BLISS

Beamline Instrument Software Support

ESRF

SpecClient user's guide

Author : Matías Guijarro

Date : 18/06/04

Contents

Figures

1. Introduction

This document is a short guide to the SpecClient package, a set of Python modules for interacting with remote Spec servers.

1.1 History

At the same time as the new client/server features were implemented in Spec, a **specclient** Python module was written to test the new capabilities. It was the first attempt to have a general purpose Python module to interface Spec with the rest of the world.

Later, development on the Bliss Framework project showed that the former **specclient** module did not suit well with to the specific needs of the project, and a different module has been written : **AsyncSpecClient**. It emphasized on an asynchronous behaviour for communicating with Spec and relied on the 'signals and slots' callback mechanism provided by the Qt GUI toolkit.

The **new SpecClient package** merges the good ideas of the two previous modules in one unified piece of software.

1.2 Features

The SpecClient package does not depend on a higher level toolkit or library. It is directly usable in a simple Python script, or in the Python interpreter in an interactive session. It is easy to bind with any GUI toolkit for creating Spec client graphical user interfaces.

The following functionalities are implemented :

- **Access to the hardware controlled by Spec** : motors and counters
- **Ability to execute command**s and get results
- **Variables watching**
- **Errors notification**[1]

The SpecClient package can be used in two ways :

- **for 'scripting'**
- **embedded in a GUI**

Generally speaking, the expected behaviour when writting scripts is that Spec objects and methods could be used a bit like if the user were in front of the Spec terminal window : connecting to Spec would block until the connection is effectively established, in the same way that we wait for the prompt when starting Spec. Moving a motor should block until the motor has finished moving, and so on. Blocking while doing something else is good.

On the contrary, when embedding SpecClient in a GUI, you almost never want an operation to 'block'. Instead, you want to launch things and being informed of changes in order to update your widgets accordingly. In this case moving a motor can be viewed as a background task, emitting notifications when the motor starts moving or stops. The GUI main loop is never interrupted, so your graphical interface still behaves normally.

The SpecClient package usually contains two set of classes : ones **'synchronous'**, dedicated to scripting and others **'asynchronous'** for using with a GUI. Asynchronous classes are suffixed with **'A'**. For example, **SpecMotor** is the synchronous class representing a motor in Spec, and **SpecMotorA** is the asynchronous flavour.

1 Due to actual Spec limitations, the errors notification support remains poorly supported

2. Getting started

This section is a quick step guide to the main aspects of the SpecClient package. After two introductory chapters, we will learn how to use the SpecClient package by studying examples. If you are interested in embedding SpecClient in a GUI please read section 3, '*Embedding SpecClient in a GUI*'.

2.1 Importing SpecClient

The SpecClient package contains different files :

```
SpecClient/
      __init__.py
      SpecMotor.py
      SpecCounter.py
      SpecCommand.py
      Spec.py
      SpecVariable.py
      SpecConnectionsManager.py
      SpecChannel.py
      SpecReply.py
      SpecWaitObject.py
      SpecConnection.py
      SpecMessage.py
      SpecEventsDispatcher.py
      SpecServer.py
```

FIG. 1: files in the SpecClient package

When importing elements from the SpecClient package, you should use a `from...import...` statement in Python :

```
from SpecClient import SpecMotor
```

FIG. 2 : importing an element from the SpecClient package

You can also use the wildcard `*` to automatically import the key elements of the package. The key elements of the package are the following modules :

- **SpecMotor,**
- **SpecCounter,**
- **SpecCommand,**
- **Spec,**
- **SpecVariable.**

Starting from here, you will learn how to use the SpecClient package through studying small pieces of code.

You are strongly encouraged to try them one by one.

Before continuing, please start a Spec version in Server mode (-S command line flag) and a Python interpreter.

2.2 Moving a motor

Moving a motor from the Python interpreter can be achieved using the **SpecMotor** class*. Each SpecMotor object represents a motor in the remote Spec version.

The SpecMotor constructor takes the following arguments :

- Spec motor name
- *'host:version'* string representing the Spec version to connect to
- optional timeout in milliseconds

If timeout occurs while trying to connect Spec, a **SpecClientTimeoutError** exception is raised.

Let's create a SpecMotor object bound to a motor named *'tth'* controlled by a *'spec'* Spec version running on the *'boom'* host :

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from SpecClient import SpecMotor
>>> m = SpecMotor.SpecMotor('tth', 'boom:spec', 500)
```

FIG. 3 : creating a new SpecMotor object

You can obtain the absolute position of a motor using the `getPosition()` method. You can move it using the `move()` method :

```
>>> m.getPosition()
12.5
>>> m.move(m.getPosition() + 1)
>>> m.getPosition()
13.5
>>>
```

FIG. 4 : getting a motor position and moving

2.3 Counting

Counting in the SpecClient package can be achieved using **SpecCounter*** objects. Each SpecCounter object represents a counter in the remote Spec version. Creating a SpecCounter object is much like the same as for a SpecMotor object :

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from SpecClient import SpecCounter
>>> c = SpecCounter.SpecCounter('sec', 'boom:spec', 500)
>>>
```

FIG. 5 : creating a SpecCounter object

* Class: SpecMotor, module: SpecMotor, package: SpecClient
* Class: SpecCounter, module: SpecCounter, package: SpecClient

The `c` object now refers to the counter *'sec'* in the *'spec'* Spec version running on *'boom'*. We can count a certain number of seconds on this counter object :

```
>>> c.count(1)
1
>>>
```

FIG. 6 : counting

The returned value is the counter's value after counting.

You can get a counter's value by calling the `getValue()` method :

```
>>> c.getValue()
1
>>>
```

FIG. 7 : obtaining a counter's value

2.4 Executing commands

The **SpecCommand*** class encapsulates a Spec command in a Python object. The SpecCommand constructor takes the following arguments :

- command name
- *'host:port'* string representing a Spec version to connect to
- optional timeout

If timeout occurs when attempting to connect, a **SpecClientTimeoutError** exception is raised.

Once you have created a SpecCommand object, you can execute the corresponding Spec command by calling the object and giving the command arguments, as if it were a Python function.

2.4.1 Executing macros

We are going to create a '*mvcmd*' object representing the '*mv*' macro in Spec, and then move a motor using the command :

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> mvcmd = SpecCommand.SpecCommand('mv', 'boom:spec', 500)
>>> mvcmd('tth', 12)
0
>>> from SpecClient import SpecMotor
>>> tth = SpecMotor.SpecMotor('tth', 'boom:spec')
>>> tth.getPosition()
12
>>>
```

FIG. 8: defining a new command and executing it with arguments

* Class: SpecCommand, Module: SpecCommand, Package: SpecClient

You can notice that the reply you get after command execution is 0. You cannot rely on the reply code when executing macros.

A **SpecClientError** exception is raised if a problem occured during macro execution. The recommended way is then to use a `try...except` clause when executing commands.

2.4.2 Executing macro functions

Executing macro functions and Spec builtin functions is much the same as with macros. The main difference is that reply values are significant. There is another point depending on the Spec server version you are connected to. Two cases :

- you are connected to a Spec server >X : Spec is able to determine if it is executing a macro or a function. There is nothing to do.
- otherwise, it is ought to you to indicate if you are executing a macro or a function, using the 'function' keyword argument when calling the command :

Latest Spec release

```
>>> from SpecClient import
SpecCommand
>>> motor_mne =
SpecCommand.SpecCommand
('motor_mne', 'boom:spec')
>>> motor_mne(5)
ssf
>>>
```

Before

```
>>> from SpecClient import
SpecCommand
>>> motor_mne =
SpecCommand.SpecCommand
('motor_mne', 'boom:spec')
>>> motor_mne(5, function=1)
ssf
>>>
```

FIG. 9: executing a function

2.4.3 Executing arbitrary strings

Any SpecCommand object can execute arbitrary strings, not only the command defined in the constructor. Be very careful when executing arbitrary strings in a remote Spec version.

```
>>> anycmd = SpecCommand.SpecCommand('', 'boom:spec', 500)
>>> anycmd.executeCommand('12+3')
15
>>>
```

FIG. 10: executing a command string

2.5 Spec objects

Spec* objects act like a remote Spec version. Spec objects can execute any command, as if it were a method of the Python object. Spec objects also define their own methods, for obtaning informations from a remote Spec version.

The constructor takes only a *'host:port'* string, representing the remote Spec version and host to connect to :

```
>>> from SpecClient import Spec
>>> spec = Spec.Spec('boom:spec')
```

FIG. 11 : creating a Spec object

* Class: Spec, Module: Spec, Package: SpecClient

Spec objects can execute any command :

```
>>> spec.mv('psvo', 1)
0
>>> spec.p('hello')
1
>>>
```

FIG. 12: executing commands with Spec objects

The same rules as for SpecCommand objects apply : when executing a macro function or Spec builtin function, if the remote Spec is an old release you have to specify the 'function' keyword argument.

Note that the return values are not significant when executing macros ; in case of error, a **SpecClientError** exception is raised.

Spec objects also have some own methods. The `getMotorsMne()` method returns a list of all the motor mnemonics defined in the connected Spec version :

```
>>> spec.getMotorsMne()
['psb', 'psf', 'psu', 'psd', 'ssb', 'ssf', 'ssu', 'ssd',
'psho', 'sampy', 'sampx', 'light', 'zoom', 'phiz', 'phiy',
'phix', 'phi', 'pshg', 'psvo', 'psvg', 'ssho', 'sshg',
'ssvo', 'ssvg', 'att1', 'att2', 'wbv', 'mbv1', 'mbv2', 'u35',
'push111', 'm2', 'mono', 'egy', 'calo', 'din', 'tor01f',
'tor02b', 'tor03b', 'mtf', 'mtb', 'mben', 'torh', 'roll',
'tory', 'mtt', 'yaw']
>>>
```

FIG. 13: the getMotorsMne() method

You also have `getVersion()` and `getName()`.

2.6 Reading and writing Spec variables

Reading and writing Spec global variables can be easily achieved using **SpecVariable*** objects. It is a thin wrapper around SpecChannel objects, provided for convenience :

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from SpecClient import SpecVariable
>>> toto = SpecVariable.SpecVariable('toto', 'boom:spec')
```

FIG. 14 : creating a SpecVariable object

The constructor takes at least two arguments :

- name of the Spec variable
- *'host:port'* string for connecting to the required Spec version
- optional timeout in milliseconds

If timeout occurs when attempting to connect, a **SpecClientTimeoutError** exception is raised.

SpecVariable objects have 4 methods :

* Class: SpecVariable, Module: SpecVariable, Package: SpecClient

- `isConnected()`: returns wether the remote Spec is connected or not
- `getValue()` : reads the corresponding channel and returns the value of the variable
- `setValue(value)` : writes `value` to the corresponding channel
- `waitUpdate(waitValue, timeout)` : waits for channel to be updated ; the two arguments are optional. If you want to wait for a specific value, assign a value to the `waitValue` keyword argument. If you want to set a timeout, specify it in milliseconds with the `timeout` keyword argument.

3. Embedding SpecClient in a GUI

SpecClient is designed to be used with a GUI. The following paragraphs show how to take advantage of this feature when embedding SpecClient in a Graphical User Interface.

3.1 Dispatching events

Internally, the SpecClient package uses events to trigger actions when receiving messages from Spec. For example, when a remote Spec server gets connected, a '*connection*' event is emitted by the corresponding SpecConnection object. Then, the 'listenning' objects act accordingly, i.e a SpecChannel object can register when the connection is established.

The events machinery is implemented in the **SpecEventsDispatcher** module within the SpecClient package. In particular, the module defines a `dispatch()` function. In order for events to be dispatched from event emitters to event receivers, you should call `SpecEventsDispatcher.dispatch()` at a regular interval. Generally, 20 milliseconds is a good polling interval ; you are free to use another value, however.

When embedding SpecClient in a GUI this is commonly achieved inside a timer object, or using the GUI main loop idle time :

Tk

```
from SpecClient import SpecEventsDispatcher

root = Tk.Tk()

root.after(20,
SpecEventsDispatcher.dispatch)

root.mainloop()
```

Qt

```
from SpecClient import SpecEventsDispatcher

self.timer = Qtimer(self)

QObject.connect(self.timer,

SIGNAL('timeout()'),
SpecEventsDispatcher.dispatch)

self.timer.start(20)
```

FIG. 15 : installing the SpecClient dispatcher in a GUI loop

3.2 The asynchronous motor class

The SpecClient package provides a motor class specially designed to be derivated : **SpecMotorA**[*]. You should use this class as a base class for your own GUI-oriented motor class. In this scheme, you define particular actions and behaviour (changing colors, putting to enabled/disabled state, etc.) by overriding methods of the SpecMotorA class. The methods will get called by the SpecClient events dispatcher while events are coming from the remote Spec server.

The SpecMotorA constructor takes the following arguments :

- Spec motor name
- *'host:version'* string representing the Spec version to connect to

The two arguments are optional ; if not specified, the SpecMotorA object is created but does nothing. You can connect to Spec later by calling the `connectToSpec()` method with the same arguments.

Methods to be overriden :

- `connected()`, called when the remote Spec version gets connected
- `disconnected()`, called when the remote Spec version gets disconnected
- `motorLimitsChanged(limit)`, called when the motor limits change in Spec. '*limit*' can be low or high limit ; use `getLimits()` to get a *(low limit, high limit)* tuple

* Class: SpecMotorA, Module: SpecMotor, Package: SpecClient

- `motorPositionChanged(absolutePosition)`, called when motor position has changed. '*absolutePosition*' is the updated absolute motor position
- `syncQuestionAnswer(specSteps, controllerSteps)`, called when answer to the famous sync question is required. Return '1' for YES, '0' for NO
- `motorStateChanged(state)`, called when the motor changed state. The possible motor states are global to the SpecMotor module and can be one of : NOTINITIALIZED, UNUSABLE, READY, MOVESTARTED, MOVING, ONLIMIT

The other useful methods in the class are :

- `getPosition()`, return the current motor absolute position
- `getState()`, return the current motor state
- `getLimits()`, return a (*low limit, high limit*) tuple
- `getDialPosition()`, return the current motor dial position
- `move(position)`, move the motor to a new absolute position
- `stop()`, abort moving ; *WARNING : due to Spec behaviour, calling* `stop()` *on a motor will stop ALL the moving motors.*

3.3 Asynchronous SpecCounter

Does not exist for the moment. Sorry for the inconvenience.

3.4 Executing commands asynchronously

The **SpecCommandA**[*] class permits to execute commands asynchronously. The constructor takes the following parameters :

- command name
- *'host:version'* string representing the Spec version to connect to

Both arguments are optional. You can call the `connectToSpec()` method with the same arguments for differed connection.

The SpecCommandA class behaves very much like the SpecCommand class (see 2.5). The only difference is that it is designed to be subclassed, and that it is not blocking. In order to provide custom waiting according to your GUI needs (message to user, disabling control, etc.) and getting results back just override the following methods :

- `beginWait()` : called when the command has just been send
- `replyArrived(reply)` : called when reply from Spec has arrived. '*reply*' is a **SpecReply** object with two main members `.data` (*returned value)* and `.error` *(boolean)*

3.5 Watching variables

Watching variables asynchronously is done with the **SpecVariableA**[**] class. The constructor takes the following arguments :

- variable name
- *'host:version'* string representing the Spec version to connect to

---

* Class: SpecCommandA, Module: SpecCommand, Package: SpecClient

** Class: SpecVariableA, Module: SpecVariable, Package: SpecClient

The two arguments are optional. You can connect to Spec later by calling the `connectToSpec()` method with the same arguments.

The methods supposed to be overriden are :

- `connected()`, called when the remote Spec gets connected
- `disconnected()`, called when the remote Spec gets disconnected
- `update(value)`, called when the variable value changed

Other useful methods :

- `getValue()`, return the current variable value
- `setValue(value)`, write the variable value

4. SpecServer

TO BE DOCUMENTED. Sorry for the inconvenience.

5. Logging

SpecClient offers some logging features relying on the logging package of the standard Python 2.3 distribution. The next paragraphs show how to set logging on or off, or to redirect logging messages to files.

5.1 What happens when importing SpecClient

When importing the SpecClient package, a new *"SpecClient"* logger is created. If there is no root logging handler previously created, the messages will go to stdout.

If there is one root logger, no specific handler will be created and messages will be dispatched as the other ones.

5.2 Set logging on / off

By default, logging is enabled. To set logging off, just call the `setLoggingOff()` method of the SpecClient package :

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import SpecClient
>>> SpecClient.setLoggingOff()
```

FIG. 16 : turn logging off

Alternatively, turning logging on is achieved by calling `setLoggingOn()`.

5.3Log to rotating files

You may prefer to send logging messages to rotating files. In this case, call the setLogFile() method, with a filename as first argument : logging messages will be send to this file. If the file gets bigger than 1 MB, a second file will be created with a different suffix. If there is more than 5 these files, the first one will be overwritten, and so on.

```
Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import SpecClient
>>> SpecClient.setLogFile('/tmp/specclient_log')
```

FIG. 17 : log to rotating files