UNIVERSITÀ DI TRENTO

Distributed Key-Value Store

Samuele Angheben
samuele.angheben@studenti.unitn.it
Sebastiano Cassol
sebastiano.cassol@studenti.unitn.it

16 September 2023

1 Introduction

The project is to implement a distributed key-value store, a particular distributed database, using *Akka Classic* toolkit. Each node and data items that belongs to the system has an associated key and, all together, form a circular space (or ring). Clients are supposed to interact with the database through any node in the ring, that becomes a *coordinator*. A major requirement of this distributed database is to provide *sequential consistency*. The goal of this project is to demonstrate the knowledge acquired during the course *Distributed System 1* (Università degli Studi di Trento).

2 Architectural choices

This section describes the system design, the architectural choices taken to satisfy the requirements described in the project description and the actors involved.

2.1 Actors

The actors involved in the system are *clients*, *coordinators* and *data nodes*. A coordinator is a data node that, on receiving a request from the client, coordinates and supervises the correct execution of the operations. As specified in the project description, only the data node can crash.

2.1.1 Client

A client simulates a user who can interact with the distributed database trough `get` and `update` operations. A *get* operation let the client read a value associated to a specific key from the distributed database. Instead, an *update* operation let the client write a new value associated to a specific key in the distributed database. A client is required to know at least one data node to start an operation.

2.1.2 Data Node

A data node is in charge to maintain one or more *data items*. A data item is a particular type of data composed by a *key*, its *value* and the *version*. The data nodes have to retrieve or store a data item when requested by a coordinator.

2.1.3 Coordinator

A coordinator is a data node that coordinates and supervises *get* and *update* operations. The election of the coordinator is simple: on receiving a request from a client, a data node becomes a coordinator. More than one coordinator can exists at the same time.

2.2 Operations

As mentioned in the previous section, a client can perform get or update operations. However, an additional set of operations to maintain the distributed database is needed: *join*, *leave* and *recovery*. These operations are performed only as external requests, simulating a management system. As a consequence, data nodes are required to be able to perform both replication and item repartitioning. These operations could be performed only when the system is *stable*.

2.2.1 Quorum-based replication

In order to provide replication, the system relies on quorums. Therefore, four system-wide parameters are required: the number of replicas N. the write quorum W, the read quorum R and the timeout T, where $W \leq N$ and $R \leq N$. Also to avoid read/write conflicts these parameters must satisfy $R + W > N$ and $W > N/2$. These values are fixed, and they are chosen in relation to many factors such as the number of data nodes and the consistency requirements.

2.2.2 Get

The operation starts with the client sending the request to a data node. Upon receiving the get request, the interested data node becomes the coordinator of that request and start searching for the data nodes responsible of the specified key. Then, it send the read requests and start collecting the responses, saving information about the data item such as the value and the version. While a node is reading, it makes sure that the key is present and that it is not locked by an update operation in order to avoid *read after write*. Once there are at least R responses that agree on the same value and the same version, the coordinator send back the value to the client. If not enough responses come or the quorum is not reached in time T, the coordinator report an error to the client.

2.2.3 Update

The operation, like for the get, starts with the client sending the request to a data node. Upon receiving the update request, the interested data node becomes the coordinator for that request and start searching for the data nodes responsible of the specified key. Then, it send the read requests and start collecting the responses, but it saves only information about the version of the data items. While the node is retrieving the version, it makes sure that the key is present and that it is not locked by another update operation, in order to avoid *write after write*. The node proceeds locking the key and sending back the version. Once there are at least W responses that agree on the same version, the coordinator sends to all the responsible nodes the

update request specifying the new value and the version and send back the data item to the client. If not enough responses come or the quorum is not reached in time T, the coordinator sends back an error to the client.

2.2.4 Join

The join operation involves a new node and a data node, that becomes the *bootstrapping node*. This one is responsible of helping and assisting the new node to join the distributed database. The operation starts with an external request that initialize the new node, that will ask to the bootstrapping node the actual group that compose the network. Once received the group, the joining node search for the keys that is responsible for, asking to the neighbors. Finally, when all the keys are found, the node ask for the data and announce itself. Each node of the group, upon receiving the announcement, removes the keys for which it is not responsible anymore.

2.2.5 Leave

The leave operation, similarly to the join operation, starts with an external request and involves the node who is requested to leave. This node announce that is leaving to all the group and then send his data to the appropriate nodes. In the meanwhile the nodes are remove from the node group of each node.

2.2.6 Crash and recover

The crash operation simply modifies the state of the node, in order to simulate a crash behaviour. Recover instead is a more complex operation. Both start with an external operation.
To recover a crashed node, a bootstrapping node is required. The operation starts with the crashed node that asks the bootstrapping node for the group, that could be changed while the node crashed. Once received, the node updates his group, drop the data for which it is not responsible anymore and asks for new data to his neighbors. Finally, the node updates his data. If something happens during the various steps, a timeout let the node recover.

3 Code and Implementation

The project concerned is implemented using *Java* with *Akka Classic* toolkit[1]. Actors and related classes are contained in *actors* and *managers* directory, while *logger* and *utils* contains useful classes for test purposes or helpers. The actor environment is represented by `DistribuedKeyValueStore.java` contained in the *database* directory. The executable is represented by `Main.java`, that interacts with the actor environment and provide several test cases. To run the project, just uncomment the test case and type `gradle run` in terminal.

3.1 Messages

Several messages are exchanged by the nodes to implement the procedures explained in section 2.2. Thus, message handlers defines the node behaviour upon receiving a specific message.

[1] JDK 17.0.7 and Gradle 8.2.1

3.1.1 Get

Messages involved in get operation are the following:

- `ClientRead(key, coordinator)` - tells the client to start a read operation through the provided coordinator
- `AskReadData(key, requestId)` - client tells the coordinator the key to read
- `ReadData(key, requestId)` - coordinator tells the data node the key to read
- `SendRead(data, requestId)` - data node send the requested data back to the coordinator
- `SendRead2Client(value, requestId)` - coordinator send the read value back to client
- `TimeoutOnRead(requestId)` - if an error occur or the key is unknown, a timeout message is delivered to the coordinator
- `ReturnTimeoutOnRead(requestId)` - if the coordinator receives a TimeoutOnRead message, a timeout message is delivered to the client

3.1.2 Update

Messages involved in update operation are the following:

- `ClientUpdate(key, newValue, coordinator)` - tells the client to start a new update operation through the provided coordinator
- `AskUpdateData(key, value, requestId)` - client tells the coordinator the key to update with the new value
- `AskVersion(key, requestId)` - coordinator tells the data nodes the key to read and ask the current version
- `SendVersion(version, requestId)` - data node send the current version back to the coordinator
- `UpdateData(key, value, version)` - once the received versions are validated, coordinator tells the data node the value and the version to write
- `ReturnUpdate(version, requestId)` - coordinator sends back to the client the new version of the data item
- `TimeoutOnUpdate(requestId` - if an error occur, the key is unknown or the data item is actually locked, a timeout message is delivered to the coordinator
- `ReturnTimeoutOnUpdate(requestId` - if the coordinator receives a TimeoutOnUpdate message, a timeout message is delivered to the client

3.1.3 Join

Messages involved in join operation are the following:

- `AskToJoin(bootstrappingNode)` - the system tells the new node who is the bootstrapping node
- `AskNodeGroup()` - joining node asks the bootstrapping node for the current group
- `SendNodeGroup(group)` - bootstrapping node sends back the current group to the node who wants to join the group
- `AskItems()` - joining node asks the neighbors for the keys they held
- `SendItems(keys)` - neighbors send back their keys
- `AskItemData(key)` - joining node, once identified its keys, asks the neighbors the values associated to its keys
- `SendItemData(key, data)` - neighbors send back the key-value pairs requested
- `AnnounceJoin(nodeKey)` - joining node announces itself to the group. Each node in the group will add the new node to the current group and will drop the data for which they are not responsible anymore

3.1.4 Leave

Messages involved in leave operation are the following:

- `AskToLeave()` - the system tells a node that it has to leave the group
- `AnnounceLeave()` - the leaving node tells the group that it is leaving the group. Each node will remove this node from the group
- `NewData(key, data)` - the leaving node sends his data. Each node will save the data for which it is responsible

3.1.5 Crash

- `AskCrash()` - the system tells a node to crash

3.1.6 Recover

Messages involved in recover operation are the following:

- `AskRecover(node)` - the system tells the crashed node the reference of the bootstrapping node
- `AskGroupToRecover()` - the crashed node asks the bootstrapping node for the group
- `SendGroupToRecover(group)` - the boostrapping node sends back the group to the node that is recovering
- `AskDataToRecover(crashedNodeId)` - the crashed node removes the data for which it is responsible and asks its neighbors for the data items

- `SendDataToRecover(data)` - the neighbors send back the requested data that will be saved by the recovering node
- `TimeoutRecover()` - if an error occur, the crashed node will simply recover
- `TimeoutSendVersion(key)` - if an error occur while retrieving a version, the crashed node will drop the affected key

3.2 Main

First of all, an instance of the actor system is required, specifying:

- number of replicas N
- write quorum W
- read quorum R
- timeout T (in ms)
- the number of data nodes (dataNodeCount)
- the number of clients (clientCount)

Once the system is ready, *DistributedKeyValueStore.java*, that represents the actor environment, provides several methods to interact with the distributed database:

- `getClient(i)` - return the actor reference of the client i
- `getDataNode(i)` - return the actor reference of the data node i
- `sendWriteFromClient(client, coordinator, key, value)` - *client* asks to *coordinator* to write the *key-value* pair in the database
- `sendReadFromClient(client, coordinator, key)` - *client* asks to *coordinator* to read the *key* in the database
- `sendUpdateFromClient(client, coordinator, key, value)` - *client* asks to *coordinator* to update the *key* with the new *value*, if the key exists
- `join(joiningNode, bootstrapNode)` - launch the join procedure, telling the *joining* node the reference of the *bootstrap node*
- `leave(leavingNode)` - make the node leave the group
- `crash(crashingNode)` - make the node crash
- `recover(recoveringNode)` - make the crashed node recover

3.3 Logs

To prove the correct behaviour of the project, a *Testing Toolkit* is provided with *Akka Classic*. Anyway, instead of using the test kit, we decided to develop a specific logging class as it was more immediate in finding bugs and showing what was happening in the system. Initialization messages aside, a log message is composed as follows:

- *message type*: write, ask write, client read, etc.
- *time*: current time with respect to the software startup
- *message data*: key, value, requestId, version, etc.
- *sending actor*: name and type[2] of the actor that is sending the message
- *receiving actor*: name and type of the actor that is receiving the message

A special status message is provided, strictly related to the logging functionality. This message asks the entire group to print the key-value pairs that every node is holding.

4 Problems

During the development of the project, two problems in particular was very difficult to approach: verifying the correct behavior of the system and achieving sequential consistency.

4.1 Logging and testing

The first problem, as mentioned in section 3.3, has been addressed developing a detailed logging mechanism which allowed us to check all the operations that occur during the execution of the software. The choice of this solution was dictated by the fact that the testing toolkit provided by Akka would have required a lot of effort and time to be used in the right way.

4.2 Sequential consistency

The second problem, that is a major requirement of the project, is reaching a consistency level that avoid the presence of different key-value pairs for the same version in the system. Since the system relies on *quorum-based replication*, **sequential consistency** is provided by the voting mechanism. As mentioned in section 2.2.1, W and R parameters represent the minimum threshold that should be reached respectively by read and write/update requests, and the correctness of these values is reflected on the consistency level. Thus, the coordinator must get the permission by multiple replicas before a read or write operation takes place. Nevertheless, the algorithm designed in the project description doesn't reach the expected level of consistency. The main problem is that, when two simultaneous write requests occur, the system may contain different data for the same key and, in particular, for the same version. To address this problem, a lock mechanism is necessary, avoiding that inconsistent values are stored in the database.

[2]DATA NODE, CLIENT NODE and COORDINATOR

5 Conclusions

In conclusion, the project described in this report provides *read* and *update* functionality for the clients and let the nodes *join* or *leave* the system and *recover* from crashes. As required, it provides *sequential consistency*. Surely, it was very interesting and absolutely challenging to implement and develop the concepts learned during the course.

6 References

To develop this project and the report, the only resources used are the material provided by the Professor and his Teaching Assistant for the *Distributed Systems 1* course (2022/2023).