Distributed Key-value store using Atomix, copycat and Catalyst

Muthukumar Suresh

December 2015

1 License

This project is released under Apache License. Many parts of the code in this project is taken from the Copycat and Atomix websites like for setting up servers, clients and statemachine. I have added code on top of it for synchronization, application usability and to achieve the project aims.

Stackoverflow code snippets proved valuable for setting up the application CLI interface and maven issues.

2 Introduction

In this project, we aim to build a distributed key-value store that is distributed in its true sense. Therefore, the distributed store needs to satisfy the 3 properties:

1. Availability
2. Consistency
3. Partitioning of data

Consistency arises out of the problem of Availability. We must ensure even in case of failure of nodes, the system continues to have a consistent state machine.

Copycat[1][2], which is is our distributed consensus framework and uses the Raft protocol. Raft maintains consistency amongst the nodes by using a log file to keep track of all committed transactions. A transaction is committed only if it has been added to the log on all nodes. Raft also performs well in the case of network partitions. It may so happen at times that due to network partition two leaders might be active at a time. But due to the requirement of a majority to accept a log entry, only 1 log will be in a valid committed state. This ensures that a client that gets a commit from the Copycat cluster is consistent with the view of a majority of the cluser, this ensures consistency.

So Raft helps us solve the problem of availability and consistency. The final requirement of any database is that of partitioning or sharding. The main reason other than replication that we want distributed database is to spread the data over different partitions. The reasoning and the criteria for partitioning can be plenty. Some may be due to locality or due to userID. While the reasons may be different, partitioning needs to ensure that the location of any piece of information is located quickly and transparently from the user.

In our project, we use consistent hashing to decide on the partition where a piece of data belongs. The loadbalancer also needs to be aware of which physical server the data is present, for that we require a log file to map the hash values to the partition address. Other DHTs like Dynamo or Google filesystem use a meta data server to keep the location of data which may not be consistent hashed to achieve a even load balance.

Another important aspect of the project, which was our secondary goal was to try and use the beautiful framework that has been buily by Jordan Halterman and the Atomix team. They have developed three projects, each of which has a clear purpose in building a distributed application.

1. Atomix - is their general distributed framework. It provides us mechanisms to create server and clients. It allows us to configure transport and data storage framework. It also runs the underlying Copycat Raft protocol. It offers various distributed resources like map and queues, etc. In my project, I have used creating my own map using copycat primitives as shown in the examples for copycat and not

atomix predefined one, I believe in terms of learning how Atomix creates the map abstraction by using copycat primitives was more useful than just using a pre-defined one. Other than these, it also offers features such as distributed lock and leader election. More details can be found here, [2]

2. copycat - is the pure distributed consensus project, I used this for my project instead of using Atomix since I wanted to make my own state machine that runs on all replicas. It is not possible to do so directly in Atomix. Copycat project allows us to create servers and clients that run the distributed protocol. It allows us to close server and leaders and automatic leader elections. It also allows us hooks into the cluster operations to react lets say when a leader election happens.

3. It is the underlying transport and communication framework, which I didnt have to deal with much since Copycat abtracted most of this out.

3 Design and Implementation

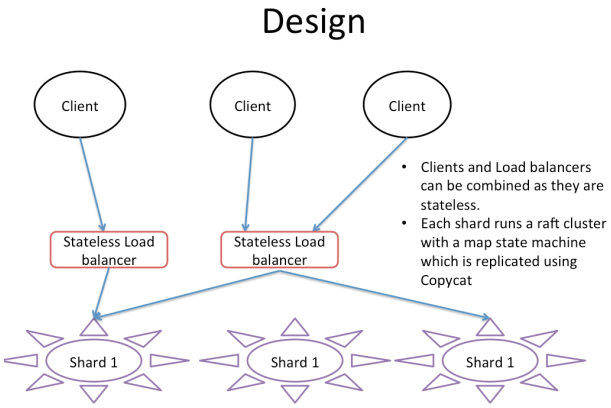

Figure 1: Design of the Atomix DDB

In figure 1, we show the design of the the distributed Key-Value Store we have designed using Atomix. Clients are the ones who want to add key-value pairs and query keys from the database. Each shard runs a raft cluster and is responsible for a subset of the data depending on the consistent hashing. The load balancer is responsible for sending the command and query to the right shard so it can store and get the data from the right cluster. The load balancer is stateless, all it needs to know is the location of the shard in terms of IP address and port number. Since it is stateless, as long as the information about the shard location is avaliable, each client can query the DB independent of others.

In terms of implementation, there are 2 main modules, the client and the server module. The server module does the task of creating the copycat servers, update the global information and partitioning the data amongst the servers and storing the mapping in a JSON state information file. Any client wishing to use the DBB has to use the config file to get the information needed.

The client module is responsible for finding the right shard to send the data to. Using consistent hashing it decides which cluster to send the data to, it consults the state information file to find out the mapping for the shard. Querying the DB is exactly the same method but we query the DB instead of sending the command to the DB.

The final module is the test module which I have designed 2 tests. First, is to test the leader election, it crashes the leader of the cluster and then sees if the cluster elects a new leader. Second, we test the consistency of the DB, we set a key value store command, immediately crash the leader and then we query the DB to see if the DB has maintained its consistency.

4 API Details

The application uses the Apache CLI command line interface to provide a neat way to add commands to the application.

Few important things to note :

1. Each command is completely independant of the other. i.e., creating a server will start a process and it is the users job to end the process when he wants to quit the server and free the sockets.
2. there is a problem with Atomix that closing servers and clients does not end the process, this maybe because threads are beign opened which are not being closed, I have tried to fix it but have not succeeded yet.

4.1 Server Setup

```
-setup NameOfDB #nodesInCluser #ofPartitions
```

Example : to set up a database with name testDB with 3 nodes in the cluster and 4 data partitions, we would use

```
Example : -setup testDB 3 5
```

This creates a server with this configuration. Now, this server is running and the state information log file has been stored. Clients can communicate with the server using this state information log file.

4.2 Client Set command

```
-set NameOfDB key value
```

Example: to add a key "bob" with value "hello" in the above db, you would use

```
Example : -set testDB bob hello
```

This command finds the right shard to commit the command to and sends a command to the raft cluster and waits for the commit message back.

4.3 Client get command

```
-get NameOfDB key
```

To query a key "bob" in the above db, you would use

```
Example : -get testDB bob
```

This command finds the right shard to query the DB and sends a query to the raft cluster and waits for the commit message back.

4.4 Clean the state

Since we are running the application on localhost it is important to keep the state clean. Also, we may want to create DB with the same name. it is a good idea to regularly do clean to clean the state information. Note: once a clean is done, all previous data is gone but it is useful during testing purposes.

```
-clean
```

cleans the state information folder.

4.5 test commands

I have designed 2 main tests to check Availability via testing the leader reelection and consistency via a test with storing and getting data after leader failure.

```
-test leaderFailure #nodesInCluster
Example : -test leaderFailure 4
```

Example : to test the case of leader failure, the above test case creates a test where a leader is crashed and checks if the other 3 nodes trigger a leader reelection.

```
-test replicationTest #nodesInCluste
Example : -test replicationTest 4
```

To test if replication happens properly even in case of failure, we send a set command to cluster and immediately crash the leader and trigger a leader reelction and when we query the cluster, it should still give us the value.

5 Installation and Running

Requires Java 1.8 and Maven.
The project is developed as a maven package so installation is a single step install. There are two ways to run this project. The first is using netbeans which is the simplest just open the project in netbeans set the run command line arguments depending on the command you want to run from the previous section.

Another method is via the command line to run the get command :

```
mvn install
mvn exec:java
-Dexec.mainClass="msuresh.raftdistdb.AtomixDB"
-Dexec.args="-get testserver hello"
```

There are other useful commands that I discovered as I was doing this project :
Due to Atomix not closing properly sometimes we may need to close ports manually, to do so in linux. run :

```
sudo lsof -PiTCP -sTCP:LISTEN
```

If mvn install fails, it is mostly because of java version mismatch, set JAVA_HOME to java 1.8. which should list the open ports by a process named java. Do kill -9 to close the java process and the associated ports.

I would suggest using netbeans as I have tested it well there. Command line might have issued, I experienced some regarding the JDK.

6 Test output

I will not show the output for all the APIs but I will show trace for the 2 test cases.

6.1 Test for replication Testing

The command :

```
mvn exec:java
-Dexec.mainClass="msuresh.raftdistdb.AtomixDB"
-Dexec.args="-test replicationTest 4"
```

The output :

```
Adding a testkey with testval to the cluster ..
Crashing leader to trigger a reelection ..
 Polling the cluster for testkey ..
The cluster returned (which should be 'testval'):testval
closing open servers..
```

6.2 Test for leader Reelection

```
mvn exec:java
-Dexec.mainClass="msuresh.raftdistdb.AtomixDB"
-Dexec.args="-test leaderFailure 4"
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building raftDistDB 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ raftDistDB ---
At host :Address[localhost/127.0.0.1:5095], initial leader :Address[localhost/127.0.0.1:5093]
At host :Address[localhost/127.0.0.1:5092], initial leader :Address[localhost/127.0.0.1:5093]
At host :Address[localhost/127.0.0.1:5093], initial leader :Address[localhost/127.0.0.1:5093]
At host :Address[localhost/127.0.0.1:5094], initial leader :Address[localhost/127.0.0.1:5093]
Crashing leader at :Address[localhost/127.0.0.1:5093]
At host :Address[localhost/127.0.0.1:5094], after reelection :Address[localhost/127.0.0.1:5092]
At host :Address[localhost/127.0.0.1:5095], after reelection :Address[localhost/127.0.0.1:5092]
At host :Address[localhost/127.0.0.1:5092], after reelection :Address[localhost/127.0.0.1:5092]
Closed server atAddress[localhost/127.0.0.1:5092]
Closed server atAddress[localhost/127.0.0.1:5095]
Closed server atAddress[localhost/127.0.0.1:5094]
Leader Reelection test is done. Program might not close properly due to a bug in Atomix. Follow manual
```

7 General Thoughts, Roadblocks and Miscellaneous

1. **The good :** The Atomix project backed by their Copycat consensus is an amazing framework for anyone who would like to build a distributed application without have to write huge amounts of code. Not only does it offer Distributed consensus, it also offers distributed locks, atomic variables and other synchronization mechanisms. Combine that with the CompletableFutures class in java, we can achieve most tasks involved in building a distributed application in a short time period.

2. **Learning curve :** Being an open source project, it comes with the usual set of problems with poor javadoc and technical manuals. It does have some decent examples on their website and distribution without which this project would not have been completed but the explanation of various APIs could be better explained. I struggled for a few days to just set up and run the servers and client, this was mainly due to the various synchronization issues that come up while setting up these servers. Also, reading the documentation about the differences between an atomix server and copycat server is important to avoid rewriting parts of the code if you choose the wrong server. They all do not have the same methods so care is required to choose the right server.

   Another important module that is a must to learn if we need to build any application using Atomix is the CompletableFuture library. What completableFuture offers us is synchronization variables and primitives to order events that happen in the application. Atomix is completely Asynchronous in the sense that server creation and client creation happen on separate thread than the main thread. Completable futures need to be used to ensure event ordering. Example: make sure that servers are all setup before a client is hooked on to the cluster to avoid race conditions. I also use completable future for testing to make sure certain events happen in order, like failure of nodes and reelection of the leader.

   I consider the knowledge I gained from using CompletableFuture as the favourite part of this project. It is indeed a well polished and useful library for any synchronous task.

3. **The bad :** Not a lot is bad in the Atomix distribution, it is still in its beta stage so documentation should be improved in the future.

Creation and tearing down of servers just dont seem to work. I spent a lot of time and tried lot of synchronization and making sure that all threads and servers are closed before the main thread ends but to no avail. I have not been able to close servers that I open, even after I use the server.close() method whihc should tear down all the resources that the server has used but there seems to be a bug here that I have not managed to fix.
Clients are instances that communicate with the server cluster. I dont think the clients communication is fool-proof. If let us say a client is given a set of members in the server and one of the servers is not online, it can lead to some strange exceptions like server not open even though the other members of the cluster are open. This problem was explained in the second test better.

4. **Coding complexity :** Overall, it was relatively simple to implement the distributed key value store after all the issues in learning curve were solved. I had to write around 600 lines of code, which is not a lot considering the complexity of the application that was developed.

   ```
   File Count : 11
   Blank : 82
   Comment : 161
   Code : 589
   ```

   For the most part, I was able to use code snippets in the examples and website in order to do the different tasks like setting up clients and servers, also setting up a state machine, which was much simpler than I expected since the state machine I required was directly in the copycat site and I just took the whole state machine from there.

   The main tasks I had to achieve was design the system, design the Command Line Interface and the parameters to the various commands. Synchronization was another task I had to take care of, which I achieved through CompletableFuture.

5. **Future Project Ideas :** I have just scratched the surface with what the Atomix project can do. There are much more interesting full scale projects that can be done. One which I have been thinking of over the past week is a distributed transaction server. The atomix project offers distributed queues to aid in this.
   Another project could be a more real-scale key-value application that can be built around it. Example, we could add authentication through kerberos, distributed configuration management using copycat so that we can swap out servers and all clients will be aware of system changes using copycats replication service.
   Lot of interesting projects can be done using Atomix and due to the presence of so many useful primitives, we can code this in a short period of time.

6. **What I would like to have done more :** I would have liked to have tested a few more scenarios. I still think there are bugs in how the client sends queries to the cluster in case of failure. I have seen that sometimes exceptions happen in tricky situations where the leader fails and the client tries to communicate with the server. But I think in a distributed application, retries would be introduced like in dynamo to wait for the system to settle down.

References

[1] Copycat distributed consensus. URL `http://atomix.io/copycat/`.

[2] Atomix distributed framework. URL `http://atomix.io/atomix/`.