What is Kafka?

<https://kafka.apache.org/intro>

Kafka is a platform for event streaming. It is the digital equivalent of the human body’s central nervous system.

Technically speaking, event streaming is the practice of

* **capturing** data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events;
* **storing** these event streams durably for later retrieval; manipulating, processing
* **reacting** to the event streams in real-time as well as retrospectively;
* **routing** the event streams to different destination technologies as needed.

Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Event streaming use cases:

* To process **payments and financial transactions** in real-time, such as in stock exchanges, banks, and insurances.
* To track and **monitor cars, trucks, fleets, and shipments** in real-time, such as in logistics and the automotive industry.
* To continuously **capture and analyze sensor data from IoT devices** or other equipment, such as in factories and wind parks.
* …

Kafka is a platform for event streaming. It boils down to the following operations:

1. To **publish** (write) and **subscribe to** (read) streams of events
2. To **store** events durably and reliably for however long you want
3. To **process** events as they occur or retrospectively

Properties:

* Distributed
* Highly scalable
* Elastic
* Fault-tolerant
* Secure

Interface of Kafka

* **Events**: recording of “something happened”. Example event:
  + Event key: "Alice"
  + Event value: "Made a payment of $200 to Bob"
  + Event timestamp: "Jun. 25, 2020 at 2:06 p.m."
* **Producers**: client applications that publish (write) events to Kafka
* **Consumers**: those that subscribe to (read and process) events in Kafka
* **Topics**: A common theme for events. Topics are like folders and events are like the files

Design overview

* It uses a client-server architecture and communicating using a high-performance TCP protocol <https://kafka.apache.org/protocol.html#The_Messages_Fetch>
* Servers are distributed across data centres and regions. Some servers form the storage layer, called the **brokers**
* Events can be produced and consumed by clients. There are support for many languages (Java, Scala, Python …) as well as REST APIs
* When events happened, they will be stored in Kafka. The duration of that storage can be set. Events are not by default deleted after consumption.
* Events are organised into **topics**. Topics can have >=0 producers and consumers.
* Topics are **partitioned**. The storage is spread across different brokers. Each event with the same event key (a customer ID, say) are stored in the same partition. There are guarantees in-place that make sure the events will be read in the right order
  + This makes the system scalable as a client application can read/write from/to many brokers at the same time, which is imperative as the size of the topic grows
* Producers never wait for consumers, which also helps with scalability
* Kafka uses its own underlying protocol to let servers and clients communicate
* Data is **replicated** across brokers for resiliency
* Uses **Apache Zookeeper** to coordinate the cluster

Design details

Motivation

Kafka is designed to work with event streaming for a large company

* High throughput
* Low-latency delivery
* Partitioned, distributed, real-time processing of feeds: to create new and derived feeds
* Fault-tolerance

Persistence

* The design team found out that using sequential storage of files on disk is as fast or even faster than using in-memory cache. This makes the persistent data structure feasible
* And having a persistent queue built on 1+TB SATA drives give O(1) read and write. The idea is to use a lot of these cheap drives. This allows data to be persistent, which has a lot of advantages. In a nutshell, kafka is trying to approach the event streaming problem using a logging solution

Efficiency

* We want writing and reading to be fast and cheap
* Small I/O problem -> batch messages together to be written and read in chunks
* Byte copying in OS -> use Linux `sendfile` system call to push data out of pagecache directly to a socket
* Network constraints -> end-to-end batch compression

Producer

* Load balancing: producer sends data to leader of the partition. All broker can answer queries on metadata about which servers are alive and which is the leader
* Asynchronous send: batching of data to send as one single packet

Consumer

* Pull: designers of Kafka adopted a pull model where consumers will pull data from the producers (as opposed to a push model where producers will publish data to all consumers)
  + This makes batch size optimal, as consumer simply pull all remaining data from the position of the last message pulled from the log
  + Another advantage is that we don’t have to worry about the rate of producing overwhelming the rate of consumption by the consumer (a sort of DoS attack)
  + A disadvantage is that if no data is available from the producer, the consumer is stuck in a “busy-wait”. A solution is to have a parameter in the pull request that allow the consumer to request to bock in a “long pool”, avoiding the busy wait.
* Consumer position:
  + Having producer and consumer agree on where is the last message sent is not so trivial. It requires some ACK mechanism
  + The idea is that topics are divided into a set of partitions, and each is consumed by exactly one consumer in the subscribing consumer group at any given time. This means the position of the consumer in each partition is a single integer, making ACK easier to do
  + This also allow consumers to rewind on previous messages. Even though this breaks the queue invariant, it is surprisingly useful
* Offline data load: load data to offline systems like Hadoop or data warehouses

Message delivery semantics

* Common guarantee options: at most once, at least once, exactly once
* To achieve exactly once
  + Producer has a notion of “commit” where if a message is committed, it stays in the log stream. And each message is sent with a unique sequence number so if a message fails to be received, resent message has a different sequence number, hence making at least once becoming exactly once
  + Consumer saves the position of the message *with* the message content itself, effectively making them share fate. This makes the consumption exactly once

Replication

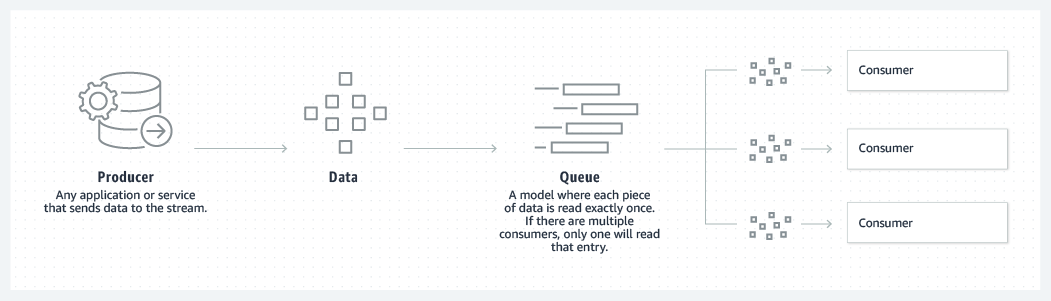
Implementation of Kafka

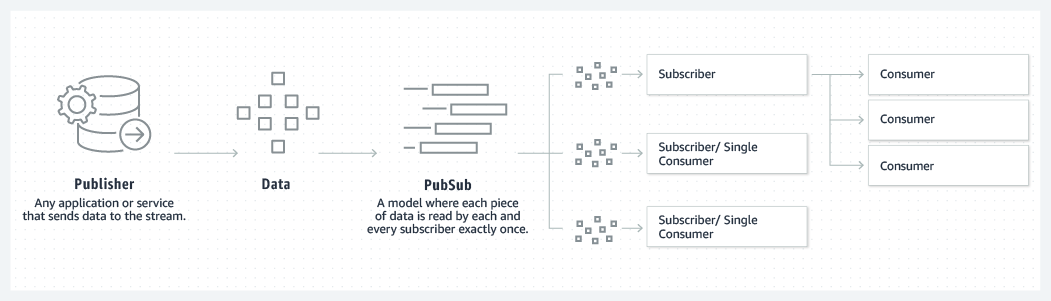
Configs

<https://kafka.apache.org/documentation/#configuration>

You would need to write specific configs for the brokers etc.

Messaging protocols:





Queue: each piece is read exactly once

PubSub: each piece is read exactly once by each subscriber

* Logs (an ordered list of events) are partitioned to allowed multiple subscribers to consume the message in their own pace, hence increasing scalability