Kafka Lite

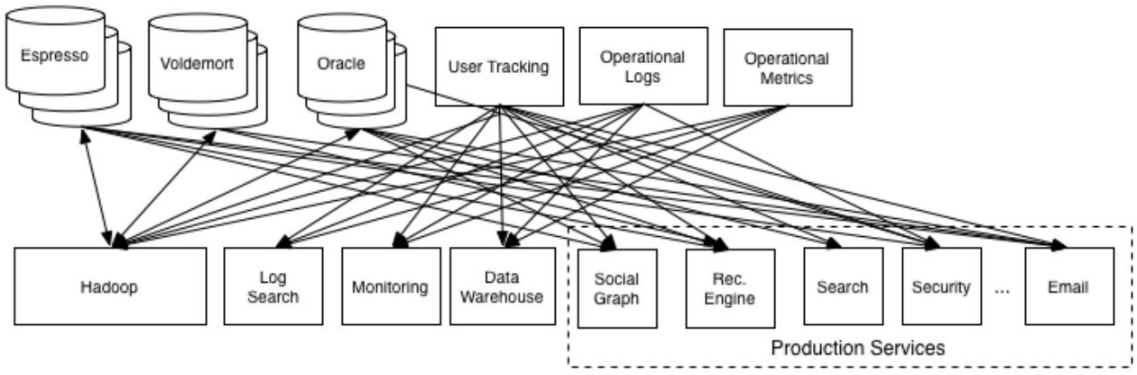
Kathy Wang (l9z9a), Dante Cerron (q7z9a), Jeanie Suparman (j9x9a), Edward Ho (z1e9)

Introduction

Kafka Lite is inspired from a widely used distributed streaming platform called Kafka (https://kafka.apache.org). Kafka is used to build real-time data pipeline and data streaming across multiple services. The service is highly scalable and fault-tolerant. Inspired by this, we are aiming to build a smaller version of Kafka that focuses on the highly reliable publishers and subscribers model, while still trying to guarantee the same ordering and resiliency that Kafka offers. We are not going to implement Streams API and Connector API that Kafka offers. We are also not going to guarantee the same real-time performance that Kafka guarantees.

Problem

In the real world, we are dealing with platforms that rely on multiple services for storing and processing data. These services need to be able to send and receive data from each other. For example, data repositories need to be able to gather data that comes from multiple data systems; furthermore, they also need to be able to communicate back to these systems. Soon enough, we are dealing with a system that looks like this:



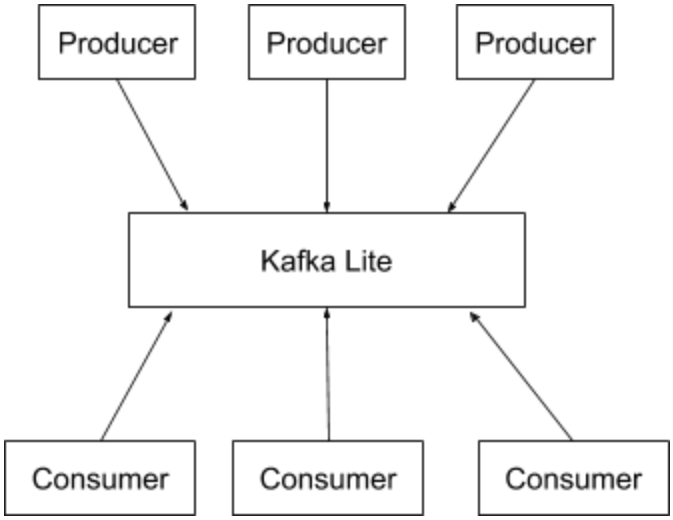
(Image taken from: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying)

where we essentially need a team of developers just to maintain the different channels that exist between the different nodes. The amount of maintenance effort increases as the number of services increases. Thus, we want a data delivery solution that doesn't grows in complexity as we increase the number of services that depend on it.

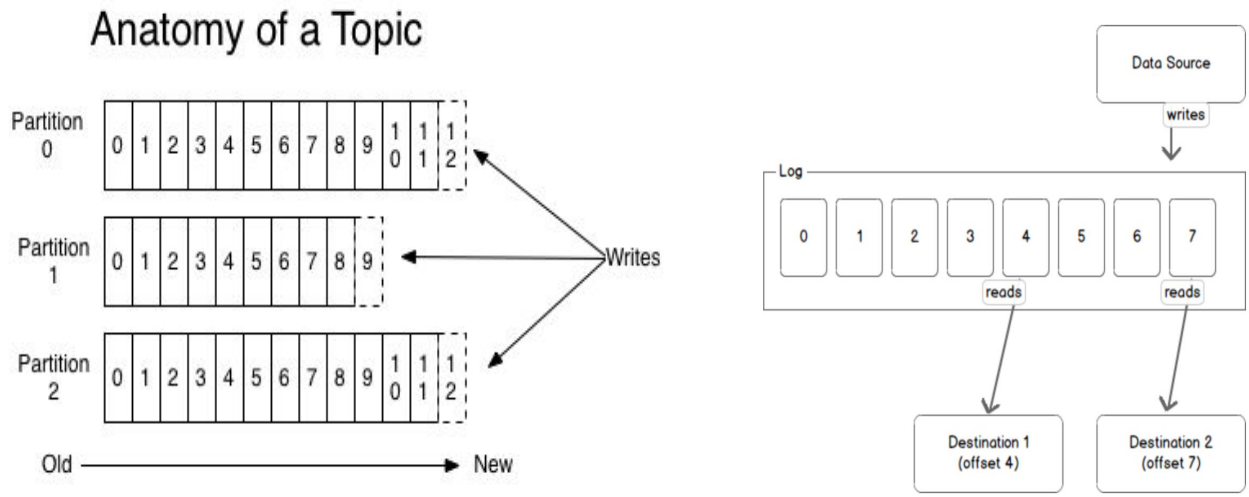
On top of that, this streaming platform needs to be highly resilient and fault tolerant. Any downtime will result in lost of valuable communications between system components. We want to be able to guarantee durability for all the messages that are received.

Approach/Solution

We are going to solve this problem by having a central messaging bus. All published data are going to be stored in Kafka Lite and any data consumption will be pulled from it. This way, the publishers of data do not have to maintain all the different channels it has with its subscribers. It only has to let Kafka Lite know that it is publishing a message. This keeps the system simple even with the increased number of publishers and subscribers.



Kafka messages are organized into different categories called *topics*. We have *Producers* who push messages to a Kafka topic and *Consumers* who pull messages from a Kafka topic. Messages in a topic are split up into different partitions. Each of these partitions are ordered and immutable logs that are append-only. Each of the logs in a partition are uniquely identified with a sequential id that we will call *offset*.

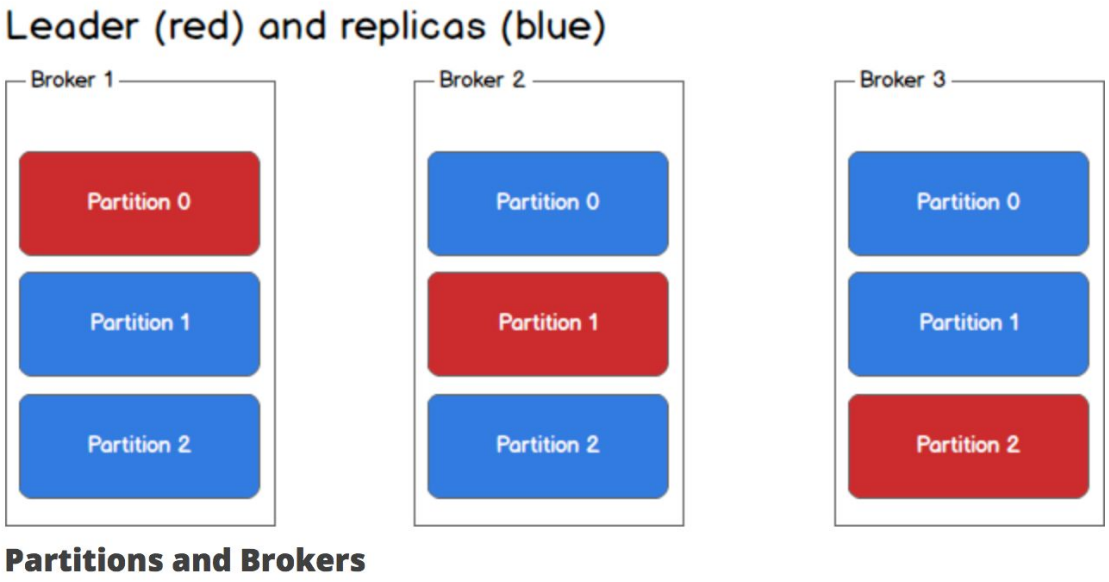


Note: the number in the list corresponds to offset
(Image taken from: https://sookocheff.com/post/kafka/kafka-in-a-nutshell/)

Kafka itself is a distributed system that runs on a cluster of nodes. We call each of the nodes a *broker*. Each broker has a replica of each partition in a topic, though only one

broker can be the *leader* of a certain partition. The rest of the brokers will be the *followers* of that partition. Any writes to a partition have to go to the leader, while followers simply replicate. Thus, the advantage of having multiple partitions is that the leader responsibility can be shared between different brokers and load balances the write operations that come in.

An example of a leader - follower configuration:



(Image taken from: https://sookocheff.com/post/kafka/kafka-in-a-nutshell/)

Maintaining Consistency and Resiliency

When a producer writes to the leader of a partition, the leader replicates the message to it's followers in a primary backup manner. Once all followers have acknowledged the message, we consider the message as **committed**. The producer will only receive acknowledgement from Kafka Lite once the message is committed. Since there can only be one producer that is writing to the leader at any given time, this will prevent write conflicts and guarantee message ordering in a given partition.

One of the guarantees we want to make is that any committed messages will survive as long as one of the in-sync brokers is alive. *We define in-sync brokers as brokers whose follower partitions are up-to-date with its leader partition*. Consumers can only consume committed messages, so they don't have to worry about data inconsistency if the partition leader fails.

When the leader fails, we want to avoid exposing consumer and producers to downtime, so a new leader of a partition must be elected in a way that all committed operations are still present in the new leader. Since the leader will not commit the operation until all in-sync brokers have acknowledged it, we can be assured that any follower who replace it will also have the most up to date messages. We will go into this in more detail in the Leader Election section.

Kafka Lite Guarantees

- Messages sent by the same producer to a topic partition will be appended to the log in the order that they are sent
- Consumer will see messages in the order that they are stored
- Ordering is guaranteed for a specific partition, not necessarily for the whole topic
- For a topic that is replicated N times, it will survive N-1 node/broker failures

These guarantees hold as long as we have at most one producer and one consumer for a given partition.

Producer

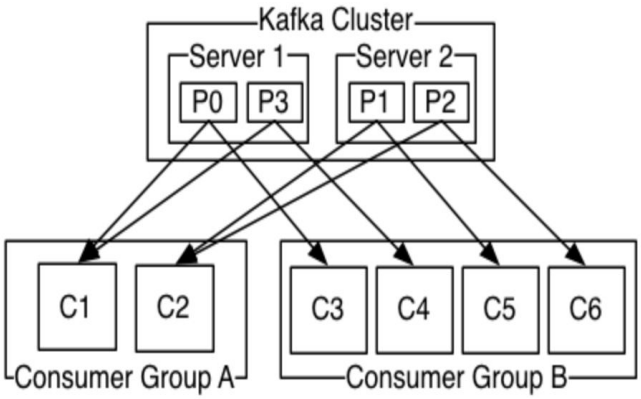
The Producer API allows applications to send streams of data to a topic. Through the API, an application can choose which partition it wants to produce the message to. Our producer library will determine who the leader for that partition is and send the message to that leader. Producer writes to a leader, which provides load balancing production in order for the writes to be serviced by a different brokers. To guarantee consistency, the producer waits and blocks until all messages have been committed before acknowledging them.

The producer is like a pool of buffer space and holds records that haven't been sent to the server. When the leader of a partition is unavailable, we set a number of retries that the Producer API can attempt to retry their action. Once the number of retries that the producer attempts exceeds the retry limit, an exception is thrown.

Similarly to **primary backup**, the *primary* handles and executes requests, which in this case is the leader. The leader then streams copies to *backups*, which are the followers, and wait for a confirmation from the followers before sending an ACK to the client.

Consumer

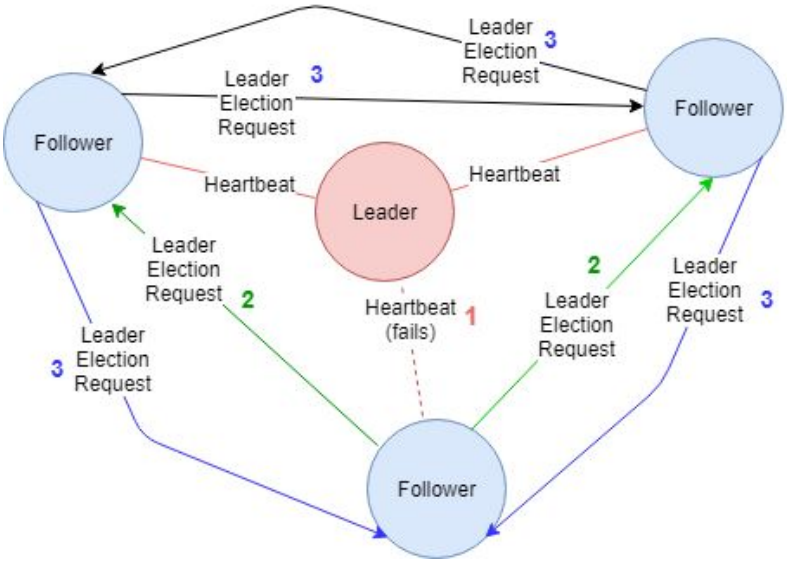
A consumer pull messages from a partition in a Kafka topic. Consumer can belong to a consumer group, where each consumer instance is pulling messages from a unique partition of a topic. A consumer group as a whole consumes from all partitions in a topic. When we have the same number of consumer instances in a group as the number of partitions, each of the instances will consume from exactly one partition. When there are more partitions than consumer instances, then an instance can consume from more than one partition.



(Image taken from: https://kafka.apache.org/intro*)*

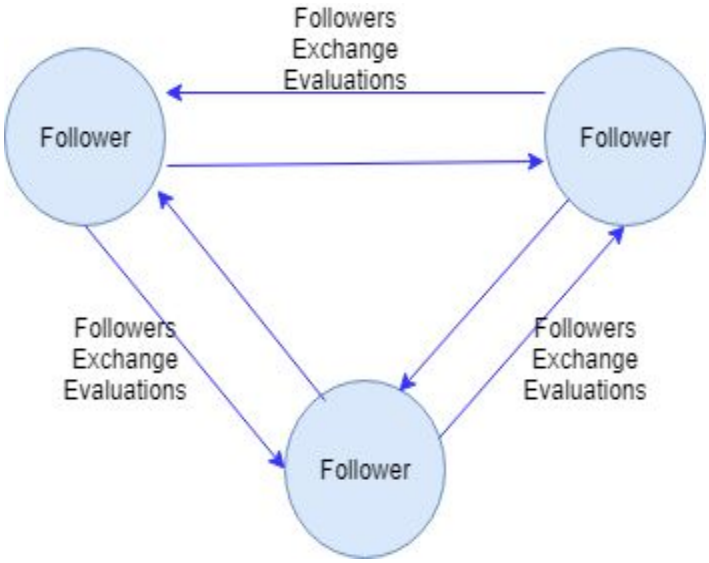
Consumer can read from a specific offset in the partition. It can even re-read offset that has been read before as long as the Kafka events are still stored. Though when a consumer crashes, it needs a way to remember where it was at in the partition once it restarted. Our consumer implementation will guarantee "at-least-once" read semantics. Consumers will read data, process it, then log it's own offset afterwards. If it crashes and recovers afterwards, it may try to process the data again which can cause duplication, but at least we know that we are not losing any data in the case of failure.

Leader Election

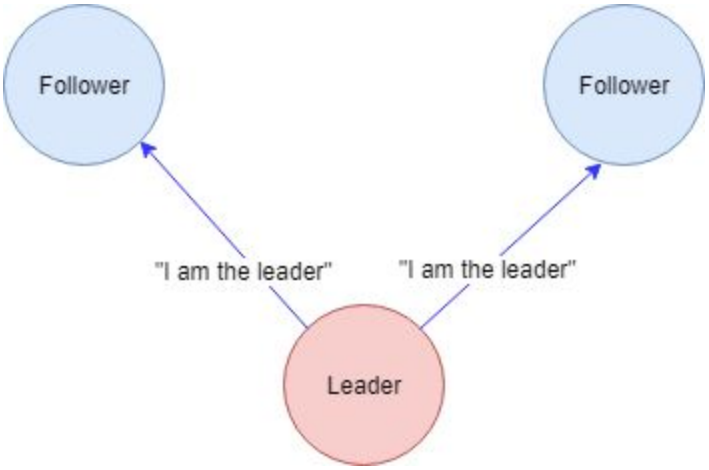


Step One:
From the point of view of a follower, the Leader Election protocol start when either the heartbeat they had with the leader fails (1 in the above diagram) or they receive a Leader Election Request from another follower (2). Follower election requests contain the current state of the partition of the follower that sent it, and will be used for comparison by other followers. A follower will stop waiting for Leader Election requests when either they have received one from all the followers they know about, or after a certain amount of time.

We will make the assumption here that if waiting for the request from a follower times out, they will also not be the most up to date follower, or there is another up to date follower that can also be selected.



Step Two:
Each follower will then compare the received requests and individually evaluate which follower actually has the most up to date replica, and communicate this evaluation to the other followers. They then wait to receive the same evaluation from other followers. In the case of ties, the most up to date will be decided based on who has the least leader partitions, and then on an arbitrary basis, but one that will be consistent across followers that are trying to decide between the same tied set.



Step Three:
Once all followers have received the evaluations, they will see which broker appears the most often. This broker will send a message to the other followers that *it* is now the leader for that partition, and normal functionality continues.

Here we make the assumption that the same broker will appear the most often from the point of view of each follower, so only one broker will send the "I am the leader" message.

Assumptions

There are a number of assumptions we will make for our implementation of Kafka Lite:

- Producers will not misbehave (holding the write lock of a partition forever)
- Brokers are trusted and non-malicious
    - Our Kafka clusters and consumer groups are operating within a closed environment and the only connections that should be made between brokers are defined by the server.
- We can detect a node failure using a timeout of 2s.
- Reliable network, no packet loss
- No nodes will be behind a NAT or firewall which will make them unreachable to each other

Testing (on Azure)

Potential test cases:

- 1 Producer, 1 Consumer, 1 Broker
- 1 Producer, 1 Consumer, 3 Brokers
- 2 Producers, 2 Consumers, 3 Brokers
- 2 Producers, 2 Consumers, 3 Brokers (1 follower fails, should not affect the other brokers)
- 2 Producers, 2 Consumers, 3 Brokers (leader fails, leader re-election)
- 2 Producers, 2 Consumers, 3 Brokers (all nodes fail, error returned to consumers and producers)

We will deploy and test our implementation on Azure and through automated unit/integration tests. We will deploy using a bash script that will be run on each Azure VM. (Thanks Jeanie!)

Timeline

Deadline	Task
March 2	● Proposal Draft due
March 9	● Proposal Final Report due ● The whole team start making API and Design decisions
March 12	● Make sure Design is finalized ● Implement Broker-Broker RPC connection and heartbeat [Jeanie] ● Data definitions for Broker started and log Kafka data [Dante] ● Producer API that is exposed to client and Producer-Broker Communication [Kathy] ● Consumer API and Consumer Group [Edward]
March 16	● Sync up with group to discuss any changes or issues ● Implement Leader-Follower replication for writes [Dante] ● Broker-Broker are able to pinpoint who are the partition leaders [Jeanie] ● Producer is able to store records and write to partitions in the topic [Kathy] ● Consumer is able to read from partitions [Edward]
March 23	● Email to assigned TA to schedule meeting and discuss project status ● Deploy to Azure VMs ● Work on Leader Election and improving resiliency ● Write test cases to cover the 'Testing' section
March 26	● Start integration testing (Producer-Brokers, Broker-Broker, Broker-Consumer) [everyone] ● Failure testing (disconnecting brokers, producers, consumers, etc) [everyone] ● Discuss and start extra credit work (possibly)
April 6	● Handin project code and final reports

	• ZeroBug Deadline
April 9-20	• Project demos

SWOT Analysis

Strengths: 1. Project is based on existing system which has been implemented in several languages 2. Numerous documentation for the project 3. Widely used	Weaknesses: 1. Complexity on server side (since we are dealing with multiple server replication) 2. Difficulty in deploying as we have a number of different nodes acting different roles
Opportunities: 1. Project can be extended to support the Kafka Streams API, Connector API and many other possible ideas 2. Multiple use cases, for example: a. Log Aggregation b. Commit log c. Messaging	Threats: 1. Commitment to exams or assignments from other courses may interfere with progress 2. Low morale due to low A2 and Project 1 marks

References

https://kafka.apache.org/
https://sookocheff.com/post/kafka/kafka-in-a-nutshell/
https://kafka.apache.org/090/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html