#Session 5 : Queues and Remote Procedure Calls

General guidelines:

- Most of the proposed exercises will led to the writing of python2.7 functions to be stored in two scripts:
  - unless specified, the script that implements the processing with name *S4_queues_tools.py*.
  - the test script that applies unit tests on all the functions written in the previous script.

  .
- In this session, Queues will be used using the RabbitMQ framework and the CloudAMQP platform as a queuing service. Client side python library will be pika (use 'pip install pika' command to install it on your own computer). Other tools related to RabbitMQ can be found here.
- All the provided functions and scripts must be documented using the Doxygen syntax.
- Special note on materials to be submitted for continuous evaluation :
  - All the instructions on script and function names must be rigorously followed. As for continuous integration Automatic code testing will be considered to evaluate your submissions and will rely on the imposed prototypes. Any mistake will then impact on your evaluation.
  - All the codes must have been verified using SonarCloud.
  - Each student must fork the following GitHub project on its own GitHub account `<https://github.com/albenoit/USMB-BachelorDIM-Lectures-Algorithms>`
  - **Once an exercise is done**, the scripts must be pushed to your GitHub repository in folder assignments/Session4.
  - **At the end of the session**, the scripts must be submitted to the main GitHub project using a pull request.

Important note : be warned by all the codes you push on GitHub are open source and any person see it, comment and report on it. Then, copy and past between students and other inappropriate submissions as well as well written algorithms and good ideas will remain accessible to any observer. As a consequence, pay attention to your submissions.

Remote Call Procedure : basic message exchange

Remote call procedure consists in requesting a remote server and waiting for an answer. On will first design a basic application: A client sends a 'hello, how fine ?' string message and the server replies 'Fine and you ?' in the same way:

1. create the server script called *rpc_ server.py* that must have the following behaviors:
   - it connects to the channel called *'rpc_ queue'* and consumes all the messages by calling the callback called *on_ request*.
   - the *on_ request* callback should print the received request and simply reply with message 'Fine and you?' as a reply, using the correlation id received within the request properties.
   - finally, the callback acknowledges the request once the answer has been sent.
2. create the client script called *rpc_ client.py* that must have the following behaviors:
   - it establishes an exclusive channel used to collect server replies. Keep the queue name as variable *callback_ queue*.
   - it creates a unique communication identifier that will serve as the correlation id of the RPC process. To do so, use the *uuid* python package and its method *uuid4*.
   - it publishes the 'Hi, how fine?' string message on the *'rpc_ queue'* while specifying the correlation id and the exclusive queue *callback_ queue* to consider when replying.
   - set the client as a consumer of the *callback_ queue* queue ; acknowledgement is not necessary.
   - define function *on_ response* to be called when a reply is received. This method checks if the correlation id corresponds to one established earlier in the process (this is a way to detect intrusions in the system). It throws an Exception if the correlation id does not match and prints the received answer in the other case.
3. Test the server and client scripts and monitor messages on the RabbitMQ instance manager on CloudAMQP.
4. Experiment with multiple clients requesting the same server.
5. Experiment with multiple servers. Use Quality of Service option used before for enhanced server load balancing.
6. Be sure your function is documented following the Doxygen syntax.
7. Commit the code to your local repository and push to your GitHub repository.

Notes:

- the best practice would be to use Object Oriented programming, especially on the client side but this is out of the scope of this assignement. Refer to RabbitMQ RPC to get an Object modeling example.

- the proposed solution is not perfectly safe. Issues such as server timout, server failure or Exceptions at the server level and so should be managed. For industry grade applications, you should take care of this and may have a look at the previous assignments to enhance the current method.

Messaging, what kind of message should you manage ?

Until now, we only played with string messages. But what if we need to deal with structured messages ? You are already familiar with XML and JSON to embed multiple data in the same message. Let's have a deeper look into the problem !

Here are some experiments to get more familiar with this important topic.

1. get back to the previous assignment and now try to send a structured message such as the dictionary *msg={'type':0, 'value':'hi, how fine?'}*.
2. run you program. Does it work fine ?
3. actually only string messages can be transmitted. You then need to *serialize* messages. To fix the previous attempt, convert the dictionary to a string using the *str()* function.
4. test the same thing if the message is a data matrix. To this end, using numpy, generate a random numbers matrix like this *numpy.random.random((20,30))* and serialize using *str()*.
5. now consider a specific serializing library. We focus on MessagePack for its good simplicity/efficacy compromise:
   - install the numpy-message package : using command *pip install msgpack-numpy*
   - import the library this way:
     import msgpack
     import msgpack_numpy as m
     import numpy as np #if Numpy is required.
   - Then, serializing any *message* variable can be done using command :
     $encoded\_message = msgpack.packb(message, default = m.encode)$
     and decoding an encoded message can be done this way:
     $decoded\_message = msgpack.unpackb(encoded\_message, object\_hook = m.decode)$.
   - test the previous message length measures on those new serialized messages. What compression factor is obtained ?

Remote Call Procedure : remote server data processing

We consider here the concept of delegating high cost processing demands. A typical example is image processing: a mobile device captures an image and sends it to a server that does some processing and replies.

We will now consider the image processing scripts you did at the last session and put them on a server and define client script that will send their images to the server to obtain the filtered result.
Note : this is really a simple example to allow you to get a working application. Such kind of application can be necessary for Internet of Things apps with low energy image capture devices. However, do not consider this for smartphone applications since such device already have enough processing power for image filtering on board. It can nevertheless be necessary for more demanding processing such as image captioning (generate text from large image or video) and other complex image recognition tasks and server centralized application.

1. Get or finish the image filtering script that you developed on Session 3. And ensure functions $invert\_colors\_manual(input\_img : ndarray) : returns\ ndarray$ and $threshold\_image\_manual(input\_img : ndarray) : returns\ ndarray$ work fine. Only if available consider the OpenCV version.

2. Create a simple RPC server script *simple_ imgProc_ rpcServer.py* called that :

   - expects a message that is a serialized numpy matrix.
   - calls the $invert\_colors\_manual(input\_img : ndarray) : returns\ ndarray$ function from the original file $S3\_imgproc\_tools.py$. and returns the filtered image to the client.

3. test the server with the dedicated client script *simple_ imgProc_ rpcClient.py*.

4. improve those scripts by:

   - consider the filters dictionary *filter_ types={'invert':'invert image colors', 'threshold':'threshold an image'}* and the message dictionary prototype to use to process image *myImage* with filter *myFilter* : *message_ proto={'filter_ type':myFilter, 'image':myImage}*
   - the client script reads an image, sends such a structured message and waits for a reply and display the filtering result.
   - the server receives and process the selected filter and returns the filtered image.

5. Be sure your function is documented following the Doxygen syntax.

6. Commit the code to your local repository and push to your GitHub repository.

Remote Call Procedure : remote server data processing WITH ROUTES

This time, consider the same pipeline but the filter selection has to be done using the routing capabilities of RabbitMQ:

1. create the client in the dedicated client script *simple_ imgProc_ rpcClient_ routes.py*.
2. this script reads an image, selects a filter and sends a message that only contains an image but the message is published using a route which name is one of the filters dictionary *filter_ types={'invert':'invert image colors', 'threshold':'threshold an image'}*
3. create the server with the dedicated client script *simple_ imgProc_ rpcServer_ routes.py*.
4. This server script is able to route the messages to specific RPC queues dedicated to each filter.
5. test the script and check program latencies on high server load.
6. Be sure your function is documented following the Doxygen syntax.
7. Commit the code to your local repository and push to your GitHub repository.