1. Introduction

In this project, we are tasked with building a distributed system to provide a unified service through a collection of independent devices geographically separated but connected by a network. The network of computers communicates and coordinates their actions to appear as a single system to the user/client (application). A key aspect of this endeavor involves acquiring a deep understanding of the algorithms currently used in real-world distributed systems. Our objective is to study these algorithms and effectively implement them in our distributed system project.

2. Problem Formulation

We have chosen to implement ZooKeeper as a service for metadata management for our scaled-down version of Kafka. At its core, Kafka works through the publisher-subscriber pattern. With the help of a message queue, 2 applications can communicate asynchronously securely and concisely. While having a single instance of a message queue might suffice for small-scale applications, it is unfeasible for just a single instance of the message queue to handle the high-volume data workloads of a large-scale application. Moreover, most enterprise-level applications require functional requirements such as Correctness/Consistency, Scalability, and Fault Tolerance.

To meet the functional requirements, Kafka has implemented key features such as Replication, Partitioning, and Offset Management. While these features help to ensure that Kafka can handle high volumes of data consistently, it requires ZooKeeper to manage the metadata that is involved in the operations.

As such, we need to build a system that can store metadata for Kafka. To ensure that our system can handle requests from various clients in a consistent, scalable, and fault-tolerant manner, we need to create a distributed ZooKeeper service.

3. Overall System Architecture

We designed an end-to-end producer-consumer Kafka use case - Ordered List of Football Goals Timeline consisting of 4 main components (Figure 1):

- ZooWeeper:
  - our simplified version of the ZooKeeper server to store metadata for Kafka
  - implemented in Go
- Mimic Queue:
  - a messaging queue as our simplified version of the Kafka broker to keep track of football goals (Event Data)
  - implemented using Express framework
- Producer:
  - HTTP requests to insert Event Data to our Mimic Queue
  - either through Postman requests OR shell script (“curl” command)
- Consumer:
  - a Frontend application to display the Event Data from our Mimic Queue
  - implemented using React framework

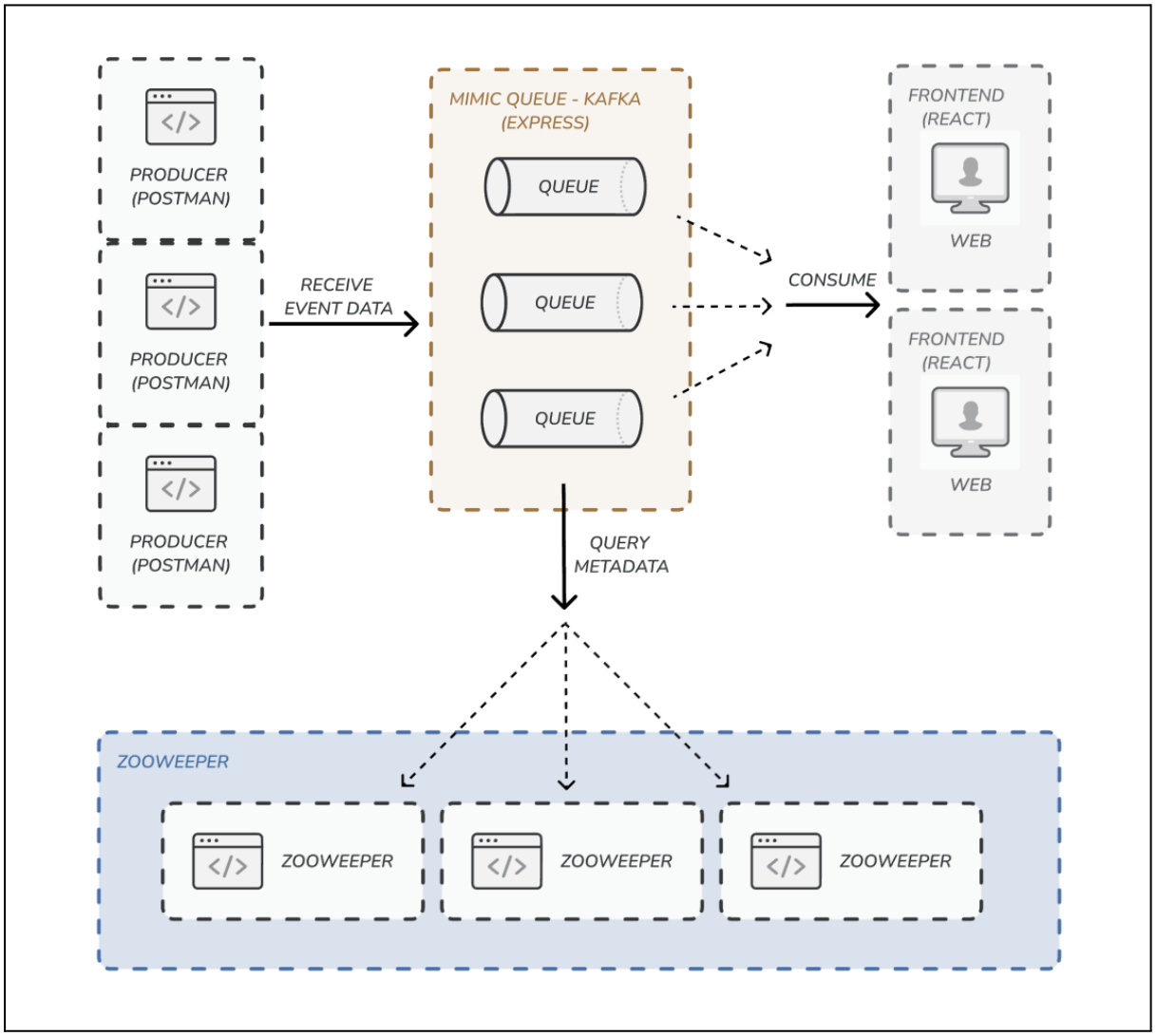

Figure 1: System Architecture

To showcase our distributed system implementation of ZooWeeper, we would

1. Deploy multiple ZooWeeper servers, Kafka clients, and Frontend applications
2. Send multiple Event Data to randomly selected Kafka brokers, each Event Data has a global timestamp in the field “Minute” of the Event Data.
3. Upon receiving a request, the Kafka broker should query the ZooWeeper to find out about other Kafka brokers in the cluster. All Kafka brokers would then update their databases based on the Event Data received (Figure 2)
4. The Frontend application would then display the Event Data from the Kafka brokers, ensuring:
    a. Event Data are ordered by the filed “Minute”
    b. Event Data across all Kafka brokers are synchronized
5. The above procedure shows our Correctness/Consistency flow. We then repeat the process taking into account Scalability and Fault Tolerance.

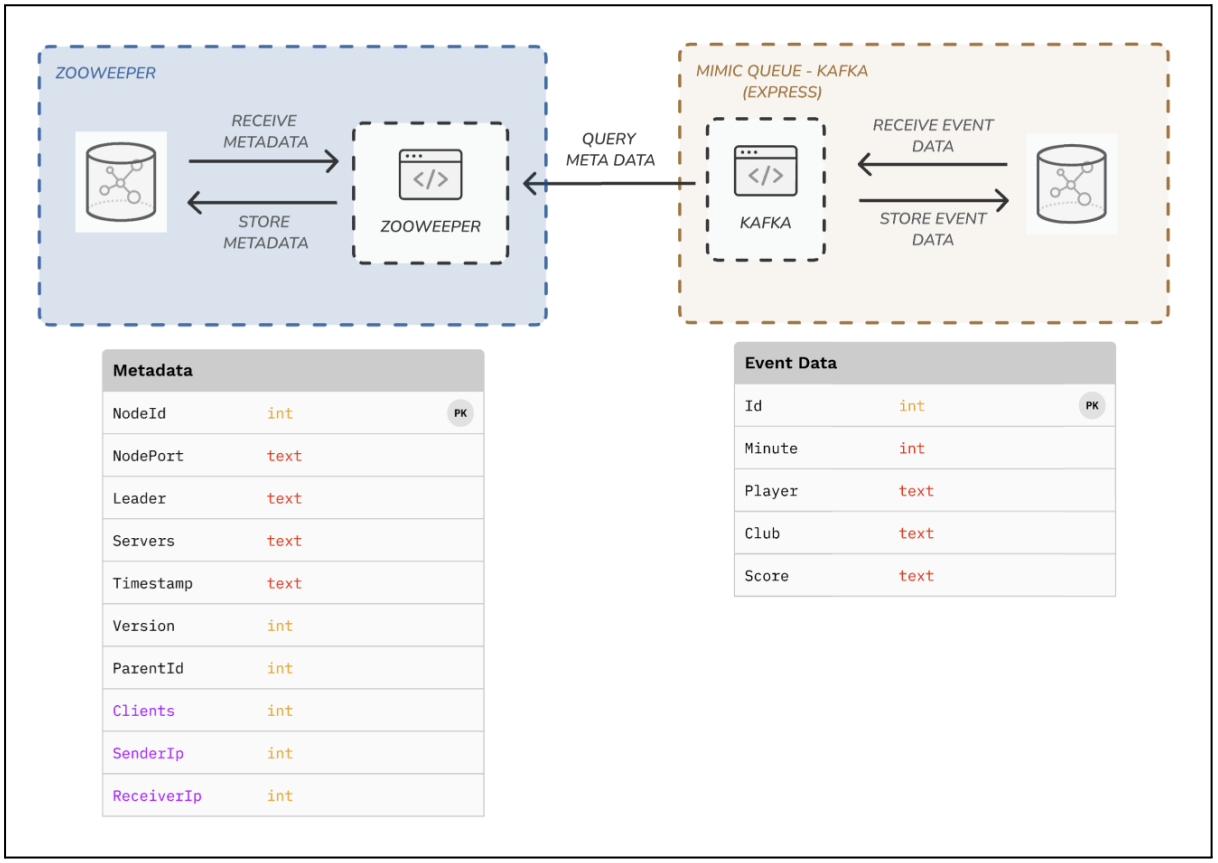

Figure 2: Main data flow (step 3) with the database schema for Metadata (in ZooWeeper Server) and Event Data (in Kafka broker). For Metadata, black fields are for ZooWeeper ensemble while pink fields are for Kafka clusters. More details in package ztree

This GIF shows our expected happy path result when a request is sent and all data is synchronized across ZooWeeper servers and Kafka brokers.

4. Design

4.1. ZooWeeper Design

We built a ZooWeeper ensemble, in which each ZooWeeper server consists of 3 main components (Figure 3):

1. **Request Processor**: package request_processor (go doc)
2. **Atomic Broadcast**: package zab (go doc)
3. **Replicated Database**: package ztree (go doc)

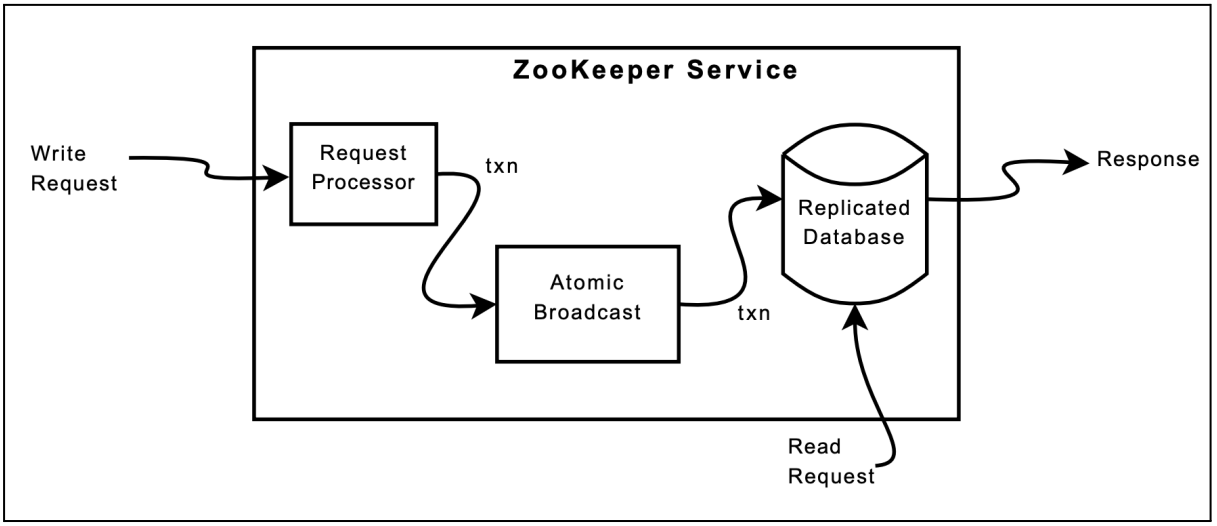

Figure 3: Internal Components (Reference: ZooKeeper)

Given our team's strength in full-stack web development, our implementations incorporate elements of logic and design typical of a full-stack application. Nonetheless, we have carefully restructured our code and added modifications when necessary to ensure that each component distinctly fulfils its intended role. Additionally, since the goal is not to build an exact clone of ZooKeeper, we would only implement features (with simplification) that are necessary to showcase our Kafka use case.

4.2. Basic Features

a. Data Synchronization

- Definition (Reference: ZooKeeper):
  - **Eventual consistency**: allows high read throughput as:
    - All Write Requests are forwarded to the Leader
    - Read Request are done locally in each server
  - **Linearization Write**: all requests that update the state of ZooKeeper are serializable and respect precedence
  - **FIFO client order**: all requests from a given client are executed in the order that they were sent by the client
- Implementation:
  - In the original design, ZooKeeper used session management together with TCP for intercommuncation so FIFO client order and linearization is trivial and built-in by default.
  - Since we don't have a session and all requests are sent randomly via HTTP, we needed to make a few modifications:
    - The Kafka broker will define the global timestamp (function **`createIncomingScore`** in server.js)
    - All Write Request will go through **`QueueMiddleware`** which uses a priority queue to order the client request by timestamp (middleware.go). This ensures FIFO client order.
    - **`WriteOpsMiddleware`** in middleware.go would then handle the Write Request by:
      - either forwarding to the Leader if the receiving server is a Follower
      - or directly **`StartProposal`** in zab if receiving server is Leader
    - **`ProposalOps`** in Atomic Broadcast proposal_ops.go using a classic 2-phase commit (Reference: Apache Active Messaging). This allows all Write Requests (aka transactions) to be executed in some form of total order in all servers. And since this order is defined by the timestamp, we ensure linearization write.
    - Metadata is then written into the Replicated Database ZTree in each server in ztree_handlers.go
  - However, despite the linearization write, only eventual consistency for read is guaranteed. This is because only the Leader could **`StartProposal`** while all servers could respond to Read Request, which might lead to stale values.

- Demo: we show 2 main data synchronization scenarios as shown in Figure 4:
  - Scenario 1: Write Request to Leader: Proposal, Ack, Commit
  - Scenario 2: Write Request to Follower: Forward, Proposal, Ack, Commit

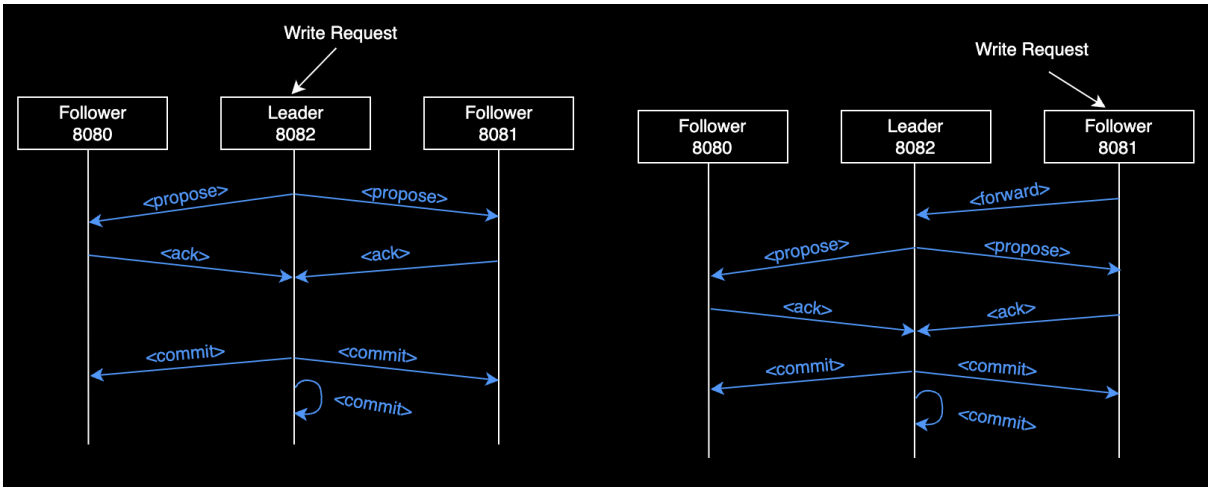

Figure 4: Data Synchronization with Write Request to Leader (left) or Follower (right) using a 2 phase commit protocol (Reference: Apache Active Messaging)

b. Distributed Coordination

- Definition:
  - **Health Check**:
    - to monitor the health and status of individual ZooWeeper servers and the ensemble as a whole.
    - each ZooWeeper server periodically conducts liveness checks to determine whether another server is alive and is responsive to client requests.
  - **Leader Election**:
    - The leader has seen the highest zxid of all the followers.
    - A quorum of servers has committed to following the leader.
- Implementation:
  - **Health Check**:
    - **`StartHealthCheck`** goroutine in zab
  - **Leader Election**:
    - We used the Bully Algorithm, ensuring not just the quorum but the whole Ensemble will identify who the new leader is.
    - **`ListenForLeaderElection`** goroutine in zab
    - **`ElectionOps`** in election_ops.go
- Demo: we show the regular Health Check and Leader Election protocol in Figure 5:
  - Scenario 1: regular Health Check
  - Scenario 2: Leader Election triggered when single server join
  - Scenario 3: Leader Election triggered when multiple servers join
  - Scenario 4: Leader Election triggered when Leader server fails
  - Scenario 5: no Leader Election triggered when Follower server fails

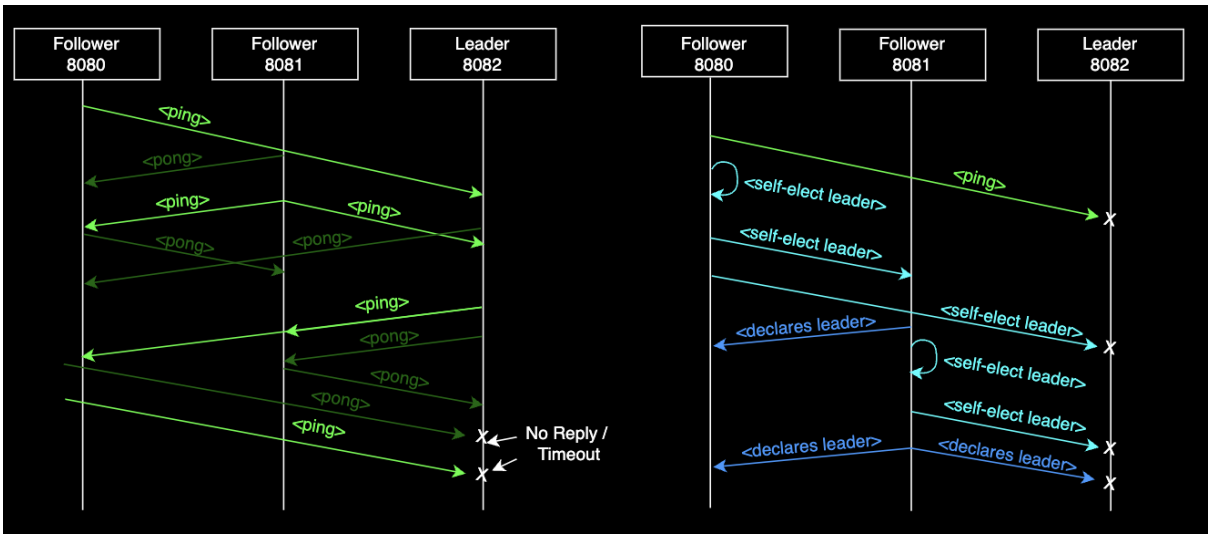

Figure 5: Distributed Coordination with Health Check (left) and Leader Election (right)

5. Correctness/Consistency

a. Definition

Our ZooWeeper implementation ensures Linearization Write, FIFO client order and eventual consistency as explained above in the Data Synchronization section.

b. Our Design

- We have a shell scripts ./send_requests.sh that simulates our Producer sending write requests to a randomly picked Kafka broker
    - Each Event Data is a football goal result (Figure 2) with the following fields:
        - Id: uid of Event Data
        - Minute: recorded time of the goal
        - Player: the player who scored
        - Club: the club that scored
        - Score: score at that point
    - All fields are randomly generated except the Minute field which is incremented in order (from 1). This simulates a global timestamp.
- Read requests are simulated using our Consumer - the Frontend application (link) showing the Event Data from our Kafka Mimic Queue:
    - Upon refresh, the Frontend application would pick a random Mimic Queue (Kafka broker) to display the Event Data
- Since we deploy multiple ZooWeeper servers, Mimic Queue (Kafka broker), and Frontend applications, we expect the following results:
    - **Eventual Consistency:**
        - Upon refreshing the Frontend application, the data displayed should eventually match all the write requests sent by the Producer (shell script)
        - There might be delays since only the Leader server processes the request and all transactions are carried out in order.
    - **Linearization Write + FIFO client order:**
        - Event Data are ordered by the filed “Minute” in the Frontend application
        - Event Data across all Kafka brokers are synchronized. This means when checking different Fronted applications or refreshing the same one, the data displayed should be the same (eventually)

c. Test Cases

- Scenario 1: Send 100 requests all to Leader server
- Scenario 2: Send 100 requests all to Follower servers
- Scenario 3: Send 100 requests randomly to the Leader and Followers

6. Scalability

a. Definition

- **Distributed Coordination**: Processing responsibilities are distributed across multiple nodes in the system. Read requests are done locally at each of the servers whereas Write requests are done only on the Leader.
- **Horizontal Scalability**: our system can easily be scaled up, such as additional ZooWeeper servers, Kafka brokers, or Frontend applications. This helps increase the Read throughput as all Read requests are done locally

b. Our Design

- Each components in all Kafka use-case (ZooWeeper server, Kafka broker and Frontend application) has a Dockerfile allow us to containerize the service and deploy using docker-compose
- We wrote a shell script ./deployment.sh::
    - First, auto-generate the docker-compose.yaml file with your chosen number of ZooWeeper servers
    - Second, run our Kafka use case with docker-compose up

c. Test Cases

We tested the scalability by sending Write Request with increasing number of servers:

Number of Servers	Response Time for 1 request (s)	Response Time for 10 requests (s)	Response Time for 100 request (s)
3	0.261195	2.22739	21.3898
5	0.326156	2.26135	22.5576
7	0.337329	2.40302	23.191
9	0.333107	2.4843	24.343

Analysis:

1. **Minimal Impact on Response Time with Server Increase**: increasing the number of servers from 3 to 9 has a marginal impact on response times for a single request. This implies that the system scales well in terms of handling a larger number of servers without a significant increase in latency for individual operations.
2. **Response Time Growth with Request Volume**: As the volume of Write Requests increases, the response time also rises. However, the growth is not exponential; for instance, going from 10 to 100 requests doesn't lead to a tenfold increase in response time, indicating that the system can handle higher loads reasonably well.

7. Fault Tolerance

a. Definition

- **Permanent Fault** such as server crashes:
  - when a server stops and no longer provides service
  - Impact on our system:
    - **Loss of Redundancy:** when a follower fails, its role and its metadata must be taken over by others, which can strain the system.
    - **Service Disruption:** when a leader fails, service might stop for a while when the leader election process is going
  - Simulation: `docker stop <container_name>`

- **Intermittent Fault** such as server crashes then restart:
  - when some servers are cut off from the rest
  - Impact on our system:
    - **Loss of Consistency:** all servers might no longer share the same view of the data, especially in write/read operations.
    - **Unavailability:** unreachable servers might prevent processing of requests or access to data if a majority is not reached
    - **Constant Service Disruption:** constant election process is required when a server fails and rejoins the network
  - Simulation: `docker restart <container_name>`

b. Our Design

- **Permanent Faul**t:
  - With Data Synchronization and Distributed Coordination features above, once a Leader server fail, a new Leader would be promoted with the same Replicated Database, allow normal operation
  - As we make use of the majority quorum for **`ProposalOps`** in proposal_ops.go, normal Write Requests are still handled as per normal when f servers fail given 2f+1 server in the ZooWeeper ensemble
- **Intermittent Fault**:
  - Due to the Loss of Consistency experienced as the server restarted, we added an additional synchronization protocol similar to the 2PC for Data Synchronization of Write Request above (Figure 6)
  - It is implemented as **`SyncOps`** in sync_ops.go

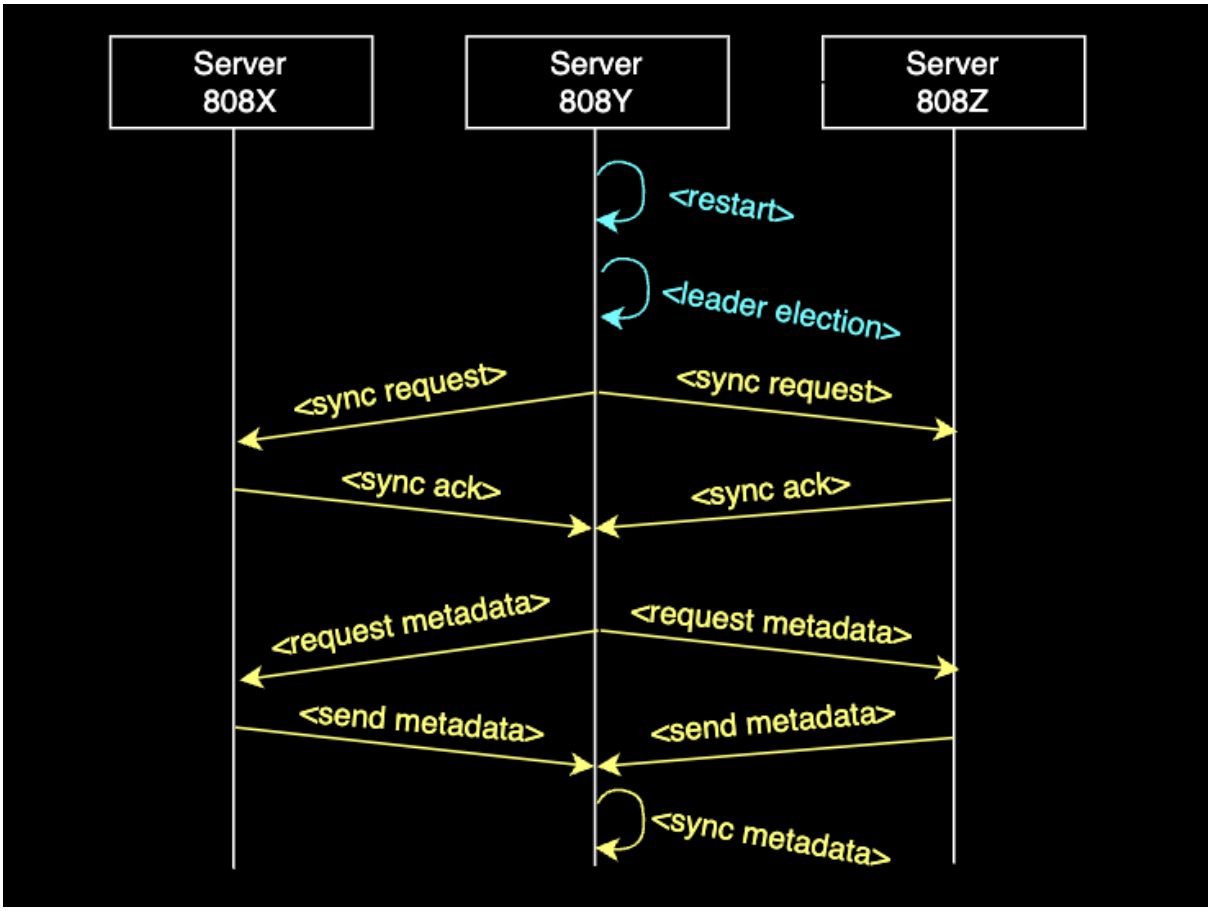

Figure 6: Synchronization protocol using a modified 2PC when a server restarts

c. Test Cases

Failure could be simulated using docker as detailed above. However, as docker-compose combines the logs of all containers together, we decided to simulate the failure by launching manually in multiple terminals instead. This allows us to show coloured log output which is easier to interpret. As we simulate the failure scenarios in our ZooWeeper server, Kafka brokers and Frontend application, we ensure our happy flow (i.e. Ordered List of Football Goals Timeline) in the Frontend application is still functional. However, the focus is still in the ZooWeeper ensemble.

- Scenario 1: Permanent Fault - Kill ZooWeeper Leader server
- Scenario 2: Intermittent Fault - Kill ZooWeeper Leader and revive
- Scenario 3: Permanent Fault - Kill Kafka broker
- Scenario 4: Permanent Fault - Kill Frontend application

8. Limitations

1. By design, the original implementation (Reference: ZooKeeper) used TCP which ensures FIFO client order directly. However, we used HTTP so we needed to add a **`QueueMiddleware`** that arranges the client to write requests in a Priority Queue based on the received Timestamp. This adds an extra layer of overhead.
2. Instead of using the filesystem, we used sqlite3 database to store metadata in our ZooWeeper server:
    a. This requires us to add an extra field (**`ParentId`**) to capture the hierarchical relationship.
    b. Additionally, we cannot use the atomic file operation (open, read, write) and have to make use of mutex and busy wait for state changes (e.g **`StartProposal`** in zab) to ensure all transactions are atomic.
3. We combined both the local metadata (for the ZooWeeper ensemble) and client metadata (for Kafka clutter) in a single sqlite3 database (Figure 3). This design is very specific and only works for our Kafka use case. Further refactoring and adjustment into multiple databases will be needed if we are to use our ZooWeeper for another application.
4. We only support Regular/Permanent ZNode since we only need this for our Kafka use case. In the future, we could explore Sequential and Ephemeral Znode
5. We did not implement Session Management which simplifies the implementation quite a bit:
    a. No more Ephemeral Znode as there is no session to keep track of when to remove the ZNode
    b. No watch mechanism for notification
    c. No sync feature to wait for all Write before Read
    d. The impending transactions in the **`QueueMiddleware`** cannot be recovered
6. What might happen if a server keeps crashing and revive? Even though we use the classic 2-phase commit for Data Synchronization, a majority quorum is also used to ensure liveness. Therefore we don't foresee any performance bottleneck. Still, further testing needs to be done to verify.

9. Concluding Remark/Reflection

Implementing a distributed system like ZooKeeper requires a deep understanding of consensus protocols, fault tolerance, and data synchronization across multiple nodes/servers. This process was both challenging and enlightening, offering a hands-on experience with the complexities of ensuring data consistency and service availability in the face of network partitions and server failures. Through this project, we have gained a profound appreciation for the robust engineering that goes into creating systems that form the backbone of many of today's scalable applications.

References

1. Zookeeper Internals: https://zookeeper.apache.org/doc/r3.9.0/zookeeperInternals.html
2. Apache Zookeeper Java implementation: https://github.com/apache/zookeeper
3. Zookeeper Paper: https://pdos.csail.mit.edu/6.824/papers/zookeeper.pdf
4. Zab Paper: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5958223