Pub-Sub Streams for Viewing Traffic

Florie Cai (g4w8), Julian Lou (g6y9a), Ruimou Xu (i6y8), Jan Tache (o5z8)

Abstract

Data streams are difficult to make distributed, fault-tolerant and also correct. Our system implements a Kafka-style distributed data stream that uses data replication clusters to handle reads and writes to our system. The data streams guarantee that an individual application's writes will appear in the order they are sent and that the data is appended to the stream in the order that the cluster receives it. A cluster can survive node failures up to half its size before it can no longer guarantee the above properties.

Introduction

We would like to use our proposed system 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 data stream. The stream where clients will write to will be determined by their coordinates via the server. Clients can read from any data stream as well.

Our implementation of the data stream uses clusters of nodes that each try to replicate the same stream. In each cluster we assign a leader to act as the central communication point that clients read and write to. Each cluster begins with *2N -1* nodes. The leader ensures not only that we have an replication factor of *N* inside the cluster before returning on writes but also that the order of writes in the entire cluster is the same as it had been received by the leader. Our cluster is designed to handle *N-1* node failures, including the leader itself, in addition to maintaining functionality with existing clients in the event of server failure.

We assume that all the nodes in our system can be trusted and will not behave adversarially, though they are allowed to fail in the system.

Terminology

We begin by introducing some terminology relevant to our system.

- **Topic**: A data stream to which a *cluster* is assigned. Our system will designate topics as quadrants on a map.
- **Server**: A singular entity which creates and stores *topics*. Internally, it maps each topic to its *Leader* so *Producers* and *Consumers* can connect to them.

- **Producer**: An entity that may create and write data to a *topic*.
- **Consumer***:* An entity that may read data from a *topic*.
- **Cluster**: A group of one *Leader* and many *Follower* nodes that is responsible for maintaining state and responding to queries pertaining to a *topic*.
    - Leader: A special node that acts as a control point for requests from *Producer* and *Consumer* nodes to the *cluster*.
    - Follower: A generic node that serves to replicate data.
    - N: the minimum number of replications that a cluster must ensure

Note: Topics and clusters are mapped bijectively.

Implementation

The main design features of our system are the server, client libraries, and most importantly, cluster implementation. Specifically, in our cluster we focus on maintaining data consistency between nodes on client writes and being able to elect a new leader from the cluster in the event of a leader failure.

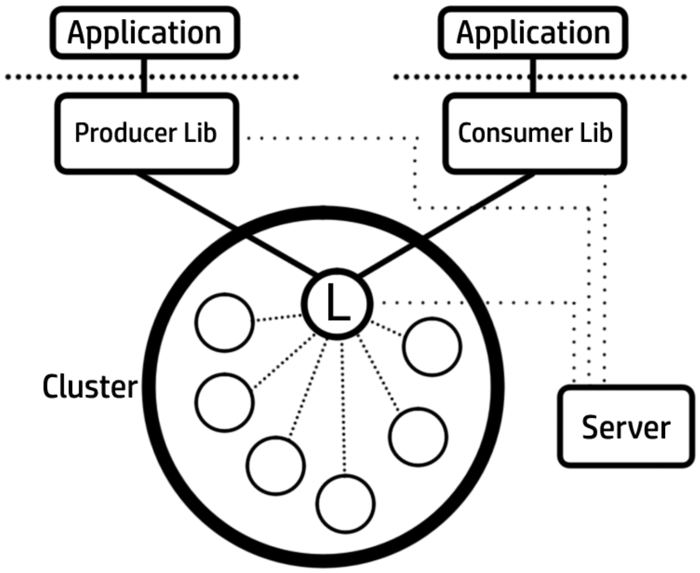

Diagram 1. The basic overview of our system.

Server

There is a single Server that is aware of all existing topics and the leader of each cluster. The Server provides a Producer node with a cluster (*1* Leader + 2*N-1* Followers, where *N* is specified in the server configuration) when it first creates a topic. Topics are assigned per cluster. On startup, the Server will wait for registrations from nodes and establish a heartbeat with them. When nodes fail, the server will also remove them from its active node list.

The only time a Producer should communicate with the Server is when it attempts to create a new topic. When this happens, the server will randomly assign connected nodes that are not yet part of a cluster to the topic the Producer requests. It then sends the list all nodes belonging to the cluster to one of them. A successful return registers that node as the leader of the cluster responsible for the new topic, which is then returned to the Producer who issued the request. If a Consumer wishes to read from a new topic, it must also request the cluster from the Server.

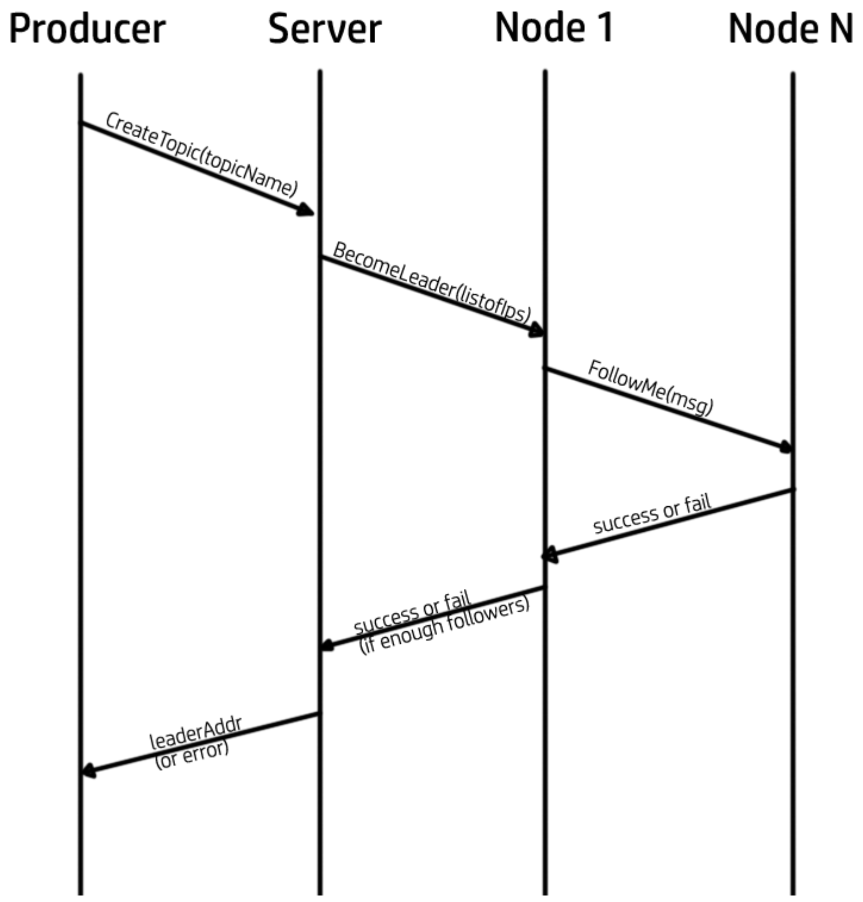

Diagram 2. Creating a new cluster.

The Server can also handle leader replacement on existing topics as a result of new leader election in the event of leader failures.

Client Libraries

One advantage of our implementation is how easily our client library is built.

The Producer API relies on only 3 calls: `OpenTopic()` and `Write()` or `Close()` to topics. `OpenTopic` returns a session with the cluster leader when successful. A successful `Write` guarantees that the data written will persist in the topic as long as the cluster does not experience a complete failure. However, when the connection to the leader fails during the write, we do not hide the error from the application and instead allow them to either try to reconnect or otherwise handle the error in their own way. The Consumer API is similar to the Producer API, with the difference being that a Consumer cannot create topics, and the `Write` call gets replaced with a `Read` call which returns the data set from the topic.

Cluster Operations

The current system implements star topology: the leader has an RPC connection to all the followers, and each follower has an RPC connection to the leader. In order to detect failures, the leader maintains a heartbeat to each follower, and each follower maintains a heartbeat back to the leader. A node failure is detected if either of the following happens: a heartbeat RPC call to a node returns with an error, or a heartbeat from a connected node has not been received for a fixed amount of time (4 seconds at the time of writing).

`Write` calls to the leader update the leader's data set and trigger async calls to all the followers in the cluster. Since we are using async calls, we use a timeout for write confirmations to the Follower nodes. The Leader will block until either (1) there are confirmed writes from *N* nodes or (2) we receive *N+1* timeouts which we consider as failures. The read operation requires no blocking and immediately returns the confirmed writes on the leader's data set.

Resource Allocation

The number of nodes in a cluster is static during normal operation; this number is set by a server configuration file. Nodes entering our system are initially held by the server without a cluster and are utilized to create a new topic when a Producer needs to create one. Because we hold our clusters in the current star topology we do not want an unnecessary amount of connections to our leader.

The Leader node has a subprocess that monitors the number of followers and compares it to the desired cluster size count. There is a leeway of 1 node to account for nodes temporarily failing but rejoining. When the difference between the follower count and desired cluster size exceeds this leeway, the subprocess will request that difference from the server and add those new nodes to the cluster. The Leader will provide the new nodes with its complete data set so upon joining the cluster, these new nodes will be complete replicas.

We chose this leeway of 1 node because if a node failed, the Leader would immediately request a node. If that node attempted to rejoin, the cluster would already be at capacity and this would defeat the purpose of committing our write data to disk and have nodes attempt to rejoin the same cluster. We could modify this leeway to be any number up to *N* nodes because we tolerate up to *N* failures. We would want to begin requesting nodes at least a few nodes before reaching the minimum in order to prevent the system from rejecting writes due to having an insufficient replication factor of *N*.

Failure Cases

Server Failure

On server failure, any existing clusters and Producer/Consumer connections will remain intact and not experience any disruptions. However, new clients and nodes will not be able to join any new clusters. During a server failure, it is possible that a cluster has enough node failures that it ends up failing reads and writes because it can no longer connect to the server and request new nodes.

Leader Failure and Consensus Protocol

The leader sends a follower list to its followers when they connect and maintains the list on all followers as nodes join or leave. With this, all nodes in the system are aware of the oldest living follower in the cluster. The followers are identified by an integer ID, with the lowest integers being the oldest followers. This list can be maintained because a follower is only removed from the list when the leader detects a follower failure. This is a guarantee we make in order to carry out the consensus protocol.

When the leader fails all the follower nodes in the system will detect this through the leader heartbeat. They will then contact the lowest follower IDs in order of their local follower list. Because of the guarantee, we know that nodes reach the same follower who is alive and the oldest alive follower will be on everyone’s lists because it could only have been removed if the follower had died. Dead followers on the lists do not matter because the nodes will move on after failing to connect and eventually connect to the oldest living follower.

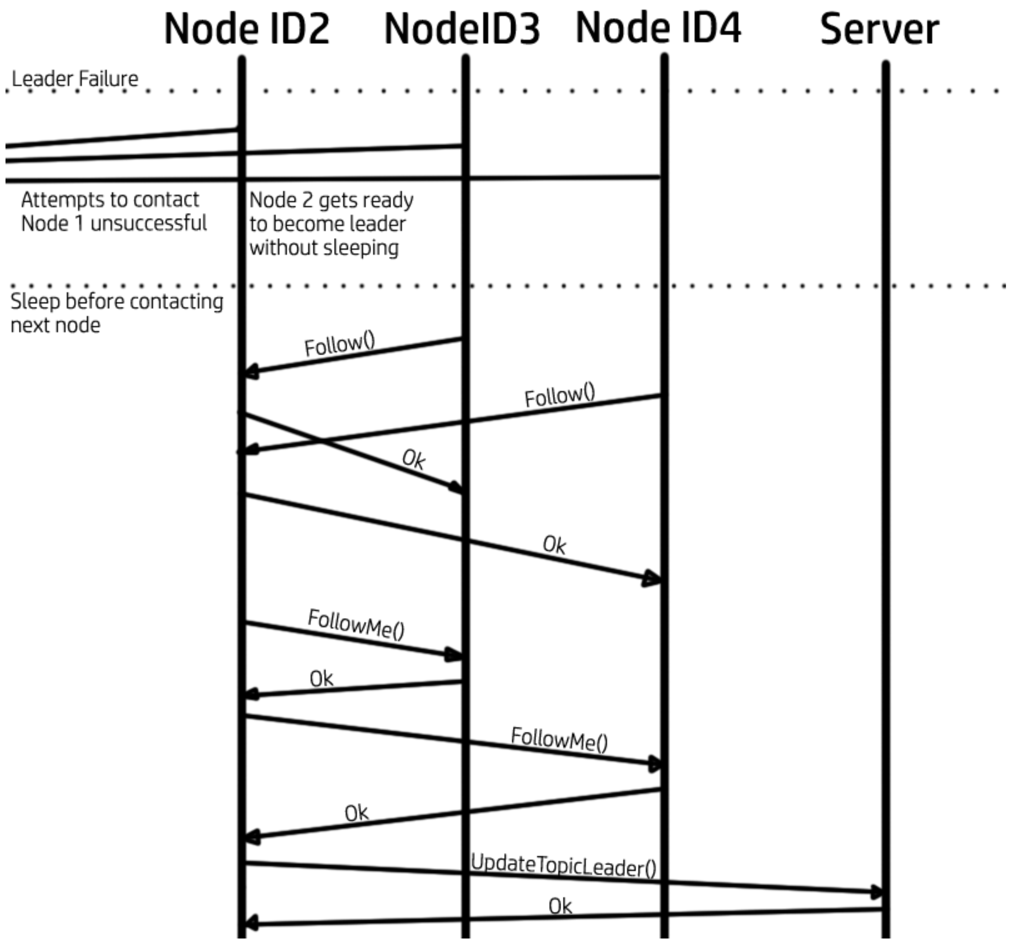
Node ID2
NodeID3
Node ID4
Server
Leader Failure
Attempts to contact Node 1 unsuccessful
Node 2 gets ready to become leader without sleeping
Sleep before contacting next node
Follow()
Follow()
Ok
Ok
FollowMe()
Ok
FollowMe()
Ok
UpdateTopicLeader()
Ok

Diagram 3. A leader failure situation with the oldest follower also failing.

When the follower gets enough connections, it will switch to the leader protocol and commit the follows on all the nodes that connected to it and then call the server to register itself as the new leader of the topic which will allow reads and writes to succeed again. If the follower fails to receive the required number of connections, it assumes that he was passed over (perhaps a network partition) and try to connect to the next follower instead. The format of this protocol resembles two-phase-commit; however, the commit-request phase is decided not by the oldest node but individually by each node.

Completion of the consensus protocol does not lose any state on the topic though all the follower IDs will be reset under the new leader. A requirement of a node becoming a leader is that it has a complete view of the data. This means when there is a newly elected leader, we must do a data synchronization with other nodes to ensure it has all the data.

Follower Failure

The leader needs to maintain at least *N* connections to be considered safe for read/write replication. It always tries to keep *2N-1* nodes in its cluster to keep *N* as the majority. On follower failure the leader will immediately try to request a new node from the server but is tolerant of up to *N-1* failures on the cluster.

Leader and Follower Failure

If the leader and fewer than *N* nodes fail on the system there are still enough nodes to reach consensus.

In the case where the leader fails and at least *N* nodes in the cluster fail in a short timespan, we cannot reform the cluster. The to-be-implemented strategy here is a timeout to detect consensus failure. There are two conditions to failing consensus: (1) the node has attempted to become leader and did not get enough followers, and (2) the node never received a “follow me” request. Once the timeout happens, the node tries again on the next entry in its follower list until it exhausts the follower list. When this happens, the node goes back to the server with its data set, which will rebuild the (hopefully) complete topic and recreate the cluster.

Leader and Follower and Server Failure

This is a catastrophic failure case that we do not plan to address.

Rejoining Nodes

Our rejoining nodes strategy is to-be-implemented. When a node rejoins, it must go to the server and notify the server what cluster it previously belonged to based on what was written to disk. The server will return the topic Leader’s IP address (which may have changed while the node went offline) and will attempt to rejoin the cluster. The Follower will create a diff of which versions it is missing and the Leader will send the missing data to the Follower to bring it up to date. If the node was offline for long enough and the topic’s cluster size is full (i.e. its original size based on the server’s `config.json`), then this node will become an orphan node and register itself with the server so it is free to join any cluster.

Challenges

Changes from Proposal

Initially, we wanted to increase the performance of our clusters in response to high traffic by delegating additional leaders under high load. However we ran into the issue that we could not maintain a serialized dataset guarantee with this and ultimately had to scrap this idea.

Cluster Topology

Stability of clusters became an issue because our star topology is very simple and very reliant on the leader. In the future, we would like to dedicate a separate workload to designing a specific cluster topology such that followers can connect to other followers and form a chain of command to the leader. Several remaining TODOs in our codebase are related to this extension of the code.

Consensus

The consensus protocol is not completely foolproof and we can unfortunately reach a failstate in rebuilding the cluster. This is because the timeout period between election and nomination attempts can cause failures, such as nodes attempting to follow a potential leader before it realizes that it needs to ready itself to be leader.

Our rejoining nodes strategy is limited given that we must go through the server rather than try to directly attempt to connect to the Leader. The reason why we chose our method is because Leaders and its Followers may change while a node has died, thus requiring the node to contact the server. It is slower to have the server act as an intermediary step, but if nodes changed often, we would have to rewrite to disk every time that happened which would become another bottleneck in our system.

General

Overall, the performance of our prototype could use improvement, but it manages to guarantee a correct serialization of data between multiple clients writing. We plan to make improvements to synchronizing the data during the consensus protocols so that integrity of the data is more likely to survive in the system even after several critical leader failures. We also want to improve our client-side library so that it stores data on drive for offline reads as well as remembers old topic leaders so it can try to circumvent the initial server access.

The demo applications are to-be-implemented. Due to time constraints we did not design more than just basic test apps. The final plan for the demo application is a program that moves producer child apps down a predefined path while a consumer app attempts to read the data stream and illustrate the accuracy of the stream. We will display this in a browser with a map of UBC and divide the campus into quadrants. We will demonstrate that producers that write to 1 topic (representing 1 quadrant) can read from multiple other quadrants.

Allocation of Work

- **Proposal**: Florie, Jan, Julian, Ruimou
- **Design**: Florie, Ruimou, Jan
- **Cluster Implementation**: Florie, Ruimou
- **Consensus Protocol**: Jan, Ruimou
- **Server Implementation:** Florie
- **Resource Allocation**: Jan
- **Demo Applications & Environment**: Jan, Julian
- **Azure Deployment**: Julian
- **Testing**: Jan, Julian, Florie, Ruimou
- **Report**: Ruimou, Florie