

Skippy Device Server

Sergi Blanch-Torné

Controls Software Engineer - Alba Synchrotron

sblanch@cells.es

November 8, 2013

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 of its purpose.

Alba has used an specific Device Server, under the name *PyVisaInstrWrapper*, to communicate with instruments like scopes, radio frequency generators, arbitrary signal generators and spectrum analyser. This device was an extension from the original PyScope, becoming it one of the internal device classes. Many improvements has been introduced to this Device Server, but nowadays it is showing the limit of this design. A complete refactoring is needed.

1 Design

The main restriction that the PyVisaInstrWrapper shows is the dependency on the PyVisa subdevice to manage the communications with the instrument. Also the growing number of query commands to the instrument has reach the limit of the basic design.

1.1 Machine state

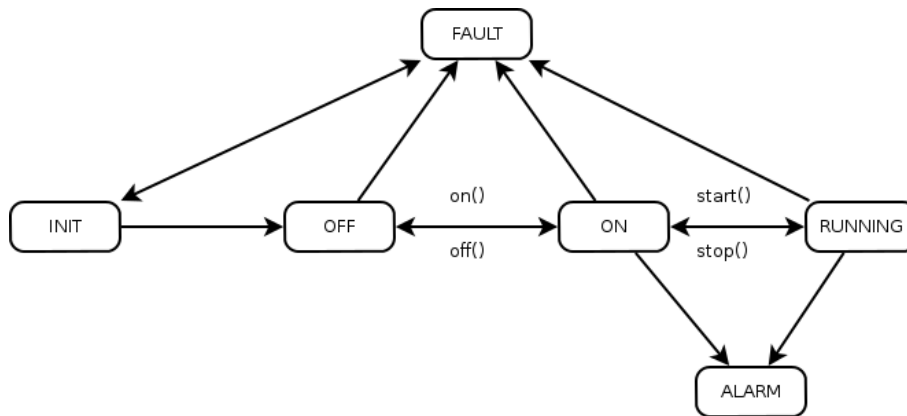


Figure 1: State machine diagram of the Device in the Skippy Device Server

TODO: “*explain the figure 1*”

1.2 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 and the class of it is a PyVisa, use it as the bridge to the instrument.
 - 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.
- 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.
- MonitoredAttributes: When the device is in RUNNING state, the attributes listed here will be monitored (having events) with a period said in the attribute TimeStampsThreshold (or different if specified with a : separator after the attrName).
- AutoOn: When device startup, try an on() to connect to the instrument automatically. Default True.
- AutoStart: When device startup, try an Start() to monitor attributes, if MonitoredAttributes is configured, automatically. Default True.

1.3 Device Commands

- IDN(): Request identification to the instrument.
- Off(): Release the communication with the instrument.
- On(): Establish communication with the instrument.
- Start(): Start an active monitoring.
- Stop(): Stop the active monitoring.
- Exec(): Expert attribute to look inside the device during execution.

1.4 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 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.

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, in an oscilloscope with 4 channels and 4 functions, will setup 8 READ.WRITE boolean attributes, called with the pattern: `State{Ch,Fn}[1..4]`. Point to comment is the `readCmd` and the `writeCmd` and the use of `lambdas`. 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.

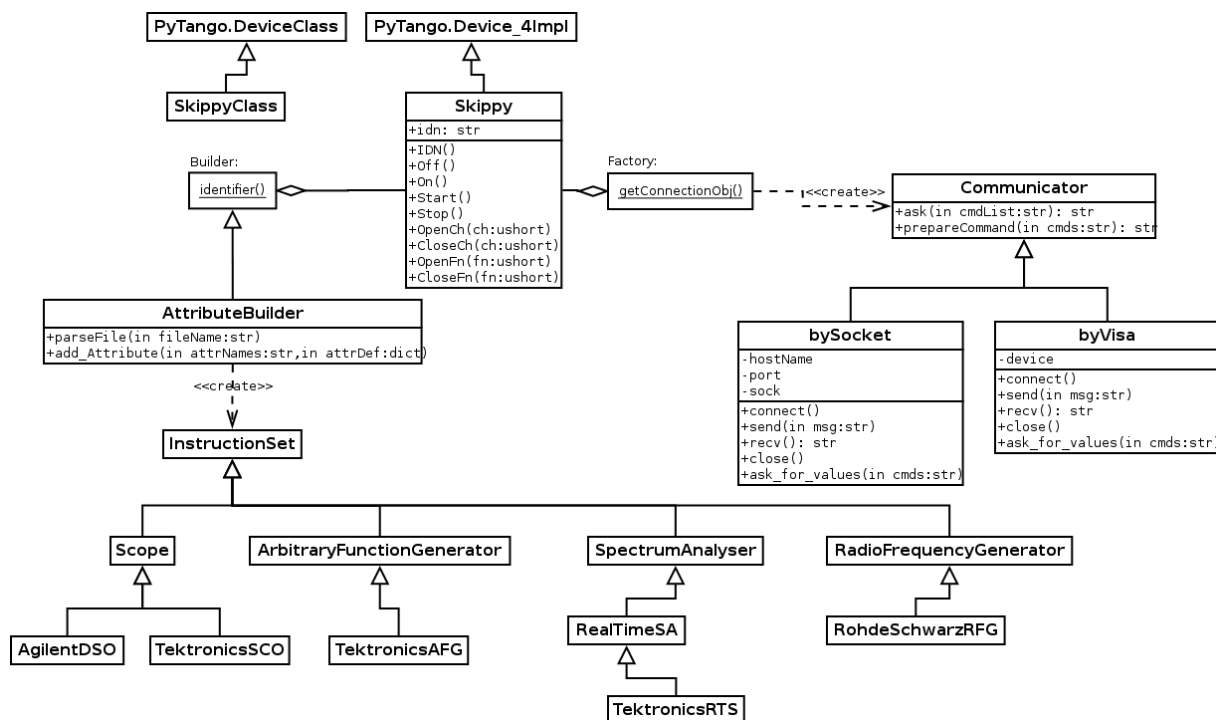
Another example of an attribute definition:

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

That example, in a RF generator, will setup 3 READ.WRITE double attributes, called **Frequency**, **FrequencyStep**, **FrequencyStepSpeed**. This describes a behaviour 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.

```
Attribute('Waveform',
        {'type': PyTango.CmdArgType.DevDouble,
         'dim': [1, 4000000],
         'readCmd': lambda ch,num: "WAVEform:SOURce_%s%d ; WAVEform:DATA?"%(ch,num),
         'channels': True,
         'functions': True,
        })
```

1.5 Class Diagram



1.6 Device features

- 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.

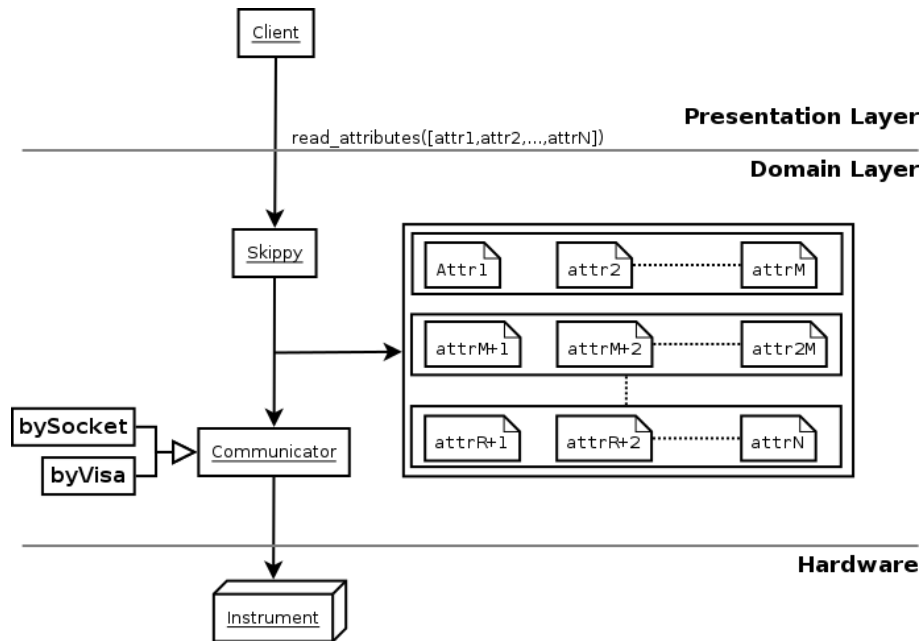


Figure 3: multiple request of readings to the instrument.

- 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 attributes, internals in the device, must be built also: a Step and a StepSpeed. On the write operation a thread should be 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.
- 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 for it in particular.
 - **TODO:** "A command way feature is required to add and remove elements from this monitoring list."
- **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."