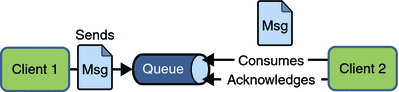
Message queuing makes it possible for applications to communicate **asynchronously**, by sending messages to each other via a queue. A message queue provides temporary storage between the sender and the receiver so that the sender can keep operating without interruption when the destination program is busy or not connected. Asynchronous processing allows a task to call a service and move on to the next task while the service processes the request at its own pace.

Both JMS and AMQP enable message-based communication in different contexts— JMS is tailored for Java applications, providing a standardized way for Java programs to communicate asynchronously. AMQP, on the other hand, is an open standard protocol that supports cross-platform messaging, making it ideal for connecting systems written in different programming languages and/or running on different operating systems.

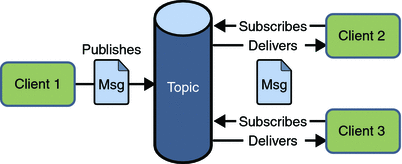
**1. Java Message Service (JMS):** It is an application program interface (API) that supports communication for the computer in a network. It is a powerful API used for receiving the messages generated during communication. Specifically, there are two types of models provided by JMS.

1. **Point to Point**: As the name itself suggests it is a one-to-one messaging mechanism, where

the sender sends a message to a single receiver. The message is available to the receiver application once it’s ready and until then the message is stored in the queue. The most important part of it is there are zero dependencies with respect to time between both the sender and the receiver application.



1. **Publish and Subscribe**: This messaging mechanism is very uniquely designed by JMS. For Example, one reader subscribes to one blog where the person is interested. Now there may be several people interested in a particular blog. And they subscribe/register to that blog. Now once a new post or topic is published on the blog, all the registered readers will get an update. This messaging model is called Publish and Subscribe.



**ActiveMQ** is an open-source message broker written in Java and consists of a full Java Message Service (JMS) client. On the other hand, **RabbitMQ** is an open-source message-broker that originally implemented the Advanced Message Queuing Protocol. So, this is the fundamental difference between ActiveMQ and RabbitMQ.

**2. Advanced Message Queuing Protocol (AMQP):** It is a protocol that is used for communication between applications. It is a lightweight protocol which supports the applications for its transfer of data. This protocol is used for it scalability and modularity with the technologies.

**Core Components of AMQP:**

1. **Exchanges**: The exchange is responsible for fetching messages and properly arranging them in the appropriate queue.
2. **Channel**: A channel is a multiplexed virtual connection between AMQP peers that is built into an existing connection.
3. **Message** **Queue**: Holds messages until they are consumed. It is bound to exchanges with **binding keys**.
4. **Binding**: Bindings are a set of predetermined instructions for queuing and exchanging. It manages message transmission and delivery.
5. **Virtual Host**: Vhost is a platform that provides isolation capabilities within the broker. Multiple vhosts may be functional at the same time, depending on the users and their access rights.

The **Advanced Message Queuing Protocol (AMQP)** is implemented by the open-source message broker **RabbitMQ.**

**AMQP Message Flow:**

Producer → Exchange → [Bindings + Routing Key] → Queue → Consumer

**There are currently three types of messaging framework:**

**Messaging Queue Frameworks** – The traditional message queue paradigm, which is to be used only when there is a fixed end-to-end messaging system to support it.

Example: Kafka, RabbitMQ, ActiveMQ, Amazon SQS etc.

**Distributed Messaging Pub-Sub Frameworks** – Publish–subscribe is a sibling of the message queue paradigm. This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data.

Example: Kafka, Apache Pulsar, Amazon SNS + SQS etc.

**Distributed Stream Processing Frameworks** – Stream processing frameworks are runtime libraries which help developers write code to process streaming data, without dealing with lower-level streaming mechanics.

Example: Kafka Stream, Apache Flink etc.

These three types of messaging frameworks and a comparison of the specific platforms available in today’s market.

* Use **Flink** for powerful real-time and event-time processing.
* Use **Kafka Streams** for lightweight, in-app stream processing.
* Use **Spark** when you need unified batch + streaming pipelines.
* Use **Dataflow** (Beam) for cloud-native, serverless solutions.
* Use **Storm/Heron** for ultra-low latency use cases.

### **Push or Pull Delivery**

Most [message queues](https://aws.amazon.com/sqs/) provide both push and pull options for retrieving messages. Pull means continuously querying the queue for new messages. Push means that a consumer is notified when a message is available (this is also called [Pub/Sub messaging](https://aws.amazon.com/pub-sub-messaging/)). You can also use long-polling to allow pulls to wait a specified amount of time for new messages to arrive before completing.

### **Exactly-Once Delivery**

When duplicates can't be tolerated, [FIFO (first-in-first-out) message queues](https://aws.amazon.com/sqs/) will make sure that each message is delivered exactly once (and only once) by filtering out duplicates automatically.

| **Feature** | **AMQP** | **JMS** | **MQTT** | **STOMP** |
| --- | --- | --- | --- | --- |
| **Type** | Protocol | Java API | Lightweight protocol | Simple text protocol |
| **Interoperable** | Yes | No (Java-only) | Yes | Yes |
| **Reliability** | High | Varies | Medium | Low–Medium |
| **Use Case** | Enterprise-grade apps | Java EE apps | IoT, sensors | Simple web messaging |

**SQS: Pull Based (can hold msg to 1 min to 14 days, default days is 4 days)**

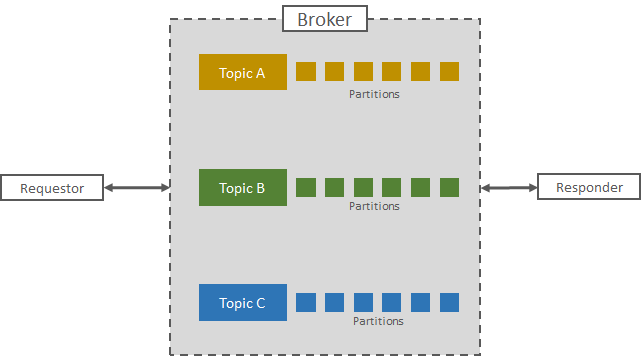
Standard Queue: 256KB Message: order not guaranteed, can deliver msg more than twice

FIFO Queue: guaranteed to deliver one. 300 Transactions per Second.

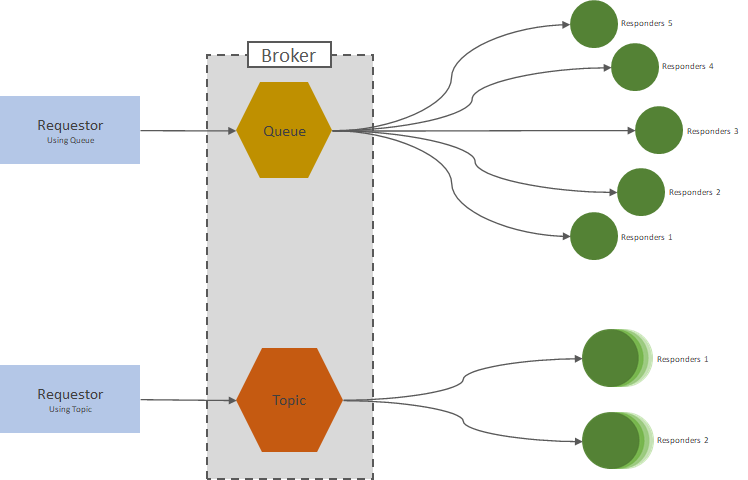
SNS: Push Based

<https://digitalvarys.com/kafka-vs-activemq-vs-rabbitmq-vs-zeromq/>

How Kafka Works



ActiveMQ supports two methods. One is Point-to-Point Communication which is called Messaging Queue that will collect all the messages from Requestors as Queue and distributes to the Responder in [Round Robin Scheduling](https://en.wikipedia.org/wiki/Round-robin_scheduling). Another Method is the Point-to-Many model called the Publish-Subscribe model. On this, Requester will publish the message in the topic, and Responder will subscribe to the Topic and start receiving the message asynchronously.



**RabbitMQ** has four major components.

* Requestor,
* Exchange,
* Queues, and
* Responder.

In this, Requestor is the service or application which sends a message to get a response from other services. Responder is the service that will receive the message and give the response. Exchange is where the logic of routing messages takes place. It will decide in which way the message will be routed to the responder. For example, Fanout, Topic Based, Direct Routing, Header exchange, and more. The queue is the buffer like a system that will have the messages in sequence and maintain a certain mechanism to distribute the Message.

