Diskless Data Analytics on Distributed Coordination Systems

by

Dayal Dilli

Bachelor of Engineering, Anna University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s):	Kenneth B. Kent, Ph. D, Faculty of Computer Science
Examining Board:	Eric Aubanel, Ph. D, Faculty of Computer Science
	David Bremner, Ph. D, Faculty of Computer Science
	Sal Saleh, Ph. D, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

December, 2013

Abstract

A distributed system contains software programs, applications and data resources dispersed across independent computers connected through a communication network. Distributed coordination systems are file-system like distributed meta-data stores that ensure consistency between processes of the distributed system. The challenge in this area is to perform processing fast enough on data that is continuously changing. The focus of this research is to reduce the disk bound time of a chosen distributed coordination system called Apache Zookeeper. By reducing the disk dependency, the performance will be improved. The shortcoming of this approach is that data is volatile on failures. The durability of the data is provided by replicating the data and restoring it from other nodes in the distributed ensemble. On average, a 30 times write performance improvement has been achieved with this approach.

Dedication

I dedicate this thesis to my parents, brother and sisters for their endless love, support and encouragement.

Acknowledgements

First and foremost I offer my sincere gratitude to my supervisor, Dr. Kenneth B. Kent, who has supported me throughout the thesis with his patience and immense knowledge.

Secondly, I would like to thank UserEvents Inc. for providing the opportunity and facilities needed to work on this project. I greatly appreciate the support and encouragement of everyone at UserEvents Inc. In particular, I extend my deepest thanks to Robin Bate Boerop, my supervisor at UserEvents for his engagement, remarks and comments throughout the research. Also, It was a pleasure working for this research along with great minded people like Jeff Thompson (CEO, UserEvents), Trevor Bernard (CTO, UserEvents), Suhaim Abdussamad, Josh Comer and Ian Bishop at UserEvents.

I also gratefully acknowledge the financial support from Natural Sciences and Engineering Research Council of Canada for this research.

Finally, I would like to thank all my friends back home far away in India and my friends in Fredericton for their continuous support and encouragement. In particular, I appreciate my home mates Jeevan, Sai and Sharath for their patience and support during the busy times of my research.

Dayal Dilli

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ZAB	Zookeeper Atomic Broadcast
WWW	World Wide Web
IMDB	In-Memory Database
ACID	Atomicity, Consistency, Isolation, Durability
DFA	Deterministic Finite Automaton
BFT	Byzantine Fault Tolerant
CAP	Consistency, Atomicity, Partition Tolerance
FIFO	First In First Out
TCP	Transmission Control Protocol
DFS	Depth First Traversal
DAG	Directed Acyclic Graph
API	Application Programming Interface
SSD	Solid State Drive
IDE	Integrated Development Environment
NIO	Network Input Output

Chapter 1

Introduction

1.1 Motivation

A distributed system contains software programs, applications and data resources dispersed across independent computers connected through a communication network. These systems allow distribution of task execution as well as distribution of task storage. Real world applications of distributed systems include distributed databases, Wireless Sensor Networks and Web based applications.

The architecture and structure of distributed computing has evolved over the past decades. Initially, the system started with concurrency and mutual exclusion among processes of a system and it has evolved to perform concurrent task execution across machines in different parts of the globe. The

challenge in this area is to perform processing and analysis fast enough with the preservation of the certain properties ensured by the distributed systems. The important need of a distributed system is to operate fast enough while preserving its performance.

Distributed coordination systems act as the primary controller of all events in a distributed system. The consistency, availability and durability needs along with the processing throughput acts as the major performance attribute in these systems. Typically, they have a filesystem like structure backed up by operations that provide co-ordination. Although the file-system of distribution co-ordination systems operate as an in-memory database [4], in order to achieve durability it stores data in some form on the disk through *Transaction logs* and *Snapshots*. This dependency on disk I/O acts as a bottleneck in the performance of these systems. The goal of this research is to reduce the disk dependency of the in-memory databases in the distributed co-ordination systems and provide durability through data clustering.

1.2 Research Objectives

The focus of this research is to review the performance of various distributed co-ordination systems and analyze them to arrive at a diskless solution that can offer the best case performance. Apache Zookeeper [11] (a subproject under Apache Hadoop [26]) is a file system like hierarchical namespace pro-

viding highly available centralized service, distributed synchronization, group services, message queue service and configuration management. Zookeeper utilizes Zookeeper Atomic Broadcast protocol (ZAB) [27] to provide distributed co-ordination. The major objective of this research is to study the working of Apache Zookeeper and implement a low disk dependent persistence technique in the Zookeeper ensemble.

1.3 Thesis Organization

The thesis is organized into six chapters. In Chapter 2, the background information required to understand the remaining parts of the thesis is presented. This includes various concepts and properties of distributed systems. A brief introduction to the core product under research namely (Apache Zookeeper) is also presented. In Chapter 3, the design ideas of the project are discussed; it begins with the requirements of the project followed by the system setup and design decisions of the research. In Chapter 4, the implementation details of the project are explained; it starts with the review of the programming structure of Apache Zookeeper followed by the modifications done on Zookeeper to implement the research ideas. In Chapter 5, the evaluation of the system is presented; it includes benchmarking the new system and comparing the results with the earlier system. Finally, Chapter 6 concludes with the outlook of the possible future extensions of this research.

Chapter 2

Literature Review

This chapter presents a review of background knowledge associated with the design of distributed systems. It will start with the introduction to distributed computing in Section 2.1 and distributed in-memory databases in Section 2.2. This is followed by a brief survey of distributed coordination systems in Section 2.3. The various design, architectural and performance decisions of a distributed coordination system are presented in the Subsections of 2.3.

As most of the research findings are implemented in Apache Zookeeper, a detailed explanation of Apache Zookeeper is presented in Section 2.4. The Zookeeper Atomic Broadcast (ZAB) protocol operating at the core of Zookeeper is described in Section 2.4.1. The major optimization of this project is implemented in the persistence layer of Zookeeper which is intro-

duced in Section 2.4.2. Finally, a survey of various other related research works are presented in Section 2.5.

2.1 Distributed Computing

Distributed Computing [1] is a branch of computer science that deals with distributed systems. It consists of computing on software and hardware components located at networked machines connected through communication protocols. Each computational machine in a distributed system has its own memory and the communication between them is accomplished through message passing. Examples of this type of system includes the internet, World Wide Web (WWW), search engines, email, social networks, e-Commerce, Online gaming, etc. The high level architecture of distributed systems is presented in Figure 2.1.

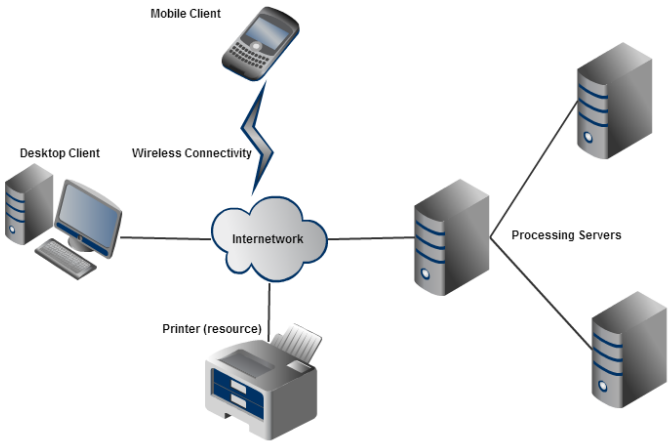

Figure 2.1: Architecture of Distributed Systems

The prime goal of distributed systems is to share resources between multiple computers. The term resources extends from hardware entities like printers and disk to software components like files, databases and data objects. The sharing of resources between a large number of computers requires very high processing throughput. In order to meet the needs of processing speed, parallel execution is normal in distributed systems. In addition to parallel processing, consistency constraint synchronizes the data between multiple nodes and ensures the same data is seen by all the nodes. Therefore, the coordination among parallel executing programs that shares resources is highly important.

Apart from the processing speed and consistency requirements, distributed systems are expected to be highly available. High availability ensures the system to be continuously serving under all operational conditions. This requires failure independence between the distributed machines and proper handling of failures. Deutch et al. explains the various pitfalls to be considered in designing an efficient distributed system [2]. In addition to high availability, the distributed cluster should continue to operate in case of partition that may result in a communication loss between nodes. This is called as *partition tolerance*. The conjunction of the above discussed goals namely consistency, availability and partition tolerance is discussed as CAP theorem in Section 2.3.3. The design of a high performance, concurrent, consistent,

and fault tolerant system will be a recurring theme throughout this thesis.

2.2 Distributed In-memory Databases

An *in-memory database* (IMDB) [4] is a database management system that primarily uses main-memory to store the data. The major motivation behind in-memory databases is to optimize the processing throughput of data access. For a sequential access database system, The disk is approximately 6 times slower than the main memory [3]. In order to keep up with the speed of the processor, main memory is used to store the database. This improves the performance the databases vastly. The main memory databases are designed based on the size of the data to be stored in the database. For larger datasets, typically main memory is used as a cache to actual data residing on the disk. For smaller datasets, the data is completely stored in memory backed up by a transaction logging or snapshot mechanism to provide durability. Thus, main memory databases serves as a high speed alternative for systems that need high processing throughput.

Although an in-memory database basically uses main-memory to store the data, it has a considerable disk dependency. In order to provide durability to the database, data in some form is backed up on the disk. One form of this backup includes transaction logging of the requests to the database. This technique helps to keep track of the data during failures. The other form of

backup involves storing the entire data on the disk at specific checkpoints. In these databases, main-memory is primarily used as a cache for the disk resident data.

The category of in-memory databases that are replicated across multiple sites operating consistently are called distributed in-memory databases. Replication and coordination of the replicas satisfy the high availability and fault tolerance needs of the distributed system. These databases are used as temporary message passing queues, key-value stores [5] or metadata stores rather than logical high-volume application data stores. So, consistency [6] and availability are given more importance than space efficiency. The major examples include Cassandra [8], Riak [9], HBase [10], Apache Zookeeper [11], Accord [12].

2.3 Distributed Coordination Systems

A *Distributed coordination system* [13] is a highly networked software system that acts as a primary controller of events in a distributed system. It ensures the coordination of activities in a distributed system by providing consistency among various entities in it. At the core of the distributed coordination system is an in-memory database that stores the control and metadata needed for coordination. A set of synchronization and ordering protocols running around this in-memory database provides the required coordination

functions. High performance is one of the major needs of distributed coordination systems as it controls every operation in the distributed system. As a large volume of this research concentrates on improving the performance of an in-memory database in a distributed coordination system, a very brief review of the design and architecture of the in-memory databases specific to distributed coordination systems is presented in the following subsections.

2.3.1 Distributed Replication

Replication is a technique of sharing data consistently between redundant resources. By maintaining consistent redundant data, it also provides fault tolerance and high availability to the system. Replication in these systems can apply to both data and computation. There are two commonly used replication strategies: active replication and passive replication. Active replication involves processing the same request at every replica and the first outcome is served as the response of the ensemble. Passive replication uses a primary machine called a leader to process the request and the results are transferred to other machines in the replica. Distributed coordination algorithms provide consensus, synchronization and fault tolerance between the replicas. The popular system models used in distributed replication are presented below.

- **Transaction Replication [15]:** This method employs the one-copy serializability [16] model where every request is treated as a transaction. The transactions are logged on a persistent store with full Atomicity,

Consistency, Isolation and Durability (ACID) [17] compliance. It involves write ahead logging [17] which records the transactions before executing the outcomes. The main goal of this model is to provide complete fault tolerance to the system in addition to consistency. Apache Zookeeper (explained in Section 2.4) is an example of a distributed coordination system using this model.

- **Virtual Synchrony:** Virtual Synchrony [21] employs a process group paradigm where a group of processes coordinate to replicate the distributed data. Processes communicate with each other through message passing. In addition to the process communication in the group, application processes communicate with the process groups. Virtual synchrony provides two levels of consistency: *strong virtual synchrony* and *weak virtual synchrony*. Strong virtual synchrony applies for applications that communicates synchronously with the process group. It provides strong ordering and consistency to the system. Weak virtual synchrony applies for applications that communicate asynchronously and there is no bound on the response time. Weak virtual synchrony provides high performance distributed coordination with weaker ordering and synchronization guarantees. Depending upon the application needs one of the ordering protocols namely FBCAST, ABCAST, CBCAST, GBCAST is implemented in the core of virtual synchrony [21]. Depending on the strictness of the ordering protocol the performance varies. The lower the ordering requirement the higher is the perfor-

mance. In this mode, failures can cause the clients to lose the response messages and the consistency remains weak. *Accord* (described in Section 2.5.1) is an example of a distributed coordination system using this model.

- **State Machine Replication [22]:** This model depicts the distributed replica as deterministic finite automaton(DFA) [18] and assumes that atomic broadcast of every event is reliable. A state machine has a set of inputs, outputs and states. A *transition function* takes a particular input and current state to define the next state. Multiple nodes in the replica maintains the same state to provide consistency. Paxos algorithm [33] uses this model for replication. Google's *Chubby* Key-Value store discussed in Subsection 2.6.1 uses Paxos algorithm in its core [32].

2.3.2 Fault Tolerant Models

Fault Tolerance is a system design model that assures the system will continue operation even when some part of the system fails. It involves categorizing the nature of failures that can possibly affect a system and the implementation of techniques to prevent and handle those failures. The major failure models in a distributed system are discussed below.

- **Crash Fail** [23]: It involves failure of the process or the machine running the process which causes the entire system to halt. A process can

fail due to the generation of incorrect results; e.g., protection fault, divide by zero and deadlocks. The system failure is an effect of hardware failures and infrastructure failures which includes power outages, network failures, etc.

- **Omission Failure [24]:** Omission Failure is a result of loss of messages in the communication. A process can complete sending the message but the recipient may fail to receive it and vice versa.

- **Commission Failure [19]:** Commission Failure is a result of incorrect processing of the request. This leads to inconsistent responses to the request that further results to corrupted local state of the system.

- **Byzantine Failures [25]:** This failure results as a combination of multiple arbitrary failures resulting in the malfunctioning of the system. The failure can be either due to crash fail, omission failure, commission failure or a combination of any number of them. The Byzantine Fault Tolerant (BFT) [20] algorithm defines a set of logical abstractions that represents the execution path of an algorithm. A faulty process is one that exhibits failure in one of the execution paths.

2.3.3 CAP Theorem

From the above review of distributed systems, it is seen that consistency, high availability and partition tolerance are necessary for efficient operation of the system. The challenge is that it is difficult to design a system that

ensures all three properties. The CAP theorem was proposed a conjecture and later proved by Eric Brewer [7]. The theorem states that

> It is impossible in a distributed system to simultaneously provide all three of the following guarantees: Consistency, Availability and Partition Tolerance(CAP) [6].

The CAP theorem verifies the limitation in the guarantees that can be provided by a high performance distributed coordination system.

2.3.4 Performance and Scalability

The CAP theorem explained the guarantees that can be assured by a logically correct distributed coordination system. In addition to it, a distributed coordination system should satisfy the following properties. They are:

- **High Performance:** The performance of the system is determined from two point of views. One is the number of requests that a system can process in a specified time period. This is also called as the throughput. The higher the throughput the better the performance. On the other hand, performance is defined from the resource requirements of a system. The lower the resources required for the operation of a system, the better is the system design.

- **Scalability:** Scalability [**?**] is the ability of a system to operate without degradation in performance with the increase in workload. The

growth rate of the users in a distributed system is increasing exponentially every year. In order to efficiently serve without degradation in performance for high workloads, the system should be scalable.

2.4 Apache Zookeeper

Apache Zookeeper [11], a replicated file-system like hierarchical namespace, is a highly available, distributed service to provide system coordination. It is the current de facto standard coordination kernel for many distributed systems. Yahoo, Facebook, Twitter, Netflix are some of the giant corporate users of Apache Zookeeper [11, 54, 55, 56]. It was first developed as a configuration manager in Apache Hadoop [26] and later refined to adapt to the requirements of various other synchronization needs. The major use cases of Zookeeper include distributed synchronization, group services, message queue service, naming and configuration management.

Zookeeper implements an in-memory hierarchical space of data nodes called znodes [27]. The clients use the znodes to implement their required coordination function. Znodes are fundamentally like files with operations like create, read, write, update, delete, etc. Zookeeper utilizes the Zookeeper Atomic Broadcast protocol (ZAB) [27] to provide distributed consensus.

2.4.1 ZAB Protocol

Zookeeper Atomic Broadcast protocol (ZAB) is a totally ordered broadcast protocol that maintains the consistency of replicas in Zookeeper. The architecture of ZAB consists of an ensemble of nodes, one of which acts as the leader. The leader acts as the replication point that coordinates and orders the requests. Zookeeper ensures the consistency and availability properties of the CAP theorem. In addition to this, it can support high performance and scalability.

The design model of Zookeeper is a *Transaction Replicated* model. Each request on Zookeeper is reframed as a transaction before being processed by ZAB. The results of the ZAB are then reflected on the znodes in the replicated in-memory database. The high level architecture of ZAB is presented in Figure 2.2.

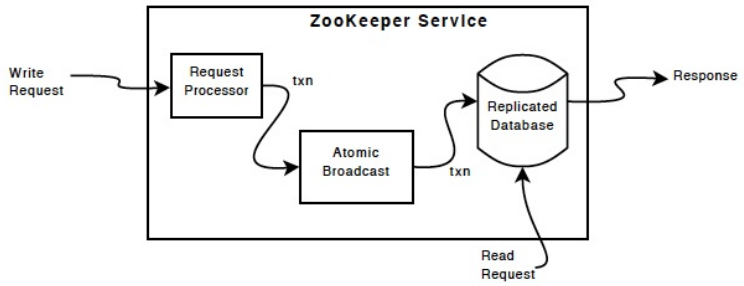

Figure 2.2: The logical components of Apache Zookeeper [27]

2.4.1.1 ZAB Guarantees

As a coordination kernel, Zookeeper requires certain guarantees to be strictly implemented by the ZAB. They are reliable delivery, total order and causal order. It also obeys the prefix property [27] to ensure correctness. The prefix property states that

> If m is the last message delivered for a leader L, any message proposed before m by L must also be delivered.

Precisely, Zookeeper guarantees eventual sequential consistency [28] with reliable delivery. The reliable delivery is ensured through a crash fail state recovery model discussed in Subsection 2.4.2.

2.4.1.2 Leader Election Algorithm

Zookeeeper follows a master slave model. The master, called the leader, serves as the control point for ordering the events. The leader election algorithm defines the sequence of steps to elect a leader for the ensemble. Zookeeper Atomic Broadcast protocol consists of two modes of operation:

- **Broadcast Mode:** Zookeeper is said to be in broadcast mode when the ensemble has a leader with a quorum of followers. The leader executes the broadcast protocol and the followers follows the results from the leader. Practically, TCP socket channels are used to guarantee FIFO delivery. When the leader receives a request, it broadcasts the proposal to the followers. Each proposal is denoted by a transaction

identifier called as a *zxid* [27]. The proposals are in the order of zxid. This ensures the ordering guarantee of Zookeeper. Each follower on receiving the proposal records a transaction log of the request on disk and sends an ACK back to the leader. The leader after it receives ACK from a quorum of servers, COMMITs the message locally as well as broadcasts it to the quorum. The followers deliver the changes to its in-memory database once it receives the COMMIT from the leader.

- **Recovery Mode:** The Broadcast Mode of ZAB works well when there is a leader with a quorum of followers. According to the ZAB assumptions, an ensemble should have f servers out of $2f+1$ servers to be operational for the quorum to function. The broadcast mode fails when more than f servers fail or the leader fails. To guarantee availability, a recovery procedure is needed to bring the ensemble back to functional operation. The recovery mode involves the correct execution of a leader election algorithm to resume the broadcasting. The leader election algorithm involves each server nominating itself with its zxid. The server with the maximum number of transactions denoted by its zxid is elected as the leader. The zxid is a 64 bit number. The lower order 32 bits are used as a counter for the transactions. The higher order 32 bits are incremented whenever a new leader is elected. The lower counter is used to determine the leader in the election algorithm. The Figure 2.3 shows the format of the zxid. Once the leader is elected, the quorum of followers synchronize with the leader to resume the broadcast mode.

Figure 2.3: Format of 64 bit zxid in Zookeeper.

2.4.2 Persistence Layer in Zookeeper

As explained in the above section, Zookeeper is a transaction replicated model following crash fail state recovery during failures. To recover the state during failures, Zookeeper uses its transaction logs and snapshots taken periodically.

2.4.2.1 Fuzzy Snapshots and Transaction Logging

The snapshot and logging in Zookeeper follows the *Fuzzy Snapshot* technique for database restoration by B. Reed et al [29]. This provides persistence to the Zookeeper data and it runs as a part of ZAB.

Any node in the cluster, receiving a proposal request, records it to the disk as a transaction log entry. When the number of transactions in the transaction logs exceeds a threshold configured using the parameter *snapCount* [27], a snapshot of the state of zookeeper is written on the disk. This snapshot, called a fuzzy snapshot, is a depth first traversal (DFS) [30] over the in-memory directed acyclic graph (DAG) [31] of Zookeeper. Figure 2.4 shows

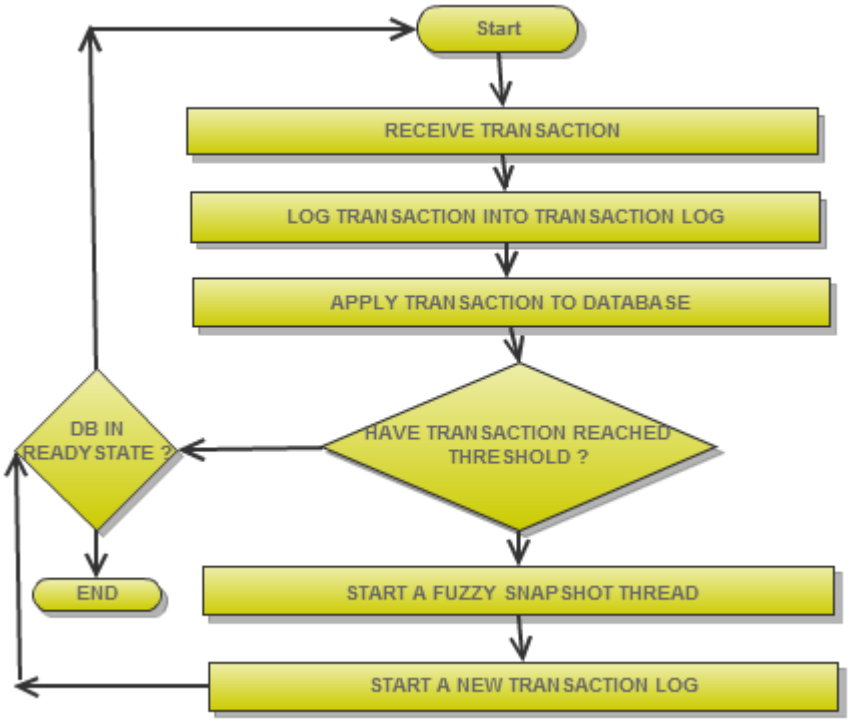

Figure 2.4: Flow diagram of fuzzy snapshot technique for Database restoration [29].

the steps in the Fuzzy snapshot and transaction logging process. The snapshot is fuzzy because there is no lock on the in-memory state when the snapshot is taken. So, the state of zookeeper at the beginning of the snapshot and the end of the snapshot may be different depending upon the requests served in the interim. As a result, the snapshot has only a partial dirty state of Zookeeper. But the request during the snapshot period is logged as transactions on the disk. So, the snapshot along with the transaction log gives the entire state of Zookeeper. During the recovery mode of Zookeeper, the

snapshot and the transaction log are used to restore the state of Zookeeper.

2.5 Why Diskless Zookeeper?

Although Zookeeper is proclaimed as an in-memory database, it heavily depends on disk to provide persistence. Essentially, whenever there is a write operation (create, set, delete) on Zookeeper, a transaction log of the operation is written on the disk. As discussed above in Section 2.4 the transaction logging mechanism blocks the processing of requests in Zookeeper. So, each request on Zookeeper is blocked for the transaction log to be written to disk before committing. This introduces a disk bottleneck to the write operations in Zookeeper. On the other hand, snapshots are fuzzy and are concurrent to the operation of Zookeeper. Since Zookeeper is a replicated in-memory database, there is a possibility to restore the data from other operational nodes during failure recovery. Hence diskless failure recovery from the cluster forms the essence of this research. It reduces the disk dependency of Zookeeper's in-memory database as well as provides failure recovery using the state of other nodes in the replicated ensemble.

2.6 Related Work

There are a few other systems developed to achieve the same goals as Zookeeper. Specially, the design aspects of Google Chubby and Accord closely resembles

the goals of our research.

2.6.1 Google Chubby

Google Chubby[32] is a proprietary distributed coordination system developed by Google Inc. It is used to provide coordination to the Google's Big Table Distributed Database. The Paxos [33] distributed coordination algorithm is the core of Chubby. It uses Berkeley DB [35] for backup. The transaction logging and snapshot mechanism in Chubby follows the conventional database logging mechanism by Birman et al. Although availability and reliability are the primary guarantees of Chubby, it heavy relies on in-memory caching for higher performance [32]. Also, The authors discuss lack of aggressive caching as one of the bottlenecks in the performance of Chubby [32]. So, a need for a less disk bound design is evident.

2.6.2 Accord

Accord [12] is another coordination kernel developed by NTT Laboratories. The goal of the project is to develop a coordination kernel with better write performance. It uses Corosync cluster engine [37] to provide distributed coordination and Berkeley DB for backup. Corosync in-turn uses virtual synchrony model for distributed consensus. The asynchronous in-memory configuration of Accord provides 18 times write speed up compared to Apache Zookeeper [12]. The speed up is mainly because of the virtual synchrony

model and the in-memory configuration of Berkeley DB [12].

Although there is a good improvement in performance, Accord has some fundamental design flaws. The in-memory mode uses Berkeley DB and is designed to operate purely in-memory [36]. This technique results in very poor durability during failures. The second disadvantage is that virtual synchrony is a weaker replication model for distributed coordination compared to state machine replication and the one-copy serialization models, because ABCAST protocol [21] in asynchronous virtual synchrony weakens total ordering for high performance as described in Subsection 2.3.1 and [21].

Chapter 3

System Design

This chapter describes the design strategies of the project. It begins with the requirements of the project in Section 3.1. The requirements are based on the problems analyzed in Chapter 2. Followed by the requirements, the approach to solve the problem is discussed in Section 3.2. The hardware and software frameworks required for the project are discussed in Section 3.3. Based on our problems and requirements, the design of the solutions are presented in Section 3.4.

3.1 Requirements

The requirements for this project are formulated depending upon the needs of a distributed system, the problem and also from the perspective of the user. The requirements are as follows:

- The implementation changes should modify only the persistence layer of Zookeeper. It should be completely transparent to the remaining modules of Zookeeper.
- The changes should not weaken the consistency guarantees of Zookeeper.
- The system should be thread safe.
- The changes should be transparent to the software developers using the Zookeeper API.
- The configuration changes should be kept minimal.
- The resource requirements of Zookeeper should not increase vastly.
- The system should be scalable with high performance.

The research ideas implemented as an optimization to the open source project Zookeeper, should be highly transparent to the users of Zookeeper. Zookeeper being a high networked, multi-threaded system should be carefully modified to change only the persistence layer. After each modification the system should be regression tested to make sure the internal workings of the system remains correct. Although, our modification may change the level of durability of Zookeeper it should not affect the consistency of Zookeeper. The ordering guarantee ensured by Zookeeper should be strictly followed by the changes.

The implementation of most of our research ideas are multi-threaded in nature. So, the implementation should be checked for thread safety whenever a critical region is accessed.

The other important requirement is that the changes should be completely transparent to the developer. It should not include any additions or modifications to Zookeeper at the API level. Any previous program written for the Zookeeper API should be able to run with our changes implemented. Also, the changes in the installation and configuration procedure of Zookeeper should be kept minimal. Given a new JAR executable of Zookeeper, the changes to the configuration parameters must be minimal compared to the existing Zookeeper installation.

In addition to these requirements, the resources needed for the operation of Zookeeper should not change. Also, the implementation of the research ideas should bring an overall improvement in performance. This improvement in performance should be as scalable as the default version of Zookeeper. Also, evaluation of all these requirements should be properly carried out and analyzed.

3.2 Approach

The tasks to achieve the requirements in Section 3.1 are divided into several subtasks. The first task involves setting up the Zookeeper code base from the open source repository and understanding the flow of control in the programming sequence. This is followed by the purging of the Transaction Logging mechanism from Zookeeper. Once the Transaction Logging mechanism is removed, regression tests should be carried out to check the correctness of the system. After that, performance tests should be run to evaluate the performance of the current system and the default Zookeeper.

The purging of the transaction logging mechanism reduces the level of persistence in Zookeeper. Hence, the failed nodes rely on the other operating nodes in the cluster to recover the data during failures. This is dealt with by the diskless clustering design introduced in Subsection 3.4.1. The second task involves analyzing the major failures that Zookeeper is prone to. This involves inducing failure in the new system and analyzing the guarantees provided by Zookeeper. To provide complete durability with low disk dependency, a model called *Durability at Failure* is designed. This is described in Subsection 3.4.2. Also, regression and failure analysis tests are carried out in every increment of our implementation.

The final task is to evaluate the performance of the modified Zookeeper.

There are two major tests involved in this task. The first test involves comparing the performance of various operations in Zookeeper between the default and modified versions. The second test analyses the time taken for a Zookeeper node to resume operation during failures. The results of this test is also compared with the default Zookeeper. The other tests include testing the resource usage of Zookeeper and the performance of use-cases for which this modification is implemented on Zookeeper.

3.3 System Setup

This section describes the system setup needed to achieve the requirements of the project. Subsection 3.3.1 explains the hardware setup and Subsection 3.3.2 explains the softwares that are used for the project.

3.3.1 Hardware Setup

The hardware and working space needed for this project are provided by UserEvents Inc. The machine has a Intel Core i7-3520M quad core processor. The machine includes a memory of 32 GB. The persistent storage in this machine is a 256 GB Solid State Drive (SSD). This machine is used in all the stages of development and evaluation. The detailed specification of the machine is presented in Appendix A.1.

3.3.2 Software Setup

The operating system used for this project is Ubuntu 12.07 64-bit desktop version. The software setup involves forking a copy of Zookeeper code from Apache software foundation's SVN repository and setting it up for development. Zookeeper is completely written in the Java programming language. To automate and ease the various processes in development and deployment the Eclipse Integrated Development Environment (IDE) is used. Eclipse IDE supports Java and also has plugins to support various other languages also. Open JDK 1.6 64-bit is the Java development kit used. In addition to this Ruby and C languages are used for some parts of testing and evaluation.

3.4 Project Design

This section describes the major design decisions pertaining to the implementation of the research ideas. It mainly focuses on the design of the diskless clustering model that reduces disk dependency with sufficient levels of durability.

3.4.1 Diskless Clustering Design

The diskless clustering model introduces the idea of persistence from a networked cluster rather than a local persistent store. Here, cluster means a set of computational nodes operating together following a consensus algorithm. This model specifically applies to a distributed system that is transaction

replicated in a leader-follower fashion and following a crash fail state recovery model. According to the diskless clustering model, whenever there is a request that updates the state of Zookeeper, a fuzzy snapshot of the in-memory state is written on the disk. As explained in Subsection 2.4.3 the fuzzy snapshot is a partial snapshot that is concurrent to the processing of requests. So, the performance is not affected by storing the fuzzy snapshot to disk. This fuzzy snapshot is converted into a clean snapshot by the operational nodes during certain checkpoints. The checkpoints are determined by detecting the failure of nodes in the Zookeeper replica that can cause either the loss of leader or quorum. So, before resuming the entire replica in recovery mode, a complete snapshot of the in-memory state is written on the disk by the non-failed nodes. One of these non-failed nodes that has the complete state persisted, is elected as the leader according to the leader election algorithm. So, the failed nodes can recover their data from the current leader of the ensemble when they resume. Also, the failed nodes will contain a partial state in their local persistent store. So, only a difference of the state is to be sent from the cluster leader. This also optimizes the restoration time of the failed node.

3.4.2 Durability at Failure Design

The above section explained at a high level the diskless clustering model. The core concept of the diskless clustering model is to restore the content of the failed nodes from the operating nodes in the replica during failures.

The level of persistence is called *Durability at Failure*. The postulates of this design are as follows:

- The first scenario is the failure of the follower node without affecting the quorum stability. In this case, the follower node loses its in-memory state temporarily. When it resumes, it is designed to restore the data from its fuzzy snapshot as well as from the Leader node that has the most recent data.

- The second scenario is the failure of the Leader node. When a Leader node fails, the nodes loses its data in the memory temporarily and unacknowledged requests in the flight permanently. The remaining nodes take a backup of their data as a snapshot on disk during the leader failure. This snapshot is a complete snapshot of the entire in-memory state. The acknowledged, uncommitted requests are written to the disk as a serialized object. Following this, the leader election algorithm takes place that elects the node with the most recent state as the Leader. When the failed Leader resumes, it restores the state from the current Leader. Also, when the new leader resumes it processes the acknowledged, uncommitted requests that are backed up during the failure. The uncommitted requests will eventually get committed by the new leader.

- The last scenario is the failure of follower nodes leading to the loss of quorum. In this case before the Leader stops serving requests, the entire

state of the Leader is backed up as a snapshot. So, when the quorum resumes the failed followers can restore the data from the Leader that has the current state.

Chapter 4

Implementation

This chapter explains the implementation of the system design discussed in Chapter 3. It starts with the study of Apache Zookeeper's code structure in Section 4.1. This is followed by the evaluation of internal data structures and programming packages used in Apache Zookeeper. After that, a detailed explanation of the implementation is presented in Section 4.2. Subsection 4.2.1 explains the Busy snapshot algorithm and Subsection 4.2.2 explains the modifications in the failure recovery mechanism of the ZAB protocol. Finally, the integration of our implementation with Zookeeper is presented in Subsection 4.2.3.

4.1 Program Structure

This section describes the core flow of control in the programming implementation of Zookeeper. Zookeeper is programmed entirely in the Java language. It uses some of the high performance network and concurrency packages in Java. The major components of the ZAB protocol involves transforming the requests into transactions, logging the transactions using a fuzzy snapshot mechanism and writing the changes to the in-memory database of Zookeeper. Along with this, the components of the ensemble includes a leader, quorum of followers and recovering nodes called *Learners*. These are the core components of Zookeeper that are relevant to our research. The implementation details of the leader election, ordering properties are not discussed as our modifications do not regress their operation and this research doesn't modify any of them. Each of the components discussed above are implemented as a thread running concurrently. The flow of control in the processing of a request in Zookeeper is shown in Figure 4.1. The details of the threaded components are as follows:

- **PrepRequestProcessor [43]:** The function of the *PrepRequestProcessor* thread is to transform the client request in the form of a transaction. It receives the request from the client and encapsulates the headers needed for the transaction. The headers include the list of transaction ids for creation and modification of node, time, operation to be performed and any data sent by the client. This transaction is

then queued in the *SyncRequestProcessor* [44]. The *PrepRequestProcessor* thread runs primarily on the leader server. Alternatively, on the follower servers, *FollowerRequestProcessor* [45] receives the request and processes it depending upon the type of the request. If it is a read request it is served locally and if it is a write request it is sent to the leader. The *PrepRequestProcessor* running on the leader then processes the request.

- **SyncRequestProcessor:** The *SyncRequestProcessor* thread is the implementation of the transaction logging and the fuzzy snapshot technique discussed in Subsection 2.4.2. The *SyncRequestProcessor* receives the transactions from the *PrepRequestProcessor*, if the request is a read request it is queued directly in the *FinalRequestProcessor* [45]. If the request is a write request, the *SyncRequestProcessor* logs the transaction to the disk, waits until the transaction is synced to the disk and then queues the request in the *FinalRequestProcessor*. Also, when the number of transactions reaches the *snapCount* threshold, a fuzzy snapshot thread is started.

- **FinalRequestProcessor:** The *FinalRequestProcessor* is the end of the processing sequence. It takes the queued transactions from the *SyncRequestProcessor* in FIFO order and applies on the in-memory database of Zookeeper. It queries the in-memory database of Zookeeper to apply the requested changes.

- **Leader [47]:** The Leader class defines the functions of the leader defined by ZAB. The major task of the leader is to maintain the consistency of the quorum. The main function of the leader is to keep the followers synchronized. Whenever a new follower emerges, the Leader should analyze the state of followers and send the updates to the new follower. This will keep the emerging or recovering follower consistent with the ensemble. The other functions of the leader includes sending a proposal to the followers, receiving acknowledgment votes from the followers and committing the updates when processing the requests.

- **Follower [48]:** The Follower class defines the functions of a follower defined by the ZAB. The main task of the follower involves forwarding the write requests to the leader through *FollowerRequestProcessor* and processing the read requests locally. The local read requests are required to be sequentially consistent with the write requests. The *getZxid* method returns the current transaction id under processing and the *getLastQueued* method returns the last transaction queued. These methods helps to ensure the sequential consistency of the read requests.

- **Learner [49]:** The learner class defines the functions of a follower recovering from failure. The main function of the learner is to connect and register the follower with the leader. The follower can be a new node or a node recovering from failure. Once the emerging node is registered with the leader, it talks and synchronizes with the leader.

After synchronization, it activates and transfers control to the follower thread which can serve requests.

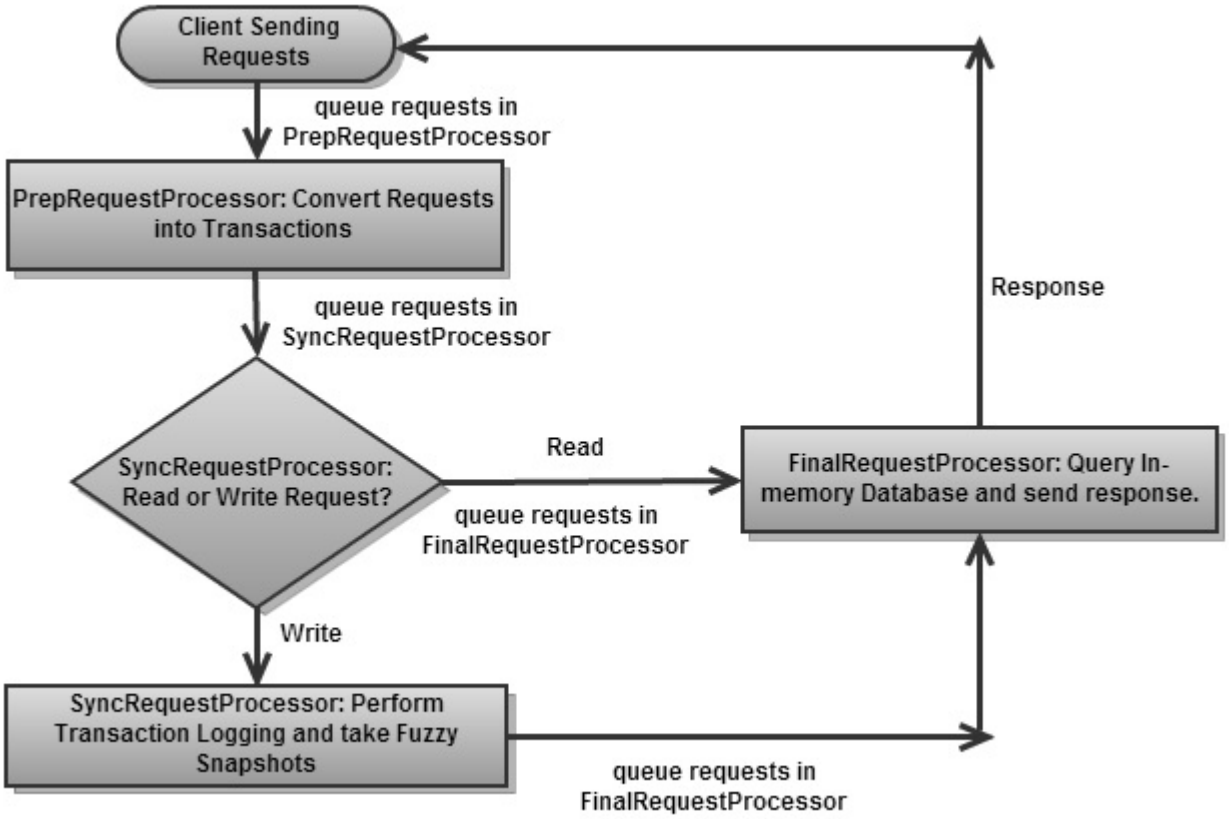

Figure 4.1: Requests flow of control between different processor queues in Zookeeper.

4.2 Internal Data Structures in Zookeeper

This section explains the internal data structures of Zookeeper. The main constructs of the Zookeeper are TCP communication between the nodes(leader

and followers), queues for transferring data between the different processors of a node, and the structure of the in-memory database.

- The TCP communication in Zookeeper is implemented through the *ServerSocketChannel* [38] class in Java. It is a part of the Network Input Output (NIO) package. It is a concurrent implementation of naive TCP. The implementation is non-blocking and event driven which offers high performance. The NIO channel consists of a pool of threads that implements a multi-threaded TCP. The documentation of NIO channels and its corresponding benchmarks can be found in [42].

- The second data structure of importance in Zookeeper are the queues used to transfer data between different processes of ZAB. *LinkedBlockingQueue* [39] under the *java.util.concurrent* package is the data structure used to implement the queues. It is a blocking queue implemented using a linked list. It provides First In First Out (FIFO) access to the data. Although *LinkedBlockingQueue* has less predictable performance for concurrent applications [39], it is the best data structure that supports the use case of Zookeeper. Another competing data structure-*ConcurrentLinkedQueue* [40], provides wait free concurrent access to the queue but certain aspects of the queue implementation make it unsuitable for Zookeeper. The memory consistency effect [40] is the important reason for the unsuitability of *ConcurrentLinkedQueue* in Zookeeper. Also, the *size* method in *ConcurrentLinkedQueue* is not

a constant time operation. There are many operations that check the size of queue in the ZAB implementation and the performance worsens in these cases by using the *ConcurrentLinkedQueue*.

- The last important data structure used in Zookeeper is the in-memory database of Zookeeper. The in-memory database is a Directed Acyclic Graph (DAG) implemented using *ConcurrentHashMap* [41] in Java. *ConcurrentHashMap* is a concurrent implementation of hash table. It provides thread safe updates to the data. The reads are not thread safe in *ConcurrentHashMap*. This means a thread can update the data while another thread is reading the data. Since, Zookeeper already orders the requests and a read on the *ConcurrentHashMap* will always see the most updated view of the data. This data structure is well suited for querying Zookeeper as well as the internal operations of fuzzy snapshot in ZAB.

4.3 System Implementation

This section explains the implementation of a low disk bound persistence layer to Zookeeper. It starts with a discussion of the Busy Snapshots algorithm followed by the strategy for the conversion of fuzzy snapshot into a pure complete snapshot. Next, the failure recovery mechanism supported by this design is discussed. Finally, the integration of our implementation with Zookeeper and the new sequence of operations are explained.

4.3.1 Busy Snapshots

As discussed in Subsection 2.4.2, the transaction logging mechanism blocks every write operation in Zookeeper. So, until the log of a request is written to the disk and synchronized, the requested operation cannot be performed. Along with this, a fuzzy snapshot is taken periodically. This snapshot is completely concurrent to the requests and it does not block the requests on Zookeeper.

The main idea behind the busy snapshot algorithm is to purge the transaction logging that is blocking the requests. Instead of writing the transaction log on the disk, the request is pushed into a newly introduced queue called the *AtomicRequestProcessor*. The *AtomicRequestProcessor* is a *LinkedBlockingQueue* that is used to atomically transfer requests from the *SyncRequestProcessor* to the *FinalRequestProcessor*. Acknowledged requests are pushed into the *AtomicRequestProcessor* queue and stored until the request processing is complete. The request in the *AtomicRequestProcessor* queue is de-queued once it is committed by the *FinalRequestProcessor*. On any failure in the interim, the requests in the *AtomicRequestProcessor* queue are backed up by other nodes in the clusters, so that they can be applied by the newly elected leader on resumption. This provides persistence to the acknowledged requests in the flight without disk dependency. Along with the *AtomicRequestProcessor*, a fuzzy snapshot thread is run whenever possible. The snapshot frequency is as follows: every write request on Zookeeper

tries to start a fuzzy snapshot. If there is no existing fuzzy snapshot thread running, a fuzzy snapshot is started. The exact frequency of the snapshot depends on the number of *znodes* in the Zookeeper state and the frequency of write requests. Precisely, the frequency of snapshots is inversely proportional to the number of *znodes* in Zookeeper. The reason is that the higher the number of *znodes* in Zookeeper, the longer is taken for a single snapshot thread to complete. So, the frequency of snapshots becomes lower. According to the Zookeeper use case, the in-memory database is not expected to store application data that can occupy large space. So, the frequency of the fuzzy snapshots is expected to be high in this algorithm since the amount of in-memory data is expected to be small. The pseudo code for the busy snapshot algorithm is shown in Algorithm 4.1.

Algorithm 4.1:Busy Snapshots

Require: Request R
  if (R **not** null) **and** (R **not** read) **and** (snapInProcess **not** alive) **then**
    Write the request into AtomicRequestProcessor queue
    Create and run a fuzzy snapshot thread called a snapInProcess
  else
    Write the request into AtomicRequestProcessor queue
    Snapshot thread already running
  end if

This algorithm backs up a partial state of Zookeeper on the disk concurrently without any overhead on the requests in Zookeeper. But as explained in Subsection 2.4.3 the Fuzzy Snapshot contains only the partial state with-

out the transaction log. When a Zookeeper node fails and resumes, it recovers the partial state from the most recent fuzzy snapshot. In the default version of Zookeeper, the missing state in the fuzzy snapshot will be restored from transaction logs. But in our algorithm, there are no transaction logs. The alternative place to find the complete state is the leader of the ensemble. So, the missing data is restored from the leader of the ensemble. This restoration mechanism is explained in Section 4.1 under the items leader and learner respectively. So, the recovering node will be eventually consistent with the ensemble.

The Busy Snapshot algorithm vastly increases the frequency of snapshots compared to the default Zookeeper. In the default Zookeeper the snapshot interval is 100,000 requests. The reason behind increasing the frequency is to minimize the number of missing data to be restored from the leader during failures. The leader in the Zookeeper ensemble can be a machine located in a geographically different location connected through communication networks. So, minimizing the amount of missing data to be restored from the leader vastly improves the failure recovery time of a node.

Increasing the frequency of snapshots increases the storage space required to store the snapshots. Essentially, only the most recent snapshot is needed for recovery during failure. So, there is a mechanism implemented to delete the old snapshots. This can be configured using the parameter *PurgeTime* and

NumberToRetain in the Zoo.cfg file. The *NumberToRetain* denotes the number of snapshots to retain on the disk. The value *NumberToRetain* should be at least 1. The *PurgeTime* denotes the time interval to clear the snapshots. It takes the value of time in milliseconds.

The Busy Snapshot algorithm works fine as long as there is a leader in the cluster that can restore the missing states of a recovering node. The problem arises when the leader or quorum fails, the state is lost as the leader from which the data to be restored itself has failed. So, there has to be some mechanism that can convert the Fuzzy Snapshot into a complete snapshot. Subsection 4.3.2 explains the mechanism and checkpoints for the conversion of a Fuzzy Snapshot into a complete snapshot during such failures.

4.3.2 Conversion of Fuzzy Snapshot to Complete Snapshot

As discussed in Subsection 4.3.1 the fuzzy snapshots taken by the busy snapshot algorithm do not persist the complete state of Zookeeper during certain failures. So, at some checkpoints the fuzzy snapshot should be converted into a complete snapshot. As discussed in Subsection 3.3.2 the level of persistence provided by our design is durability at failure. So, failures at some nodes in the ensemble are the best checkpoints at which the fuzzy snapshot can be converted into a complete snapshot.

According to the ZAB implementation whenever a leader or the quorum fails, the entire ensemble is restarted in recovery mode. In the recovery mode, the leader election algorithm takes place to elect the leader and form the quorum. Our goal is to make the elected leader contain the recent complete state. Also, according to ZAB guarantees a Quorum fails when more than f nodes fail out of *2f+1* nodes. Also, it is assumed that the entire ensemble never fails. This is because when more than f servers fail, the remaining server nodes are stopped from serving requests. So, the probability of remaining non-operational servers failing is very low. The proposal is to make at least one of the non-operational server nodes contain the complete state when restoring.

Firstly, let us consider the case of leader failure. When the leader fails, the other followers in the ensemble restart in recovery mode and the leader election algorithm begins. According to our design before restarting the other nodes in recovery mode, the entire final state of Zookeeper is taken as a snapshot into the disk. Since the leader has failed there cannot be any write operating under processing on the in-memory database. This is because, the write requests are ordered and broadcasted through the leader and the failure of leader stops the processing of write requests. Thus the snapshot taken at this time will contain the recent and entire state of zookeeper. So, when the nodes start back in recovery mode, at least one of the non-failed nodes will

contain the recent complete state and it will be elected as the leader according to the postulates of the leader election algorithm. The implementation of the complete snapshot involves the detection of a leader failure event in the follower class and a call to take a snapshot of the in-memory state of Zookeeper before shutdown. The Algorithm 4.2 shows the sequence of steps handled at this checkpoint. The Second case is the quorum failure. Ac-

Algorithm 4.2: OnLeaderFailure

1. Stop FollowerRequestProcessor.
2. Back up the requests in the AtomicRequestProcessor queue as a serialized object on disk and stop SyncRequestProcesssor.
3. Stop FinalRequestProcessor.
4. Call takeSnapshot() and snapshot the complete in-memory state.
5. Restart in recovery mode.

cording to ZAB assumptions, when a quorum fails at least *f+1* servers fails and at most *f-1* remain non-operating waiting for the quorum to form again. This is another checkpoint at which the requests are not served. So, at this stage a complete snapshot of the Zookeeper state is taken by the other non-failed nodes. When the quorum re-forms back one of the non-failed *f-1* nodes must have the complete recent state and it is elected as the leader. From the leader, the other nodes can now restore the data missing in their state. The steps involved in this algorithm are listed in Algorithm 4.3. Whenever the new leader is elected, it first plays the acknowledged, uncommitted requests on the AtomicProcessorQueue left by the previous failed leader. Once these requests are played back, the leader resumes serving the requests of the clients. This ensures the uncommitted requests in flight to be durable during

failures.

Algorithm 4.3: OnQuorumLoss

if (isLeader()) **then**
1. Stop PrepRequestProcessor.
else
2. Stop FollowerRequestProcessor.
end if
3. Back up the requests in the AtomicRequestProcessor queue as a serialized object on disk.
4. Stop SyncRequestProcessor and FinalRequestProcessor.
5. Call takeSnapshot() and snapshot the complete in-memory state.
6. Put the Server in zombie mode and wait for the quorum to form.

4.3.3 Failure Recovery

This section is an extension of the previous Section. It discusses the durability guarantees provided by the modified persistence layer. As explained in Chapter 3, there are three main cases of failure to be considered in Zookeeper's replicated environment. They are failure of a follower with stable quorum, failure of leader and failure of follower with quorum loss.

4.3.3.1 Follower Failure Recovery and Ensemble events

This section discusses the recovery of a failed follower in the ensemble. In this case, it is assumed that the failure of a follower doesn't cause the quorum to fail. So there is a leader with a quorum of followers still operating the Zookeeper ensemble.

In a Zookeeper ensemble when a single follower fails due to a crash, it immediately goes offline. The other nodes in the ensemble see this failure but still does not change their routine working if the failure of this follower do not affect the quorum or the leader. When the failed follower resumes, it checks for the snapshots on the disk. If there is no snapshot, it directly registers with the leader and starts synchronizing it's complete state with the leader as a Learner. This case happens only in the case of a new node joining the ensemble. In the other case, if the node has a snapshot on the disk, it restores with the partial state in the Fuzzy Snapshot. Then, it registers with the leader and starts synchronizing the missing state. The second case is the most prominent one in this type of failure recovery.

4.3.3.2 Leader Failure Recovery and Ensemble events

This scenario involves the failure of the leader in the ensemble. The failure of the leader leads to the restart of all the nodes in the ensemble in recovery mode. This is followed by the leader Election algorithm to elect a leader to bring the ensemble into operation.

When a leader fails due to a crash, it immediately goes offline. All the unacknowledged requests in flight in the PrepRequestProcessor and SyncRequestProcessor gives a time out error. The clients of these requests are notified to send the requests again with the help of TCP timeout error. The other nodes in the ensemble observe the failure of the leader. According to

our strategy discussed in Subsection 4.3.2, the timeout from leader fires an *OnLeaderFailure* checkpoint handler in the non-failed follower nodes. So, all the other non-failed nodes before they stop take a complete snapshot of their state into their persistent store. In addition, they also backup the requests in the FinalRequestProcessor queue. Before backing up the FinalRequestProcessor, the uncommitted requests in the AtomicRequestProcessor queue are written on the disk as a serialized object. This is to ensure that the uncommitted requests in flight are persisted. Then, the nodes restart in recovery mode. When they restart in recovery mode, the leader Election algorithm takes place. In the complete snapshots, the node with most recent state has the maximum *zxid*. Hence, it is elected as the leader according to the leader election algorithm. The remaining nodes join the leader to form the quorum and brings back the Zookeeper ensemble.

4.3.3.3 Quorum Failure Recovery and Ensemble events

This scenario involves the failure of a follower that causes the ensemble to lose quorum. So, the leader and the other non-failed nodes go into the zombie mode to prevent further process crashes. In this state the Zookeeper ensemble cannot serve any requests until the quorum is formed.

When a follower encounters a crash fail, it immediately goes offline. In this case, the failure of this follower causes the loss of quorum. When the leader encounters the loss of quorum, it stops all the PrepRequestProcessor,

SyncRequestProcessor and the FinalRequestProcessor threads. The leader backs up its complete state as well as the requests in the AtomicRequestProcessor queue into the disk and waits for the quorum to form. The other follower nodes in the ensemble also encounters the loss of quorum and repeats the same steps as the leader to take a complete snapshot. When the failed node recovers, the leader election algorithm starts and elects the leader. The elected node will have the complete state in the snapshot as well as the uncommitted requests in flight at the time of crash. The remaining nodes will join the leader to bring the ensemble to operation.

4.3.4 Integration with Zookeeper

This section deals with the integration of the Busy Snapshot algorithm and the failure recovery mechanisms with Zookeeper. It also explains the changes in state of the ensemble during failures and the durability provided by the new scheme.

The major code changes in the implementation of this algorithm are in the *SyncRequestProcessor*, *Leader* and the *Follower* classes. The Busy Snapshot algorithm explained in Subsection 4.3.1 is implemented as a part *SyncRequestProcessor*. It is a modification to the Fuzzy Snapshot technique and is implemented in *SyncRequestProcessor*. The flowchart in Figure 4.2 shows the sequence of steps in the Busy Snapshot algorithm. The second major code modification is in the leader and the follower threads to handle failure of other

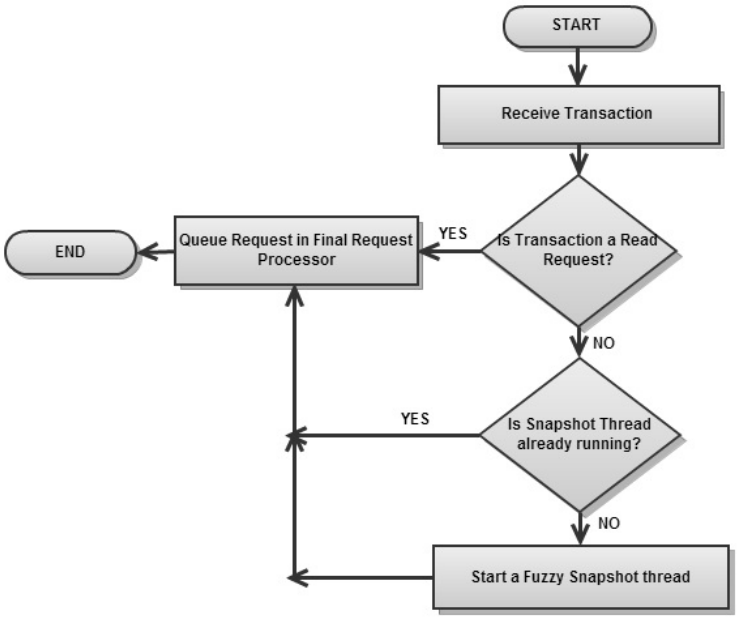

Figure 4.2: Flow diagram of Busy snapshot algorithm.

nodes. In the leader branch of *QuorumPeer* thread, a handler is written to implement the checkpoint activities during the failure of quorum as discussed in Subsection 4.3.2. Similarly, in the follower branch of *QuorumPeer* thread, a handler is implemented to perform the checkpoint activities during the leader failure or quorum failure as discussed in Subsection 4.3.3.

The third addition to Zookeeper is the implementation of a functionality to clear the obsolete snapshots from the persistent store. The class named DisklessPurgeSnapshot is implemented for this. The new class has two attributes namely PurgeTime and NumberToRetain. These two attributes are analogous to the configuration parameters PurgeTime and NumberToRe-

tain defined in Subsection 4.3.1. This class implements the Java thread (Runnable) interface that deletes the old snapshots from the disk between the interval specified by the *PurgeTime*. This thread is started as a part of starting the quorum in the *QuorumPeerMain* class and it runs for the entire life cycle of Zookeeper. Figure 4.3 shows the class diagram and Figure 4.4 shows the flow chart of the *DisklessPurgeSnapshot* thread.

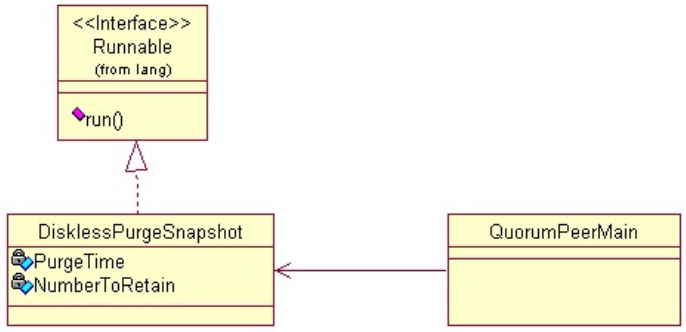

Figure 4.3: Class diagram of DisklessPurgeSnapshot class.

Finally, in this section we discuss the overall life cycle of different nodes in a Zookeeper ensemble. It discusses the sequence of transitions that happens to a Zookeeper node from its inception. Figure 4.5 shows the changes in state of a node during different activities in a Zookeeper ensemble.

1. A node joining the Zookeeper ensemble, is initially in *New* state. From the *New* state, the node immediately undergoes the leader election

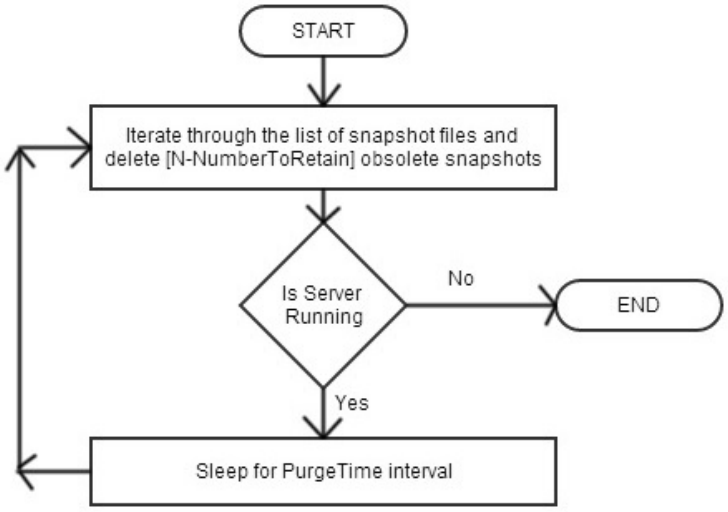

Figure 4.4: Flow diagram of DisklessPurgeSnapshot thread. N denotes the total number of snapshots on the disk.

algorithm if there is quorum available. Otherwise the node moves to the Zombie state waiting for the quorum to form. If the quorum can be formed or already exists the new node is elected either as a follower or a leader according to the leader llection algorithm and moves to the appropriate elected state. If the node is being elected a follower, it moves through a state called *learner* where it synchronizes with the ensemble Leader. Once the synchronization is over it moves to the *follower* state where it can serve requests.

2. The second major state is the *leader* state. In the *leader* state, the node keeps on broadcasting requests. This works fine until some failure occurs. The first failure can be a crash of the leader itself. In this case

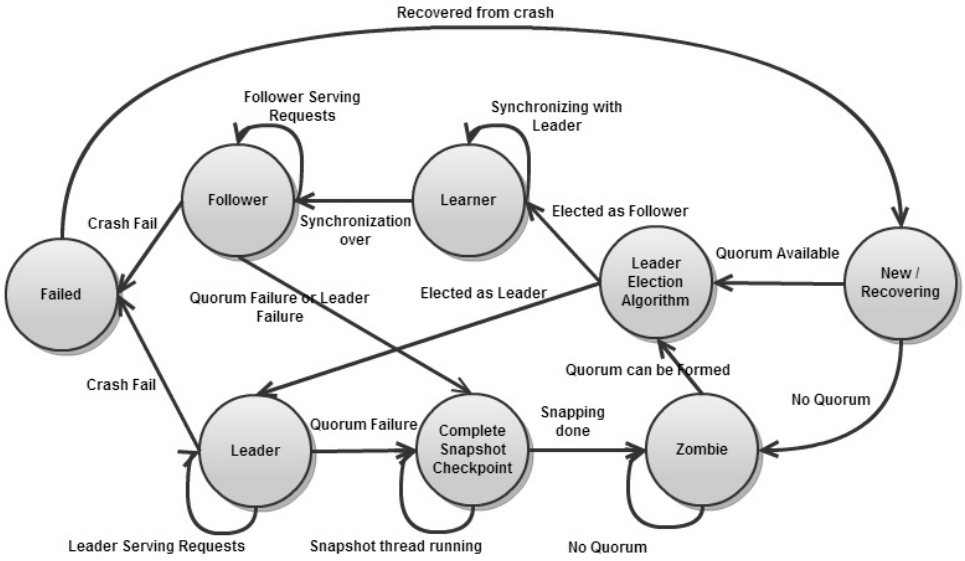

Figure 4.5: State transitions of a node during different events in a Zookeeper ensemble.

the Leader node moves to the *Failed* state and remains there until it is restarted by some means. When the failed Leader is recovered, it moves to the *New* state and follows the transition from *New* state as mentioned in step 1. The second transition from this state can be the loss of quorum. In this case the Leader backs up the uncommitted requests in its *AtomicRequestProcessor* Queue and stops serving further requests. It moves to the *Complete Snapshot Checkpoint* state and takes a complete snapshot of its state. Once the snapshot is over, it makes a transition to *Zombie* state where it waits for the Quorum to form and proceeds with recovery mode again.

3. The last important state is the *Follower* state. In this state, the fol-

lower node keeps serving requests according to the ZAB protocol. The followers stops operating either when the Leader fails or the quorum fails. In either of the cases, the follower node moves to the *Complete Snapshot Checkpoint* state to backup a complete snapshot into the disk. From the *Complete Snapshot Checkpoint* state, it follows the same sequence of steps as mentioned in step 2.

Chapter 5

Testing and Evaluation

This chapter involve testing and evaluating the research implementations. It starts with testing the entire system for correctness and regressions. These tests are explained in Sections 5.2.2 and 5.2.3 respectively. This is followed by the evaluation of the integrated system in Section 5.3. The evaluation involves measuring the various performance metrics of the system and benchmarking it according to the standards. This is dealt with in Subsection 5.3.1. Once the performance results are computed, they are compared with the existing system. The results of the performance comparison and the inferences are explained in Subsection 5.3.2. Finally, the trade-off between our two main research goals namely performance and durability are explained in Subsection 5.3.3.

5.1 Approach

First, as a proof of our implementation the system needs to be tested for correctness. The testing in our project involves two tasks. The first task is to test the correctness of the implementations. This is done through sanity testing as discussed in Subsection 5.2.2. Sanity testing verifies the validity of the implementations achieved by the research ideas namely *Diskless clustering* and *Durability at Failure*. The second task involves regression testing the product which is explained in Subsection 5.2.3. As the ideas of this research are implemented as changes to Apache Zookeeper, regression testing is a good tool to prove that the core properties and working of Apache Zookeeper are never compromised.

Once the system is tested for correctness, the next task is to evaluate the performance of the system. The evaluation involves comparing the performance of the current system with the performance of the default Zookeeper. The major metric involves measuring the throughput of various operations in Zookeeper, recovery time of the ensemble during failures and the performance of use cases benefiting from the Diskless Zookeeper. Apart from this, the resource utilization of the current and the existing implementation is compared and analyzed. Finally, a brief analysis of the trade-off between performance and durability of the resultant Zookeeper is discussed. This mainly concentrates on analyzing the performance gains obtained through the research

implementations and the trade-off with the durability that offered this gain in performance.

5.2 Testing

Testing is the process of evaluating a product for the conformance to requirements and quality. Testing takes a set of input and the output produced verifies that it conforms to the requirements in Section 3.2.1. The following section explains in detail the methodologies and techniques used in our testing.

5.2.1 Sanity Testing

Sanity testing involves a quick evaluation of the implementation. It is used to check whether the implementation is working sensibly to proceed with further testing. So, the task is to check the list of claims that are implemented for correctness. The main implementations of the research are:

1. The Busy Snapshot algorithm

2. The failure recovery mechanism during a node recovery.

The first part of this testing involves verifying the claims of the Busy Snapshot algorithm. The main goal of the algorithm is to purge the transaction logging mechanism and take only snapshots in a fuzzy interval as explained in Section 4.3.1. So, the test involves checking the *dataDir* for the data that is

written on the disk. After creating some *znodes* in Zookeeper, if the *dataDir* contains only snapshots then the Busy Snapshot algorithm is working correctly. In the other case, if there are logs written to disk or if there are no snapshots then the algorithm fails. The Algorithm 5.1 shows the sequence of steps in the test case. The result of this test algorithm returns *True* on a

Algorithm 5.1: Busy Snapshots Test

Require: Zookeeper Directory Path(P)
Ensure: True
  Change directory to P
  Get *dataDir* from Zoo.cfg
  Create a *znode* in Zookeeper
  Change directory to the path mentioned by *dataDir*
  Get the files in the directory into *Files[]*
  if *Files[]* **contains** Transaction Logs **then**
    return False
  else if *Files[]* **contains** Snapshots **then**
    return True
  else
    return False
  end if

Zookeeper running the Busy Snapshots algorithm correctly.

The other part of the sanity testing involves testing the failure recovery mechanism implemented in Zookeeper. For this test, Zookeeper ensemble consisting of 3 nodes is created. Hence, the failure of 1 node can be tolerated by the quorum. If more than 1 node fails, then the quorum fails in this scenario. Failures are essentially induced by restarting a server node. While restarting, checks can be made to verify whether the node recovers its

Algorithm 5.2: Failure Recovery Test

Require: Zookeeper Server Address(F1,F2,L)
Ensure: True

```
Create some Random Number of znodes in Zookeeper
Restart Server F1 and Wait till it resumes
Get the Number of znodes in F1 as F1N and L as LN
if F1N == LN then
  Kill L
  Check if Leader Election Works properly. Assume F2 now becomes new
  Leader(L1)
  Get the number of znodes in L1 as L1N
  if L1N == LN then
    Restart the killed Leader. Previous Leader L now becomes F2.
    Get the number of znodes in F2 as F2N
    if F2N == L1N then
      Restart F1 and F2. Wait for the servers to restart
      Get the number of znodes in F1, F2, L1 as F1N, F2N and L1N
      respectively
      if (F1N == L1N) and (F2N == L1N) then
        return True //All Failure recovery scenarios passed.
      else
        return False
      end if
    else
      return False
    end if
  else
    return False
  end if
else
  return False
end if
```

data from the ensemble properly. *Zookeeper Four Letter Words* [50] (FLW) monitoring tool is used to assist this test. The FLW is used to check the various parameters of the Zookeeper server like the *zxid*, current Leader, *znodes* count, server status etc. Essentially, in our tests, *znodes* count helps to prove that the different Zookeeper nodes contain the same number of *znodes*. This can be used to check whether a recovering node properly restores it's data from the cluster. The algorithm 5.2 lists the sequence of steps involved in the test. The Algorithm induces different types of failure discussed in Section 3.3 and verifies whether the failed node is able to recover and join the quorum properly. Also, this test verifies the durability guarantee ensured by Zookeeper. The durability guarantee is ensured by checking the consistency of failed nodes after rejoining the ensemble.

5.2.2 Regression Testing

The main goal behind regression testing this project is to check that the original operations on Zookeeper have not changed. The first part of regression testing involves testing the fundamental operations like create, read and delete on Zookeeper. The sequence of steps for this test is listed in Algorithm 5.3. The algorithm returns True only if all the operations runs properly without any exceptions. This ensures the proper working of the various request processors in Zookeeper with our research modifications. Also, this test uses the same client library as the Zookeeper API. This verifies that our implementation is transparent to the software developer and the operations

in Zookeeper. The Second part of the regression testing involves testing

Algorithm 5.3: Regression Test-Operations

```
Require: Zookeeper Server Address
Ensure: True
  Connect to Zookeeper Server
  try
    Create a node /a with data "a"
    Get the data from node /a and display it
    Delete the node /a
    return True
  catch exception
    return False
  end
```

the proper restoration of data into Zookeeper during failures. This involves creating a sequence of znodes into a Zookeeper ensemble, inducing failure of Zookeeper nodes in between and checking the proper restoration of znodes into the in-memory database of Zookeeper. The algorithm 5.4 lists the steps in this test. Two threads are created in this test. One of them keeps creating sequential znodes in Zookeeper. The other thread randomly induces failures as mentioned in the algorithm. After all types of failures are induced, both the threads are stopped. The number of znodes create requests sent and the number of znodes in the Zookeeper server are checked for equality. If the number of znodes are equal, then the test case passes showing the proper working of the restoration mechanism in Zookeeper during failures.

Algorithm 5.4: Regression Test-Restoration

Require: Zookeeper Server Address
Ensure: True
Connect to Zookeeper Server.
Create Thread 1 that keeps creating Sequential znodes on a Zookeeper ensemble.
Create Thread 2 that randomly causes failure of the Zookeeper ensemble.

Stop Thread 1 and Thread 2.
Count the number of znodes created by Thread 1 as N1 .
Count the number of znodes in each of the servers as N2.
if N1 == N2 **then**
  return True
else
  return False
end if
end

5.3 Evaluation of the System

This section involves evaluating the performance of Zookeeper. The benchmarks utility designed to measure the performance is based on the Zookeeper smoke test in [51]. This benchmark suite is customized and redesigned to fit our needs to measure the various functionalities as well as the resource requirements of Zookeeper. The same benchmark suite is used to evaluate the default and the modified versions Zookeeper. The results of these benchmarks are used to analyze the variation in performance with the existing Zookeeper.

5.3.1 Benchmarking

The set of benchmarks for this research is divided into five groups. They mainly concentrate on the performance of various operations in Zookeeper and the time latency to restore a zookeeper server node during failure. In addition to the performance evaluation of basic operations in Zookeeper, the performance of various internal operations like leader election and snapshotting that influences the failure recovery are also presented. Finally, the example use case, Message Queue, is analyzed and the improvement in performance by using the new persistence layer is discussed.

5.3.1.1 Zookeeper Operations Benchmark

As discussed earlier, Zookeeper is like a file system. The major operations are write and read. The write operation - *Create* and the read operation - *Get* are analyzed in our benchmarks. The system configuration used in our tests is listed in Table 5.3.1. Apart from this, the Zookeeper configuration used for the default and modified Zookeeper are listed in Appendix A.2 and A.3 respectively.

The major performance metric analyzed in our benchmarks is the throughput of the operations. The throughput defines the number of requests that can be served by the Zookeeper ensemble in a given time. This benchmark measures the variation of throughput over time. With a Zookeeper client requesting at it's maximum synchronous speed over time, the test shows the

Parameters	**Values**
Zookeeper Version	Zookeeper-3.4.5
Number of Nodes in the Ensemble	3
Java	OpenJDK 1.6 64-bit
Java Heap Size	xmx8192m, xms512m
Network	Localhost Loopback
Number of Requesting Client threads	60
Client Request Mode	Synchronous

Table 5.1: System Configuration for Benchmarks.

maximum number of requests that can be served by the Zookeeper. The test is performed with 60 parallel client threads writing on Zookeeper and the average throughput of each thread is measured as the throughput. The client threads are limited to 60 which is the optimal scalability limit defined by Zookeeper. To achieve consistent results, the test is repeated 3 times and the average of the results is used as the benchmark. This scenario is similar to the way Zookeeper is used in production. The algorithm for this test is listed in Algorithm 5.4. The test code template for this benchmark is listed in Appendix. A.11. The graph in Figure 5.1 and Figure 5.2 shows the evaluation of *Create* and *Get* operations in Zookeeper. The benchmark results for *Create* and *Get* operations are presented in Appendix A.5 and A.6 respectively.

As shown in Figure 5.1, the write performance has a 30 times speed up compared to the normal disk based Zookeeper in both the tests. This is mainly due to the reduction in the disk dependency of the persistence layer.

Algorithm 5.5: Zookeper Operations Timer Test

Require: Zookeeper Server Address (IP:PORT), Test Duration in seconds (TD)
Ensure: Throughput
  Connect to Zookeeper Server (IP:PORT)
  Measure the start time as ST in seconds.
  while $(CurrentTime - ST) \leq TD$ **do**
    Perform the Operation to be Benchmarked.
    Number of Completed Requests, NOC++.
  end while
  Throughput = NOC / TD
  return Throughput

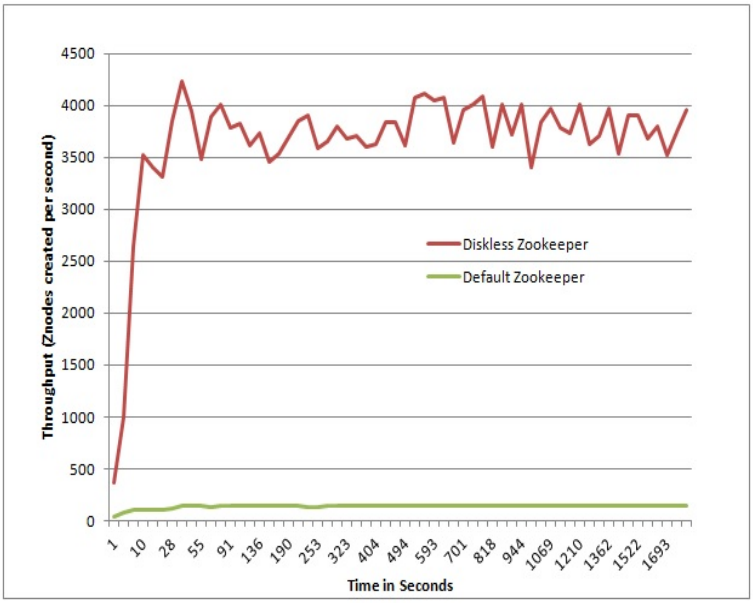

Figure 5.1: Zookeeper - Write performance evaluation. Throughput of znodes creation is measured against time. The throughput values are measured on a test over a duration of 30 minutes with a client sending requests at its maximum synchronous speed.

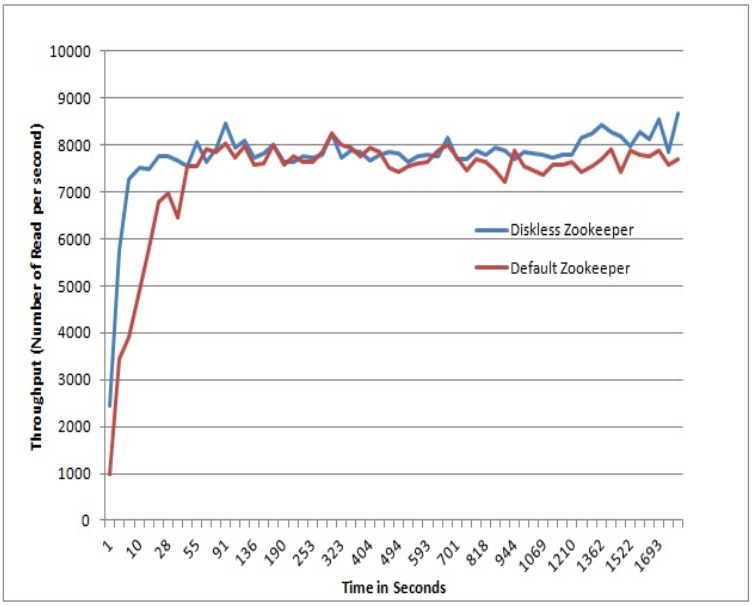

Figure 5.2: Zookeeper - Read performance evaluation. Throughput of znodes read is measured against the time. The throughput values are measured on a test over a duration of 30 minutes with a client sending read requests at its maximum synchronous speed.

Also from Figure 5.2, the performance of read operations has not changed, it remains similar to the performance in the existing Zookeeper. The read requests in Zookeeper are not dependent on the disk, so there is no variation in the read performance between the default and diskless Zookeeper.

5.3.1.2 Snapshot Latency

This section measures the time taken for the major operation in the Busy Snapshot algorithm. It is the time taken to complete a Fuzzy Snapshot of the Zookeeper's in-memory state. The latency of this operation is important

because a complete snapshot of the in-memory is taken during failures in addition to the fuzzy snapshots which in-turn affects the restoration time of the Zookeeper nodes. So, overhead involved in restoration can be calculate based on the Snapshot latency.

The first internal operation is the time taken for a Fuzzy snapshot. A Fuzzy Snapshot is a depth first scan (DFS) over the directed acyclic graph (DAG) of the Zookeeper's in-memory database. Also, the in-memory graph structure of Zookeeper does not have edges connecting the siblings and the number of edges equals the number of *znodes*. The time complexity of DFS is linear over the size of the Graph. Precisely, the time complexity is *O(E)* where E is number of edges in the Graph. So, the time taken for the fuzzy snapshot increases linearly with the number of the *znodes* in the in-memory database. This time taken for taking a snapshot directly affects the restoration time of a Zookeeper node as a snapshot is taken before restoration during failures. This restoration time evaluation is presented in the following section. The Figure 5.3 shows the variation of the snapshot time with the number of *znodes* in Zookeeper's state. Appendix A.6 lists the results of the snapshot latency test.

5.3.1.3 Restoration Time

The restoration time is the time required for the Zookeeper ensemble to resume operation during failures. The time for the restoration is measured

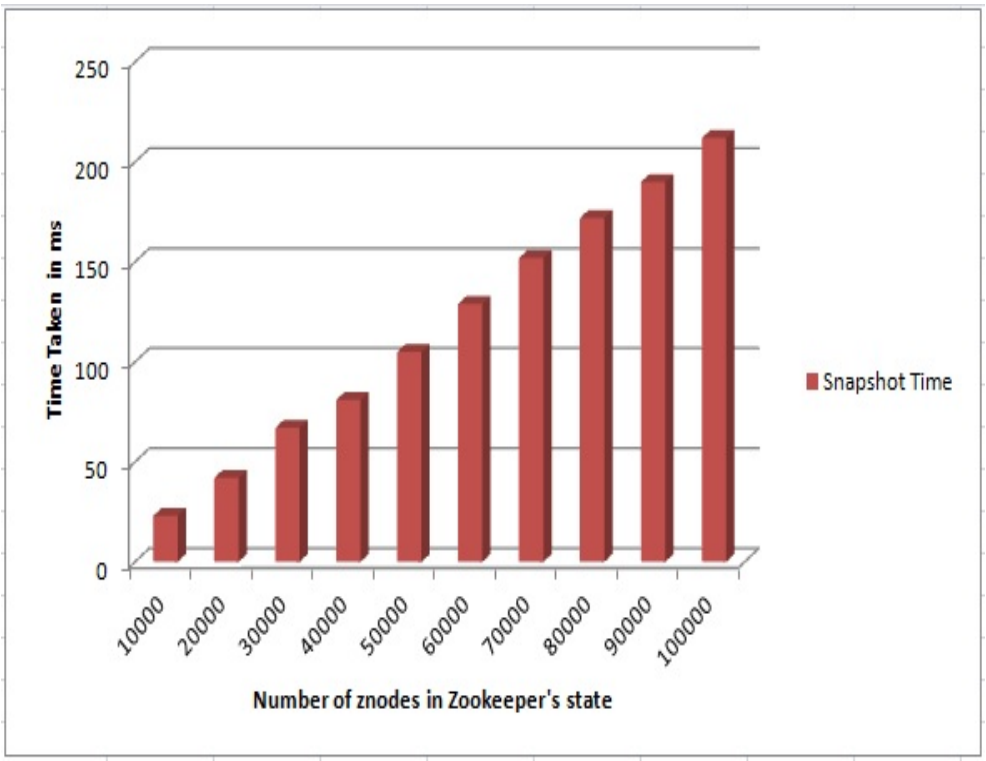

Figure 5.3: Snapshot latency evaluation graph. The graph measures the time taken for backing up a snapshot on the disk with a given number of znodes in the Zookeeper state. The size of 1 *znode* equals 300 MB.

only during the period when there is a quorum available for the ensemble to resume operation. Hence the time in the Zombie state where the ensemble is waiting for the quorum to form is not considered. Precisely, the time taken for a node to resume operation is the sum of time taken for the Fuzzy Snapshot during shutdown and the time taken for the Leader Election algorithm when the quorum is formed. The graph in Figure 5.4 shows the comparison of the time taken to recover a Zookeeper node between default and diskless

Zookeeper.

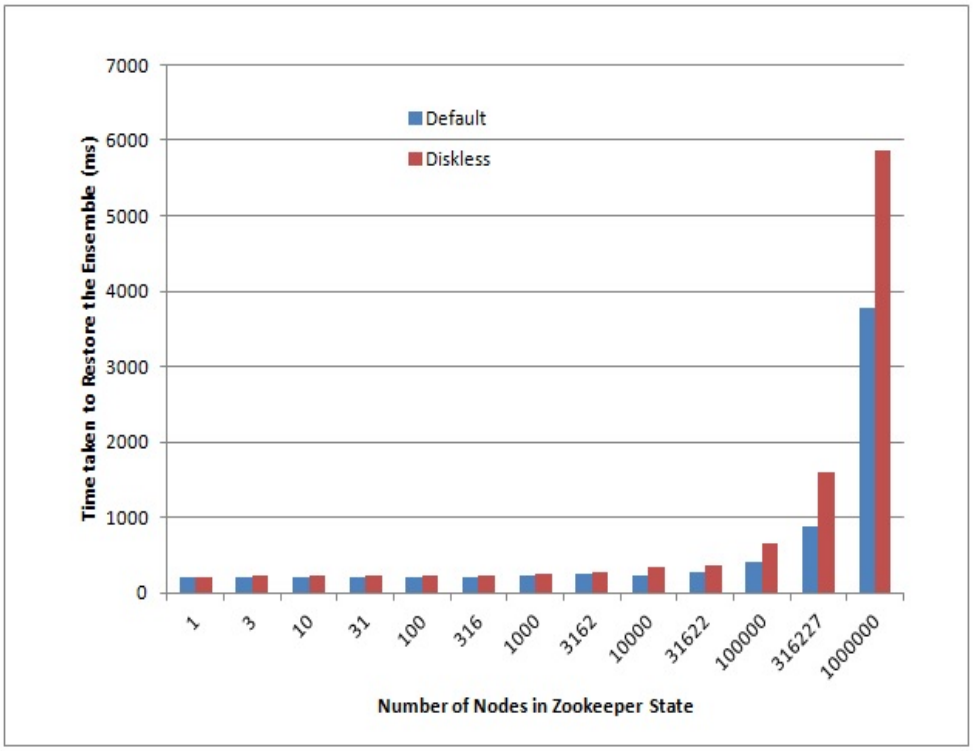

Figure 5.4: Zookeeper ensemble restoration latency evaluation. The graph measures the time taken for ensemble restoration during failure with a given number of znodes in the Zookeeper state. Leader failure is the type of failure induced in this test. The size of 1 *znode* equals 300 MB.

5.3.1.4 Resource Utilization

This section explains the resources used for the operation of Zookeeper. The main resources to be monitored is the percentage of CPU consumed for the operation of Zookeeper. The CPU percentage is monitored because we in-

crease the frequency of the Fuzzy Snapshot thread and also a new thread to delete obsolete snapshots. The other parameters like Java heap size and direct mapped memory are not measured as our modification does not regress these properties. Visual VM [52] is used to monitor these parameters. Figure 5.5 shows the average CPU percentage used by diskless and default Zookeeper. The test is run by monitoring a Zookeeper server when it is serving write requests at a rate of 4000 znodes per second. The test is run for a duration of 30 minutes repeated three times and the average values over the time period are taken as the benchmark. The benchmark results for this test are listed in Appendix A.7. The diskless Zookeeper on average has 0.3% more CPU usage than the default Zookeeper. This increase in CPU usage can be contributed to the increase in the frequency of the snapshot thread and the other thread to delete old snapshots. However, this increase in CPU usage is a very low overhead and is negligible.

5.3.1.5 Use Case: Message Queue Benchmark

The main idea behind the low disk bound Zookeeper is to improve the write performance of Zookeeper. Message Queue is one of the major use cases of Zookeeper that has a write intensive work load. In this use case Zookeeper is basically used as a Queue. Producer processes creates data as a Sequential znode into Zookeeper. sequential znodes are numbered by their order of creation. A consumer process reads and deletes the znode in the increasing order of the sequential number. The create and delete operations are

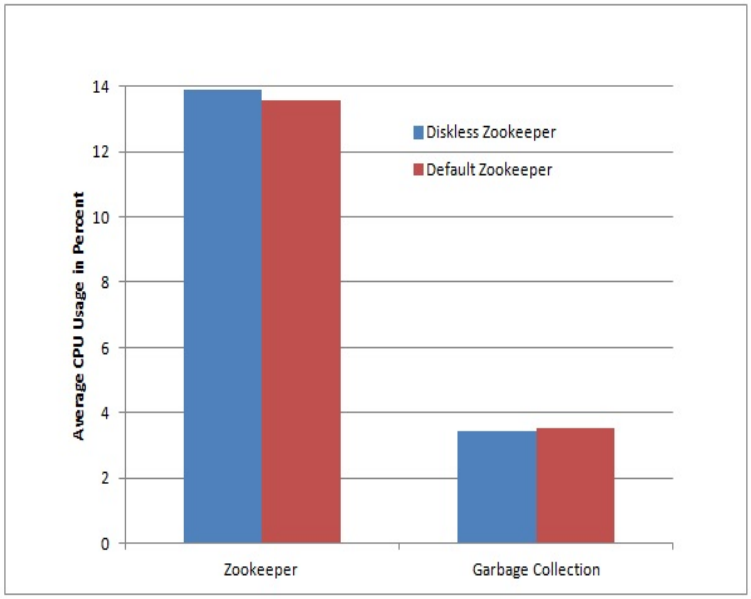

Figure 5.5: CPU utilization of Zookeeper. The first bar shows the total CPU utilized by Zookeeper. The second bar shows the percentage CPU used by garbage collection for Zookeeper.

the write operations in this test. The graph below shows the performance comparison of Message Queue throughput between the default and diskless Zookeeper. As shown in Figure 5.6, the diskless Zookeeper has a 32x speedup in throughput than the default Zookeeper. The program for the message queues test is listed in Appendix A.12 and the benchmark results are in Appendix A.8. The Figure 5.7, shows the performance comparison between other optimized message queues like Kafka, ActiveMQ with the message queue created using Zookeeper. As seen in the graph, the message queue using diskless Zookeeper clearly performs better than the others. The other message queues like Kafka and ActiveMQ has very high in-memory, asyn-

chronous mode performance compared to Zookeeper. But their persistent, highly available and synchronous mode performance is lesser than zookeeper.

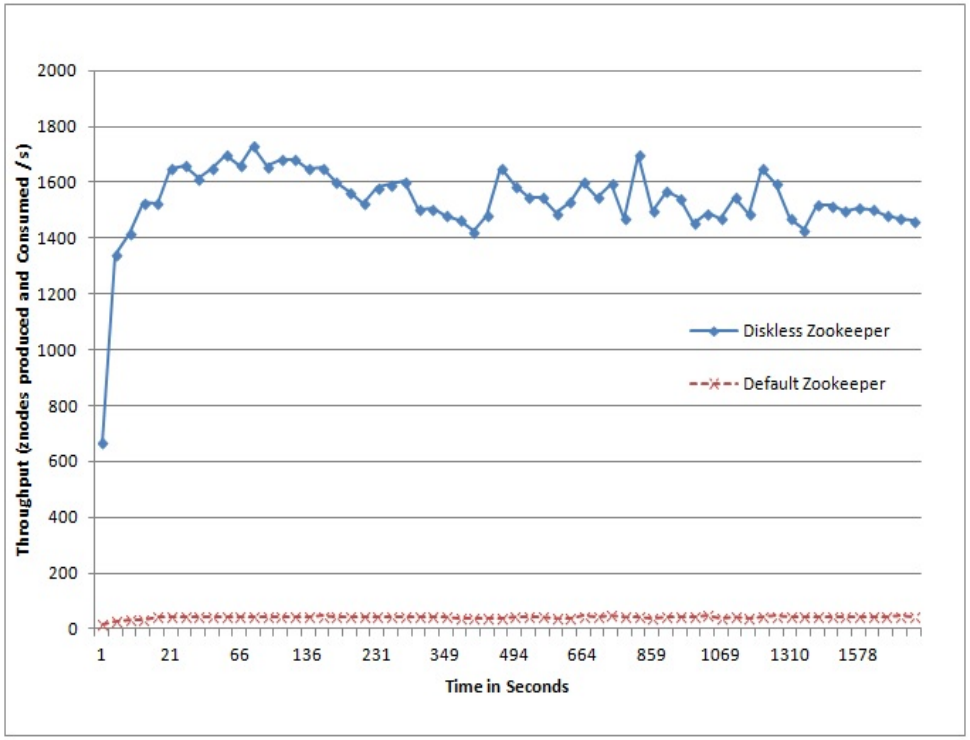

Figure 5.6: Zookeeper as a Message Queue benchmark. Throughput is a cumulative number of znodes produced and consumed per second. The throughput values are measured on a test over a duration of 30 minutes with a client producing at it's maximum synchronous speed.

5.3.2 Results and Inferences

The above section explains the various benchmarks to analyze the performance of Zookeeper. As proposed, the diskless Zookeeper achieves better

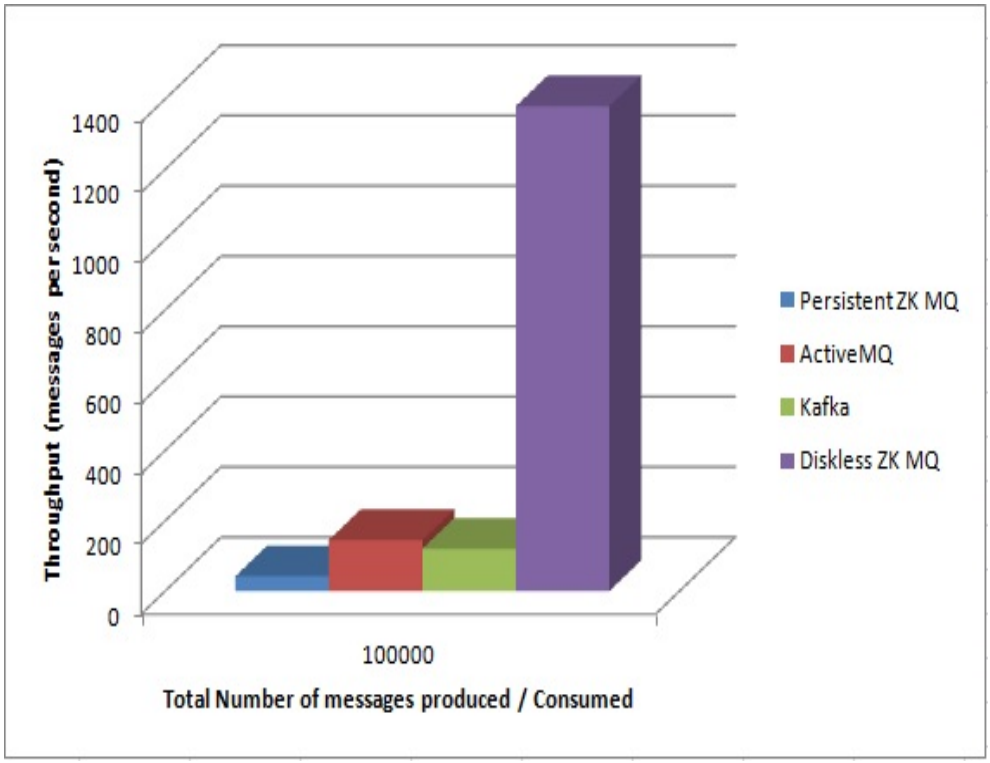

Figure 5.7: Performance comparison between Kafka, ActiveMQ and Zookeeper as message queue. Throughput is a cumulative number of znodes/messages produced and consumed per second. This throughput is measured on a test over a batch of 100000 producer/consumer job. The messages queues are configured with 3 nodes/brokers operating in persistent and synchronous mode.

performance than the default Zookeeper. We achieved a 30x write throughput with the same resource requirement and reliability guarantee. The performance of Message Queues also improved by approximately 32 times. This improvement is due to the new reduced disk dependent persistence layer of Zookeeper offered by the diskless Clustering design. These improvements in the performance came at a slight cost of durability. Here the cost of

durability does not denote the loss of data. It is the slight increase in the restoration time of Zookeeper ensemble during failure. The Benchmark results of the restoration time can be seen in Appendix A.7. The increase in the restoration time is because of the checkpoint activities during failure. As discussed in Subsection 3.4.2, during Quorum failure or Leader failure, the non-failed nodes backs up a complete snapshot of their state before starting again in recovery mode. The overhead in the restoration time is this time taken for the snapshot. As discussed in Subsection 5.3.1.2 the snapshot latency varies linearly with the size of Zookeeper ensemble. But according to the Zookeeper use-case, Zookeeper is not expected to store large data sets. So, the number of *znodes* in Zookeeper is expected to be low. The increase in the restoration time due to the complete snapshots at checkpoints is not a big overhead. At the highest scalable end of Zookeeper, the increase in time for the restoration of Zookeeper state with 1 million *znodes* is only 0.5 times higher than the restoration time in default Zookeeper. So, the overhead due to the checkpoint activities during failure is not a bottleneck compared to performance gains.

5.3.3 Performance Vs Durability Trade-off

Performance and durability have always been a opposing set of parameters right from the inception of database systems. The best example to illustrate this trade-off is the buffering of data when writing to disk. Disk buffering increases the performance of write throughput but failure of the system before

writing the data can cause the loss of all buffered data. Similar is the case of in-memory databases. The flexibilities introduced in persistence for improving performance is always at the cost of durability. One classic example is Berkeley DB. The Pure in-memory version of Berkeley DB has very good performance but at the cost of zero durability.

In our case of system design, we take advantage of the replicated nature of the in-memory database to convert the one copy serializability model into a diskless clustering model. By implementing this, we also achieved a very good improvement in the write performance. But this model of persistence can be applied only to replicated databases and not to the standalone systems. This is one of the prime design criteria of our system. Also, as discussed in Section 3.4.4, the level of durability provided by our system is *Durability at Failure*. The non-failed nodes during the failure of ensemble are used as a backup store to provide durability. This is achievable because one of our system assumptions states that at least one node remains non-failed in the replica at all times. Although, this system provides complete durability with low disk dependency, the major design decisions are built on top of the assumptions and guarantees of Zookeeper Atomic Broadcast protocol. This made the design of a high performance distributed coordination system with low disk dependency and high durability achievable.

Chapter 6

Conclusion and Future Work

This thesis presented the successful implementation of a low disk dependent persistence layer to a transaction replicated distributed coordination system called Apache Zookeeper. Detailed introduction to distributed system concepts, in-memory databases and the performance bottlenecks due to the one copy serializability model was explained in order to show the motivation behind this research. The research also analyzed and evaluated various previous design models to improve the performance of distributed coordination systems.

As the design is implemented on Apache Zookeeper, the thesis presented a very detailed description about Zookeeper and it's related protocols and algorithms. The Zookeeper Atomic Broadcast protocol and the Fuzzy Snapshot algorithm for database restoration are emphasized. Following this, the

bottleneck to the write requests due to sequential processing of transaction logs was discovered as the major problem. Also, the research shows the overhead of writing the transaction logs to disk.

In order to reduce the disk dependency, two design schemes namely *Diskless Clustering* and *Durability at Failure* were proposed. The major goal of this design is to provide durability from a replicated cluster rather than the local store. By the implementation of this design, it is shown that sequential writing of transaction logs to the disk can be avoided. As a part of this design, Busy Snapshot algorithm defined the modifications to the persistence layer in Zookeeper. Busy Snapshot algorithm uses Fuzzy snapshots from the local store and complete state from the replica to restore a node during recovery. The mechanism and checkpoints to convert the Fuzzy Snapshot into a complete snapshot is defined by the design model called *Durability at Failure*. *Durability at Failure* uses failure of an ensemble node as a checkpoint to convert the Fuzzy Snapshot into a complete snapshot in the non-failed nodes of the replica. This is built on top of the assumption that at least one node remains non-failed in the replica.

Chapter 4 presented the implementation details of Zookeeper. The data structures and programming details are analyzed with respect to the implementation of the design. Following this, in Chapter 5, evaluation of the research implementation is presented. Various benchmarks are used to mea-

sure the performance of the modified Zookeeper and it is compared with the default Zookeeper. It is shown that the modified Zookeeper achieves 30 times improvement in the write performance. The performance improvement came at a 0.5 times increase in the failure restoration time of Zookeeper. This trade-off between performance and durability is discussed in the Subsection 5.3.4.

As a result of this research, a successful implementation of a less disk bound durability mechanism to the in-memory database of Apache Zookeeper has been achieved. Although this system design provides a good durability, the major design decisions are designed on top of the assumptions and model of Apache Zookeeper. They are (1) The in-memory database of Zookeeper is designed to store data sets of small size. (2) The transaction replication model and crash fail state recovery scheme combined with the assumption that atleast one of the nodes in the replica remain non-failed served as a foundation on which the current system is designed.

Further development in this research would be to extend this model to the in-memory databases that can store large data-sets backed up by a disk. This would involve backing up of data to the disk without affecting the request processing by defining granular checkpoint schemes. Also, a diskless clustering mechanism to restore large data-sets from the replica has to be defined. This research involves optimizing the data restoration mechanism

and latency of data recovery from the cluster for large data sets. In the complete snapshot taken during the checkpoint at failures, the entire state of Zookeeper is backed up as a new complete snapshot. This could be optimized to store only the missing states from the last recent snapshot to disk, such that the complete state could be formed. This can reduce the amount of data to be backed up on disk during failures which could improve the restoration time of the ensemble.

Bibliography

[1] Coulouris, G. F. (2009). *Distributed Systems: Concepts and Design, 4/e.* Pearson Education India.

[2] Deutsch, P. (1992). The eight fallacies of distributed computing. URL: http://today. java. net/jag/Fallacies. html., Access Date: 7/2/2013.

[3] *Disk Vs Memory comparison*, URL:`http://queue.acm.org/detail.cfm?id=1563874`, Access Date: 9/5/2013.

[4] Garcia-Molina, H., & Salem, K. (1992). *Main memory database systems: An overview.* Knowledge and Data Engineering, IEEE Transactions on, 4(6), 509-516.

[5] Seeger, M., & Ultra-Large-Sites, S. (2009). *Key-Value stores: a practical overview.* Computer Science and Media.

[6] Stonebraker, M. (2010). *Errors in Database Systems, Eventual Consistency, and the CAP Theorem.* Communications of the ACM, BLOG@ ACM.

[7] Brewer, E. A. (2000, July). *Towards robust distributed systems.* In PODC (p. 7).

[8] Lakshman, A., & Malik, P. (2010). *Cassandra: a decentralized structured storage system.* ACM SIGOPS Operating Systems Review, 44(2), 35-40.

[9] *Basho Riak*,URL:`http://basho.com/riak/`, Access Date: 5/10/2013.

[10] George, L. (2011). *HBase: the definitive guide.* O'Reilly Media, Inc..

[11] Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010, June). *ZooKeeper: wait-free coordination for internet-scale systems.* In Proceedings of the 2010 USENIX conference on USENIX annual technical conference (Vol. 8, pp. 11-11).

[12] *Accord*, http://www.osrg.net/accord/, Access Date: 5/10/2013.

[13] Tanenbaum, A. S., & Van Steen, M. (2002). *Distributed systems (Vol. 2).* Prentice Hall.

[14] Huc, C., Levoir, T., & Nonon-Latapie, M. (1997). *Metadata: models and conceptual limits.* In IEEE Computer Society metadata conference (pp. 1-12).

[15] San Andres, R. J., Choquier, P., Greenberg, R. G., & Peyroux, J. F. (1999). *U.S. Patent No. 5,956,489.* Washington, DC: U.S. Patent and Trademark Office.

[16] Bornea, M. A., Hodson, O., Elnikety, S., & Fekete, A. (2011, April). *One-copy serializability with snapshot isolation under the hood.* In Data Engineering (ICDE), 2011 IEEE 27th International Conference on (pp. 625-636). IEEE.

[17] Ramez, E. (1994). *Fundamentals of database systems.* Pearson Education India.

[18] Hopcroft, J. E. (2008). *Introduction to Automata Theory, Languages, and Computation, 3/E.* Pearson Education India.

[19] KNEzEVIC, N. (2012). *A High-Throughput Byzantine Fault-Tolerant Protocol.* Federal Polytechnic Of Lausanne.

[20] Castro, M., & Liskov, B. (1999, February). *Practical Byzantine fault tolerance.* In OSDI (Vol. 99, pp. 173-186).

[21] Birman, K., & Joseph, T. (1987). *Exploiting virtual synchrony in distributed systems.* (Vol. 21, No. 5, pp. 123-138). ACM.

[22] Schneider, F. B. (1993). *Replication management using the state-machine approach, Distributed systems.*

[23] Skeen, D., & Stonebraker, M. (1983). *A formal model of crash recovery in a distributed system.* Software Engineering, IEEE Transactions on, (3), 219-228.

[24] Budhiraja, N., Marzullo, K., Schneider, F. B., & Toueg, S. (1993). *The primary-backup approach.* Distributed systems, 2, 199-216.

[25] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., & Toueg, S. (2006, June). *Consensus with byzantine failures and little system synchrony.* In Dependable Systems and Networks, 2006. DSN 2006. International Conference on (pp. 147-155). IEEE.

[26] Hadoop, A. (2009). *Hadoop.* 2009-03-06]. http://hadoop. apache. org.

[27] Reed, B., & Junqueira, F. P. (2008, September). *A simple totally ordered broadcast protocol.* In proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (p. 2). ACM.

[28] Kobashi, H., Yamane, Y., Murata, M., Saeki, T., Moue, H., & Tsuchimoto, Y. (2010). *Eventually Consistent Transaction.* In Proceedings of the 22nd IASTED International Conference on Parallel and Distributed Computing and Systems, p103-109.

[29] Reed, B. C., & Bohannon, P. (2010). *U.S. Patent No. 7,725,440.* Washington, DC: U.S. Patent and Trademark Office.

[30] Tarjan, R. (1972). *Depth-first search and linear graph algorithms.* SIAM journal on computing, 1(2), 146-160.

[31] Horowitz, E. (1999). *Fundamentals of computer algorithms.* Galgotia Publications.

[32] Burrows, M. (2006, November). *The Chubby lock service for loosely-coupled distributed systems.* In Proceedings of the 7th symposium on Operating systems design and implementation (pp. 335-350). USENIX Association.

[33] Lamport, L. (2001). *Paxos made simple.* ACM Sigact News, 32(4), 18-25.

[34] Birman, K. P., Joseph, T. A., Raeuchle, T., & El Abbadi, A. (1985). *Implementing fault-tolerant distributed objects.* Software Engineering, IEEE Transactions on, (6), 502-508.

[35] Olson, M. A., Bostic, K., & Seltzer, M. I. (1999, June). *Berkeley DB.* In USENIX Annual Technical Conference, FREENIX Track (pp. 183-191).

[36] *Writing Applications using Berkeley DB*,URL:`http://docs.oracle.com/cd/E17076_02/html/articles/inmemory/C/index.html`, Access Date: 5/10/2013.

[37] Dake, S. C., Caulfield, C., & Beekhof, A. (2008, July). *The corosync cluster engine.* In Linux Symposium (Vol. 85).

[38] *Server Socket Channel*, URL:`http://docs.oracle.com/javase/6/docs/api/java/nio/channels/ServerSocketChannel.html`, Access Date: 5/14/2013.

[39] *Linked Blocking Queue*, URL:http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/LinkedBlockingQueue.html, Access Date: 5/14/2013.

[40] *Concurrent Linked Queue*, URL:http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html, Access Date: 5/14/2013.

[41] *Concurrent Hash Map*, URL:http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html, Access Date: 5/14/2103.

[42] *Server Socket Vs Server Socket Channel in Java*, URL:http://www2.sys-con.com/itsg/virtualcd/java/archives/0705/schreiber/index.html, Access Date: 5/16/2013.

[43] *PrepRequestProcessor*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/PrepRequestProcessor.html, Access Date: 5/23/2013.

[44] *SyncRequestProcessor*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/SyncRequestProcessor.html, Access Date: 5/23/2013.

[45] *FinalRequestProcessor*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/FinalRequestProcessor.html, Access Date: 5/23/2013.

[46] *FollowerRequestProcessor*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/quorum/FollowerRequestProcessor.html, Access Date: 5/23/2013.

[47] *Leader*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/quorum/Leader.html, Access Date: 5/23/2013.

[48] *Follower*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/quorum/Follower.html, Access Date: 5/23/2013.

[49] *Learner*, URL:http://people.apache.org/~larsgeorge/zookeeper-1215258/build/docs/dev-api/org/apache/zookeeper/server/quorum/Learner.html, Access Date: 5/23/2013.

[50] *Zookeeper Commands: Four Letter Words*, Access Date: 5/28/2013.

[51] *ZK smoke test*, URL:https://github.com/phunt/zk-smoketest, Access Date: 5/31/2013.

[52] *Visual VM*, URL:http://visualvm.java.net/, Access Date: 5/31/2013.

[53] *Apache Zookeeper in Twitter*, URL:http://highscalability.com/zookeeper-reliable-scalable-distributed-coordination-system, Access Date: 5/31/2013.

[54] *Apache Zookeeper in Netflix*, URL:http://techblog.netflix.com/2011/11/introducing-curator-netflix-zookeeper.html, Access Date: 5/31/2013.

[55] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., ... & Aiyer, A. (2011, June). *Apache Hadoop goes realtime at Facebook.* In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (pp. 1071-1080). ACM.

[56] Fan, W., & Bifet, A. (2013). *Mining big data: current status, and forecast to the future.* ACM SIGKDD Explorations Newsletter, 14(2), 1-5.

Appendix A

A.1 System Specifications

Processor	
CPU	3rd Gen Core i7, Intel 3 i7-3520M / 2.9 GHz
Max Turbo Speed	3.6 GHz
Number of Cores	Quad-Core
Hyperthreading	Yes, upto 8 threads.
Cache	4 MB
64-bit Computing	Yes
Chipset	Mobile Intel QM77 Express
Memory	
RAM	32 GB (4 x 8 GB)

Technology	DDR3 SDRAM
Speed	1600 MHz / PC3-12800
Form Factor	SO DIMM 204-pin
Storage	
Hard Drive	256 GB Samsung 830 Series SSD / Write(320 MB/s), Sequential Reads(520 MB/s)
Interface	Serial ATA-300

Table A.1: System Specification for Benchmarks

A.2 Default Zookeeper Configuration

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the log and snapshot is stored.
dataLogDir=/var/logs/zookeeper-3.4.5
dataDir=/var/zookeeper-3.4.5
```

```
# the port at which the clients will connect
clientPort=2186
# Don't Buffer while writing, Sync Data to Disk
forceSync=yes
# Quorum Port Information
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

A.3 Diskless Zookeeper Configuration

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
dataDir=/var/dzookeeper-3.4.5
# the port at which the clients will connect
```

clientPort=2183

Obsolete Snapshot Purge Interval

PurgeTime=600000

Number of Snapshots to Retain

NumberToRetain=3

Quorum Port Information

server.1=localhost:2788:3788

server.2=localhost:2789:3789

server.3=localhost:2790:3790

A.4 Zookeeper Create operation performance Benchmark

	Throughput(Number of creates per second)	
Time (s)	Diskless Zookeeper	Default Zookeeper
1	369	44
3	1000	75
6	2646	104
10	3523	101
15	3409	100
21	3317	107

28	3856	115
36	4233	141
45	3944	147
55	3485	148
66	3892	137
78	4013	145
91	3788	139
105	3822	142
120	3620	143
136	3738	140
153	3451	138
171	3531	144
190	3697	141
210	3856	146
231	3904	137
253	3590	136
276	3648	142
298	3803	146
323	3686	145
349	3705	144
376	3602	147

404	3627	139
433	3839	138
463	3839	140
494	3613	143
526	4079	141
559	4109	145
593	4050	142
628	4072	139
664	3647	143
701	3962	146
739	4011	139
778	4085	140
818	3598	143
859	4006	140
901	3716	142
944	4004	145
988	3398	139
1023	3840	146
1069	3969	142
1113	3788	139
1161	3737	140

1210	4008	142
1260	3631	143
1310	3709	139
1362	3974	142
1413	3542	143
1467	3901	139
1522	3909	141
1578	3679	146
1635	3804	138
1693	3520	142
1752	3749	141
1830	3954	145

Table A.2: Zookeeper *Create* operation performance Benchmark

A.5 Zookeeper Get operation performance Benchmark

	Throughput (Number of Get per second)	
Time(s)	Diskless Zookeeper	Default Zookeeper
1	2443	1000
3	5765	3456

6	7263	3899
10	7516	4899
15	7494	5827
21	7751	6789
28	7771	6974
36	7658	6458
45	7554	7542
55	8069	7542
66	7649	7923
78	7901	7856
91	8452	8023
105	7958	7745
120	8080	7968
136	7738	7568
153	7814	7614
171	8008	7999
190	7649	7566
210	7651	7775
231	7775	7635
253	7725	7645
276	7792	7854

298	8238	8246
323	7721	8016
349	7869	7956
376	7859	7764
404	7677	7943
433	7788	7841
463	7864	7516
494	7820	7421
526	7634	7546
559	7762	7613
593	7798	7645
628	7770	7891
664	8149	8002
701	7685	7745
739	7692	7456
778	7879	7701
818	7798	7654
859	7930	7469
901	7873	7218
944	7703	7896
988	7846	7562

1023	7834	7469
1069	7787	7369
1113	7744	7589
1161	7792	7569
1210	7790	7645
1260	8167	7436
1310	8236	7548
1362	8420	7689
1413	8292	7921
1467	8178	7415
1522	7967	7869
1578	8275	7789
1635	8134	7769
1693	8536	7869
1752	7844	7569
1830	8666	7699

Table A.3: Zookeeper *Get* operation performance Benchmark

A.6 Zookeeper Snapshot Latency Benchmarks

Number of Znodes in Zookeeper State	Snapshot time(ms)

1	1
3	1
10	2
31	3
100	3
316	3
1000	7
3162	13
10000	23
31622	68
100000	198
316227	737
1000000	2067

Table A.4: Zookeeper Snapshot Latency Benchmark

A.7 Zookeeper CPU Usage Benchmark

	CPU Usage percent	
Time(m)	Diskless Zookeeper	Default Zookeeper
1	15.3	8.7
2	14.3	15.1
3	14.1	13.3

4	13.9	13.3
5	13.8	13
6	13.3	14.2
7	13.4	14.1
8	14.3	14.4
9	13.6	17.9
10	14.4	14.2
11	7.3	13.4
12	14.2	12.6
13	14.1	13
14	13.3	10.1
15	17.5	10.3
16	17.3	10.5
17	16.8	11.2
18	15.3	14
19	15.4	14.2
20	14.8	12.9
21	16.4	14.2
22	15.8	14.2
23	16.1	14.4
24	16.5	14.4

25	13.1	14.6
26	16	11.2
27	15.5	5.3
28	15.7	11.5
29	13.4	11.9
30	14.3	14.9
31	16.1	15.3
32	14.3	14.9
33	14.2	15.8
34	12.4	15.5
35	11.4	15.1
36	12.4	14.7
37	12.5	15.2
38	13.3	13.5
39	13.8	12.3
40	12	14.1
41	12.4	10.2
42	12.7	12.3
43	12.4	13.5
44	9.4	14.1
45	14.7	13.6

46	14.4	14.4
47	14.7	14.2
48	14.1	14.4
49	13.4	14.4
50	13.5	14.3
51	12.2	14.3
52	12.7	11.8
53	14.8	14.6
54	13.2	13.7
55	13.7	14.2
56	11.8	13.4
57	10.1	14.3
58	13.9	14
59	13.4	14.2
60	13.6	14.4
61	13.9	14.6
62	13.8	14.3
Average	13.87	13.57

Table A.5: Zookeeper CPU Usage Benchmark

A.8 Zookeeper Message Queue Benchmark Result

	Throughput(Operations per second)	
Time(s)	Diskless Zookeeper	Default Zookeeper
1	667	14
3	1339	25
6	1416	30
10	1527	35
15	1525	41
21	1650	42
28	1662	42
36	1615	43
45	1650	44
55	1699	45
66	1660	45
78	1729	44
91	1658	43
105	1682	42
120	1683	44
136	1648	45
153	1653	46

171	1600	42
190	1566	43
210	1523	44
231	1581	43
253	1592	45
276	1604	44
298	1503	42
323	1506	43
349	1480	41
376	1466	40
404	1425	39
433	1480	38
463	1653	40
494	1587	43
526	1548	42
559	1546	41
593	1489	40
628	1530	39
664	1601	46
701	1547	45
739	1594	47

778	1469	42
818	1698	41
859	1497	40
901	1568	43
944	1543	45
988	1456	41
1023	1489	47
1069	1469	39
1113	1546	41
1161	1487	40
1210	1648	45
1260	1594	46
1310	1469	42
1362	1430	41
1413	1520	43
1467	1516	45
1522	1498	44
1578	1510	42
1635	1502	44
1693	1480	41
1752	1469	46

1830	1463	41

Table A.6: Zookeeper as Message Queue Benchmark

A.9 Zookeeper operations evaluation test code template

```
public class ZKPerformanceTimer implements Watcher,
    Runnable {
     ZooKeeper zk;
     String Node = "/" + "event_p1";
     String znode;
     long time = 0;
     int failedNodes = 0;
     public static String data = null;
     {
          StringBuilder dataBuilder = new StringBuilder();
          for (int i = 0; i <= 280; i++) {
               dataBuilder.append("x");
          }
          data = dataBuilder.toString();
     }
     public void connect(String Server) throws Exception {
          zk = new ZooKeeper(Server, 20000, this);
     }
     public void run() {
```

```
    while (true) {
        try {
            List<String> list = zk.getChildren("/",
               this);
            while (list != null) {
                while (list.size() != 0) {
                    System.out.println("deleting");
                    zk.delete("/" + list.get(0), -1);
                    list.remove(0);
                }
            }
        } catch (KeeperException ex) {
            System.exit(0);
            Logger.getLogger(ZKRestorePerfTest.class.
               getName()).log(Level.SEVERE, null, ex);

        } catch (InterruptedException ex) {
            System.exit(0);
            Logger.getLogger(ZKRestorePerfTest.class.
               getName()).log(Level.SEVERE, null, ex);
        }
    }
}
public void createandset() {
    try {
        znode = zk.create(Node, data.getBytes(), Ids.
           OPEN_ACL_UNSAFE,CreateMode.
```

```
            EPHEMERAL_SEQUENTIAL);
    } catch (Exception e) {
        System.out.println("exception: " + e);
        failedNodes++;
        long start = System.currentTimeMillis();
        try {
            Thread.sleep(2000);
            System.out.println("time to restore-top:" +
                time);
            zk = new ZooKeeper("
               127.0.0.1:2186,127.0.0.1:2187,127.0.0.1:2188
               ", 20000, this);
            System.out.println("time to restore:" +
               time);
            time = 0;
        } catch (Exception ex) {
        }
    }
}

public void close() throws Exception {
    zk.close();
}
public static void main(String[] args) throws Exception
    {
    long exponentBase = 10, i = 0, k, start = 0;
    long end = 0, start1 = 0, end1 = 0, difference = 0;
```

```
long interim, LoopDelimiter = 0, difference1 = 0;
double diffInSeconds = 0.0, j = 0.0;
double diffInSeconds1 = 0;
LoopDelimiter = 1000;
FileWriter fstream;
int option;
String LogLocation, LogLocConfig;
String SnapCount = "", Machine;
String ServerVersion, NOI = "1";
Scanner in = new Scanner(System.in);
System.out.println("Enter the following details:");
System.out.println("Log Location:\n1.Disk\n2.AWS
   EBS\n3.RamDisk\n4.None");
option = in.nextInt();
String Server = args[0];
LoopDelimiter = Integer.parseInt(args[1]);
LogLocation = args[2];
SnapCount = args[3];
Machine = args[4];
ServerVersion = args[5];
NOI = args[6];
String forceSync = args[7];
fstream = new FileWriter("PerformanceTestResults.
   csv");
BufferedWriter csv = new BufferedWriter(fstream);
ZKPerformanceTimer test = new ZKPerformanceTimer();
csv.append("Test Name,Transaction Log Location,");
```

```
csv.append("SnapCount Value,Machine,");
csv.append("Server Version,");
csv.append("Operation,");
csv.append("Number Of ZK Instances in Ensemble,");
csv.append(Execution Time(s),");
csv.append("Number of Successful Requests,");
csv.append("Sync Rate(Znodes/s),");
csv.append("Async Rate(Znodes/s");
csv.append("\n");
test.connect(Server);
test.createandset();
for (k = 0; k < LoopDelimiter; k++) {
    i = 0;
    start = System.currentTimeMillis();
    while (((System.currentTimeMillis() - start) <
       (k * 1000))) {
        i++;
        test.createandset();
    }
    end = System.currentTimeMillis();
    difference = k * 1000;
    diffInSeconds = ((double) difference) / 1000;

    System.out.println("i:" + i);
    System.out.println("Actual difference in Time:
       " + (end - start));
```

```
            System.out.print(k + "\t\t" + (diffInSeconds) +
                "\t\t");
            System.out.print(Math.ceil((double) i / (
               diffInSeconds)));
            csv.append("ZK Performance Test-Write,");
            csv.append(LogLocation + ",");
            csv.append(SnapCount + ",");
            csv.append(Machine + ",");
            csv.append(ServerVersion);
            csv.append(",Create,");
            csv.append(NOI + ",");
            csv.append(diffInSeconds + ",");
            csv.append(i + ",");
            csv.append(Math.ceil((double) i / (
               diffInSeconds)) + ",");
            csv.append(Math.ceil((double) i / (
               diffInSeconds)));
            csv.append("\n");
          }
        csv.close();
        fstream.close();
        System.out.println(test.failedNodes);
        System.out.println("Performance Test Successful");
    }

    public void process(WatchedEvent event) {
        if (event.equals("SyncConnected")) {
```

```
            System.out.println("Connected");
        }
    }
}
```

A.10 Zookeeper Message Queue Benchmark Test Code

```
public class MQTimer implements Watcher {

    public static ZooKeeper zk;
    public ZooKeeper zk1;
    static Integer mutex;
    String root;

    MQTimer(String address) {
        try {
            System.out.println("Starting ZK:");
            zk = new ZooKeeper(address, 3000, this);
            zk1 = new ZooKeeper(address, 3000, this);
            mutex = new Integer(-1);
            System.out.println("Finished starting ZK: " +
                zk);
        } catch (IOException e) {
            System.out.println(e.toString());

        }
```

```
    //else mutex = new Integer(-1);
}

synchronized public void process(WatchedEvent event) {
    synchronized (mutex) {
        //System.out.println("Process: " + event.
            getType());
        mutex.notify();
    }
}

static public class Queue extends SyncPrimitive {

    static List<String> list1;

    Queue(String address, String name) {
        super(address);
        this.root = name;
        // Create ZK node name
        if (zk != null) {
            try {
                Stat s = zk.exists(root, false);
                if (s == null) {
                    zk.create(root, new byte[1500], Ids
                        .OPEN_ACL_UNSAFE,
                            CreateMode.PERSISTENT);
```

```
            }
        } catch (KeeperException e) {
            System.out.println("Keeper exception
               when instantiating queue: "
                    + e.toString());
        } catch (InterruptedException e) {
            System.out.println("Interrupted
               exception");
        }
    }
}
boolean produce(int i, String data) throws
   KeeperException, InterruptedException {
    ByteBuffer b = ByteBuffer.allocate(4);
    byte[] value;
    b.putInt(i);
    value = b.array();
    String znode = zk.create(root + "/element",
       data.getBytes(), Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT_SEQUENTIAL);
    return true;
}

int consume(List<String> Children) throws
   KeeperException, InterruptedException {
    int retvalue = -1;
    Stat stat = null;
```

```
            String element = "";
            // Get the first element available
            while (true) {
                synchronized (mutex) {
                    List<String> list = Children;
                    if (list.size() == 0) {
                        System.out.println("Going to wait")
                            ;
                        Thread.sleep(2000);
                    } else {

                        element = list.get(0);
                        list.remove(element);
                        byte[] b = zk.getData(root + "/" +
                            element,
                                false, stat);
                        zk.delete(root + "/" + element, -1)
                            ;
                        ByteBuffer buffer = ByteBuffer.wrap
                            (b);

                        return 1;
                    }
                }
            }
        }
    }
```

```
public static void main(String args[]) throws Exception
    {
    MQTimer MQT = new MQTimer(args[0]);
    queueTest(MQT, args);

}

public static void queueTest(MQTimer MQ, String args[])
    throws Exception {
    final Queue q = new Queue(args[0], "/app1");
    String data = null;
    {
        StringBuilder dataBuilder = new StringBuilder()
            ;
        for (int m = 0; m <= 10000; m++) {
            dataBuilder.append("x");
        }
        data = dataBuilder.toString();
    }
    System.out.println("Input: " + "127.0.0.1:2188");
    long k, i = 0;
    String LogLocation, LogLocConfig, SnapCount = "";
    String Machine, ServerVersion, NOI = "1", forceSync
        = "no";
    Integer max = new Integer("30");
    long start = 0, end = 0, diff = 0;
```

```
int m = 0;
double diffInSeconds = 0.0, j = 0.0;
FileWriter fstream;
long LoopDelimiter = max, exponentBase = 10;
LoopDelimiter = Integer.parseInt(args[1]);
LogLocation = args[2];
SnapCount = args[3];
Machine = args[4];
ServerVersion = args[5];
NOI = args[6];
forceSync = args[7];
fstream = new FileWriter("PerformanceTestResults-MQ
   .csv");
BufferedWriter csv = new BufferedWriter(fstream);
csv.append("Test Name,");
csv.append("Transaction Log Location,");
csv.append("SnapCount Value,");
csv.append("Machine,Server Version,");
csv.append("forceSync,");
csv.append("Operation,");
csv.append("Number Of ZK Instances in Ensemble,");
csv.append("Number of Znodes Request Per Client,");
csv.append("Number Of Clients,");
csv.append("TotalNumber of Nodes Request,");
csv.append("Sync Execution Time(s),");
csv.append("Sync Rate(Znodes/s),");
csv.append("Async Execution Time(s),");
```

```
csv.append("Async Rate(Znodes/s)");
csv.append("\n");
System.out.println("Message Passing queue using
   zookeeper");
System.out.println("Number of Nodes---Time in
   Seconds---Through(ZKnodes/s)");
final int k1 = 0;
Thread th = new Thread(new Runnable() {

    @Override
    public void run() {
        while (true) {
            try {
                q.produce(k1, "aaaaaaaaaaaaaaaaa")
                    ;
                //  System.out.println("Producing")
                    ;
            } catch (KeeperException ex) {
                Logger.getLogger(MQTimer.class.
                    getName()).log(Level.SEVERE,
                    null, ex);
            } catch (InterruptedException ex) {
                Logger.getLogger(MQTimer.class.
                    getName()).log(Level.SEVERE,
                    null, ex);
            }
        }
```

```
    }
});
th.start();
Thread.sleep(2000);

System.out.println(MQ.zk1.getChildren("/app1",
   false).toString());
for (i = 1; i < LoopDelimiter; i++) {
    start = System.currentTimeMillis();
    m = 0;
    while (((System.currentTimeMillis() - start) <
       (i * 1000))) {

        if (m == 0) {
            Queue.list1 = MQ.zk1.getChildren("/app1
               ", false);
            Collections.sort(Queue.list1);
        }
        q.consume(Queue.list1);
        m++;
    }
    end = System.currentTimeMillis();
    diff = end - start;
    diffInSeconds = ((double) diff) / 1000;
    System.out.println("Time difference" + diff);
    csv.append("ZK Performance Test-MQ,");
    csv.append(LogLocation + ",");
```

```
            csv.append(SnapCount + ",");
            csv.append(Machine + ",");
            csv.append(ServerVersion + ",");
            csv.append(forceSync);
            csv.append(",Create+GetChildren+Get+Delete,");
            csv.append(NOI + "," + i + ",1,");
            csv.append(m + "," + (diffInSeconds) + ",");
            csv.append(Math.ceil((double) m / (
               diffInSeconds)));

            csv.append("\n");
            System.out.print(i + "\t\t" + diffInSeconds);
            System.out.print("\t\t" + Math.ceil((double) m
               / (diffInSeconds)));
            j++;
        }
        th.stop();
        csv.close();
    }
}
```

Vita

Candidate's full name: Dayal Dilli

University attended:
Master of Computer Science, 2013
University of New Brunswick,
Fredericton, Canada.

Bachelor of Engineering, Computer Science 2012,
BSA Crescent Engineering College,
Anna University,
Chennai, India.