Zookeeper

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

ZooKeeper is a distributed co-ordination service to manage large set of hosts. Co-ordinating and managing a service in a distributed environment is a complicated process. ZooKeeper solves this issue with its simple architecture and API. ZooKeeper allows developers to focus on core application logic without worrying about the distributed nature of the application.

The ZooKeeper framework was originally built at "Yahoo!" for accessing their applications in an easy and robust manner. Later, Apache ZooKeeper became a standard for organized service used by Hadoop, HBase, and other distributed frameworks. For example, Apache HBase uses ZooKeeper to track the status of distributed data. This tutorial explains the basics of ZooKeeper, how to install and deploy a ZooKeeper cluster in a distributed environment, and finally concludes with a few examples using Java programming and sample applications.

Audience

This tutorial has been prepared for professionals aspiring to make a career in Big Data Analytics using ZooKeeper framework. It will give you enough understanding on how to use ZooKeeper to create distributed clusters.

Prerequisites

Before proceeding with this tutorial, you must have a good understanding of Java because the ZooKeeper server runs on JVM, distributed process, and Linux environment.

Copyright & Disclaimer

Table of Contents

1. ZOOKEEPER – OVERVIEW

ZooKeeper is a distributed co-ordination service to manage large set of hosts. Co-ordinating and managing a service in a distributed environment is a complicated process. ZooKeeper solves this issue with its simple architecture and API. ZooKeeper allows developers to focus on core application logic without worrying about the distributed nature of the application.

The ZooKeeper framework was originally built at "Yahoo!" for accessing their applications in an easy and robust manner. Later, Apache ZooKeeper became a standard for organized service used by Hadoop, HBase, and other distributed frameworks. For example, Apache HBase uses ZooKeeper to track the status of distributed data.

Before moving further, it is important that we know a thing or two about distributed applications. So, let us start the discussion with a quick overview of distributed applications.

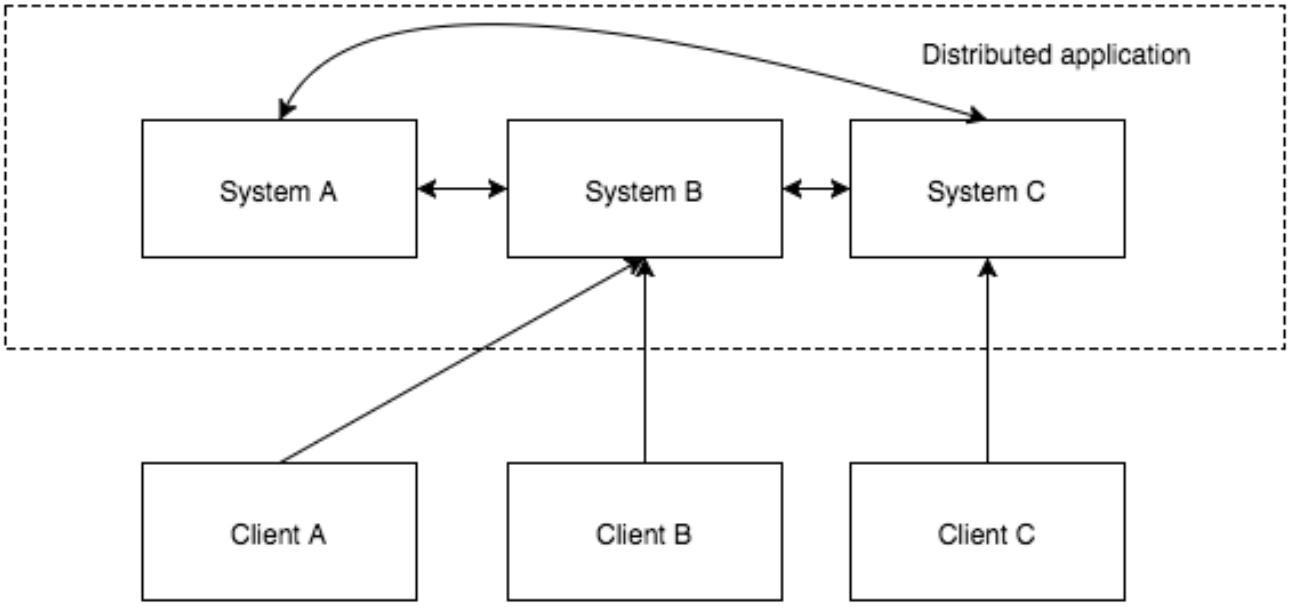
Distributed Application

A distributed application can run on multiple systems in a network at a given time (simultaneously) by coordinating among themselves to complete a particular task in a fast and efficient manner. Normally, complex and time-consuming tasks, which will take hours to complete by a non-distributed application (running in a single system) can be done in minutes by a distributed application by using computing capabilities of all the system involved.

The time to complete the task can be further reduced by configuring the distributed application to run on more systems. A group of systems in which a distributed application is running is called a **Cluster** and each machine running in a cluster is called a **Node**.

A distributed application has two parts, **Server** and **Client** application. Server applications are actually distributed and have a common interface so that clients can connect to any server in

the cluster and get the same result. Client applications are the tools to interact with a distributed application.



Benefits of Distributed Applications

- **Reliability** – Failure of a single or a few systems does not make the whole system to fail.
- **Scalability** – Performance can be increased as and when needed by adding more machines with minor change in the configuration of the application with no downtime.
- **Transparency** – Hides the complexity of the system and shows itself as a single entity / application.

Challenges of Distributed Applications

- **Race condition** - Two or more machines trying to perform a particular task, which actually needs to be done only by a single machine at any given time. For example, shared resources should only be modified by a single machine at any given time.
- **Deadlock** – Two or more operations waiting for each other to complete indefinitely.
- **Inconsistency** – Partial failure of data.

What is Apache ZooKeeper Meant For?

Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintain shared data with robust synchronization techniques. ZooKeeper is itself a distributed application providing services for writing a distributed application.

The common services provided by ZooKeeper are as follows:

- **Naming service** – Identifying the nodes in a cluster by name. It is similar to DNS, but for nodes.
- **Configuration management** – Latest and up-to-date configuration information of the system for a joining node.
- **Cluster management** – Joining / leaving of a node in a cluster and node status at real time.
- **Leader election** – Electing a node as leader for coordination purpose.
- **Locking and synchronization service** – Locking the data while modifying it. This mechanism helps you in automatic fail recovery while connecting other distributed applications like Apache HBase.
- **Highly reliable data registry** – Availability of data even when one or a few nodes are down.

Distributed applications offer a lot of benefits, but they throw a few complex and hard-to-crack challenges as well. ZooKeeper framework provides a complete mechanism to overcome all the challenges. Race condition and deadlock are handled using **fail-safe synchronization approach**. Another main drawback is inconsistency of data, which ZooKeeper resolves with **atomicity**.

Benefits of ZooKeeper

Here are the benefits of using ZooKeeper:

- **Simple distributed coordination process**
- **Synchronization** – Mutual exclusion and co-operation between server processes. This process helps in Apache HBase for configuration management.
- **Ordered Messages**
- **Serialization** – Encode the data according to specific rules. Ensure your application runs consistently. This approach can be used in MapReduce to coordinate queue to execute running threads.
- **Reliability**
- **Atomicity** – Data transfer either succeed or fail completely, but no transaction is partial.

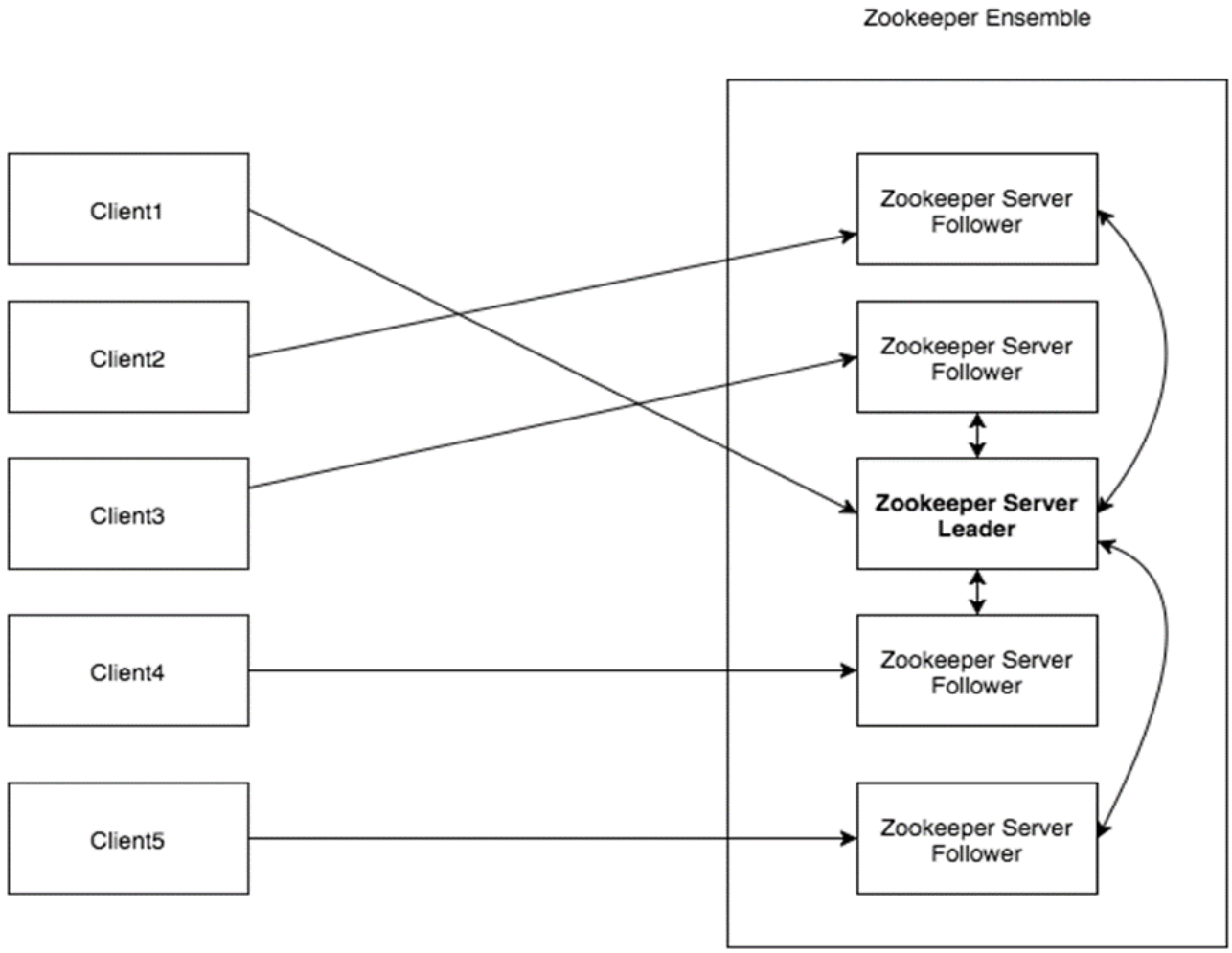
2. ZOOKEEPER – FUNDAMENTALS

Before going deep into the working of ZooKeeper, let us take a look at the fundamental concepts of ZooKeeper. We will discuss the following topics in this chapter:

- Architecture
- Hierarchical namespace
- Session
- Watches

Architecture of ZooKeeper

Take a look at the following diagram. It depicts the "Client-Server Architecture" of ZooKeeper.



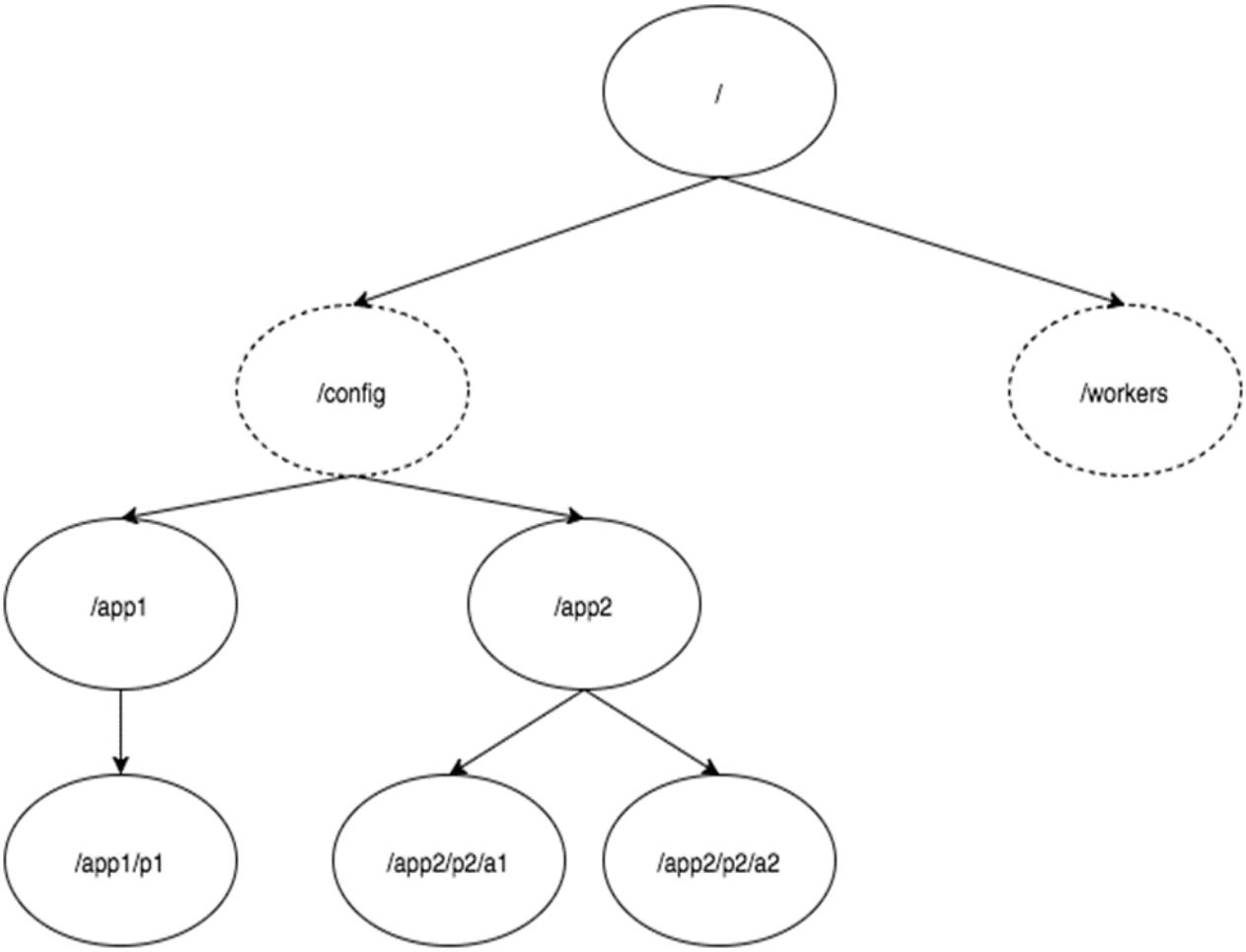
Each one of the components that is a part of the ZooKeeper architecture has been explained in the following table.

Part	Description
Client	Clients, one of the nodes in our distributed application cluster, access information from the server. For a particular time interval, every client sends a message to the server to let the sever know that the client is alive. Similarly, the server sends an acknowledgement when a client connects. If there is no response from the connected server, the client automatically redirects the message to another server.
Server	Server, one of the nodes in our ZooKeeper ensemble, provides all the services to clients. Gives acknowledgement to client to inform that the server is alive.
Ensemble	Group of ZooKeeper servers. The minimum number of nodes that is required to form an ensemble is 3.
Leader	Server node which performs automatic recovery if any of the connected node failed. Leaders are elected on service startup.
Follower	Server node which follows leader instruction.

Hierarchical Namespace

The following diagram depicts the tree structure of ZooKeeper file system used for memory representation. ZooKeeper node is referred as **znode**. Every znode is identified by a name and separated by a sequence of path (/).

- In the diagram, first you have a root **znode** separated by "/". Under root, you have two logical namespaces **config** and **workers**.
- The **config** namespace is used for centralized configuration management and the **workers** namespace is used for naming.
- Under **config** namespace, each znode can store upto 1MB of data. This is similar to UNIX file system except that the parent znode can store data as well. The main purpose of this structure is to store synchronized data and describe the metadata of the znode. This structure is called as **ZooKeeper Data Model**.



Every znode in the ZooKeeper data model maintains a **stat** structure. A **stat** simply provides the **metadata** of a znode. It consists of *Version number*, *Action control list (ACL)*, *Timestamp*, and *Data length*.

- **Version number**: Every znode has a version number, which means every time the data associated with the znode changes, its corresponding version number would also increased. The use of version number is important when multiple zookeeper clients are trying to perform operations over the same znode.

- **Action Control List (ACL)**: ACL is basically an authentication mechanism for accessing the znode. It governs all the znode read and write operations.

- **Timestamp**: Timestamp represents time elapsed from znode creation and modification. It is usually represented in milliseconds. ZooKeeper identifies every change to the znodes from "Transaction ID" (zxid). **Zxid** is unique and maintains time for each transaction so that you can easily identify the time elapsed from one request to another request.

- **Data length**: Total amount of the data stored in a znode is the data length. You can store a maximum of 1MB of data.

Types of Znodes

Znodes are categorized as persistence, sequential, and ephemeral.

- **Persistence znode**: Persistence znode is alive even after the client, which created that particular znode, is disconnected. By default, all znodes are persistent unless otherwise specified.

- **Ephemeral znode**: Ephemeral znodes are active until the client is alive. When a client gets disconnected from the ZooKeeper ensemble, then the ephemeral znodes get deleted automatically. For this reason, only ephemeral znodes are not allowed to have a children further. If an ephemeral znode is deleted, then the next suitable node will fill its position. Ephemeral znodes play an important role in Leader election.

- **Sequential znode**: Sequential znodes can be either persistent or ephemeral. When a new znode is created as a sequential znode, then ZooKeeper sets the path of the znode by attaching a 10 digit sequence number to the original name. For example, if a znode with path **/myapp** is created as a sequential znode, ZooKeeper will change the path to **/myapp0000000001** and set the next sequence number as 0000000002. If two sequential znodes are created concurrently, then ZooKeeper never uses the same number for each znode. Sequential znodes play an important role in Locking and Synchronization.

Sessions

Sessions are very important for the operation of ZooKeeper. Requests in a session are executed in FIFO order. Once a client connects to a server, the session will be established and a **session id** is assigned to the client.

The client sends **heartbeats** at a particular time interval to keep the session valid. If the ZooKeeper ensemble does not receive heartbeats from a client for more than the period (session timeout) specified at the starting of the service, it decides that the client died.

Session timeouts are usually represented in milliseconds. When a session ends for any reason, the ephemeral znodes created during that session also get deleted.

Watches

Watches are a simple mechanism for the client to get notifications about the changes in the ZooKeeper ensemble. Clients can set watches while reading a particular znode. Watches send a notification to the registered client for any of the znode (on which client registers) changes.

Znode changes are modification of data associated with the znode or changes in the znode's children. Watches are triggered only once. If a client wants a notification again, it must be done through another read operation. When a connection session is expired, the client will be disconnected from the server and the associated watches are also removed.

3. ZOOKEEPER – WORKFLOW

Once a ZooKeeper ensemble starts, it will wait for the clients to connect. Clients will connect to one of the nodes in the ZooKeeper ensemble. It may be a leader or a follower node. Once a client is connected, the node assigns a session ID to the particular client and sends an acknowledgement to the client. If the client does not get an acknowledgment, it simply tries to connect another node in the ZooKeeper ensemble. Once connected to a node, the client will send heartbeats to the node in a regular interval to make sure that the connection is not lost.

- **If a client wants to read a particular znode**, it sends a **read request** to the node with the znode path and the node returns the requested znode by getting it from its own database. For this reason, reads are fast in ZooKeeper ensemble.

- **If a client wants to store data in the ZooKeeper ensemble**, it sends the znode path and the data to the server. The connected server will forward the request to the leader and then the leader will reissue the writing request to all the followers. If only a majority of the nodes respond successfully, then the write request will succeed and a successful return code will be sent to the client. Otherwise, the write request will fail. The strict majority of nodes is called as **Quorum**.

Nodes in a ZooKeeper Ensemble

Let us analyze the effect of having different number of nodes in the ZooKeeper ensemble.

- If we have **a single node**, then the ZooKeeper ensemble fails when that node fails. It contributes to "Single Point of Failure" and it is not recommended in a production environment.

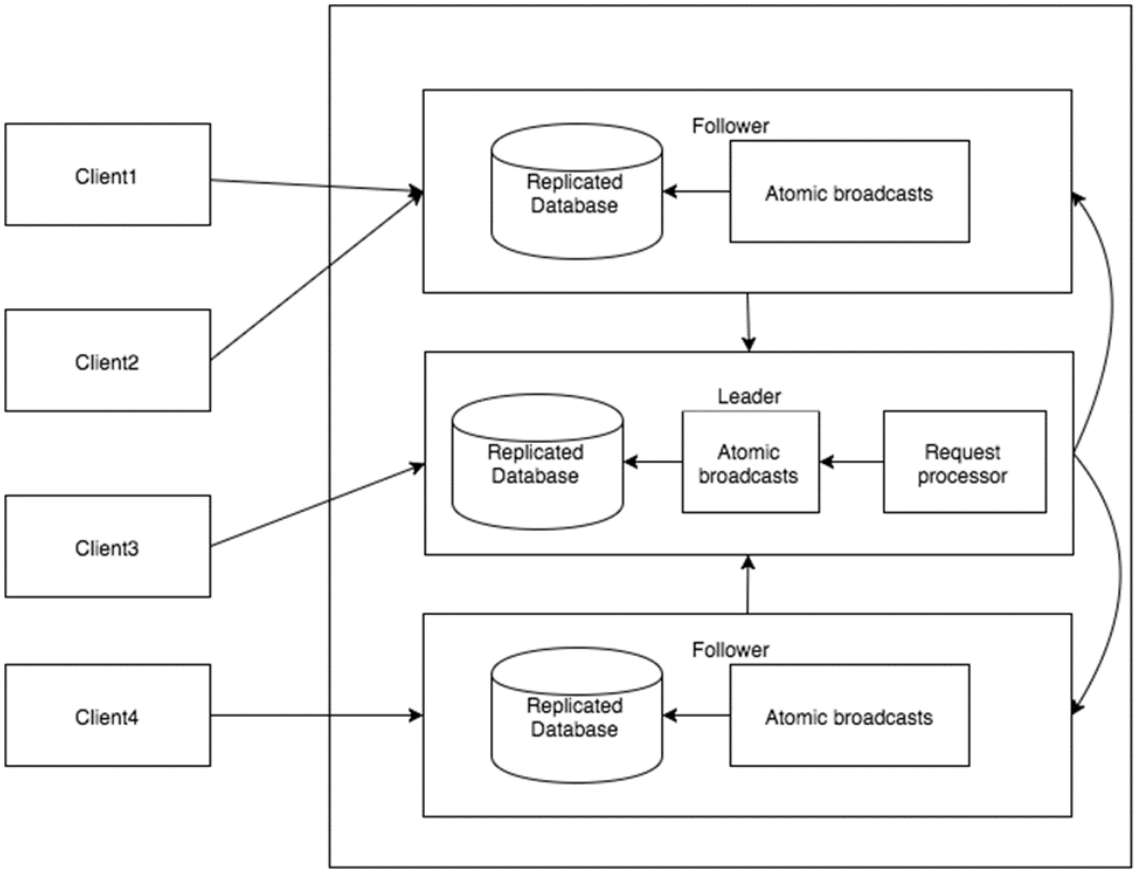
- If we have **two nodes** and one node fails, we don't have majority as well, since one out of two is not a majority.

- If we have **three nodes** and one node fails, we have majority and so, it is the minimum requirement. It is mandatory for a ZooKeeper ensemble to have at least three nodes in a live production environment.

- If we have **four nodes** and two nodes fail, it fails again and it is similar to having three nodes. The extra node does not serve any purpose and so, it is better to add nodes in odd numbers, e.g., 3, 5, 7.

We know that a write process is expensive than a read process in ZooKeeper ensemble, since all the nodes need to write the same data in its database. So, it is better to have less number of nodes (3, 5 or 7) than having a large number of nodes for a balanced environment.

The following diagram depicts the ZooKeeper WorkFlow and the subsequent table explains its different components.



Component	Description
Write	Write process is handled by the leader node. The leader forwards the write request to all the znodes and waits for answers from the znodes. If half of the znodes reply, then the write process is complete.
Read	Reads are performed internally by a specific connected znode, so there is no need to interact with the cluster.
Replicated Database	It is used to store data in zookeeper. Each znode has its own database and every znode has the same data at every time with the help of consistency.
Leader	Leader is the Znode that is responsible for processing write requests.
Follower	Followers receive write requests from the clients and forward them to the leader znode.
Request Processor	Present only in leader node. It governs write requests from the follower node.
Atomic broadcasts	Responsible for broadcasting the changes from the leader node to the follower nodes.

4. ZOOKEEPER – LEADER ELECTION

Let us analyze how a leader node can be elected in a ZooKeeper ensemble. Consider there are **N** number of nodes in a cluster. The process of leader election is as follows:

1. All the nodes create a sequential, ephemeral znode with the same path, **/app/leader_election/guid_**.

2. ZooKeeper ensemble will append the 10-digit sequence number to the path and the znode created will be **/app/leader_election/guid_0000000001**, **/app/leader_election/guid_0000000002**, etc.

3. For a given instance, the node which creates the smallest number in the znode becomes the leader and all the other nodes are followers.

4. Each follower node watches the znode having the next smallest number. For example, the node which creates znode **/app/leader_election/guid_0000000008** will watch the znode **/app/leader_election/guid_0000000007** and the node which creates the znode **/app/leader_election/guid_0000000007** will watch the znode **/app/leader_election/guid_0000000006**.

5. If the leader goes down, then its corresponding znode **/app/leader_electionN** gets deleted.

6. The next in line follower node will get the notification through watcher about the leader removal.

7. The next in line follower node will check if there are other znodes with the smallest number. If none, then it will assume the role of the leader. Otherwise, it finds the node which created the znode with the smallest number as leader.

8. Similarly, all other follower nodes elect the node which created the znode with the smallest number as leader.

Leader election is a complex process when it is done from scratch. But ZooKeeper service makes it very simple. Let us move on to the installation of ZooKeeper for development purpose in the next chapter.

5. ZOOKEEPER – INSTALLATION

Before installing ZooKeeper, make sure your system is running on any of the following operating systems:

- Any of Linux OS – Supports development and deployment. It is preferred for demo applications.
- Windows OS – Supports only development.
- Mac OS – Supports only development.

ZooKeeper server is created in Java and it runs on JVM. You need to use JDK 6 or greater.

Now, follow the steps given below to install ZooKeeper framework on your machine.

Step 1: Verifying Java Installation

We believe you already have a Java environment installed on your system. Just verify it using the following command.

```
$ java -version
```

If you have Java installed on your machine, then you could see the version of installed Java. Otherwise, follow the simple steps given below to install the latest version of Java.

Step 1.1: Download JDK

Download the latest version of JDK by visiting the following link and download the latest version.

http://www.oracle.com/technetwork/java/javase/downloads/index.html

The latest version (while writing this tutorial) is JDK 8u 60 and the file is "jdk-8u60-linux-x64.tar.gz". Please download the file on your machine.

Step 1.2: Extract the files

Generally, files are downloaded to the **downloads** folder. Verify it and extract the tar setup using the following commands.

```
$ cd /go/to/download/path
$ tar  -zxf  jdk-8u60-linux-x64.gz
```

Step 1.3: Move to opt directory

To make Java available to all users, move the extracted java content to "/usr/local/java" folder.

```
$ su
```

```
password: (type password of root user)
$ mkdir /opt/jdk
$ mv jdk-1.8.0_60 /opt/jdk/
```

Step 1.4: Set path

To set path and JAVA_HOME variables, add the following commands to ~/.bashrc file.

```
export JAVA_HOME =/usr/jdk/jdk-1.8.0_60
export PATH=$PATH:$JAVA_HOME/bin
```

Now, apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step 1.5: Java alternatives

Use the following command to change Java alternatives.

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_60/bin/java 100
```

Step 1.6

Verify the Java installation using the verification command (**java -version**) explained in Step 1.

Step 2: ZooKeeper Framework Installation

Step 2.1: Download ZooKeeper

To install ZooKeeper framework on your machine, visit the following link and download the latest version of ZooKeeper. http://zookeeper.apache.org/releases.html

As of now, the latest version of ZooKeeper is 3.4.6 (ZooKeeper-3.4.6.tar.gz).

Step 2.2: Extract the tar file

Extract the tar file using the following commands:

```
$ cd opt/
$ tar  -zxf  zookeeper-3.4.6.tar.gz
$ cd zookeeper-3.4.6
$ mkdir data
```

Step 2.3: Create configuration file

Open the configuration file named **conf/zoo.cfg** using the command **vi conf/zoo.cfg** and all the following parameters to set as starting point.

```
$ vi conf/zoo.cfg


tickTime=2000
dataDir=/path/to/zookeeper/data
clientPort=2181
initLimit=5
syncLimit=2
```

Once the configuration file has been saved successfully, return to the terminal again. You can now start the zookeeper server.

Step 2.4: Start ZooKeeper server

Execute the following command:

```
$ bin/zkServer.sh start
```

After executing this command, you will get a response as follows:

```
$ JMX enabled by default
$ Using config: /Users/../zookeeper-3.4.6/bin/../conf/zoo.cfg
$ Starting zookeeper ... STARTED
```

Step 2.5: Start CLI

Type the following command:

```
$ bin/zkCli.sh
```

After typing the above command, you will be connected to the ZooKeeper server and you should get the following response.

```
Connecting to localhost:2181
.................
.................
.................
Welcome to ZooKeeper!
.................
.................
```

```
WATCHER::


WatchedEvent state:SyncConnected type: None path:null
[zk: localhost:2181(CONNECTED) 0]
```

Stop ZooKeeper Server

After connecting the server and performing all the operations, you can stop the zookeeper server by using the following command.

```
$ bin/zkServer.sh stop
```

6. ZOOKEEPER – CLI

ZooKeeper Command Line Interface (CLI) is used to interact with the ZooKeeper ensemble for development purpose. It is useful for debugging and working around with different options.

To perform ZooKeeper CLI operations, first turn on your ZooKeeper server (*"bin/zkServer.sh start")* and then, ZooKeeper client (*"bin/zkCli.sh"*). Once the client starts, you can perform the following operation:

- Create znodes
- Get data
- Watch znode for changes
- Set data
- Create children of a znode
- List children of a znode
- Check Status
- Remove / Delete a znode

Now let us see above command one by one with an example.

Create Znodes

Create a znode with the given path. The **flag** argument specifies whether the created znode will be ephemeral, persistent, or sequential. By default, all znodes are persistent.

- **Ephemeral znodes** (flag: *e*) will be automatically deleted when a session expires or when the client disconnects.

- **Sequential znodes** guaranty that the znode path will be unique.

- ZooKeeper ensemble will add sequence number along with 10 digit padding to the znode path. For example, the znode path */myapp* will be converted to */myapp0000000001* and the next sequence number will be */myapp0000000002*. If no flags are specified, then the znode is considered as **persistent**.

Syntax

```
create  /path /data
```

Sample

```
create /FirstZnode "Myfirstzookeeper-app"
```

Output

```
[zk: localhost:2181(CONNECTED) 0] create /FirstZnode “Myfirstzookeeper-app”
Created /FirstZnode
```

To create a **Sequential znode**, add **-s flag** as shown below.

Syntax

```
create -s /path /data
```

Sample

```
create -s /FirstZnode second-data
```

Output

```
[zk: localhost:2181(CONNECTED) 2] create -s /FirstZnode “second-data”
Created /FirstZnode0000000023
```

To create an **Ephemeral znode**, add **-e flag** as shown below.

Syntax

```
create -e /path /data
```

Sample

```
create -e /SecondZnode “Ephemeral-data”
```

Output

```
[zk: localhost:2181(CONNECTED) 2] create -e /SecondZnode “Ephemeral-data”
Created /SecondZnode
```

Remember when a client connection is lost, the ephemeral znode will be deleted. You can try it by quitting the ZooKeeper CLI and then re-opening the CLI.

Get Data

It returns the associated data of the znode and metadata of the specified znode. You will get information such as when the data was last modified, where it was modified, and information about the data. This CLI is also used to assign watches to show notification about the data.

Syntax

```
get /path
```

Sample

```
get /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode
“Myfirstzookeeper-app”
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 16:15:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 22
numChildren = 0
```

To access a sequential znode, you must enter the full path of the znode.

Sample

```
get /FirstZnode0000000023
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode0000000023
“Second-data”
cZxid = 0x80
ctime = Tue Sep 29 16:25:47 IST 2015
mZxid = 0x80
mtime = Tue Sep 29 16:25:47 IST 2015
pZxid = 0x80
```

```
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 13
numChildren = 0
```

Watch

Watches show a notification when the specified znode or znode's children data changes. You can set a **watch** only in **get** command.

Syntax

```
get  /path [watch] 1
```

Sample

```
get /FirstZnode 1
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode 1
"Myfirstzookeeper-app"
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 16:15:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 22
numChildren = 0
```

The output is similar to normal **get** command, but it will wait for znode changes in the background. <Start here>

Set Data

Set the data of the specified znode. Once you finish this set operation, you can check the data using the **get** CLI command.

Syntax

```
set /path /data
```

Sample

```
set /SecondZnode Data-updated
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /SecondZnode “Data-updated”
cZxid = 0x82
ctime = Tue Sep 29 16:29:50 IST 2015
mZxid = 0x83
mtime = Tue Sep 29 16:29:50 IST 2015
pZxid = 0x82
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x15018b47db00000
dataLength = 14
numChildren = 0
```

If you assigned **watch** option in **get** command (as in previous command), then the output will be similar as shown below:

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode “Mysecondzookeeper-app”


WATCHER: :


WatchedEvent state:SyncConnected type:NodeDataChanged path:/FirstZnode
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
```

```
mZxid = 0x84
mtime = Tue Sep 29 17:14:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 23
numChildren = 0
```

Create Children / Sub-znode

Creating children is similar to creating new znodes. The only difference is that the path of the child znode will have the parent path as well.

Syntax

```
create /parent/path/subnode/path  /data
```

Sample

```
create /FirstZnode/Child1 firstchildren
```

Output

```
[zk: localhost:2181(CONNECTED) 16] create /FirstZnode/Child1 “firstchildren”
created /FirstZnode/Child1
[zk: localhost:2181(CONNECTED) 17] create /FirstZnode/Child2 “secondchildren”
created /FirstZnode/Child2
```

List Children

This command is used to list and display the **children** of a znode.

Syntax

```
ls /path
```

Sample

```
ls /MyFirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 2] ls /MyFirstZnode
[mysecondsubnode, myfirstsubnode]
```

Check Status

Status describes the metadata of a specified znode. It contains details such as Timestamp, Version number, ACL, Data length, and Children znode.

Syntax

```
stat /path
```

Sample

```
stat /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 1] stat /FirstZnode
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 17:14:24 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 23
numChildren = 0
```

Remove a Znode

Removes a specified znode and recursively all its children. This would happen only if such a znode is available.

Syntax

```
rmr /path
```

Sample

```
rmr /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 10] rmr /FirstZnode
[zk: localhost:2181(CONNECTED) 11] get /FirstZnode
Node does not exist: /FirstZnode
```

Delete (**delete /path**) command is similar to **remove** command, except the fact that it works only on znodes with no children.

7. ZOOKEEPER – API

ZooKeeper has an official API binding for Java and C. The ZooKeeper community provides unofficial API for most of the languages (.NET, python, etc.). Using ZooKeeper API, an application can connect, interact, manipulate data, coordinate, and finally disconnect from a ZooKeeper ensemble.

ZooKeeper API has a rich set of features to get all the functionality of the ZooKeeper ensemble in a simple and safe manner. ZooKeeper API provides both synchronous and asynchronous methods.

ZooKeeper ensemble and ZooKeeper API completely complement each other in every aspect and it benefits the developers in a great way. Let us discuss Java binding in this chapter.

Basics of ZooKeeper API

Application interacting with ZooKeeper ensemble is referred as **ZooKeeper Client** or simply **Client**.

Znode is the core component of ZooKeeper ensemble and ZooKeeper API provides a small set of methods to manipulate all the details of znode with ZooKeeper ensemble.

A client should follow the steps given below to have a clear and clean interaction with ZooKeeper ensemble.

- Connect to the ZooKeeper ensemble. ZooKeeper ensemble assign a Session ID for the client.
- Send heartbeats to the server periodically. Otherwise, the ZooKeeper ensemble expires the Session ID and the client needs to reconnect.
- Get / Set the znodes as long as a session ID is active.
- Disconnect from the ZooKeeper ensemble, once all the tasks are completed. If the client is inactive for a prolonged time, then the ZooKeeper ensemble will automatically disconnect the client.

Java Binding

Let us understand the most important set of ZooKeeper API in this chapter. The central part of the ZooKeeper API is **ZooKeeper class**. It provides options to connect the ZooKeeper ensemble in its constructor and has the following methods:

- **connect** – connect to the ZooKeeper ensemble
- **create** – create a znode
- **exists** – check whether a znode exists and its information
- **getData** – get data from a particular znode
- **setData** – set data in a particular znode

- **getChildren** – get all sub-nodes available in a particular znode
- **delete** – get a particular znode and all its children
- **close** – close a connection

Connect to the ZooKeeper Ensemble

The ZooKeeper class provides connection functionality through its constructor. The signature of the constructor is as follows:

```
ZooKeeper(String connectionString, int sessionTimeout, Watcher watcher)
```

Where,

- **connectionString** – ZooKeeper ensemble host.
- **sessionTimeout** – session timeout in milliseconds.
- **watcher** – an object implementing "Watcher" interface. The ZooKeeper ensemble returns the connection status through the watcher object.

Let us create a new helper class **ZooKeeperConnection** and add a method **connect**. The **connect** method creates a ZooKeeper object, connects to the ZooKeeper ensemble, and then returns the object.

Here **CountDownLatch** is used to stop (wait) the main process until the client connects with the ZooKeeper ensemble.

The ZooKeeper ensemble replies the connection status through the **Watcher callback**. The Watcher callback will be called once the client connects with the ZooKeeper ensemble and the Watcher callback calls the **countDown** method of the **CountDownLatch** to release the lock, **await** in the main process.

Here is the complete code to connect with a ZooKeeper ensemble.

Coding: ZooKeeperConnection.java

```
// import java classes
import java.io.IOException;
import java.util.concurrent.CountDownLatch;

// import zookeeper classes
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.ZooKeeper;
```

```
import org.apache.zookeeper.AsyncCallback.StatCallback;
import org.apache.zookeeper.KeeperException.Code;
import org.apache.zookeeper.data.Stat;

public class ZooKeeperConnection {

     // declare zookeeper instance to access ZooKeeper ensemble
     private ZooKeeper zoo;

     final CountDownLatch connectedSignal = new CountDownLatch(1);

     // Method to connect zookeeper ensemble.
     public ZooKeeper connect(String host) throws IOException,InterruptedException {
           zoo = new ZooKeeper(host,5000,new Watcher() {
                 public void process(WatchedEvent we) {
                       if (we.getState() == KeeperState.SyncConnected) {
                             connectedSignal.countDown();
                       }
                 }
           });

           connectedSignal.await();
           return zoo;
     }

     // Method to disconnect from zookeeper server
     public void close() throws InterruptedException {
           zoo.close();
     }
}
```

Save the above code and it will be used in the next section for connecting the ZooKeeper ensemble.

Create a Znode

The ZooKeeper class provides **create method** to create a new znode in the ZooKeeper ensemble. The signature of the **create** method is as follows:

```
create(String path, byte[] data, List<ACL> acl, CreateMode createMode)
```

Where,

- **path** – Znode path. For example, /myapp1, /myapp2, /myapp1/mydata1, myapp2/mydata1/myanothersubdata
- **data** – data to store in a specified znode path
- **acl** – access control list of the node to be created. ZooKeeper API provides a static interface **ZooDefs.Ids** to get some of basic acl list. For example, ZooDefs.Ids.OPEN_ACL_UNSAFE returns a list of acl for open znodes.
- **createMode** – the type of node, either ephemeral, sequential, or both. This is an **enum**.

Let us create a new Java application to check the **create** functionality of the ZooKeeper API. Create a file **ZKCreate.java**. In the main method, create an object of type **ZooKeeperConnection** and call the **connect** method to connect to the ZooKeeper ensemble.

The connect method will return the ZooKeeper object **zk**. Now, call the **create** method of **zk** object with custom **path** and **data**.

The complete program code to create a znode is as follows:

Coding: ZKCreate.java

```
import java.io.IOException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.ZooDefs;

public class ZKCreate {

     // create static instance for zookeeper class.
     private static ZooKeeper zk;
```

```
     // create static instance for ZooKeeperConnection class.
     private static ZooKeeperConnection conn;

     // Method to create znode in zookeeper ensemble
     public static void create(String path, byte[] data) throws
KeeperException,InterruptedException {
            zk.create(path, data, ZooDefs.Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);
     }

     public static void main(String[] args)
     {
            // znode path
            String path = "/MyFirstZnode";                // Assign path to znode

            // data in byte array
            byte[] data = "My first zookeeper app”.getBytes();    // Declare data

            try {
                  conn = new ZooKeeperConnection();
                  zk = conn.connect("localhost");

                  create(path, data);       // Create the data to the specified path

                  conn.close();

            } catch (Exception e) {
                  System.out.println(e.getMessage());    //Catch error message
            }
     }
}
```

Once the application is compiled and executed, a znode with the specified data will be created in the ZooKeeper ensemble. You can check it using the ZooKeeper CLI **zkCli.sh**.

```
cd /path/to/zookeeper
```

```
bin/zkCli.sh
>>> get /MyFirstZnode
```

Exists – Check the Existence of a Znode

The ZooKeeper class provides the **exists method** to check the existence of a znode. It returns the metadata of a znode, if the specified znode exists. The signature of the **exists** method is as follows:

```
exists(String path, boolean watcher)
```

Where,

- **path** – Znode path
- **watcher** – boolean value to specify whether to watch a specified znode or not

Let us create a new Java application to check the "*exists*" functionality of the ZooKeeper API. Create a file "*ZKExists.java*". In the main method, create ZooKeeper object, "*zk*" using "*ZooKeeperConnection" object*. Then, call "*exists*" method of "*zk*" object with custom "*path*". The complete listing is as follow

Coding: ZKExists.java

```
import java.io.IOException;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.data.Stat;

public class ZKExists {

     private static ZooKeeper zk;
     private static ZooKeeperConnection conn;

     // Method to check existence of znode and its status, if znode is available.
     public static Stat znode_exists(String path) throws
KeeperException,InterruptedException {
            return zk.exists(path, true);
     }
```

```
     public static void main(String[] args) throws
InterruptedException,KeeperException {

            String path= "/MyFirstZnode";    // Assign znode to the specified path

            try {
                  conn = new ZooKeeperConnection();
                  zk = conn.connect("localhost");

                  Stat stat = znode_exists(path);     // Stat checks the path of the znode
                  if(stat!= null) {
                        System.out.println("Node exists and the node version is " +
stat.getVersion());
                  } else {
                        System.out.println("Node does not exists");
                  }
            }
            catch(Exception e) {
                  System.out.println(e.getMessage());    // Catches error messages
            }
     }
}
```

Once the application is compiled and executed, you will get the below output.

```
Node exists and the node version is 1.
```

getData Method

The ZooKeeper class provides **getData** method to get the data attached in a specified znode and its status. The signature of the **getData** method is as follows:

```
getData(String path, Watcher watcher, Stat stat)
```

Where,

- **path** – Znode path.

- **watcher** – Callback function of type **Watcher**. The ZooKeeper ensemble will notify through the Watcher callback when the data of the specified znode changes. This is one-time notification.

- **stat** – Returns the metadata of a znode.

Let us create a new Java application to understand the **getData** functionality of the ZooKeeper API. Create a file **ZKGetData.java**. In the main method, create a ZooKeeper object **zk** using the **ZooKeeperConnection** object. Then, call the **getData** method of **zk** object with custom **path**.

Here is the complete program code to get the data from a specified node:

Coding: ZKGetData.java

```
import java.io.IOException;
import java.util.concurrent.CountDownLatch;


import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.data.Stat;


public class ZKGetData {


     private static ZooKeeper zk;
     private static ZooKeeperConnection conn;


     public static Stat znode_exists(String path) throws
KeeperException,InterruptedException {
            return zk.exists(path,true);
     }


     public static void main(String[] args) throws InterruptedException, KeeperException {


            String path = "/MyFirstZnode";
            final CountDownLatch connectedSignal = new CountDownLatch(1);
```

```
            try {
                  conn = new ZooKeeperConnection();
                  zk = conn.connect("localhost");

                  Stat stat = znode_exists(path);
                  if(stat != null) {
                        byte[] b = zk.getData(path, new Watcher() {
                              public void process(WatchedEvent we) {
                                    if (we.getType() == Event.EventType.None) {
                                          switch(we.getState()) {
                                                case Expired:
                                                      connectedSignal.countDown();
                                                      break;
                                          }
                                    } else {
                                          String path = "/MyFirstZnode";

                                          try {
                                                byte[] bn = zk.getData(path,
false, null);
                                                String data = new String(bn,
"UTF-8");
                                                System.out.println(data);

                                                connectedSignal.countDown();
                                          } catch(Exception ex) {
                                                System.out.println(ex.getMessage());
                                          }
                                    }
                              }
                        }, null);
                        String data = new String(b, "UTF-8");
                        System.out.println(data);

                        connectedSignal.await();
```

```
                } else {
                        System.out.println("Node does not exists");
                }
            }
            catch(Exception e) {
                System.out.println(e.getMessage());
            }
      }
}
```

Once the application is compiled and executed, you will get the following output.

```
My first zookeeper app
```

And the application will wait for further notification from the ZooKeeper ensemble. Change the data of the specified znode using ZooKeeper CLI **zkCli.sh**.

```
cd /path/to/zookeeper
bin/zkCli.sh
>>> set /MyFirstZnode Hello
```

Now, the application will print the following output and exit.

```
Hello
```

setData Method

The ZooKeeper class provides **setData** method to modify the data attached in a specified znode. The signature of the **setData** method is as follows:

```
setData(String path, byte[] data, int version)
```

Where,

- **path** – Znode path
- **data** – data to store in a specified znode path.
- **version** – Current version of the znode. ZooKeeper updates the version number of the znode whenever the data gets changed.

Let us now create a new Java application to understand the **setData** functionality of the ZooKeeper API. Create a file **ZKSetData.java**. In the main method, create a ZooKeeper object **zk** using the **ZooKeeperConnection** object. Then, call the **setData** method of **zk** object with the specified **path**, new data, and version of the node.

Here is the complete program code to modify the data attached in a specified znode.

Code: ZKSetData.java

```
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import java.io.IOException;

public class ZKSetData {

     private static ZooKeeper zk;
     private static ZooKeeperConnection conn;

     // Method to update the data in a znode. Similar to getData but without watcher.
     public static void update(String path, byte[] data) throws
KeeperException,InterruptedException {
            zk.setData(path, data, zk.exists(path,true).getVersion());
     }

     public static void main(String[] args) throws InterruptedException,KeeperException {

            String path= "/MyFirstZnode";

            byte[] data = "Success".getBytes();     //Assign data which is to be updated.

            try {

                  conn = new ZooKeeperConnection();
                  zk = conn.connect("localhost");

                  update(path, data);        // Update znode data to the specified path
```

```
            }
            catch(Exception e) {
                  System.out.println(e.getMessage());
            }
      }
}
```

Once the application is compiled and executed, the data of the specified znode will be changed and it can be checked using the ZooKeeper CLI, **zkCli.sh**.

```
cd /path/to/zookeeper
bin/zkCli.sh
>>> get /MyFirstZnode
```

getChildren Method

The ZooKeeper class provides **getChildren** method to get all the sub-node of a particular znode. The signature of the **getChildren** method is as follows:

```
getChildren(String path, Watcher watcher)
```

Where,

- **path** – Znode path.
- **watcher** – Callback function of type "Watcher". The ZooKeeper ensemble will notify when the specified znode gets deleted or a child under the znode gets created / deleted. This is a one-time notification.

Coding: ZKGetChildren.java

```
import java.io.IOException;
import java.util.*;

import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.data.Stat;
```

```
public class ZKGetChildren {

     private static ZooKeeper zk;
     private static ZooKeeperConnection conn;

     // Method to check existence of znode and its status, if znode is available.
     public static Stat znode_exists(String path) throws
KeeperException,InterruptedException {
            return zk.exists(path,true);
     }

     public static void main(String[] args) throws InterruptedException,KeeperException {

            String path= "/MyFirstZnode";                 // Assign path to the znode

            try {
                  conn = new ZooKeeperConnection();
                  zk = conn.connect("localhost");

                  Stat stat = znode_exists(path);        // Stat checks the path

                  if(stat!= null) {

                  //“getChildren” method- get all the children of znode.It has two
args, path and watch
                        List<String> children = zk.getChildren(path, false);
                        for(int i = 0; i < children.size(); i++)
                              System.out.println(children.get(i));    //Print
children's
                  } else {
                        System.out.println("Node does not exists");
                  }
            }
            catch(Exception e) {
                  System.out.println(e.getMessage());
            }
```

```
    }
}
```

Before running the program, let us create two sub-nodes for **/MyFirstZnode** using the ZooKeeper CLI, **zkCli.sh**.

```
cd /path/to/zookeeper
bin/zkCli.sh
>>> create /MyFirstZnode/myfirstsubnode Hi
>>> create /MyFirstZnode/mysecondsubmode Hi
```

Now, compiling and running the program will output the above created znodes.

```
myfirstsubnode
mysecondsubnode
```

Delete a Znode

The ZooKeeper class provides **delete** method to delete a specified znode. The signature of the **delete** method is as follows:

```
delete(String path, int version)
```

Where,

- **path** – Znode path.
- **version** – Current version of the znode.

Let us create a new Java application to understand the **delete** functionality of the ZooKeeper API. Create a file **ZKDelete.java**. In the main method, create a ZooKeeper object **zk** using **ZooKeeperConnection** object. Then, call the **delete** method of **zk** object with the specified **path** and version of the node.

The complete program code to delete a znode is as follows:

Coding: ZKDelete.java

```
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.KeeperException;

public class ZKDelete {
```

```
   private static ZooKeeper zk;

   private static ZooKeeperConnection conn;

   // Method to check existence of znode and its status, if znode is available.
   public static void delete(String path) throws KeeperException,InterruptedException {
         zk.delete(path,zk.exists(path,true).getVersion());
   }

   public static void main(String[] args) throws InterruptedException,KeeperException {

         String path= "/MyFirstZnode";          //Assign path to the znode

         try{
               conn = new ZooKeeperConnection();
               zk = conn.connect("localhost");

               delete(path);          //delete the node with the specified path
         }
         catch(Exception e) {
               System.out.println(e.getMessage());    // catches error messages
         }
   }
}
```

8. ZOOKEEPER – APPLICATIONS

Zookeeper provides a flexible coordination infrastructure for distributed environment. ZooKeeper framework supports many of the today's best industrial applications. We will discuss some of the most notable applications of ZooKeeper in this chapter.

Yahoo!

The ZooKeeper framework was originally built at "Yahoo!". A well-designed distributed application needs to meet requirements such as data transparency, better performance, robustness, centralized configuration, and coordination. So, they designed the ZooKeeper framework to meet these requirements.

Apache Hadoop

Apache Hadoop is the driving force behind the growth of Big Data industry. Hadoop relies on ZooKeeper for configuration management and coordination. Let us take a scenario to understand the role of ZooKeeper in Hadoop.

Assume that a **Hadoop cluster** bridges **100 or more commodity servers**. Therefore, there's a need for coordination and naming services. As computation of large number of nodes are involved, each node needs to synchronize with each other, know where to access services, and know how they should be configured. At this point of time, Hadoop clusters require cross-node services. ZooKeeper provides the facilities for **cross-node synchronization** and ensures the tasks across Hadoop projects are serialized and synchronized.

Multiple ZooKeeper servers support large Hadoop clusters. Each client machine communicates with one of the ZooKeeper servers to retrieve and update its synchronization information. Some of the real-time examples are:

- **Human Genome Project** – The Human Genome Project contains terabytes of data. Hadoop MapReduce framework can be used to analyze the dataset and find interesting facts for human development.

- **Healthcare** – Hospitals can store, retrieve, and analyze huge sets of patient medical records, which are normally in terabytes.

Apache HBase

Apache HBase is an open source, distributed, NoSQL database used for real-time read/write access of large datasets and runs on top of the HDFS. HBase follows **master-slave architecture** where the HBase Master governs all the slaves. Slaves are referred as **Region servers**.

HBase distributed application installation depends on a running ZooKeeper cluster. Apache HBase uses ZooKeeper to track the status of distributed data throughout the master and region servers with the help of **centralized configuration management** and **distributed mutex** mechanisms. Here are some of the use-cases of HBase:

- **Telecom** – Telecom industry stores billions of mobile call records (around 30TB / month) and accessing these call records in real time become a huge task. HBase can be used to process all the records in real time, easily and efficiently.

- **Social network** – Similar to telecom industry, sites like Twitter, LinkedIn, and Facebook receive huge volumes of data through the posts created by users. HBase can be used to find recent trends and other interesting facts.

Apache Solr

Apache Solr is a fast, open source search platform written in Java. It is a blazing fast, fault-tolerant distributed search engine. Built on top of **Lucene**, it is a high-performance, full-featured text search engine.

Solr extensively uses every feature of ZooKeeper such as Configuration management, Leader election, node management, Locking and syncronization of data.

Solr has two distinct parts, **indexing** and **searching**. Indexing is a process of storing the data in a proper format so that it can be searched later. Solr uses ZooKeeper for both indexing the data in multiple nodes and searching from multiple nodes. ZooKeeper contributes the following features:

- Add / remove nodes as and when needed

- Replication of data between nodes and subsequently minimizing data loss

- Sharing of data between multiple nodes and subsequently searching from multiple nodes for faster search results

Some of the use-cases of Apache Solr include e-commerce, job search, etc.