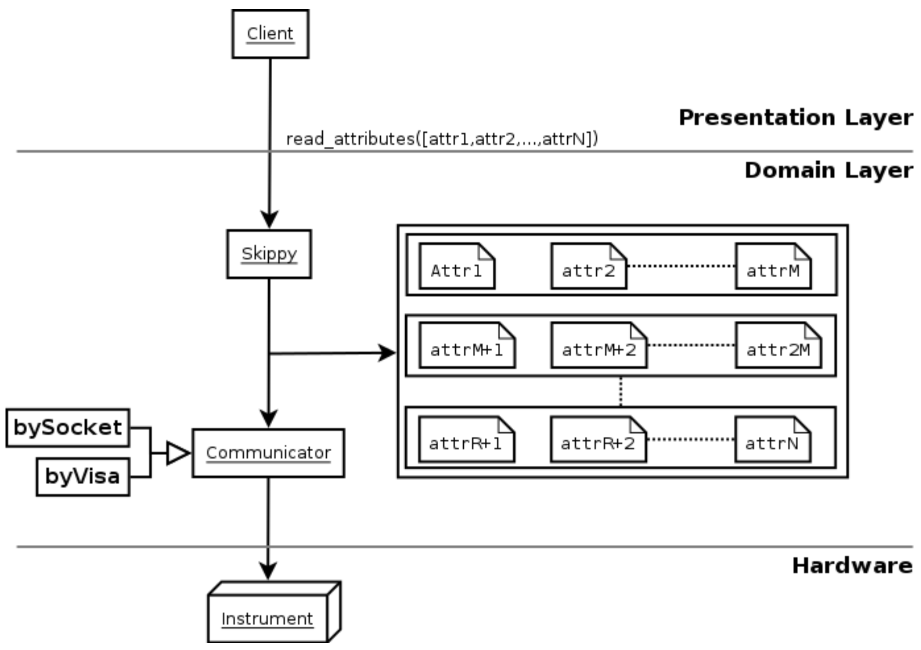

Figure 3: multiple request of readings to the instrument.

for it in particular.

  – When the monitoring is by the specific period, different attributes with the same period, should be read by the same monitor thread.

- The use of the commands (*add* and *remove*) to setup this feature is recommended. Together with a set of commands to setup specific monitoring periods.
- When an attribute is monitored, a `read_attr()` returns the cached value (no hardware read).

`TODO:` "*Once this monitoring feature is available, other attributes not monitored that have reached some reading frequency, would be included in the monitoring loop (not emitting events) in order to reduce the load of the readings. The frequency reading should be monitored to notice when this has reduced over another threshold to avoid unnecessary readings.*"
`FIXME:` "*When many attributes are being read, and not by* `read_attributes([])`*, prioritise. (This happens, for example, when opens the device with* Atkpanel*)* "

3.6 Watchdog

`TODO:` "*Explain*"

4 Testing

4.1 Scripts

`TODO:` "*describe the* `Testing` *directory files and usage.*"

4.2 Known bugs

- `FIXME:` "*When all the monitored attributes are removed, the list ends inconsistent and dirty when newer attributes are added to this list.*"
- `FIXME:` "*The timestamp of an attribute come from when it was read from the hardware. But in case of an* `INVALID` *value, it will point forever to when it has decay: this produces issues when a plot show this and other attributes with different time scale.*"
- `FIXME:` "*The an instrument is powered off, if there is no attribute monitor the device will not see it until access the hardware. That is, request the* state *before write will not update the information.*"