Characterizing and Improving the General Performance of Apache Zookeeper

by

Chandan Bagai

Bachelor of Engineering(Computer Science and Engineering)
Chitkara University, 2008

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

Masters of Computer Science

In the Graduate Academic Unit of your GAU

This report is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May,2014

Abstract

Coordination has become one of the most vital requirements in a scenario where a group of systems are communicating with each other over a network trying to accomplish goals. In some scenarios for instance, some high end distributed applications also require sophisticated coordination primitives such as leader election, rather than just agreeing over certain parameters. Implementing these types of primitives is rather difficult, due to their complexity, and their vulnerability to errors may lead to application failure. ZooKeeper, an open source distributed coordination service maintained by Apache takes care of all such issues. Its wait-free nature and strong ordering ensures synchronization and makes it ideal for developing high performance applications. ZooKeeper is inspired by other services such as *Chubby*[1], and at the protocol level from *Paxos*[2]. The main goal of this report is to enhance the performance of ZooKeeper for ensuring its normal functioning. The performance of ZooKeeper has been increased, and none of the alterations have been done at the protocol level. The implementation involves modifications to the marshalling and demarshalling mechanisms followed by request processors and use of some queue implementations. The resulting performance impacts have been evaluated as well.

Acknowledgements

I am glad to thank those who made this report possible. Therefore I would like to especially thank my family, who supported me not only while writing this report. In particular I would like to thank my advisor Dr. Kenneth B. Kent from the University of New Brunswick. His strong motivation and guidance helped me to finish my report. I am obliged and feel honored to accomplish my research under him. Furthermore, I would also like to thank my colleague Marcel Dombrowski. Without their help, writing this report would have not been possible.

Table of Contents

List of Figures

List of Symbols, Nomenclature or Abbreviations

IP :	Internet Protocol
ZAB :	ZooKeeper Atomic Broadcast Protocol
SQS :	Simple Queue Service
LBQ :	Linked Blocking Queue
CLQ :	Concurrent Linked Queue
HDD:	Hard Disk Drive
RAM:	Random Access Memory
ACL :	Access Control List
API :	Application Programming Interface
JVM :	Java Virtual Machine
CSV :	Comma Seperated Values

Chapter 1

Introduction and Motivation

1.1 Introduction to the Project

Apache ZooKeeper [10] is an open source, fault tolerant distributed coordination service, in which huge amounts of continuously changing data are gathered and monitored. These can be for instance data produced by social networks. It is currently maintained by Yahoo and the Apache Software Foundation. ZooKeeper is used for cloud computing in order to provide essential synchronization and group services in a distributed environment.

This project focuses on characterizing and improving the general performance of Apache ZooKeeper. This was one of the major parts of the main project called Diskless Data Analytics which deals with Big Data for analyzing business trends. Furthermore, disk dependency was the bottleneck for performance. But, the shortcoming of this approach is that data has to ”expire” at some point to provide a bound on the quantity of data. In addition, there needs to be a collaborative sharing of data between nodes in order to provide assurances that data does not become corrupted due to hardware failures or an overabundance of data transactions.

1.2 Motivation

There are lots of other services such as Chubby [1] etc. that provide coordination in different forms as discussed in Section 2.1. ZooKeeper maintains an in memory database and provides coordination services by encapsulating distributing coordination algorithms such as the ZooKeeper Atomic Broadcast (ZAB) protocol [5] discussed in Section 2.4. It is a highly reliable service and clients use it for bootstrapping, storing configuration data, managing failure recovery, etc. It achieves its reliability and availability through replication [5]. For any distributed application the fundamental requirement is that the client is serviced in order and without any waiting time.

ZooKeeper not only possesses a wait free property, but also guarantees FIFO client ordering and linearizable writes [5]. ZooKeeper achieves FIFO ordering by using a pipelined architecture allowing thousands of outstanding requests while achieving a low latency and high service rate. Linearizable writes mean updating the in memory database of ZooKeeper or applying the updates in order.

1.3 Objectives

ZooKeeper provides a client library through which clients connect and submit their requests to the server. The server on the other hand intercepts the request, processes it, and responds back to the client. Connection and session management are also carried out by this library. This library has a major role in this project as all the tests that are used to measure performance use this library.

The objectives for this project are stated below:

- Determine the factors impacting performance.
- Assess ZooKeeper performance through different tests.
- Analysis of the results before making any modification to the code.
- Plot performance matrices on the basis of above results.
- Modify code in order to increase performance.
- Assess ZooKeeper performance again via same tests as above.
- Analyze results after modification to the code.
- Plot performance matrices on the basis of new results obtained above.

- Show the contrast in performance between the two performance matrices, before and after the modifications to the code have been made.
- Benchmark results and infer conclusions.

This report is structured as follows: in Chapter 2, the background to understand the remainder of this report is given. This includes information about Apache ZooKeeper, its components, working mechanisms, and related work. Furthermore, this chapter also provides an introduction to the concepts being used such as queues and serialization.

The design of the project is explained in Chapter 3. It elaborates on the project requirements, configurations being used, and the approach to accomplish these project goals. Chapter 4 starts off with the code review of ZooKeeper providing insights as to how a request is submitted and processed, and how the client is serviced. Furthermore, different performance aspects of ZooKeeper will be introduced and it will be discussed how they have been implemented in order to enhance performance.

Chapter 5 covers the testing and evaluation part of ZooKeeper including the algorithms of the test suite which measures its performance. The analysis of the results and graphs generated from the above test suite will be used for benchmarking and drawing conclusions. Lastly, Chapter 6 will focus on the summarizing report and describing future work.

Chapter 2

Background and Related Work

Most distributed applications [6][7] require co-ordination if being implemented on a large scale in order to accomplish its goal of synchronization, concurrency, etc. It can be in different forms. For instance, configuration is the simplest form of coordination and involves setting up operational parameters (which can be static or dynamic) over which computers or systems communicate in a distributed environment. Locks are also used in these scenarios to implement coordination primitives where mutual exclusion is required for using critical resources. If the application only requires coordination then it all depends on the type of application.

2.1 Queues

This section will emphasize on queues and their different implementations. Queues are a type of collection that is primarily used for storing elements before processing them. Almost all queues function in a FIFO manner but priority queues are an exception to this. This section will cover the type of queues used in high performance applications requiring concurrent access along with their respective methods. Generally, the queues used for these types of applications function on algorithms [8] that can be subdivided into two types of categories shown below:

- Blocking algorithms: In this type of algorithms, slow processes prevent faster processes from completing their operations on shared objects.
- Non-blocking algorithms: They typically are in the form of compare and swap algorithms that are wait-free and do not encounter starvation. It ensures that all the processes complete their computation in a finite number of steps ensuring synchronization.

2.1.1 Linked Blocking Queues

Blocking queues are the parent implementation of linked blocking queues [9]. They do not accept null values as mentioned in the official documentation [9]. Linked blocking queues are thread-safe optionally-bounded blocking queues used by distributed applications. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.

2.1.2 Concurrent Linked Queues

Concurrent Linked Queues [10] are found in the `java.util.concurrent` package. They are unbounded thread-safe queues used by applications where meeting concurrency is a vital goal. It is based on a compare and swap algorithm as discussed above. They also do not permit the presence of any null elements and its `size()` method is not a constant-time operation.

Section 4.3 demonstrates the performance aspects [11] of linked blocking queues and concurrent linked queues, and also bestows their usage with ZooKeeper.

2.2 Distributed Coordination Services

Coordination services have become a vital part of computing. Especially, in distributed applications [12] where a program is executed on multiple machines within a network. These type of applications require synchronization and in order to achieve their goals, the network computers communicate and coordinate their actions by passing messages. For managing the sending and reception of messages between different computers or machines, distribution coordination services come into play. There are several distributed coordination services such as Chubby [1], Amazon SQS [3], and Apache ZooKeeper [13].

2.2.1 Chubby

Chubby [1] is a locking service with strong synchronization guarantees. It uses lock methods such as open and close. It has a file-system-like interface, and uses an agreement protocol to guarantee the consistency of the replicas. Chubby does not allow clients to connect to any of the servers in the ensemble except for the leader. Ensemble refers to the group of servers forming the service. Chubby has less relaxed and a more complex consistency model than any other coordination services.

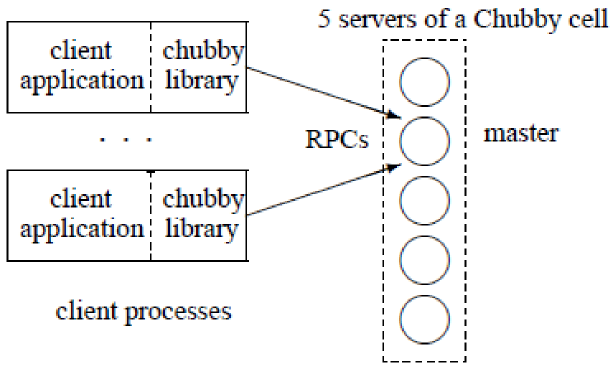


Figure 2.1: Chubby Architecture [1].

The purpose of this lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals are reliability, availability, etc. Chubby [1] does not emphasize throughput and data storage. Chubby has two main components: a server, and a client library. The server handles requests and all the communication between clients and servers is managed by the client library. Figure 2.1 demonstrates the architecture of Chubby. The ensemble or chubby cell is referred to as the group of servers responsible for keeping the service running. The servers or replicas undergo leader election using a consensus protocol. After the election the master is elected for a certain period of time which is renewable. Only the master is responsible for initiating any reads or writes to the database.

Clients keep sending requests to the master until it stops responding or its master lease has ended. The write requests are broadcasted to the replicas in order to have them acknowledged by the majority. The read requests are handled by the master alone. If the master lease ends or the master fails, then a new master is elected by the replicas after undergoing leader election.

2.2.2 Amazon Simple Queue Service

Amazon SQS [1] focuses only on the queuing and can be stated as a highly scalable, hosted queue for storing messages as they travel between computers. Its main focus is just to store messages and deliver them to different components in a distributed environment. The components [3] of Amazon SQS are as follows:

- System Components

- Queues
- Messages in the queues

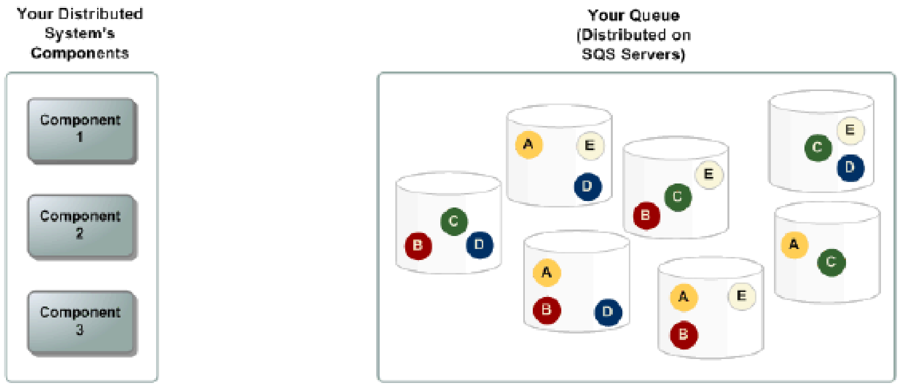

Your Distributed System's Components
Component 1
Component 2
Component 3
Your Queue (Distributed on SQS Servers)
A
E
D
B
C
D
E
B
C
C
E
D
A
B
D
A
E
B
A
C

Figure 2.2: Amazon SQS Structure[1]

Amazon SQS [1] has a different way of providing coordination. Messages that are stored in SQS undergo a lifecycle which is responsible for processing messages (send, receive, delete). Figure 2.2 bestows the structure of Amazon SQS. It can be seen from the figure that multiple SQS servers store messages in them. A redundant infrastructure is being followed as multiple copies of messages such as A,C,E, etc. is being maintained by different servers. There are different components that are responsible for different tasks. Some components are responsible for reading and writing from the queue, while some handle the sending and reception of messages.

Amazon SQS has the following features [1]:

- Redundant infrastructure
- Variable message size
- Access control
- Delay queues

Lastly, another important fact is that Amazon SQS is not open source i.e. the source code is not available for customization, and the customer needs to pay for the services as per their application usage [16] in comparison to the other coordination services.

2.2.3 Apache Hadoop

The Apache Hadoop [5] project is an open source project which supports distributed applications involving big data. Big data can be described as a collection of complex data sets which are too large to be processed using traditional data processing applications. Apache has evolved rapidly into a major information technology corporation and acts as an umbrella for a number of sub-projects. It is not a single product, but a collection of components such as ZooKeeper, Hive, HDFS etc.

Apache Hadoop [5][3] is widely deployed in organizations around the globe, including many of the worlds leading internet and social networking businesses. MapReduce is the key algorithm that the Hadoop MapReduce engine uses to distribute work around a cluster.

2.2.3.1 Apache Zookeeper Service Overview

ZooKeeper, a project incorporated within the Hadoop [5] ecosystem, is nowadays very much in demand due to its wide applicability for building distributed systems. ZooKeeper uses wait free characteristics of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems [14]. It does so to provide a simple, yet a powerful coordination service rather than a conventional approach of using locks in order to guarantee the order of operations being applied on these objects. It aims to provide a simple and high performance kernel for building more complex coordination primitives at the client.

The service can tolerate f crashes or failures, only if it has at least (2f +1) servers. For instance, if there are 9 servers then the service is capable of tolerating up to 4 failures. The reason lies in the design of Zookeeper which says that the service will work if the majority of servers (called a quorum) are available for servicing a request. Figure 2.4 shows the logical makeup of ZooKeeper [14].

The design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. Read requests are serviced from the local database which have a replica of the

ZooKeeper state. Write requests are transformed from ZooKeeper requests to idempotent transactions and sent through the broadcasting protocol used by ZooKeeper, called ZooKeeper Atomic Broadcast protocol (ZAB)[10] before a response is generated.

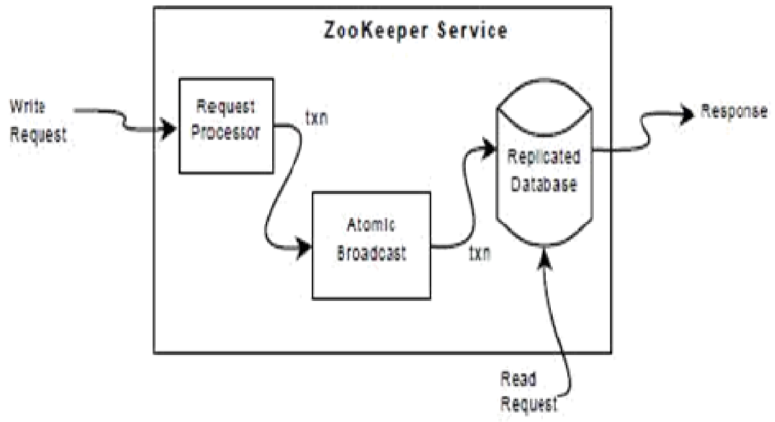


Figure 2.3: Logical Components of Zookeeper[4]

Requests are sent through the leader and it carries on the transformations of those requests to idempotent transactions. The leader can calculate the state of a new record as it has a perfect view of the replicated database. ZooKeeper[17][13] makes use of an in memory database and stores transaction logs and periodic snapshots on disk or wherever specified (e.g. a ramdisk or a dedicated device). ZooKeeper enforces some requirements on its core broadcast protocol i.e. ZAB. The requirements [4] are as follows:

- Reliable delivery: A successful delivery of a message ensuring that all replicas have a consistent state.

- Total order: Its main focus is that all the servers delivering messages must follow the same order. For instance, if one server delivers messages in the order Z,o,o,K,e,e,p,e,r then all the servers in the ensemble must follow the same order for delivering messages.

- Casual order: Ensures that the message that was proposed first must be ordered first for delivery. Moreover, if at any point the leader changes, then any previously proposed messages will be ordered before the messages from the new leader.

Other performance requirements are low latency, high throughput and smooth failure handling.

2.2.3.2 Zookeeper Atomic Broadcast Protocol (ZAB)

ZAB is considered to be the heart of ZooKeeper and is one of its vital components. ZAB [5] is a crash-recovery protocol designed for ZooKeeper. Basically ZAB is responsible for leader election, synchronizing the replicas, broadcasting updates and performing failure recovery. ZAB was inspired by a protocol called Paxos. Paxos [5] did not allow multiple outstanding transactions and could bear message loss and reordering. Moreover, Paxos is not as efficient as ZAB in recovering from crashes.

Epoch is an integer and is stated to be a period of time used for keeping track of transactions that are simple state changes. In ZAB, at a particular instant of time a peer can be found only in one of the three non-persistent states i.e following, leading, and election. ZAB carries out its functioning via the three phases listed below. Before proceeding to any of the phases it starts in Phase 0 that is termed as the leader election phase. The phases are as follows [5]:

- Phase 1: Discovery
- Phase 2: Synchronization
- Phase 3: Broadcast

Phase 1 and 2 are responsible for maintaining a consistent state of the ensemble or quorum especially in recovery from crashes. If no crashes occur then ZAB tends to stay in phase 3. If at any point of time some failure occurs with the leader, then ZAB can switch to the leader election phase. The client can guarantee the consistency of its state or replica by issuing a sync request. Read requests are served by the ZooKeeper server [5] . Write requests are converted into transactions and are serviced. Transactions are uniquely identified by transaction identifiers zxid, which is a 64-bit number. The zxid has two parts: the epoch and a counter. The high order 32-bits are used for the epoch and the lower order 32-bits for the counter [5]. The epoch number represents a change in leadership. Every new leader has its own epoch number. In case of a new leader the counter is set to zero and the epoch is incremented.

The discovery phase [5] is responsible for detecting and gathering the most recent transactions among the quorum and modifying the epoch in case a new leader comes into existence, ensuring not even a single commit of proposals by the previous leader. The synchronization phase [5] helps in maintaining consistency among the peers, and the broadcast phase takes the responsibility of handling new state changes, enabling the application to broadcast transactions.

2.3 Coordination Primitives

This section will demonstrate how ZooKeeper plays a vital role in implementation of coordination primitives such as configuration management[2], as in broadcasting configuration information to all the systems in a distributed environment. The approach ZooKeeper uses to implement these wait-free primitives, such as configuration management, group membership etc, will be described in the following sections.

2.3.1 Configuration Management

Configuration [2] can consist of both static or dynamic operational parameters. ZooKeeper [17][13] can be helpful to achieve dynamic configuration for distributed applications. For instance, bootstrapping information is stored on a znode, Zc. Now, whenever any system starts, it obtains the required information by reading Zc and setting the watch flag to true in order to ensure that system reading information is notified of any changes made to the bootstrapping information. Watches are one time triggers, and it is mandatory for processes to set the flag to true in order to have notifications of all the updates and avoid reading of any stale information.

2.3.2 Double Barriers

Double barriers [2] are helpful in providing synchronisation both at the beginning and end of computation. For instance, consider a familiar concept of an operating system. Whenever race conditions are raised, a process enters the critical section. When the job is done, the process leaves the critical section. The irony is that in the critical section only one process can enter, but in double barrier, multiple processes can enter and finish their computation ensuring synchronisation.

Consider a znode Zb as a barrier. Processes that want computation to be done are associated with Zb by creating a child node under Zb. After the computation ends, the processes delete the child node created by them. There are some entry and exit conditions for Zb. Processes are allowed to enter the barrier when maximum number of associations(maximum number of child nodes created) are accomplished for that barrier i.e. Zb in this case.

[2]. The exit condition is met when all the processes have removed their child nodes. Watches are used for checking whether the entry or exit conditions have been met.

2.4 Zookeeper Applications

There are a lot of users [18] of ZooKeeper, and some of them are Yahoo,Facebook, LastFm, Amazon, AOL, etc.

Yahoo Message Broker(YMB) [2] is a publish-subscribe system where clients can publish and subscribe to a large number of distinct topics. Each topic is replicated on two servers, i.e. primary and backup, in order to ensure reliable delivery of information. Zookeeper manages distribution of topics among the servers managing YMB, deals with failure detection of the servers, and control system operations. Figure 2.2 demonstrates the components of YMB.

There exists a root named broker domain that has a znode called nodes having all the active servers forming the YMB service. Each active server creates an ephemeral node (a node which is session bound) under nodes storing load and status information. Centralized control is allowed through shutdown and migration-prohibited nodes, and it is monitored by all the servers. The topics node consists of child nodes in which each node consists of specific information about a particular topic managed by YMB. YMB also follows the same primary-backup scheme for storing topic information [4].

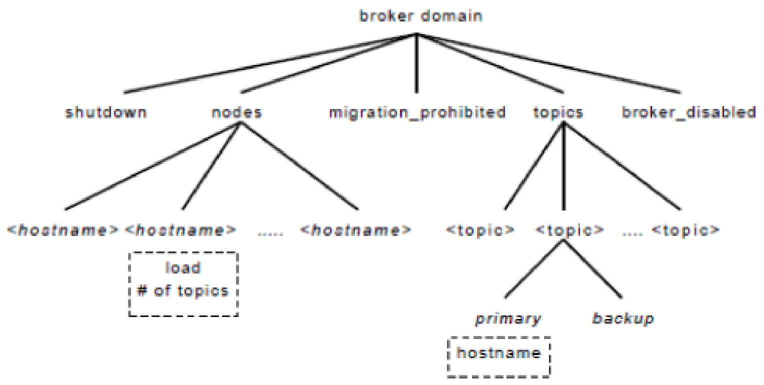


Figure 2.4: Layout of Yahoo Message Broker [2]

Facebook uses Hadoop [16] for generating brief reports after analysing big data. Facebook carefully assesses these reports and tries to find patterns such as which application is mostly being liked by users or which page is getting the highest number of hits. Advertisements are a form of big data and they are using Facebook intensely. Another area where Hadoop and ZooKeeper go hand in hand is in maintaining the integrity of the website, thus protecting it from attackers [18][19].

LastFm uses Hadoop due to its unique feature of being open source. The Hadoop Data File System(HDFS) allowed LastFm to store their backups without incurring any cost. The pliable Hadoop framework also provides support for distributed computational algorithms. Hadoop also helped LastFm to increase its scalability. Hadoop has become an integral part of LastFm architecture that uses it for performing tasks such as assessment and analysing of logs, network processing, and report generation. [18]

The New York Times is also a user of Hadoop. This company uses Hadoop for transforming archives of old microfiche media to formats such as PDFs in order to enhance the scalability, searchability and usability of the content generated after analyzing big data [18].

Chapter 3

Project Design

This chapter explains the design of this project. The requirements for this project will be explained in Section 3.1, followed by the approach to accomplish the requirements in Section 3.2, and emphasize on the configurations being used in Section 3.3.

3.1 Project Requirements

For this research project, the requirements are as follows:

- Inspect and estimate the factors that may affect the performance of ZooKeeper [13].
- Modify the source code of ZooKeeper in order to improve its performance ensuring that it functions properly and the same way as before the changes were made.

- Establish benchmarks for the modified version of ZooKeeper.
- Evaluate the altered version of ZooKeeper and infer conclusions.

In order to comply with the requirements there should not be any change in the behaviour of ZooKeeper [13]. The unchanging behaviour means that the applications using ZooKeeper for reception, processing, and service of requests should not be able to see any changes except for performance impacts.

Another consideration to be kept in mind is to make the code modifications as minimal as possible. Files that involve critical code should be altered carefully. Watchfulness is required as each and every change can lead to instabilities in other parts of the code; therefore alterations should be made only where necessary. If changes in different parts need to be made, then these changes need to be evaluated in order to ensure they are working correctly. The normal functioning of ZooKeeper after the alterations progresses to the requirement that everything is the same, except for the performance impacts which should be noticeable.

For the evaluation of this project, ZooKeeper performance needs to be measured twice. In the first measurement, ZooKeeper is functioning without the changes, and in the second measurement ZooKeeper functions after the changes have been made. The metrics for evaluation include throughput and latency. Details of the evaluation approach are provided in Section 5.2.

Lastly, after the evaluation phase is completed, comparisons will be drawn between the performances obtained in two different scenarios. Now, conclusions can be made, and some benchmarks will be established.

3.2 Approach

To be able to meet the requirements of Section 3.1, the general task needs to be composed into the different subtasks as follows:

- Firstly, understanding the basic mechanism of ZooKeeper such as to how it works through its official documentation and research papers.
- Investigate the core components of ZooKeeper, such as request processing pipeline and ZooKeeper Atomic Broadcast Protocol (ZAB).
- Familiarity with the concept of nodes to understand the parameters that would be used to gauge the performance of ZooKeeper. The parameters are throughput, latency, and response time. Throughput is the number of operations or requests serviced in a particular instance of time. Latency refers to the amount of time taken by ZooKeeper to complete the assigned task. Response time refers to the time taken by ZooKeeper to notify a client in some cases as to how much time ZooKeeper required to respond back to clients with a particular type of request.

- Examine the area where there is a scope of improvement i.e. investigating around the request processing pipeline.

- Create a test suite comprising of different tests that focus on measuring the performance of ZooKeeper. This test suite involves familiarity with the API docs as a preliminary step.

- Ensure that ZooKeeper is running properly.

- Execute the test suite with different configurations on ZooKeeper service to acquire results, as explained in Section 3.3.

- Analyze the results and plot performance matrices or graphs.

- Modify the source code of ZooKeeper ensuring that there are no instabilities in the code after the changes and ZooKeeper service is running and functioning normally.

- Rerun the test suite after the modi

  cations have been made with different configurations to gather new results as discussed in Section 3.3.

- Analyze the new results and plot performance matrices again.

- Draw a contrast between the graphs generated in both cases.

- Infer conclusions regarding the results.

3.3 Configurations

The ZooKeeper test suite was executed multiple times with different configurations. The machine used for this project consists of an Intel Core i5-2600 processor, which consists of four cores. Furthermore, this computer featured 6 gigabytes of RAM. The evaluation of this project will also be done on this hardware. The operating system used for this project is the Ubuntu 12.04 64-bit version. The test suite discussed in Section 5.1 was first executed by storing the logs in traditional hard disk and later the log storage was changed to ramdisk. The test suite executed faster in ramdisk. So, out of 6 gigabytes of RAM, 4 gigabytes was mounted and used for log storage.

The ZooKeeper server can run in two modes: standalone and replicated. The standalone mode consists of a single server that services requests. On the contrary, the replicated mode consists of at least 3 servers because more than half of the servers have to be running to keep the service up. In the replicated mode, all servers share the same configuration file and have a distinguishing feature as a myid file which consists of a unique ID given to each server.

The servers are allotted different IPs and quorum ports, these ports are used for leader election. After leader election occurs, a leader is elected, and the clients connect to the leader, which then carries on with the handling of requests and synchronizing with the followers. Some results in regard to both modes will be discussed in Section 5.1. These results were gathered using a machine which had 32 gigabytes of RAM and an Intel Core i7 processor. Out of 32 gigabytes of RAM, only 10 gigabytes of RAM were dedicated for the test suite by mounting it.

Chapter 4

Realization

This chapter describes the implementation side of this project. Section 4.1 starts with an introduction into the structure of the ZooKeeper code. After that outlines Section 4.2 some performance aspects in regard to the type of znodes in ZooKeeper. Next, Section 4.3 will talk about performance aspects of queues and their usage in ZooKeeper. Section 4.4 solely focuses on the implementation part of this project. Section 4.4.1 then describes the concept as well as the usage of request processors in ZooKeeper. Section 4.4.2 will talk about the serialization and its performance aspects [20][21] in ZooKeeper. Lastly, Section 4.4.3 will discuss logging mechanisms in ZooKeeper and its performance impacts.

4.1 ZooKeeper Architecture

This section will give an overview of the ZooKeeper architecture and how its source code is organized into distinct packages. ZooKeeper consists of a quorum of servers, where one is the leader and the remaining servers are the followers. The clients connect to any of the servers present in the quorum and issue read and write requests. These requests are made possible by manipulating znodes through ZooKeeper's client API library. Now on the basis of the type of request it is appropriately handled and a response is generated as discussed in Chapter 2. Figure 4.1 and 4.2 demonstrate different packages present in ZooKeeper. These figures also provide a broader view of the functions being performed by each of the packages.

Figure 4.1 consists of four packages in which the `org.apache.zookeeper.jmx` package does its work through jconsole. By default it is switched off but its settings can be customized. The second package is `org.apache.zookeeper.zookeeper.client` which provides four letter commands such as `srvr, ruok`, etc. that are used to check the status of ZooKeeper server. Next is the `org.apache.zookeeper` package that takes care of managing socket connections. It also provides client libraries that enable a client to make a request. The last package is `org.apache.zookeeper.server.quorum` which performs critical tasks, such as conducting leader election and handling broadcast of messages, thus providing distributed coordination ensuring synchronization.

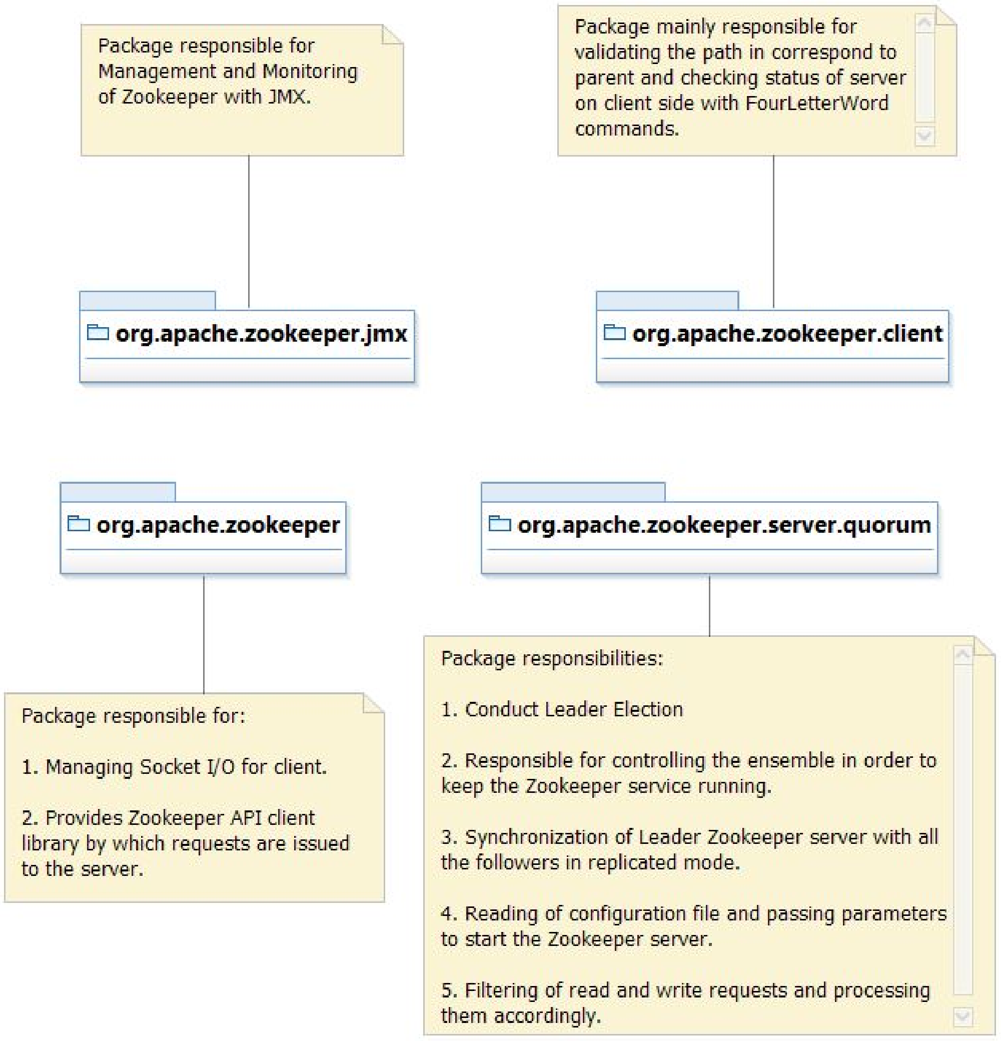

Package responsible for Management and Monitoring of Zookeeper with JMX.
Package mainly responsible for validating the path in correspond to parent and checking status of server on client side with FourLetterWord commands.
org.apache.zookeeper.jmx
org.apache.zookeeper.client
org.apache.zookeeper
org.apache.zookeeper.server.quorum
Package responsible for:
1. Managing Socket I/O for client.
2. Provides Zookeeper API client library by which requests are issued to the server.
Package responsibilities:
1. Conduct Leader Election
2. Responsible for controlling the ensemble in order to keep the Zookeeper service running.
3. Synchronization of Leader Zookeeper server with all the followers in replicated mode.
4. Reading of configuration file and passing parameters to start the Zookeeper server.
5. Filtering of read and write requests and processing them accordingly.

Figure 4.1: Packages in ZooKeeper

Figure 4.2 is composed of packages such as `org.apache.zookeeper.auth` that focusses on the authentication with the help of Access Control Lists(ACLs). The package that handles the logging part and is responsible for recovering from crashes or failures is `org.apache.zookeeper.persistence`. There

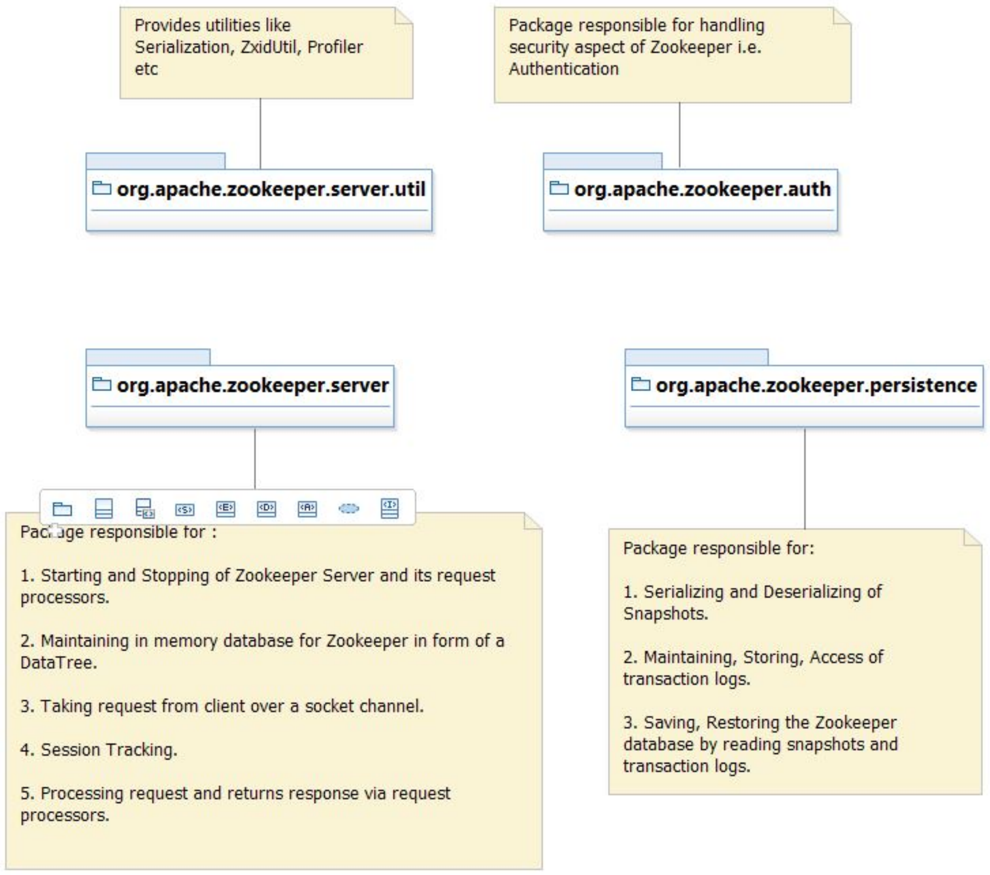

Provides utilities like Serialization, ZxidUtil, Profiler etc
org.apache.zookeeper.server.util
Package responsible for handling security aspect of Zookeeper i.e. Authentication
org.apache.zookeeper.auth
org.apache.zookeeper.server
org.apache.zookeeper.persistence
Package responsible for :
1. Starting and Stopping of Zookeeper Server and its request processors.
2. Maintaining in memory database for Zookeeper in form of a DataTree.
3. Taking request from client over a socket channel.
4. Session Tracking.
5. Processing request and returns response via request processors.
Package responsible for:
1. Serializing and Deserializing of Snapshots.
2. Maintaining, Storing, Access of transaction logs.
3. Saving, Restoring the Zookeeper database by reading snapshots and transaction logs.

Figure 4.2: Additional ZooKeeper Packages

is a utility package present named `org.apache.zookeeper.server.util`. Lastly, another vital package is `org.apache.zookeeper.server` that is liable for managing the ZooKeeper server and handling requests from clients.

ZooKeeper is fast and efficient in obtaining its goal of achieving synchronization and coordination for complicated high performance distributed applications. The robust architecture of ZooKeeper has taken care of almost all the aspects which may be problematic from the perspective of applications using it. These aspects make it a preferred choice among all of the other coordination services. The aspects or guarantees ZooKeeper makes are expressed below [18]:

- Request handling as atomic operations avoiding any partial updates.
- Avoids split-brain: ZooKeeper ensures that all of its servers are up to date and no server is lost. Each server knows about all the servers in the quorum and keeps on sending continuous checks to each other ensuring connectivity. It is also helpful in avoiding data corruption.
- Homogeneous system view: All the clients are exposed with the same interface of ZooKeeper service regardless of which server of the quorum a client is connected to.
- Reliability and Consistency: This is ensured with the help of ZooKeeper atomic broadcast protocol that lies in the center of ZooKeeper.
- Timeliness: Continuous up to date information is presented to the client as long as it is connected to ZooKeeper.

4.2 Hierarchical Data Node Performance

ZooKeeper uses *znodes* as discussed in the background chapter. These are used by clients to implement their coordination tasks and denote an in-memory space as having some data. These *znodes* form a hierarchical namespace which is termed as a data tree. This namespace not only provides abstraction but also provides better organization of application meta-data used for coordination purposes. Access rights can also be associated with this namespace which is better from a security point of view. Clients access a given node with the help of conventional UNIX notation used for specifying system paths [4][2].

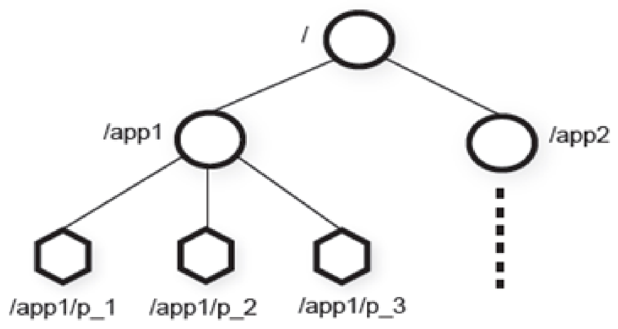


Figure 4.3: DataTree [2]

In Figure 4.3 the node / is the parent node and underneath it are its children. Child nodes are accessed with the help of a parent node. Basically there are two types of znodes that can be created by a client, these are ephemeral and persistent nodes. Ephemeral nodes are non children nodes that simply correspond to the session in which they were created. They are deleted automatically as soon as the session terminates, but can be deleted explicitly as well. On the other hand are persistent nodes that clients manipulate by creating and deleting them explicitly.

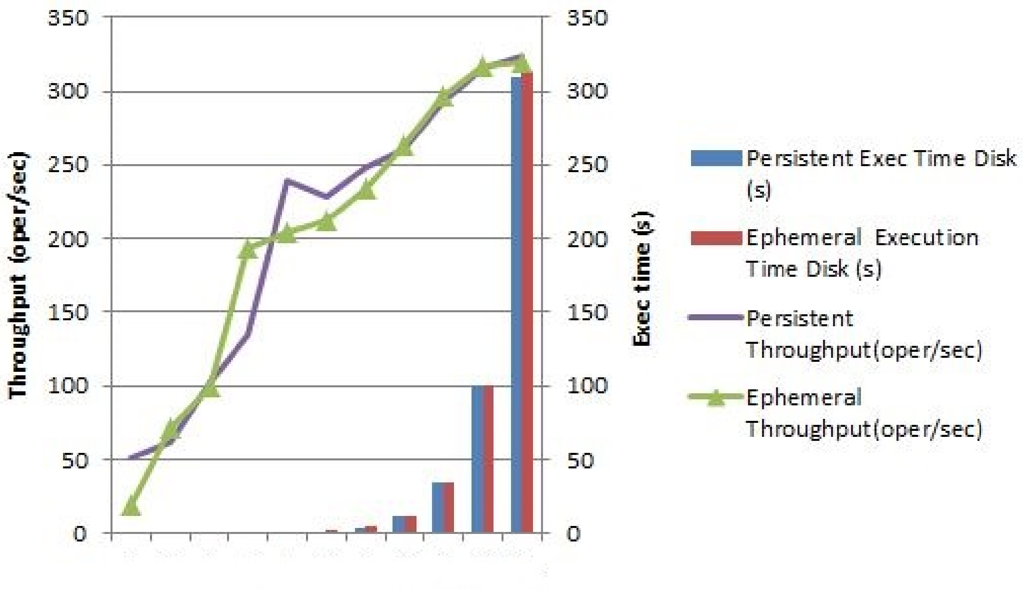


Figure 4.4: Persistent Node Performance on Disk

Some variations can be incorporated with the above nodes such as setting up a sequential flag and binding it to the node. If the sequential flag is set, then ZooKeeper appends a unique monotonic increasing counter to the name of the parent node in order to identify it uniquely. For instance, suppose the parent node `zkTest1` and its sequential flag is set. The child nodes that will be created will be named as `zkTest100000001`, `zkTest100000002` and so on.

The concept of watches has been implemented in ZooKeeper in order to notify clients about the changes. For instance, a node named `zkTest100000001` is created with a watch flag set then as soon as any operation is carried on

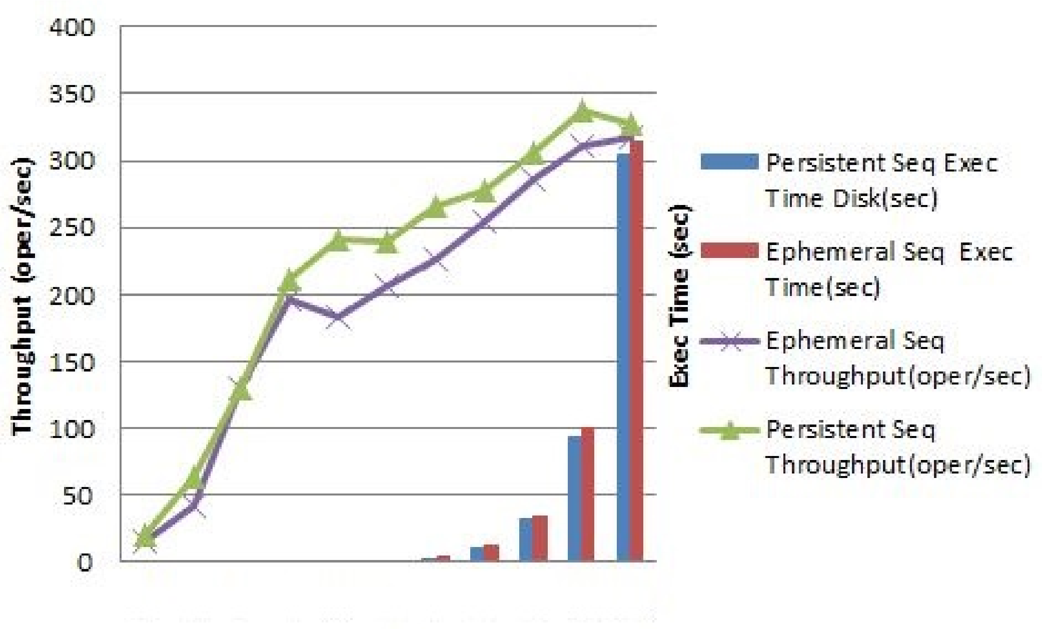


Figure 4.5: Persistent Sequential Node Performance on Disk

this particular node, i.e. the version number is changed (incremented), the client is notified about the change. An important fact about the watch flag is that they are one time triggers which are associated with the session.

After executing the test suite in Section 5.1 some results were extracted and graphs were plotted between various parameters. After analyzing the graphs it can be seen how ZooKeeper behaves under different scenarios and configurations. Conclusions can be drawn in order to interpret these results.

Another important fact which needs to be mentioned is that the ZooKeeper server version also played an important role for the tests as the same tests provided different results. For example, one test which just focussed on the

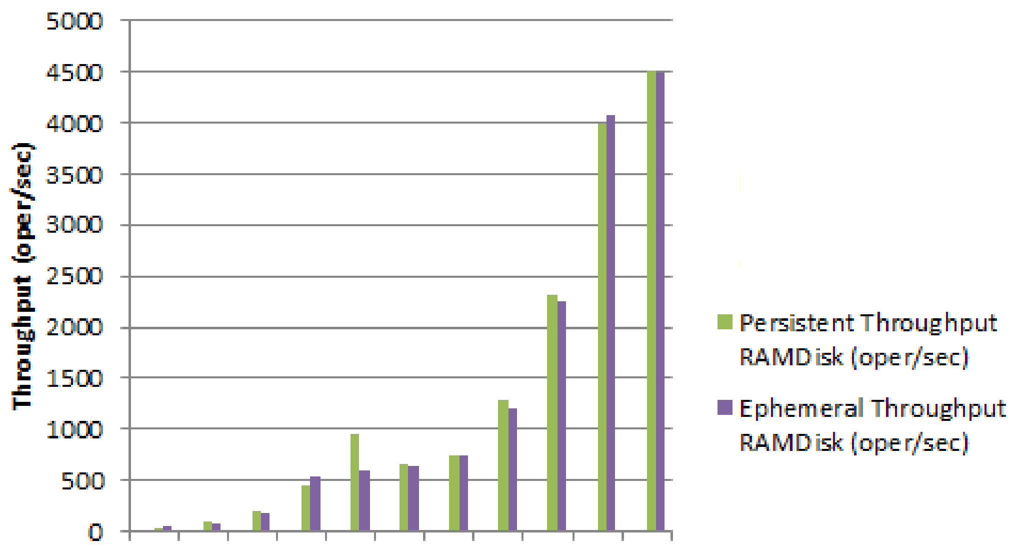


Figure 4.6: Persistent Node Performance on Ramdisk

znode creation based on its type (whether it is ephemeral or persistent) gave different figures for different ZooKeeper versions (3.3.6 and 3.4.5).

It has been found that persistent znodes have higher throughput and less latency as compared to persistent sequential nodes because these involve extra processing done by the request processors. Especially for ephemeral nodes the server has to clean up after the session expired. Therefore it needs to maintain some data structure for these nodes. All of the tests were carried out with the stable version of ZooKeeper (i.e. 3.4.5). Moreover, all results have been stored in comma separated values (CSV) files so that the same file can be used for exporting data. Figures 4.4, 4.5, 4.6, and 4.7 support the statements made above. Figures 4.4 and 4.5 correspond to graphs taken

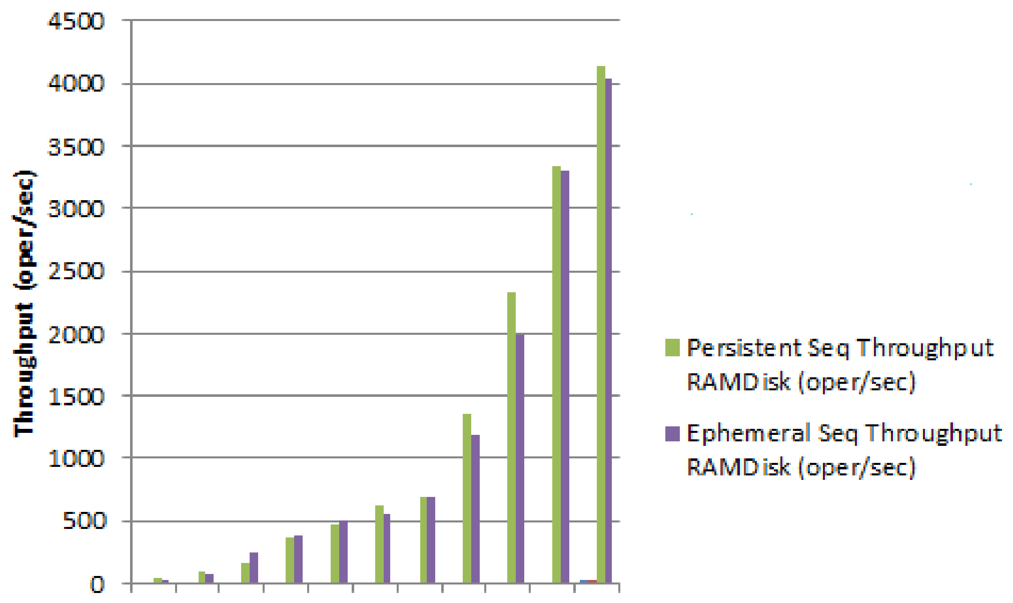


Figure 4.7: Persistent Sequential node performance on ramdisk

by executing the test suite on a traditional hard disk and on the contrary Figures 4.6 and 4.7 correspond to ramdisk. Figures 4.4, 4.5, 4.6, 4.7 depict performance aspects in case of *znodes* based on their type.

4.3 Queue Performance Aspects

Prior to using queues in ZooKeeper, it is advisable to test which queue would lead to better performance. A sample program was written to test the performance of queues. The performance was tested for both linked blocking queues [9] and concurrent linked queues [10] as described in Section 2.2. The comparison was made based on the time taken by the methods of both the queues. For instance, methods such as `put()` and `take()` belong to a linked blocking queue and on the other hand, `offer()` and `poll()` perform the same function but belong to a concurrent linked queue.

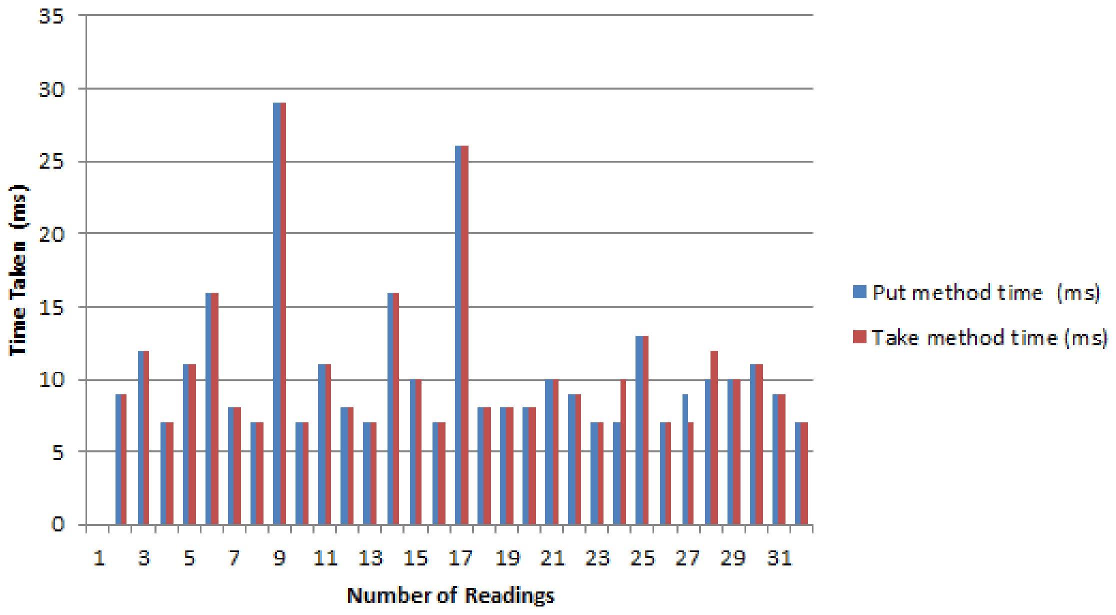


Figure 4.8: Performance of Linked Blocking Queue

It has been found that on average the `offer()` method is 1.3 times faster than the `put()` method in inserting requests on the queue and, similarly the `poll()` method is 1.6 times faster than the `take()` method in deleting requests from a queue. Figures 4.8 and 4.9 demonstrate the performance given by the methods of corresponding queues. Figures 4.10 and 4.11 are used to show a general comparison between the queues and to justify the reason why linked blocking queues were replaced with concurrent linked queues in ZooKeeper.

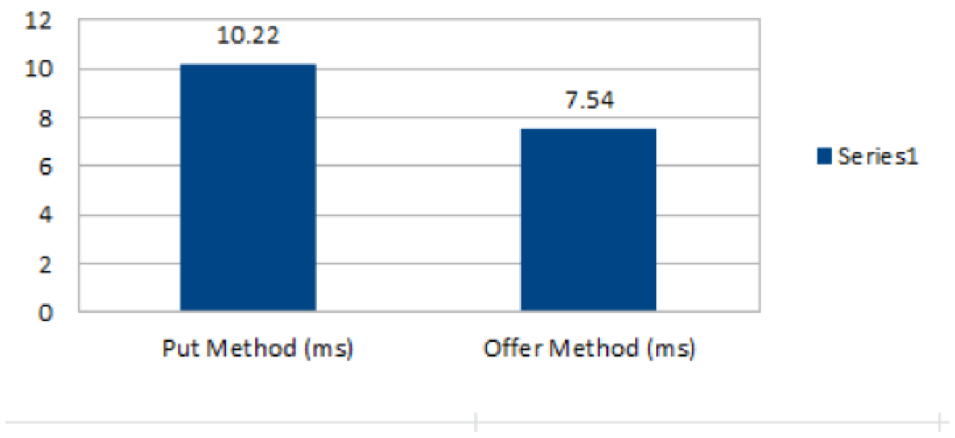


Figure 4.9: Comparison between Methods Offer and Put

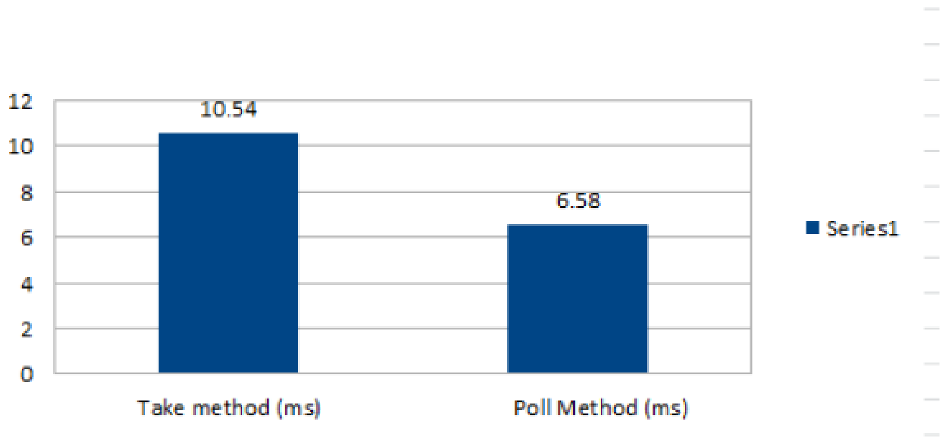


Figure 4.10: Comparison between methods Take and Poll

4.4.1 Request Processors

Request processors form a critical part of the ZooKeeper service. Whenever a write request is received by a leader it computes the future state and converts the request into idempotent transactions. The version number is incremented to depict that it is the future state. For instance, consider a `setData` request. If everything goes perfectly it is converted to `setDataTxn` and, if it does not, then a corresponding `errorTxn` is generated. The complete code can be seen in Appendix A.2 but fewer lines are presented below in Section 4.4.1.2.

Client API Library

This ZooKeeper Client API library belongs to org.apache.zookeeper package, and the presence of this library enables the clients to submit their requests to ZooKeeper. The methods along with their semantics and a little description that are used to make requests in ZooKeeper are as follows:

- **create(path, data, nodeType, flags)**: Creates a type of znode based on the nodeType at the path specified, stores data in it and returns the name of the node created.

- **delete(path, version):** Deletes the znode at the path specified if the znode has the expected version. -1 is considered to be the default version.

- **getData(path, watch):** Returns the meta-data associated with the znode at the path given. The watch flag is used for notifications.

- **setData(path, data, version):** Writes data to the znode.

- **getChildren(path, watch):** Returns all the child nodes for a znode present at the given path.

These methods are the synchronous version of the ZooKeeper API. The asynchronous versions are also available and are used in scenarios where multiple tasks need to be executed in parallel using callbacks. The test suite described in Chapter 5 also involves tests where the asynchronous version is used. The throughput was higher for the asynchronous version. Almost all the methods mentioned above take an expected version number that is mandatory for applying updates. If the actual version number of the znode does not match the expected version number then the update cannot be applied. For testing reasons a default version number can be used i.e. -1.

Code Snippets

This section presents the code as to how request processors transform write requests to idempotent transactions.

```
import java.util.concurrent.ConcurrentLinkedQueue;
ConcurrentLinkedQueue <Request> submittedRequests =
               new ConcurrentLinkedQueue <Request>();
function run () {
  try {
               Request request ;
               do {
                       request = submittedRequests.poll();
                  } while ( request == null );
```

```
    }catch( Exception e)
              {
              LOG. error (" Unexpected exception ", e);
              }
        pRequest ( request );
}
```

In the above code the request processor takes into account all the requests to be processed through the `poll()` method of the concurrent linked queue.The code below bestows as to how a write request such as setData is transformed to `setDataTxn`.

```
function pRequest (Request request)
{
        SetDataRequest setDataRequest = new SetDataRequest();
        pRequest2Txn(request.type, zks.getNextZxid(), request,
                                    setDataRequest, true);
}

 function pRequest2Txn (int type, long zxid, Request request,
                                    Record record, boolean deserialize)
 {
       SetDataRequest setDataRequest = (SetDataRequest) record;
       path = setDataRequest.getPath();
       version = setDataRequest.getVersion();
       int currentVersion = nodeRecord.stat.getVersion();
       if ( version != -1 && version != currentVersion)
       {
```

```
                throw new KeeperException.BadVersionException(path);
        }
  version = currentVersion + 1;
  request.txn = new SetDataTxn(path, setDataRequest.getData(),version);
}
```

The code below demonstrates the generation of *errorTxn* in case of failure. The *lastOp* is the type of request, i.e. `setData` represented as *SETD*. The acronyms such as *rsp* and *err* stand for response and error correspondingly.

```
function processRequest(Request request)
{
 try{
        case OpCode.setData :
         {
                lastOp = " SETD ";
                rsp = new SetDataResponse (rc. stat);
                err = Code .get(rc.err);
                break ;
         }
  }catch ( Exception e)
        {
         LOG . error (" Failed to process " + request , e);
        }
 }
```

4.4.2 Marshalling Mechanisms

This section emphasizes the different types of marshalling mechanisms such as serialization [22][23] and externalization. This section also shows the workflow for both serialization and externalization along with their merits and demerits. This section demonstrates the performance aspects of marshalling mechanisms. The reason as to why externalization is preferred over serialization is justified by means of Figures 4.12 and 4.13. The section also shows usage of these marshalling mechanisms with Apache ZooKeeper along with its performance aspects.

4.4.2.1 Comparison of Serialization and Externalization

Serialization mechanism refers to the process of transforming the state of persistent object into a format that can be stored such as bytes. This process is also sometimes termed as the deflating or marshalling of an object. This process is used by Remote Method Invocation(*RMI*) to pass objects between Java Virtual Machines(*JVMs*), and to store user sessions in web applications, etc. In order for a class to become serializable it needs to implement a marker interface termed as Serializable. On the other hand, there exists an Externalization interface which does the same process as serialization, but the code for marshalling and demarshalling needs to be written explicitly for this case. However, there are performance benefits to this overhead [24].

Externalization leads to higher performance than the default serialization. The primary reason is that serialization is a recursive algorithm and the identity of all the classes, its parent classes, instance variables and their information, is written to the serialization stream. This writing of meta-data is the bottleneck for performance and consumes a lot of time. On the other hand, externalization only writes the class identity to the serialization stream, and the class bears all the responsibility to save and restore the contents of its instances. Thus, complete control is handed over to the class. Class determines what data it needs to marshall. Secondly, in the default serialization mechanism, a unique ID is generated and termed the *serialversionUID* , this generation is again time consuming, which is not the case with externalization. Lastly, apart from performance impacts, using externalization also leads to a reduction of file size [24].

4.4.2.2 Generic Tests and Results

After theoretically analysing the advantages of Externalization over Serialization, some practical results were required. Prior to altering the default serialization and deserialization mechanism in ZooKeeper, a general test was conducted to compare the performance of the externalization mechanism against the default serialization mechanism. The time taken to serialize and deserialize a simple object was calculated in this test. The test was executed multiple times. The readings were analysed, and graphs were plotted based on the results. The process was repeated with the default serialization mechanism changed to externalization. The figures in this section illustrate that externalization performs better than default serialization.

The code specified below is for the test using the default serialization mechanism. The `writeObject()` and `readObject()` methods perform the marshalling and demarshalling, and no customized code needs to be written.

```
public class Employee implements Serializable
{
      function serializeObject(Employee object)
      {
             ByteArrayOutputStream baos = new ByteArrayOutputStream();
             ObjectOutputStream oos =new ObjectOutputStream(baos);
             oos.writeObject(object);
      }
      function deserializeObject(byte [] rowObject)
      {
             ObjectInputStream ois = new ObjectInputStream
                                    (new ByteArrayInputStream(rowObject));
             Employee res = (Employee)ois.readObject();
      }
}
public class Employee implements Serializable
{
      function serializeObject(Employee object)
      {
             ByteArrayOutputStream baos = new ByteArrayOutputStream();
             ObjectOutputStream oos =new ObjectOutputStream(baos);
```

```
                oos.writeObject(object);
        }

        function deserializeObject(byte [] rowObject)
        {
                ObjectInputStream ois = new ObjectInputStream
                                        (new ByteArrayInputStream(rowObject));
                Employee res = (Employee)ois. readObject();
        }
}
```

The below code demonstrates the use of the externalization mechanism for the test. The `writeExternal()` and `readExternal()` methods carry out the marshalling and demarshalling process, but code must be written for these methods.

```
public class Employee1 implements Externalizable
{
        function serializeObject(Employee1 object)
        {
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                ObjectOutputStream oos =new ObjectOutputStream(baos);
                object.writeExternal(oos);
        }
        function deserializeObject(byte [] objectBytes)
        {
                ObjectInputStream ois = new ObjectInputStream
                                        (new ByteArrayInputStream(objectBytes));
                Employee1 res1 = new Employee1();
                res1.readExternal();
        }
        function writeExternal(ObjectOutput objectOutput)
        {
                // customised code for marshalling
        }
       function readExternal (ObjectInput objectInput)
       {
          // customised code for demarshalling
       }
}
```

Figures 4.12 and 4.13 show the individual performance [25][26] of both the serialization and externalization mechanism. Figure 4.14 demonstrates the performance comparison of the marshalling process for both serialization and externalization, and that the externalization marshalling process is `3.1` times faster. Figure 4.15 shows the performance comparison of the demarshalling process for both serialization and externalization, and depicts that the demarshalling process of default serialization is `1.52` times faster than externalization.

4.4.2.3 Serialization with ZooKeeper

ZooKeeper uses serialization for storing snapshots on the disk that contain all the updates. These snapshots supersede all the logs and are useful when recovery needs to be performed in order to retrieve a consistent state of ZooKeeper. The serialization mechanism has been altered to externalization due to the facts stated in Section 4.4.2.1. Some alterations have been made to the source code of ZooKeeper which have lead to performance im-

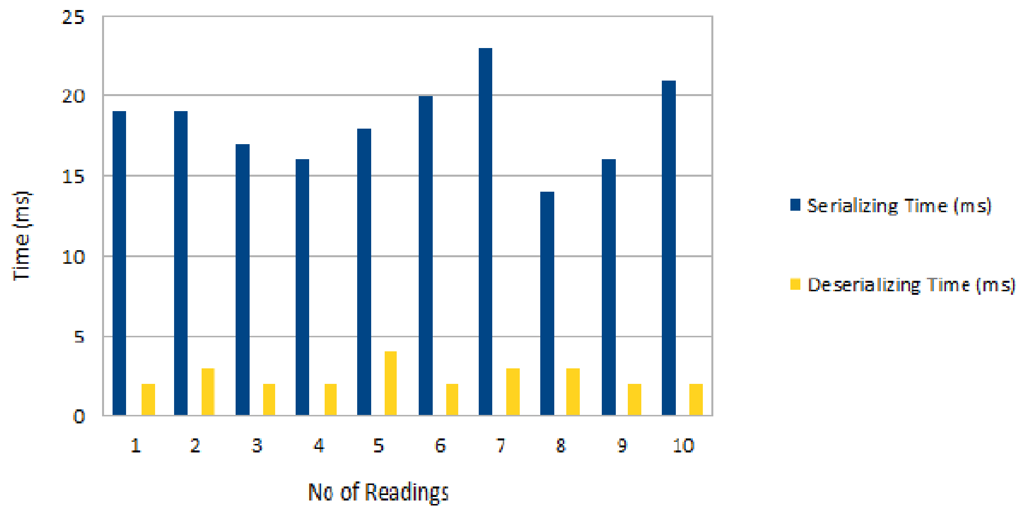


Figure 4.11: Performance of Serialization Mechanism

provements. These improvements will be demonstrated in Chapter 5. Some of the main files that have been altered will be found in Appendix A.1 and A.2.

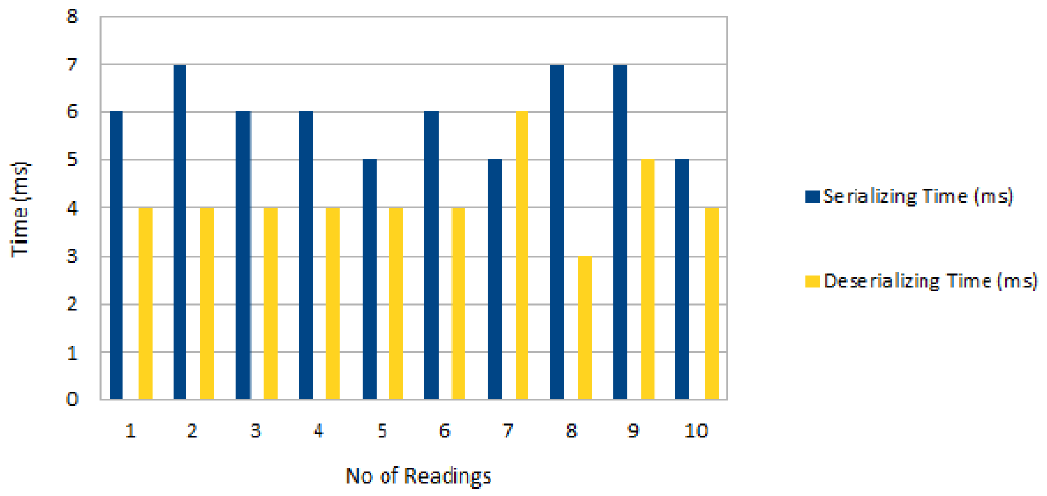


Figure 4.12: Performance of Externalization Mechanism

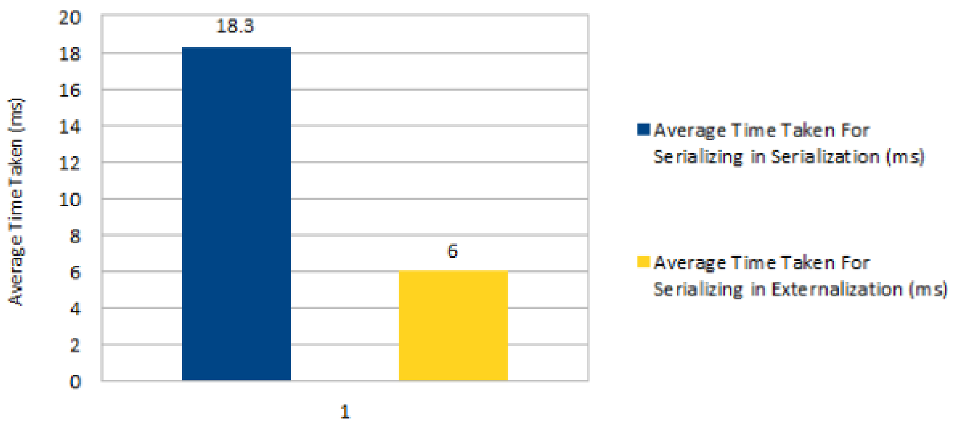


Figure 4.13: Comparison of Serialization Mechanism

4.4.3 Logging Mechanism in ZooKeeper

This section covers the logging aspects in ZooKeeper. The set of commands specified in the Figure 4.16 are used to mount and create a ramdisk as a data directory, so that all the logs and snapshots are stored in it. There are immense performance improvements in the tests by storing the logs to ramdisk. These improvements will be shown in Chapter 5. Also, log4j is used as a logging infrastructure in ZooKeeper. Log4j levels are namely `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, etc [13].

Figures 4.16 and 4.17 demonstrate the procedures or control flow as to how marshalling and demarshalling of a znode occurs. Figure 4.16 shows how marshalling occurs in ZooKeeper. Firstly, the ZooKeeper server takes a snapshot of the database upon its startup, and then through the marshalling

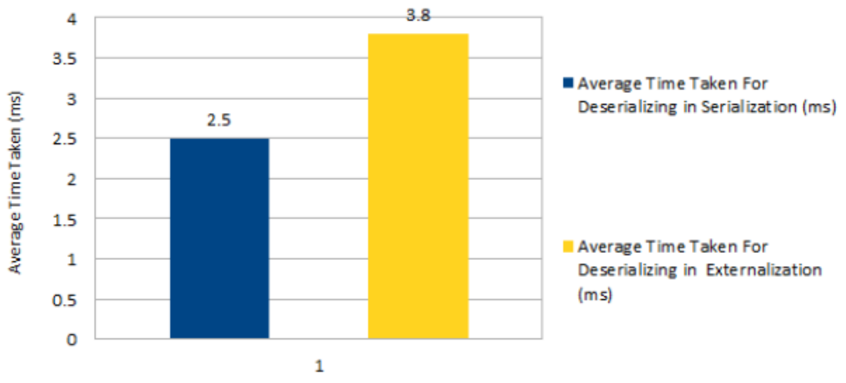
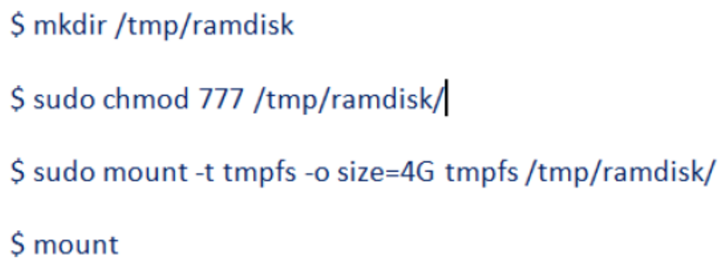


Figure 4.14: Comparison of Deserialization Mechanism



```
$ mkdir /tmp/ramdisk

$ sudo chmod 777 /tmp/ramdisk/

$ sudo mount -t tmpfs -o size=4G tmpfs /tmp/ramdisk/

$ mount
```

Figure 4.15: Set of commands

code written in the `writeExternal()` method the *znode* is serialized to disk. ZooKeeper only considers failures such as power outages or full system crashes, so in that case hardware is prone to failure. So, in that case keeping log files in either hard disk or ramdisk would be similar except noticeable performance impacts i.e. high performance when logs are being stored on ramdisk.

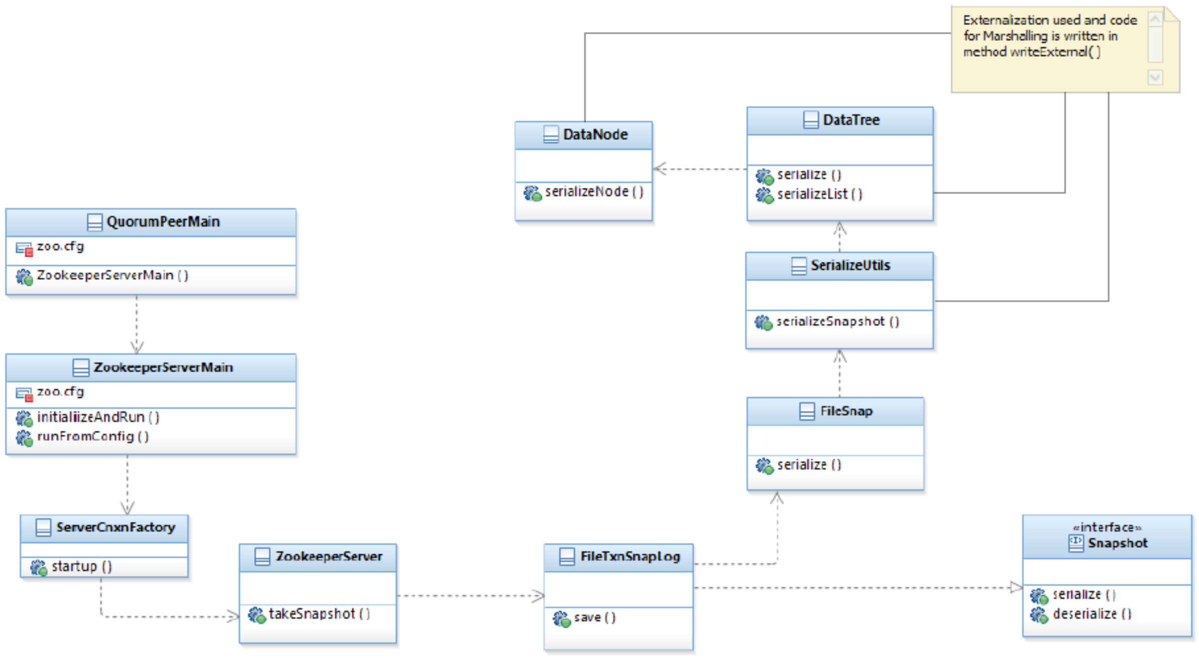


Figure 4.16: Serializing Control Flow in ZooKeeper

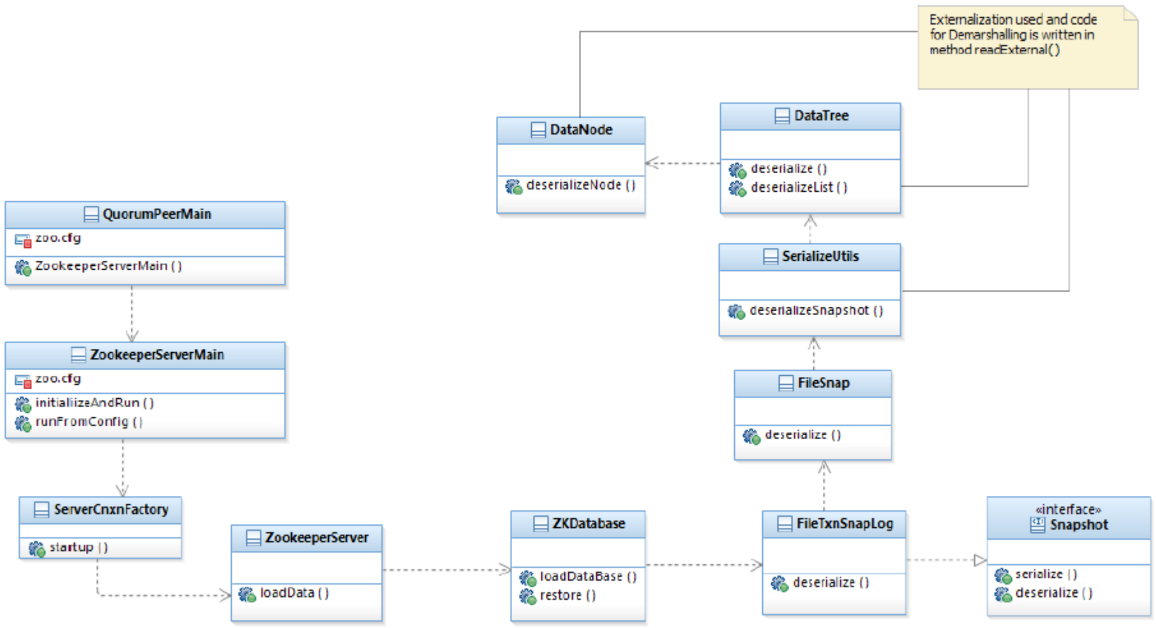


Figure 4.17: Deserializing Control Flow in ZooKeeper

Similarly, in Figure 4.17, the ZooKeeper server calls the `restore()` method to read the snapshot. After reading the snapshot through the `readExternal()` method, the demarshalling of the required *znode* can be done.

Chapter 5

Testing and Evaluation

This section will demonstrate the approach adopted to test and evaluate the modifications made to the ZooKeeper code in order to increase its performance. Section 5.2 will provide an explanation of the test suite that has been used in order to test ZooKeeper. This section will also show some of the graphs that will provide justification for the performance enhancement that has been accomplished after modifying the ZooKeeper source code. Some conclusions regarding the behavior of ZooKeeper can also be inferred after analysing the graphs. Some of the code listings can be seen in Appendix A.1 and A.2.

5.1 Approach

The major aim of the project is to improve the performance of ZooKeeper. In order to accomplish the project goals stated in Chapter 3, a test suite has been made. The test suite consists of tests that are used to analyze the performance of ZooKeeper. Execution of the test suite described in this section has generated some results. These results can be used for a base measurement of performance. Furthermore, these results can also lead to new insights on where performance improvements can be made. Also, the results obtained have been used to plot some graphs showing various parameters. After analyzing the graphs, ZooKeeper's behavior under different scenarios and configurations can be observed.

This chapter evaluates the approach followed in order to accomplish the goals. The alterations to ZooKeeper code shown in Chapter 4 will be evaluated in this chapter. Upon successful integration of those changes in ZooKeeper, the test suite is executed again in order to check for performance impacts. Apart from performance impacts another important check that needs to be performed is the normal functioning of ZooKeeper after incorporating the changes. Another important fact which needs to be stated is that the ZooKeeper server version also played an important role. The znode creation based on its type (whether it is ephemeral or persistent) resulted in different figures for different ZooKeeper versions (3.3.6 and 3.4.5).

5.2 Test Suite Description

This section emphasises the test suite along with its description, which evaluates the approach followed in order to enhance the general performance of ZooKeeper. ZooKeeper also supports request handling with multiple servers present in the quorum. ZooKeeper has two modes `Standalone` and `Replicated`. In Standalone, only one server is present for request handling. On the contrary, in replicated mode atleast 3 servers are present in the quorum for request handling. All of the 3 servers can communicate with each other through a common configuration file called `myid` file. The *IP* addresses of all the servers along with quorum ports i.e. ports used for leader election are written in this file. Most of the testing was accomplished using a single server due to hardware limitation, but some tests were also executed in replicated mode using a high performance machine. For instance, recovery of ZooKeeper after leader failure, quorum failure, etc.

5.2.1 Performance Comparison of Different Versions of ZooKeeper

The machine used for this test consists of an Intel Core i7 processor. Moreover, out of 32 gigabytes of RAM 10 gigabytes was used to execute the test by mounting as a ramdisk. This subsection will describe a preliminary test involving the creation and setting of 100,000 *znodes* as performed with different versions of ZooKeeper, i.e. 3.3.6 and 3.4.5, in order to investigate

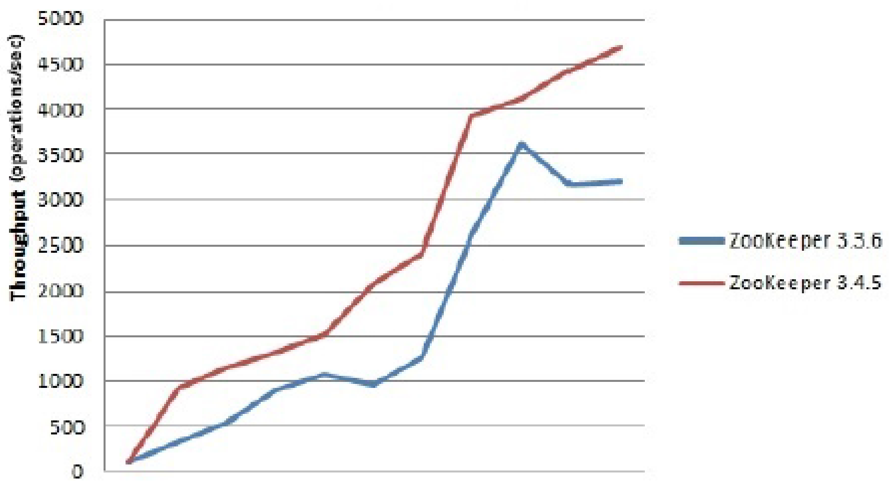


Figure 5.1: Throughput Comparison of Different Version of ZooKeeper

which version bestows better performance. Figures 5.1 and 5.2 demonstrate latency and throughput comparisons for different versions of ZooKeeeper correspondingly.

Figure 5.1, shows that ZooKeeper 3.3.6 takes approximately `31 seconds` to complete the test, while ZooKeeper 3.4.5 (default version) takes only `21 seconds` to finish the test. Also, if throughput is considered, maximum throughput achieved for ZooKeeper 3.3.6 is approximately `3100 events per second`, but for ZooKeeper 3.4.5 maximum throughput accomplished has crossed `4500 events per second`. Figures 5.1 and 5.2 reveal that ZooKeeper 3.4.5 is the most stable version and yields better performance, so it has been used for executing the entire test suite.

The machine that has been used to evaluate the rest of the test suite is an Intel Core i5-3210M processor. It is a quad-core processor machine having 6 gigabytes of RAM. Furthermore, only 4 gigabytes of RAM have been used to execute the tests.

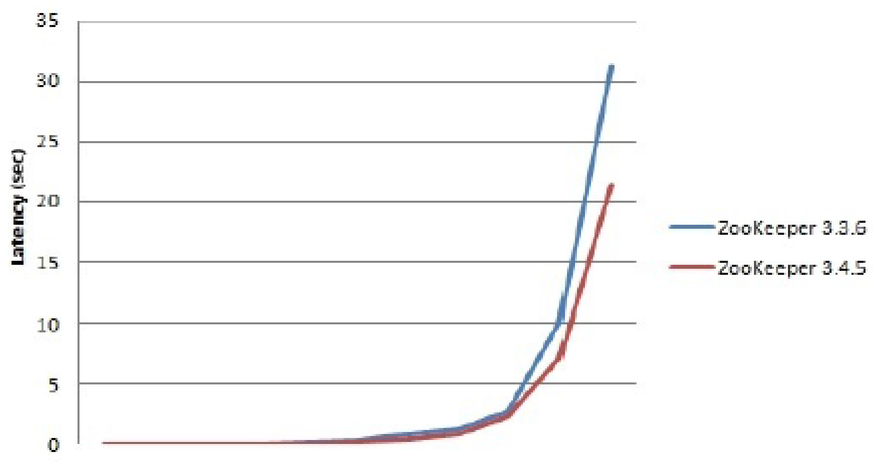


Figure 5.2: Latency Comparison of Different Version of ZooKeeper

5.2.2 ZooKeeper Performance with Varying API Version

This section will illustrate the performance of ZooKeeper under various loads and configurations via a test that involved the creation of `10,000` ephemeral znodes, and writing 300 bytes of data. This test was performed twice. The first run used the synchronous version of the ZooKeeper's [16]client API library [27]. On the second run, the same was performed using the asynchronous version of the API. Results show that the asynchronous version has much higher performance than the synchronous version due to its flexibility of keeping multiple outstanding transactions. Both ramdisk and traditional hard disk were separately used as the data directory. The preliminary step was to connect to a ZooKeeper server, create a specified number of znodes and write data to them. This whole process was timed in order to gather results. The results were stored in CSV files which have been used to plot the graphs. The test was repeated twice. First measurement was taken with the default implementation of ZooKeeper, while the second measurement was obtained with the modified version of ZooKeeper.

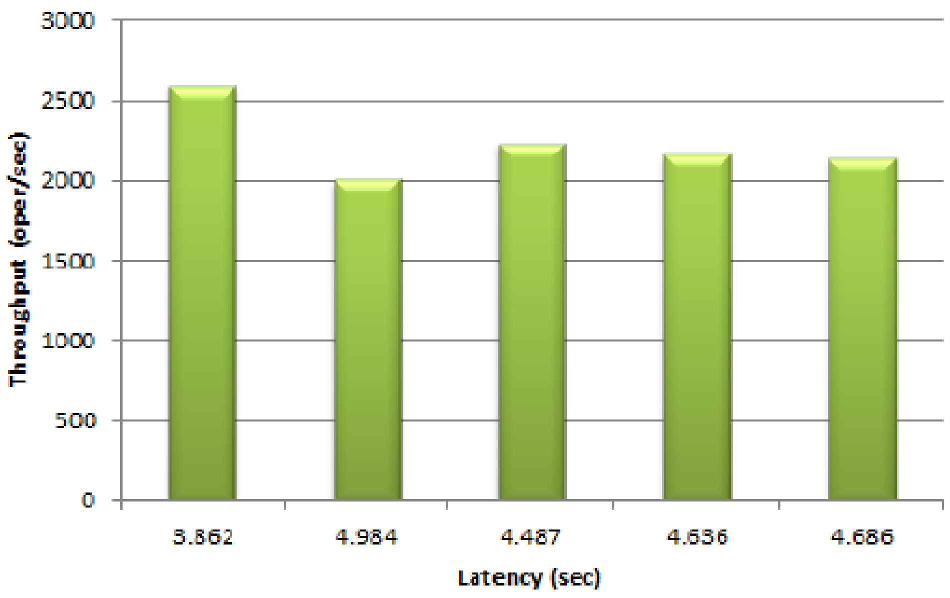


Figure 5.3: Performance Test in ZooKeeper Sync mode on RAM

Figure 5.3 shows the performance of the default version of ZooKeeper using synchronous API. Adopting the same version of API [27], i.e. synchronous, Figure 5.4 exposes the performance of the modified version of ZooKeeper.

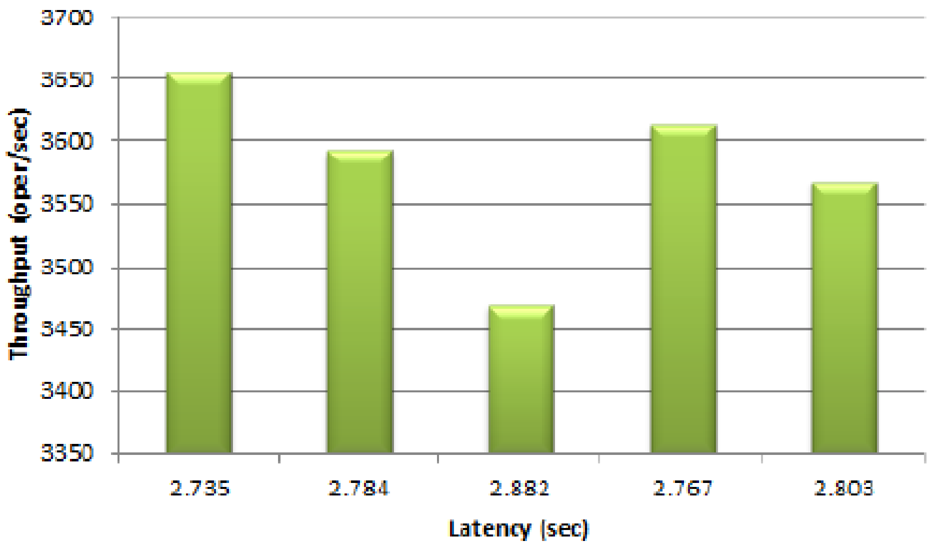


Figure 5.4: Performance Test with modified version of ZooKeeper Sync mode on RAM

The change in performance can be seen by analysing Figures 5.3 and 5.4. For the unmodified version of ZooKeeper, the average latency has crossed `4 seconds` and the maximum throughput achieved is around `2500 operations per second`. Regarding the modified version of ZooKeeper, the throughput has crossed `3600 events per second` and the latency on average is `2.8 seconds`. Figures 5.5 and 5.6 demonstrate the performance of ZooKeeper that uses the asynchronous version of the API. All other tests utilize only the synchronous version of the ZooKeeper API.

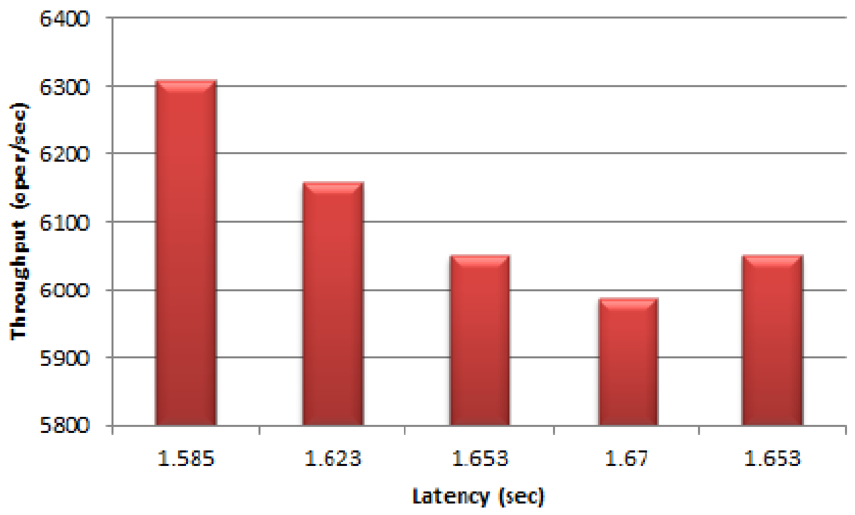


Figure 5.5: Performance Test in ZooKeeper Async mode on RAM

5.2.3 ZooKeeper Performance with Queues

This section describes the test that corresponds to a familiar problem termed the Producer-Consumer problem. In this test, the problem has been implemented using queues along with the ZooKeeper API. The `create()` method acts as a Producer and creates the specified number of znodes, and the `delete()` method acts as a Consumer deleting the znodes created. This test would again justify replacing linked blocking queues by concurrent linked queues. This test was also conducted by utilizing both linked blocking queues and concurrent linked queues one after another and was executed with both the default ZooKeeper as well as the modified version of ZooKeeper.

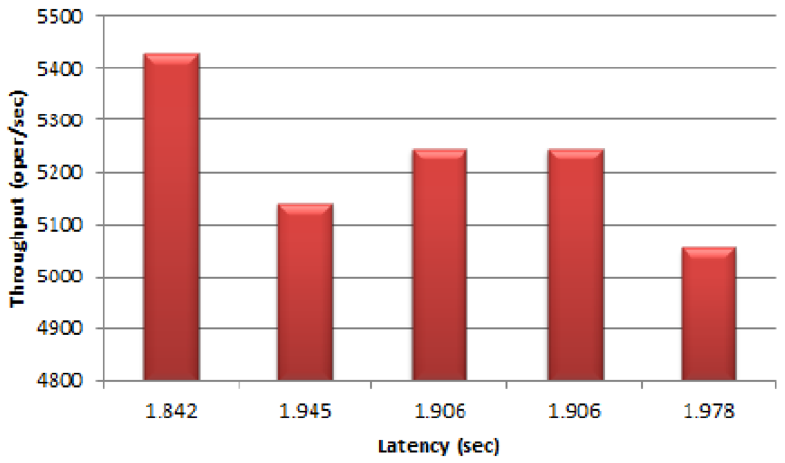


Figure 5.6: Performance Test with Modified Version of ZooKeeper Async mode on RAM

Figures 5.7 and 5.8 demonstrate the performance for the producer-consumer problem using the unmodified version of ZooKeeper. These graphs also show statistics based on the type of the queue used. For instance, Figure 5.7 shows the performance aspect for linked blocking queues, and performance aspects for concurrent linked queues are shown in Figure 5.8. Even if the default version is being used, then also concurrent linked queues show better performance than linked blocking queues. The average latency and throughput for the case of linked blocking queues is `3.5 seconds` and `2900 nodes per second` respectively, as depicted in Figure 5.7. Similarly, when the test was executed with concurrent linked queues, the average latency comes out to approximately `3.2 seconds` and throughput around `3100`, expressed in Figure 5.8.

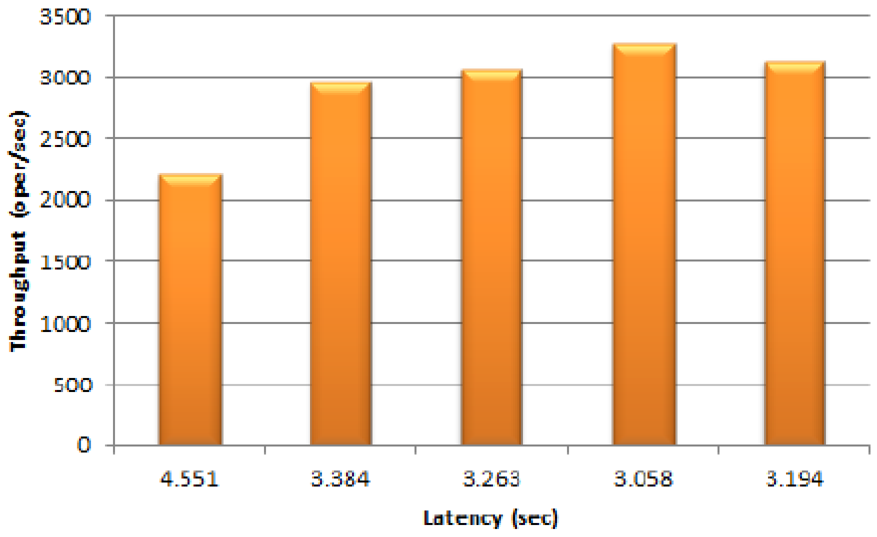


Figure 5.7: Performance of the Producer-Consumer for Linked Blocking Queue with the Default Version of ZooKeeper

This Producer-Consumer test was executed again with the modified version of ZooKeeper. As expected, it led to better results than the previous tests carried out with the default version of ZooKeeper, for both the cases of queues, but the concurrent linked queue still performed much better than the linked blocking queue, as demonstrated in Figures 5.9 and 5.10. On average the latency and throughput for linked blocking queue in this case comes out to be approximately `3.4 seconds` and `3000 nodes per second` respectively. For concurrent linked queues the results are `2.5 seconds` and `3900 nodes per second`.

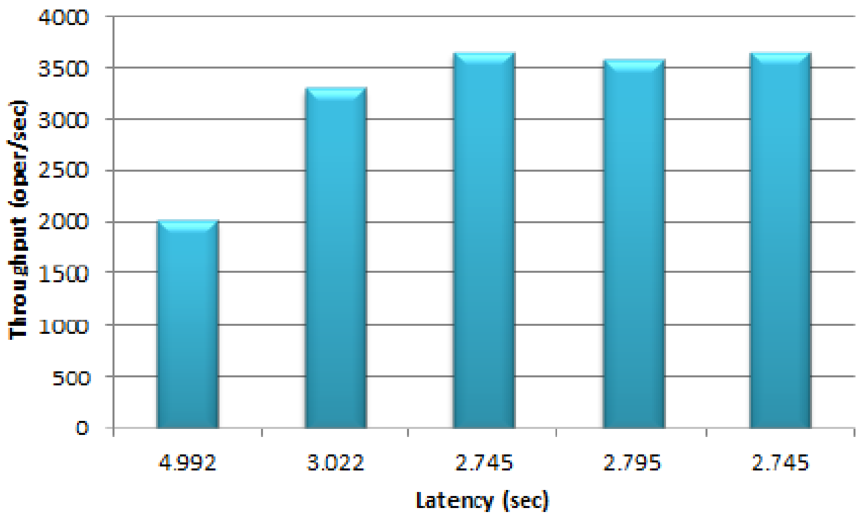


Figure 5.8: Performance of the Producer-Consumer for Concurrent Linked Queue with the Default Version of ZooKeeper

5.2.4 Behaviour of ZooKeeper for Producer-Consumer using Threads

Another implementation of the same Producer-Consumer problem was tested using the concept of threads that will be described in this section. Producer-Consumer was implemented as two independent threads and creation and deletion of znodes was respectively done by `create()` and `delete()` methods. This test was executed with both the default as well as the modified version of ZooKeeper. The results are expressed in Figures 5.11 and 5.12 respectively. The test with the modified version of ZooKeeper shows better performance than the default version. As this test involved

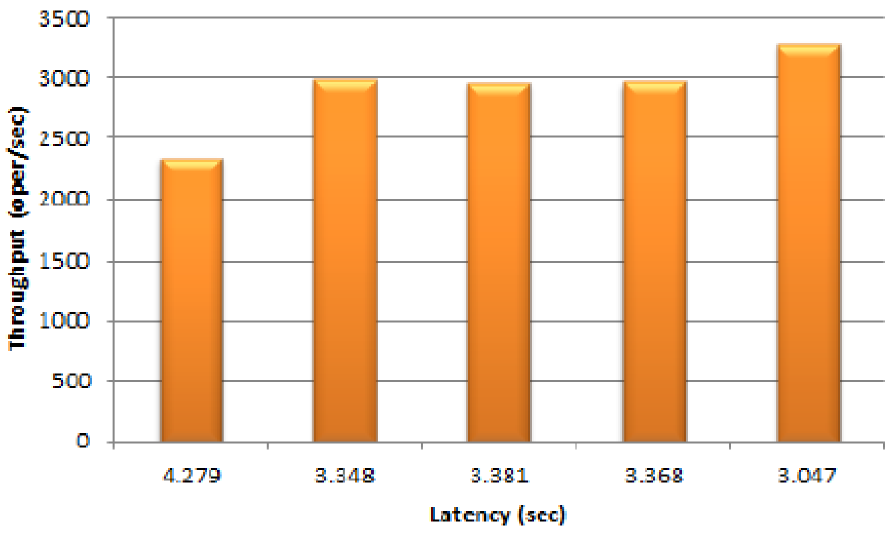


Figure 5.9: Performance of the Producer-Consumer using Linked Blocking Queue with Modified Version of ZooKeeper

threads, the complete process is taken into account. For the modified version, the average latency and throughput are roughly `2.7 seconds` and `3700 nodes per second` respectively, while in the default case they are approximately `2.95 seconds` and `3300 nodes per second`.

5.2.5 Lower Bound on ZooKeeper's Write Performance

A test was designed to obtain a lower bound on ZooKeeper. For this test, the conventional file writing performance was tested in contrast to the ZooKeeper's write performance. The test was made to execute with both the default and modified version of ZooKeeper. The results are exposed in

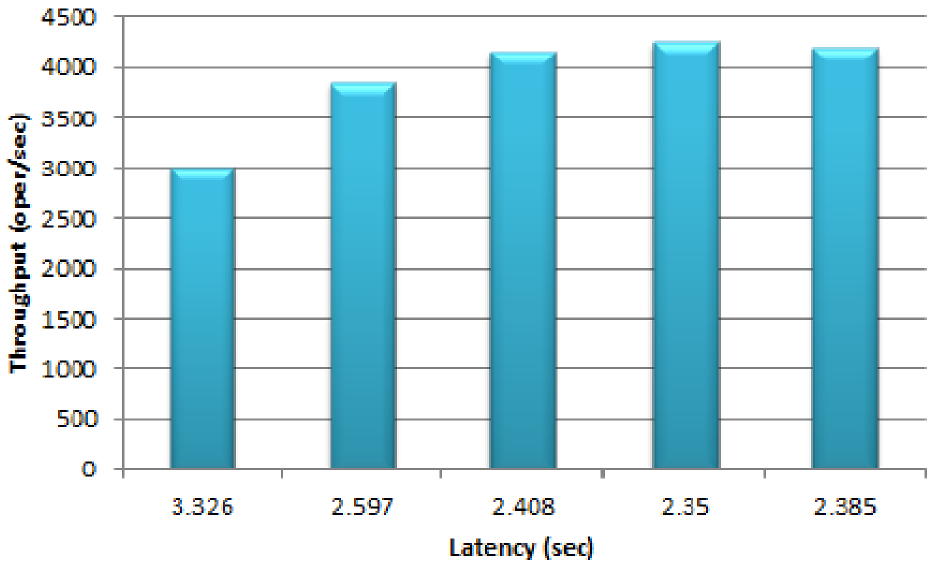


Figure 5.10: Performance of the Producer-Consumer using Concurrent Linked Queue with Modified Version of ZooKeeper

Figures 5.13 and 5.14. Conventional file write performance means time taken to write data to disk i.e I/O. On the contrary, ZooKeeper write performance can be termed as time taken by ZooKeeper to write data (same size) to a particular *znode*.

Figure 5.13 demonstrates the comparison time taken with the help of labels for both the conventional file write and ZooKeeper write time taking into account the default implementation. Similarly, Figure 5.14 also shows comparison of file-write time with ZooKeeper-write time using a modified version of ZooKeeper. Two reasons that could be responsible for the slight difference in file write performance bestowed in Figures 5.13 and 5.14 could be due to the possibility that a higher portion of CPU was given to that process during the second run. Secondly, there could have been some caching as well. The implication to this experiment was that the modified version of ZooKeeper is trying to cover the difference in its write performance in contrast to the conventional file write I/O. But, there still exists a performance gap between ZooKeeper and file write I/O (lesser than that with the default version of Zookeeper) which is subject to future work.

5.2.6 Performance Comparison for Varying Request Type

This last section involves a test which corresponded to the number of requests based on their type. ZooKeeper is intended for read intensive work loads as mentioned in the official documentation [13]. An experiment was performed in order to investigate how ZooKeeper behaves if the number of write requests are more than read requests, and vice versa. This test was experimented with the modified version of ZooKeeper. The output for the test is gathered and expressed in Figure 5.15. When the number of read requests was equal to or greater than the write requests, then the output exposed is very much as expected i.e. the latency tends to become constant

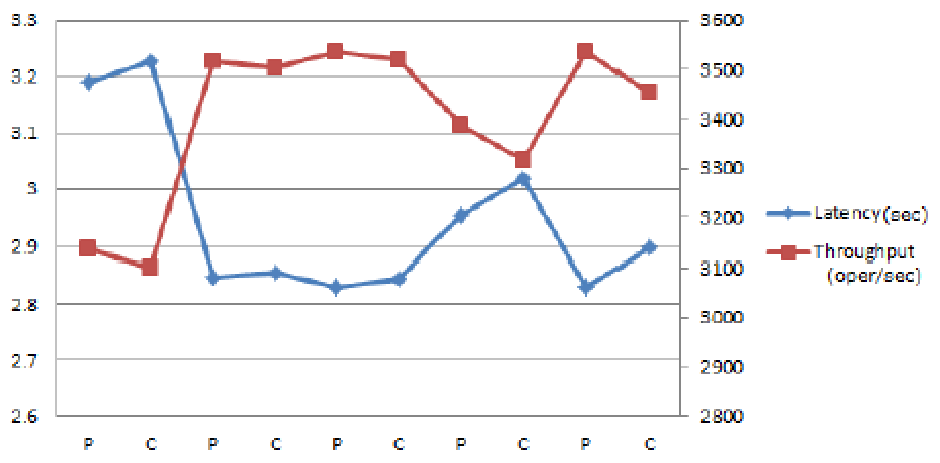


Figure 5.11: Performance of the Producer-Consumer using Threads with the Default Version of ZooKeeper

near 5.7 seconds. On the other hand, when the number of write requests was greater than the number of read requests, then an unexpected behavior is seen with latency. The magnitude for latency tend to follow an ideal pattern of being small in magnitude for most operations, but the magnitude starts rising at the end of request processing.

The reason for this abrupt behavior can just be conjectured after analysing all the other results described in this section. The reason may be the different time taken by different methods.

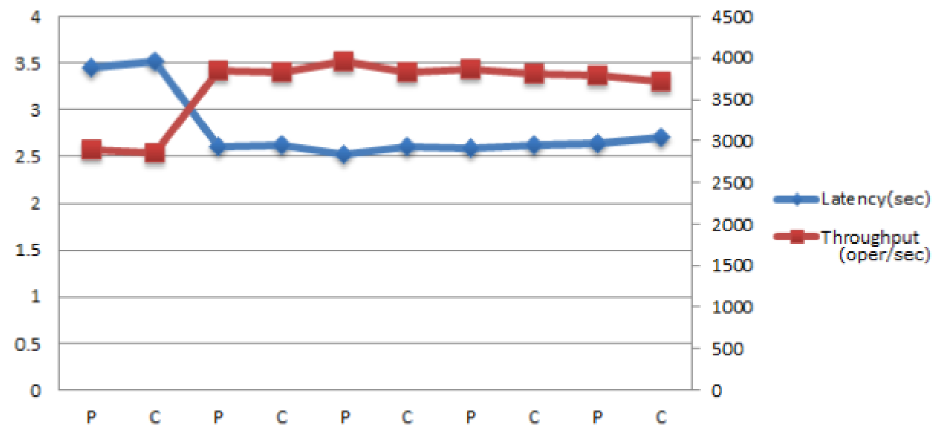


Figure 5.12: Performance of the Producer-Consumer using Threads with Modified Version of ZooKeeper

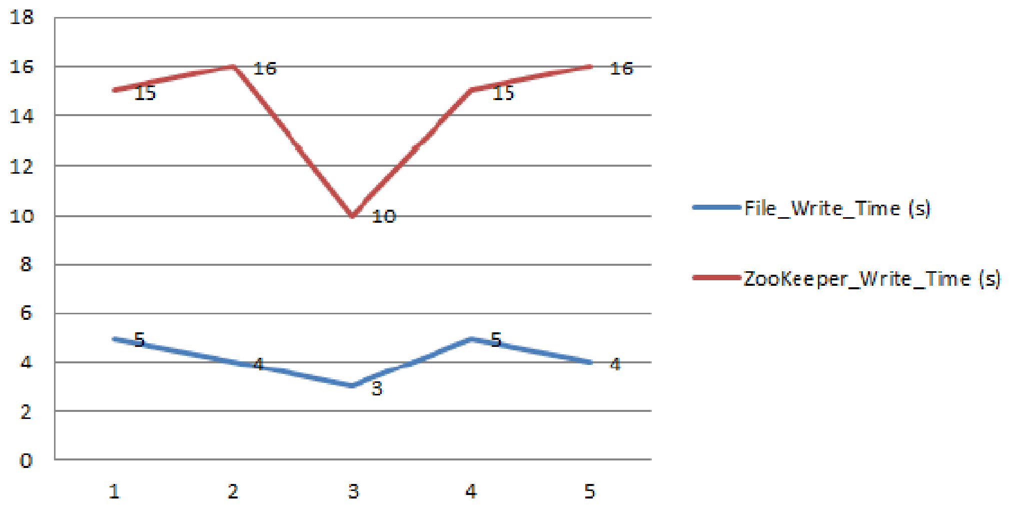


Figure 5.13: Comparison of File-Write Time versus ZooKeeper-Write Time using the Default Version of ZooKeeper

For instance, methods such as `create()` and `set()` used for issuing write

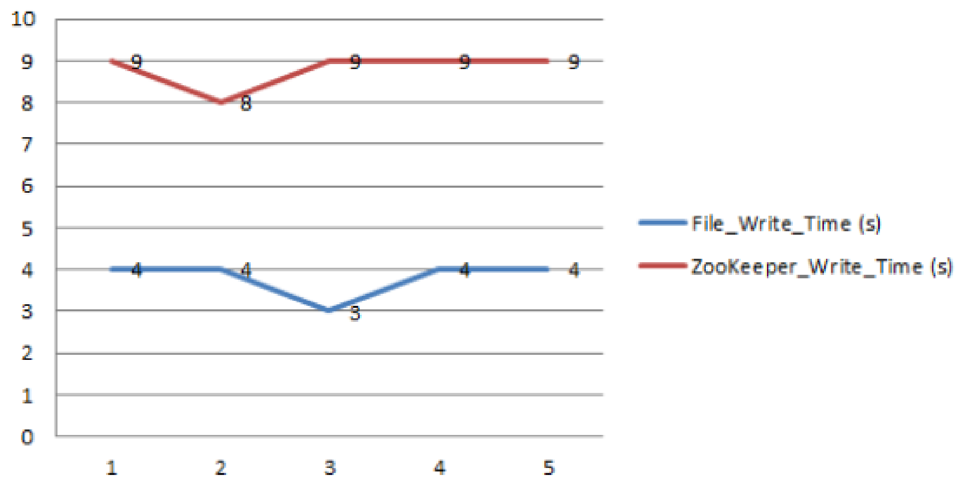


Figure 5.14: Comparison of File-Write Time versus ZooKeeper-Write Time using a Modified Version of ZooKeeper

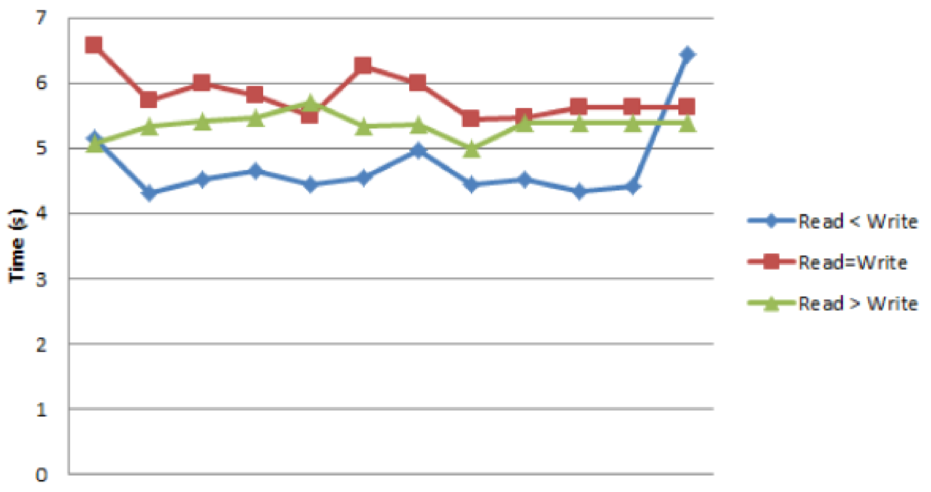


Figure 5.15: Time taken

requests might take less time than methods used for read requests such as `getData()` and `getACL()`. This is possible as complete data tree needs to be traversed for serving read requests.

Lastly, ZooKeeper uses wait-free characteristics and all the data structures being used are also thread safe as well. So, guarantees are that none of the changes to the server would result in race conditions or inconsistencies among multiple servers.

Chapter 6

Conclusion and Outlook

The main focus of this research was to improve the general performance of ZooKeeper. This report has shown the successful implementation of ZooKeeper accomplishing all stated goals, especially performance enhancement. The goals have been achieved by modifying the existing implementation of ZooKeeper. After some background research, code review, and understanding the control work flow of ZooKeeper it has been found that the request processing pipeline is the area that involves a scope of improvement. Thus, modifying the default marshalling and demarshalling mechanisms used by request processors have lead to performance enhancement.

The default marshalling technique used is the default serialization and the modified implementation involves externalization. The default implementation with serialization as a recursive algorithm, requires that all the objects that can be reached through the target object that can be reached through the target object including instance variables are also serialized. This overhead is a big bottleneck for performance. Also serialization uses reflection to determine values and while persisting the actual object it also includes metadata in the stream which degrades the performance. On the other hand, externalization provides the user full control over what to serialize (only class identifier is persisted and no metadata). Chapter 4 has demonstrated the performance impacts of the externalization over default serialization. The marshalling in case of externalization is 3:1 times faster than the default serialization. However, this performance comes at a cost, that custom marshalling and demarshalling methods need to be written.

Performance benefits have also been made by replacing the blocking queue implementations by concurrent linked queues. The methods such as `put()` and `take()` belong to linked blocking queues. The methods used by concurrent linked queues such as offer() and poll() are wait-free. Chapter 4 demonstrates on average the `offer()` method is 1.3 times faster than the `put()` method in inserting requests on the queue, and similarly the poll() method is 1.6 times faster than the take() method in deleting requests from the queue. Lastly, all the alterations made to the default ZooKeeper implementation for performance enhancement have been evaluated in Chapter 5 ensuring proper functioning of ZooKeeper.

Bibliography

[1] M. Burrows, "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, 2006.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "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, 2010.

[3] Amazon, "Amazon simple queue service," June 2013. `http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/Welcome.html`.

[4] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, p. 2, ACM, 2008.

[5] A. Medeiros, "Zookeepers atomic broadcast protocol: Theory and practice," 2012.

[6] Igvita, "Distributed coordination with zookeeper," March 2013. `http://www.igvita.com/2010/04/30/distributed-coordination-with-zookeeper/`.

[7] D. S. Laboratory, "Distributed coordination / zookeeper," July 2013. `http://lsrwww.epfl.ch/page-93925-en.html`.

[8] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275, ACM, 1996.

[9] Oracle, "Linked blocking queue," April 2013. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html`.

[10] Oracle, "Concurrent linked queue," April 2013. `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`.

[11] J. Lobby, "Concurrency: Blocking queues and you," June 2013. `http://www.javalobby.org/java/forums/t16278.html`.

[12] H. Scalability, "Zookeeper - a reliable, scalable distributed coordination system," September 2012. `http://highscalability.com/blog/2008/7/15/`

zookeeper-a-reliable-scalable-distributed-coordination-syste.html.

[13] Apache, "Apache zookeeper," September 2012. http://zookeeper.apache.org/.

[14] A. W. Services, "Amazon sqs pricing," December 2012. http://aws.amazon.com/sqs/pricing/.

[15] Hortonworks, "Apache hadoop: Projects and distributions," October 2012. http://hortonworks.com/hadoop/.

[16] Apache, "Apache hadoop," May 2013. http://wiki.apache.org/hadoop/.

[17] W. J. Iliff, "A zookeeper training project.," *Parks and Recreation*, 1972.

[18] I. Andreea-Alexandra, "Creating a cloud engine using zookeeper.," *UNIVERSITATEA ALEXANDRU IOAN CUZA IAI FACULTATEA DE INFORMATIC*, 2012.

[19] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, *et al.*, "Apache hadoop goes realtime at facebook," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1071–1080, ACM, 2011.

[20] CowTalk, "Replacing standard jdk serialization using jackson (json/smile), java.io.externalizable," April 2013. http://www.cowtowncoder.com/blog/blog.html.

[21] T. J. C. Monkey, "Java serialization: Using serializable and externalizable and performance considerations," March 2013. http://thejavacodemonkey.blogspot.ca/2010/08/java-serialization-using-serializable.html.

[22] Oreilly, *Hadoop: The Definitive Guide, 3rd Edition.* January 2013. net.pku.edu.cn/.../2009-Book-Hadoop%20the%20Definitive%20Guide.pdf.

[23] Oreilly, *Java RMI.* April 2013. http://oreilly.com/catalog/javarmi/chapter/ch10.html#footnote-1.

[24] Javarevisited, "Jusfortechies," May 2013. http://www.jusfortechies.com/java/core-java/externalization.php.

[25] G. Explains, "Diff between externalizable and serializable in java," May 2013. http://geekexplains.blogspot.ca/2008/06/diff-between-externalizable-and.html.

[26] Javarevisited, "Difference between serializable and externalizable in java serialization," April 2013. http://javarevisited.blogspot.ca/2012/01/serializable-externalizable-in-java.html.

[27] A. ZooKeeper, “Zookeeper 3.4.5 api,” November 2012. `http://zookeeper.apache.org/doc/r3.4.5/api/`.

[28] Apache, “/[apache-svn]/zookeeper,” January 2013. `http://svn.apache.org/viewvc/zookeeper/`.

Appendix A

Code Samples

Some code listings based on usage for this project have been enlisted in this section, but the complete source code can be found at [28] svn repository.

A.1 Modified Marshalling Code

```
 public class DataTree implements Externalizable {
void serializeNode(ObjectOutputStream oa, StringBuilder path)
                pathString = path.toString();
       node = getNode(pathString);
       if (node == null) {
           return;
       }
       String children[] = null;
       synchronized (node) {
       scount++;
```

```
        DataTree d=new DataTree();
        d.writeExternal(oa);
        Set<String> childs = node.getChildren();
        if (childs != null) {
        children = childs.toArray(new String[childs.size()]);
          }
        }
        path.append('/');
        int off = path.length();
        if (children != null) {
        for (String child : children) {
                // since this is single buffer being resused
                // we need
                // to truncate the previous bytes of string.
                path.delete(off, Integer.MAX_VALUE);
                path.append(child);
                serializeNode(oa, path);
            }
        }
    }
    int scount;
    public boolean initialized = false;
  private void deserializeList(Map<Long, List<ACL>> longKeyMap,
              ObjectInputStream ia) throws IOException {
       DataTree d=new DataTree();
       try {
                d.readExternal(ia);
        } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
     while (map > 0) {
     if (aclIndex < val_list) {
          aclIndex = val_list;
           }
        List<ACL> aclList = new ArrayList<ACL>();
        ACL acl = new ACL();
        aclList.add(acl);
        longKeyMap.put(val_list, aclList);
        aclKeyMap.put(aclList, val_list);
        map--;
        }
    }
private synchronized void serializeList(Map<Long,
         List<ACL>> longKeyMap,ObjectOutputStream oa)
         {
       DataTree d=new DataTree();
       map=longKeyMap.size();
       d.writeExternal(oa);
       Set<Map.Entry<Long, List<ACL>>> set =
         longKeyMap.entrySet();
       for (Map.Entry<Long, List<ACL>> val : set) {
                val_list=val.getKey();
                d.writeExternal(oa);
```

```
            List<ACL> aclList = val.getValue();
           d.writeExternal(oa);
        }
    }
public void serialize(ObjectOutputStream oa, String tag)
                               throws IOException {
      scount = 0;
      serializeList(longKeyMap, oa);
      serializeNode(oa, new StringBuilder(""));
      DataTree d=new DataTree();
      if (root != null) {
       d.writeExternal(oa);
      }
    }
     String pathString="/";
     DataNode node;
     long val_list;
     int map;
     @Override
         public void readExternal(ObjectInput in)
                               throws IOException,
                       ClassNotFoundException {
              // TODO Auto-generated method stub

     pathString=in.readUTF();
     node=(DataNode) in.readObject();
     map=in.readInt();
```

```
      val_list=in.readLong();
    }
        @Override
        public void writeExternal(ObjectOutput out)
                        throws IOException {
                // TODO Auto-generated method stub
         out.writeUTF(pathString);
         out.writeObject(node);
         out.writeInt(map);
         out.writeLong(val_list);
        }
    public void deserialize(ObjectInputStream ia, String tag)
                        throws IOException {
     deserializeList(longKeyMap, ia);
     nodes.clear();
     DataTree d1=new DataTree();
     try {
                        d1.readExternal(ia);
                } catch (ClassNotFoundException e2) {
                        // TODO Auto-generated catch block
                        e2.printStackTrace();
        }
        while (!pathString.equals("/"))
        {
      DataNode node = new DataNode();
      DataTree d=new DataTree();
      try {
```

```
                        d.readExternal(ia);
                } catch (ClassNotFoundException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                }
      nodes.put(pathString, node);
      int lastSlash = pathString.lastIndexOf('/');
      if (lastSlash == -1) {
      root = node;
       } else {
     String parentPath = pathString.substring(0, lastSlash);
     node.parent = nodes.get(parentPath);
    node.parent.addChild(pathString.substring(lastSlash + 1));
      long eowner = node.stat.getEphemeralOwner();
      if (eowner != 0) {
          HashSet<String> list = ephemerals.get(eowner);
           if (list == null) {
               list = new HashSet<String>();
               ephemerals.put(eowner, list);
                }
               list.add(pathString);
                }
            }
    try {
                d.readExternal(ia);
        } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
      }
        nodes.put("/", root);
        // we are done with deserializing the
        // the datatree
        // update the quotas - create path trie
        // and also update the stat nodes
        setupQuota();
    }
```

```
public class SerializeUtils implements Externalizable {

     public SerializeUtils() {
                // TODO Auto-generated constructor stub
        }
public static void deserializeSnapshot(DataTree dt,
           ObjectInputStream ia, Map<Long, Integer> sessions)
               throws IOException {
   SerializeUtils s1=new SerializeUtils();
   try {
        s1.readExternal(ia);
                } catch (ClassNotFoundException e) {
                        // TODO Auto-generated catch block
        e.printStackTrace();
                }
   while (count > 0) {
    try {
     s1.readExternal(ia);
                  } catch (ClassNotFoundException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
                  }
       sessions.put(id, to)
        count--;
        }
}
public static void serializeSnapshot(DataTree dt,
        ObjectOutputStream oa, Map<Long, Integer> sessions)
                          throws IOException {
 HashMap<Long, Integer> sessSnap =
                  new HashMap<Long, Integer>(sessions);
 count=sessSnap.size();
 SerializeUtils s=new SerializeUtils();
 s.writeExternal(oa);
 for (Entry<Long, Integer> entry : sessSnap.entrySet()) {
        id=entry.getKey().longValue();
        to=entry.getValue().intValue();
        s.writeExternal(oa);
    }
 @Override
        public void readExternal(ObjectInput in)
                                        throws IOException {
                  // TODO Auto-generated method stub
        count=in.readInt();
```

```
                to=in.readInt();
                id=in.readLong();
    }
 @Override
        public void writeExternal(ObjectOutput out)
                                                throws IOException {
                // TODO Auto-generated method stub
                out.writeInt(count);
                out.writeInt(to);
                out.writeLong(id);
        }
}
```

```
public class FileSnap implements SnapShot,Externalizable {
    public void deserialize(DataTree dt,
                        Map<Long, Integer> sessions,
            ObjectInputStream ia) throws IOException {
         SerializeUtils.deserializeSnapshot(dt,ia,sessions);
    }
    public void serialize(DataTree dt,Map<Long,
                                        Integer> sessions,
            ObjectOutputStream oa) throws IOException {
          SerializeUtils.serializeSnapshot(dt,oa,sessions);
    }
    String path="/";
 @Override
    public void readExternal(ObjectInput in)
                                throws IOException {
```

```
                // TODO Auto-generated method stub
     path=in.readUTF();
    }
 @Override
        public void writeExternal(ObjectOutput out)
                                                throws IOException {
                // TODO Auto-generated method stub
                        out.writeUTF(path);
        }
```

A.2 Code to Demonstrate the Usage of Queues with ZooKeeper

```
public class PrepRequestProcessor extends Thread
                                implements RequestProcessor {
 RequestProcessor nextProcessor;
    ZooKeeperServer zks;
    ConcurrentLinkedQueue<Request> submittedRequests =
                 new ConcurrnetLinkedQueue<Request>();
    public PrepRequestProcessor(ZooKeeperServer zks,
           RequestProcessor nextProcessor) {
                super("ProcessThread(sid:" + zks.getServerId()
              + "␣cport:" + zks.getClientPort() + "):");
     this.nextProcessor = nextProcessor;
     this.zks = zks;
  }
```

```
public void run() {
                try {
            while (true) {
                Request request = submittedRequests.poll();
                        if (Request.requestOfDeath == request) {
                            break;
                        }
                pRequest(request);
                } catch (Exception e) {
             LOG.error("Unexpected␣exception", e);
        }
        LOG.info("PrepRequestProcessor␣exited␣loop!");
  }
```

Vita

Candidate's full name: Chandan Bagai
University attended:
Bachelor of Engineering Computer Science 2012
Chitkara University, India