Distributed key-value store with data partitioning and replication

Andrea Zorzi [190964]
andrea.zorzi-1@studenti.unitn.it

Davide Pedranz [189295]
davide.pedranz@studenti.unitn.it

***Abstract*—Distributed Hash Tables are a very effective and widely used way to store data in a distributed system. In this project, we implemented a distributed key-value peer-to-peer store with data partitioning, replication and fault tolerance inspired by Amazon DynamoDB.**

I. Introduction

Amazon DynamoDB[1] is a NoSQL key-value database that achieves high performance, reliability and fault tolerance thanks to data partitioning and replication on multiple nodes. The architecture is fully distributed and allows to add or remove nodes based on the current load and desired performances.

In this project, we built a simple distributed key-value store with support for data partitioning and replication, inspired to Amazon DynamoDB. We will first explain how to compile and run the code. Then, we will describe the architecture of the systems. Finally, we will focus on key design choices and their impact on the system.

II. Build and Run the project

The project uses the Gradle[2] Build Tool to handle dependencies and build tasks. In order to compile and build the project, the following commands should be run from the root folder of the project:

```
$ ./gradlew node
$ ./gradlew client
```

This will generate 2 JAR archives `build/libs/node.jar` and `build/libs/client.jar` with the compiled Java classes and all the dependencies needed to run the project. In particular, you can run a node with

```
$ java -jar build/libs/node.jar [COMMAND]
```

and a client with

```
$ java -jar build/libs/client.jar [COMMAND]
```

After building the project, you can bootstrap the system by running the first node with the following command:

```
$ java -jar build/libs/node.jar bootstrap
```

Once the first node is running, the other commands defined in the project guidelines can be used to perform operations on the system.

[1]https://aws.amazon.com/dynamodb/
[2]https://gradle.org/

In order to work properly, node and client's commands requires that some environment variables have been previously defined:

- `HOST` The IP of the machine where node or client is executed. *localhost* will be used as default.
- `PORT` The TCP port where to bind the node or the client.
- `NODE_ID` A unique ID for the new node that will join the system (*only for "bootstrap" and "join" command*).
- `STORAGE_PATH` The path where storage file for a node will be saved. If not provided, */tmp* path will be used as default.

An example of sequence of commands to run the system is available in the `README.md` file in the project root. The example shows how to run the bootstrap node, add some other nodes, run some read and write queries and force a node to leave the system. Here we omit the commands for brevity.

III. Testing

The project has been completely tested with unit test cases written with JUnit[3]. It is possible to run them with the following command:

```
$ ./gradlew check
```

The command compiles the project and run the test suite. The result is shown in the standard output. Please note that some test cases will spawn many times multiple Akka actors, so the suite can take some minutes to run.

IV. Architecture

The software is written in Java and based on the Akka[4] framework. The project is divided in four main packages:

- **`it.unitn.ds1.client`** Contains the commands which can be executed by a client to perform *read, write, leave* operations on the system.
- **`it.unitn.ds1.messages`** Contains the classes implementing all the types of messages that are exchanged by the Akka system.
- **`it.unitn.ds1.node`** Contains the implementation of an Akka Actor representing a node of the Distributed Hash Table (DHT) system and the logic used to handle system messages. It contains also some classes used by a node to manage the status of a client request.

[3]http://junit.org/
[4]http://akka.io/

- **`it.unitn.ds1.storage`** Contains the implementation of the file storage used by nodes to store hash tables.

The project has 2 entry point to manage the DHT system, one for the node and one for the client:

- `it.unitn.ds1.Node`
- `it.unitn.ds1.Client`

Both allow to manage the DHT system from the command line. The first Java class allows to manage the nodes which are part of the Akka system. The second class allows to perform operations on data contained in the system. In particular, operations triggered from clients are implemented using the Java Command design pattern. Every operation (*read, write, leave of a node*) is performed by a specific command which contact the requested node, submit the client request to the node, wait for the node response until a timeout is triggered and finally print to the command line the result of the operation.

All the business logic of a node is contained in the `it.unitn.ds1.node.NodeActor` class. This class handle client's operation request messages and communicate with the other system's nodes to carry out the requested operations. The status of each client request (consisting in *quorums, hash table values, node replies, etc.*) is handled by the node thanks to the identifiers associated to every request message and some "status classes" (`it.unitn.ds1.status` package). This allows the system to handle multiple requests at the same time.

The class `it.unitn.ds1.SystemConstants` defines the parameters of the system such as the read and write quorums, the replication factor and the waiting timeouts. These parameters must be changed before building the project.

V. Design Choices

A. Bootstrap command

We decided to introduce a dedicated *bootstrap* command (not defined in the project's guidelines) to run the first node of the system. Since there is no node to join at the beginning, the proposed join command with no argument may be misleading.

B. Ack on write success

When a client requests to update the value for a given key in the DHT, the node selected to handle the request starts the update procedure, contacts the other nodes and waits for the write quorum achievement. When the quorum is reached, the node sends to the other interested nodes a write request in order to update the value in the DHT and waits for the write acknowledges. The whole update operation is considered successful (and so presented as successful also to the client) only if a number of nodes equal to the write quorum number has acknowledged to the appointed node before the timeout is reached. In the other cases, the operation is considered failed even if part of involved nodes (less of the write quorum number) updated the value. We have chosen to implement the update operation using write acknowledgements because, without this check, a node could report to the client a success for a write operation even if the new value has not been written to any node due to nodes crashes or network issues. This makes the system more robust against partial failures.

C. Redistribution of keys on leaving node

When a node leaves the system, it must send its keys to the next node that will handle its keys. However, in the particular case where the number of nodes of the system is equal to the replica factor, it may happen that the leaving node sends its keys to itself before leaving. In this case, the most updated keys handled and replicated only on the leaving node would be lost. In order to avoid this scenario, a node leaving the system sends its keys to all the nodes that get responsible for them even if the key was already replicated on that nodes. This guarantees that the most recent value for each key is replicated enough in the system.

D. Operation's feasibility

To ensure the system consistency, a quorum must be reached for each read or write operation. Since each node maintains an internal list of the nodes in the system, it is possible to verify if the quorum can be reached or not before the actual poll. This check avoids the generation of unnecessary messages and provides a quicker feedback to the client in case there are not enough nodes in the system to complete the requested operation. In case there are enough nodes in the system, a normal poll is hold.

E. Akka's Ask Pattern

The client needs to contact a node in the Actor's System in order to request some operations. The normal way to do this in Akka is to start a new Actor System, start an actor, make the actor send a message to the designed node and wait for a response. Since this is a common use case and the standard approach is quite cumbersome, Akka provides the Ask Pattern[5] to solve the problem. In a single method call, Akka takes care of sending a message to the target and waiting for a response. We have used this pattern for all client's commands.

F. Java Assertions

In addition to the test suite, the code contains some assertions that verify some invariants at runtime. This helps to guarantee a correct run of the software and track down possible bugs. Assertions also helped the development and debugging process. The Java VM disables assertions by default. It is possible to enable them using the `-ea` flag of the `java` command.

VI. Conclusions

In this project, we built distributes key-store value with support for data partitioning and replication, inspired by Amazon DynamoDB. The software is very simple, yet able to dynamically scale up and down based on the current load, run on different machines, support partial failures and recover from crashes.

[5]http://doc.akka.io/docs/akka/2.4.17/java/howto.html