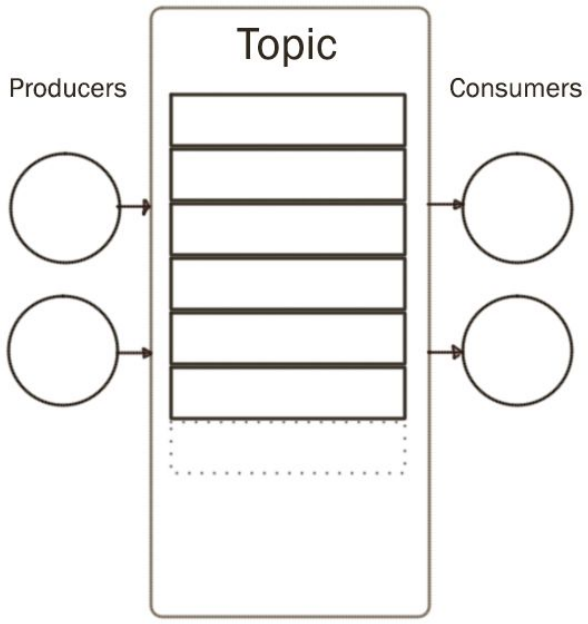
416 Project Proposal - Pub-Sub Streams for Viewing Traffic

Florie Cai - g4w8
Julian Lou - g6y9a
Ruimou Xu - i6y8
Jan Tache - o5z8

Background

Kafka is a distributed streaming platform that allows clients to read, write, and store data in a fault-tolerant fashion. By default, Kafka decouples clients writing to streams from clients reading from streams. which are continuously updating data sets.This means that those writing to a stream are not aware of who is reading the stream and likewise, those reading from the stream are oblivious to who is writing to the stream. Producers write to a topic (a named stream) and Consumers, only knowing the topic name, can pull data from this stream without having to know who the Producers are. Kafka guarantees successful writes are not lost even if there is a server failure.



Basic Topic Stream: The producer writes and client reads are not directly written to the dataset but handled by the topic instead. The topic is constantly growing with input from producers.

Problem

We want to generate heat maps representing street traffic. A map will be divided into equal-sized quadrants and the location data for each quadrant is stored in a single topic. For a client to read traffic data in the form of GPS coordinates from any topic, they must also participate by sending their GPS coordinates to their appropriate topic. The topic where they will write to will be determined by their coordinates via the server. We assume that these clients accurately send their coordinates and are not adversarial in that they would spoof their location. For the scope of this project, we also assume that Producer clients will always write to the same topic, i.e. all coordinates they write to the stream will be contained in one quadrant.

This is a distributed problem because we don't want all the connections sending GPS data to a single server due to the heavy load and single point of failure. Instead we want to implement a system where data lives in replicated nodes that form a cluster and can be retrieved from these clusters when a consumer client reads from them.

Approach

Our approach will use a Kafka-based system of data topics that represent geographical quadrants. Apps will write to a topic with their GPS coordinates and consume GPS coordinate data from any topics it requests to. Each app will participate in a leader-follower cluster where the leader is the main point of connection for topics and the followers act as a redundant data store for the leaders if the leader fail. There will be a central server that maintains the list of topics and is also responsible for providing Producers with clusters when they try to create a new topic.

Our Proposed System

There are 4 types of nodes in our system:

1) Server
2) Producers
3) Consumers
4) Cluster (Clusters are not nodes in itself. They are a conceptual grouping of nodes. Clusters are like a black-box to and from which we write data and pull data. In this proposal, we will usually call a group of one Leader and *X* Followers a Cluster if it is interacting with the Producer or Consumer API since to these clients, they should be unaware of how our system attempts to provide fault tolerance)
    a) Leaders
    b) Followers

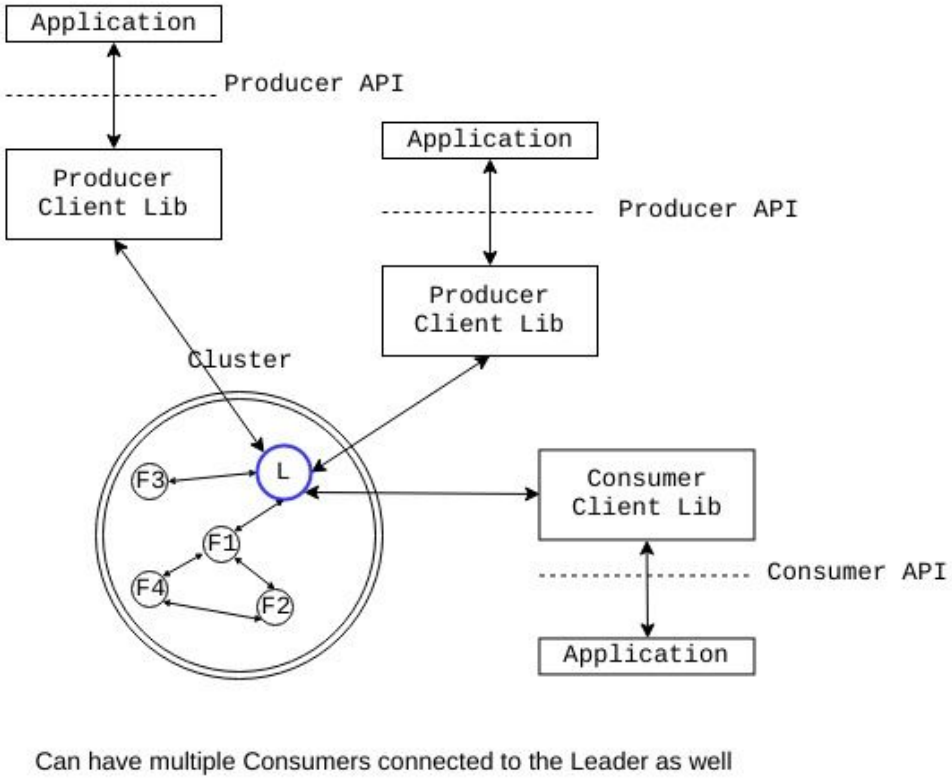


Diagram 1

Producers can create topics or send to any existing topic once they are connected to a Leader. Consumers can read any existing topic, provided they know its name.

Topics in our system will be non-overlapping, adjacent quadrants on a map. Clients can only produce data to one topic at a time but can consume data from multiple topics. The topic they will write data to will be dictated by their GPS location. To limit the scope of this project, we assume that an application's GPS coordinates are always producing to the same topic; therefore, they are always within the same quadrant on a map.

Leader and Follower nodes act as a single cluster. From the perspective of a Consumer and Producer, they are unaware of the difference between Leaders and Followers. From the Producer and Consumer's library, they are only given cluster connection to in order to write or read data from and are otherwise oblivious to how the cluster operates. The data held by the nodes in a cluster represent the state of a particular topic. The Producer/Consumer Library uses the Server to find the connection to a particular Leader/Follower cluster in charge of a topic.

Server

There is a single Server node that is aware of all existing topics and which nodes belong to which clusters. A Server provides a Producer node with a cluster (1 Leader + *X* Followers where *X* is specified in a server configuration) when it first creates a topic. Topics are assigned per cluster. On startup, the Server node is responsible for listening to requests from Producer nodes attempting to create a new topic. The only time a Producer should communicate with the Server is when it attempts to create a new topic. If a Consumer wishes to read from a new topic it must also request the Server for the cluster. The Consumer is also capable of remembering which topics it read from before in order to reduce server load.

Clusters' Network Topology

A server provides a cluster to the Producer when a Producer wants to create a Topic. The Server will create a Cluster that is a directed acyclic graph. Each node is guaranteed *M* connections (this number will be decided based on a static server config). The Server will choose the initial Leader in a cluster. The immediate Followers of the Leader will be provided with a list of siblings. This is for the case when a Leader fails, and immediate Followers need to join together so they can run the consensus protocol in order to elect a new Leader.

Load-Balancing Clusters

Our system will attempt to optimize for a high-number of Writes. Generally we can expect that not all topics are equally written to and as a result the clusters will not be uniformly utilized. In areas where there are not enough nodes to create a viable cluster to maintain traffic data or in areas where there is too much traffic data for any one node to handle, we will implement strategies to allow nodes to migrate from being in one cluster to another. We also allow multiple Leaders in a cluster, as long as all nodes in the Cluster reach a consensus following our consensus protocol outlined below.

Electing Additional Leaders

Under heavy traffic, we attempt to elect an additional Leader in order to reroute some Writes to this newly elected Leader. The Cluster must have at least `minNumFollowers + 1` because one of its Followers will be promoted to be an additional Leader.. For example, Cluster A's Leader is being hammered with Writes. It then attempts to find a node in its own Cluster to promote to being a new Leader. Leader A will select an arbitrary node, transfer any missing data the node needs in order to be eligible to be a Leader. It is then promoted to become a Leader through our consensus protocol. At this point, the topology of having 2 Leaders will match Diagram 2.

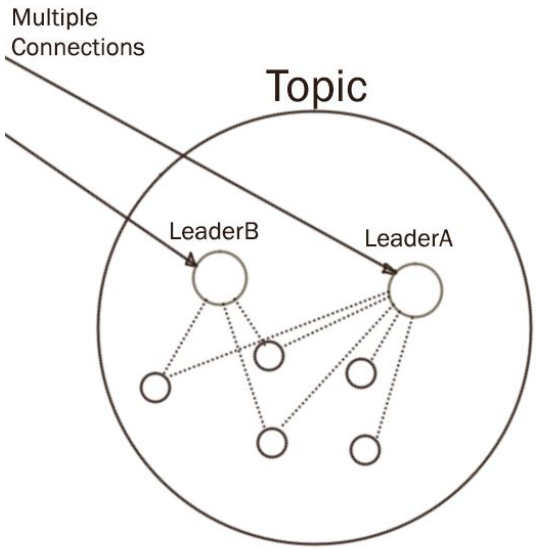


Diagram 2
Leader A originally had 6 followers. Leader B is elected from one follower. Since N=5,
the minimum number of Follower nodes Leader A must provide to Leader B is Floor(N/2) + 1 = 3.

In both ways of electing an additional Leader, we still need to guarantee the order of the data. Since there are overlapping Followers, we will use them as the source of truth of determining data order. This also means that when a Follower returns a successful Write, it also needs to return the index where the Write is in the dataset.

Demoting Additional Leaders

There may be bursts of heavy Writes which cause the Cluster to elect additional Leaders. Once these Writes subsides, we perform a garbage-collection for Leaders and begin to demote them to become Followers. The Leader with the most data gets to stay on as Leader (as we mentioned previously, with multiple Leaders, Leaders will not have a complete dataset as they only receive a subset of Writes). The rest of them are demoted to Followers and if these nodes become underutilized, the Cluster will notify the Server of available nodes it wishes to graciously donate to other Clusters.

Data Consistency

We need to ensure that the followers have data that is up to date with the leader's data versioning so that the integrity of the cluster is maintained even during leader failure. We will follow an eventually-consistent model. The Leader node floods its newly received data from the Producer to its Followers. Each Follower must write the new data to disk and confirm with the Leader that it has written successfully. In addition, it will propagate confirmations up to the Leader so that the Leader eventually receives enough (the majority + some safety buffer) unique confirmations to determine that the write is complete.

The "correct" version of a file is determined by the most recent version that the nodes hold. Since we do not return from an operation until the Leader has verified that the majority of the cluster has received the operation, then even in the case of a leader failure, we know that the most recent version is distributed among the majority. Given that the network is still connected, we can propagate these most-recent versions and to ensure consistency.

Node Failures

Server Failure

The Server can disconnect and fail. Since the Server contains the complete list of clusters and its respective IP Addresses, no new topics can be created while the server is down. This also implies that our load-balancing protocol can only work while the server is online since Clusters ask the Server for under-utilized nodes. On the other hand, nodes and clusters will not immediately fail if there is no server available, so any established clusters and clients will continue operating normally.

Cluster Failures

Follower nodes are replicas of the Leader and any of these nodes can fail. If a Follower fails, a Leader checks if it meets the minimum number of followers. If it does, it can continue to operate as if the follower did not fail. If the number of followers dips below the minimum, the Leader will request followers from other Clusters under the load-balancing protocol.

In the case of a Leader failing (disconnect), our system attempts to promote one of the Leader's Followers to become the new Leader. The new Leader will notify the Server that it is the new Leader node and from then on, Producers and Consumers will connect to the new Leader on their next Read/Write.

When a Leader node has failed and a Producer application attempts to write to a topic, this will be a blocking operation until either a new Leader has been elected or there are no nodes that are eligible to be a Leader node. In the case of no eligible Leader nodes, an error of `InsufficientReplicasError` is returned.

Consensus Protocol

The followers contain a list of other followers. The follower list is maintained and updated by the Leader as new followers join and leave the cluster and updates are sent to all followers. In the event a leader dies, the followers will connect to *N* other followers on a follower list. The followers will know the completeness of their dataset by the versioning of their writes. Tiebreakers will be based on the order the followers connected to the leader based on the follower list. The election of the first leader follows a similar two-phase commit protocol. When the follower knows that it should be the leader it will send a declaration to the other follower on it's connections and ask for them to consent to it becoming the new leader. If a follower receives a declaration that has a lower completeness value than their own nomination then they will reply with their higher nomination to overwrite the lower one. On the other hand a follower can give consent of the declaration they receive is more suitable than their own and overwrite their nomination with the new declaration. A follower that successfully receives consent from all it's other connections will know that it is the most suitable follower on the cluster and adopt the leader protocol and message the server and followers as the new leader. If the follower receives an overwriting declaration then it will send its consent instead.

Promotion of Follower to Leader

In order for a Follower to be eligible to become a Leader, a Follower must have all the data that the previous Leader has sent an ACK for.

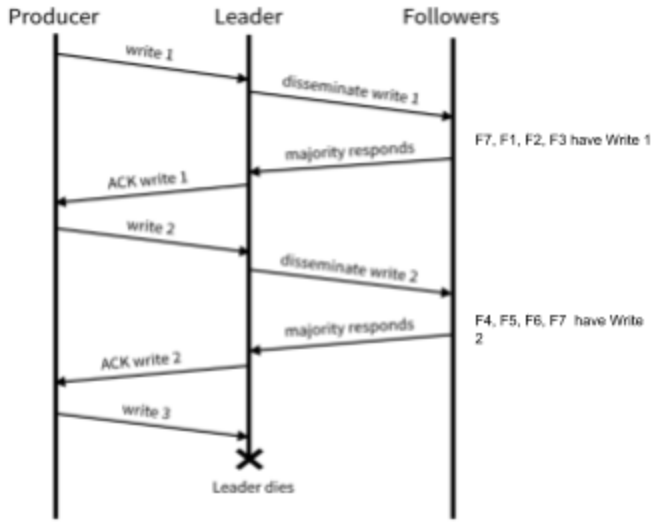


Diagram 3

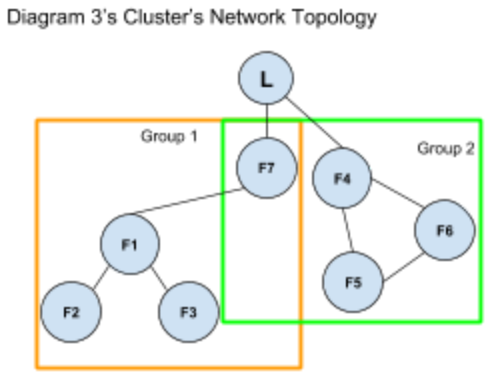


Diagram 4

In **Diagram 3**, after the Leader dies, any Follower that has both Writes 1 and Writes 2 are eligible for promotion. In this example, only F7 meets this requirement.

In **Diagram 4**, once the Leader dies, we will have 2 segregated networks (F7 and F4 in the **Diagram 4**) . In our system, the immediate Followers of a Leader know the Leader's immediate children (this is part of the Leader promotion process - Leaders tell its Followers of its immediate Followers). So in the diagram, F4 and F7 know of each other, and can therefore connect and create a single network. The Cluster then proceeds to the consensus protocol to select a new Leader.

However, there can be the case where not only the Leader dies, but F7 also dies during the process of trying to find a new Leader. F7 was not promoted to be a Leader yet, so its Followers, F1, are unaware of F4's existence. This causes 2 segregated networks that cannot be joined **and** each network is missing data (Group 1 missing Write 2, Group 2 missing Write 1). In this situation, a leader cannot be elected and the Followers will notify the Server to rebuild the cluster and elect a new leader. If the Server is offline, this means that this Cluster is stuck until the Server comes back online.

Rejoining Nodes

A node may rejoin a topic with an outdated dataset and receive writes from the leader that will update the dataset to the most current version. Because of node migrations for load balancing and failures/disconnects we cannot guarantee a cluster is the same, however we can always guarantee the same topic is there to join. When the node rejoins a topic it will be treated as a completely new follower and be inserted into the cluster.

Application APIs

Producer API:

- `CreateTopic(topicName) -> cluster, err`
  On success, returns the cluster (which contains the cluster Leader's IP Address, and the minimum number of replica nodes the cluster guarantees when a Write succeeds. The number is determined by the server) where the client will write its GPS coordinates.
  On failure, can return the following errors:
  - `DuplicateTopicNameError` - A Cluster containing `topicName` already exists
  - `DisconnectedServerError` - Cannot connect to the server.
- `GetCluster(topicName) -> cluster, err`
  On success, returns the cluster containing an IP address with the given `topicName`.
  On failure, can return the following errors:
  - `TopicClusterDoesNotExistError` - No Cluster with `topicName` exists
  - `DisconnectedServerError` - Cannot connect to the server

- `Write(topicName, gpsCoordinates) -> err`
  On success, returns no error
  On failure, can return the following errors:
  - `InsufficientReplicasError` - The write could not be guaranteed to be replicated at least *N* times (*N* is specified by the server's configuration). This error can also be caused by the cluster not being able to come to a consensus on electing a new leader if the original Leader node has failed.

Consumer API:

- `GetCurrentLocationCluster(gpsCoordinates) -> cluster, err`
  Returns the associated cluster for a given GPS Coordinate
  On failure, can return the following errors:
  - `TopicClusterDoesNotExistError` - The given `gpsCoordinates` are outside of the quadrants of the existing topics

- `GetCluster(topicName) -> cluster, err`
  On success, returns the cluster that the client will connect to and read from
  On failure, can return the following errors:
  - `TopicClusterDoesNotExistError` - No Cluster with `topicName` exists
  - `DisconnectedServerError` - Cannot connect to the server

- `Read(topicName) -> gpsCoordinates, err`
  On success, returns a list of all the GPS coordinates that have been written to this Topic.
  On failure, can return the following errors:
    - `DataNotAvailable` - There is at least one successful Write call to this Topic that could not be retrieved.

Cluster API

- `Declare(Nomination) -> Response`
  Returns either a consent response or a more suitable nomination.
- `Elect(Nomination) -> Response`
  Returns once all followers have confirmed they accepted the new leader, the leader will then switch to the leader mode and other nodes will follow it.
- `Write(data)`
  Flooding procedure initiated by leader to send out a write.
- `Confirm(followerID)`
  Confirm that `followerID` has successfully written. These calls propagate to the Leader.
- `Follow(followerSettings) -> Listof ipAddress, err`
  A follower will call this on a Leader and receive IP addresses to connect to. Can return a `DisconnectedError` if the Leader is not alive.
- `FollowMe(ipAddress) -> err`
  A Leader or Server will call this on a follower to follow the given IP address for cluster creation or node migration. Can return a `DisconnectedError` if the `ipAddress` is not able to be connected to.
- `Borrow(N) -> Listof ipAddress, err`
  A Leader will call this on another Leader to get N followers to take from. Can return a `InsufficientNodesError` if there are not enough followers to give.
- `Connect(ipAddress)-> err`
  A node will connect and give the IP address to connect back on. Can return a `DisconnectedError` if that follower is not alive

The Cluster API is largely internal so semantics are not as clearly defined for this class and stands to change based on the needs of the system.

Testing & Demo

In this section, the term “LF node” will be used to mean a Leader/Follower node.

Application to be Built

We will be deploying our central server on Azure as well as our LF nodes for creating clusters. The system that lives on azure will essentially be the topic streams. We also hope to deploy the traffic apps and the heatmap app on Azure. The test app will be an app with a direct connection built into all traffic apps to generate precise GPS data so that we can compare accuracy with the heatmap app.

Basic System Tests

Note: not comprehensive - more complicated tests shall also be run

- Basic disallowed operation
    - 4 LF nodes, 4 clusters
    - None of the LF nodes should be operating, as none of the clusters meet the minimum followers allowed for operation
- Normal operation
    - Run 20 LF nodes, in 4 clusters, with 20 clients
    - Loads are equally balanced, so no load balancing should occur
    - No failures of any component
    - Should observe activity on heatmap
- Single client failure
    - Run 20 LF nodes, in 4 clusters, with 4 clients (1 per cluster)
    - Should look exactly the same as normal operation in the logs
    - There should be an observable difference on the heatmap showing one of the quadrants lacking activity
    - Normal operation for entire system (eventually)
- Single leader failure
    - Run 20 LF nodes, in 4 clusters, with 20 clients
    - Kill a single leader node
    - Should observe a new leader being elected in the logs
    - Client operation may be interrupted for some time, but should recover eventually
    - Normal operation for entire system (eventually)
- Single follower failure
    - Run 20 LF nodes, in 4 clusters, with 20 clients
    - Kill a single follower node in one cluster
    - Clients that are connected to that client may experience an interruption, but should recover eventually
    - Normal operation for entire system (eventually)
- Multiple follower failure, causing disallowed operation
    - Run 20 LF nodes, in 4 clusters, with 20 clients
    - Kill 3 LF nodes in the same cluster, such that there is 1 leader and 1 follower (disallowed operation)

        - The producers and consumers for this cluster should no longer work properly, and the leader+follower should not be operating
- Server failure
    - Run 20 LF nodes, in 4 clusters, with 20 clients
    - Kill the server
    - Existing operation should continue as normal
    - Advanced testing will include: test that load balancing does not work, test that new clients cannot connect
- Overloaded leader
    - Run 20 LF nodes, in 4 clusters, with 20 clients (all 20 clients on one cluster)
    - Should observe a new leader created in the logs
    - Advanced test: kill server -> leader should do a cluster split

Timeline:

Due Date	Tasks	Assigned To
March 14	● Create Azure Deployment Scripts and VMs ● Setup Basic Server ● Create General LF nodes that can connect to server	Julian
March 18	● LF Nodes can store and serve data but do not form clusters yet (single node clusters per topic) ● Basic Producer and Consumer Apps ● Azure Testing works with the basic setup	Ruimou
March 21	● Multiple LF nodes will form clusters ● Test stability on Azure	Jan
March 25	● Load-Balancing Protocol (Merging clusters - multiple leaders) ● Test Basic Tests on Azure	Florie
March 28	● Leader Promotion Protocol ● Test Multiple Followers and Node Failures on Azure	Julian
March 31	● Test on Azure ● Fix major issues we discover	Ruimou
April 4	● Add Logging to support GoVector/ShiViz if time allows ● Finish all low priority issues and bugs with remaining time	Jan does GoVector, rest of us on bugs
April 6	● Final Report Done	All

SWOT Analysis

Strengths

- Same team as proj1 so know each other's working styles
- We are only handling 6 API calls, most of which are pretty straight-forward (i.e. return whatever the cluster says) - so we can focus on the distributed system instead of the API details

Weaknesses

- We are still inexperienced with writing in Go
- One of us doesn't have a laptop, so cannot work away from home
- Low morale from poor demo on p1 and a2

Opportunities

- Distributed traffic management is very relevant to the growing trend of self-driving cars which could utilize this system to determine traffic while on the road
- Apache Kafka is a well known and well used service that we can use as high level reference material
- Many possible libraries to implement the heatmap visualization app

Threats

- Modeling and creating the application could use more time than expected - randomly moving cars that make some form of sense, as well as heatmap generation
- Distributed real-time traffic data collection does not seem to have many practical resources to borrow so we are essentially winging it in the domain
- Data synchronization is difficult - maintaining that leaders and followers replicate data could have performance issues and race conditions will be highly problematic
- Our system has a "if it works, you can't see it" kind of problem so it makes it difficult to demo. We are trying to show that our implementation of replicated clusters to maintain consistent data works so we'll have to play around with up to how many node failures we should support.
- Difficult to demo clusters merging and showing that our consensus protocol is working. Will have to rely on logging to prove that it is working, as the heatmap will not show this.
- Jan is working 20h a week among 3 other classes
- Most members have issues with balancing the heavy project workload with other courses also putting out heavy workload at this time

References:

https://en.wikipedia.org/wiki/Geographic_coordinate_conversion, accessed March 8, 2018
https://kafka.apache.org/documentation/, accessed March 1, 2018