RabbitMQ

Anjali

I. What is RabbitMQ

These days software systems have evolved so much. Differernt applications have to communicate with each other in different ways. Therefore, there is a need for another communication layer for applications to communicate internally as well as with clients effectively. Different messages need to be delivered to different clients and applications and such an important functionality should not be a roadblock for a robust system. Thus the requirement of scalable systems led to the development of **message queues** and **message brokers**. A message broker is a system which can receive messages from more than one sender, then determine the correct destination and thus route the message along the right path. Thus the responsibility to deliver the message falls on such intermediate message oriented middlewares as opposed to sender or routers directly. Such softwares implement a very efficient message delivery system. One another important feature of message brokers is that they can transform and deliver messages in different formats. The architecture of message brokers is such hat they control and manage all incoming and outgoing messages. A message queue is just the basic data structure queue which holds messages. they provide the asynchronous feature required to build a distributed system.

RabbitMQ is a very powerful message broker software that implements the advanced message queuing protocol (AMQP) to solve a lot of messaging problems that exist in other messaging protocols that were used earlier. **AMQP** is just like an enhanced transport layer. AMQP gives standardization to different formats of routing and messages. To enable interoperability within different systems it defines a standard that can be applied to all interacting softwares.The whole operation of AMQP is called a message flow which gives the details of the message life cycle. The message flow starts with the producer creating the message and forwarding it to an exchange and ends at the consumer receiving that message.

RabbitMQ is an open source tool which is very easy to deploy and lightweight. It standardises messaging by defining producers, brokers and consumers. The advantage of coupling is that it increases loose coupling and scalability. RabbitMQ has good routing capabilities and is a reliable protocol. It can be scaled through clustering. The availability of resources is high in this protocol and the management and monitoring is efficient. It is secure. It is traceable and debugging is easy in it. The architecture of RabbitMQ is pluggable. Though RabbitMQ has different components, it acts like a single logical unit. The customer sees cluster as a single node. Also there is automatic metadata replication. Clustering is simple in RabbitMQ because of Erlang.

Message Brokers like RabbitMQ have a very important task of managing messages. They route the messages to the correct queue. They also control the queue size. Another responsibility of the broker is to ensure that the message has reached the receiver. As they behave like middlewares and take up the responsibility of directing a mesage to its intended destination, these softwares are called message oriented middlewares. They introduce the concept of abstraction and interoperability.They are the most important part of the whole messaging system and hence the most optimisation needs to be done here.

II. BASIC WORKING

So first as this protocol is used to send messages from sender to receiver, we need to understand how is the message seen by distributed system. **Producer** produces a message which is sent to a broker server like RabbitMQ.The least responsibility in this set up is of the producer as it does not hold the responsibility of delivering the message to the right or intended receiver.There is no limit as to how many messages can a producer send. Producers can be in the form of software applications, processes or software modules. Message is the main part of the messaging system based on which the whole architecture is planned. All the information is contained in the message. A **message** has two parts - payload and label.The payload indicates the data that one wants to transmit which could be jpeg or a json object or some other data and label is the thing which describes the data present in the payload. The label also helps in determining who should get a copy of the message sent.It is different from TCP in the sense that in TCP sender and receiver is already defined but here the label in the message helps RabbitMQ take care of the receivers. So producers create messages and label them for routing on various paths as intended. We can also say that a producer emits messages to exchange.

Now comes the role of a **consumer**. The last point of the entire messaging system are consumers. The job of consumers is simple. A consumer is attached to a broker server and it subscribes to a queue. The queue can be thought of as a named mailbox. Whenever a message arrives in a particular queue or mailbox to which consumer has subscribed, it receives that message from the queue. RabbitMQ enables this message exchange through queues and by checking subscribed consumers. The consumer only receives payload. It has no idea about the label and hence no idea about who the producer or sender of that message is. The consumer would only know of the producer if the producer has included that information in the message.

Next comes the **channel**. Whether the consumer wants to subscribe or publisher wants to publish to RabbitMQ, they both have to connect to it. They connect to rabbit through a channel. Connecting to rabbit creates a TCP connection between the producer or consumer and the rabbit broker. Once that connection is open and the subscriber is authenticated, then an AMQP channel is created. This channel is not a real one but a virtual one inside the already created TCP connection, and all the AMQP commands are issued over this channel. We set up a channel because each time establishing and tearing down a TCP connection is expensive and hence not efficient. The channel gives the same privacy as TCP but over a single connection which is a great improvement. There are three important parts of an AQMP message - **exchanges**, **queues** and **bindings**. The exchange is where the messages are published by the producers, queues are where the messages finally end up and received by the consumers and bindings describe how the messages are routed from exchange to particular queues.

III. **QUEUES**

Queues are like labeled mailboxes. This is where messages reach after the publisher publishes them on an exchange and they get lined up here while waiting to be received by the consumers. Consumers receive messages from queue in two ways. One way is by using the basic .consume command which is preferred when the consumer is processing many messages from a queue or needs to receive messages instantaneously from a queue as soon as they enter the queue. The above way is a way of persistent subscription to the queue.

Another way is when the consumer needs to receive only a single message from the queue. This can be done by .get command. As this is a request for a single message only one message will be received by the consumer. To receive the next message, the consumer has to use the

.get command again. The .get command should not be used in a loop as it is not efficient. In that case .consume command should be used.

Now, if there are no subscribers for a message, then the message waits in the queue. And as and when a consumer subscribe to the queue, the message is sent to the subscriber. When there are multiple consumers, the messages received by the queue are sent to the consumers in a round robin fashion. Each and every message that is received by the consumer has to be acknowledged. The acknowledgement can either be sent the consumer or it can be accomplished by the .ack command in AMQP. The acknowledgements are not for the producer but for the rabbit to ensure that the message has been delivered to the consumer. If the acknowledgement does not reach rabbit for any reason then the rabbit will consider that message to be undelivered and will deliver the message to the next subscribed consumer. The above situation can arise when a subscriber disconnects before acknowledging or the app crashes.

IV. EXCHANGES AND BINDINGS

The message reach the queue via the exchanges. Whenever publisher wants to deliver the message to the queue it does so by sending the message to the exchange. Then there are rules defined in RabbitMQ which determine to which queue should the message be delivered. These rules are known as routing keys. A queue is linked or bound to an exchange through a routing key. When a message is sent to the broker, it will have a routing queue which is matched by the RabbitMQ to the routing keys in the bindings. If the keys match then the message will be delivered to that particular queue else it will be discarded.

So far we have established that the broker routes the messages from **exchanges** to queues based on routing keys but there is one point that is missed which is how does rabbitMQ handle delivery to multiple queues. This is done by the different types of exchanges provided by the protcol. Or we can say that the message distribution depends on the exchange type. There are four types of exchange available - fanout, direct, topic and headers. A special kind of exchange is default.

The **default** exchange is created automatically for each queue by RabbitMQ.It works by comparing the routing key with the queue name. The default exchange makes it seemingly possible to send a message directly to a queue. RabbitMQ supports multiple messaging protocols.

Apart from default the above mentioned four exchanges implement different routing algorithms. The **direct** exchange implements a very simple algorithm. First a message queue binds to the exchange by using a routing key. Then the publisher publishes a message on the exchange with a routing key. If the routing key matches, then the message is delivered to the corresponding queue else it is discarded. The direct exchange must be implemented by the broker which should include a default exchange with an empty string as its name. And when a queue is declared it will be bounded to that exchange automatically using the queue name as the routing key.

The **header** uses the same algorithm as the direct exchange. It is the most powerful type of exchange in AMQP. This exchange sends messages based on the message headers, whether they match or not. This exchange does not pay attention to routing key. While creating exchanges, headers for exchanges are also specified. Thus message headers are matched with these headers. It is somewhat like the direct exchange, the only difference is it makes use of headers instead of routing keys.

The next type of exchange is **fanout** exchange.This exchange multicasts the message received to all the bound queues. The algorithm used here again is simple in that the message will be delivered to all the queues attached to the exchange. The message queue gets bound to the exchange with no arguments. Whenever a publisher sends the message to the exchange the message is passes on to the message queues withouth any conditions.

The last exchange to be discussed is the **topic** exchange. It is used in the situations where many different messages cab arrive at the same queue from different sources. This is a complex task to handle. Hence is done by this exchange. The flow of topic exchange type is as follows. At first, a message queue using a specific routing pattern binds to the exchange. Then publisher or producer sends a message to the exchange using a specific routing key. The message is passed to the message queue only on the condition that the routing pattern and the routing key match.

The term binding has been used quite a few times now, so let's look at what a binding means. A **binding** connects an exchange with a queue using a binding key and then an exchange compares the binding key with the routing key. Or it can be said that bindings are the rules that are used by exchanges to route messages to several message queues. Therefore bindings help in identifying the message queue in which the message should be sent. The **routing key** is the element that determines binding. Thus bindings give the answer to how the queues know which exchange they need to fetch messages from.This is where all the components of AMQP protocol end.

We can now look at how the broker is actually deployed in real world. RabbitMQ is shipped within a web application. The various functionalities of the software can be accessed very easily by setting it up in a desktop.

The various protocols that are supported by RabbitMQ are:

- AMQP
- MQPP
- HTTP
- STOMP
- web STOMP

This is how RabbitMQ works which is required in today's complicated software architectures and frameworks. The work of bringing together different softwares is achieved through messaging by using a message broker like RabbitMQ. Hence, another step towards integrating heterogeneous systems which might be running different operating systems and using different messaging formats.

REFERENCES

Videla, Alvaro (Apr. 2012). *RabbitMQ in Action.* https://sd.blackball.lv/library/RabbitMQ_in_Action_(2012).pdf.