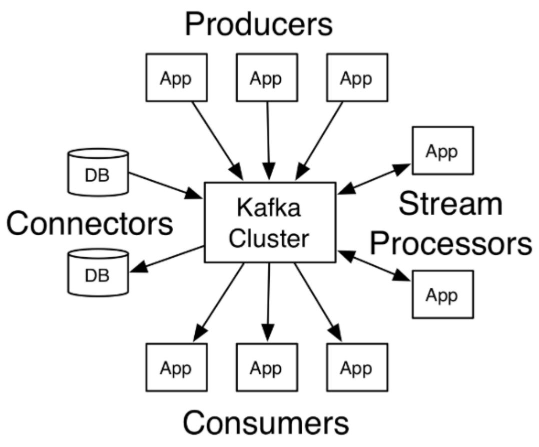
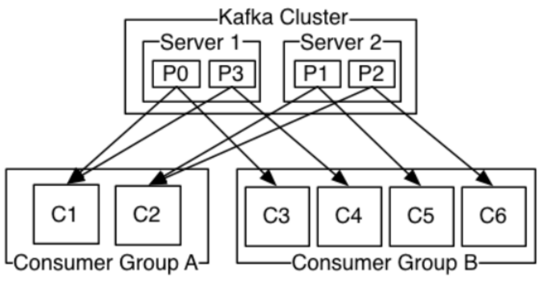
Quick Intro to Kafka

Kafka is a messaging system. From the ground up it has been designed to provide high throughput, fast performance, scalability and high availability.

Lets quickly go over the **basics of Kafka**.

- ***Producers*** of the messages ***publishes*** to the ***Topics***
- ***Consumers subscribes*** to the ***Topics***
- ***Messages*** are ***array of bytes***. They can be **JSON objects**, **Strings** etc
- ***Topics*** are ***logs of messages***
- **Kafka** is run as **a *Cluster*** on one or more servers that can span multiple datacenters.
  Each of servers is called **a *Broker***





Kafka has four core APIs:

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The Streams API allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

In Kafka the communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. This protocol is versioned and maintains backwards compatibility with older version. We provide a Java client for Kafka, but clients are available in many languages.

MQ/JMS Versus Kafka

1. No concept of Queue in Kafka i.e., no P2P model

In Kafka there is no concept of Queue and hence no send or receive for putting/getting messages from the queue. Publish-subscribe is the only paradigm available as a messaging model. Producers of the messages *Publish*a message to the *Topic* and *Consumer* receives messages by *Subscribing* to the topic. This publish-subscribe paradigm is very similar between MQ/JMS and Kafka - the difference is under the covers that we will discuss next.

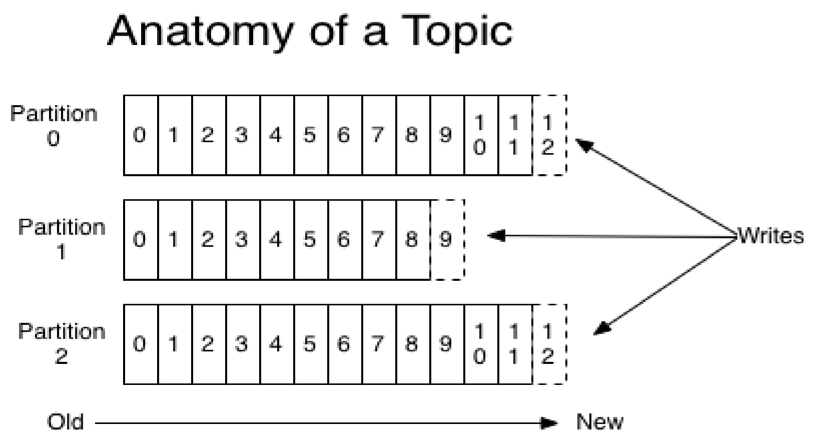
2. Message *Persistence*

Typical JMS providers (IBM MQ, Rabbit MQ, Active MQ ..) implement the topics in a such way that the messages published to the topic are sent to a common storage (memory or/and persistent store) from where they are picked up by the subscribers. In MQ/JMS systems once the message is read it is removed from the storage and is no more available. Kafka retains the messages even after all the subscribers have read the message. The rentention period is a configurable parameter.

In a typical MQ/JMS consumer implementation, the message is deleted by the messaging system on receiving an ACK/Commit. If for some reason the message gets processed but fails before the ACK/Commit, it would lead to message being read more than once. This problem has been addressed by Kafka by way of message retention and state management based on the consumer offset.

3. Topic partitioning

Kafka has implemented the topics as partitioned logs. A partition is an ordered, immutable sequence of messages that is continually appended to. This is similar to *database log,* for that reason the partition is also referred to as the *commit-log.* This is one of the biggest differences between MQ/JMS and Kafka. The partitioning of the topic leads to its high throughput (and parallelism).



- Partitions for the same topic are distributed across multiple brokers in the cluster
- Partitions are replicated across multiple servers; number of replicas is a configurable parameter
- Each Partition has one server as a ***leader*** and a number of servers as ***followers***
- Each Server acts a leader for some of its partitions and as a follower for some other
- The Producers are responsible for choosing which message to assign to which partition within the topic based on key assigned to message.

5. Message sequencing

In MQ/JMS there is no gurantee that the messages will be received in a sequence in which they were sent. In Kafka though the sequence is maintained at a partition level. In other words, if the topic is configured with a single partition then the messages are received in the same order that they were sent in.

6. Message reads

The consumer of the messages in Kafka issues a fetch request to the broker leading the partition it wants to consume. As part of the fetch, consumer specifies the offset from which the message in the log is read from. This is very different from the MQ/JMS messaging system where First In First Out (FIFO) is the way messages are read off the queue/topic. The other thing that happens is that with offset based control, the consumer can re-read the same message which is not possible in MQ/JMS (yes you can do it with browse but that is not what it is intended for).

This rewinding mechanism can be very handy in some situation. E.g., if you received a batch of messages and processed it with buggy code, you may fix the code and re-run the processing on the messages by resetting the offset.

7. Load balancing

In the case of MQ/JMS the load balancing required messaging systems to be designed using some clustering mechanism and the onus of distributing the load across the cluster members was on the producer sending the messages. The Kafka nodes publish the ***metadata*** which tells the producer which servers are alive in the cluster, where the leader for the partitions are. This allows the client to send message to the appropriate server (and partition) thus distributing the message load across the cluster members.

8. Automatic failover & High availability

Traditional MQ/JMS implementations did not have the concept of message replication, but some systems built it over a period of time; those replication features at most times were not leveraged in favor of simplicity. In Kafka, as decribed earlier the messages are replicated (leader-followers) for each topic's partitions across a configurable number of servers. This inherently leads to an architecture that provides automatic failover to replica thus leading to high availability.