Zookeeper Tutorial: Designing a distributed system using zookeeper and Java

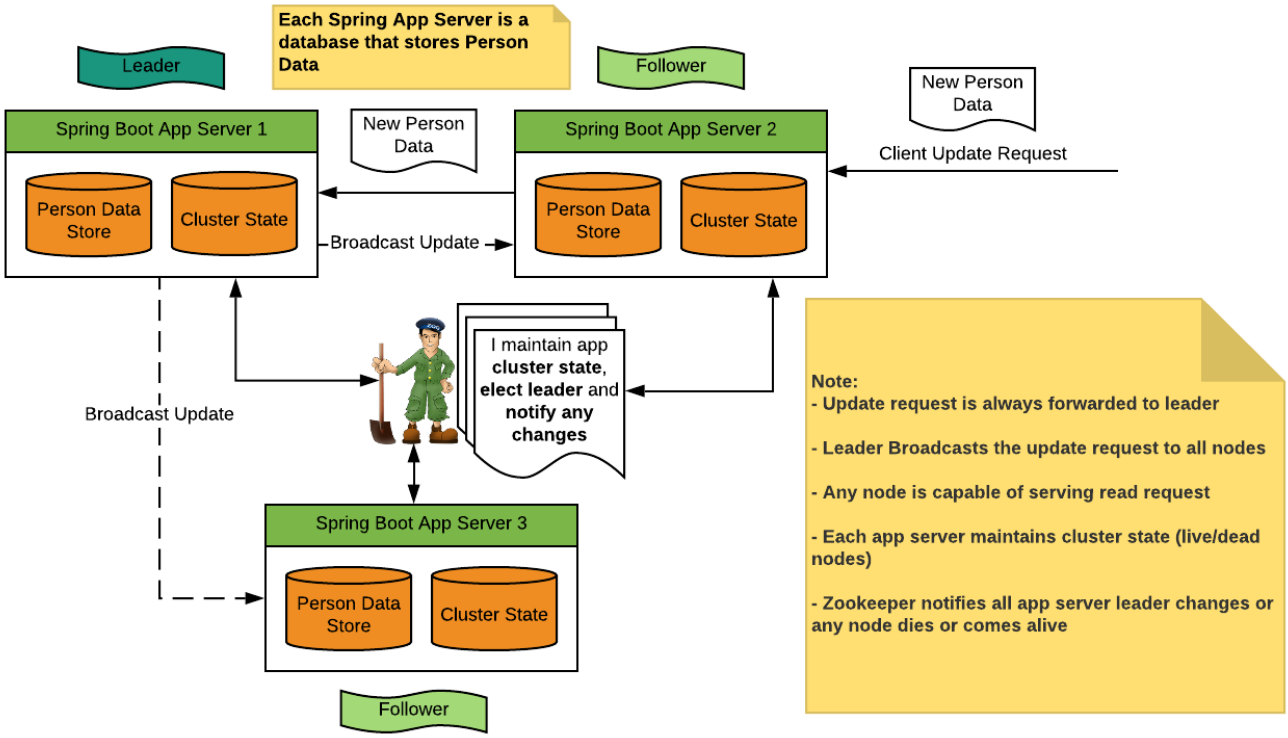
Bikas Katwal Follow
Mar 31, 2019 · 16 min read

Overview

I recently started learning Apache Zookeeper, and I have built a small distributed system that includes replicated spring boot app servers modeling a distributed database, that uses zookeeper as the backbone for maintaining cluster state and leader election.

With this blog, I intend to share my work/knowledge and give you a feel of Apache Zookeeper. Feel free to provide any feedback at comment section :)

With that let's look at the below high-level diagram of the system which we will be designing and implementing using Zookeeper:



A distributed database using spring boot as the database server

Key features we will build:

- Model a database that is *replicated across multiple servers*.

- The system should *scale horizontally*, meaning if any new server instance is added to the cluster, it should have the latest data and start serving update/read requests.
- *Data consistency*. All update requests will be forwarded to the leader, and then the leader will *broadcast data to all active servers and then returns the update status.*
- Data can be read from any of the replicas without any inconsistencies.
- All server in the cluster will store the cluster state — Information like, *who is the leader and server state(list of live/dead servers in the cluster*). This info is required by the leader server to broadcast update request to active servers, and active follower servers need to forward any update request to their leader.
- In the event of a change in the *cluster state(leader goes down/any server goes down)*, all servers in the cluster need to be notified and store the latest change in local cluster data storage.

We will use zookeeper as our coordination service to manage cluster state information and notify all servers in the cluster in case of any change in cluster state.

Before we start talking about the design and implementation of the above system, let's get familiar with Zookeeper. (Skip below section if you familiar with Zookeeper)

Zookeeper Overview

What is Apache Zookeeper (ZK)? It is a library that enables coordination in distributed systems. Below are some of the distributed systems coordination problem that zookeeper solves:

- **Configuration management —** Managing application configuration that can be shared across servers in a cluster. The idea is to maintain any configuration in a centralized place so that all servers will see any change in configuration files/data.
- **Leader election —** Electing a leader in a multi-node cluster. You might need a leader to maintain a single point for an update request or distributing tasks from leader to worker nodes.
- **Locks in distributed systems —** distributed locks enables different systems to operate on a shared resource in a mutually exclusive way. Think of an example where you want to write to a shared file or any shared data. Before updating the shared resource, each server will acquire a lock and release it after the update.
- **Manage cluster membership —** Maintain and detect if any server leaves or joins a cluster and store other complex information of a cluster.

> *Zookeeper solves these problems using its magical tree structure file system called* ***znodes,*** *somewhat similar to the Unix file system. These znodes are analogous to folders and files in a Unix file system with some additional magical abilities :) Zookeeper provides primitive operations to manipulate these znodes, through which we will solve our distributed system problems.*

Key Znode features you need to know:

- Znodes can store data and have children Znode at same time
- It can store information like the current version of data changes in Znode, transaction Id of the latest transaction performed on the Znode.
- Each znode can have its access control list(ACL), like the permissions in Unix file systems. Zookeeper supports: **create, read, write, delete, admin(set/edit permissions)** permissions.
- Znodes ACL supports username/password based authentication on individual znodes too.
- Clients can set a watch on these Znodes and get notified if any changes occur in these znodes.
  These change/events could be a change in znodes data, change in any of znodes children, new child Znode creation or if any child Znode is deleted under the znode on which watch is set.

Supported Zookeeper Operations:

Operations	Description
create	create a znode in a specified path
delete	delete a znode from a specified path
getData	get data associated with a znode
getChildren	get list of child znodes for a specified znode path
exists	tells if znode exists in specified path
setData	set the data of a znode
getACL	get znode's permissions
setACL	set permission to znode

Below are some of the operations in ZK command line interface:

- create() — creating a znode “/test_znode”

```
[zk: localhost:2181(CONNECTED) 0] create /test_znode "this is data"
Created /test_znode
[zk: localhost:2181(CONNECTED) 1]
```

creating znode

- getData() or get — get on "/test_znode"

```
[zk: localhost:2181(CONNECTED) 1] get /test_znode
this is data
cZxid = 0x5a9
ctime = Thu Mar 28 12:17:16 IST 2019
mZxid = 0x5a9
mtime = Thu Mar 28 12:17:16 IST 2019
pZxid = 0x5a9
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 12
numChildren = 0
```

get operation on znodes

- Creating child znodes under "/test_znode" and displaying all children using "ls" operation a.k.a getChildren()

```
[zk: localhost:2181(CONNECTED) 0]
[zk: localhost:2181(CONNECTED) 0] create /test_znode/child_1 "this is child one"
Created /test_znode/child_1
[zk: localhost:2181(CONNECTED) 1] create /test_znode/child_2 "this is child two"
Created /test_znode/child_2
[zk: localhost:2181(CONNECTED) 2] ls /test_znode
[child_2, child_1]
[zk: localhost:2181(CONNECTED) 3]
```

- Deleting a znode

```
[zk: localhost:2181(CONNECTED) 3] delete /test_znode/child_1
[zk: localhost:2181(CONNECTED) 4] ls /test_znode
[child_2]
[zk: localhost:2181(CONNECTED) 5]
```

- getAcl() — get permissions of "/test_znode".

```
[zk: localhost:2181(CONNECTED) 5] getAcl /test_znode
'world,'anyone
```

```
: cdrwa
[zk: localhost:2181(CONNECTED) 6]
```

Znode Types and their Use Cases

1. **Persistent Znode:** As the name says, once created these Znodes will be there forever in the Zookeeper. To remove these Znodes, you need to delete them manually(use delete operation).
   As we learn this type of Znode never dies/deleted automatically, we can store any config information or any data that needs to be persistent. All servers can consume data from this Znode.
   Note: If no flag is passed, by default persistent znode is created.
   Example: Solr Cloud, uses these znodes to store server configuration and schema of database/collections.

2. **Ephemeral ZNodes:** These znodes are automatically deleted by the Zookeeper, once the client that created it, ends the session with zookeeper.
   Zookeeper clients keep sending the ping request to keep the session alive. If Zookeeper does not see any ping request from the client for a period of configured session timeout, Zookeeper considers the client as dead and deletes the client session and the Znode created by the client.
   You might have already guessed the use case of these znodes. Let's say you want to maintain a list of active servers in a cluster. So, you create a parent Znode "/live_servers". Under it, you keep creating child Znode for every new server in the cluster. At any point, if a server crashes/dies, child Znode belonging to the respective server will be deleted. Other servers will get a notification of this deletion if they are watching the znode "/live_servers".
   It is created using -e flag

3. **Ephemeral Sequential Znode:** It is same as ephemeral Znode, the only difference is Zookeeper attaches a sequential number as a suffix, and if any new sibling Znode of the same type is created, it will be assigned a number higher than previous one.
   This type of znode is created using -e -s flag.
   Let's say, we want to create two ephemeral sequential Znodes "child_nodeA" and "child_nodeB" inside "test_znode" parent Znode. It will attach sequence number "0000000000" and "0000000001" as the suffix.

```
[zk: localhost:2181(CONNECTED) 9] create -e -s /test_znode/child_nodeA "this first is epemeral seq node data"
Created /test_znode/child_nodeA0000000000
[zk: localhost:2181(CONNECTED) 10] create -e -s /test_znode/child_nodeB "this second is epemeral seq node data"
Created /test_znode/child_nodeB0000000001
[zk: localhost:2181(CONNECTED) 11] ls /test_znode
[child_nodeB0000000001, child_nodeA0000000000]
[zk: localhost:2181(CONNECTED) 12]
```

This type of znode could be used in the leader election algorithm.
Say I have a parent node "/election", and for any new node that joins the cluster, I add an ephemeral sequential Znode to this "/election" node. We can consider a server as the leader if any server that created the znode has the least sequential number attached to it. So, even if a leader goes down, zookeeper will delete corresponding Znode created by the leader server and notify the client applications, then that client fetches the new lowermost sequence node and considers that as a new leader. We will talk in detail about leader election in the later section.

4. **Persistent Sequential Znode:** This is a persistent node with a sequence number attached to its name as a suffix. We will rarely be using this one. I did not find any use case. If you guys can think of any use case, please leave in the comment section :)

Some of the common Zookeeper Recipes:

Leader Election

We will discuss three algorithms for leader election.

Approach 1:

1. A client(any server belonging to the cluster) creates a persistent znode ***/election*** in Zookeeper.
2. All clients add a watch to **/election** znode and listen to any children znode deletion or addition under **/election** znode.
3. Now each server joining the cluster will try to create an ephemeral znode ***/leader*** under node ***/election*** with data as hostname, ex: ***node1.domain.com***
   Since multiple servers in the cluster will try to create znode with the same name(***/leader***), only one will succeed, and that server will be considered as a leader.
4. Once all server in the cluster completes above step, they will call getChildren("/election") and get the data(hostname) associated with child znode "/leader", which will give the leader's hostname.
5. At any point, if the leader server goes down, Zookeeper will kill the session for that server after the specified session timeout. In the process, it will delete the node **/leader** as it was created by leader server and is an ephemeral node and then Zookeeper will notify all the servers that have set the watch on ***/election*** znode, as one of the children has been deleted.

6. Once all server gets notified that the leader is dead or leader's znode(/leader) is deleted, they will retry creating "/leader" znode and again only one server will succeed, making it a new leader.

7. Once the /leader node is created with the hostname as data part of the znode, zookeeper will again notify all server(as we have set the watch in step 2).

8. All servers will call getChildren() on "/election" and update the new leader in their memory.

The problem with the above approach is, each time ***/leader*** node is deleted, Zookeeper will send the notification to all servers and all servers will try to write to zookeeper to become new leader at the same time creating a ***herd effect.*** If we have a large number of servers, this approach would not be the right idea.
Ways to avoid, **herd effect** could be:
(i) by restricting the number of servers that take part in the election and allow only a few servers to update **/election** znode
OR
(ii) by using sequential znode, which I will explain in the next approach.

Approach 2: Using Ephemeral Sequential Znode

1. A client(any server belonging to the cluster) creates a persistent znode ***/election***.

2. All clients add a watch to **/election** znode and listen to any children znode deletion or addition under **/election** znode.

3. Now each server joining the cluster will try to create an ephemeral sequential znode ***/leader-<sequential number>*** under node ***/election*** with data as hostname, ex: ***node1.domain.com***
   Let's say three servers in a cluster created znodes under /election, then the znode names would be:
   /election/leader-00000001
   /election/leader-00000002
   /election/leader-00000003
   Znode with least sequence number will be automatically considered as the leader.

4. Once all server completes the creation of znode under **/election,** they will perform getChildren("/election") and get the data(hostname) associated with least sequenced child node "***/election/leader-00000001***", which will give the leader hostname.

5. At any point, if the current leader server goes down, Zookeeper will kill the session for that server after the specified session timeout. In the process, it will delete the node "***/election/leader-00000001***" as it was created by leader server and is an ephemeral node and then Zookeeper will send a notification to all the server that was watching znode **/election**.

6. Once all server gets the leader's znode-delete notification, they again fetch all children under **/election** znode and get the data associated with the child znode that has the least sequence number(***/election/leader-00000002***) and store that as the new leader in its own memory.

In this approach, we saw, if an existing leader dies, the servers are not sending an extra write request to zookeeper to become the leader, leading to reduce network traffic.
But, even with this approach, we will face some degree of ***herd effect*** we talked about in the previous approach. When the leader server dies, notification is sent to all servers in the cluster, creating a herd effect.
But, this is a design call that you need to take. Use approach 1 or 2, if you need all servers in your cluster to store the current leader's hostname for its purpose.
If you do not want to store current leader information in each server/follower and only the leader needs to know if he is the current leader to do leader specific tasks. You can further simplify the leader election process, which we will discuss in approach 3.

Approach 3: Using Ephemeral Sequential Znode but notify only one server in the event of a leader going down.

1. Create a persistent znode ***/election***.

2. Now each server joining the cluster will try to create an ephemeral sequential znode ***/leader-<sequential number>*** under node ***/election*** with data as hostname, ex: ***node1.domain.com***
   Let's say three servers in a cluster created znodes under /election, then the znode names would be:
   /election/leader-00000001
   /election/leader-00000002
   /election/leader-00000003
   Znode with least sequence number will be automatically considered as a leader.

3. Here we will not set the watch on whole/**election** znode for any children change(add/delete child znode), instead, each server in the cluster will set watch on child znode with one less sequence.
   The idea is if a leader goes down only the next candidate who would become a leader should get the notification.

So, in our example:
- The server that created the znode ***/election/leader-00000001*** will have no watch set.
-The server that created the znode ***/election/leader-00000002*** will watch for deletion of znode ***/election/leader-00000001***
-The server that created the znode ***/election/leader-00000003*** will watch for deletion of znode ***/election/leader-00000002***

4. Then, if the current leader goes down, zookeeper will delete the node ***/election/leader-00000001*** and send the notification to only the next leader i.e. the server that created node ***/election/leader-00000002***

That's all on leader election logic. These are simple algorithms. There could be a situation when you want only those server to take part in leader election which has the latest data if you are creating a distributed database.
In that case, you might want to create one more node that keeps this information, and in the event of the leader going down, only those servers that have latest data can take part in an election.

Distributed Locks

Suppose we have "n" servers trying to update a shared resource simultaneously, say a shared file. If we do not write these files in a mutually exclusive way, it may lead to data inconsistencies in the shared file.

We will manipulate operations on znode to implement a distributed lock, so that, different servers can acquire this lock and perform a task.

Algorithm for managing distributed locks is the same as leader election with a slight change.

1. Instead of the /election parent node, we will use **/lock** as the parent node.
2. Rest of the steps will remain the same as in leader election algorithm. Any server which is considered a leader is analogous to server acquiring the lock.
3. The only difference is, once the server acquires the lock, the server will perform its task and then call the delete operation on the child znode it has created, so that next server can acquire lock upon delete notification from zookeeper and perform the task.

Group Membership/Managing Cluster state

In Zookeeper it is pretty simple to maintain group membership info using persistent and ephemeral znodes. I will talk about a simple case where you want to maintain information about all servers in a cluster and what servers are currently alive.
We will use a persistent znode to keep track of all the servers that join the cluster and zookeeper's ability to delete an ephemeral znodes upon client session termination will come handy in maintaining the list of active/live servers.

1. Create a parent znode ***/all_nodes***, this znode will be used to store any server that connects to the cluster.

2. Create a parent znode ***/live_nodes***, this znode will be used to store only the live nodes in the cluster and will store ephemeral child znodes. If any server crashes or goes down, respective child ephemeral znode will be deleted.

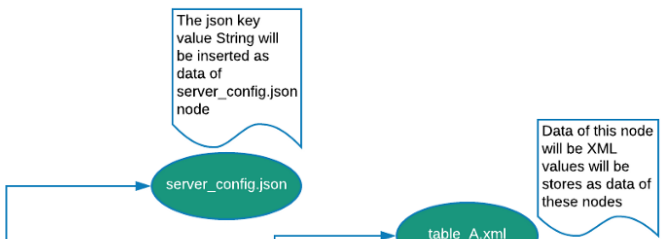
3. Any server connecting to the cluster will create a new **persistent znode** under ***/all_nodes*** say ***/node1.domain.com***. Let's say another two node joins the cluster. Then the znode structure will look like:
   /all_nodes/node1.domain.com
   /all_nodes/node2.domain.com
   /all_nodes/node3.domain.com
   You can store any information specific to the node in znode's data

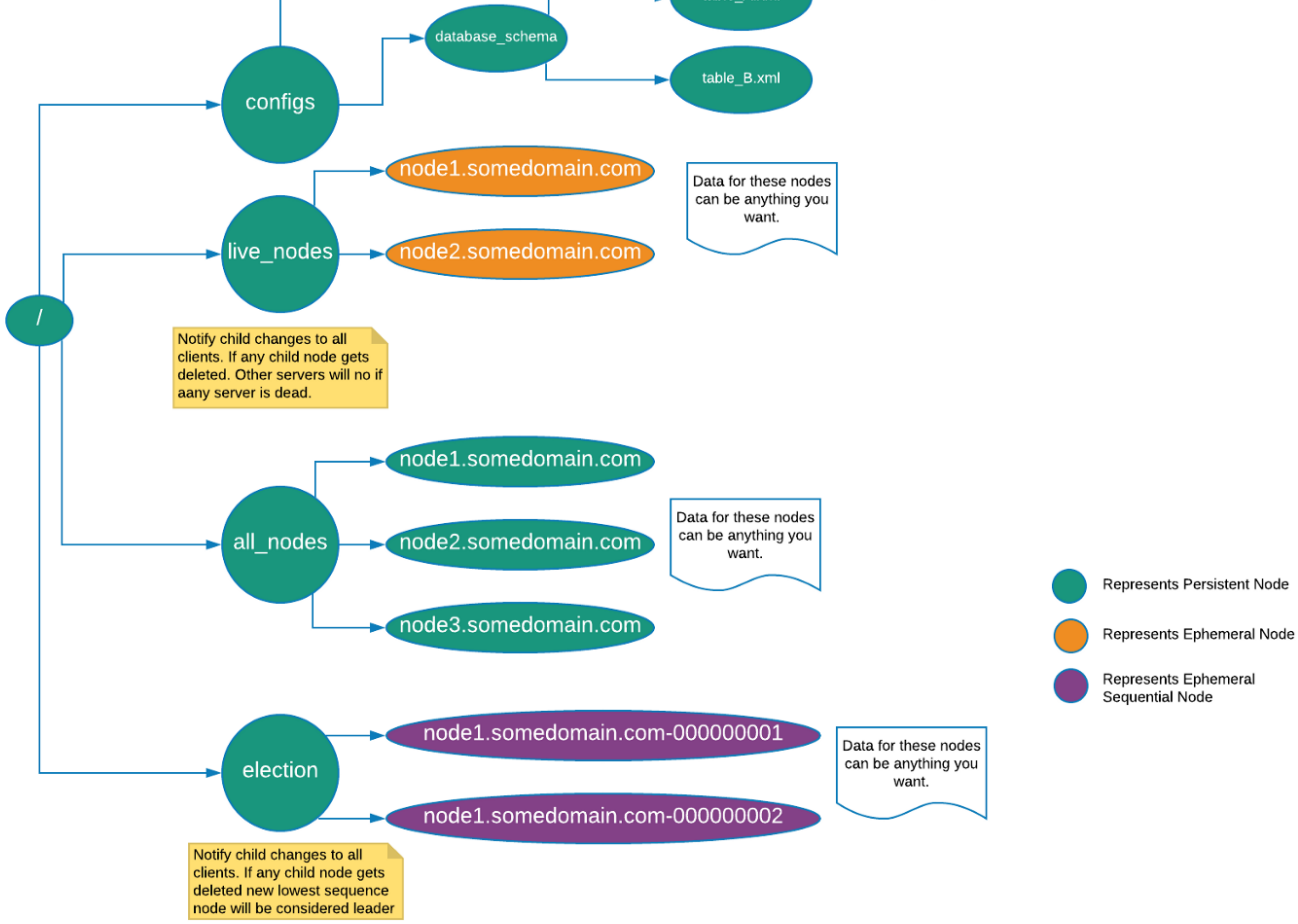
4. Any server connecting to the cluster will create a new **ephemeral znode** under ***/live_nodes*** say ***/node1.domain.com***. Let's say another two node joins the cluster. Then the znode structure will look like:
   /live_nodes/node1.domain.com
   /live_nodes/node2.domain.com
   /live_nodes/node3.domain.com

5. Add a watch for any change in children of ***/all_nodes.*** If any server is added or deleted to/from the cluster, all server in the cluster needs to be notified.

6. Add a watch for any change in children of ***/live_nodes.*** This way all servers will be notified if any server in the cluster goes down or comes alive.

With that let's look at, how a zookeeper Znode structure looks like for a typical distributes application:



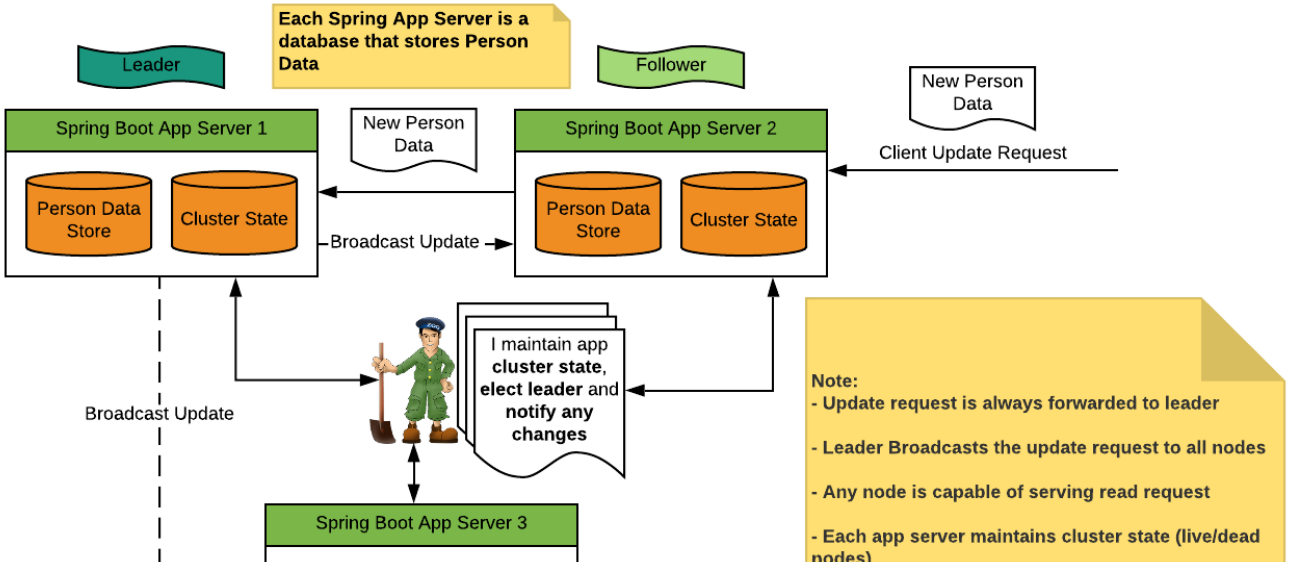


Designing and Implementing a Sample Distributed Database

Overview

For our demo purpose, we will be using a standalone zookeeper server.
Ideally, for a production environment, you might want to run multiple zookeeper servers — 3 or 5 or more. But, it has to be an odd number of servers. Why an odd number?
Read: Why an odd number of servers in the ZK Cluster.

Let's go back to our first diagram.



A distributed database using spring boot as the database server

In the above diagram:

- Three Spring boot App server running on port 8081, 8082 and 8083 is used as a database that **stores Person data** (*List<Person>*).
- Each spring boot server **connects to a standalone zookeeper server during startup** passed as VM argument (*-Dzk.url=localhost:2181*).
- Each spring boot app server will **maintain and store the cluster info** in its memory. This cluster info will tell current active servers, the current leader of the cluster and all nodes that are part of this cluster.
- We will create 2 GET APIs, to get info about the cluster and person data and 1 PUT API to save Person data.
- Any Person update request coming to App server will be sent to Leader and which will broadcast the update request to all live servers/followers.
- Any server coming up after being dead will sync Person data from the leader.

Setup

- Install and start Apache Zookeeper in any port. Follow guide: https://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html
- We have used below dependency in our project, this artifact has all the zookeeper primitive APIs that we need and simpler implementation of zookeeper watchers.

```
<dependency>
  <groupId>com.101tec</groupId>
  <artifactId>zkclient</artifactId>
  <version>0.11</version>
</dependency>
```

Implementation

In the implementation, we mainly will focus on:

1. The zookeeper operations and algorithms that we need to implement to solve the leader election problem and to maintain active/inactive servers list.
2. The listeners/watchers implementation that is needed for an app server to get notified in the event of leader change or any server going down.
3. Tasks that our spring boot app server(database) needs to perform during startup like creating necessary nodes, registering watchers, etc.
4. API to update the person data.

Zookeeper operations

We need the following zookeeper operations:

- To create all the required znodes(/election, /all_nodes, /live_nodes) before we start our application server.
- To create ephemeral znodes inside /live_nodes.
- To create ephemeral sequential znodes inside /election.
- get children operations on /live_nodes, /all_nodes, and /election znodes.
  Example get operation for the leader:

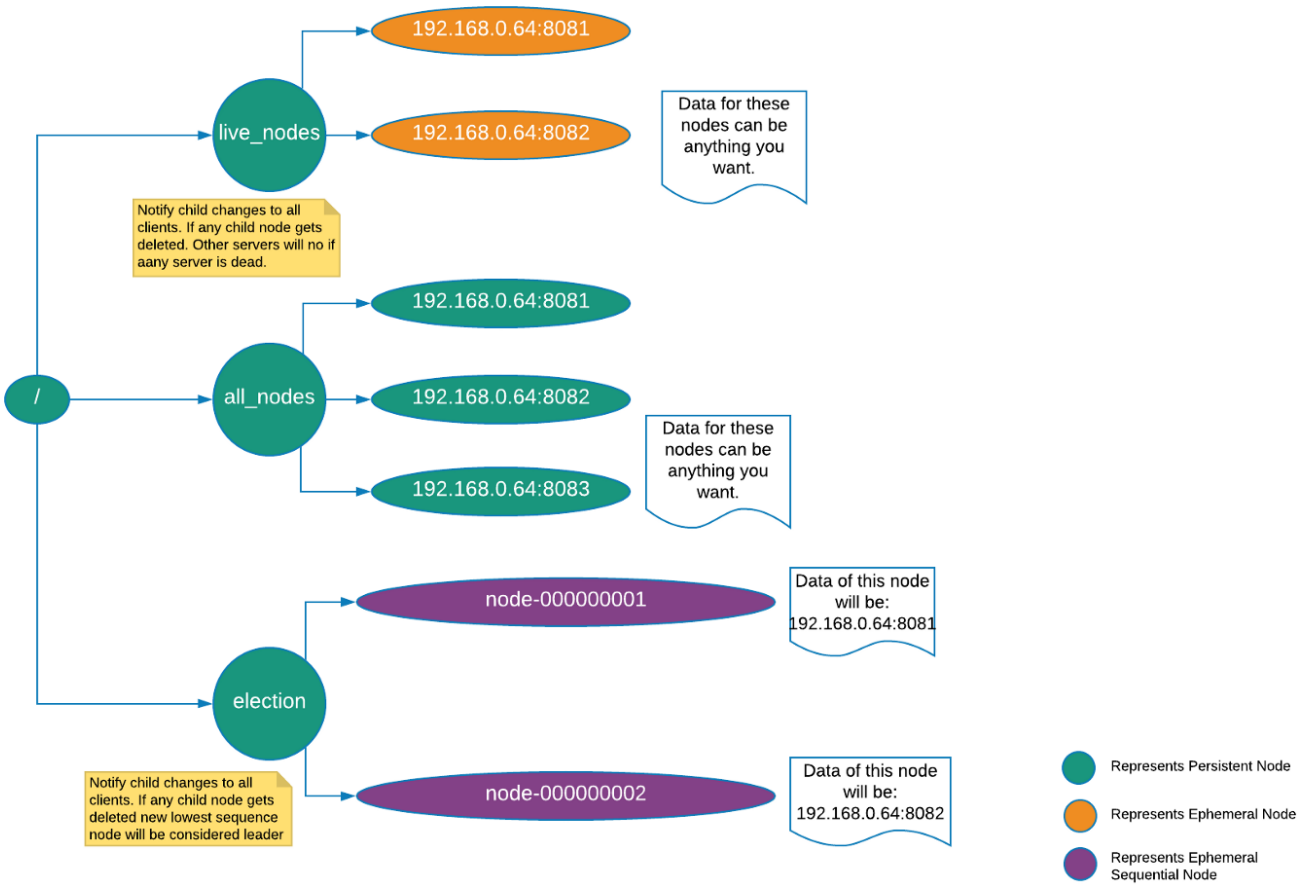
```
1   public String getLeaderNodeData2() {
2       if (!zkClient.exists("/election")) {
3         throw new RuntimeException("No node /election2 exists");
4       }
5
6       // fetch all children under /election
7       List<String> nodesInElection = zkClient.getChildren("/election");
8
9       //get the least sequenced znode, say "node-00000001", this znode will be considered leader
10      Collections.sort(nodesInElection);
11      String masterZNode = nodesInElection.get(0);
12
13      //get the data associated with znode "/election", which will give "host:port" of leader
14      return getZNodeData("/election".concat("/").concat(masterZNode));
15    }
```

FetchNewLeaderApproach2.java hosted with ♥ by GitHub view raw

- APIs to register our watchers, to capture cluster state change and leader change.

> *Check ZkServiceImpl.java for implementation — follow comments*

Znode structure for this application:



Listeners/Watchers

We need four watchers in our application:

1. Watcher for any change in children of /all_nodes, to identify and server addition/deletion to/from the cluster and update local ClusterInfo object.

```
@Slf4j
public class AllNodesChangeListener implements IZkChildListener {

  /**
   * - This method will be invoked for any change in /all_nodes children
   * - During registering this
   * listener make sure you register with path /all_nodes
   * - after receiving notification it will update the local clusterInfo object
   *
   * @param parentPath this will be passed as /all_nodes
   * @param currentChildren current list of children, children's string value is znode name whic
   *     set as server hostname
   */
  @Override
  public void handleChildChange(String parentPath, List<String> currentChildren) {
    log.info("current all node size: {}", currentChildren.size());
    ClusterInfo.getClusterInfo().getAllNodes().clear();
```

```
    ClusterInfo.getClusterInfo().getAllNodes().addAll(currentChildren);
  }
}
```

AllNodesChangeListener.java hosted with ♥ by GitHub view raw

2. Watcher for change in children in /live_nodes, to capture if any server goes down and then update the local ClusterInfo object

```
@Slf4j
public class LiveNodeChangeListener implements IZkChildListener {

  /**
   * - This method will be invoked for any change in /live_nodes children
   * - During registering this listener make sure you register with path /live_nodes
   * - after receiving notification it will update the local clusterInfo object
   *
   * @param parentPath this will be passed as /live_nodes
   * @param currentChildren new list of children that are present in /live_nodes, children's str
   */
  @Override
  public void handleChildChange(String parentPath, List<String> currentChildren) {
    log.info("current live size: {}", currentChildren.size());
    ClusterInfo.getClusterInfo().getLiveNodes().clear();
    ClusterInfo.getClusterInfo().getLiveNodes().addAll(currentChildren);
  }
}
```

LiveNodeChangeListener.java hosted with ♥ by GitHub view raw

3. Watchers to capture the change in leader, listening to the change in children of znode /election. Then fetch the least sequenced znode from the list of children and make it new leader server.

```
@Slf4j
@Setter
public class MasterChangeListenerApproach2 implements IZkChildListener {

  private ZkService zkService;

  /**
   * listens for deletion of sequential znode under /election znode and updates the
   * clusterinfo
   *
   * @param parentPath
   * @param currentChildren
   */
```

```
  @Override
  public void handleChildChange(String parentPath, List<String> currentChildren) {
    if (currentChildren.isEmpty()) {
      throw new RuntimeException("No node exists to select master!!");
    } else {
      //get least sequenced znode
      Collections.sort(currentChildren);
      String masterZNode = currentChildren.get(0);

      // once znode is fetched, fetch the znode data to get the hostname of new leader
      String masterNode = zkService.getZNodeData("/election".concat("/").concat(masterZNode));
      log.info("new master is: {}", masterNode);

      //update the cluster info with new leader
      ClusterInfo.getClusterInfo().setMaster(masterNode);
    }
  }
}
```

MasterChangeListenerApproach2.java hosted with ♥ by GitHub view raw

4. Watcher for every new session establishment with zookeeper. Application session with zookeeper might end if zookeeper doesn't receive any ping within the configured session timeout, this could happen due to temporary network failure or GC pause or any other reason.
Once the session of a server is killed by the zookeeper, Zookeeper will delete all ephemeral znodes created by this server, leading to deletion of znode under /live_nodes. So, if the session is established at any later point, we need to re-sync data from the current master and create znode in /live_nodes to notify all other servers that an existing server has become active.

```
@Slf4j
@Setter
public class ConnectStateChangeListener implements IZkStateListener {

  private ZkService zkService;

  @Override
  public void handleNewSession() throws Exception {
    log.info("connected to zookeeper");

    // sync data from master
    syncDataFromMaster();

    // add new znode to /live_nodes to make it live
    zkService.addToLiveNodes(getHostPostOfServer(), "cluster node");
    ClusterInfo.getClusterInfo().getLiveNodes().clear();
    ClusterInfo.getClusterInfo().getLiveNodes().addAll(zkService.getLiveNodes());
```

```
18
19      // re try creating znode under /election
20      // this is needed, if there is only one server in cluster
21      String leaderElectionAlgo = System.getProperty("leader.algo");
22      if (isEmpty(leaderElectionAlgo) || "2".equals(leaderElectionAlgo)) {
23        zkService.createNodeInElectionZnode(getHostPostOfServer());
24        ClusterInfo.getClusterInfo().setMaster(zkService.getLeaderNodeData2());
25      } else {
26        if (!zkService.masterExists()) {
27          zkService.electForMaster();
28        } else {
29          ClusterInfo.getClusterInfo().setMaster(zkService.getLeaderNodeData());
30        }
31      }
32    }
33  }
```

ConnectStateChangeListener.java hosted with ♥ by GitHub view raw

Application Startup Tasks

OnStartupApplication.java runs during application startup and performs below tasks:

1. Create all parent znodes /election, /live_nodes, /all_nodes, if they do not exist.
2. Add the server to cluster by creating znode under /all_nodes, with znode name as *host:port* string and update the local ClusterInfo object.
3. Set ephemeral sequential znode in the /election, to set up a leader for the cluster, with suffix "*node-*" and data as "*host:port*".
4. Get the current leader from zookeeper and set it to ClusterInfo object.
5. sync all Person data from leader server.
6. Once sync completes, announce this server as active by adding a child znode under /live_nodes with "*host:port*" string as the znode name and then update the ***ClusterInfo*** object.
7. In final step register all listeners/watchers to get notification from zookeeper.

OnStartupApplication.java:

```
1   public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
2       try {
3
4         // create all parent nodes /election, /all_nodes, /live_nodes
5         zkService.createAllParentNodes();
6
```

```
7         // add this server to cluster by creating znode under /all_nodes, with name as "host:port"
8         zkService.addToAllNodes(getHostPostOfServer(), "cluster node");
9         ClusterInfo.getClusterInfo().getAllNodes().clear();
10        ClusterInfo.getClusterInfo().getAllNodes().addAll(zkService.getAllNodes());
11
12        // check which leader election algorithm(1 or 2) need is used
13        String leaderElectionAlgo = System.getProperty("leader.algo");
14
15        // if approach 2 - create ephemeral sequential znode in /election
16        // then get children of  /election and fetch least sequenced znode, among children znodes
17        if (isEmpty(leaderElectionAlgo) || "2".equals(leaderElectionAlgo)) {
18          zkService.createNodeInElectionZnode(getHostPostOfServer());
19          ClusterInfo.getClusterInfo().setMaster(zkService.getLeaderNodeData2());
20        } else {
21          if (!zkService.masterExists()) {
22            zkService.electForMaster();
23          } else {
24            ClusterInfo.getClusterInfo().setMaster(zkService.getLeaderNodeData());
25          }
26        }
27
28        // sync person data from master
29        syncDataFromMaster();
30
31        // add child znode under /live_node, to tell other servers that this server is ready to se
32        // read request
33        zkService.addToLiveNodes(getHostPostOfServer(), "cluster node");
34        ClusterInfo.getClusterInfo().getLiveNodes().clear();
35        ClusterInfo.getClusterInfo().getLiveNodes().addAll(zkService.getLiveNodes());
36
37        // register watchers for leader change, live nodes change, all nodes change and zk session
38        // state change
39        if (isEmpty(leaderElectionAlgo) || "2".equals(leaderElectionAlgo)) {
40          zkService.registerChildrenChangeWatcher(ELECTION_NODE_2, masterChangeListener);
41        } else {
42          zkService.registerChildrenChangeWatcher(ELECTION_NODE, masterChangeListener);
43        }
44        zkService.registerChildrenChangeWatcher(LIVE_NODES, liveNodeChangeListener);
45        zkService.registerChildrenChangeWatcher(ALL_NODES, allNodesChangeListener);
46        zkService.registerZkSessionStateListener(connectStateChangeListener);
47      } catch (Exception e) {
48        throw new RuntimeException("Startup failed!!", e);
49      }
50    }
```

OnStartup.java hosted with ♥ by GitHub view raw

Update request API for Person Data

In our system, all write requests are processed by the leader, and then the leader will broadcast “Person” data to all active servers.

We need an API for update request that needs to perform below task:
(i) If the update request is from the leader, then save it to the local database(List<Person>).
(ii) If the request is coming from the client and the server receiving request is one of the followers, then it should forward update request to the leader.
(iii) Once the leader receives the update request, it will broadcast the update to all followers and save in its local storage also.

PUT /person/{id}/{name} API

```
@PutMapping("/person/{id}/{name}")
public ResponseEntity<String> savePerson(
    HttpServletRequest request,
    @PathVariable("id") Integer id,
    @PathVariable("name") String name) {

  String requestFrom = request.getHeader("request_from");
  String leader = ClusterInfo.getClusterInfo().getMaster();

 //if request is from leader, save to local storage and return
 if (!isEmpty(requestFrom) && requestFrom.equalsIgnoreCase(leader)){
    Person person = new Person(id, name);
    DataStorage.setPerson(person);
    return ResponseEntity.ok("SUCCESS");
 }
 /* If request is from client and this server is leader, just save data locally and broadcast da
    else forward update reques to leader server */
  if (amILeader()) {
    List<String> liveNodes = ClusterInfo.getClusterInfo().getLiveNodes();

    int successCount = 0;
    for (String node : liveNodes) {

      if (getHostPostOfServer().equals(node)) {
        Person person = new Person(id, name);
        DataStorage.setPerson(person);
        successCount++;
      } else {
        String requestUrl =
            "http://"
                .concat(node)
                .concat("person")
                .concat("/")
                .concat(String.valueOf(id))
                .concat("/")
```

```
                .concat(name);
        HttpHeaders headers = new HttpHeaders();
        headers.add("request_from", leader);
        headers.setContentType(MediaType.APPLICATION_JSON);

        HttpEntity<String> entity = new HttpEntity<>(headers);
        restTemplate.exchange(requestUrl, HttpMethod.PUT, entity, String.class).getBody();
        successCount++;
      }
    }

    return ResponseEntity.ok()
        .body("Successfully update ".concat(String.valueOf(successCount)).concat(" nodes"));
  } else {
    String requestUrl =
        "http://"
            .concat(leader)
            .concat("person")
            .concat("/")
            .concat(String.valueOf(id))
            .concat("/")
            .concat(name);
    HttpHeaders headers = new HttpHeaders();

    headers.setContentType(MediaType.APPLICATION_JSON);

    HttpEntity<String> entity = new HttpEntity<>(headers);
    return restTemplate.exchange(requestUrl, HttpMethod.PUT, entity, String.class);
  }
}
```

PersonUpdateAPI.java hosted with ♥ by GitHub view raw

> Note: Actual code in the repo uses both the approach 1 and 2 for leader election discussed in this article, based on VM argument passed at runtime. So, use below command to start application with approach 2:
>
> `java -Dserver.port=8081 -Dzk.url=localhost:2181 -Dleader.algo=2 -jar target/bkatwal-zookeeper-demo-1.0-SNAPSHOT.jar`

Installation and Usage

Follow the ReadMe instruction in the repository

Conclusion

In this article, I talked about Apache Zookeeper and its recipes and we built a sample replicated system using spring boot, depicting how a distributed system can be built

using Apache Zookeeper.
You can find the complete code and API details in the GitHub repo Zookeeper-demo.

Zookeeper Leader Election Distributed Systems Zookeeper Java Implementation