Kafka: a Distributed Messaging System for Log Processing

Log processing has become a critical component of the data pipeline for consumer internet companies. Kafka is a distributed messaging system that is developed for collecting and delivering high volumes of log data with low latency. Many early systems for log processing required log files to be downloaded offline to perform the analysis, whereas Kafka is capable of performing log analysis real time by exploiting advantages of distributed messaging system.

A stream of messages of a particular type is defined by a topic. A producer can publish messages to a topic. The published messages are then stored at a set of servers called brokers. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers. To subscribe to a topic, a consumer first creates one or more message streams for the topic. The messages published to that topic will be evenly distributed into these sub-streams. Each message stream provides iterator interface which iterates over every message and processes the payload.

To improve the efficiency of Kafka, it builds on 3 design principles – simple storage, efficient transfer and stateless broker. Kafka has a very simple storage layout. Each topic corresponds to a logical log. Physically, a log is implemented as a set of segment files of approximately the same size (e.g., 1GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For efficient transfer, Kafka does not cache messages in process. Hence it has very little overhead in garbage collection. A typical approach to sending bytes from a local file to a remote socket involves reading data from the storage media to the page cache in an OS, copying data in the page cache to an application buffer, copying application buffer to another kernel buffer, sending the kernel buffer to the socket.

Kafka has the concept of consumer groups. Each consumer group consists of one or more consumers that jointly consume a set of subscribed topics, i.e., each message is delivered to only one of the consumers within the group. Different consumer groups each independently consume the full set of subscribed messages and no coordination is needed across consumer groups. The consumers within the same group can be in different processes or on different machines. Kafka makes a partition within a topic the smallest unit of parallelism. This means that at any given time, all messages from one partition are consumed only by a single consumer within each consumer group. Additionally Kafka does not have any central ‘master’ node. Adding a master would have complicated things as then we need to worry about master failure. Instead Kafka uses highly available consensus service Zookeper for coordination. Kafka uses Zookeeper for detecting the addition and the removal of brokers and consumers, triggering a rebalance process in each consumer when the above events happen, and maintaining the consumption relationship and keeping track of the consumed offset of each partition. Kafka only guarantees at least once delivery i.e duplicate messages might be delivered. If an application cares about duplicates then it should handle it themselves.

Like a messaging system, Kafka employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. By focusing on log processing applications, Kafka achieves much higher throughput than conventional messaging systems. It also provides integrated distributed support and can scale out. Kafka is being successfully used by LinkedIn for both online and offline application.