**Rabbit MQ Notes: Notes 1/21/2022**

Either the RabbitMQ broker can create the message exchange by default OR a publisher can create its own exchange on? the broker.

**Types of Exchanges:**

* + Direct
    - used for uni-casting
  + FanOut
    - used for broadcasting
  + Topic
    - used for multi-casting
  + Headers
    - works on the message header and not on the routing key / payload

Configuring a subscriber with a *callback function* that activates anytime a new message hits the queue that subscriber is connected to is called the “pull-back method” for consuming.

* This is the preferred method of message consumption

**Section 2: Default Exchange**

In section2 in the Udemy course, he starts by just using the default-created direct exchange.

* You might need to utilize Python3.9 instead of Python3.8

When you have a created queue and would like to list the queues and see what messages there are from the terminal you can use:

*rabbitmqctl list\_queues*

*rabbitmqctl list\_exchanges (als*o useful)

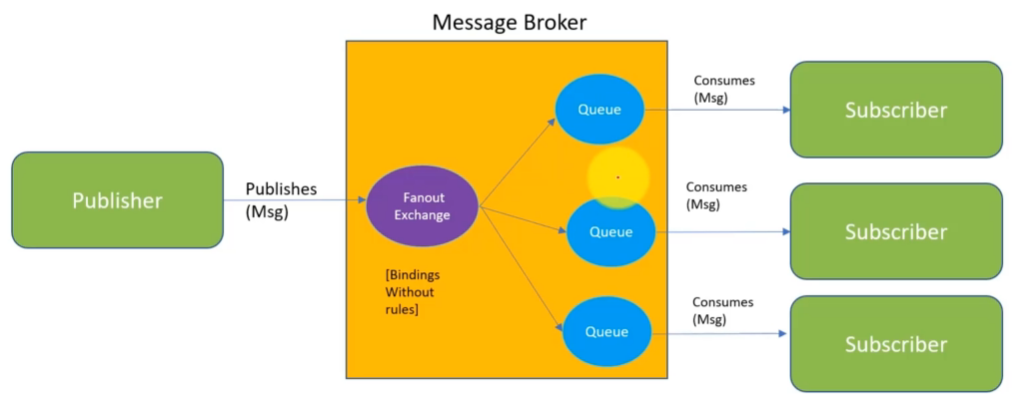
*rabbitmqctl list\_bindings (als*o useful)

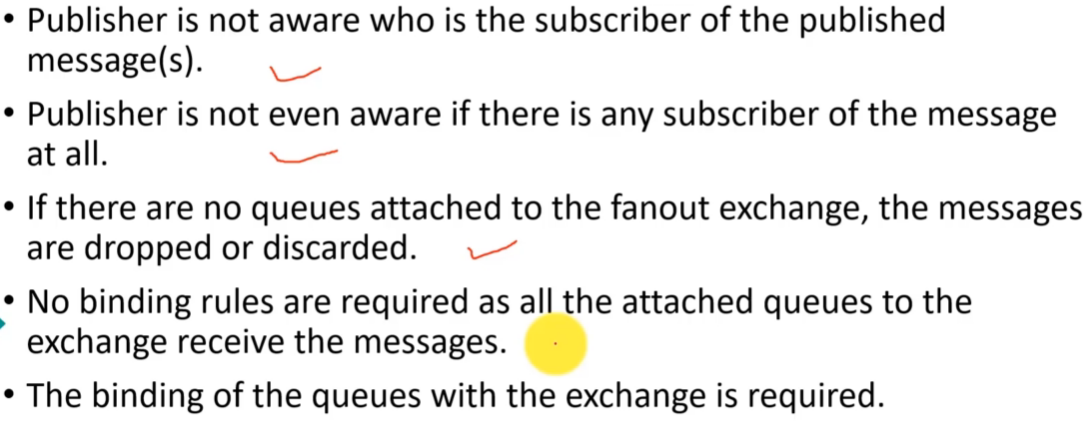
When you have more than one consumer subscribed to the same default queue, they will receiver messages from the queue consecutively in round-robin style. This is the default behavior.

One problem is that the queue remains even if the subscriber’s job is done and it has consumed all of the messages.

* Using exclusive queues solves this problem by deleting the queue when its job is complete. But then this requires further configuration of the subscribers and there can be only one subscriber to each exclusive queue, thus requiring a fanout exchange.

**Section 3: Fanout Exchange**

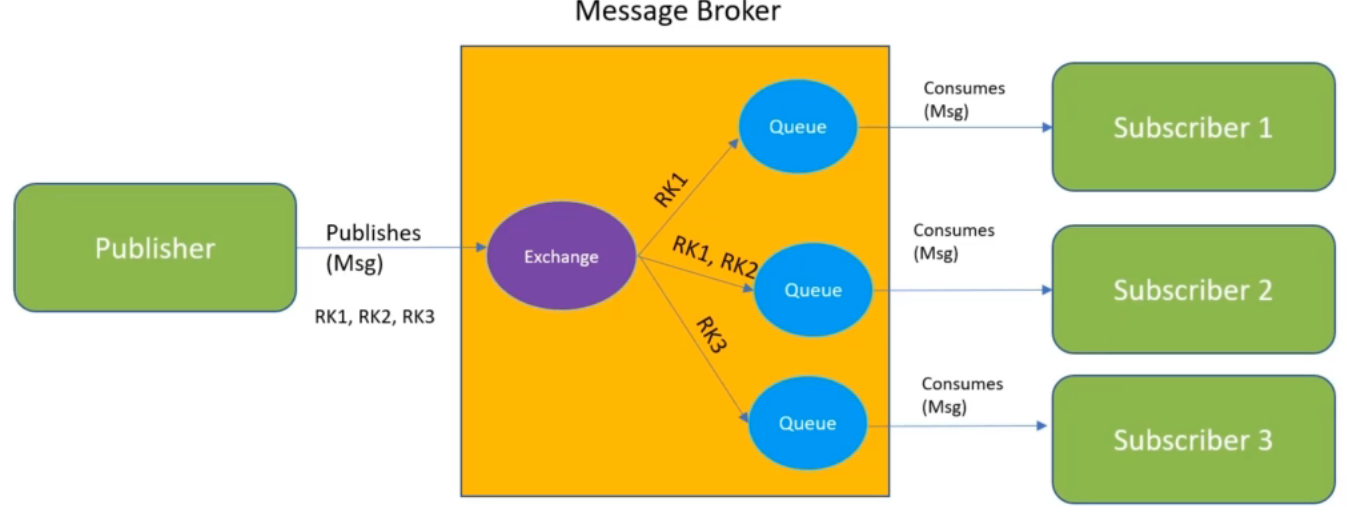
****

****

He claims in a fanout exchange the behavior is that if there are not queues attached then messages are discarded, but I think this likely has something to do with deleting the exchange after broadcasting the messages as in the line of code below and this MUST NOT be the behavior even if it is best practice.

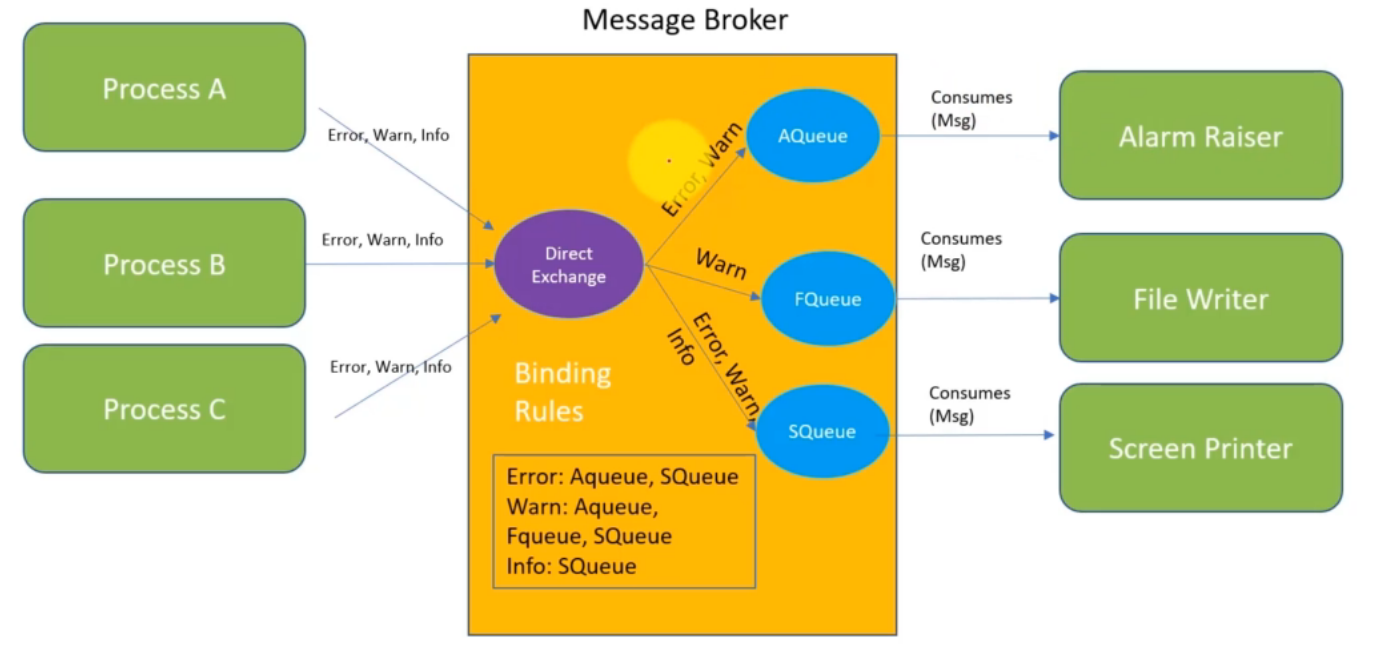
channel.exchange\_delete(exchange='br\_exchange', if\_unused=False)

**Section 4: Selective Routing for Direct Exchanges**

****

An Example of Selective Routingfor a system with processes A,B,C which all produce different logging messages like *error, warn,* and *info*.

* Here we need to implement multiple bindings where multiple subscribers use the same routing key in their binding with the exchange (like “Warn” but not “Info” which is only being consumed by the Screen Printer subscriber)

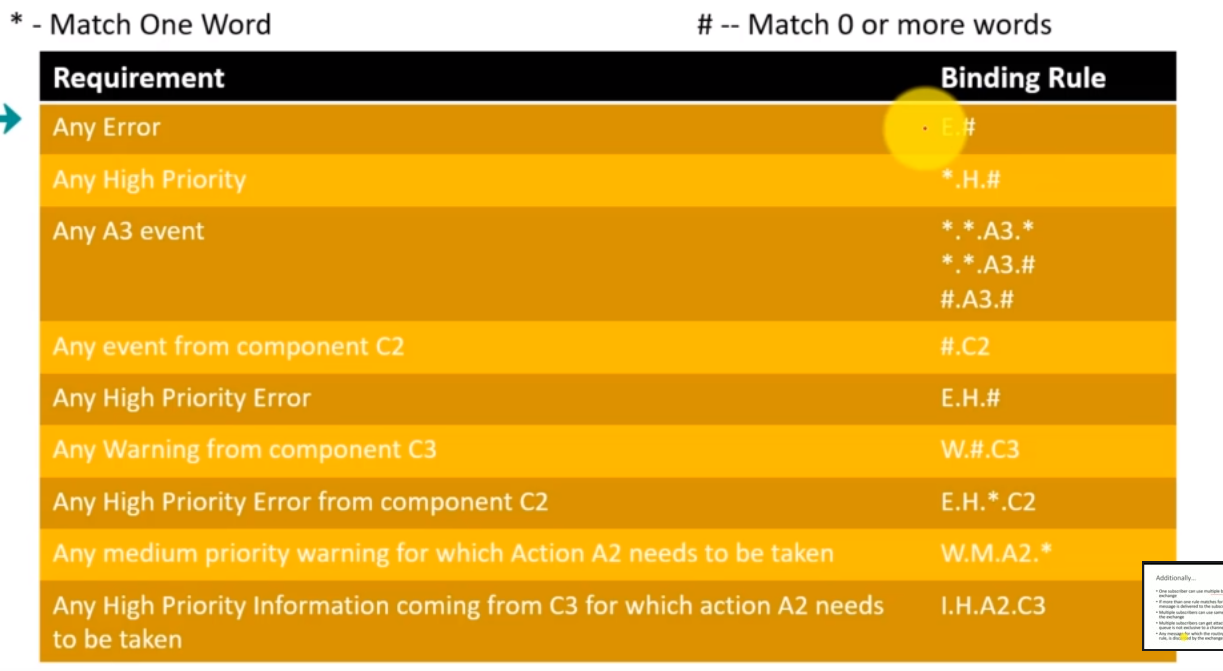
****

**Section 5: Topic Exchange --- Pattern-Based Routing:**

In this example we will consider messages based on multiple classifications:

* Severity
* Priority
* Actions to be take
* Generating Component

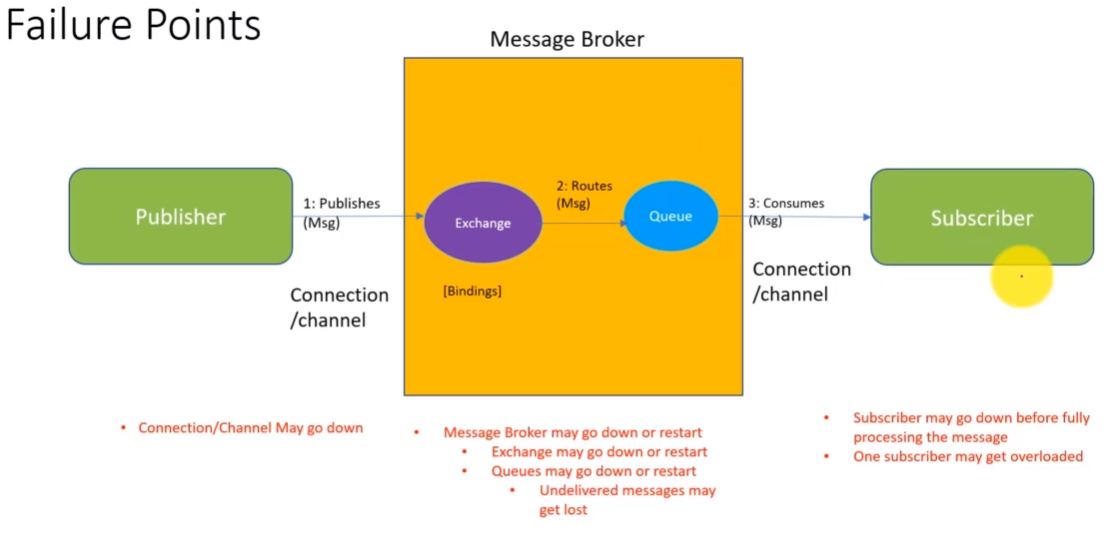
In this case our Routing Key will look like: *<Severity>.<Priority>.<Action>.<Component>*



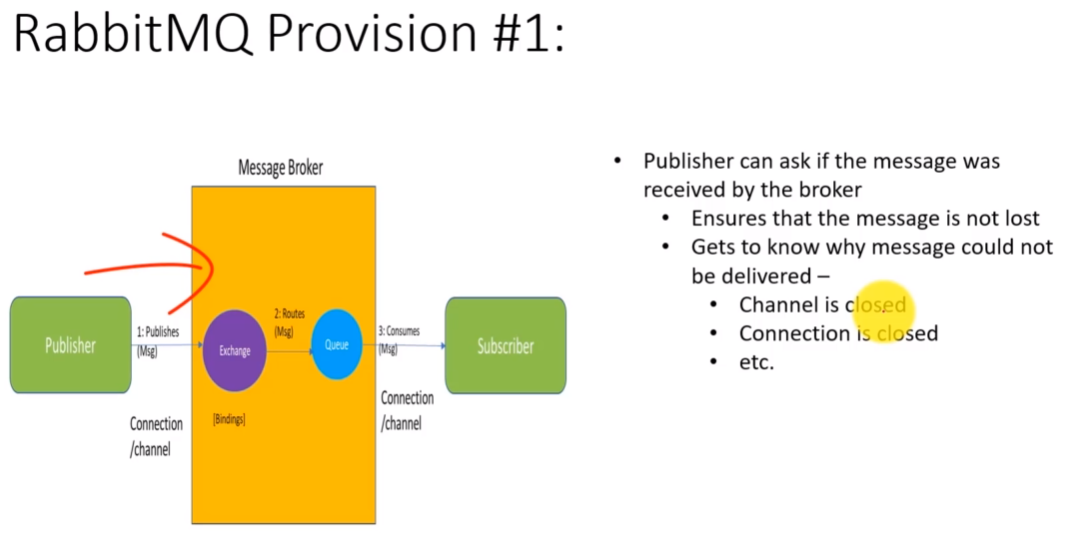
In the case that:

* a subscriber creates an exchange/queue/binding
* then a publisher attaches to that exchange/queue/binding infrastructure
  + publishes a message(s)
  + deletes the exchange
* and the subscriber is still there
* and the publisher attaches itself again
* the bindings **originally created by the subscriber** are now gone!
  + Obviously this is a problem we need to solve

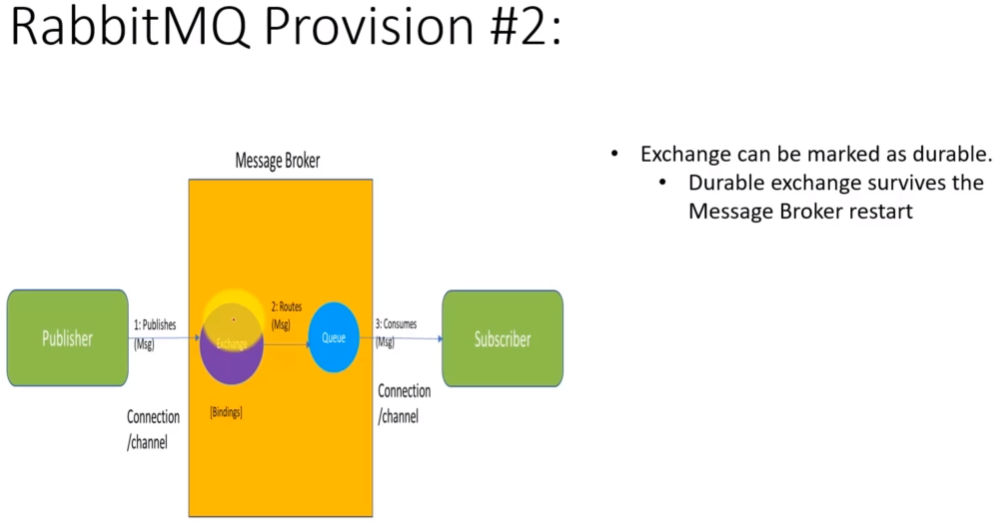
**Section 6: Reliable Communication**

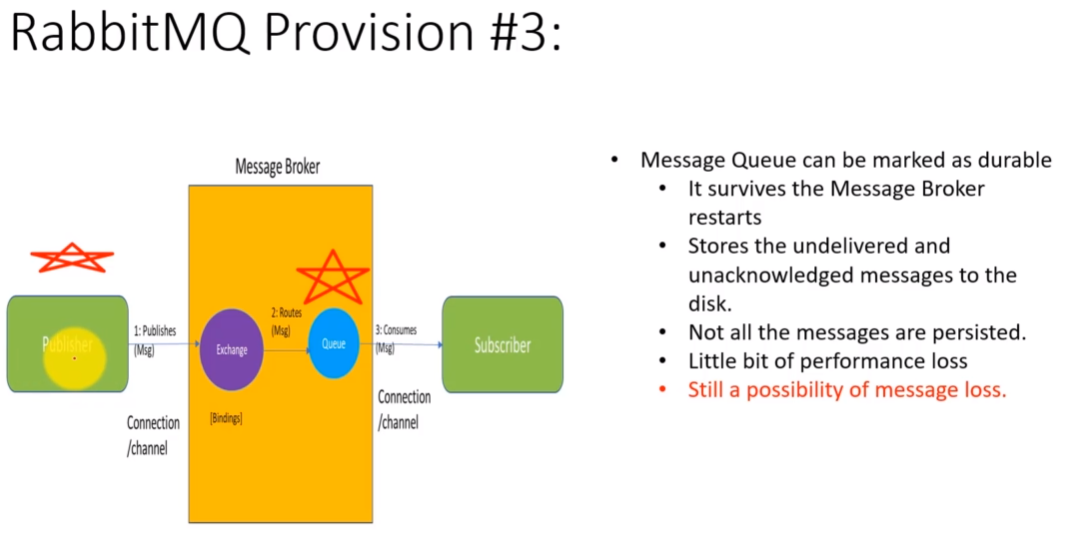


1) For such a publisher failure point:



2) For such a exchange failure point:

 3) For such a queue failure point:



For both 2) and 3) the “durability” solution persists the messages to disk in the background in order to achieve this durability. Something **important** to note here is that you can selectively determine which messages are persisted to disc by marking some/all of them with various “delivery\_mode” options as in the example code where delivery\_mode=2

was set for all messages.

4) For a subscriber failure point where the subscriber fails before finished processing of a message occurs:

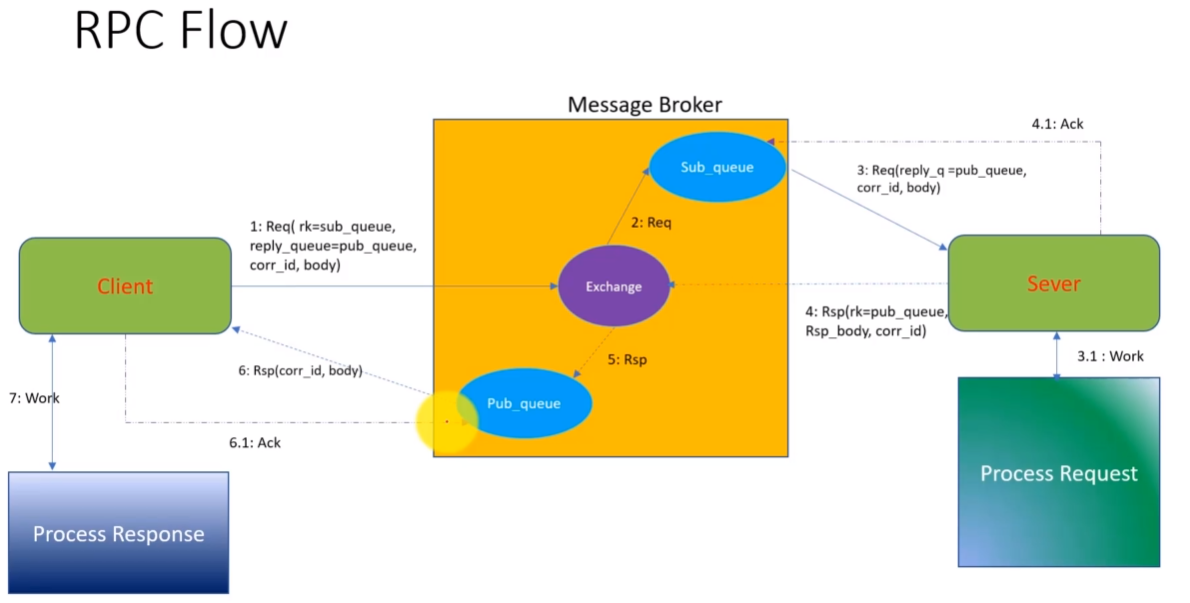
* The solution here is to avoid having the subscriber send its acknowledgement immediately after reception, but instead send the acknowledgement after having processed the message.
  + Generally queues only delete message after receiving such an acknowledgement

5) For a subscriber failure point where the subscriber is overwhelmed by message volume:

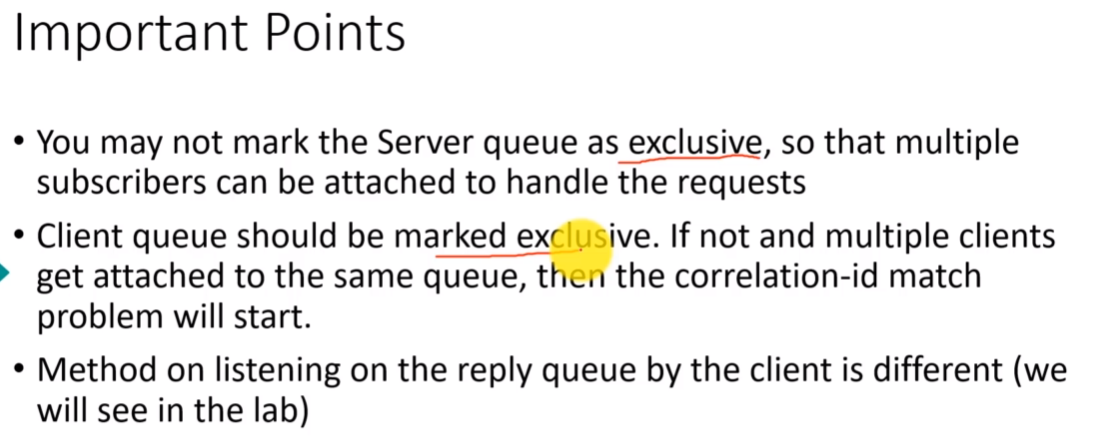
* You can configure a queue to only send more messages to a subscriber once it receives acknowledgements and to route messages to other subscribers in the interim during high volume message traffic

**Section 7: RPC – Remote Procedure Call**

This is a messaging variant where the publisher sends the request and then waits for a prescribed response from the subscriber to take its next action.



This kind of infrastructure implies the addition of another reply queue (here called *Pub\_queue*). It also adds a ***correlation\_id*** which is passed back and forth for message receipt integrity.

**Important Notes:**

In the publisher/client code for this section he implemented everything using an object-oriented approach because of the need to save various variables across function calls.