ParKazoo: A partitioned scheme for scaling ZooKeeper

by

Arjun Naik

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MS DISTRIBUTED SYSTEMS ENGINEERING
INFORMATICS FACULTY
TU DRESDEN

JANUARY, 2015

ParKazoo: A partitioned scheme for scaling ZooKeeper

Abstract

With the advent of cloud computing it has become very common for large applications to be built using distributed platforms. While distributed systems offer a cost advantage they also create new challenges. Coordination and synchronization mechanisms which are common on single computers need to be reinvented for a distributed environment. Most distributed applications require a central store for coordination tasks like synchronization, configuration and task queues. Coordination services are used for this purpose and should have the property of being performant, fault-tolerant and scalable. ZooKeeper is distributed co-ordination service which is used in many distributed applications. It is based on the ZAB protocol which ensures consistency and total-ordering of the data updates on the data. However this protocol is limited by the overhead for quorum between the participant nodes. As the size of the cluster grows larger the overhead increases. But to increase the throughput of the cluster the size of the cluster should be increased by adding more nodes, to support more number of connected clients. We attempt to solve the quorum overhead problem by partitioning the data namespace. Multiple sets of ZooKeeper clusters are used so that each one is reponsible for a part of the complete namespace. This way the load of the data management is partitioned between clusters and performance of the ensemsble can scale as the load increases.

Acknowledgments

I would like to thank everyone who helped me during every step of my thesis work, without whom this undertaking would not have been possible. I would like to thank Prof. Dr. Christof Fetzer and the Systems Engineering Chair for giving me this opportunity to work on scaling the throughput of ZooKeeper. My gratitude to all the members of the Systems Engineering Chair who provided feedback and criticism to improve my work.

Secondly I would like to thank my supervisor Dipl. Inf. Katja Tietze for her guidance and supervision along the way. Her guidance, support and encouragement throughout my thesis helped shaped it. She guided me through every step of the process and pointed me in the right direction when I needed it.

And finally I would like to thank my parents and my friends in Dresden for their encouragement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Distributed Coordination Services are a critical component of many distributed applications. There have been several implementations of such algorithms in the past. Google published a paper on an internal service called Chubby [Burrows, 2006]. This sparked an interest in using similar services in other distributed application. ZooKeeper [Hunt et al., 2010] was developed by Yahoo to replicate some the functionality of Chubby. ZooKeeper was implemented in Java and is today maintained by the Apache Software Foundation. It is also a component in other distributed frameworks like Hadoop [White, 2009] and Cassandra [Lakshman & Malik, 2010]. There have been other implementations of coordination services like Doozer [Ketelsen et al., 2015] and etcd [CoreOS, 2015]. These projects are relatively new and are based on the Raft [Ongaro & Ousterhout, 2013] consensus protocol which is not extensively deployed in working systems.

The scale and size of distributed applications increases everyday. Coordination services like ZooKeeper need to keep up with this increasing traffic demands. The throughput rate of ZooKeeper is limited by three main factors:

1. Local Disk Throughput
2. Network Speed
3. Local Processing Capacity.

For every write operation the ZooKeeper cluster requires a consensus between a quorum number of member nodes which agree on the proposed update, process it and finally acknowledge that they have committed the change. This slows down the response time. However the number of connected clients and the fault-tolerance of the cluster can be scaled only by increasing the number of servers in the ZooKeeper cluster, that is the number of nodes participating in the consensus. These competing but opposing forces makes scaling the throughput of ZooKeeper difficult.

We try to solve these problem by partitioning the data which would be normally held by a single cluster and distributing them among a number of ZooKeeper clusters. We also implement this solution completely on the client-side, so that it does not necessitate any modification to the the ZooKeeper source code. The client library is implemented in such a way that the consistency properties of ZooKeeper are also upheld.

At this point, it has to be mentioned that this solution is not an attempt to reinvent consensus algorithms. It attempts to solve a problem using existing time-tried technologies and concepts. ZooKeeper has been deployed in several large scale projects and there exists bindings to connect to ZooKeeper in several programming languages. There also exists a large body of documentation supporting the use and deployment of ZooKeeper. It's also based on Java which is a common choice for large applications. This solutions attempts to provide a solution for existing distributed applications which require higher throughput than what can be currently provided by the ordinary ZooKeeper cluster.

Chapter 2 discusses the related work around coordination services and consensus algorithms. It introduces some of the motivations and the intentions for the implementing ZooKeeper.

Chapter 3 introduces ZooKeeper which is the backbone of the proposed solution. It explains important concepts like how data is stored in ZooKeeper and the common operations.

Chapter 4 explains the implementation of ParKazoo. All of operations which are implemented are explained in detail.

In *Chapter 5* the tests which are used to confirm the functionality of ParKazoo are explained. Some tests from Kazoo which are reused to confirm the functionality of ParKazoo are also discussed.

Chapter 6 is the Evaluation chapter which discusses the results of testing the Throughput and the Latency of ParKazoo against the original ZooKeeper.

Chapter 7 describes some of the limitations in the proposed solution.

Chapter 8 proposes improvements which could be implemented but are out of the scope of the current thesis.

2
Related Work

2.1 Earlier and Related Systems

Camelot [Hastings, 1990] was an early software implementation of a service for the Mach operating system which proposed a locking service using transactions in a distributed environment. Using Camelot application developers could build *encapsulated*, *shared* and *recoverable* objects. This system does not restrict the size of the data being shared and it is also unique in that it is one of the first system which implemented the coordination and synchronization completely in software and could be run on different types of hardware.

Boxwood [MacCormick et al., 2004] presents a distributed storage medium with an emphasis on fault-tolerance. Boxwood differs from similar earlier system due its use of higher level data abstractions as synchronization primitives. Previous attempts to synchronize data across a distributed storage focused on lower level data abstractions

like disk blocks. Boxwood instead used a B-Tree [Skiena,] to store data.

dynamoDB [DeCandia et al., 2007] is a distributed Key-Value store which can be configured so that its *Consistency* and *Availability* properties can be adjusted according to the CAP Theorem [Gilbert & Lynch, 2002]. The objective of dynamoDB was to provide a highly available data store even in the presence of node failures. But this requires that the any data consistency issues must be handled by the developer on a per application policy basis. dynamoDB also uses Consistent Hashing [Karger et al., 1997] so as to reduce the amount of data migration and equally distribute the migration load in case of node failure.

2.2 Similar Systems

Any discussion of ZooKeeper is incomplete without mentioning Chubby [Burrows, 2006]. Chubby was developed to perform some of the same functions as ZooKeeper with a few minor exceptions. Chubby was initially designed to be a locking service which provided coarse-grained locks to its clients. The locks are advisory and are leased for a specified amount of time. Secondly, although Chubby delivers change events to clients asynchronously, the developers of Chubby also implemented client side caching to mitigate the effect of polling. Chubby exposes a file-system like data structure to its clients. Chubby is also a core component of several large-scale distributed applications like the Google File System [Ghemawat et al., 2003] and MapReduce [Dean & Ghemawat, 2008].

More recently Flavio Juqueira, one of the creators of ZooKeeper proposed a partitioned implementation [Junqueira, 2010] of ZooKeeper. This proposal is similar to our implementation However it also includes another component on the main ensemble which routes the requests from the clients to the destination clusters. The distribution of the znodes can also be dynamic, which means that the client which creates a znode can specify at runtime to which cluster the nodes gets mapped. This complicates the use of the system because the application developer needs to keep compute and map nodes to individual clusters. This proposal also relies on the assumption that the requests are always distributed uniformly across the tree. If there are hotspots which

develop in a part of the tree then that part of the tree has to be split and spread across multiple ensembles.

There was also an attempt to modify the ZooKeeper core [Biligiri et al., 2014] itself with multiple quorums. Instead of single leader in a quorum, every server has the chance to be the leader for a subset of the data. The author's argument is that this would lead to more uniform utilization of resources in case of hotspots.

2.3 Coordination Protocols

At the heart of every coordination service like ZooKeeper is a consensus algorithm. Paxos [Lamport, 2001] is the most famous and common consensus algorithm and modified versions of it have been implemented in Chubby [Burrows, 2006] and other services including ZooKeeper. Most consensus algorithm are based on the idea of State Machine Replication [Schneider, 1990].

The Chandra-Toueg Algorithm [Chandra & Toueg, 1996] is another consensus algorithm which is designed to handle Byzantine faults which is overlooked by Paxos. The biggest drawback of Paxos is that it is hard to understand and there have been attempts to simplify the algorithm [Chandra et al., 2007] [Lampson, 2001]. Raft [Ongaro & Ousterhout, 2013] is an attempt to create a consensus algorithm which is easy to understand and can be more easily used to build coordination services.

ZooKeeper uses the ZAB broadcast algorithm [Junqueira et al., 2011]. The ZAB algorithm ensures any state changes which are broadcast are received in the order they are broadcast. ZAB also is used in the election of the primary which processes the state change requests.

...ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client.

—Authors of ZooKeeper: Wait-free coordination for Internet-scale systems

3 Zookeeper

In the simplest of terms ZooKeeper is a distributed service with the goal to assist in coordination and synchronization between its clients.

3.1 ZooKeeper Architecture

Figure 3.1 depicts the architecture of a three server ZooKeeper cluster. There are three nodes*(Server 1, Server 2, Server 3)* which accept requests from clients to store, update and delete data. These nodes cooperate with each other and coordinate the writes so that they contain identical copies of data. These nodes are called as ZooKeeper servers. When a ZooKeeper server is started it has to be provided with an ID and also the addresses of the other ZooKeeper servers. All the servers should contain an identical copy of the ID and the address information.

The servers keep their respective datastores in memory which restricts the amount

of data which can be stored. The servers also keep a log of the transactions which were performed so that the data can be recovered in case of failures.

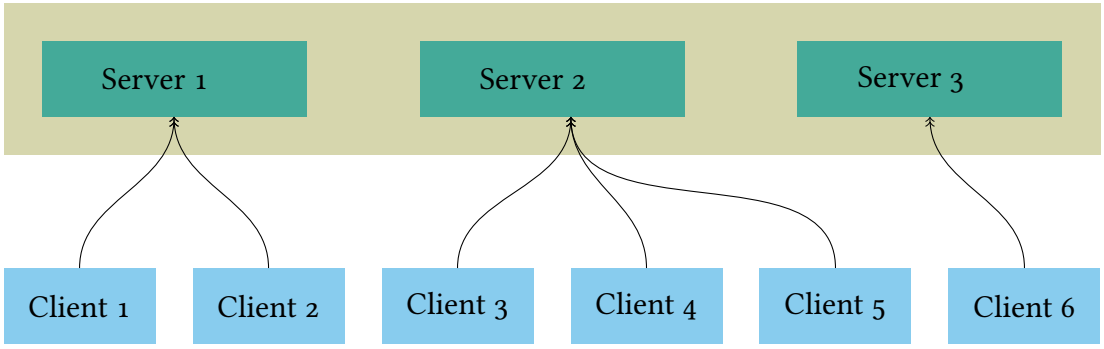


Figure 3.1: ZooKeeper Architecture

The ZooKeeper servers communicate with each other using a consensus protocol called ZAB [Junqueira et al., 2011] which is modeled on Paxos but is not entirely similar. When a request is received on the servers the request is forwarded to the leader which computes the necessary changes and then broadcasts it to the rest of the servers. When a majority of the servers respond with an acknowledgment the transaction can be committed and a response can be sent to the client.

The size of the quorum is the factor against which performance is measured. Three is the smallest number of servers required for a quorum. One failure is tolerated but both the other two servers must be always available. This setup generally gives the fastest throughput. A quorum needs three or more odd number of servers. Five servers is the next largest and seven and nine come after that. Quorums larger than these are generally not used in production.

For requests which do not modify the state of the server like data read requests, there is no need for consensus. All the servers contain identical copies of the data and any read request can be processed locally on the server.

The clients are initialized with the addresses of all or some of the addresses of the servers in the ZooKeeper cluster. The client then chooses one of the servers and tries

to connect to it through a TCP connection. If the connection cannot be made or the connection breaks during operation then the client attempts to connect to another server.

3.1.1 Client Sessions

When a ZooKeeper client connects to a ZooKeeper cluster this creates a session associated with that instance of the connection. Whenever the client loses the connection to the server it attempts to reconnect to another server in the cluster. If it cannot reconnect within a specified amount of time then the session associated with that client connection is deleted. When the client subsequently connects at some point in the future then a new session is created. Any locks which are held by a client are deleted when the session is lost. The server periodically sends a pulse to the client to check if it is still alive. If it does not receive an acknowledgment it resends the pulse again. Finally after a preconfigured timeout the client is considered to be lost and the session is deleted.

3.1.2 ZooKeeper Data Organization

ZooKeeper data is presented like a UNIX file system. There are no files and directories, only *znodes*. Every znode can have data associated with it and also have children. Other meta-data about the znode like the creation timestamp, the owner of the znode, the version of the data and any access-control information is also stored along with the data. Figure 3.2 shows a part of tree typical in ZooKeeper cluster. The circle at the top represents the root node and always has the path /. In this example the root node has two child nodes viz. */app1* and */app2*. The znode */app1* in turn has 3 children called *p_1*, *p_2* and *p_3*. These are leaf nodes and are represented as hexagons. The full path for znode *p_1* would be */app1/p_1*.

ZooKeeper has two types of znodes: *regular* and *ephemeral* znodes. Ephemeral znodes are present only as long at the client which created them is connected to the cluster. If the connection drops or the client crashes causing the connection to drop the the ephemeral znode is deleted. Ephemeral znodes also are different because they

cannot have child znodes. Attempting to create a child znode for an ephemeral znode throws an exception. Regular znodes are persisted across sessions. There are also sequential znodes which have a counter appended to their names. All sequential nodes which are siblings and have the same name have an incrementing value appended to the name as sequential znodes are created. Znodes can also be both sequential and ephemeral in which case the znodes have a counter value appended to the end of names and they are deleted when the client which created them is disconnected.

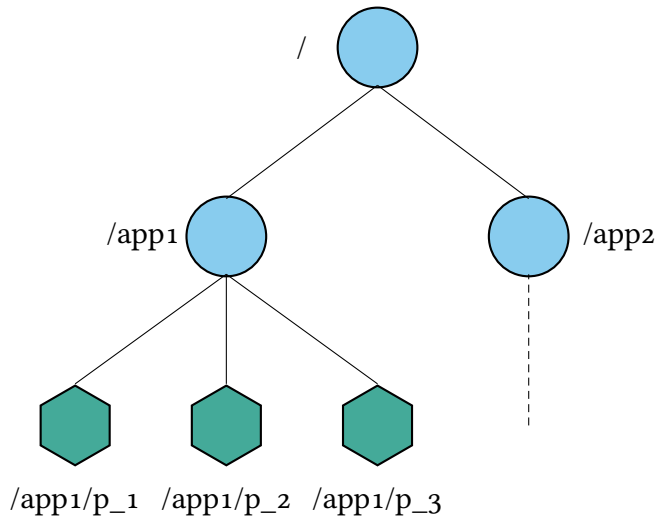


Figure 3.2: ZooKeeper Data Organization

3.1.3 ZooKeeper Guarantees

ZooKeeper has two ordering guarantees:

1. **Linearizable Writes:** All the update operations on the cluster can be serialized and they respect precedence.
2. **Client FIFO Order:** All operations from a client are executed in the order they were sent from the client.

ZooKeeper also has the following **Liveness** and **Durability** guarantees: As long as a quorum of servers are available the service can accept, process and commit requests. Any previously committed changes will be always present as long as a quorum of servers are available.

3.1.4 Cluster Reconfiguration

The current version of ZooKeeper cannot be reconfigured dynamically. To add more servers to a cluster the cluster has to be first stopped. The new servers have to be added to the configuration information of the cluster members and then finally the cluster has to be restarted. This is a manual process which is prone to errors and can cause down-time due to misconfiguration or split brain. This entails downtime. But it is also important that new servers be added dynamically to balance load and to replaces servers which have crashed. By utilizing some of the properties of the ZooKeeper system a reconfiguration method [Shraer et al., 2012] was proposed which can rebalance the clients dynamically and migrate the data without shutting down the cluster.

3.1.5 Why not a Distributed Database?

At a first glance, it may appear that a Distributed Database like Cassandra or BigTable, HBase may solve this coordination problem. However these databases are actually built using a distributed co-ordination service like ZooKeeper. ZooKeeper which is maintained by the Apache Foundation is used as a component in Cassandra. Similarly Chubby which was developed as a co-ordination service by Google is used in GFS and BigTable.

3.2 Kazoo Library

Since the connections are made through TCP there are several languages which have libraries to communicate with ZooKeeper. The Python [Van Rossum & Drake, 2002] library which is provided with the ZooKeeper package is quite small and contains limited features. Kazoo is a Python library which contains many features some of which

are borrowed from Netflix's Curator [Apache, 2015]. It is a pure Python implementation which means that it doesn't have any external C dependencies. It has support for older versions of ZooKeeper viz. 3.4 and 3.3. It has also support for evented IO through gevent. It also provides many higher order primitives like Watchers, Locks, Barriers. For these reason Kazoo is used as the base for the ParKazoo library. ParKazoo tries to maintain the same API as Kazoo. It can be in fact for most applications used as a drop-in replacement for Kazoo.

3.2.1 Library API

The Kazoo library provides the following functions. Some of them are identical to the functions in the Python library which is bundled along with ZooKeeper. They are:

Initialization of Kazoo

```
def __init__(hosts, timeout=10.0, randomize_hosts=True)
```

To initialize a *Kazoo* client object we need a list of server addresses to which the client can connect. This is provided as a Python *list* of *string*s. The *timeout* argument specifies how long before the client should throw an *Exception* when it cannot connect to the servers. The *randomize_hosts* parameter indicates if a random server from the list should be selected or the first server should be chosen. The client object also provides the following functionality.

1. Connectivity Checks
2. Client State Notifications
3. State Listeners

Sync Operations

```
def sync(path)
```

If a client wants to perform a read operation on a znode but it wants all the pending operations on that znode to be synchronized first, it calls the *sync* operation on that path. A sync is like a partial update operation. It does not require a consensus between the servers. But it must be sent to the leader and wait till all operation on that path are processed and committed.

Server Commands

```
def command(cmd='ruok')
```

ZooKeeper servers provide administration commands which can be used to obtain information about the configuration, the statistics of the performance of the cluster and to reset the statistics. There are also commands to list the currently connected clients and the watches present on the znodes. These commands are can be run from the client by using the *command()* operation.

Create Node

```
def create(path, value, ephemeral=True, sequential=True, make_path=
False, watch=None)
```

This call creates a znode in the tree structure. The *path* parameter indicates the path of the node, the *value* parameter should be a byte array containing the data to be stored. *make_path* indicates if the the ancestor znodes should also be created if they do not yet exist. The *watch* parameter leaves a watch on the node which invokes a callback function when the node is modified. The type of change event and the corresponding new data is passed as an argument to the callback function. The *sequential* and *ephemeral* flags indicate if the znode to be created should be of those corresponding types.

Read a Node

```
def get(path, watch=None)
```

Fetching the data is possible when the znode is present in the data tree. If the znode does not exist then an *Exception* indicating this is thrown. Also the *watch* argument can be used to leave a watch on the znode which is triggered when the znode is modified.

Deleting a Node

```
def delete(path, recursive=False)
```

If the node has children then the znode can be deleted by setting the *recursive* argument to *True*. Otherwise the delete call throws an exception which indicates that the znode is not empty.

Getting Children of a Node

```
def get_children(path, watch=None, include_data=False)
```

This call is used to retrieve all the the children of a znode. It can also set a watch which notifies the client when new znodes are created below the znode in question. By default, this call only returns the names of the child nodes. If the data of the child znodes is also required then the *include_data* parameter should be set to *True*.

These are only some of the operations which are provided by the Kazoo library. The ones listed above are used as primitives in the construction of the ParKazoo library. There are also corresponding asynchronous versions for each of the above. The difference between the synchronous and asynchronous versions of ZooKeeper operations is that the client blocks until the result of the operation is returned in case of synchronous operations. In the case of synchronous operations the function call instead returns a deferred object which can be used to notify the application when the result is ready. The is mostly used in single threaded evented programs.

3.2.2 Library Recipes

ZooKeeper was designed to be simple and does not directly implement complex synchronization and coordination mechanisms generally required by most applications.

It does not provide higher order constructs like locks, queues or counters. However these lower order primitives can be used to construct such functions. The Kazoo library includes in itself some of the higher level constructs which can be used by the application programmer. Some of them are listed below. There constructs are also later implemented in the ParKazoo library without much change from their original Kazoo implementations.

Barrier

Barriers are synchronization mechanisms for concurrent threads of execution. Multiple ZooKeeper clients which have a common barrier will all block on the barrier until a certain condition is met and the barrier is removed programmatically Double barriers can be used to synchronized the beginning and the end of a distributed asynchronous task. Double barriers differ from regular barrier because they wait for a predetermined number of clients to arrive at the barrier. When the number of clients is reached all of them cross the barrier together. When the barrier section is complete the same process occurs when the clients are leaving the barrier.

Counter

This is a race-free counter which allows multiple nodes to share a counter value. A Counter type object can be used directly with the plus and minus operators in the Python application. However only integer and float values are permitted.

Election

When multiple clients want to perform a distributed task which requires one of the to be a leader/primary they can use the Election construct. ZooKeeper will then elect one of the participating clients as the leader and this client will be notified through a callback function which was passed to the *Election* object when it was created. This construct also provides the functionality to query for all the client participating in the contest to be the leader.

Locks

Locks are the most commonly used synchronization mechanism in concurrent programs. Only one client can acquire the lock and hold it at any point of time. ZooKeeper does not provide mandatory locks. Advisory locks are implemented in the Kazoo library A lock can be created by a client on a specific znode. Subsequently any number of clients can compete to acquire the lock. If the lock cannot be acquired then the client blocks till the lock is acquired. The acquirement of the lock can be also canceled at any point before acquisition. Also the list of all the contenders for the lock can be obtained.

Kazoo also provides *Semaphore* objects which are similar to the Python Semaphores in the *threading* module [Foundation, 2015]. Semaphore are similar to locks but also allow multiple client to hold the Semaphore. The Kazoo Semaphore object can be initialized with an initial count which indicates the number of available resources. Whenever one of the clients acquires the semaphore the count is decremented. This way the number of clients accessing the distributed resource can be regulated. Once a client is done with a Semaphore then it is released. Only as many number of clients which will fit into the Semaphore are then notified. This prevents the ZooKeeper servers from being overwhelmed with requests to acquire the Semaphore.

Partitioner

This construct is used to divide the members of a set among the members of a party, such that every member receives zero or more items of and each item is only given to one member. And example of this would be a task queue. Here the tasks ares the items of the set and the clients of ZooKeeper are the party members which receive the tasks.

Party

Parties are used to keep a registry of nodes. Client nodes may enter or leave a Party which updates the membership registry of the party.

Queue

Kazoo supports simple *Queue*s and a improved implementation called *LockingQueue* which provides locking and priority support. The simple Queue can be used to add items into the queue. The consumers of the queue can remove items from the queue. However if the the consumer client which acquired an item from the Queue crashes then the queue item is lost even though it might not have been processed. The LockingQueue provides items to consumers but does not remove the item from the queue. It creates an ephemeral node corresponding to the item which indicates that the item is being processed. In case the consumer client crashes then the ephemeral znode is deleted and another consumer client can acquire the item. Once the consumer finishes processing the item it should explicitly inform the LockingQueue that it has processed the queue item at which time the queue removes the item from the queue.

Watchers

A client may need to watch for only certain types of changes to a znode, such as changes to the data of the znode or its children. The watches left on the znodes by the client calls like *get*, *get_children* or *set* are triggered whenever the znode is updated irrespective of the type of event. The watchers provide more fine grained control. They can also be disabled by returning a False Boolean function from the callback function.

3.2.3 Transactions

ZooKeeper has the facility to perform atomic transactions which consist of a sequence of operations. A transaction allows a client to perform a sequence of operations on the cluster such that the operations and their results are not interleaved with operations from other clients. From the client's perspective it appears like the sequence of operations is processed as a single operation on the ZooKeeper cluster. This is a common feature on many databases and can be a useful feature in constructing more complex functionality. An application which uses the Kazoo client can request a *Transaction* object from the Kazoo client. Then it performs *get*, *create*, *update* and *delete* operations

within the scope of this transaction. Then finally it calls *commit()* on the *Transaction* object. This sends the transaction to the clusters which performs the operations while upholding the constraints of the transaction. If the transaction can be completed then the Kazoo client informs the application by returning *True* to the application. If the transaction fails then it is automatically retried. If after a fixed number of retries the transaction cannot be still completed then a *False* is returned to the application.

4
Implementation

4.1 ParKazoo Architecture

The entire ParKazoo server set is called a ParKazoo ensemble. An ensemble contains multiple ZooKeeper clusters. When a client wants to connect to a ParKazoo ensemble it has to know the addresses of all the clusters, which could be all the servers of the cluster or a subset. If the information about all the clusters are not given to the client then the client cannot function correctly. For this reason the addresses of servers in the clusters needs to be stored centrally where it is accessible to all clients.

Figure 4.1 depicts a ParKazoo ensemble with three clusters, with each cluster having three servers. All the clients are connected to all the cluster. However each client is connected to only one server in the cluster.

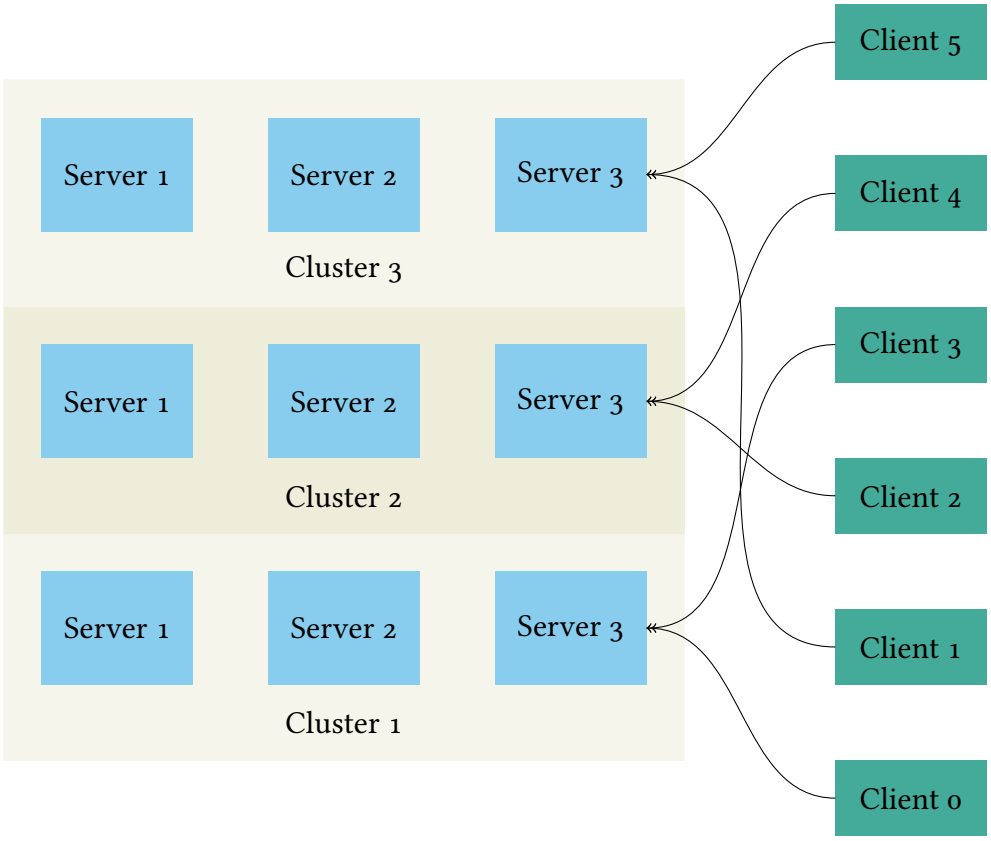


Figure 4.1: ParKazoo Architecture

4.2 Client Initialization

The information about all the constituent clusters are stored on one cluster which is part of the ParKazoo ensemble or on separate ZooKeeper cluster. The choice of the storage location is irrelevant as long as all the clients can access it. The configuration information could also be stored on all the clusters of the ParKazoo ensemble but this would require that the information is kept consistent. When the client has to be initialized only one of the server addresses has to be provided. The client connects to the server, reads the information about all the clusters and then initializes the individual Kazoo clients with the addresses of all the individual clusters.

4.3 Mapping of Nodes

As previously discussed the namespace of the data is divided between the clusters of the ParKazoo ensemble. To do this a node has to be mapped to a single cluster. The mapping procedure must always return the same cluster. This way the two different clients will always know where to place the node and where it can be retrieved from.

Algorithm 4.1 Algorithm to map the nodes

ht!

```
procedure MapNode(path, clusters)
    parent_path ← ParentPath(path)
    hvalue ← SHA256(parent_path)
    cluster_count ← Length(cluster)
    selected ← hvalue mod cluster_count
    return clusters[selected]
end procedure
```

Algorithm 4.1 shows the algorithm to map nodes to clusters. The most important step in this procedure is to produce a hash value of the path string. The SHA-256 algorithm is used for this purpose. This can be replaced with any other algorithm which has the same properties. The replacement hashing scheme should always the same produce the same output on different platforms for the same path. To place a *znode* in the data tree the mapping function is used to find the cluster. For the lookup of this node the mapping function gives the clusters on which the *znode* and the data for it can be retrieved.

4.4 ParKazoo Operations

Every ParKazoo operation consists of some Kazoo operations wrapped in a single ParKazoo operation. Figure 4.2 depicts the basic algorithm which is common to all ParKazoo operations. The algorithm begins by accepting the path of the znode to be operated upon and the type of operation. Then the algorithm establishes if the parent

znode exists on the cluster of the target znode. If it does not exists then algorithm creates it if the operation is a modifying operating like *create*. Otherwise an *Error* is returned. Finally the operation is performed and the result of the operation is returned.

4.4.1 Node Creation

Before a node is created the parent needs to be checked. If the parent is an ephemeral node then an error is reported. The cluster of the parent node is obvious from the path of the child node. The call to create the node also contains flags to indicate if the node is an ephemeral node and/or a sequential node. If the node doesn't exist then a the path of nodes up to that point are also created.

```
def create(self, path, value=b"empty", ephemeral=False, sequence=False
  , makepath=False, acl=None):
  try:
      p_value, stat = p_cluster.get(parent_path)
      if stat.owner_session_id is not None:
          raise NoChildrenForEphemeralsError
  except NoNodeError:
      pass
  else:
      makepath = True
  clusters = self._get_path_clients(path)
  cluster = self._hash_function(clusters, path)
  return cluster.create(path, value=value, ephemeral=ephemeral,
 sequence=sequence, makepath=makepath, acl=acl)
```

The Figure 4.3 shows the steps in the creation of a node. The node to be created */node2/child1*. The node *child1* and its parent *node2* maybe present on 2 different clusters. If the *make_path* argument is not set to True then the parent node *node2* should exist. The destination cluster of *child1* maybe different from its parent *node2*. In that case the *node2* needs to be created on the destination cluster in the first step. This is

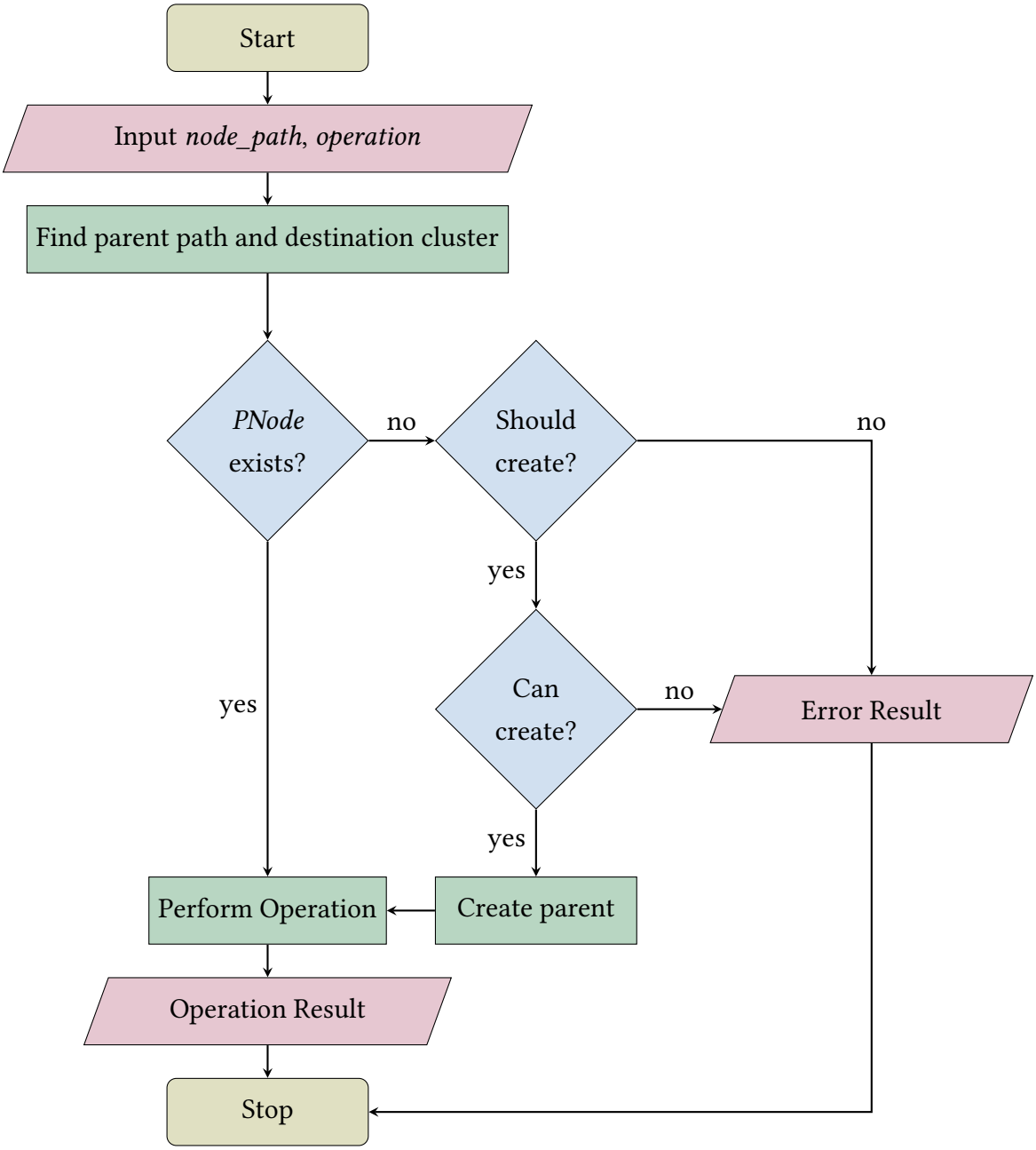


Figure 4.2: ParKazoo Operation Algorithm

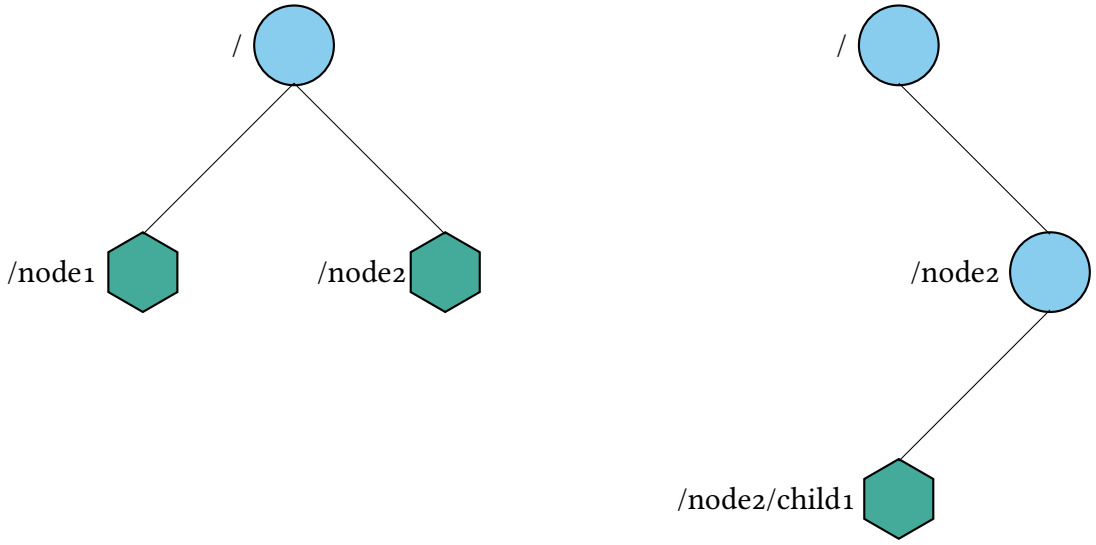


Figure 4.3: ParKazoo Create Operation

done with the *makepath* argument. Then the node is created with the create call.

4.4.2 Ensuring Path

The Kazoo library provides an API call to ensure that a path exists. If the path does not exist then it is created recursively. This approach is also used in ParKazoo.

```
def ensure_path(self, path, acl=None):
  parent, child = ospath.split(path)
  if parent is not '/':
      self.ensure_path(parent, acl=acl)

  parent_clusters = self._get_path_clients(parent)
  parent_cluster = self._hash_function(parent_clusters, parent)
  _, parent_stat = parent_cluster.get(parent)
  if parent_stat.owner_session_id is not None:
      raise NoChildrenForEphemeralsError

  clusters = self._get_path_clients(path)
```

```
  cluster = self._hash_function(clusters, path)
  return cluster.ensure_path(path, acl)
```

4.4.3 Check for Node Existence

The original functionality from Kazoo is reused after it has been wrapped in the mapping to find the right cluster.

```
  def exists(self, path, watch=None):
    clusters = self._get_path_clients(path)
    cluster = self._hash_function(clusters, path)
    return cluster.exists(path, watch)
```

4.4.4 Getting Node Data

Fetching the data for a node is straightforward. Hash the parent path of the node and find the cluster to which the node is mapped. Then perform the *get* using the Kazoo client for that cluster to fetch the data.

```
  def get(self, path, watch=None):
    clusters = self._get_path_clients(path)
    cluster = self._hash_function(clusters, path)
    return cluster.get(path, watch)
```

4.4.5 Setting Value of Node

To set the value of a node find the cluster to which the node is mapped. Then using the Kazoo client for that cluster perform the set operation.

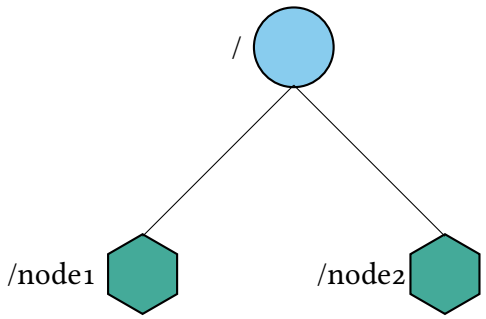
```
  def set(self, path, value, version=-1):
    clusters = self._get_path_clients(path)
    cluster = self._hash_function(clusters, path)
    return cluster.set(path, value=value, version=version)
```

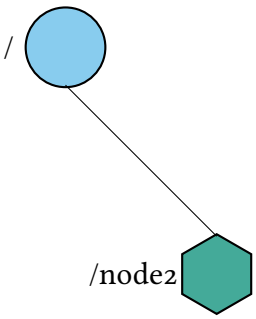
4.4.6 Finding children of a Node

In the first we using the *exists* call we ensure that the node for which the children are requested actually exists. Then we check if the parent node exists on the cluster which corresponds to the cluster for its child znodes. We also ensure the path(create the parent znode) in case of future requests which will trigger the watch function when child znodes are created at some point in the future.

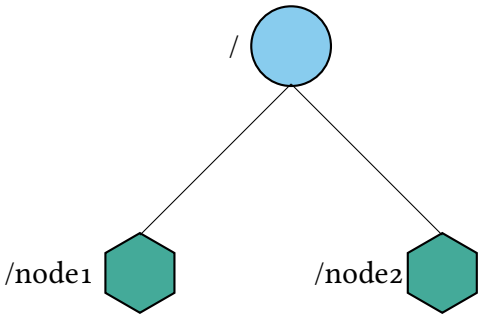
```
def get_children(self, path, watch=None, include_data=False):
  node_exists = self.exists(path)
  clusters = self._get_path_clients(path)
  full_path = _prefix_root(self.chroot, path)
  full_path = '%s/' % full_path if not full_path.endswith('/') else
 full_path
  hashobj = hashlib.md5()
  hashobj.update(full_path.encode('UTF-8'))
  value = int(hashobj.hexdigest(), 16)
  selected_cluster = value % len(sorted(self.c.items()))
  clusters = self._get_path_clients(path)
  children_cluster = clusters[selected_cluster]
  try:
     return children_cluster.get_children(path, watch=watch,
 include_data=include_data)
  except NoNodeError:
     if node_exists:
         children_cluster.ensure_path(path)
         return children_cluster.get_children(path, watch=watch,
 include_data=include_data)
     else:
         raise
```

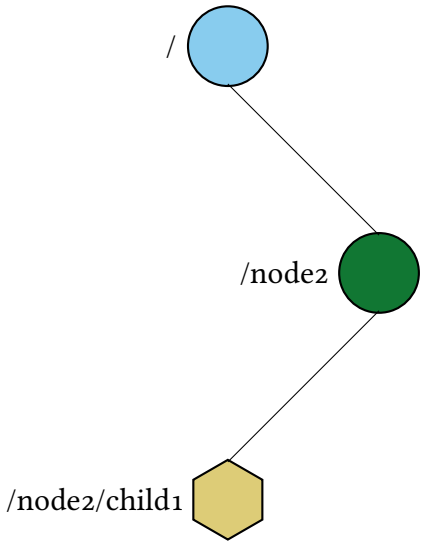
In the Figure 4.4 a query is made to the node */node2* to find its children. The children of */node2* maybe present on another cluster. In that case the parent node's existence





(a) Initial State





(b) Child Created

Figure 4.4: ParKazoo: Get Children Operation

has to be confirmed on the destination cluster. If the node does not exist then it has to be created. This is done so that watches can be left on the node in case nodes are created later. The the */node2* is queried on the destination cluster. Figure 4.4a shows the operation. The left side is the tree structure in the cluster which contains the node */node2*. It also has a sibling */node1*. The children of */node2* will get mapped to the cluster on the right. When there is a request to find the children for */node2* then a corresponding node is created on the children cluster. A watch is left on this newly created node. At some point of time in the future, as show in Figure 4.4b when the node */node2/child1* is created the watch set on the node in the previous step is triggered.

4.4.7 Transactions

Transactions are used on ZooKeeper to speed up a sequence of operations. Either all the operations of a transaction are committed or the whole transaction is rolled back. This ensures that operations can be completed because only a single consensus is required to complete the whole transaction rather than an individual consensus for each operation in the transaction.

But in a ParKazoo ensemble there are multiple clusters and it's not possible to perform a consensus where the transaction is operating on znodes which are present on different clusters. Although this appears to be a substantial disadvantage, most transactions operate on sibling nodes. Later in the evaluation of the recipes we show these transactions are still used. Since all siblings are present on the same cluster in the case of strong consistency transactions can still be used to process the operations. In fact, most complex recipes like Locks, Barriers, Double Barriers which use transactions can still continue operating with this limitation.

To implement the transaction the original Kazoo Transaction object is wrapped with basic verification logic which ensures the operations of the transaction are all operating on the same cluster. Every time an operation is added to the transaction the cluster corresponding to that operation is noted. Before the commit occurs the list of clusters is verified so that it contains only a single cluster.

The *Transaction* object provides the following operations:

```
1. def create(path, value='', ephemeral=False, sequence=False)
2. def delete(path, version=-1)
3. def set_data(path, value, version=-1)
4. def check(path, version)
5. def commit_async()
6. def commit()
```

5
Specification Compliance

The Kazoo library has several unit tests which verifies every aspect of its functionality and ensures that every build complies with the ZooKeeper client standards. After implementing the ParKazoo library we use the same tests to ensure compliance with the same standard. We discuss some of the tests from the original Kazoo library as well as the tests which were implemented specifically for ParKazoo.

The unittest [Hamill, 2004] framework used by the Kazoo library is called *nose*. Unit tests are designed so that they test on a small part or ”unit” of program. In a procedural program a unit is could be a function and in an object-oriented program it could be an interface, a class or a method.

The unit test setup consists of totally 9 servers each running ZooKeeper. The 9 servers are divided into 3 clusters each running 3 servers. All the servers are actually locally running ZooKeeper processes. To one of the clusters we write the information to connect to all the clusters including itself. Then the server addresses of this cluster

is passed to the ParKazoo clients used in the unit tests.

5.1 Kazoo Tests

The tests which are reused from the original Kazoo library are listed below:

5.1.1 Barrier Tests

There are tests to check the correctness of both the *SingleBarrier* and *DoubleBarrier*. For the single barrier test, we check that the barrier doesn't wait on a barrier that is non-existent and that it successfully waits on a barrier that is initialized and exists.

For the double barrier we first test the basic functionality of creating the barrier, entering it, then removing it and leaving it. Then we test that the DoubleBarrier works with two and three threads. Finally we test if the barrier functionality is unchanged when the corresponding paths for the barriers exist and they don't. *Barrier* and *DoubleBarrier* internally use *create*, *delete* and *ensure_path* calls on the ParKazoo client.

5.1.2 Counter Tests

We test that the counter functions correctly for both integer and float values. We also ensure that an error is raised when invalid values are added to the counter.

The *Counter* class internally uses *ensure_path*, *set*, *get* and *retry* operations of the ParKazoo client. In the *set* operation the *version* parameter of the set operation is used so that only one client updates the client at a time.

5.1.3 Lock Tests

The lock test are quite extensive since locks are one of most commonly used use case for ZooKeeper. First the basic functionality of the lock is checked using just one thread. We spawn a thread which attempts to acquire the lock. We ensure that the thread's candidacy becomes visible on the main thread using the *contenders()* method in the *Lock* class. In the second test multiple threads are created with their own clients all of which attempt to acquire a common lock. In the meanwhile, the main thread

acquires the lock first and get a list of contenders for the lock which should be the clients in the other threads. Then the test checks if the lock is acquired and released in the same order of the contenders obtained in the previous step. The next test ensures that an acquired lock is released when the client session is lost. This done done by forcing the client to disconnect. Upon reconnection the client should indicate that the lock has been released. The non-blocking call to acquire a lock is also verified for correct execution. We also ensure that when the acquisition of the lock is canceled an exception is raised. Double acquisitions and multiple acquire-and-release on the same lock are checked for correctness.

Lock class uses the *get_children()*, *delete()*, *create()* and *exits()* methods in the ParKazoo client. It also makes heavy use of watches.

5.1.4 Semaphore Tests

The tests for the *Semaphore* test basic functionality. The first tests the simple acquisition and release of a Semaphore of size 1. Then it tests that a semaphore of size 1 cannot be acquired more than once. Then the non-blocking acquisition and release of semaphores is tested. The the *holders()* property of the semaphore which gives the members which have already acquired the *Semaphore*. Finally edge cases like session loss, inconsistent maximum lease parameters is checked so that semaphore behaves as expected.

Semaphore uses the *exists()*, *ensure_path()*, *get()*, *get_children()* and *delete()* operation of the ParKazoo client.

5.1.5 Partitioner Tests

The unit test for *SetPartitioner* tests the basic functionality by checking if a one member party acquires all the members in a set. Then a two member party is used to check if the members of the set are divided equally and if the partitioner still functions correctly when the membership of the party is expanded. Finally we verify for the condition when the members of the party are larger than the number of items available.

Partitioner internally uses the *Lock* and *Party* class for its implementation It also uses session watches which listens for broken sessions.

5.1.6 Party Tests

The unit test for party first checks the basic functionality. It does this by adding members to the *Party* and confirming that they are registered as party members as returned by the *data* attribute on the *Party* class. And lastly the unit test checks that party functions correctly when an existing node is reused to create a Party and when the Party node disappears.

5.1.7 Queue Tests

For basic queues the unit tests check if the item are added in the FIFO(First in First Out) order. Then it checks a newly created *Queue* is empty. And lastly it checks that the priority of items added into the queue is honored while dequeuing.

The *Queue* and *LockingQueue* internally use the *Transaction* and *sync()*. It also uses *ensure_path()*, *retry()*, *delete()* and *delete()* methods on the ParKazoo client.

5.1.8 Watchers Tests

There are unit tests for the various types of watches *DataWatcher*, *ChildrenWatcher* and *PatientChildrenWatcher*. There are tests for verifying the basic functionality of each them. The different Watch classes are implemented in such a way that they can also be used as decorators. The tests verify that both styles of usage function correctly. And finally there are unit tests for edge cases like bad watch functions, invalid nodes and expired sessions.

The various types of watchers internally use the *get()*, *get_children()* and *retry()* methods on the ParKazoo client object.

To summarize the unit tests test every functional aspect of the ParKazoo client. The functions of client tested by every unittest is listed in the the Table 5.1.

Test	Functionality
Barrier	create, delete, ensure_path
Counter	ensure_path, set, get, retry
Lock	get_children, delete, create, exists
Semaphore	exists, ensure_path, get_children, delete
Partitioner	watch
Party	get_children, create
Queue	sync, ensure_path, retry, delete
Watcher	get, get_children, retry

Table 5.1: ParKazoo UnitTest functionality checks

5.2 ParKazoo Tests

There are tests which are implemented to specifically verify the functionality of ParKazoo. They are as follows:

5.2.1 Mapping Tests

For every path there should be a mapping to a cluster in the ParKazoo. We start by testing that a path gets mapped to a cluster. So the cluster which is the result of the mapping function should be one of the members of the cluster list. Second we test that the result of the mapping should be the same every time the mapping function is executed. The test also checks that the the result is the same if the path contains a trailing slash and without it. Finally the test verifies that sibling nodes always get mapped to the same cluster.

5.2.2 Transaction Tests

The basic functionality of a transaction is tested by creating a node, modifying it and finally reading the value which should be the correct value. Then we check that the transaction will complete when we modify sibling nodes through a transaction. Fi-

nally we verify that there is an exception when we attempt to perform a transaction which the operations are mapped to different clusters.

5.2.3 Configuration and Setup Tests

The configuration has to be read from a primary ZooKeeper cluster. A valid configuration has to produce a properly configured ParKazoo client. To test this we use a single clustered ParKazoo cluster which contains its own information.

5.2.4 Client State and Connectivity Tests

The client should provide information about about the currently connectivity status of the ParKazoo ensemble. After the client connects to all the cluster, one of the constituent is broken manually. This should move the connection status to indicate that the same in the global state. When all the clusters are connected the global state should also reflect the same.

5.2.5 Listeners Tests

The application can attach listeners to listen to connection state changes. The first test checks that the listeners deliver the connection state changes in order to the application Then test checks that the *Disconnected* event is delivered whenever one of the connections is manually broken. Finally the *Connecting*and *Connected* events should be delivered when the client reconnects.

5.2.6 Hash Function Tests

The ParKazoo client provides a default hash function for hashing the paths. This function should return the same value irrespective of the platform on which it is executed. The basic object hash function which is provided by the Python standard library is platform dependent and is hence unsuitable for our requirements. To remedy this the hash function is configured to use the SHA-256 algorithm. The test ensures that the function returns the same value for the same string after it has been processed.

6
Evaluation

The peak write throughput performance and the corresponding latencies of the original ZooKeeper and ParKazoo are compared in our evaluation. The tests are performed on a cluster(machine pool) of 50 machines. Each machine is equipped with an Intel® Xeon™ E5405 CPU clocked at 2.00GHz and 8GB System Memory. All the machines have a etXtreme BCM5754 Gigabit Ethernet PCI Express network card. For the purpose of these tests we use the ZooKeeper version 3.4.

Each machine has its own local storage as well as a networked storage which is shared and common between all nodes in the cluster. The executable binaries and scripts are stored on the shared storage so that they are available from all the machines in the cluster. The configuration files for each node is stored on the local storage.

The machines in the cluster are monitored using Ganglia [Sacerdoti et al., 2003]. The Fabric library is used to orchestrate and coordinates the steps of the testing process. Fabric is a library for application deployment [Spotswood & Srinivasan, 2003]

and system administration.

6.1 Test Introduction

The design goals for ParKazoo was to support higher throughput rates than what is currently supported through ZooKeeper. In order to test if we have achieved our goal, we test for one simple metric. We find the maximum load, that is the maximum number of operations per second(read or write) which ParKazoo can support. We use the same test for ZooKeeper to compare and determine the improvement which was gained. For this we setup a ZooKeeper cluster or a ParKazoo ensemble of a certain size on some nodes in the machine pool. On the other nodes of the machine pool we run processes that execute the tests and determine the benchmark.

6.2 Test Setup

The first step is to start the required the server set, either ZooKeeper or ParKazoo and configure them. Then the machines which make the requests are are configured. Conceptually the create and delete operations are equivalent because they both require consensus between a quorum of the servers. Hence both of them are considered as write operations. The requester nodes create and delete *znodes* on the servers. The created nodes are not retained on the server because the performance of the cluster gradually diminishes as the amount of data stored increases.

The machines in the machine pool have one of the following role:

6.2.1 Orchestrating Node

Only a single machine is used to perform this role. It is in charge of starting the machines which perform the other roles. It also coordinates the tasks of collecting and aggregating the results and finally cleanup. Algorithm 6.1 shows the algorithm of the main orchestrating process.

6.2.2 Server Nodes

The server nodes run the ZooKeeper server processes. They are either configured to run in the regular ZooKeeper configuration or as a ParKazoo ensemble.

6.2.3 Client Nodes

The test client nodes run a Python script which forks and creates P number of slave processes as shown in Algorithm 6.2. Each slave process in turn spawns T number of threads which is shown in Algorithm 6.3. Each threads get its own client object which it can use to make requests to the main ensemble. Algorithm 6.4 summarizes the actions performed in a test client thread. A ParKazoo/Kazoo barrier is used to wait for all the processes to initialise and spawn their threads. Once all the processes reach this barrier and they cross it and begin to execute a loop. The loops contains a *create* to create a node in the ParKazoo/ZooKeeper tree structure. The size of the created znode is 1MB. Upon successful creation of the node the thread records the the current time and also the time it took to execute the operation. After that it removes the same node using the *delete* operation. After this it records the timestamps and execution time just like for the create operation.

After a prefixed amount of time has elapsed the value of the finish node is set on the cluster. All the test client threads have a *DataWatch* on this node. When the value is set it indicates to the threads to stop executing the requests loop. After this every processes collects the timestamps-duration pair lists from every thread and writes it to disk.

6.3 Testing Process

The objective of the testing process is to find the peak throughput and the corresponding latencies. The setup consists an ensemble of servers. For ZooKeeper there is a cluster with three servers and for ParKazoo there is an ensemble of three clusters with three servers each. While it would be expected that a 9-Server ParKazoo ensemble should be compared against a 9-Server ZooKeeper cluster, it is more fair to compare

Algorithm 6.1 Algorithm for Orchestrating Node

```
procedure PerformTests(p, n, t, d)   ▷ p processes, t threads, n clients, d duration
    servers ← AllocateMachines(numServers)
    ConfigureServers(servers)
    StartServers(servers)
    clients ← AllocateMachines(n)
    barrier ← CreateBarrier()
    finish ← CreateFinish()
    counter ← CreateCounter()
    for all c in clients do
        RunClientTest(c, barrier, finish, p, t, servers)
    end for
    while counter < p * t * n do                ▷ Wait for all clients to initialize
        Sleep(1)
    end while
    ClearBarrier(barrier)
    Sleep(d)
    SetFinish(finish)
    while counter > 0 do
        Sleep(1)
    end while
    StopServers(servers)
end procedure
```

it against a 3-Server ZooKeeper cluster. Both the three server ZooKeeper cluster and nine server cluster can tolerate a single server failure. Also since the performance of a cluster is inversely proportional to the size of cluster, the fastest ZooKeeper configuration is a three server configuration. This is an advantage which a nine server ParKazoo ensemble inherently possesses. The procedure for the tests is described as follows:

Algorithm 6.2 Local Test on Client Nodes

```
procedure ClientTest(processCount, threadCount, finish, barrier, serverInfo, counter)
    processes ← []
    for i ← 1, processCount do
        p ← TestProcess(threadCount, finish, barrier, counter, serverInfo)
        Push(processes, p)
    end for
    WaitProcessesJoin(processes)
end procedure
```

Algorithm 6.3 Individual Process on Test Client

```
procedure TestProcess(threadCount, finish, barrier, counter, serverInfo)
    threads ← []
    for i ← 1, threadCount do
        thread ← TestThread(finish, barrier, counter, serverInfo)
        Push(threads, thread)
    end for
    WaitThreadsJoin(threads)
    requests ← []
    for all t in threads do
        r ← GetRequests(t)
        Push(requests, r)
    end for
    DiskWrite(requests)
end procedure
```

1. In the beginning a single client node is initialized to make requests. It contains only a single thread with its own client. The average throughput for a thread count is recorded. Then the number of threads is incremented and the throughput for every thread count is recorded. The number of threads is incremented till there is no increase in the throughput. The throughput starts decreasing after a

Algorithm 6.4 Thread on Test Client

```
procedure TestThreads(finish, barrier, counter, serverInfo)
    client ← CreateClient(serverInfo)
    StartClient(client)
    IncrementCounter(client, counter)
    WaitOnBarrier(client, barrier)
    requestPairs ← []
    while not IsSet(client, finish) do
        t1 ← CurrentTime()
        node ← CreateNode(client)
        t2 ← CurrentTime()
        Push(t2 − t1, t2)
        t1 ← CurrentTime()
        DeleteNote(client, node)
        t2 ← CurrentTime()
        Push(t2 − t1, t2)
    end while
    DecrementCounter(client, counter)
    CloseClient(client)
    return requestPairs
end procedure
```

certain number of threads. The number of threads for which maximum throughput is obtained is used in the next step of the testing process. This gives T which is the maximum number of threads which can be run before the performance degrades due to context switching and other threading limitations.

2. Continuing with a single testing node the test is repeated with T number of threads and an increasing number of processes. The number of processes is incremented and the throughput is measured for each process count. The throughput again decreases after a certain process count. This process count tipping

point gives P which is number of processes for maximum throughput on a single node.

3. Finally we test with multiple testing nodes. Each node has T threads and P number of processes. For T number of threads and P number of processes increase the number of test nodes until maximum throughput is achieved.

Multiple nodes make requests to the ZooKeeper cluster or the ParKazoo ensemble. Each node has multiple processes and each process has in turn multiple threads. Each thread has its own Kazoo or ParKazoo client object to make requests to the servers. Kazoo/ParKazoo provides a recipe for barrier which is used by the clients to initialize and wait. Once all the client nodes reach the barrier it is cleared. To do this there is a common counter which is incremented whenever the client finishes initializing. The main orchestrating process waits for this counter to reach the precomputed value. Then it clears the barrier. This way all the client nodes can start making their requests at the same time. Every request is recorded with a timestamp and the time it took to complete the request. All these timestamped requests from each thread are collected and finally they are written to the disk. At the end of the run all the files containing the requests timestamp and duration are collected. They are then aggregated and processed. They are bucketed by the second in which they were executed. This gives the number of requests from all the test nodes that occurred in a second. The average request rate and their latency is computed and recorded.

The Figure 6.1 shows the throughput over a period of 100 seconds for a single node with a single thread and process. As we can see the throughput is better for ParKazoo than for ZooKeeper. The latencies for the requests are show in Figure 6.2. The latencies for ParKazoo are in the range of 200 milliseconds to 300 milliseconds. Where as for ZooKeeper they are spread across the entire range from 100 to 600 milliseconds. From these initial reading we can assume that the performance for ParKazoo is nominally better than for an equivalent ZooKeeper setup for the minimal configuration.

Table 6.1 lists the throughput for a single testing node with a single process. The throughput for ParKazoo is initially higher. But it does not scale linearly as more threads are added. Around the 80 threads mark the throughput for ZooKeeper in-

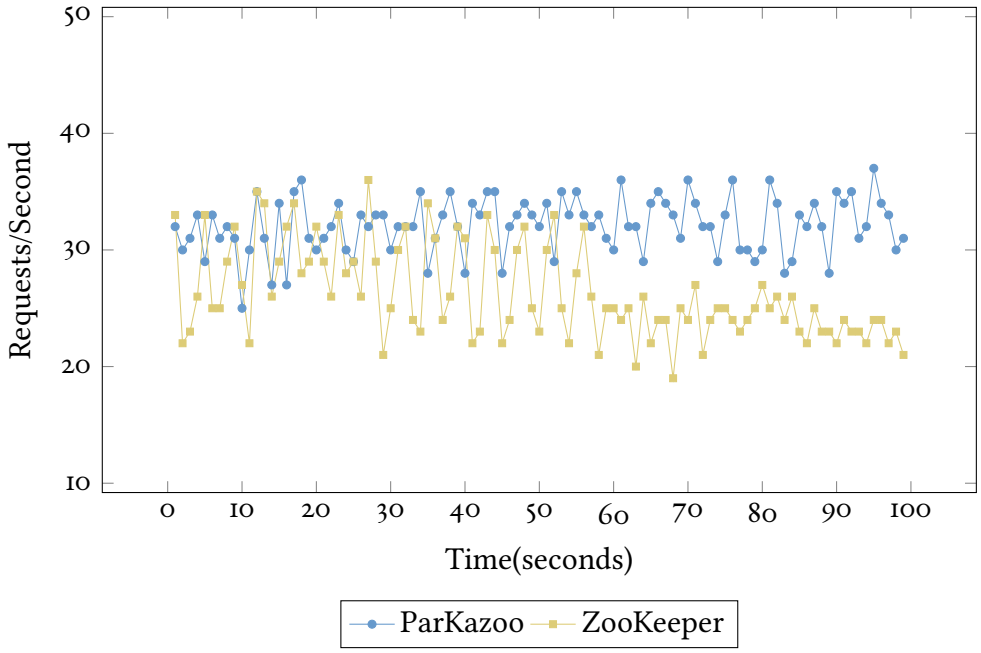
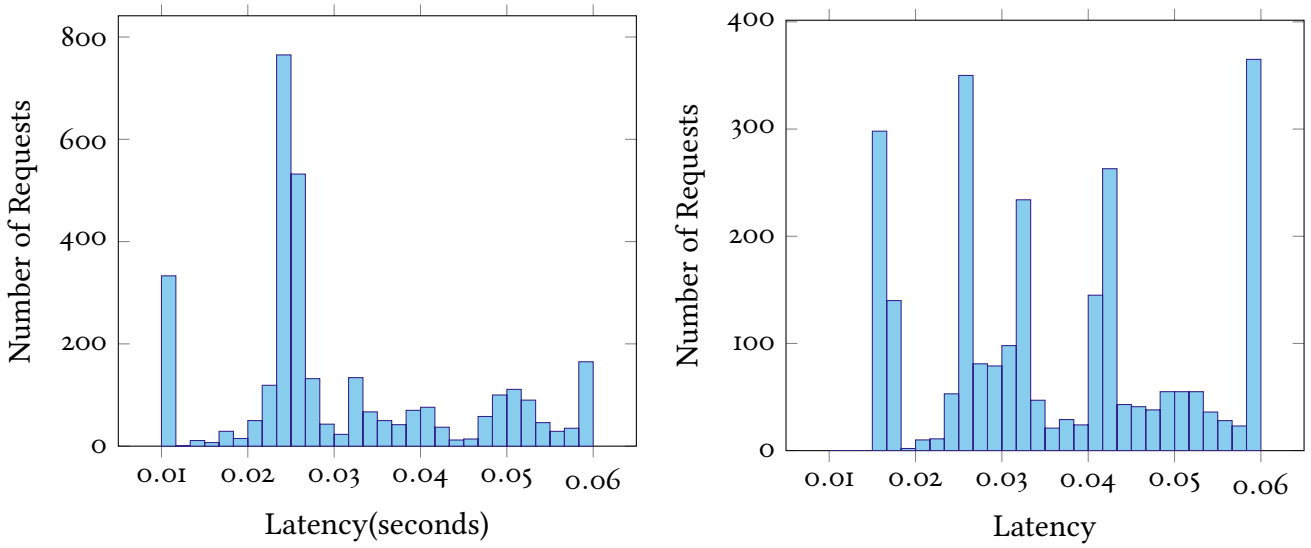


Figure 6.1: ParKazoo Throughput for Single Note, Thread and Process.



(a) ParKazoo

(b) ZooKeeper

Figure 6.2: Request Latency for Single Node, Thread and Process

Threads	Average Throughput	Mean Latency	Median Latency
1	26.76	0.037097	0.031448
5	51.29	0.097152	0.092078
10	71.74	0.138727	0.136078
20	91.98	0.215128	0.211797
40	127.61	0.307994	0.306246
80	200.81	0.392830	0.390069
120	284.02	0.423037	0.417495
160	354.79	0.452509	0.452077
200	362.69	0.549497	0.551253
210	367.17	0.577139	0.580336
220	357.33	0.612315	0.609688
240	354.17	0.674204	0.669845

(a) ZooKeeper

Threads	Average Throughput	Mean Latency	Median Latency
1	28.17	0.035403	0.033219
5	74.20	0.067198	0.066372
10	101.78	0.098097	0.094363
20	141.98	0.160644	0.137943
50	197.90	0.252970	0.252138
100	226.12	0.439794	0.441875
110	228.93	0.479758	0.481142

(b) ParKazoo

Table 6.1: Average Throughput, Mean Latency and Median Latency for a single node with a single process

creases over that of ParKazoo. In fact after more than 110 threads are added to the ParKazoo the process crashes due to the high number of open socket connections. This is of course an operating system limitation but exposes one of the flaws in the design of ParKazoo, that of unscalability. However this situation is unlikely to occur

in a production system. The throughput of ZooKeeper increases as additional threads are added to a thread count of 210 threads. For ParKazoo the thread count of 110 is chosen as the highest thread count.

Processes	Average Throughput	Mean Latency	Median Latency
1	362.69	0.549497	0.551235
2	653.21	0.610248	0.610136
6	1701.45	0.708220	0.668472
8	1279.72	1.232167	1.127271

(a) ZooKeeper

Processes	Average Throughput	Mean Latency	Median Latency
1	226.12	0.439794	0.441875
2	337.60	0.527642	0.518607
3	526.46	0.565287	0.549471
5	838.93	0.594981	0.573028
6	1002.21	0.597969	0.570774
7	1136.48	0.618671	0.581756
8	1210.93	0.658900	0.608366
9	1195.36	0.750113	0.666098
10	1149.30	0.868395	0.748578

(b) ParKazoo

Table 6.2: Average Write Throughput for a single ZooKeeper client node with a multiple processes

Then in the next step the number of processes is increased. The Table 6.2 lists the average throughput for a single node with increasing number of processes for both ZooKeeper and ParKazoo. The Table 6.2a shows the throughput for ZooKeeper. For eleven processes the maximum throughput is reached. According to Table 6.2b the maximum throughput for ParKazoo is with 8 processes.

Then the test is repeated with the number of threads and processes determined and multiple test nodes. The Table 6.3 lists the throughputs for the tests. From the Table 6.3a the maximum throughput for ZooKeeper is achieved with 5 nodes. The

maximum average throughput is 5421.84 requests/second. From Table 6.3b the maximum possible throughput for ParKazoo is 11430.90 requests/second. That is not the maximum possible throughput achievable When the tests were conducted with more nodes the clients started disconnecting because of the maximum number of connections which is supported had been reached. The infrastructure cannot support more concurrent connections.

Nodes	Average Throughput	Mean Latency	Median Latency
1	1701.45	0.708220	0.668472
2	3501.71	0.692168	0.655367
3	5299.55	0.688530	0.651348
4	6477.30	0.753467	0.727466
5	5864.40	0.941874	0.832860

(a) ZooKeeper

Nodes	Average Throughput	Mean Latency	Median Latency
1	1210.93	0.658990	0.608366
2	2402.26	0.665377	0.606665
3	3561.05	0.673851	0.615165
4	4773.83	0.673322	0.615663
5	5836.77	0.688369	0.644749
6	6962.63	0.691840	0.637960
7	8146.70	0.690039	0.638991
10	10624.13	0.768104	0.684975
12	11648.35	0.838369	0.738114
14	12837.89	0.888452	0.784030

(b) ParKazoo

Table 6.3: Average Throughput for writes in multiple client nodes with a multiple processes

The throughput rate for the test for both ZooKeeper and ParKazoo are shown in the Figure 6.3. The Figure 6.4 shows the latencies for requests from the same tests. The latencies for the ZooKeeper from Figure 6.4a is in the range of 1.5 seconds to 1.75

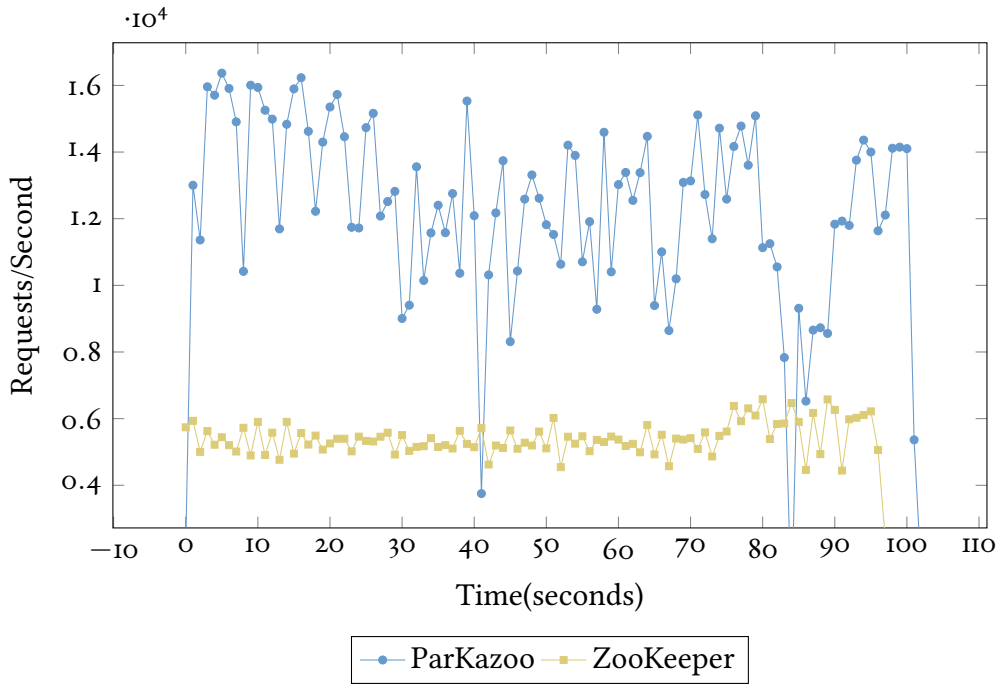
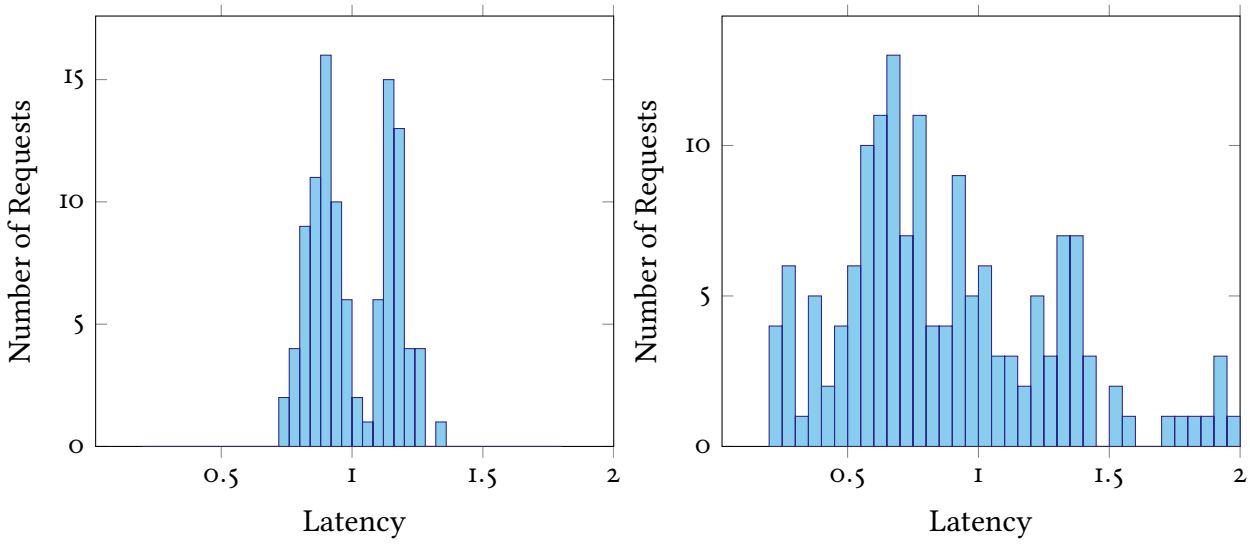


Figure 6.3: ParKazoo Throughput

seconds. And for ParKazoo from Figure 6.4b the latency for the requests is in the range of 250 milliseconds to 1 second. The performance of ParKazoo is better both in terms of throughput and request latencies.

6.4 Read Performance

For pedantry's sake we also compare the performance of the read operation. The test for the read performance is the same as the writes. But instead of create and delete operation the read operation is done in a loop. When the test client threads are initialised they also create a few hundred znodes with 1MB of data which are then read in the loop. Table 6.4 shows the performance for a single process with an increasing number of threads. The highest throughput is for 18 threads in ZooKeeper and 20 threads in ParKazoo. The experiments are repeated with increasing number of processes and the results are listed in Table 6.5 . For ZooKeeper the highest read throughput is for 18 processes and for ParKazoo with 20 processes. Finally the number of test client



(a) ZooKeeper Latencies

(b) ParKazoo Latency

Figure 6.4: Latency for Maximum Throughput

nodes are increased incrementally with the results in Table 6.6. For ZooKeeper the highest throughput achieved is 145461.05 reads/seconds for 6 nodes. But the test for ParKazoo again runs into the problem os disconnections due to high number of open connections but the highest achieved read throughput is 304516.43 requests/second for 16 nodes. This is twice the throughput achieved for ZooKeeper. Although there is no conclusive result the initial measurements prove promising.

6.5 Conclusion

As we can see from the throughput rates when the number of clients is smaller the throughput of the ParKazoo is almost 100%-120% higher than the equivalent ZooKeeper setup. However when the number of ParKazoo clients on a single node increases the throughput rate fails to grow and at a certain point drops below the throughput rate of the the equivalent ZooKeeper test node. But the total throughput of ZooKeeper plateaus out at around 6300 requests/seconds. However the request throughput of ParKazoo reached around 11000 requests/second in our tests.

Threads	Average Throughput	Mean Latency	Median Latency
1	3161.10	0.001907	0.001758
2	5219.46	0.002270	0.002138
4	12732.17	0.001883	0.001744
8	21061.39	0.002285	0.002035
12	23391.68	0.003083	0.002898
16	25552.62	0.003775	0.003350
18	25659.22	0.004220	0.003642
20	24948.05	0.004812	0.004301

(a) ZooKeeper

Threads	Average Throughput	Mean Latency	Median Latency
1	2238.19	0.000447	0.000443
5	2304.32	0.002161	0.002045
10	2371.39	0.004187	0.003861
15	2125.61	0.007047	0.006580
20	2156.82	0.009220	0.008520
40	2097.87	0.019055	0.017524

(b) ParKazoo

Table 6.4: Average Throughput, Mean Latency and Median Latency for a single node with a single process

The reason for the deteriorated performance of ParKazoo when more processes and threads are present is probably due to the higher number of open connections. This issue has need to be investigated further.

Processes	Average Throughput	Mean Latency	Median Latency
1	3161.10	0.001907	0.001758
2	5216.46	0.002270	0.002138
4	12732.17	0.001883	0.001744
8	21061.39	0.002285	0.002035
12	23391.68	0.003083	0.002698
16	25552.66	0.003775	0.003350
18	25659.22	0.004220	0.003642
20	24948.05	0.004812	0.004301

(a) ZooKeeper

Processes	Average Throughput	Mean Latency	Median Latency
1	2371.39	0.004187	0.003861
3	7181.68	0.004160	0.003855
5	11392.93	0.004385	0.003960
6	12695.13	0.004713	0.004196
10	18000.76	0.005554	0.004504
12	18757.29	0.006402	0.004911
15	19799.74	0.007615	0.005919
18	19908.29	0.009055	0.007310
20	19615.54	0.010227	0.008226
25	19221.17	0.013055	0.010500

(b) ParKazoo

Table 6.5: Average Read throughput for a single ZooKeeper and ParKazoo client node with a multiple processes

Nodes	Average Throughput	Mean Latency	Median Latency
1	25659.22	0.004220	0.003642
3	76472.63	0.004245	0.003655
6	145461.05	0.004487	0.003874

(a) ZooKeeper

Nodes	Average Throughput	Mean Latency	Median Latency
1	19615.54	0.010227	0.008226
3	58905.98	0.010221	0.008336
6	119882.66	0.010050	0.005513
9	179346.34	0.010083	0.005475
10	186502.48	0.010805	0.008657
11	105863.90	0.010753	0.008611
12	225566.19	0.010718	0.008579
14	263947.05	0.010685	0.008590
16	304516.43	0.010577	0.008500

(b) ParKazoo

Table 6.6: Average read throughput for writes in multiple client nodes with a multiple processes

7 Limitations

7.1 Data Inconsistency

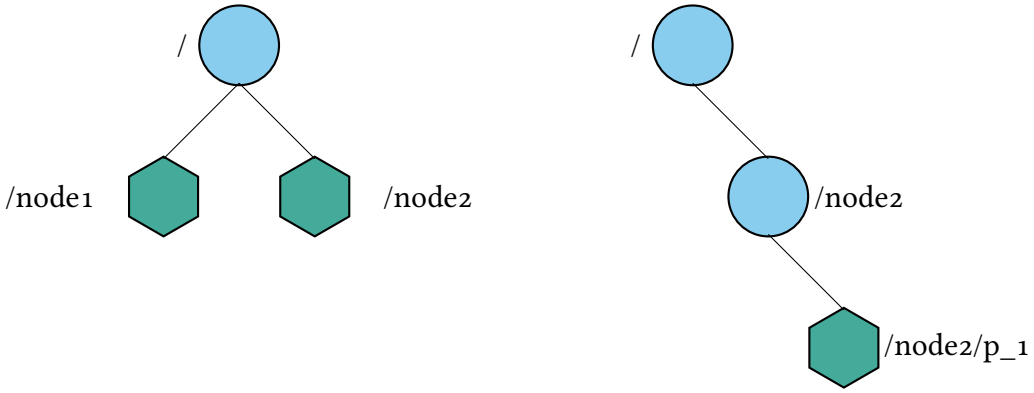
Not all operations are completely data safe. The recursive *delete()* operation is an example of such an operation. The delete() operation iterates over the client connections and then deletes the subtrees. Assume that the client crashes after deleting the subtrees only a few clusters. Then if one of the nodes which is missing then the whole tree gets recreated. So a node which is supposed to be deleted still exists. Figure 7.1 shows how inconsistency can be caused by the recursive *delete()* operation. In the first step there is a node */node1/p_1* which is on a cluster shown on the right. The cluster on the left contains the parent node */node1*. If the recursive delete operation deletes the parent node first and crashes then the node */node1/p_1* still exists. If the node */node1* is created again then the child gets automatically created.

7.2 Ordering of Operations

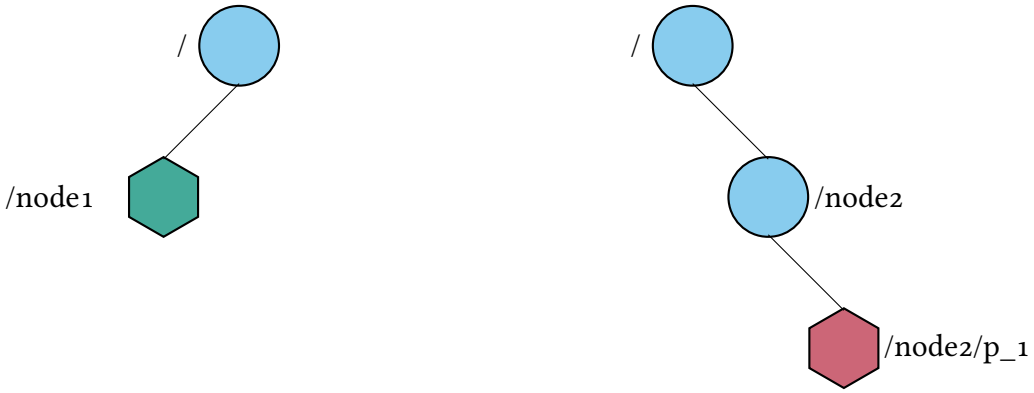
ZooKeeper provides the guarantee of client operation order. This means that all the operations which are issues by a client are executed in the order they are sent to the server. But with ParKazoo this assumptions will not hold true. Assume two operations issued by a client. The first operation is mapped to one cluster and the second operation gets mapped to another cluster. If the operations are issued one after the other in a normal ZooKeeper operation the result of the first operation is returned first. But with ParKazoo the result of the second operation could be returned first because the operations are executed on different clusters.

7.3 Limited Client Connections

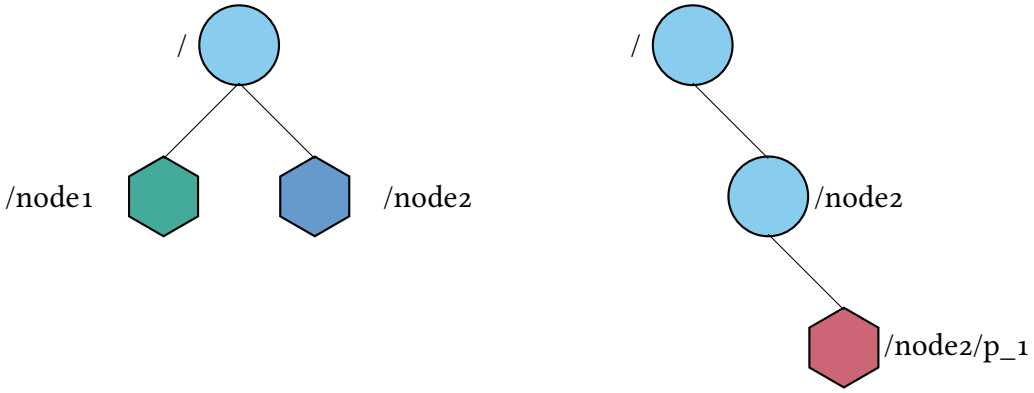
The evaluation tests revealed that there is a natural limit to the number of connected clients. Although in a production system it is would quite uncommon to have so many clients this still is a limiting factor for the throughput of the entire system.



(a) The Initial State of Data



(b) Parent Node Deleted



(c) Child Node Created

Figure 7.1: Recursive Delete Operation Inconsistency

8
Future Improvements

The implementation of ParKazoo could be improved to obtain better performance.

8.1 Auto-Reconfiguration of Ensemble

Future versions of ZooKeeper will support auto-reconfiguration of cluster members while the cluster is still in operation. To do this the new members should be added to the cluster. But the rule for this is that during the switchover the common subset of members between the new and old members of the quorum should contain a minimum quorum number of servers and one of them should be elected as the leader. After that the old cluster members which are are no longer required are switched off. To do this the clients are rebalanced in such a a way that minimum number of disconnections and reconnections should happen.

To reconfigure the ParKazoo client sets a watch on the configuration node. The

configuration node should have a version number associated with it. When the watch is triggered due to a reconfiguration the client receives the update. It then should re-read the configuration information. After a random amount of time the server-list on the client should be updated. The client internally determines if all the connections should be moved from one server to another server.

8.2 Weak Consistency

The ParKazoo system relies on the fact that sibling nodes are mapped to the same cluster. However if the application programmer is not aware of this fact then this can cause situations where all the traffic is directed at a single cluster in the ensemble. But if the programmer does not require the guarantees of primary order which are provided by ZooKeeper then the ensemble could be configured in weak consistency mode. In this case the destination cluster of the node is determined by its own path rather than the path of the parent. This way the nodes will be uniformly distributed across the clusters.

8.3 Asynchronous Operations

Currently all the operations internally use the synchronous version of the Kazoo operations. Because they are synchronous they are executed in order. By using the asynchronous versions of the operations the operations can be executed in parallel. For example in the *create* operation the time to create a node can be decreased. If the checks to ensure that the parent node exists and is not an ephemeral znode and the call to create a node are issued in parallel, the time for the complete operation is the time required for the longer sub-operation. In the case of recursive delete operation all the delete operations can be parallelized.

Bibliography

[Apache, 2015] Apache (2015).

[Biligiri et al., 2014] Biligiri, P., Kar, A., & Kanak, A. (2014). Improving write throughput scalability of zookeeper by partitioning namespace.

[Burrows, 2006] Burrows, M. (2006). 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.

[Chandra et al., 2007] Chandra, T. D., Griesemer, R., & Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (pp. 398–407).: ACM.

[Chandra & Toueg, 1996] Chandra, T. D. & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 225–267.

[CoreOS, 2015] CoreOS (2015). etcd - github.

[Dean & Ghemawat, 2008] Dean, J. & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 107–113.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41 (pp. 205–220).: ACM.

[Foundation, 2015] Foundation, P. S. (2015). threading — thread-based parallelism. `https://docs.python.org/3.4/library/threading.html`. [Online; accessed 26-January-2015].

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03 (pp. 29–43). New York, NY, USA: ACM.

[Gilbert & Lynch, 2002] Gilbert, S. & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 51–59.

[Hamill, 2004] Hamill, P. (2004). *Unit Test Frameworks: Tools for High-Quality Software Development.* " O'Reilly Media, Inc.".

[Hastings, 1990] Hastings, A. B. (1990). Distributed lock management in a transaction processing environment. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on* (pp. 22–31).: IEEE.

[Hunt et al., 2010] Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8 (pp.9̃).

[Junqueira, 2010] Junqueira, F. P. (2010). Hadoop wiki: Partitioned zookeeper.

[Junqueira et al., 2011] Junqueira, F. P., Reed, B. C., & Serafini, M. (2011). Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on* (pp. 245–256).: IEEE.

[Karger et al., 1997] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., & Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97 (pp. 654–663). New York, NY, USA: ACM.

[Ketelsen et al., 2015] Ketelsen, B., St. Martin, E., Rarick, K., & Ratnakumar, S. (2015). Doozerd - github.

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

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

[Lampson, 2001] Lampson, B. (2001). The abcd's of paxos. In *PODC*, volume 1 (pp.ĩ3).

[MacCormick et al., 2004] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., & Zhou, L. (2004). Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, volume 4 (pp. 8–8).

[Ongaro & Ousterhout, 2013] Ongaro, D. & Ousterhout, J. (2013). In search of an understandable consensus algorithm. *Draft of October*, 7.

[Sacerdoti et al., 2003] Sacerdoti, F. D., Katz, M. J., Massie, M. L., & Culler, D. E. (2003). Wide area cluster monitoring with ganglia. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on* (pp. 289–298).: IEEE.

[Schneider, 1990] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 299–319.

[Shraer et al., 2012] Shraer, A., Reed, B., Malkhi, D., & Junqueira, F. P. (2012). Dynamic reconfiguration of primary/backup clusters. In *USENIX Annual Technical Conference* (pp. 425–437).

[Skiena,] Skiena, S. S. The algorithm design manual, 1997. *Stony Brook, NY: Telos Pr*, 504.

[Spotswood & Srinivasan, 2003] Spotswood, M. & Srinivasan, S. (2003). Systems and methods for application deployment.

[Van Rossum & Drake, 2002] Van Rossum, G. & Drake, F. L. (2002). Python reference manual release 2.2. 1.

[White, 2009] White, T. (2009). *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.".