Review - *Kafka*: A Distributed Messaging System for Log Processing

Name: Kunal Baweja
UNI: kb2896
COMS E6998 Cloud Computing & Big Data (Fall 2016) October 14, 2016

Introduction
Kafka is a distributed and scalable messaging system to collect and deliver large volumes of log data, up to multiple terabytes with low latency at high throughput rates.

This data needs to be analyzed efficiently and effectively for better user engagement, activities, targeted marketing and system performances. Most of the analytics solutions involve taking the log data off the production servers, storing on offline storages and then processing over that data, however sometimes there is a need to support real-time applications with low latency, which in turn requires quick processing of logs and that is what *Kafka* is aimed at.

Motivation
The motivation to develop *Kafka* at LinkedIn stemmed from their need to process log data, offline as well as online, quickly and as per the paper the off-the-shelf solutions were an overkill for the task due to unnecessary features such as message acknowledgement for each log message, lack of focus towards throughput and lack of distributed systems support.

Architecture and Design Principles

- *Kafka* defines a stream of messages of particular type as a *topic* to which a *producer* can publish log messages. These messages are stored at a set of servers called *brokers* and a *consumer* can subscribe to one or more topics from the brokers and pull messages for consumption(processing).
- *Kafka's* API design tends to be quite simple, wherein a message is defined to containt just a paylod of bytes and the user can choose any serialization method to encode the messages. For efficiency reasons, the producer can publish multiple messages in a single publish request.
- For subscribing and pulling messages for a topic, a consumer first creates one or more message streams for the topic and the messages published to that topic are evenly distributed among these streams, where each stream provides an iterator interface over the continual stream of messages produced. This iterator, unlike traditional iterators, blocks if there are no more messages to be consumed.
- A distributed *Kafka* architecture consists of multiple brokers, a topic is divided into partitions for load balancing and each broker stores one or more of these partitions. Multiple producers and consumers can publish messages at the same time.

Efficiency of *Kafka*
Kafka employs the following design decisions to make the system efficient:

1. **Simple Storage**: Each partition of a topic corresponds to a logical log. Physically a log is implemented as a set of segment files of approximately the same size and every time a producer publishes a message, the broker appends it to the last segment file and these files are flushed to disk only after a configurable number of log messages have been recorded or

a pre determined time has elapsed. The messages are exposed to consumers only after they have been flushed.

To avoid the overhead of maintaining auxiliary index structures to map message ids to individual messages, *Kafka* addresses messages by their logical offset in the logs. So basically, successive message ids are constructed by adding the length of a message to its id, the first one starting from zero.

Messages are always consumed sequentially by a consumer from a given partition on a broker node and acknowledgement for receiving the complete message is sent by sending back the offset number as seen by consumer. This essentially signals the broker to keep the next batch of messages from the partition ready for consumption by the consumer, where the batch size is specified by the consumer process.

On the broker nodes, a sorted list of offsets is maintained in memory where each value represents the first message id of the partitions and upon receivng consumer request, broker nodes quickly search through this list to locate the partition containing the requested messages.

2. **Efficient Transfer**: Similar to the publishig of messages, the pulling of messages by consumers also involves fetching multiple messages at a time, although the API seems to be iterating over individual messages. These data transfers typically range into few hundred kilobytes and prove to be efficient as the communication overhead is reduced.

3. **Stateless broker**: *Kafka* does not tracks the amount of information consumed by each consumer. This reduces a lot of complexity usually involved in other similar systems and the storage overhead, bytat the expense of making it a bit tricky to delete a log message as the brokers do not know whether all consumers have consumed the messages required by them. This problem is solved by *Kafka* by using a time based Service Level Agreement for retention poliy wherein a message is automatically deleted if it has been retained in the broker node for longer than a specified time, like 7 days.

Distributed Coordination

In the distributed setting of *Kafka*, each producer can publish a message to either a randomly selected partition or determined by a partitioning key and a partitioning function.

For consumption of messages, *Kafka* introduces the concept of *consumer groups* where each group consists of onr or more consumers that consume a set of subscribed topics jointly i.e each message is delivered to only one of the consumers within the group. Different consumer groups independetly consume the full set of subscribed messages and no coordination is needed between the groups. But within consumer group, messages stored in the brokers are evenly distributed among the consumers without too much overhead. This distribution involves making following decisions:

1. A partition within a topic is considered the smalles unit of parallelism i.e at any given time all messages from a partition are consumed only by a single consumer from a consumer group. This avoids the need to co-ordinate access of multiple consumers to the same partition via locking mechanisms and other overheads.

2. There is no centralised system/master node to control the distributed servers, brokers and consumers, instead the consumers co-ordinate within themselves, each acting as an equal node and using the higlhy available Zookeeper service for consensus. Brokers do not require

co-ordination.

Delivery Guarantees
The *Kafka* distributed messaging system provides following guarantees:

1. In general *Kafka* guarantees at-least-once delivery. In most cases a message is delivered exactly once to a consumer group, however if a consumer crashes then the consumer process that takes over in its place might receive the same message more than once. This duplication of messages needs to be handled by the application as per its requirements.
2. *Kafka* also guarantees that messages from a partition are delivered in order to a consumer. However there is no guarantee on the ordering of messages coming from different partitions.
3. *Kafka* stores a CRC for each message in the log. In case of any I/O error on the broker, *Kafka* runs a recovery process to remove those messages with inconsistent CRCs.
4. In case a broker fails, any messages stored on it but not yet consumed are also lost if the damage is irrepairable, otherwise there is still a guarantee to recover those logs and bring back up the node.

Performance Analysis
In comparison to ActiveMQ and RabbitMQ message systems, *Kafka* broker nodes performed far ahead, reporting at least several orders of magnitude of more messages published per second in comparison to ActiveMQ system and at least twice more than the messages published by RabbitMQ. These performance enchancements are dure to following reasons:

- *Kafka* producer does not wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
- *Kafka* has a more efficient storage format, i.e treating them as bytes with user choice for encoding. On average the overhead is 9 bytes, whereas ActiveMQ reports overhead of 144 bytes.

The consumer nodes of *Kafka* reported consuming upto 22,000 messages per second, about four times more than ActiveMQ and RabbitMQ. This performance enchancement can be attributed to following reasons:

- Fewer bytes are transferred between broker and consumer process because of the efficient storage format of *Kafka*.
- The broker in both ActiveMQ and RabbitMQ had to maintain the delivery state of each message, keeping the threads busy in verification, this is not required in *Kafka* and hence better performance.
- *Kafka* also reduces the transmission overhead by sending multiple messages batched together at once.