ZooKeeper fundamentals, deployment, and applications

Mark Grover

August 27, 2013

Apache ZooKeeper is a high-performance coordination server for distributed applications. It exposes common services -- such as naming and configuration management, synchronization, and group services -- in a simple interface, relieving the user from the need to program from scratch. It comes with off-the-shelf support for implementing consensus, group management, leader election, and presence protocols. In this article, we will explore the fundamentals of ZooKeeper, then walk through a guide to set up and deploy a ZooKeeper cluster in a simulated miniature distributed environment. We will conclude with examples of how ZooKeeper is used in popular projects.

Let's start with why you would want to use ZooKeeper. ZooKeeper is a building block for distributed systems. When designing a distributed system, there is typically a need for designing and developing some coordination services:

- **Name service**— A name service is a service that maps a name to some information associated with that name. A telephone directory is a name service that maps the name of a person to his/her telephone number. In the same way, a DNS service is a name service that maps a domain name to an IP address. In your distributed system, you may want to keep a track of which servers or services are up and running and look up their status by name. ZooKeeper exposes a simple interface to do that. A name service can also be extended to a group membership service by means of which you can obtain information pertaining to the group associated with the entity whose name is being looked up.
- **Locking**— To allow for serialized access to a shared resource in your distributed system, you may need to implement distributed mutexes. ZooKeeper provides for an easy way for you to implement them.
- **Synchronization**— Hand in hand with distributed mutexes is the need for synchronizing access to shared resources. Whether implementing a producer-consumer queue or a barrier, ZooKeeper provides for a simple interface to implement that. Check out the examples on the Apache ZooKeeper wiki on how to do so (see Related topics).
- **Configuration management**— You can use ZooKeeper to centrally store and manage the configuration of your distributed system. This means that any new nodes joining will pick up the up-to-date centralized configuration from ZooKeeper as soon as they join the system. This also allows you to centrally change the state of your distributed system by changing the centralized configuration through one of the ZooKeeper clients.

- **Leader election**— Your distributed system may have to deal with the problem of nodes going down, and you may want to implement an automatic fail-over strategy. ZooKeeper provides off-the-shelf support for doing so via leader election.

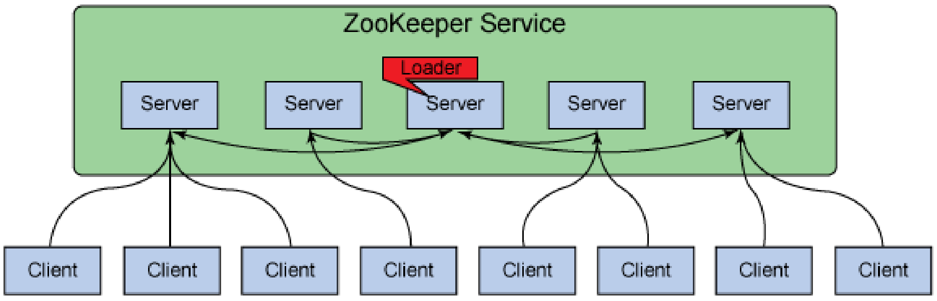
While it's possible to design and implement all of these services from scratch, it's extra work and difficult to debug any problems, race conditions, or deadlocks. Just like you don't go around writing your own random number generator or hashing function in your code, there was a need that people shouldn't go around writing their own name services or leader election services from scratch every time they need it. Moreover, you could hack together a very simple group membership service relatively easily, but it would require much more work to write it to provide reliability, replication, and scalability. This led to the development and open sourcing of Apache ZooKeeper, an out-of-the box reliable, scalable, and high-performance coordination service for distributed systems.

InfoSphere® BigInsights™ Quick Start Edition is IBM's big data offering based upon the open source Apache Hadoop project. It includes ZooKeeper, along with other big data technologies as well as IBM technologies that extend the platform's value. In this article, we are just using ZooKeeper, but see Related topics for more on InfoSphere BigInsights, including a link to download the product.

ZooKeeper, while being a coordination service for distributed systems, is a distributed application on its own. ZooKeeper follows a simple client-server model where *clients* are nodes (i.e., machines) that make use of the service, and *servers* are nodes that provide the service. A collection of ZooKeeper servers forms a ZooKeeper *ensemble.* At any given time, one ZooKeeper client is connected to one ZooKeeper server. Each ZooKeeper server can handle a large number of client connections at the same time. Each client periodically sends pings to the ZooKeeper server it is connected to let it know that it is alive and connected. The ZooKeeper server in question responds with an acknowledgment of the ping, indicating the server is alive as well. When the client doesn't receive an acknowledgment from the server within the specified time, the client connects to another server in the ensemble, and the client session is transparently transferred over to the new ZooKeeper server.

Figure 1 depicts the client-server architecture of ZooKeeper.

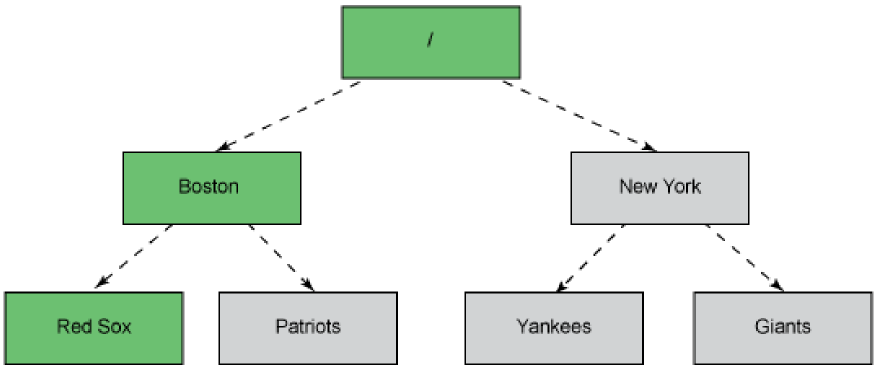
Figure 1. Client-server architecture of ZooKeeper



ZooKeeper has a file system-like data model composed of *znodes.* Think of znodes (ZooKeeper data nodes) as files in a traditional UNIX-like system, except that they can have child nodes.

Another way to look at them is as directories that can have data associated with themselves. Each of these directories is called a znode. Figure 2 shows a diagram representing the same hierarchy as sports team in two cities.

Figure 2. Diagram representing hierarchy of sports team in two cities



The znode hierarchy is stored in memory within each of the ZooKeeper servers. This allows for scalable and quick responses to reads from the clients. Each ZooKeeper server also maintains a transaction log on the disk, which logs all write requests. This transaction log is also the most performance critical part of ZooKeeper because a ZooKeeper server must sync transactions to disk before it returns a successful response. The default maximum size of data that can be stored in a znode is 1 MB. Consequently, even though ZooKeeper presents a file system-like hierarchy, it shouldn't be used as a general-purpose file system. Instead, it should only be used as a storage mechanism for the small amount of data required for providing reliability, availability, and coordination to your distributed application.

When a client requests to read the contents of a particular znode, the read takes place at the server that the client is connected to. Consequently, since only one server from the ensemble is involved, reads are quick and scalable. However, for writes to be completed successfully, a strict majority of the nodes of the ZooKeeper ensemble are required to be available. When the ZooKeeper service is brought up, one node from the ensemble is elected as the leader. When a client issues a write request, the connected server passes on the request to the leader. This leader then issues the same write request to all the nodes of the ensemble. If a strict majority of the nodes (also known as a *quorum*) respond successfully to this write request, the write request is considered to have succeeded. A successful return code is then returned to the client who initiated the write request. If a quorum of nodes are not available in an ensemble, the ZooKeeper service is nonfunctional.

A quorum is represented by a strict majority of nodes. You can have one node in your ensemble, but it won't be a highly available or reliable system. If you have two nodes in your ensemble, you would need both to be up to keep the service running because one out of two nodes is not a strict majority. If you have three nodes in the ensemble, one can go down, and you would still have a functioning service (two out of three is a strict majority). For this reason, ZooKeeper ensembles typically contain an odd number of nodes because having four nodes gives you no benefit over having three with respect to fault-tolerance. As soon as two nodes go down, your ZooKeeper service goes down. On five-node cluster, you would need three to go down before the ZooKeeper service stops functioning.

Now that we have figured out that we should have an odd number of nodes, let's figure out how many nodes we need in our ZooKeeper ensemble. Reads are always read from the ZooKeeper server connected to the client, so their performance doesn't change as the number of servers in the ensemble changes. However, writes are successful only when written to a quorum of nodes. This means that the write performance decreases as the number of nodes in the ensemble increases since writes have to be written to and coordinated among more servers.

The beauty of ZooKeeper is that it is up to you how many servers you want to run. If you would like to run one server, that's fine from ZooKeeper's perspective; you just won't have a highly reliable or available system. A three-node ZooKeeper ensemble will support one failure without loss of service, which is probably fine for most users and arguably the most common deployment topology. However, to be safe, use five nodes in your ensemble. A five-node ensemble allows you to take one server out for maintenance or rolling upgrade and still be able to withstand a second unexpected failure without interruption of service.

Consequently, three, five, or seven is the most typical number of nodes in a ZooKeeper ensemble. Keep in mind that the size of your ZooKeeper ensemble has little to do with the size of the nodes in your distributed system. The nodes in your distributed system would be clients to the ZooKeeper ensemble, and each ZooKeeper server can handle a large number of clients scalably. For example, HBase (a distributed database on Hadoop) relies upon ZooKeeper for leader election and lease management of region servers. You can have a large 50-node HBase cluster running with a relatively small (say, five) node ZooKeeper ensemble.

Set up and deploy a ZooKeeper ensemble

Now let's set up and deploy a ZooKeeper ensemble with three nodes. Here, we will be using the latest version of ZooKeeper at the time of writing: 3.4.5 (see Related topics for download information). The nodes we will use for this demo are named zkserver1.mybiz.com, zkserver2.mybiz.com, and zk3server3.mybiz.com. Here are the steps you would need to follow on each of the nodes to start the ZooKeeper server:

1. Download the install JDK, if not already installed (see Related topics). This is required because ZooKeeper server runs on JVM.
2. Download the ZooKeeper 3.4.5. tar.gz tarball and un-tar it to an appropriate location.

Listing 1. Downloading the ZooKeeper tarball and un-tarring to the appropriate location

```
wget http://www.bizdirusa.com/mirrors/apache/ZooKeeper/stable/zookeeper3.4.5.
tar.gz tar xzvf zookeeper3.4.5.tar.gz
```

3. Create a directory for storing some state associated with the ZooKeeper server: `mkdir /var/lib/zookeeper`. You may need to create this directory as root and later change the owner of this directory to the user you'd like to run ZooKeeper server as.
4. Set up the configuration. Create or edit the zookeeper3.4.5/conf/zoo.cfg file to look something like Listing 2.

Listing 2. Setting up the configuration

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zkserver1.mybiz.com:2888:3888
server.2=zkserver2.mybiz.com:2888:3888
server.3=zkserver3.mybiz.com:2888:3888
```

An important thing to note is that ports 2181, 2888, and 3888 should be open across all three machines. In this example, config, port 2181 is used by ZooKeeper clients to connect to the ZooKeeper servers, port 2888 is used by peer ZooKeeper servers to communicate with each other, and port 3888 is used for leader election. You may chose any ports of your liking. It's usually recommended that you use the same ports on all of the ZooKeeper servers.

5. Create a /var/lib/zookeeper/myid file. The contents of this file would be just the numeral 1 on zkserver1.mybiz.com, numeral 2 on zkserver2.mybiz.com, and numeral 3 onzkserver3.mybiz.com. Listing 3 shows the output of cat on this file from zkserver1.mybiz.com.

Listing 3. Output of cat

```
mark@zkserver1.mybiz.com:~# cat /var/lib/zookeeper/myid
1
```

Now you are ready to start the ZooKeeper servers on each of these machines.

Listing 4. Starting the ZooKeeper servers

```
zookeeper3.4.5/
bin/zkServer.sh start
```

Now, you can start a CLI client from one of the machines you are running the ZooKeeper server.

Listing 5. Starting the CLI client

```
zookeeper3.4.5/
bin/zkCli.sh server
zkserver1.mybiz.com:2181,zkserver2.mybiz.com:2181,zkserver3.mybiz.com:2181
```

The client supplies a list of servers, one of which is arbitrarily chosen for the connection. In case the connection to that server is lost down the road, another server from the list is picked and the client's session is transferred to that server. Once the client has been started, you can create, edit, and delete znodes. Let's create a znode at `/mynode` with `helloworld` as the associated data.

Listing 6. Creating a znode at `/mynode`

```
[zk: 127.0.0.1:2181(CONNECTED) 2] create /mynode
helloworld
Created /mynode
```

Now let's verify and retrieve the data at `/mynode`.

Listing 7. Verifying and retrieving the data at `/mynode`

```
[zk: 127.0.0.1:2181(CONNECTED) 6] get /mynode
helloworld
cZxid = 0x200000005
ctime = Sat Jul 20 19:53:52 PDT 2013
mZxid = 0x200000005
mtime = Sat Jul 20 19:53:52 PDT 2013
pZxid = 0x200000005
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 11
numChildren = 0
```

You would notice that when getting data for a znode, the client is also returned some metadata pertaining to the znode. Some important fields in this metadata are epoch timestamps related to when the znode was created and last modified (`ctime` and `mtime`), version of the data that changes every time the data is modified (`dataVersion`), the length of the data (`dataLength)`, the number of children of this znode (`numChildren`). We can now remove the znode.

Listing 8. Removing the znode

```
[zk: 127.0.0.1:2181(CONNECTED) 7] rmr /mynode
```

Let's create another znode at `/mysecondnode`.

Listing 9. Creating another znode

```
[zk: 127.0.0.1:2181(CONNECTED) 10] create /mysecondnode
hello
Created /mysecondnode
```

Now let's verify and retrieve the data at `/mysecondnode`. This time, we are supplying and optional parameter `1` at the end. This parameter sets a onetime trigger (called *watch*) for the data at `/mysecondnode`. If another client modifies the data at `/mysecondnode`, this client would get an asynchronous notification. Note that the notification is sent only once and no further notifications for the change in data would be sent unless the watch is set again.

Listing 10. Verify and retrieve the data at `/mysecondnode`

```
[zk: 127.0.0.1:2181(CONNECTED) 12] get /mysecondnode
1
hello
cZxid = 0x200000007
ctime = Sat Jul 20 19:58:27 PDT 2013
mZxid = 0x200000007
mtime = Sat Jul 20 19:58:27 PDT 2013
pZxid = 0x200000007
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Now, from a different client (say, from a different machine), change the value of data associated with `/mysecondnode`.

Listing 11. Change the value of the data associated with `/mysecondnode`

```
[zk: localhost:2181(CONNECTED) 1] set /mysecondnode
hello2
cZxid = 0x200000007
ctime = Sat Jul 20 19:58:27 PDT 2013
mZxid = 0x200000009
mtime = Sat Jul 20 20:02:37 PDT 2013
pZxid = 0x200000007
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
```

You will notice that you get a watch notification on the first client.

Listing 12. Getting a watch notification on the first client

```
[zk: 127.0.0.1:2181(CONNECTED) 13]
WATCHER::
WatchedEvent state:SyncConnected type:NodeDataChanged path:/mysecondnode
```

Moving on, since znodes form a hierarchical namespace, you can also create subnodes.

Listing 13. Creating subnodes

```
[zk: localhost:2181(CONNECTED) 2] create /mysecondnode/
subnode 123
Created /mysecondnode/
subnode
```

You can get additional stat metadata about a znode.

Listing 14. Getting additional stat metadata about a znode

```
[zk: 127.0.0.1:2181(CONNECTED) 14] stat /mysecondnode
cZxid = 0x200000007
ctime = Sat Jul 20 19:58:27 PDT 2013
mZxid = 0x200000009
mtime = Sat Jul 20 20:02:37 PDT 2013
pZxid = 0x20000000a
cversion = 1
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 1
```

In the above examples, we use the ZooKeeper CLI client to interact with the ZooKeeper server. ZooKeeper comes with Java™, C, Python, and other bindings. You can turn your Java, C, or Python applications into ZooKeeper clients by calling the client APIs via those bindings.

Applications of ZooKeeper

Due to its versatile use cases in distributed systems, ZooKeeper has a diverse set of practical applications. We will list some of those applications here. Most of these uses have been taken from the Apache ZooKeeper wiki. A more complete and up-to-date list is available there as well. See Related topics for links to these technologies:

- Apache Hadoop relies on ZooKeeper for automatic fail-over of Hadoop HDFS Namenode and for the high availability of YARN ResourceManager.
- Apache HBase, a distributed database built on Hadoop, uses ZooKeeper for master election, lease management of region servers, and other communication between region servers.
- Apache Accumulo, another sorted distributed key/value store is built on top of Apache ZooKeeper (and Apache Hadoop).
- Apache Solr uses ZooKeeper for leader election and centralized configuration.
- Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications. Mesos uses ZooKeeper for fault-tolerant replicated master.
- Neo4j is a distributed graph database that uses ZooKeeper for write master selection and read slave coordination.
- Cloudera Search integrates search functionality (via Apache Solr) with Apache Hadoop using ZooKeeper for centralized configuration management.

Conclusion

Implementing your own protocols for coordinating a distributed system can be a frustrating and time-consuming process. This is where ZooKeeper shines. It's a stable, simple, and high-performance coordination service that gives you the tools you need write correct distributed applications without worrying about race conditions, deadlocks, and inconsistency. Next time you find yourself writing a distributed application, leverage ZooKeeper for all your coordination needs.

Related topics

- Learn more about using ZooKeeper in IBM's InfoSphere BigInsights product.
- Follow along with a tutorial on Programming ZooKeeper from the ZooKeeper wiki.
- Figure 1 comes from the the Apache.org ZooKeeper wiki.
- Be sure to read "Build server-cluster-aware Java applications."
- Learn about the Producer-Consumer problem from Wikipedia.
- Learn about the Barriers from Wikipedia.
- Read Writing and compiling Java programs.
- Check out the *Big Data Glossary*, by Pete Warden, O'Reilly Media, ISBN: 1449314597, 2011.
- You can download the ZooKeeper release tarball.
- Get the Java 7 JDK.
- Apache Hadoop relies on ZooKeeper for automatic fail-over of Hadoop HDFS Namenode and for the high availability of YARN ResourceManager.
- Apache HBase a distributed database built on Hadoop, uses ZooKeeper for master election, lease management of region servers, and other communication between region servers.
- Apache Accumulo, another sorted distributed key/value store is built on top of Apache ZooKeeper (and Apache Hadoop).
- Apache Solr uses ZooKeeper for leader election and centralized configuration.
- Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications. Mesos uses ZooKeeper for fault-tolerant replicated master.
- Neo4j is a distributed graph database that uses ZooKeeper for write master selection and read slave coordination.
- Cloudera Search integrates search functionality (via Apache Solr) with Apache Hadoop uses ZooKeeper for centralized configuration management.
- Download InfoSphere BigInsights Quick Start Edition, available as a native software installation or as a VMware image.
- Find resources to help you get started with InfoSphere Streams, IBM's high-performance computing platform that enables user-developed applications to rapidly ingest, analyze, and correlate information as it arrives from thousands of real-time sources.
- Download InfoSphere Streams, available as a native software installation or as a VMware image.