SEE TEXT ONLY

Building Distributed Applications with Apache Zookeeper

By Steven Kreuzer

Distributed systems are very complex beasts. While each system is built to solve a unique problem, they all share a common requirement of having a way for all nodes to communicate with each other in a reliable, fault-tolerant, and scalable fashion. On the surface, the problem of how to create a system that allows for reliable distributed communication and coordination appears to be fairly trivial, and you will find no shortage of academic papers that describe these algorithms in great detail. After a while, it may become tempting to just roll your own solution, but ask anyone who has done this before and I am sure you will find no shortage of horror stories that will quickly sour you on that idea. Since it is safe to assume that your time will be better spent not having to debug subtle race conditions and deadlocks, one could make a strong argument that you will be better served by deploying an existing solution that has already seen widespread adoption by large, open-source projects, universities, and companies across all types of industries. One such piece of software is Zookeeper, which is an Apache project focused on building a robust system to implement distributed system primitives that developers can immediately put to use when writing a distributed application. Originally created at Yahoo!, it is now an actively developed and vital component of the Hadoop ecosystem. While many Hadoop-related projects make extensive use of Zookeeper, aside from Java, it has no outside dependencies, which makes it incredibly simple to introduce it into your environment.

The Zookeeper Data Model

At its core, Zookeeper provides just a few basic operations that can be used to form many of the design patterns that are necessary in a distributed system. Zookeeper allows applications to communicate and coordinate with each other through hierarchical, key-value stores referred to as zNodes. The interface is done in such a way that it closely resembles a typical UNIX file system, with each zNode acting as either a file that is able to store up to a megabyte of data or as a directory that can contain multiple child zNodes. In addi-

tion, each zNode has some additional metadata such as the ACLs, creation time, modification time, and a version number associated with it.

Zookeeper allows for the creation of two different types of zNodes: persistent and ephemeral. When a new zNode is created, it will be persistent by default. Just as the name implies, the zNode will remain available until it is explicitly removed through the delete function. An ephemeral node will only persist while the client who created the zNode is connected. If the client's session is disconnected due to either a crash or explicit termination, Zookeeper will remove the node. Ephemeral nodes are very useful for providing host and service discovery and can also be used as a simple way to detect faults in a distributed system. While ephemeral zNodes can be created as a child of persistent zNodes, an ephemeral zNode does not have the ability to have child zNodes. In Zookeeper, an ephemeral zNode will always be a leaf node.

Zookeeper offers two additional components that, when combined with zNodes, allow you to easily replicate very complex behavior in a straightforward fashion. Both persistent or ephemeral nodes can be part of an atomic sequence whose names are automatically assigned a monotonic number which is maintained by the parent zNode. Zookeeper guarantees that this 10-digit number will always be unique and greater than any other child zNode created under the parent zNode. These sequential zNodes can be used as building blocks to easily implement a distributed locking mechanism if your application has such a requirement. Zookeeper also features the concept of a watch, which allows a client to request that it be notified when a change is made to a specific zNode. Watches can be used as simple mechanisms to create asynchronous, event-based systems and easily implement leader election algorithms. A watch in Zookeeper is a one-time trigger. After a change occurs and the notification is sent, the client must once again register the watch to receive notifications of future updates.

Getting Up and Running

Zookeeper was added to the FreeBSD ports tree in 2012, which makes it incredibly simple to get up and running with a single instance for testing and development. Because Zookeeper was defined to work out of the box and requires very little configuration to get going, it should not be necessary for you to have to make any changes to the config file until you are ready to deploy into your production environment.

```
# pkg install zookeeper
# sysrc zookeeper_enable=YES
zookeeper_enable:  -> YES
# cp /usr/local/etc/zookeeper/zoo_sample.cfg /usr/local/etc/zookeeper/zoo.cfg
# service zookeeper start
Starting zookeeper.
```

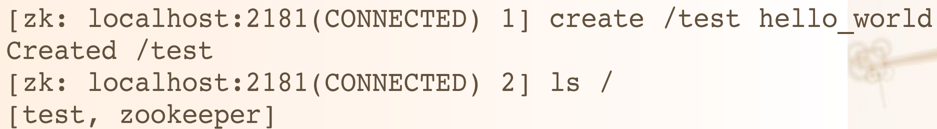
Once the server is up and running, you can connect to the server using the `zkCli.sh` command and try out a few commands to verify that everything is working as expected.

```
$ zkCli.sh
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is enabled

WATCHER::

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

Let's first start off by creating a new zNode called "test" and populate it with the data "hello_word." After that is done, we can list the contents of the root zNode and see that the new "test" zNode exists.



```
[zk: localhost:2181(CONNECTED) 1] create /test hello_world
Created /test
[zk: localhost:2181(CONNECTED) 2] ls /
[test, zookeeper]
```

The contents of the "test" zNode can be retrieved by issuing a get on the zNode. In this example the string "hello_world" along with some additional metadata about the zNode itself will be returned.

```
[zk: localhost:2181(CONNECTED) 3] get /test
hello_world
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x6
mtime = Tue Dec 29 15:39:07 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 11
numChildren = 0
```

The contents of the "test" zNode can also be updated using the "set" command. You will notice that after the command is run, metadata such as "mtime," "dataVersion" will also automatically be updated as well. Once the update has completed, another "get" can be issued to verify that the contents of the zNode have been modified.

```
[zk: localhost:2181(CONNECTED) 4] set /test freebsd_journal
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x7
mtime = Tue Dec 29 15:49:46 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
```

```
[zk: localhost:2181(CONNECTED) 5] get /test
freebsd_journal
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x7
mtime = Tue Dec 29 15:49:46 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
```

Once this zNode is no longer needed, a "delete" can be issued to completely remove the zNode from the data store.

```
[zk: localhost:2181(CONNECTED) 6] delete /test
[zk: localhost:2181(CONNECTED) 7] ls /
[zookeeper]
```

Moving into Production

While having a single instance of Zookeeper running in stand-alone mode is perfectly fine to use for testing or to do some quick prototyping, it does introduce a single point of failure that should be addressed before mission critical applications start to depend on it. Once you are ready to introduce Zookeeper into a production environment, you will want to deploy it configured to run in what's referred to as quorum mode. While Zookeeper is used for coordinating distributed applications, it, too, is a distributed application in which independent machines can form a cluster and elect a leader. The cluster, which is referred to as an ensemble, will replicate its data to all members, and as long as a majority of the nodes in the ensemble are online, all the services that Zookeeper provided will be available.

When researching the requirements for an ensemble, it is recommended that you start with at least three machines, but for most environments it is strongly encouraged to have at least five machines at a minimum. The reason for this is that in a three-node cluster, the loss of a single node is tolerable because two of the three remaining machines will still count as a majority. However, let's say you remove a node from the ensemble for regular scheduled maintenance and during this time you were to have another node unexpectedly fail. Now that the quorum is lost, the remaining nodes will switch to peer election mode and will disconnect existing clients and refuse new connections until a new leader has been elected. This particular failure scenario can be avoided by starting with a minimum of five nodes, which will allow the cluster to tolerate the loss of up to two nodes. While it is possible to configure Zookeeper to operate in read-only mode if the quorum is lost, determining if this feature is appropriate for your environment will mainly be driven by the requirements of your application, and as a result, the default behavior is to simply stop serving client connections.

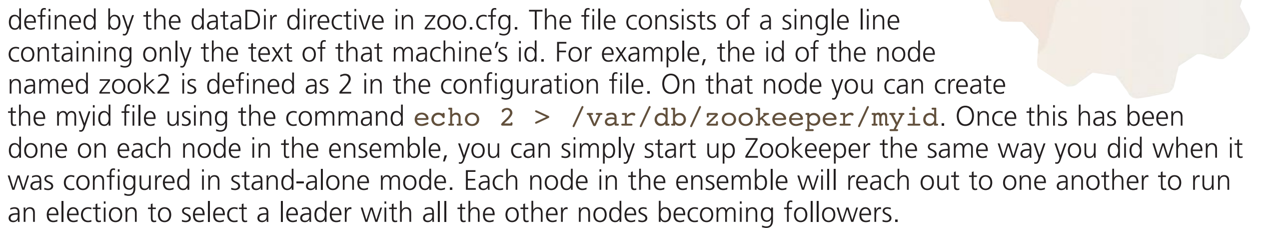
Since Zookeeper's main focus is to ensure that data is distributed in a reliable manner, another recommendation when deploying an ensemble is to always have an odd number of machines. This prevents the possibility of having a "split-brain" in which certain nodes become segmented from the other nodes but continue to operate independently. In this scenario, these machines can fall out of sync with the other half and once the failure has been resolved, the cluster won't know how to reconcile the differences. To combat this, Zookeeper uses a majority count, and a new node will be selected as the leader.

At the heart of Zookeeper is an atomic messaging system designed to keep each member of the ensemble in sync. The node that is elected as a leader will receive all writes and is responsible for publishing those changes to all the other members acting as a follower of the leader node. Zookeeper guarantees that data will eventually be consistent across all members of the ensemble by ensuring that data is always delivered in the same order it is sent. A message will only be delivered after all messages sent before it have been delivered, and while it's possible that two clients may not have the exact same point in time view, they will always observe the changes in the same order.

In quorum mode, all nodes have a copy of the same configuration file and will know about every other machine that is a member of the ensemble. This is accomplished by appending additional lines into the zoo.cfg file in the form of **server.id=host:quorum_port:election_port**.

```
# cat /usr/local/etc/zookeeper/zoo.cfg
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/db/zookeeper
clientPort=2181
server.1=zook1:2888:3888
server.2=zook2:2888:3888
server.3=zook3:2888:3888
```

Within the ensemble, each node must have a unique id between 1 and 255 assigned to it. The node is informed of its id by reading the contents of the "myid" file stored in the directory, which is



defined by the dataDir directive in zoo.cfg. The file consists of a single line containing only the text of that machine's id. For example, the id of the node named zook2 is defined as 2 in the configuration file. On that node you can create the myid file using the command `echo 2 > /var/db/zookeeper/myid`. Once this has been done on each node in the ensemble, you can simply start up Zookeeper the same way you did when it was configured in stand-alone mode. Each node in the ensemble will reach out to one another to run an election to select a leader with all the other nodes becoming followers.

One of the big advantages is that it's not much work at all to go from stand-alone mode on a single machine to the highly fault-tolerant quorum mode distributed across multiple machines in your environment. What's even better is that from a developer's standpoint, all the interfaces are still the same, and so no additional changes to the application will be necessary. One of the most appealing features of Zookeeper is that the system is designed to require little maintenance after the initial setup, and all the complexity is hidden away from the end user, which makes it easy to integrate.

Scaling Zookeeper

Normally within a distributed system, you can scale up to handle an increase in workload by adding additional capacity and spreading your application across more nodes. However, because each member of the ensemble needs to reach agreement on every transaction, as you add more voting members, the write performance of the ensemble will start to decrease. In addition to leader and follower roles, Zookeeper also allows members to join the ensemble and act as observers. When assigned this role, an observer will not take part in the agreement step of the atomic broadcast protocol. Instead, it will just accept transactions that have already been agreed upon by other followers in the quorum. The main goal of an observer is to provide read scalability without having to compromise on write performance. In addition, because observers do not vote or participate in leader elections, they can be more heavily loaded with read-only clients, which allows you to reduce the load placed on follower nodes that are required to participate.

Configuring a node to act as an observer is a fairly straightforward procedure. The node needs to be informed that it should act as an observer by adding the line `peerType=observer` into the configuration file for that particular node. In addition, all the other nodes are informed which servers are acting as observers by appending `:observer` at the end of the server config line.

```
server.1=zook1:2888:3888
server.2=zook2:2888:3888
server.3=zook3:2888:3888
server.4=zook4:2888:3888:observer
```

Once these configuration changes are in place you can restart the cluster normally and clients will be able to connect to an observer node in exactly the same way they would if it was a follower node.

Conclusion

When done incorrectly, a distributed system can quickly become an unmanageable mess that is hard to use and even more difficult to debug. As datasets become larger and workload demands become more intense we are quickly approaching a point in time when to get more work done, your only option will be to distribute the processing to as many machines on the network as possible. Because of this shift in technology, it comes as no surprise that Zookeeper has quickly become such a popular open-source project and has gained such widespread adoption. If you ever find yourself needing to introduce complex functionality into your application, Zookeeper will quickly become an important piece of infrastructure in your software stack. ●

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.