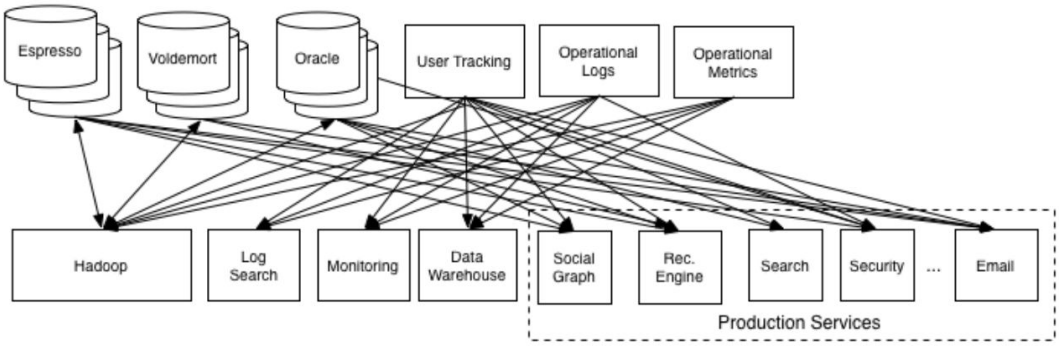
Kafka Lite Report

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

1 Introduction

1.1 Problem

In the real world, we are dealing with platforms that rely on multiple services that 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 and communicate back to these systems. Soon enough, we are dealing with a system that looks like this:

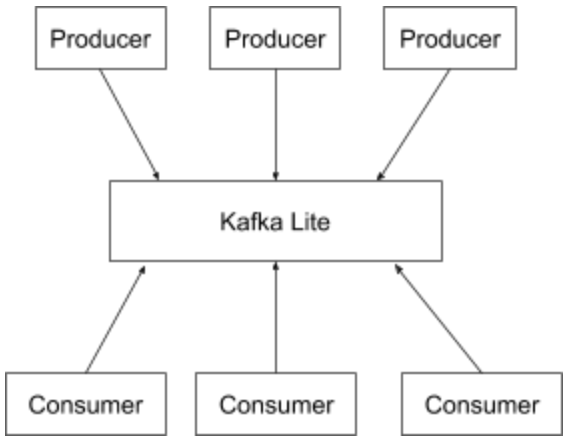


(Image taken from: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying)

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. In this project, we are going to build a data delivery solution that doesn't grows in complexity as we increase the number of services. 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.

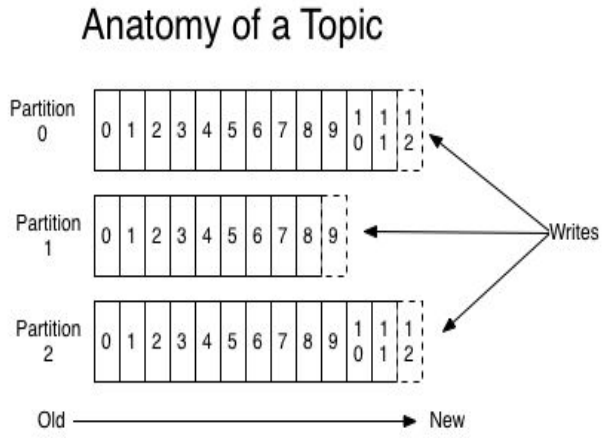
1.2 Approach/Solution

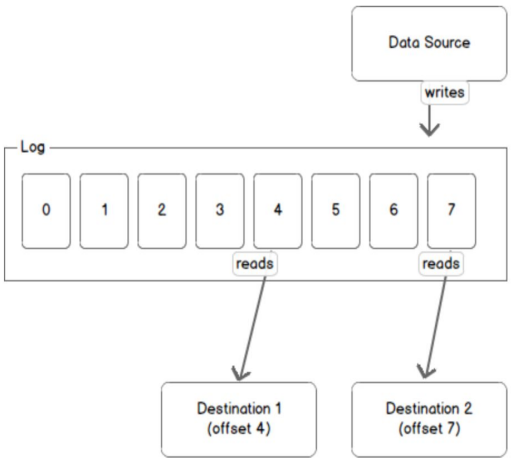
We are going to solve this problem by having a central messaging bus. All published data are going to be stored in Kafka Lite and any data consumption will be pulled from it. This way, the publishers of data do not have to maintain all the different channels it has with its subscribers; they only have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



system simple even with the number of publishers and subscribers increases.

As shown on the diagram above, we have *Producers* who push messages to Kafka Lite and *Consumers* who pull messages from it. Kafka Lite messages are organized into different partitions in a single *topic*. The decision on which message goes to which topic is decided by the producer application. Note that Kafka Lite only supports a single topic unlike Apache Kafka. Each of these partitions are ordered and immutable logs that are append-only. Each of the log message in a partition are uniquely identified with a sequential id that we will call *offset*.



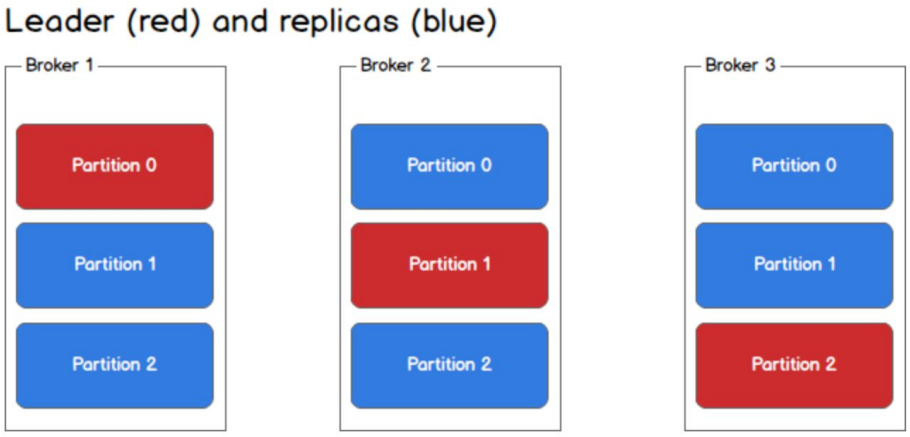


Note: the number in the list corresponds to offset

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

Kafka Lite 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 every partitions, though only one broker can be the *leader* of a partition. The rest of the brokers will be the *followers* of that partition. Any writes to a partition have to go through the leader, while followers simply replicate in a primary-backup manner. 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 this configuration:

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



Partitions and Brokers

1.3 Maintaining Consistency and Resiliency

Once all followers have replicated a 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 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 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.

1.4 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.

2 Design

2.1 Consumer API

func MountConsumerGroup(serverAddr string, numConsumers int)
Returns (cg ConsumerAPI, err error)

- Returns an instance of *ConsumerAPI* and *error*
- Mounts a consumer group and returns an instance of the consumer lib API where you can call the below functions

ConsumerAPI.Consume(partitionIds []uint8, offset uint, n uint)
Returns (msgs map[uint8][][]byte, err error)

- Returns an array of **n** msgs for clientId starting at **offset** on a random broker for each partitionId given
- Creates a manifest where partitionIds are evenly distributed among consumers. Starts consumers and gives them each a diff manifest. Consumers will connect to broker and read from partitionIds in their manifest. Then, get result and update result map with messages.

ConsumerAPI.Poll(timeout int)
Returns []ConsumerRecord

- Returns all messages for every partition and also the last offset read for each one in the form of ConsumerRecord
- Takes in a timeout (in seconds) and tries to read all messages for all partitions within the given timeout

ConsumerAPI.Unmount()
Returns error

- Cleans up everything, returns nil if successful
- Closes connection to the server

2.2 Producer API

Mounting the Producer
func MountProducer(serverAddr string) (ProducerInstance, error)

- Producer (given server address by client) creates an RPC connection with the server to prepare for grabbing brokers on write.

Adding a Message
func (p *ProducerInstance) AddMessage(partitionId int, data [256]byte) error

- Producer will call GetLeaderIp to contact Broker to figure out who is the leader of the partitionId. It can throw InvalidPartitionIdError for any invalid partition Ids
- After receiving *leaderIp* from GetLeaderIp we make another call WriteToPartition(given *partitionId, data, and leaderIp*) to write the message to the partition
- Throws an BrokerDisconnectError or nil

func (p *ProducerInstance) AddMessageTimeout(partitionID int, data[256]byte, timeout int) error

- Will call AddMessage multiple times with the given partitionId and data, but it will not return error as long as the operation has not timed out. This gives the cluster a chance to complete leader elections

Writing to a Partition in the Topic (See Shiviz Diagram below)
func (B *Broker) WriteToPartition(args *WriteArgs, reply *string) error
A call to Broker to write to partition:

- The leader streams copies to *backups*, which are the followers, and wait for a confirmation from the followers before sending an ACK to the client.
- (1st Phase) Check with peers (aka other brokers) to determine what they perceive the leaderIp is for the given partitionId
- Once all the ACKs returns to the original broker, we check for the majority vote for leaderIp to make sure that the leaderIP for partitions has remained consistent (this is mostly just to get another safeguard out of the ack part of the first phase of 2PC

- (2nd Phase) We do a check for whether the majority voted leaderIp is the same as the original Broker's address before writing to disk locally and replicating the write to the other Brokers

Retrieving the Leader Ip address of a Partition
func (B *Broker) GetLeaderIp(partitionId *int, reply *string) error

- A call to contact the Broker to return the ip address of the leader of the partition
- Throw LeaderNotFoundError if unable to determine leader IP or it is an empty string

Unmounts the Producer
func (p *ProducerInstance) UnMount() error

- Cleans up everything, returns nil if successfully closing the connection between Producer and Server

2.3 Broker

Once registered with the server, a broker has rpc connections, and maintains heartbeat with the server and every other broker.

Partition Leader Initialization
how to assign which broker leads what partition upon initialization?

- if you are the only broker, you lead all partitions
- if you join a network and another broker already exist: (See Shiviz diagram below)
    - first, request the partition state (who is leading what partition) from any active broker (they should all have the same state)
    - if there is a broker that leads more than half of the partitions, send a request to that broker to see if it wants to share leadership load
    - the receiver will then share the leadership load and broadcast this to all peers

Writing Messages

- Each message is written to append-only logs we call partitions. Sorted by partition ID, they are always replicated amongst a configurable number of peers. (config.json file)
- Replication is done via a 2 Phase Commit protocol with Primary-Backup

Heartbeat

- Server gets started up first, then brokers get started up with the serverAddr as an argument
- The new broker gets the addresses of all the live brokers from the server, and all the live brokers get the address of the new broker
- The brokers must heartbeat the server to stay in the topology, and brokers heartbeat each other too (all-to-all)

Leader Election (See Shiviz Diagram)

- For any broker, if its heartbeat with a peer times out, it will check if this peer was a leader of any partition. If it is, the leader election protocol is triggered (can happen for more than one partition simultaneously)
- Step 1: Ask for election data from peers (the length of the partition that just lost its leader, how many partitions they lead, their ip;port and their brokerID). This will also trigger the leader election procedure on peers if they haven't started it.
- Step 2: Decide who the leader should be based on the following criteria (in order of importance)
    - The most up to date (longest) partition
    - The least number of leader partitions (in case of ties)
    - The broker with the smallest ID number (in the case of another tie, this criterium will be the consistent tie breaker)
- Step 3: Broadcast this decision, and wait to receive election decision from all alive peers
- Step 4: Make sure that the majority of the peers agree on who the new leader is, otherwise retry (go back to step 1). We are ensuring consensus safety while compromising liveliness.
- Step 5: Now we know who the new leader is. If a given broker is the leader, it will broadcast this to the whole broker network (more like flooding than broadcast, it is more redundant but there will be a smaller likelihood of inconsistent partition state)

Kafka File Structure

Directory name: broker id (Taken from CLI input)

File name: {partition id}.kafka

3 GoVector and Shiviz Integration

3.1 Partition Leader Initialization (Including requesting to share leader load)

3.2 Leader Election

3.3 Write Replication Protocol

4 Evaluation

Due to time constraints, we are unfortunately unable to implement intensive automated testing that includes integration testing between Producers, Brokers and Consumers. We have been mostly logging our operations in order to see that it has the expected behaviour. For communications between brokers, we have tested that multiple new brokers joining a network will try to update their local Kafka, request for partition metadata, and request to share leadership role if applicable. We have also seen that failing a broker that is leading a partition will trigger a leader election. We saw that even when the leader is leading more than one partitions, a consensus on the new leader is still achieved. We have also tried failing all brokers and leaving one alive, and the system still behaves as expected. Due to a number of protocols relying on a majority vote, we require an odd number of brokers in a cluster.

For the Producer side, we have tested out that failures on the broker side will not be visible from the application side as the write operation will simply retry until it timeouts. A write operation that is received properly by a broker will be replicated amongst its peers as we have seen in the Shiviz diagram above. We also check the kafka files to see that these new messages are indeed replicated for all the brokers. For the Consumer side, we tested the polling functionality by running it manually. We saw that the consumer receives messages from all partitions up to that point of time, and wait for new messages to come in. When we run a client that produces message afterwards, we can also see that these new messages are received by the consumer application. In addition, we are able to read N messages given partitionId(s) and an offset to start from.

We are also dealing with a lot of different nodes with different roles in order to run Kafka Lite, deploying them and running them on Azure can be a pain. To deal with this complexity, we wrote a number of shell scripts that uses Azure CLI tools to automate these workflows.

Our work in progress includes stress testing our system on Azure VMs to handle multiple node failures, joins, or overall performance for a large number of nodes; as well as further improving aspects our demo (i.e. the web application and HTTP server). In addition, more testing must be conducted in order to find any potential bugs or edge cases we may need to handle.

5 Limitations and Discussions

The system is limited in various ways, such as our centralized server that we rely on for bootstrapping and sharing IPs. An important limitation of Kafka Lite is supporting only reading and writing to a single topic. The write operation is not asynchronous because it blocks until a leader successfully updates all followers. There's also a lack of functionality to delete Kafka data after certain time. Another limitation is that Broker ID has to be specified on CLI manually, and there is no notion of keeping track of ID uniqueness between brokers. Related to this limitation is if brokers that are leaders fail too often, multiple leader elections *can* happen simultaneously, but while leader election is running for a partition, no writes can be made to it. Thus, blocking progress on the producer side.

We definitely spent a lot of time trying to figure out the initialization process of the Kafka Brokers, i.e. which broker is leading what partition, how to share Kafka state, etc. We originally wanted to expose a notion of Client ID from the Producer API so that the application don't have to know anything about partition, as it is an internal infrastructure. Right now, we leave it to the application to decide which partition to write to, but this should be something we internally decide. Though we realized that deciding which client ID map to which partition ID is a whole new consensus process, we decided to scrap out the notion of clientID to simplify this project.

For the write operation, we originally wanted to log the events when we completed the first phase or the second phase of the protocol, so that upon failure, we know where we can recover from. We then realized that it is redundant to log these events if the broker is the follower, as everytime we restart a broker, it asks every leader of the partitions what are the messages that it miss. So even if the follower died during write replication, it will catch up on the missed messages even without an explicit recovery procedure. Failures on leaders will just return a write failure error, which is handled on our Producer side with a write retry. If the Leader has started the write replication process and died, and if one of the follower has replicated this write, the leader election will pick this follower as the new leader and we won't be losing any messages. Note that we are guaranteeing at-least-once semantics and message duplicate may occur.