#Session 4 & 5 : Queues and Messaging

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-2020>`
  - **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.

Introduction to simple Queues: one publisher, one reader

1. First setup your CloudAMQP account using the official documentation and the information provided with the lesson.
2. copy the basic example provided with the lesson to define 3 scripts:
   - the basic script *simple_ queue_ publish.py* able to connect to the queue named *'presentation'* and publish a string message containing the user name.
   - the basic script *simple_ queue_ read.py* able to reader all the messages available into the queue *'presentation'*.
   - the script *queue_ publish_ read.py* using the *argparse* library. Propose an argument to the user that enable to switch between publish or read mode. To do so, create an argument parser instance and use the *add_ argument* activated by the command option '–read' with option *action='store_ true'* to activate read mode... get into the python documentation to come over this problem.
3. Test your scripts and monitor the message queuing events on your RabbitMQ instance of your CloudAMQP account.
4. Modify your reader scripts to count the number of read messages.
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.

Larger scale application : multiple concurrent queue readers

The following approach is of interest at least for Web and mobile applications to delegate data processing to other applications. However, when processing a message takes too much time, user experience may be impacted by delays and single thread server load is suboptimal. To come over such issues, using concurrent readers come to the rescue! This concept enables a kind of multi-threading approach that enables parallel message processing. Then one can take full advantage of system resources and significantly reduce response time. Let's experiment !

1. Update the basic script *simple_ queue_ publish.py* to activate persistent message modes when using command line option *–concurrency*.
2. Update the basic script *simple_ queue_ read.py* to activate message acknowledging able when using command line option *–concurrency*.
3. Prepare a publishing routine that uses the *simple_ queue_ publish.py* script to emit 1000 persistent messages to the same queue. **Take care of your publishing limits if you rely on a Little Nemur active plan**.

4. Prepare a reading routine that creates at least two readers.
5. Run the two routines and check how readers do receive the messages. What happens if a reader is slower than the other(s) ? Simulate delays by making a reader sleep using the *sleep* function of the *time module*.
6. interrupt a reader that is reading a message and observe the consequent issues.
7. Following the lesson slides, add safety parameters and load balancing options. Observe the effect by restarting experiments.
8. merge the changes into the *queue_publish_read.py* script and validate again.
9. **Performance comparison**: compare the two approaches (non concurrent v.s. concurrent message reading) in terms if processing speed and CPU load. To this aim, first stack a large number of messages in the queue. Then, monitor (time and load) only the message reading phase.
10. Be sure your function is documented following the Doxygen syntax.
11. Commit the code to your local repository and push to your GitHub repository.

Message Broadcasting

Just like social networks where messages from a single user can be broadcasted to all its subscribers. RabbitMQ can handle this ! Messages are sent to a RabbitMQ instance where subscribers are registered. Each subscriber has its own queue, can connect at any time and receives all the last messages starting from the connexion time stamp.

1. create the script *publish_fanout.py* able to connect to the exchange called 'posts' and publish a string message.
2. create the reading subscriber script *read_subscriber.py* able to connect to the same exchange and process the captured messages.
3. test the approach in the following conditions
   - a single publisher regularly emits messages to the exchange **but no subscriber is registered**.
   - a single publisher regularly emits messages **while 2 subscriber are already registered**.
   - a single publisher regularly emits messages **while one subscriber is already registered and a second one connects**.
4. Be sure your function is documented following the Doxygen syntax.
5. Commit the code to your local repository and push to your GitHub repository.

Going further, Message routing

One problem to play with queues, find your way !
One will to design a publishing system accessible throw the exchange 'baby_caramail' where:

1. each subscriber first emits a presentation message to the channel 'presentation' where a login is given as a string.
2. once presented, each subscriber messages are sent to the channel named 'posts'.
3. a specific process listens to the 'presentation' queue and counts the user. It must count each user only a single time (pop out multiple presentations of the same user). It displays the list of known users each time a new presentation message is received.
4. at least one subscriber to the 'posts' channel prints all the received messages.

Do do so:

1. create the script *publish_presentation_post.py* able to connect to the exchange called 'caramail' and publish a string on the channel selected threw the command line option '–signin'. If '–signin' command is used, then a presentation message is send, else this will be considered as a post message.
2. create the user monitor script *user_monitoring.py* able to connect to 'presentation' queue and print the list of known user when receiving any new presentation message.
3. In the same way, create subscribers that could also be integrated into the script *publish_presentation_post.py*.
4. Be sure your function is documented following the Doxygen syntax.
5. Commit the code to your local repository and push to your GitHub repository.

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, 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, generate a random numbers matrix like this *np.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.