A Quick Guide To Understanding RabbitMQ & AMQP

In the most recent projects I’ve worked on, the teams I’ve been a part of have decided on the direction of a [microservice](https://microservices.io/patterns/microservices.html) architecture pattern. The main motivation for this has been to overcome the challenges typically faced in [monolith](https://microservices.io/patterns/monolithic.html) architectures. However, when dealing with microservices, you’re likely to face the issue of communication between the different services that the system is comprised of. So how did we go about solving this? [**RabbitMQ**](https://www.rabbitmq.com/). But what on earth is RabbitMQ and how did it help solve this problem we faced? Let’s start with the basics of understanding the underlying technology of RabbitMQ.

**Message Queuing**

Message queuing facilitates effective communication between applications by sending messages. It also serves as a temporary shelter for messages by storing them while the destination application is busy or not connected.

**What is a message?**

It is a data transporter between a sender application and a receiver application. What kind of data does the message consist of? It could be a signal to inform an application to start processing a task, or tell an application about the completion of a task by another application. The message could hold crucial information required by the application for its own processing.

**Basic Architecture of a Message Queue**

All messages are generated by a **Producer** Application and pushed into a Message Queue. This process is known as *Enqueuing*. Pushed messages will stay in this queue until a **Consumer** Application connects and fetches this message. This process is known as *Dequeuing*. Both the Enqueue and Dequeue process are carried out independently by the Producer and Consumer applications, and because of this independent nature of producer and consumer, we’ve got the option to hold a message in a message queue, waiting for a consumer to fetch it.

So this process of a message getting pushed to a message queue by a producer application and getting consumed by a consumer application is referred to as **Message Queueing**.

**Message Brokers**

Message brokers take care of the connections between applications. Based on the requirement, a bi-directional connection will be created between each application system and message broker. Messages are then transported via this connection. Message brokers act as a hub to route appropriate messages to appropriate destinations. In a nutshell, a Message Broker routes appropriate messages to their appropriate destinations, similar to the way a Telephone Switching Office works.

**AMQP Implementation in RabbitMQ**

[AMQP](https://www.amqp.org/) is the core protocol for [RabbitMQ](https://www.rabbitmq.com/) (a Message Broker), but it also supports STORM, MQTT and HTTP through the use of plugins.

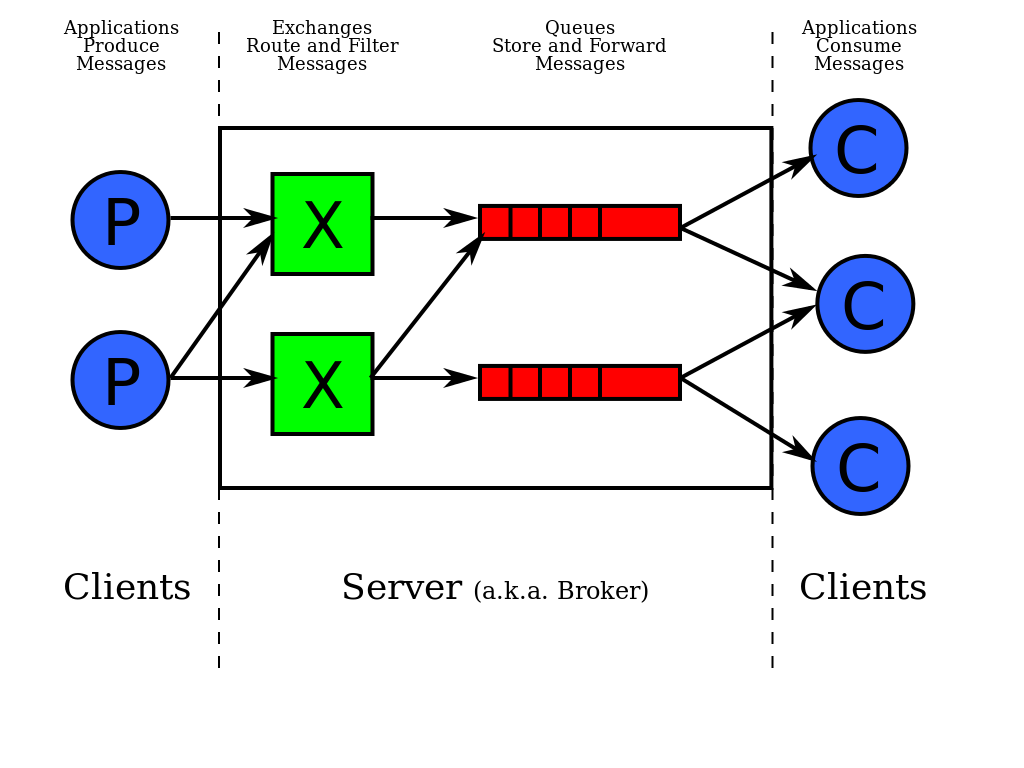
STOMP — a simple text based messaging protocol

MQTT — is a binary protocol known for its lightweight messaging

HTTP — you’re probably familiar with this one. It is not a messaging protocol, but management plugins in RabbitMQ use HTTP to send and receive messages.

[Several companies](https://stackshare.io/rabbitmq) have RabbitMQ as a message broker with the AMQP implementation.

**AMQP Protocol**



AMQP Model

It is a protocol primarily used for message oriented middleware. Some of the impressive features it offers are message orientation, queuing, reliability, security and routing.

AMQP mandates the behaviour of message publisher and message consumer for seamless transportation of messages between different applications built by different vendors in different programming languages.

**Message flow in RabbitMQ**

There are three AMQP entities in RabbitMQ:

* Exchange
* Binding
* Queues

Messages published by a publisher are first received by the Exchange in RabbitMQ, then Exchanges will distribute message copies to Queues. To send appropriate messages to the appropriate queues, rules called Bindings are used.

Since Queues are consumer facing, it’s crucial that the Exchanges route messages to the appropriate Queues, and Bindings play an important role in this.

Once the messages reach Queues, messages can be delivered to the appropriate consumer or consumers can fetch the messages from the Queues.

When publishing a message, a publisher can pass attributes along with the message. The attributes can be used by RabbitMQ and the consumer applications.

What happens when a message fails to deliver to a consumer? This can occur due to a network or application failure. If either of these failures was to occur, our system could potentially lose the message forever.

To address this issue, AMQP has a delivery acknowledgement mechanism in place. So a message will not be completely removed from a Queue unless we send a positive acknowledgment from the consumer. In the case of a negative acknowledgment, the message can be re-sent to the consumer or it can be dropped depending on the configuration settings by the publisher when sending the message.

In AMQP 0–9–1, the applications has more control over the message it sends, the application can define the exchange type that has to receive the message, and the queue where the message has to be saved. It can also define the routing scheme which binds the Exchange with Queues and it can control the message fetching and distribution logic on the Queues.