Lab 6 - RabbitMQ and Docker-Compose

Introduction:

Welcome to Lab 6! In this lab you’ll learn how RabbitMQ message brokering works. you’ll also try out how you can use Message brokering with docker.

**General Instructions:**

* **Preparation**: Watch the **demo video** and read the instructions in this document carefully before you go to the lab or start doing the experiment.
* You do **not** need AWS for this experiment. You can use the lab machines with ubuntu installed.
* You should complete **TASK-1 (Work Queues)** in the lab and submit the SS to Edmodo on the same day.
* You can complete **TASK-2 (Publish/Subscribe)** and submit SSs to Edmodo on the same day or within 2 days of your lab.

**RabbitMQ**

RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, MQ Telemetry Transport, and other protocols.

# **What can RabbitMQ do for you?**

Messaging enables software applications to connect and scale. Applications can connect to each other, as components of a larger application, or to user devices and data. Messaging is asynchronous, decoupling applications by separating sending and receiving data.

You may be thinking of data delivery, non-blocking operations or push notifications. Or you want to use publish / subscribe, asynchronous processing, or work queues. All these are patterns, and they form part of messaging.

**RabbitMQ is a messaging broker** - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

**Docker-Compose**

**Compose** is a tool for defining and running multi-container **Docker** applications. With **Compose**, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

**Objective**

In this lab experiment we will observe how two of the important messaging technique from RabbitMq work, Namely:

1. Work Queues
2. Publish/Subscribe

We will run the messaging application’s and the RabbitMq server as the microservices using docker-compose.

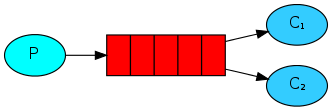
Prerequisites:

* Docker Installed on your system. (Refer to Task1 steps in the Lab5 manual)
* Understand how to write yaml files.

**TASK-1 (Work Queues)**

In this task we will implement the “Work Queues” Message brokering service. We are going to have three services running using Docker compose.

The main idea behind Work Queues (aka: Task Queues) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a task as a message and send it to the queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

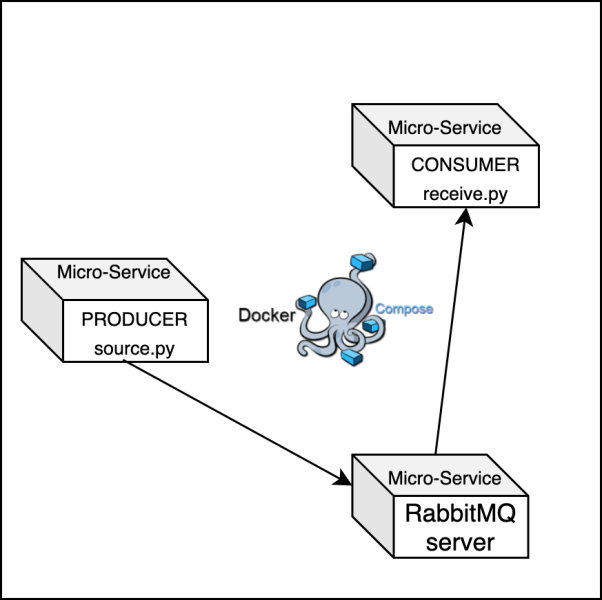


**Background:**

We will build a multi-container application having **three services**. One service will run your RabbitMq Server, Other two will be your Producer service and worker service(consumer).

The Architecture:

1. The producer will Run a Python Application which will push the message to the Queue.
2. The producer will push a certain amount of messages on the Queue.
3. The consumer will run a Python Application which will pop a message from the Queue and print it.
4. Scaling Multiple workers (consumers), the messages will be shared among the workers(consumers).



You are already provided with the application file for the producer service. A producer service run’s an application “source.py”. The producer service will push the message into the queue by connecting to the RabbitMq service running in the same docker-compose network.

What will you be doing:

* Build the python application for the worker service (consumer service).
* Write the YAML file to build the services.
* Scale the Number of workers while you do docker-compose.

**Deliverables SS:**

* **1A.png (SS of the docker compose output)**
* **1B.png (SS of the docker ps output) ( showing what containers are running)**
* **1C.png (SS of the yaml file)**

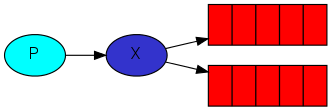
Note:

* please download the source folder given to you
* The source folder will have two directories namely “send” and “receive’. It will also consist of one empty yml file. Please maintain the directory structure given to you.
* The “send” and “receive” folder will have a Dockerfile.
* The “send” folder has “source.py” file.
* The “receive” folder does not has the “receive.py” file.
* We are gonna use the terms “consumer” and “workers” Interchangeably.

**Steps:**

* Build the python application for workers(consumer service).
* The worker (consumer) application should pop the message available in the queue and print it.
* The idea is when we will have multiple workers running( consumers) the messages will be shared among the workers.
* You can get the code for the worker application from [here](https://www.rabbitmq.com/tutorials/tutorial-two-python.html). Make sure you name the file as ‘receive.py” and change the queue name according to what’s given in “source.py” file. Also change the hostname by looking into “source.py”.
* Add the “receive.py” file to receive folder.
* Edit your yml file and add the below mentioned requirements. yaml file will have the skeleton code for you to begin with.
* You need to have three services one service is your rabbitmq server, other two are your producer and consumer. Below you can find the service specifications.
  + RabbitMQ service
    - Service Name: rmq
    - image: rabbitmq:3.8.3-alpine
  + Producer service
    - Service Name: producer
    - build context: “send”
    - Dockerfile: Use the dockerfile inside the send folder
    - command: sh -c "sleep 20 && python source.py" (would suggest to use the exact same command)
    - links: rmq
    - dependency: rmq
  + Consumer service
    - Service Name: consumer
    - build context: “receive”
    - Dockerfile: Use the dockerfile inside the receive folder
    - command: sh -c "sleep 15 && python receive.py" (would suggest to use the exact same command)
    - links: rmq
    - dependency: rmq
* The producer service should use the Dockerfile present inside the send folder.
* The consumer service should use the Dockerfile present inside the receive folder.
* Understand the concept of context.
* Also edit the message being sent as the Message should Include your **SRN** in “source.py”.
* Now run your docker services by scaling the number of consumers to **4.**
* Identify what command to run in order to scale the services while you start them.
* We are scaling the consumer service to create 4 workers. You can observe that the message sent by the sender is shared by the workers(consumers). Take the required SS.

**TASK-2 (**Publish/Subscribe)

****

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an exchange. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the exchange type.

**Background:**

We will build a multi-container application having **three services** just like the previous task. One service will run your RabbitMq Server, Other two will be your Producer service and worker service(consumer).

The Architecture:

1. The producer will publish messages to the exchange on the RabbitMq server.
2. The producer will Run a Python Application which will publish the message to the exchange.
3. The consumer will run a Python Application which is subscribe to the exchange using Queues.
4. Scaling to Multiple workers (consumers), All the messages will be received by all the consumers.

**Deliverables SS:**

* **1A.png (SS of the docker compose output)**
* **1B.png (SS of the docker ps output) ( showing what containers are running).**

**Steps:**

* Please find the “emit\_log.py” and ”receive\_log.py” from this [link](https://www.rabbitmq.com/tutorials/tutorial-three-python.html).
* Understand what each of the applications does.
* Your producer service should run the “emit\_log.py” Application.
* Your consumer service should run the ”receive\_log.py” Application.
* You can use the previous yaml file for this task.
* Just replace the old application files with the new application files.
* Modify the yaml file to run the new application’s.
* Modify the application’s to connect to the RabbitMq server. The producer application should push your “SRN'' as the message.
* You need to start the services and make sure you scale the consumer service while you start.
* you need to scale the consumer service to **5**.
* Take appropriate SS.
* If you are using the same “Source” directory for this task, then make sure you delete your Images and Containers because if you run the service after modifying the application files it won’t build the new image for your services. Or you can refer to this [process](https://stackoverflow.com/questions/32612650/how-to-get-docker-compose-to-always-re-create-containers-from-fresh-images).

**Reference:**

* Hint: We will be using the same application files provided in this link: <https://www.rabbitmq.com/tutorials/tutorial-two-python.html>
* Docker-compose and scaling: <https://medium.com/@karthi.net/how-to-scale-services-using-docker-compose-31d7b83a6648>