26/10/2021

IO 6: WEBLAB TUTORIAL OF TECHNICAL DESIGN AND IMPLEMENTATION – SERVER IMPLEMENTATION (PYTHON)

Co-funded by
the European Union

1. INTRODUCTION

A remote lab interface developed with EjsS can easily communicate with the lab hardware/software using the Remote Interoperability Protocol (RIP). For this, two things are needed: EjsS' RIP element (see the **Client implementation** manual) and a RIP server implementation.

This manual describes how to use the RIP server implemented in Python. There is another one for the LabVIEW implementation of the RIP server (see the **Server implementation (LabVIEW)** manual). The Python implementation addressed in this document is extremely useful when the lab equipment is controlled with MATLAB/SIMULINK, Node.js, Arduino, Red Pitaya boards, etc. as it presents interfaces for all these solutions. Chapter 2 of this manual explains how to configure the RIP server and use it to publish your control programs as web services that can be consumed by EjsS applications with the RIP element.

But first, you need to know where to get the Python implementation of the RIP Server. The software is available at https://github.com/UNEDLabs/rip-python-server. For the most advanced users (those who want to do some development), a document presenting the specification of the Remote Interoperability Protocol can be found here: https://github.com/UNEDLabs/rip-spec.

2. USING THE PYTHON IMPLEMENTATION OF THE RIP SERVER

The Python implementation of the RIP Server is distributed as source code. You can get the most recent version in the github repository (https://github.com/UNEDLabs/rip-python-server). Download it as a zip file or, if you have git installed, clone the repository opening a terminal and typing the following command:

```
git clone https://github.com/UNEDLabs/rip-python-server
```

This guide assumes you have a Python 3 distribution already installed. If not, visit https://www.python.org/ and download a recent version.

The following sections will guide you on how to: 1) set up a Python virtual environment and check whether your rip-python-server is working (2.1), 2) configure your application (2.2), and 3) extend the server to fit your specific needs (2.3).

2.1. Setting up the virtual environment

The rip-server-python depends on the third-party libraries *cherrypy*, *ujson*. Depending on your application, you also may need to install *oct2py* or *MATLAB engine* for Python. You can use your system wide installation of Python, but we consider advantageous to use a virtual environment. Though there are several tools that you can use, this guide uses *virtualenv*. You can install the tool with the following command:

```
pip3 install virtualenv
```

Then, create a new virtual environment:

```
virtualenv -p python3 venv
```

The `-p python3` option is to ensure we are using the correct version, and `venv` is the folder where the enviroment is stored. From now on remember you need to add the prefix `venv/bin/` to execute the commands inside the virtual environment, and not on your system installation. The following step is to install the dependencies:

```
venv/bin/pip install cherrypy ujson
```

Depending on the implementation module you want to use for your application (Matlab, Arduino, etc.), you may need to install other required modules. For example, for Arduino, you would also need to run:

```
venv/bin/pip install serial pyserial
```

The project's code is configured for Matlab by default. Therefore, the following will not work unless you have Matlab installed. To change the implementation module to a different one (Arduino, for example) see Section 2.2.

Finally, you can start the RIP server:

```
venv/bin/python3 App.py
```

The RIP server should start listening in port 8080. To check if it is actually working, open your browser and enter the URL: http://localhost:8080/RIP. You should get a JSON response as shown in Figure 1. The response contains the RIP server metadata describing the capabilities of the server and the experiences hosted in it. You can read the RIP specification for implementation details.

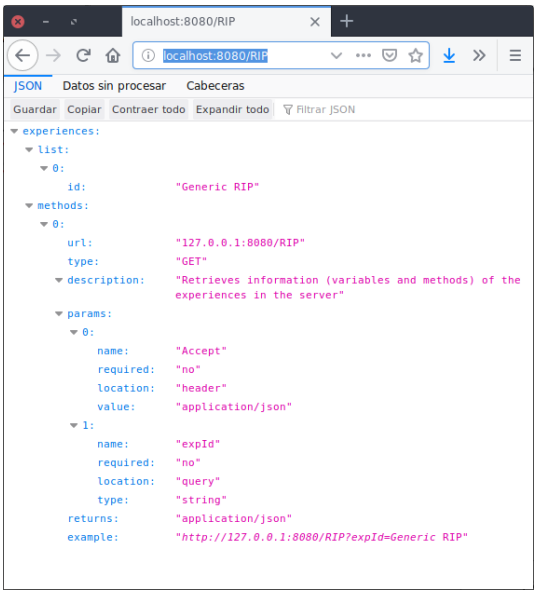

Figure 1. Testing the response of the RIP server.

2.2. Defining your application

The rip-python-server is roughly divided in two tiers: the first one that implements the RIP functionality, and the second one that implements the access to the hardware. As the communication tier is usually not subject to changes, to set up the server you only need to bind the application to a specific hardware implementation in the configuration file `AppConfig.py`, and optionally to provide some extra information about the experience hosted in the server: *readables* and *writables* exposed, authors, keywords, etc. You can find some examples of how to do it in the folder `config-examples`. The following listing correspond to a basic configuration for testing:

```
# This file contains the configuration of the RIP server application.
config = {
  'server': {
    'host': '127.0.0.1',
    'port': 8080,
  },
  'control': {
    'impl_module': 'RIPGeneric',
    'info': {
      'name': 'Generic RIP',
      'description': 'A generic implementation of RIP',
      'authors': 'J. Chacon',
      'keywords': 'Raspberry PI, RIP',
      'readables': [{
        'name':'time',
        'description':'Server time in seconds',
        'type':'float',
        'min':'0',
        'max':'Inf',
        'precision':'0'
      }],
      'writables': []
    }
  }
}
```

Figure 2. RIP server experience configuration.

You need to specify the following fields:

1. `config.control.impl_module`: contains the name of the module that implements the hardware access. You can use any of the built-in implementations (`RIPMatlab`, `RIPOctave`, …) or define your own control module, as will be explained later.
1. `info`: contains the definition of the experience, including the name of the experience, a short description, authors and keywords and the list of *readables*, elements that can be read, and *writables*, elements that can be written.

As before, you can launch the RIP server to test the new configuration, with the following command:

```
venv/bin/python3 App.py
```

The application will listen to the host and port specified in the configuration. Open a new browser window and verify that the server is working.

2.3. Creating a new hardware interface.

In case the built-in hardware adapters do not fit your application, you can write your own code and integrate it with the RIP server. To that end you will have to create a facade that expose your low-level hardware access through the RIP API (see the RIP specification document). To make your task easier, the class `RIPGeneric` (defined in `rip/RIPGeneric.py`) provides the RIP common functionality, so you can subclass it to add your code. At a minimum, you need to do the following tasks:

2. Import the class `RIPGeneric` and any other library you need to use.
3. Create a subclass extending `RIPGeneric`.
    a. Define the constructor `__init__()` with the initialization code.
    b. Define how to read and write server objects, overriding the methods `set` and `get`.
    c. Report the variables that should be sent in the periodic updates, overriding the method `getValuesToNotify`.
4. The following listing contains an example of a minimum implementation provided as a template (note that you need to put your code inside the folder 'rip'). The code creates a server that provides two readable objects: `time`, that return the server uptime, and `random`, that return a random number, and one writable: `seed`, to modify the random generator seed. The server accepts SSE connections and report random numbers periodically to the connected clients.

```
import random
from rip.RIPGeneric import RIPGeneric

class RIPAdapterTemplate(RIPGeneric):
  '''
  RIP Adapter Template
  '''

  def default_info(self):
    return {
      'name':'RIPAdapterTemplate',
      'description':'A template to extend RIP Generic',
      'authors':'J. Chacon',
      'keywords':'Adapter Template',
      'readables':[{
        'name':'time',
        'description':'Server time in seconds',
        'type':'float',
        'min':'0',
        'max':'Inf',
        'precision':'0',
```

```
    },

    {
      'name':'random',
      'description':'Random value generator',
      'type':'float',
      'min':'0',
      'max':'1',
      'precision':'0'
    }],
    'writables': [{
      'name':'seed',
      'description':'Random seed',
      'type':'float',
      'min':'0',
      'max':'1',
      'precision':'0'
    }],
  }
```

Figure 3. RIPAdapterTemplate - Definition.

Code in Figure 3 is straightforward. First, it imports the standard package random that will be used to generate the random numbers and `RIPGeneric`, and then it creates a class named `RIPAdapterTemplate` that extends `RIPGeneric`. The method `default_info` provides the default definition of the experience, that can be overridden in AppConfig.py.

```
def set(self, expid, variables, values):
  '''
  How to write server variables
  '''
  n = len(variables)
  for i in range(n):
    try:
      n, v = variables[i], values[i]
      if v in self._get_writables():
        self.n = v
    except:
      pass

@property
def seed(self):
  return self._seed

@seed.setter
def seed(self, value):
  random.seed(value)


def get(self, expid, variables):
  '''
  How to read server variables
  '''
  toReturn = {}
  n = len(variables)
  for i in range(n):
    name = variables[i]
    if v in self._get_readables():
      toReturn[name] = random.rand
  return toReturn

@property
def random(self):
  return random.random()

@random.setter
def random(self, value):
  pass
```

Figure 4. RIPAdapterTemplate – Methods get and set.

Figure 4 shows the definition of methods `set()` and `get()`. These methods just check if the corresponding variable is writable(readable) and writes(reads) the value. Using Python *properties*, the objects are mapped to the methods provided by the package `random`.

Finally, the method `getValuesToNotify` returns the variables that will be reported periodically to the user (using the SSE channel). The method should return a list where the first element is a list containing the name of the variables notified and the second element is another list with the corresponding values.

```
def preGetValuesToNotify(self):
  pass

def getValuesToNotify(self):
  '''
  Variables to include in periodic SSE updates
  '''
  return [['time', 'random'], [self.sampler.lastTime(), self.random]]

def postGetValuesToNotify(self):
  pass
```

Figure 5. RIPAdapterTemplate - Periodic updates.

Optionally, you can override the method `preGetValuesToNotify` and `postGetValuesToNotify` in case you need to execute some actions before and after reading the variables, respectively. For example, in an implementation that uses a MATLAB session, you can use `preGetValuesToNotify` to run some MATLAB code that updates the workspace, prior to reading the values of the variables.