学校代码： 10385　　分类号：

学　　号： 1425161012　　密　级：

华侨大学
HUAQIAO UNIVERSITY

本科毕业论文

使用 Zookeeper 来构建具有 Consistent 和 Fault-Tolerant 特性的分布式键值存储系统

ZiStore: A Consistent, Fault-Tolerant Distributed Key-Value Storage System using Zookeeper

院（系）： 计算机科学与技术学院

专　　业： 软件工程 2 班

届　　别： 2019 届

学　　号： 1425161012

姓　　名： 罗智文

指导老师： 柳欣　副教授

企业导师：

论文提交日期：二零一九年五月二十三日

Abstract

For the current distributed application, Zookeeper is a distributed coordination service from Yahoo's open source project. Zookeeper can provide a set of primitives that distributed application can build higher-level services for synchronization, configuration maintenance, and groups and naming. Therefore, Zookeeper has become a powerful mechanism to implement the current popular distributed system.

Besides, the present data store tends to be distributed to enable the scaling of the data across multiple instances of commodity hardware. So, Zookeeper is a powerful tool to construct a distributed database due to the features of Zookeeper. For example, Zookeeper can mainly manage leader election and configuration metadata. Accurately, Zookeeper can track the status of the master server and slave servers for leader election. Indeed, Zookeeper can handle failure detection in the distributed system by managing the configuration metadata.

Currently, consistency and fault-tolerant to the network partition are two significant problems in the distributed file system. In this paper, the objective of this project is to design a distributed file system, which is a consistent, fault-tolerant distributed key-value store using Zookeeper. We can call this system as ZiStore. To make this distributed file system more completed, ZiStore puts the partition awareness in this system so that the node can automatically recover in case of failure. This system would implement a distributed key-value store with a rotating lock server to guarantee the successful leader election. This system would use a sequential consistency model to apply durable replica consistency.

In conclusion, this paper has made a significant contribution to the exploration of using Zookeeper as the leading service to design a key-value store. After testing this system, this paper shows that ZiStore can guarantee consistency and increases almost 10% overhead compared with local networks. After summarizing the advantage and disadvantage of this distributed file system, the future work becomes probably by improve the key-value store and use other protocol to make the query more effectively.

Keywords: Distributed File System, Zookeeper, Key-Value, Consistency, Fault tolerance, Partition Awareness, Sequential consistency

Contents

Introduction

With the development of the Internet, information and data grow explosively. Therefore, using the scale-up method[1] is difficult to improve system performance. The scale-out way[1] becomes a better solution to solve the bottlenecks of storage systems. However, designing a distributed system is more complicated than a signal machine. There are many problems in a distributed system, such as consistency, node failure, disk failure, and network partition. According to Brew's CAP Theorem[2], no distributed system can achieve consistency, availability, and partition tolerance at the same time. Stonebraker[3] highlighted that consistency and fault-tolerant(system availability) should be a better design choice.

Replica consistency is a significant issue of distributed systems. Any inconsistency in replicas is intolerable for some applications. Currently, there are some popular replica consistency protocols such as two-phase commit protocol[4] and sequential consistency model[5], etc. With three or more replicas, the two-phase commit protocol may not guarantee durable consistency and fault-tolerant. Therefore, the sequential consistency model becomes the best solution to ensure durable replica consistency and system availability.

For fault-tolerant (System Availability), the node failure frequently happens in a distributed system when running on general servers. So, how to provide service after a node is down is the problem that we should solve.

This project presents a new distributed key-value storage system, called ZiStore, which mainly uses Zookeeper, a service for coordinating processes of distributed applications, to implement a distributed file system, which is a consistent, fault-tolerant distributed key/value store. Based on the features of Zookeeper such as critical infrastructure, Zookeeper[6] can provide a high-performance kernel for implementing more powerful coordination primitives at the client. What's more, Zookeeper[7] has the interface which has the wait-free aspects of shared registers with an event-driven mechanism, and it is like cache invalidations of the distributed file system. Therefore, Zookeeper can provide a vital coordination service which can implement a consistent, fault-tolerant distributed key/value store. Indeed, the Zookeeper[7] interface supports a

high-performance service implementation — a per client guarantee of linearizability and FIFO execution of requests for all requests. Linearizability[8] is also a useful function of Zookeeper. These functions can enable a high-performance processing pipeline with reading requests on local servers with rotating lock.

A key-value store[9] is a type of NoSQL database. It has a simple interface with only a two-column table like a hash map. Each record has two fields: key and value. The type of benefit is string or binary; the kind of key can be integer or string/binary. There are many implementations and design of the key-value store, including in disk persistent and memory based. In-memory based key-value store is often used for caching data; disk persistent key-value store is used for storing data permanently in the file system.

This distributed file system has three essential features:

- Consistency: Any node can create Key/value pairs. Any node in the Zookeeper session can query the system to find the current value of a key in the key-value database. Replication will be sequentially consistent, which means that no node will be able to read an out-of-date amount.
- Fault tolerance: The distributed storage system overall will tolerate the simultaneous failure of a minority of nodes. Any data that was held only by those nodes could be lost, but the system will otherwise continue to function the operations.
- Partition awareness: If a node or participant becomes aware that it is in a minority partition, it will cease to operate and abort any pending operations.

Chapter 1 Distributed File System and Zookeeper

1.1. Introduction

A File system[10] mainly defines how files and directory structure is maintained. A file system functionalities are direction management, file management, space management, and metadata management. A distributed file system is a sophisticated file system with data stored on a server. The data was stored on the local client machine. The distributed file system would allow access and process the data.

The distributed file system can share information and files among users on a network either by controlled or in an authorized way. In DFS(Distributed File System)[1], the server allows the client users to share the store data and files because of storing the information and the data locally. However, the servers have a permit to control the data and give access control to the clients. This paper would mainly talk discuss how to implement a distributed file system, the goals and challenges of designing a distributed file system without focusing on analyzing the distributed file system.

For the structure of the distributed file system[10], there are three main vital components, such as client, server, and service. A service is a software entity running on one or more machines, and providing a particular type of function to a priori unknown clients. A server is a service software running on a single computer. Client process that can invoke a service using a set of operations that forms its client interface. A collection of first file operations forms a client interface for a file service (create, delete, read, write). Client interface of a distributed file system should be transparent, not distinguish between local and remote files. Figure 1.1 shows an overview of a distributed file system.

There are four goals of design a distributed file system[10]. First, the distributed file system can share filesystem that will look the same as a local filesystem, which could make the distributed file system look like a complete system. Second, the distributed file system needs to scale to many TB's of data or many users. Then, the fault tolerance is also an important feature, which is a non-functional requirement in the distributed

file system. Finally, the distributed file system has a high-level performance compared with a non-distributed file system.

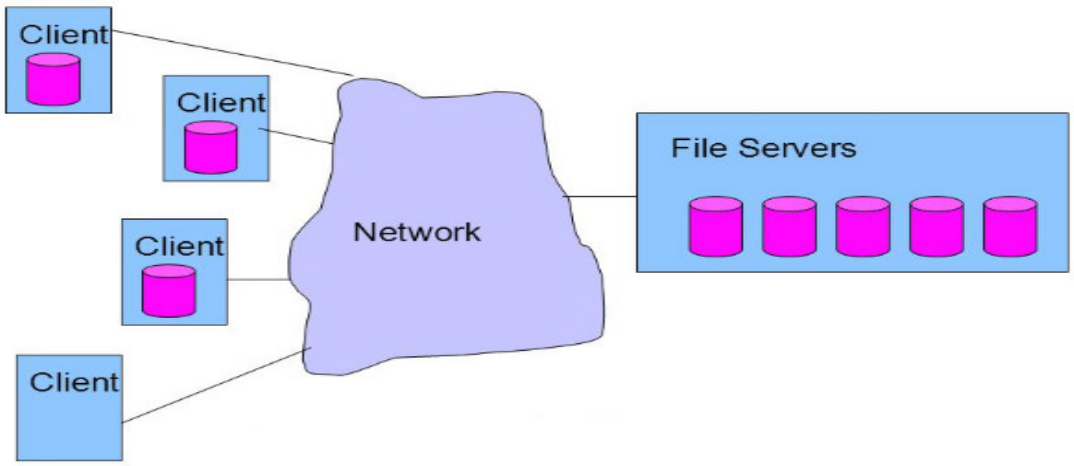

Figure 1.1 Overview of DFS

On the other hand, there are also four main challenges[11] while implementing the distributed file system. First of all, heterogeneity is a big problem which would block the processing of the distributed file system. Specifically, heterogeneity means different kinds of computers could have different types of network links. Second, if there are lots of users in the distributed file system, the scale also would be an issue. The distributed file system needs to assess the maximum number of user before beginning to implement. Because the file system would share information and files among users on a network, the security of the users also could cause the problem. But, the essential parts are concurrency. The distributed file system need to promise the Thread-safe during processing.

1.2. Zookeeper Advantage

Zookeeper[6] is a distributed coordination service from Yahoo. Initially, the Zookeeper maintained as Apache project and used widely. The Zookeeper has three powerful features: highly available, fault tolerant and performant. The advantage of Zookeeper is that the user doesn't need to implement Paxos[12] for maintaining group membership, distributed data structures, distributed locks, and distributed protocol. The Zookeeper also has three guarantees: liveness, atomic updates, and durability.

1.3. Zookeeper Guarantees

- Liveness: if a majority of Zookeeper servers are active and communicating the service will be available.
- Atomic updates: A write is either entirely successful or entirely failed.
- Durability: if the Zookeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover.

1.4. Zookeeper Session

For the sessions of the Zookeeper[7], each client maintains a meeting with a single Zookeeper server, and meetings are valid for some time window. As shown in figure 1.2. Also, If the client discovers it's disconnected from the zookeeper server, attempts to reconnect to a different server before the session expires.

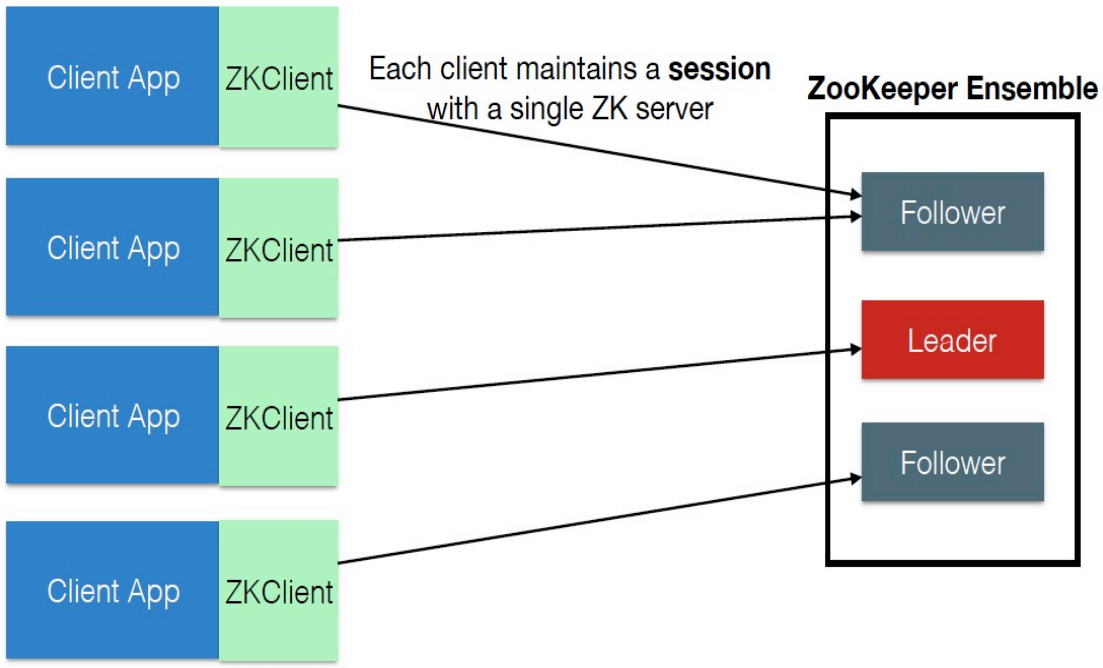

Figure 1.2: Each client maintains a session with a single Zookeeper

With the Zookeeper session alive, the processing of reading value is showing in Figure 1.3:

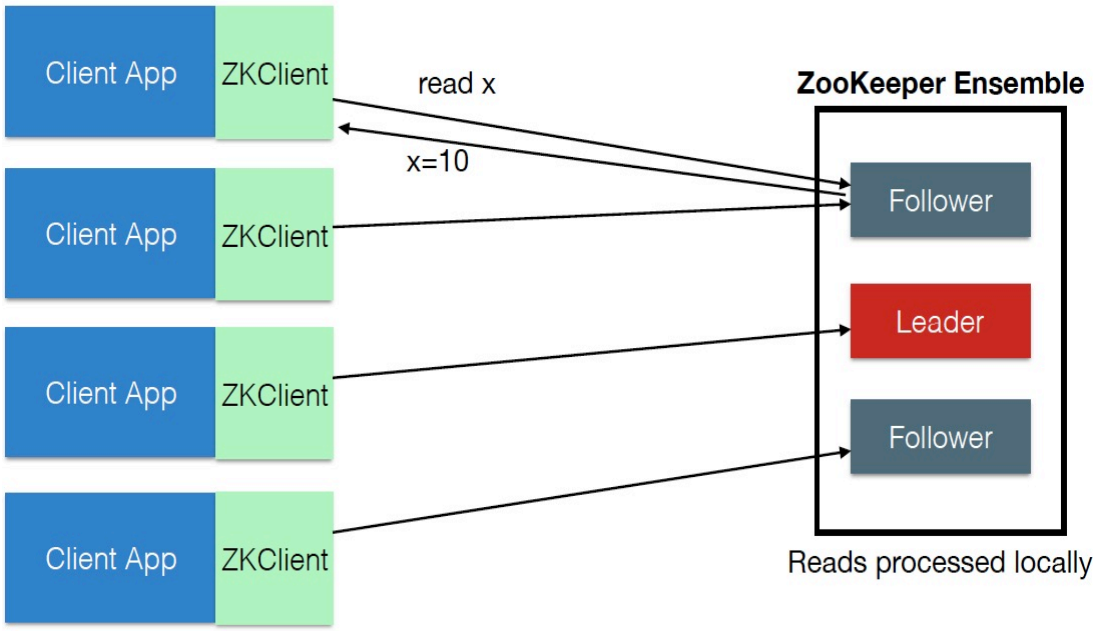

Figure 1.3: Client reads a value with a Zookeeper session

With the Zookeeper session live, the processing of writing value is showing in Figure 1.4:

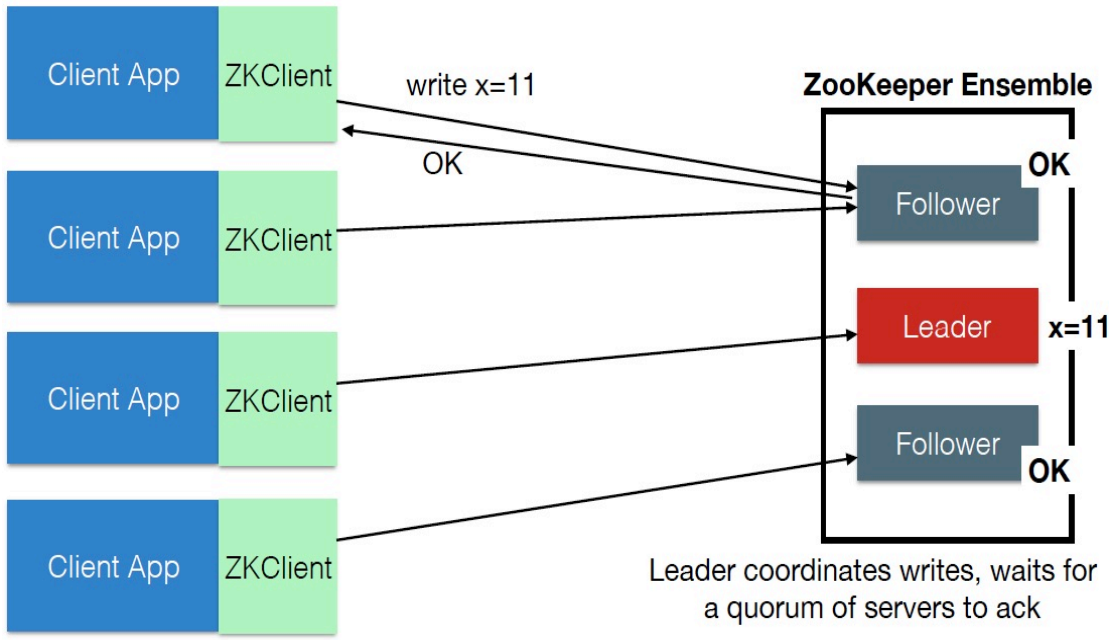

Figure 1.4: Client writes value with a Zookeeper session

1.5. Data Model

The data model of Zookeeper[7] provides a hierarchical namespace, and each node is called a Znode. Zookeeper offers an API to manipulate these nodes. The namespace provided by ZooKeeper is much like that of a standard file system. A name is a sequence of path elements separated by a slash (/). A path identifies every node(Znode) in ZooKeeper's namespace. Correctly, Zookeeper's hierarchical namespace and name are shown in Figure 1.5:

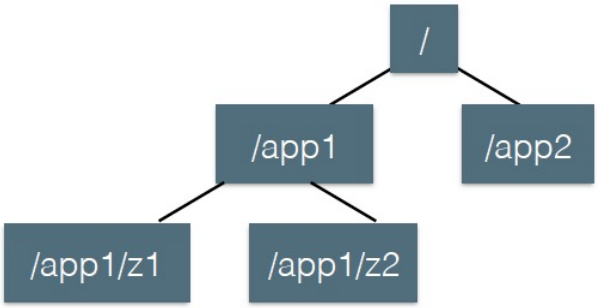

Figure 1.5: Zookeeper's hierarchical namespace

1.6. Znode Types

Znodes[6] is in-memory data, but not for storing general data-just metadata. Znodes map to some client abstraction such as locks. Also, Znodes maintain counters and timestamps as metadata. Zonde has two different types: Regular Znodes and Ephemeral Zondes.

- Regular znodes can have children znodes, and created and deleted by clients explicitly through API.
- Ephemeral znodes cannot have children and create by clients explicitly. Also, Ephemeral znodes deleted by clients or removed automatically when client session that created them disconnects.

1.7. The Consistency of Zookeeper

- Sequential consistency of writes: All updates are applied in the order they are sent, linearised into a total rule by the leader.
- Atomicity of writes: The consequence of updates is either succeed or fail.
- Reliability: Once writing has been applied, it will persist until its overwritten, as long as a majority of servers don't crash.
- Timeliness: Clients are guaranteed to be up-to-date for reads within a time bound – after which you either see the newest data or are disconnected.

Zookeeper[7] can provide the sequential consistency of writes, which match the features of this project. To make this project useful and powerful as a distributed field system, it has to guarantee that any node can query the system to find the current value of a key, and no node will be able to read an out-of-date value, which requires the sequentially consistent. So, Zookeeper becomes one of the most useful coordinate services to build this system.

1.8. Failure Handling in Zookeeper

Apps using Zookeeper[6] need to be aware of the potential failures that can occur and act appropriately. Zookeeper client will guarantee consistency if it is connected to the server cluster. This behavior is shown in Figure 1.6.

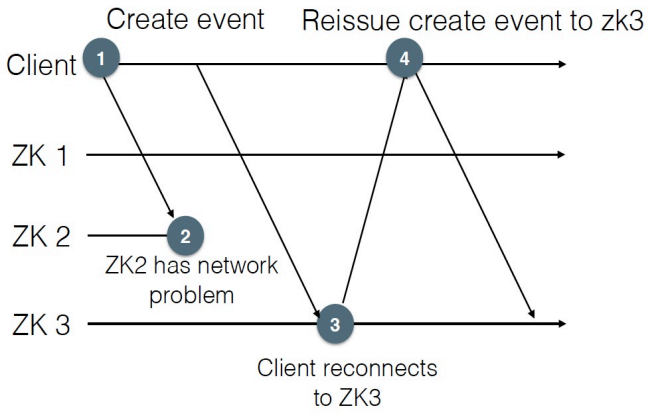

Figure 1.6: Client is connected to the server cluster

Specifically, if in the middle of an operation, the client would receive a **ConnectionLossException**. Also, the client gets a disconnected message. Client can't tell whether or not the transaction was completed though – perhaps it was completed before the failure. Clients that are divided cannot receive any notifications from Zookeeper.

1.9. Dangers of ignorance

Each client[6] needs to be aware of whether or not it's connected: when disconnected, cannot assume that it is still included in any way in operations. By default, the Zookeeper client will not close the client session because it's disconnected. As shown in Figure 1.7, even if the client1 is disconnected, Zookeeper keeps the client 1 session alive until client1 reconnected.

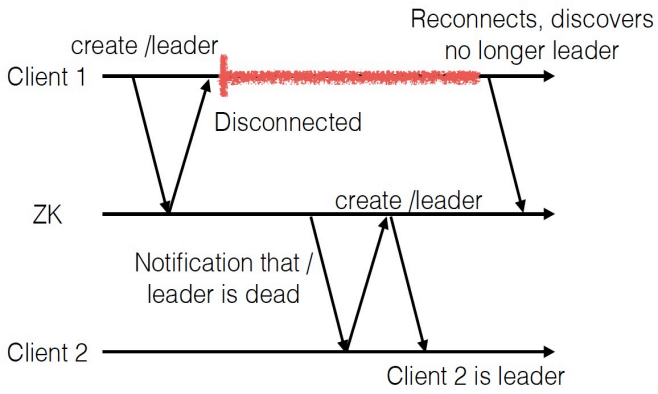

Figure 1.7: Zookeeper client will not close the client session

Chapter 2 Related Work

The scholars have made a lot of different researches on the Zookeeper. For example, according to "*Zookeeper: wait-free coordination for internet-scale systems*," Patrick and Mahadev demonstrates that ZooKeeper[6] achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas. For the Zookeeper Coordination Service[6], Raluca etal gave a scalability solution which consists in partitioning the service state. For the Consistency and Fault Tolerance[12], Paolo Viottl and Marko Vukolic, from IBM Research, provide a structured and comprehensive overview of different consistency notions that appeared in distributed systems.

Vineet, from the University of Waterloo, mentions the intention[13], which is to bring some features to the distributed key-value store. What's more, according to "*Precomputing architecture for flexible and efficient big data analytics*," Nigel and Xuguang etal highlights that Zookeeper[14] provides specialized cluster management, which has better integration with other big data platforms. Thanh and Minh demonstrate that key-value stores[15] become more and more important in cloud storage services and large-scale high-performance applications. Then, they design a database using Zookeeper for coordination of the configuration of the database instance in "*Zing Database: high-performance key-value store for large-scale storage service*." Based on "*Cloud cover: monitoring large-scale clouds with Varanus,*" Nguyen and Nguyen use the ZooKeeper[16] in cloud computing because it provides fault-tolerant mechanisms for service discovery, coordination, and orchestration.

In Industrial production areas, using Zookeeper in the storm to coordinate the state of the machines in the synchronous cluster, it doesn't need a high load request on the machine performance. For example, Jstorm also uses Zookeeper for consistency services. Moreover, Dubbo uses Zookeeper as the registration centre. In Alibaba, Zookeeper used as a database-based registration centre, because the service would keep available even though the half of the machine has broken in the Zookeeper cluster. Specifically, the consumer and provider of the Dubbo service is based on the Zookeeper.

However, the zookeeper participation would become weaker (Only accepting some heartbeat packets) after the long connection between the consumer and the

provider, unless there is a new provider or consumer which joins or leaves and need to update the node data. Therefore, the load is also very low so that the company doesn't need to consider performance issues. What's more, Zookeeper also plays a role in monitoring various services. HBase uses Zookeeper, and Kafka clusters also rely on ZooKeeper. Kafka realises load balancing and dynamic cluster expansion of producers on the consumer side through ZooKeeper.

Currently, Brew's CAP theorem[2] makes a significant contributed to the distributed systems because the theory shows that it is impossible for all distributed system having the three following guarantees: consistency, fault-tolerant (Availability) and partition tolerance. Therefore, most of the distributed storage system would choose two of the above goals based on the theorem. Besides, most relate national databases use the two-phase commit protocol, such as MySQL. It can have strong consistency, but the availability and partition tolerance are poor.

Nagarajan and Gupta[5] provide an efficient approach for enforcing sequential consistency without requiring post-retirement speculation, which is a consistency model based on message passing. In sequential consistency, there is some total order of operations. Sequential consistency requires reads and writes to be ordered in program order, which can cause significant performance overhead.

Zookeeper[6] uses basic Paxos[13] algorithm to select the leader node, which controls the data update. If the older master fails or disconnects, the Zookeeper session will choose a new leader(Master). The Zookeeper can guarantee strong consistency. Because Zookeeper's design goal is high availability service and provides distributed lock service for other distributed system, it is not a dedicated distributed storage system. Therefore, this project needs to use sequential consistency to improve performance.

In sum, this project would design Zistore by using Zookeeper cluster and sequential consistency model. What's more, this project also would use some optimization techniques, such as rotating lock server. It would guarantee not only strong consistency but also keep the system fault tolerance.

Chapter 3 Design of Storage System

3.1. Architecture

In this project, we define an interface for distributed key-value store implementation. Based on the key and value of each record, all data are divided into different ranges. The architecture of ZiStore is shown in Figure 3.1.

In ZiStore, the essential components of this project include client, Coordination service(Zookeeper), and storage server. Specifically, the storage server[10] comprises the leader and the followers. Following Figure 3.1, the replica's number is three. There are two followers and one leader in Figure 3.1. Based on Figure 3.1, both leader and follower can provide read service, but the client only sends write request to the leader.

However, the client also can send a writing request to the follower, but the follower would first give this write request to the leader. After the leader finishes the write request, the leader would synchronize data to followers based on the sequential consistency model and fail-over protocol, which would require analysis later. In this project, ZiStore uses Zookeeper for auxiliary election to simplify the leader election process.

Besides, Zookeeper[7] as a coordination server also monitors the leader and follower's state. Therefore, this storage system can find a new leader automatically. Based on the sequential consistency model, each write request should be assigned an order number to indicate its execution order. If the computer creates a new node in this storage system, the node will run a Zookeeper client and connect with zookeeper server. At the same time, the leader would synchronize data to it. Finally, the local storage engine(cache) is a key-value store, which is a simple database: data(values) are stored and indexed by a key – effectively a big HashMap.

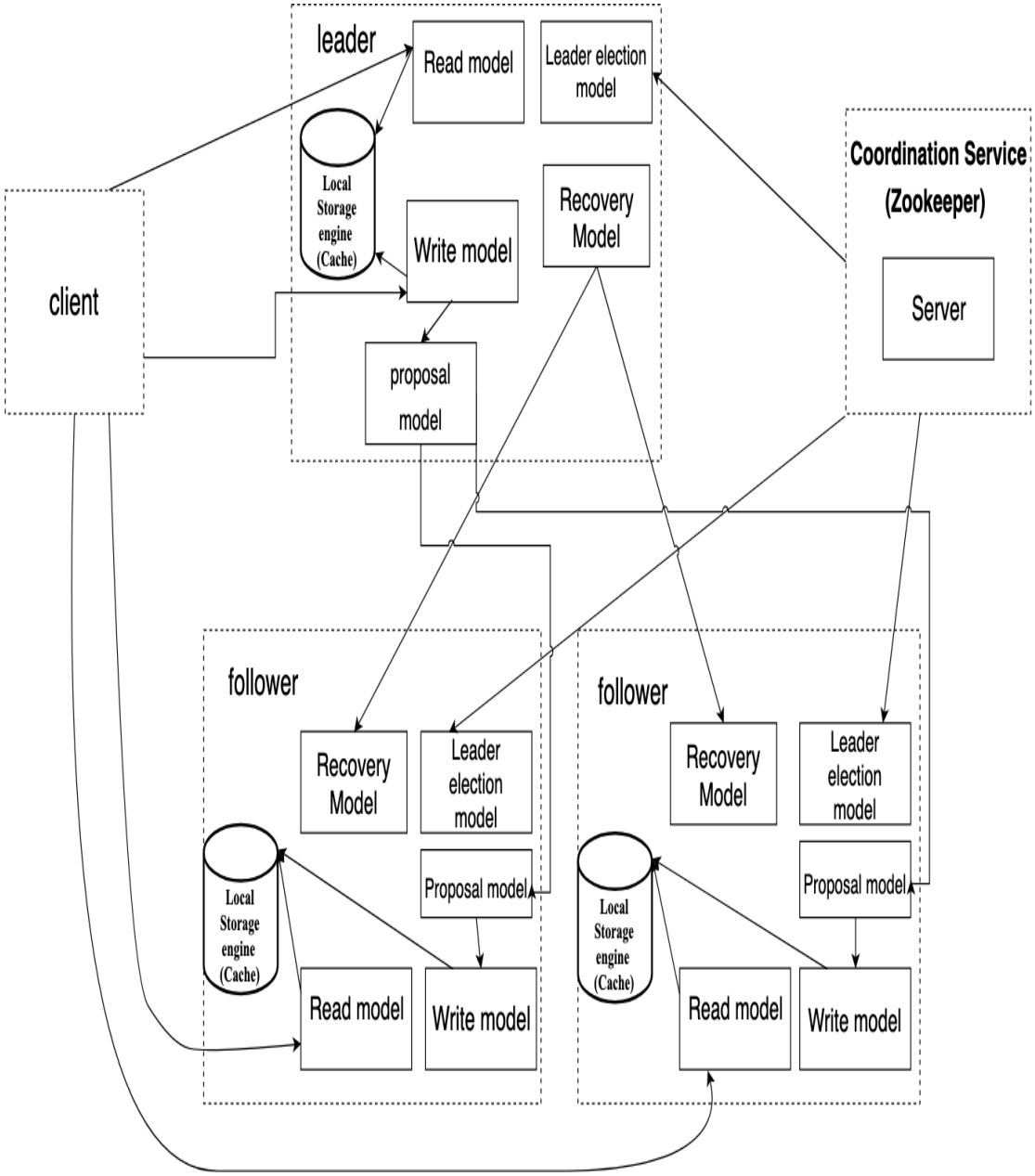

Figure. 3.1 ZiStore Architecture

3.2. Sequentially Consistent Model

Sequentially consistent model[5] requires and writes to be ordered in program order, which can cause significant performance overhand. In sequentially consistent, each CPUs operation appears in order. All CPUs see results according to that order(read most recent writes). Figure 3.2 shows an example of sequentially consistent.

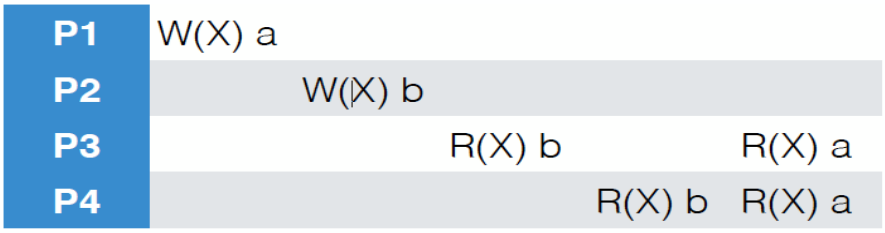

Figure 3.2 Sequentially Consistent Model

In this project, this system uses exclusively reentrant locks to guarantee sequentially consistent. Specifically, this distributed storage system uses a synchronized API to implement sequential consistency model.

3.3. Protocol Analysis

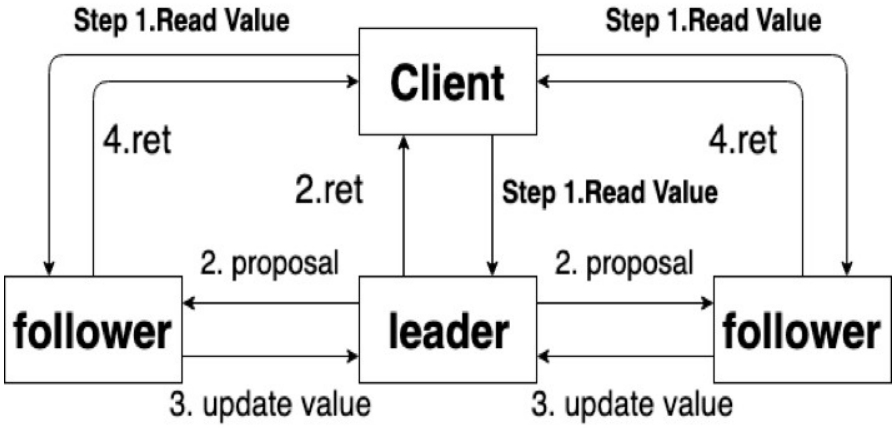

Figure 3.3 Read Protocol Flow Chart

The necessary process of the distributed read protocol used by ZiStore is shown in Figure 3.3.

To read an essential Key, if the node(leader or follower) currently has the value for that critical cached (e.g., from previously reading it or writing it), then the system will directly return the value of the key, without contacting any other node or leader. Otherwise: To read a key K, a node N will contact the leader and retrieve the value. If the leader does not have the key, it will return ***null*** or throw an exception. Clients must not cache ***null*** values. If the leader has the key on its local cache, it will record the fact that node N now has a cached copy of this key's value.

(1) If the node is disconnected from Zookeeper: If node N does not currently hold a live Zookeeper session (e.g., it timed out), then it will throw an **IOException** for any read operation. It is OK to return a stale, cached value if node N is disconnected from Zookeeper but has not detected that disconnection yet.

(2) If the node is not able to reach the leader and the leader is disconnected from Zookeeper: If node N is unable to contact the leader but does currently hold a live Zookeeper session (and the leader does not have a live Zookeeper session), then it will first participate in the leader election described below, and then complete the read as described above.

(3) If the node is not able to reach the leader, but the leader is still active in Zookeeper: If node N is unable to contact the leader, but Zookeeper indicates that the leader and client are both still dynamic in Zookeeper, then the client must wait for the leader to become available (this is the default RMI behaviour when an RMI endpoint is not reachable). You do not have to consider the case of the leader being available on Zookeeper at the start of an operation, and then disconnecting from Zookeeper while a transaction is in progress.

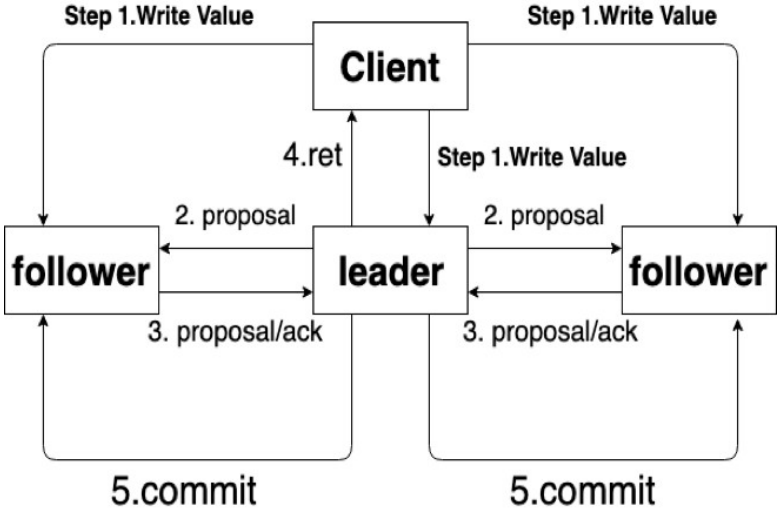

Figure 3.4 Write Protocol Flow Chart

The necessary process of the distributed write protocol used by ZiStore is shown in Figure 3.4.

To write a key K, node N will ask the leader to update the value. The leader will:

- Notify all clients to invalidate their cache, and clear its list of clients which have this key cached.
- Update the value.
- Add node N to the (now emptied) list of clients with this key cached.
  (only one write to a key can occur at a time, and no reads can occur during a write.)

(1) The node that wants to write the value is disconnected from Zookeeper: If node N does not currently hold a live Zookeeper session (e.g., it timed out), then it will throw an **IOException** for any write operation.

(2) If the leader is disconnected from the client: If node N is not able to contact the leader, but does currently hold a live ZooKeeper session (and the leader does not), then it will first participate in a leader election described below, and then complete the write as described above.

(3) If the node is not able to reach the leader, and the leader is disconnected from Zookeeper: The leader must not begin any writes until it validates that it holds a valid Zookeeper session. If it finds it does not hold a valid Zookeeper session, it must throw an **IOException**.

(4) If the leader is unable to contact a client with a cached version of that key: The leader must wait until all **invalidate** messages are acknowledged. However, if a client becomes disconnected from Zookeeper, and the leader detects this, it should ignore the failure of the **cance**l message (since Zookeeper agrees that that node has failed). You do not have to consider the case of the client being available on Zookeeper at the start of an operation, and then disconnecting from Zookeeper while an invalidate is in progress.

(5) If the node is not able to reach the leader, and the node does not have the value cached, but the leader is still active in Zookeeper: If node N is unable to contact the leader, but Zookeeper indicates that the leader and client are both still dynamic in Zookeeper, then the client must wait for the leader to become available (this is the default RMI behaviour when an RMI endpoint is not reachable). You do not have to consider the case of the leader being available on Zookeeper at the start of an operation, and then disconnecting from Zookeeper while a process is in progress. If you have the value cached, you can serve it directly.

Chapter 4 Implementation

4.1 Component of Storage Node

As shown in Figure 4.1, the essential components of a storage node include a key-value storage engine, sequentially consistent model, and a zookeeper. Indeed, the sequentially consistent model can guarantee replica consistency among multiple nodes, and this model is shown in Section 3.3. In this project, the key-value store database is the storage engine.

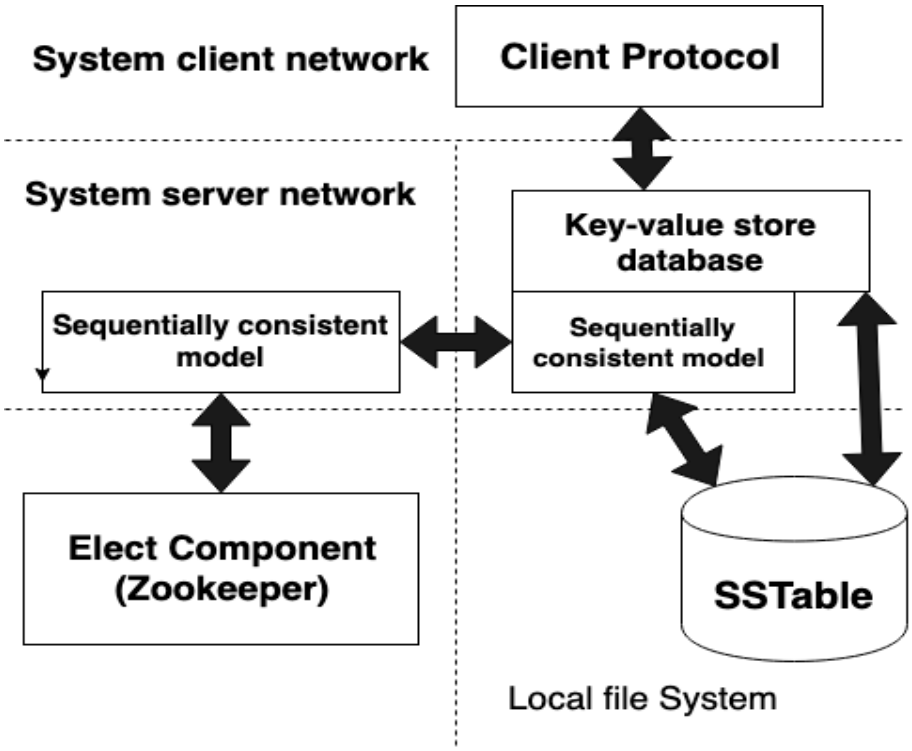

Figure 4.1 Component of Storage Node

4.1.1. Leader Logic Design

The implementation framework of the leader handles the client request by reading and writing value operation.

(1) Leader Initialization

If this is the first leader, then the leader won't need to do anything to initialize itself – there will be no data at the start, and hence no nodes will have any copies of any values, and no nodes will currently hold any ownership of any keys. However, if this is a leader being promoted after a prior leader failed or was disconnected, then it will merely initialize itself using whatever cached data it had at the time it was developed. Hence, some key/value pairs might be lost (if only the leader had them). When a node detects that the leader has changed (and that it is not the leader), it must flush its entire cache.

(2) Leader Execution Logic

For leader execution logic, the implementation of reading logic is simple; ZiStore can directly read the required data from the local database(cache). Figure 3.3 shows the process of reading logic. However, the implementation of write logic is complex, and it is the core of leader execution logic. Correctly, write thread gets write request from the client, then it will send a proposal message to the follower. After receiving all the ACK message from followers, ZiStore can write this request into the local key-value store engine and return Ret message to the client. Finally, the leader will send commit message to followers, as shown in Figure 3.4.

(3) Leader Election

As shown in Figure 4.2, if there is currently no leader (e.g., when the system starts up, or when the leader becomes disconnected), then any nodes who can still contact a quorum of Zookeeper nodes will participate in a leader election. This system will use Curator's **LeaderLatch** to perform the poll. A simple way of doing leader election with ZooKeeper is to use the SEQUENCE|EPHEMERAL flags when creating znodes that represent "proposals" of clients. The idea is to have a znode, say "/election," such that each znode creates a child znode "/election/guide-n_" with both flags SEQUENCE|EPHEMERAL. With the sequence flag, ZooKeeper automatically appends a sequence number that is greater than anyone previously attached to a child of "/election." The process that created the znode with the smallest added sequence number is the leader.

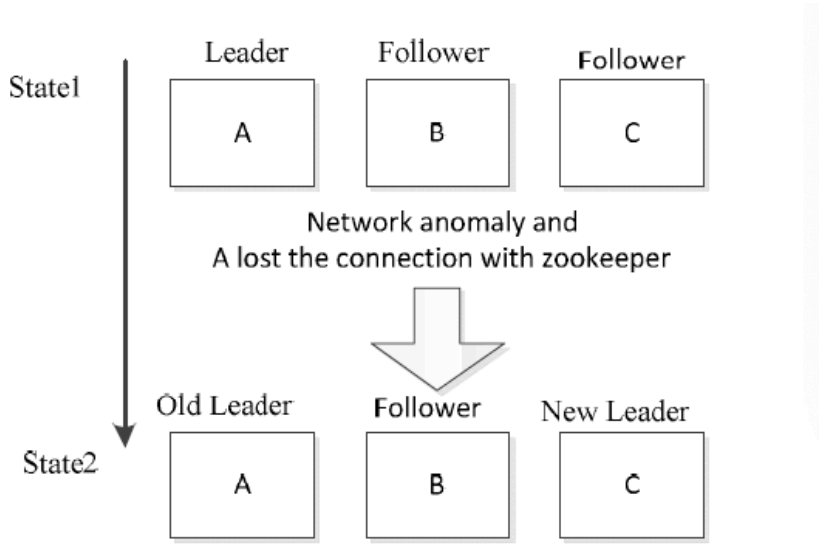

Figure 4.2 Two Leaders appear

4.1.2. Follower Logic Design

The underlying implementation framework of the follower is similar to the Leader, but the follower works simpler than a leader. The design of the read logic of follower is almost the same as the leader. However, the follower does not need to process the client to write directly. When the follower gets the write request from the client, it will send the message to the leader. After the leader receives the message, the following step change to leader writing execution logic.

(1) Initialization Node

KVServer will automatically get passed a Curator (Zookeeper) client object, To manage the connection, the system needs to: Create an Ephemeral, **PersistentNode** to represent the **KVServer** in the pool of all active servers; Create a **LeaderLatch** to elect and maintain a leader of the group. The following description would show the Instruction of initClient.

Data: localClientHostname, localClientPort

Result: null

Function name: void initClient(localClientHostname, localClientPort)

This callback is invoked once your client has started up and publish an RMI endpoint. In this callback, you will need to set-up the Zookeeper connections, and post your RMI endpoint into Zookeeper (publishing the hostname and port). This function also needs to set up any listeners to track Zookeeper events. localClientHostname is the client's hostname, which other clients will use to contact you. localClientPort is the client's port number, which other clients will use to contact you.

This system would use the Zookeeper client ***zk***, should create the ephemeral node at the path ***ZK_MEMBERSHIP_NODE*** + ***"/"*** + ***getLocalConnectString()***. This system would create the **LeaderLatch** at the way **ZK_LEADER_NODE**. Make sure to specify the **id** of the LeaderLatch to identify it (in the constructor), accurately passing ***getLocalConnectionString ()*** as the id. Figure 4.2 would the Java code:

```
myMembership = new PersistentNode(zk, CreateMode.EPHEMERAL,
    false, ZK_MEMBERSHIP_NODE + "/" + getLocalConnectString(), new byte[0]);
myMembership.start();

leaderLatch = new LeaderLatch(zk, ZK_LEADER_NODE, getLocalConnectString());
leaderLatch.addListener(new LeaderLatchListener() {
    @Override
    public void isLeader() {
        System.out.println(getLocalConnectString() + " Leader");
    }

    @Override
    public void notLeader() {
        System.out.println(getLocalConnectString() + " Not leader");
    }
});
try {
    leaderLatch.start();
} catch (Exception e) {
    // e.printStackTrace();
}
```

Figure 4.2: Java code of Ephemeral Node Setup

Key to the successful operation of your key-value store will be a global, shared understanding of which nodes are currently alive and connected to the system.

Otherwise, reads and writes might become stalled by partitioned or crashed nodes. You'll use Curator's **PersistentNode** abstraction to track which nodes are currently active participants.

After finishing the ephemeral node setup, we need to create a tree cache for tracking node membership. On the tree cache, we also need to add an event listener to track this tree cache. The meaning of this event listener is that complete the leader election.

Specifically, if there is not a leader currently, this event listener would wait until a leader has been elected. When a node detects that the leader has changed, this listener will track the new leader. If the temporary node becomes a leader, after a prior leader failed or disconnected, then it will merely initialize itself. Otherwise, if it is not the leader, it must flush its entire cache.

4.2 Algorithm of Reading and Writing

To complete this project, concurrent and distributed system are two crucial part of this project. Specifically, there are three critical features of parallel and distributed systems, such as consistency, fault tolerance, and partition awareness. What's more, Zookeeper also is an essential technology in this project, which would play a critical role in this system to control the Key/value store.

Therefore, the Key/value store would be the last technology in this project. In this project, Key/value stores are simple databases: data (values) are stored and indexed by a key — effectively a big HashMap. For the Key/Value store in this project, read and write are two crucial parts in this project. Indeed, there are many kinds of situations during reading and writing values, and some condition could cause the bug. Therefore, the following part would analyze each different location and find the salutation.

4.2.1. Reading Values Implementation

Implement the external API (which is used by the text interface and the tests) as well as the peer-to-peer API that is used by non-leaders to invoke functions on the

leader, and by the leader to invoke functions on the followers. The following description would show the Client-facing API of getValue.

Data: key

Result: value

Throw: IOException

Function name: String getValue(key)

Algorithm 1: String getValue(key)

```
if node N does not currently hold a live Zookeeper session then
    throw IOException
if node N == leader then
    return value
else
    if node N == follower then
      check if the cache contains key
    else
      if the cache doesn't contain key request value from the leader
          indicates that the leader and client are both still active in
          Zookeeper, then
              the client must wait for the leader to become available
              while (no reach leader)
                  if the leader does not have live Zookeeper session,
                  node N will first participate in the leader election,
                  leaderID the gets updated, then
                      complete the read
                      value = connectionToKVStore(leaderID). getValue
                      (key, getLocalConnectString ());
                      if value not null, update the locally cached value then
                          reachLeader = true

    return value
```

Algorithm 1: Fetching Value

This function would retrieve the value of a key. The return value is the value of the key or null if there is no such key. This function will throw IOException if this client or the leader is a disconnection from Zookeeper.

First, if node N does not currently hold a live Zookeeper session, then it will throw an **IOException** for any read operation. Then, node N needs to check itself. If it is a leader, this function can return value. Otherwise, if this node is a follower, then this function needs to check the cache. If the cache contains key, also this function can return the value from the cache.

Up to now, if the cache doesn't include essential, then the node N requests value from the leader. If node N is unable to contact the leader, but Zookeeper indicates that the leader and client are both still active in Zookeeper session, then the client must wait for the leader to become available. If the leader does not have a live Zookeeper session, node N will first participate in the leader election and complete the processing of reading value. Especially if the value is not null, the node N needs to update the locally cached value. The algorithm of getValue is shown above.

To communicate with each node, this system also needs to implement the following RMI-based API:

Data: key, fromID
Result: value
Throw: RemoteException
Function name: String getValue(key, fromID)

This function would request the value of the key. The node requesting this value is expected to cache it for subsequent reads. This command should only be called as a request to the leader. The client sends the key request. fromID is the ID of the client making the request(as return by AbstractKVStore, getLocalConnectString()). Finally, this function should return the value of the key, or null if there is no value for this key. This function does not throw any exceptions because RMI only throws the RemoteException if the connection fails.

If the leader's cache contains the key, the getValue get the list of the followers who have the key cached. After that, this operation would add the requesting follower to the

follower list with crucial cached.

4.2.2. Writing Values Implementation

Implement the external API (which is used by the text interface and the tests) as well as the peer-to-peer API that is used by non-leaders to invoke functions on the leader, and by the leader to invoke functions on the followers. The Client-facing API of setValue is shown below.

Data: key, value
Result: void
Throw: IOException
Function name: void setValue(key, value)

Algorithm 2: setValue(key, value) – part1

```
if node N == leader then
  synchronized(getCommon(key)) // getCommon will analyze in Section 4.3
     Notify all clients to invalidate their cache
     if (followerIDs != null) then
        if the leader is unable to contact a client with a cached
        The version of that key. then
              the leader must wait until all the invalidate message are acknowledged
              while ( no invalidateAllClient)
                    However, if a client becomes disconnected from Zookeeper,
                    and the leader detects this, then
                                it should ignore the failure of the invalidate message
                                clear its list of clients which have this key cached
```

Algorithm 2: setValue(key, value) – part1

This function would update the value of a key. After upgrading the value, this new value will be locally cached. This function will throw IOException if this client or the

leader is disconnected from Zookeeper.

First of all, if node N does not currently hold a live Zookeeper session, then it will throw an IOException for any write operation.

If node N is the leader, then it will notify all clients to invalidate their cache. Indeed, if the leader is unable to contact a client with a cached version of the key, the leader must wait until all cancel message is acknowledged. However, if a client becomes disconnected from Zookeeper, and the leader detects this situation, node N should ignore the failure of the invalidate message. In the end, the node N should clear its list of clients which have this key's cache. The algorithm of setValue is shown above.

But, if the follower finds the key, the follower will ask the leader to update the value. If node N is not able to contact the leader, but Zookeeper indicates that the leader and client are both still active in Zookeeper, then the client must wait for the leader to become available. If the leader does not have a live Zookeeper session, then it will first participate in a leader election. After leader election successful, the node N needs to update value into the local cache. The algorithm is shown below.

Algorithm 2: setValue(key, value) – part2
else *node N* != *leader* the follower will ask the leader to update the value **if** node N is no able to contact the leader, but Zookeeper indicates that the leader and client are both still active in Zookeeper, **then** the client must wait for the leader to become available **while** (no *reachLeader*) **if** the leader does not have a live Zookeeper session, **then** it will first participate in a leader election **then** leaderID gets update complete the write

Algorithm 2: setValue(key, value) – part2

To communicate with each node, this system also needs to implement the following RMI-based API:

Data: key, fromID

Result: value

Throw: RemoteException

Function name: void getValue(key, fromID)

This function would request the value of a key. The node requesting this value is expected to cache it for subsequent reads. This command should only be called as a request to the leader. fromID is the ID of the client making the request (as returned by AbstractKVStore.getLocalConnectionString()). This function finally would return the value of the key, or null if there is no value for this key. This function does not throw any exceptions (RMI throws the RemoteException if the connection fails).

When the leader is disconnected from Zookeeper session, if the node finds it does not hold a valid Zookeeper session, it must throw an IOException. Using synchronized notify all clients to invalidate their cache. Then, if the leader is unable to contact a client with a cached version of that key: the leader must wait until all invalidate message is acknowledged.

However, if a client becomes disconnected from Zookeeper, and the leader detects this situation, it should ignore the failure of the cancel message. Finally, this node needs to clear its list of clients which have this key's cache and update the value. Also, this function needs to add node N to the list of clients with this key's cache.

4.3 Fault Tolerance

4.3.1. Avoiding Byzantine Faults

To implement fault tolerance, this project firstly needs to solve an essential problem of this system: Byzantine Faults. In this system, there are five different situations, which could cause Byzantine Faults.

First, leader neglects to send invalidate to some client when the new leader shows up after leader election, or the follower neglects to invalidate. On this situation, the

client would get the wrong value from the follower. Besides, if the leader gives the false value for reading, it would also cause Byzantine Faults. What's more, follow pretends to be a leader, and the follower does incorrect read or write. In Figure 3.2, the client tries to read the leader and follower at the same time, which is an example of the last case.

To solve the Byzantine Faults in this project, function invalidates key (String key) plays an essential role in preventing the problem which happened during the translation of service invalidated key. The basic instruction of function invalidate key:

Data: key,
Result: null
Throw: RemoteException
Function name: void invalidateKey(key)

This function instructs a node to invalidate any cache of the specified key. This method is called by the leader, targeting each of the clients that have cached this key. This function does not throw any exceptions (RMI throws the RemoteException if the connection fails). The algorithm is shown below.

Algorithm 3 - void invalidateKey(key)

Void invalidateKey(key):
  Synchronized(getCommon(key))
  Cache.remove(key)
  Synchronized(lockMap)
  lockMap.remove(key)

Algorithm 3 - void invalidateKey(key)

4.3.2. Thread-safe

Thread safety is a computer programming concept applicable in the context of multi-threaded programs. Thread safety implies multiple threads can write/read data in the same object without memory inconsistency errors. In a highly multi-threaded

application, a thread-safe program does not cause side effects to shared data.

In this system, the thread-safe can promise that there would happen Byzantine Faults. If the node N sync read leader and follower, which tries to be a leader, then this distributed system would have Byzantine Faults. In the end, node N could get two different value, which would violate consistency. Therefore, to make this distributed system thread-safe, I use a function getCommon when retrieving the key.

This function can synchronize on a single familiar object for all the instances of Strings that are Object.equals(Object) among them (you need to synchronize concurrent access to the system for the same user). Also, it's ubiquitous to choose this single everyday object as the value of a Hashtable with the keySet being the String values. The algorithm is shown below.

Algorithm 4 - Object getCommon(value)

Object getCommon(value)
  Synchronized(lockMap)
  Object common = lockMap.get(value)
  If (common == null) then
  common = new Object
  lockMap.put(value, common)
  return common

Algorithm 4 - Object getCommon(value)

4.4 Recovery

4.4.1. Connection State

For the stateChanged, this function would firstly check the connection state. When a node is reconnected to a quorum of Zookeepers, it will check to see how many other nodes are there. If there are none, it will perform the leader election and then assume a leadership role and can initialize itself using any cached data that it has. If, however,

there are other nodes present at the moment that it reconnects (regardless of who the leader is). The node will flush its entire local cache, including all values and cache information. The following function description detects when the client becomes connected to and disconnected from Zookeeper.

Data: client, newState

Result: void

Function name: void stateChanged(client, newState)

This function is called when there is a state changed in the connection.

4.4.2. Disconnection and Reconnection

When a node is reconnected to a quorum of Zookeepers, it will check to see how many other nodes are there. If there are none, it will perform the leader election described above and then assume a leadership role and can initialize itself using any cached data that it has. If, however, there are other nodes present at the moment that it reconnects (regardless of who the leader is/was), the node will flush its entire local cache, including all values and cache information.

This system would use the connectToKVStore method (in AbstractKVStore) to connect from one client to another (e.g., to the leader or a follower). The parameter that you pass to it is the same value that gets returned from getLocalConnectString().

```
protected IKVStore connectToKVStore(String connectionString) throws RemoteException, NotBoundException
    String[] d = connectionString.split(":");
    synchronized (connectedTo) {
        if (!connectedTo.containsKey(connectionString)) {
            Registry registry = LocateRegistry.getRegistry(d[0], Integer.valueOf(d[1]));
            IKVStore kvs = (IKVStore) registry.lookup(IKVStore.RMI_NAME);
            connectedTo.put(connectionString, kvs);
        }
        return connectedTo.get(connectionString);
    }
}
```

Figure 4.3: Connect to other KVStores using connectToKVStore.

4.4.3. Follower and Leader Recovery

Follower recovery contains local database recovery and remote recovery. Individually, the node would first contract with Zookeeper server to check the leader. If there is a leader in the Zookeeper session, then the ordinary follower will be a follower. But, the follower needs to flush its entire local cache.

On the other side, if there is not a leader on the Zookeeper session, then Zookeeper would check the number of the node, and begin to a new leader election. After having a new leader, the follower will also flush its entire local. For the leader recovery, when the leader goes down, the Zookeeper will elect a new leader. When the leader's ID gets updated, the write operation will complete.

4.5 Interface

The key-value store will expose just two simple operations to the end-user on the shell:

- getValue(String key)
- setValue(String key, String value)

To make this system more completed, this project also would provide other available commands on the shell:

- get: Get a key
- list: List clients
- new-client: Create a new client
- put: Set a key
- rmi-down: Disable a client's inbound RMI services

- rmi-up: Resume a client's inbound RMI services
- zk-down: Disable a client's access to ZooKeeper
- zk-up: Resume a client's access to ZooKeeper

Chapter 5 Performance Evaluation

5.1. Write Latency

In the distributed storage system, write latency is still a significant factor in evaluating the system and protocol performance. For this distributed storage system, the system key a low write latency between append write and overwrite method, and overwrite decrease about 20% write latency because this system has a Zookeeper server and sequentially consistent model. This test injects from 16 to 1024 records with the same size into the system. The size of writing requests ranges from 512 Bytes to 16989 Bytes totally, and all operations are synchronous. Based on Figure 5.1, we can conclude that the append writes and overwrite do not have a significant difference. Both of their write delay increase with the size of write requests, because this system uses a sequentially consistent model. With the size of write request becomes more significant, the average write delay increases faster because of the parameter of the key-value store. The results demonstrate that the write latency is not higher between append write and overwrite. (Key: 16 bytes, Values: 16 bytes)

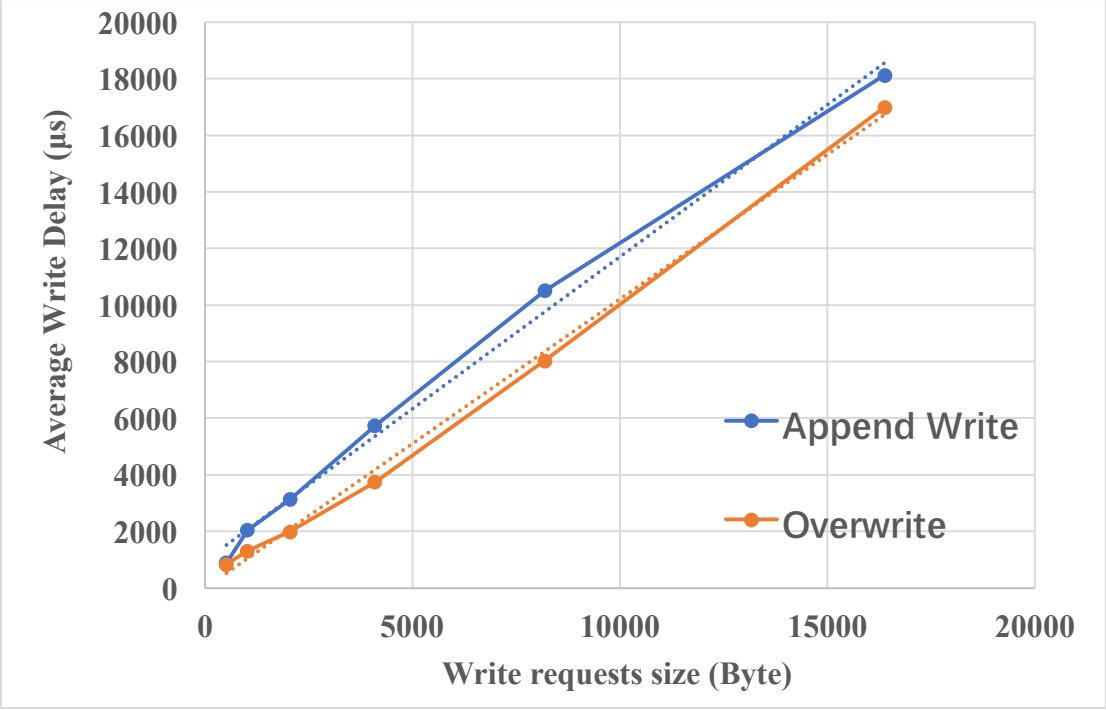

Figure 5.1 Write Latency

5.2. Comparison with Zookeeper

According to Figure 5.2, the test compares the write performance of with the local operations. To measure the latency of the local operation, the leader node does not send the data to other follower nodes. The system would use asynchronous write mode. There are three follower nodes to measure the write latency of ZiStore. Figure 5.2 shows that the overhead of the ZiStore protocol is small, which increase by about 20% depreciation over the local operation. For this situation, there are many reasons. For example, this system uses a rotating lock server which can reduce the overhead caused by locking; this system deals with a written request with write disk operation and network communication working in parallel. (Key: 16 bytes, Values: 16 bytes)

Figure 5.2 Compare ZiStore with local operation

5.3. System Scalability

As shown in Figure 5.3, this paper compares the write latency of ZiStore with four different numbers of nodes in the system to test system scalability. In this part, this system still uses a synchronous write mode. When the size of the write request is more

than 10000 bytes, four and three nodes is almost two times as slow as the two or one nodes. This is because the system would have more lock during the synchronous, but the highest write latency of four nodes is still not very high. Therefore, this system still can work effectively.

Furthermore, we even can use many different methods to optimise ZiStore. For example, this system can improve the overwrite log system, disk and network, and write queue. The results show that ZiStore has an average write performance with the increasing nodes in the system. (Key: 16 bytes, Values: 16 bytes)

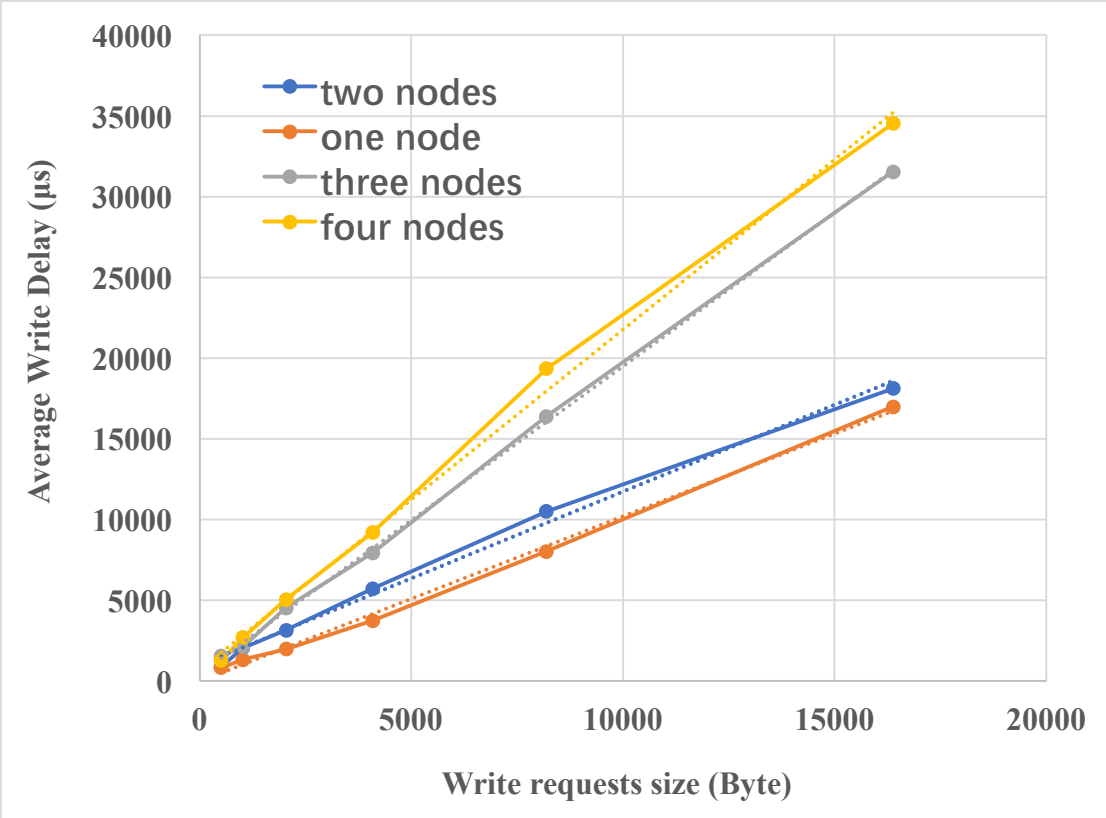

Figure 5.3 System Scalability

Table 5.4 shows each write delay when there are three follower nodes in this system. The size of writing requests ranges from 512 Bytes to 16989 Bytes totally, and all operations are synchronous. Based on the table, we can conclude that all write delay fluctuates seriously. For example, some write pause is almost two times as big as average write delay. This situation would demonstrate this system could have hidden scalability problem for sequentially consistency model.

Table 5.1 Example of the Three Nodes

Write requests size (Byte):	Write Delay (μs):					Average (μs):
512	984	1564	2510	846	1572	1534
1024	1643	2431	1987	3620	989	2134
2048	6437	3217	5214	3640	4137	4529
4096	9461	6437	5710	8167	9990	7953
8192	17843	14310	9640	26004	14178	16395
16384	15431	45970	25471	54317	16471	31532

5.4. system recovery

For the distributed storage system, node failure is frequent on the commodity machine. In the ZiStore, we set replica's number like four, if one or two follower nodes goes down, the system can run normally, but if three nodes failure, this system would stop service based on the fault tolerance requirement.

This system will find a new leader when the leader goes down. This process is speedy. Accurately, this system can complete the leader election less than 1.5s latency, which does not have a severe impact on the regular running of the operation. This is because once the leader goes down, Zookeeper would immediately perceive this situation and notify the other nodes, the system can elect a new leader by comparing the metadata of existing followers.

As shown in Figure 5.5, this system first puts 1000 records inside. Then, each new follower would join in this system and recovers these 1000 records. The size of the write request is between 512 to 16384 bytes. The experiments show that the recovery of 1000*512 bytes size only 7s and 1000*16384 bytes only needs 23s.

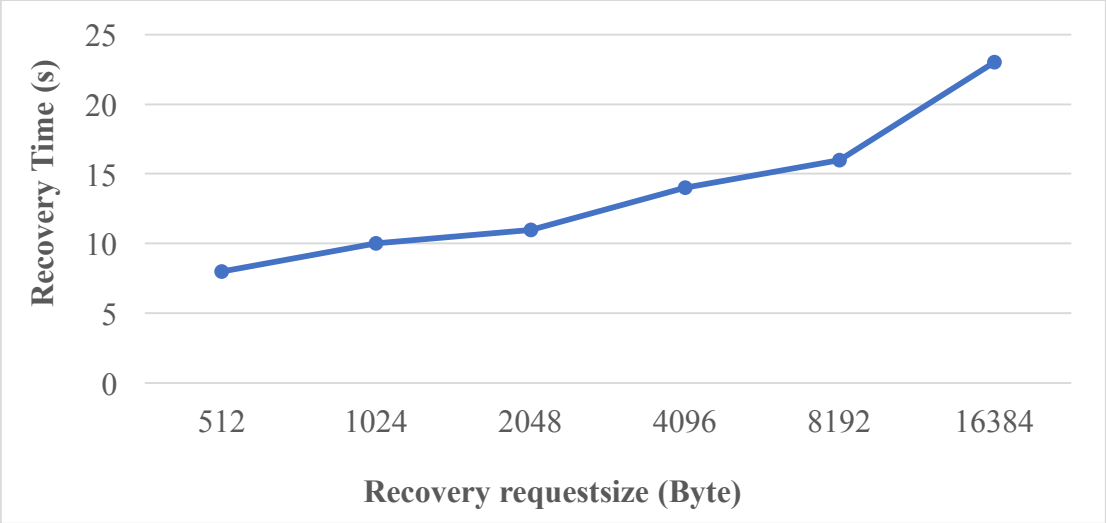

Figure 5.5 Follower Recovery Time

5.5. read latency

In this system, the read operation is not related to the sequential consistency model and the protocol, so this evaluation only needs to read data from local key-value store engine. Therefore, the read performance is almost the same as the key-value store (a simple hash map). As shown in Figure 5.6, the record size ranges from 512 to 16384 bytes. The results demonstrate that the read latency is only 160μs when the record size is 512 bytes, and the read latency increases as the record size increases. On the other hand, the local read performance is much bigger than the read latency of ZiStore.

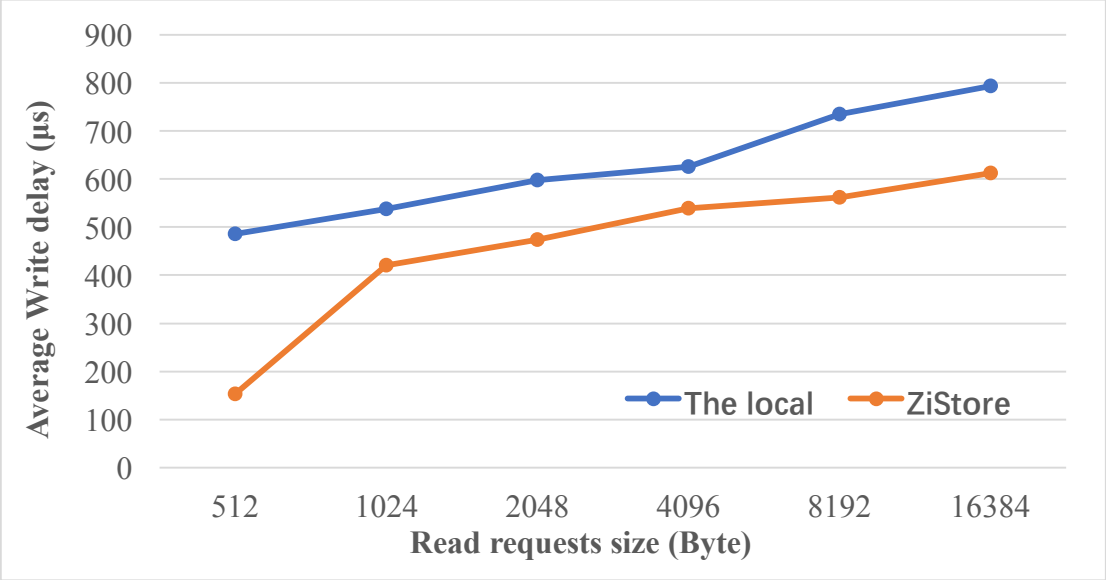

Figure 5.6 Read Latency

5.6. Engine evaluation

In this evaluation, this paper uses Yahoo! Cloud System Benchmark (YCSB)[17] to compare the performance of ZiStore and some popular key-value store: LevelDB and Kyoto Cabinet. The test cases are described as follows:

- Writing 2^10 key-value pairs with variable value size in one thread.
- Writing 2^10 key-value pairs with variable values size in four thread.

The tables 5.7 show the result of the benchmark below. The number in the table is the throughput, which means the number of operations per second. For the first part, the key-value store engines are set up with one writing thread with keys of 16 bytes and values of 16 bytes. But, ZiStore is not the highest number of operations per second and uses the most prolonged time writing the key-value pairs. The second part shows that the key-value store engines are set up with four writing thread with the key of 16 bytes and values of 16 bytes. The results show that ZiStore has the highest number of operations per second and take a shorter time writing the key-value pairs.

Table 5.2 Compared with the throughput of different storage system

Key: 16 bytes	Value: 16 bytes	
DBType	One Writing Thread: Throughput	Four Writing Threads: Throughput
ZiStore	57337	105422
LevelDB	86811	92440
KyotoCabinet	85837	60450

In conclusion, this table demonstrates that the performance of the key-value store engine in ZiStore is not very effective compared with the current popular key-value store. However, the throughput of ZiStore becomes better than other key-value stores with more writing threads, because ZiStore has Zookeeper server and sequentially consistency model, which can improve the performance of ZiStore.

Chapter 6 Conclusion and Future Work

6.1. Conclusion

In Chapter One, this paper mainly demonstrates the process of designing a distributed file system and Zookeeper, distributed transaction and commit protocol. Due to the purpose of this project, it is vital to understand the goals and challenges of designing a distributed file system. This paper discusses the features of Zookeeper such as Zookeeper advantage, Zookeeper guarantees and Zookeeper Session.

That three parts mainly shows the reason why choose Zookeeper as distributed coordination service in this project. Data model and Znode types demonstrate the structure of storing data and node inside Zookeeper. Zookeeper provides sequential consistency, which significantly matches the purpose of this project. What's more, Zookeeper can support failure handling for the client in Zookeeper session. This function can guarantee consistency for each Zookeeper client.

In Chapter two, this paper analyzes the related researches about Zookeeper and the meaning of Zookeeper in industrial production areas. It shows that Zookeeper has become a powerful distributed coordination service to implement the distributed system.

After having a basic understanding of Zookeeper, Chapter three and four begin to demonstrate the design of this storage system and implementation. First of all, this paper shows the architecture of this system, which has three important components. Then, this paper shows an important model in this system – sequentially consistent model. Chapter five is an essential part of this paper because it demonstrates the process of implementing the key-value server with no failover. By using the solutions from Chapter three to guarantee fault tolerance, this system can achieve writing and reading values' operation without bug for fault tolerance.

Then, this distributed file system also need to have consistency and partition awareness. Focusing on consistency and partition awareness, there are three different situations during reading values, and the conditions need to be analyzed step by step. After knowing each case for reading values, this paper begins to show reading values implementation. This part demonstrates the strategy and the pseudo-code of the

getValue function. It is the same for the writing values operation, but this part is more complicated due to the distributed transaction. Therefore, there are five different situations during writing values. Either reading or writing values, firstly needs to check the leader's status and client's status, which is on Zookeeper session or not.

In the conclusion, this paper designs and implements a consistency, fault-tolerance distributed key-value storage system, called ZiStore. In this system, we use sequentially consistent model and rotating lock server. ZiStore has a good performance and lower protocol overhead. The results demonstrate that this system is a consistency and fault-tolerance system. Through using Zookeeper, a high available service module, to do leader election, it can improve the system performance and simplify the design.

6.2. Future works

This section lists the threads of future work that are envisioned for this distributed file system.

In this distributed file system, the current query way is not very useful in the client's key-value database. Specifically, current key-value store requires the key to match all the value until finding the correct value for the getValue operation. This situation will cause massive energy consumption if the dataset is huge. Therefore, this system still can work on finding a better query way to reduce time consumption.

Besides, this system can use other protocol to improve the performance. For example, the Paxos-base protocol[18] as the tool to implement this key-value storage system. For the result, we can find the sequentially consistent model would have hidden problem with increasing the nodes. Therefore, this system could add a high available log system between the database and protocol. Finally, this system just use a basic key-value store database like a simple hash map. If this system can choose to use other key-value store database such as levelDB, the performance of this system could work better.

References

[1] Michael M, Moreira J E, Shiloach D. Scale-up x Scale-out: A Case Study using Nutch/Lucene[J]. IEEE International Parallel and Distributed Processing Symposium. IPDPS, 2007: 1–8.

[2] Brewer E A. Towards Robust Distributed Systems[C]. PODC, 2000: 7.

[3] Viotti P, Vukolić M. Consistency in Non-Transactional Distributed Storage Systems[C]. ACM Computing Surveys (CSUR), 2016, 49(1): 1–34.

[4] Raz Y, Vardi M Y, Gottlob G. The Dynamic Two Phase Commitment (D2PC) protocol[C]. ICDT, 1995, vol. 893: 162–176.

[5] Lin C, Nagarajan V, Gupta R, Rajaram B. Efficient sequential consistency via conflict ordering[C]. ACM SIGPLAN Notices, 2012, *47*(4): 273–286.

[6] Halalai R, Sutra P, Riviere E, Felber P. ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service[C]. Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium, 2014: 67–78.

[7] Hunt P, Konar M, Junqueira F, Reed B. Zookeeper: wait-free coordination for internet-scale systems[C]. USENIX technical conference, ATC, 2010.

[8] Ward J, and Barker A. Cloud cover: monitoring large-scale clouds with Varanus[J]. Journal of Cloud Computing. 2019, *4*(1): 1–28.

[9] Nguyen T, Nguyen M. Zing Database: high-performance key-value store for large-scale storage service[J]. Vietnam Journal of Computer Science, 2015, *2*(1): 13–23.

[10] Li S, Liu F, Shu J, Lu Y, Li T, Hu Y. A Flattened Metadata Service for Distributed File Systems[C]. Parallel and Distributed Systems. IEEE Transactions On, 2012, *29*(12): 2641–2657.

[11] DeWitt D J, Katz R H, Olken F, Shapiro L D, Stonebraker M R, Wood D. Implementation Techniques for Main Memory Database Systems[C]. SIGMOD, 1984: 1–8.

[12] Viotti P, Vukolić M. Consistency in Non-Transactional Distributed Storage Systems[C]. ACM Computing Surveys (CSUR), 2016, *49*(1): 1–34.

[13] Chandra T D, Griesemer R, Redstone J. Paxos Made Live: An Engineering Perspective[C]. PODC, 2007: 398–407.

[14] Franciscus N, Ren X, Stantic B. Precomputing architecture for flexible and efficient big data analytics[J]. Vietnam Journal of Computer Science, 2018, *5*(2): 133–142.

[15] Kaur A, Thind T. Optimized Fault Tolerance in Distributed Environment[J]. International Journal of Computer Applications, 2015, *97*(21): 30–35.

[16] Nguyen T, Nguyen M. Zing Database: high-performance key-value store for large-scale storage service[J]. Vietnam Journal of Computer Science, 2015, *2*(1): 13–23.

[17] Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB[C]. Proceedings of the 1st ACM Symposium on Cloud Computing. ACM, 2010: 143–154.

[18] Bolosky W J, Bradshaw D, Haagens R B. Paxos replicated state machines as the basis of a high-performance data store[C]. Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. NSDI, 2011: 11.

Acknowledgement

I would like to thank Prof. Liu Xin for his helpful comments and advice for improving this paper. This project's basic idea is the CS475 Spring 2018 – Concurrent and Distributed Systems' project from Prof. Jonathan Bell on George Mason University.

Secondly, I would also like to thank my parents and friends who helps me a lot in finalizing this project within the limited time frame.