Distributed Key Value Store in Akka

Luca Mosetti[1], Davide Moletta[2]

Abstract
This report discusses the development of the Distributed Systems course project which task was to create a Distributed Key-Value Store in Akka for data replication and partitioning, with a focus on ensuring sequential consistency. The system involves multiple nodes with unique identifiers and provides clients with the ability to perform read and write operations across the network. The report outlines the system's design, assumptions, and methodologies used to meet the project's requirements. It also explains how the system ensures sequential consistency, considering factors like quorum, locks, and potential deadlocks. Lastly, great importance will be given to messages exchanged across nodes during the execution.

[1] *luca.mosetti-1@studenti.unitn.it*
[2] *davide.moletta@studenti.unitn.it*

Introduction

For the Distributed Systems course project we were tasked to create a distributed key-value store for data replication and partitioning. The project takes inspiration from Amazon's Dynamo [1]. In this report, we illustrate our project choices, structure and assumptions. Moreover, we explain how our system ensures sequential consistency among replicas.

The system consists of various nodes with unique identifiers, these nodes are used to store data in a key-value database. Each k -entry is stored by at most N nodes, chosen by following a clockwise strategy. For example, if the system consists of nodes 10, 20, 30, 40, 50, 60 and we have $N = 4$, the value with the key 15 can be stored by nodes 20, 30, 40, 50.

The system allows clients to get and put values in the store in a parallel fashion.

Finally, nodes may crash, recover, join or leave the system, thus the network can change during the execution. We developed a system that satisfies each request while guaranteeing sequential consistency.

1. System design

Figure 1 illustrates the project directory tree.

`Main.java` is a straightforward example of how to use the system. `node` and `client` folders contains the sources which describe the nodes and clients behaviour respectively. `root` folder contains the sources useful to invoke operations on the system by transparently respecting the project assumptions defined in section 2. Lastly, `test` folder contains unit and use-case tests.

In an effort to simplify the actors' responsibilities, all primary Akka actor - namely `Root`, `Node` and `Client` - follow the delegation pattern defined in section 1.1.

1.1 Delegation pattern

The delegation pattern was employed to simplify the management of operations. When a node is assigned a task, it creates a new Akka actor to which it entrusts the operation. The child actor executes the operation on behalf of the original node and transmits the outcome of the operation back to the initiating node. This pattern primarily addresses the challenge of monitoring operations, particularly those executed concurrently.

In fact, with this approach, delegated actors are responsible for executing a single operation, thus simplifying the underlying logic governing these operations.

Furthermore, were the operations handled by the node, a system for tracking ongoing operations would be necessary. For example, when a message is received, the node would need to determine the operation to which the message is directed and respond accordingly. Instead, by using delegates, messages are simply sent to specific actors.

In brief, the delegation pattern leads to less error-prone and more concise source code. Ultimately, the delegation pattern simplifies the testing process.

For the sake of simplicity, in this report we will refer to a delegated actor for node X as $\overline{X}$.

1.2 Finite State Machine

Every actor in the system is a Finite State Machine. That is, every actor can change logic only after receiving a message. The most evident example is the replica node.

Each replica node in the system can be in one of three states: minimal, redundant, or crashed (fig. 2). We use these states to know a priori if some operations can be performed or not.

Minimal When a node is in this state, the network topology counts exactly N nodes. In this state we instantly abort any Leave, since this would break **S.3**.

Redundant When a node is in this state, the network topology counts at least $N+1$ nodes.

Crashed This state is assigned to a crashed node. In this state, the node ignores any operation. The only exception is

```
src/
|-- main/
|   |-- java/it/unitn/
|   |   |-- client/
|   |   |   `-- Client.java
|   |   |-- node/
|   |   |   |-- Joining.java
|   |   |   |-- Leaving.java
|   |   |   |-- Node.java
|   |   |   |-- Reading.java
|   |   |   |-- Recovering.java
|   |   |   `-- Writing.java
|   |   |-- root/
|   |   |   |-- DidOrDidnt.java
|   |   |   |-- GetOrPut.java
|   |   |   |-- Root.java
|   |   |   `-- RootImpl.java
|   |   |-- utils/...
|   |   |-- Config.java
|   |   `-- Main.java
|   `-- resources/...
`-- test/java/it/unitn/
    |-- extensions/...
    |-- node/
    |   `-- NodeTest.java
    |-- root/
    |   `-- RootTest.java
    `-- utils/...
```

Figure 1. Project directory tree

Recover that is used to let crashed node rejoin the network.

2. Assumptions

Here we list the project assumptions:

A.1 The network is FIFO and reliable;

A.2 Replicas may join, leave, recover or crash one at a time when the network is idle;

A.3 Replicas may serve multiple clients;

A.4 Clients may operate on the same keys in parallel;

A.5 The network must contain at least one non-crashed node.

3. Specifications

Specific design decisions were taken. Contrary to assumptions, specifications are guaranteed by system assertions.

Our design choices are the following:

S.1 The unique node ID must be greater or equal to 0;

S.2 The timeout time $T > 0$;

S.3 The network cannot count less than N nodes.

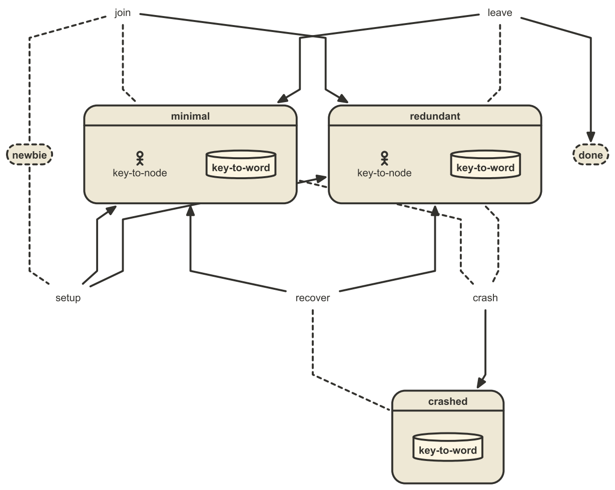

Figure 2. Node states

4. Quorum

To guarantee sequential consistency among the replicas, the project make use of quorum-based writes and reads. Such implementation is grounded in the quorum-based protocol proposed by Robert H. Thomas [2] and generalized by David K. Grifford [3]. Three parameters are configurable at compile time: N, W and R, the number of replications, the number of replicas required to perform a write and a read respectively. While choosing N, W, R, the system ensures that the following hold:

- $N > 0$;
- $R, W \in (0, N]$;
- $N < R + W$
  to ensure parallel **Get** and **Put** operations have (at least) a node in common;
- $N < W + W$
  to ensure parallel **Put** operations have (at least) a node in common.

5. Locks

Together with the quorum-based **Put** and **Get**, the project make use of writing key-locks. Such locks are granted by replicas to coordinators performing a **Put** operation. Each replica can assign at most one lock for each k-entry. Locks are eventually freed with Unlock.

5.1 Data operations

Here we list all the operations that make use of the locks.

5.1.1 Get

The **Get** operation is invoked by a client that wants to read the k-entry. This operation is triggered by a Get on k. The replica receiving such a request will be the coordinator C. $\overline{C}$ will send a Read to the N replicas responsible for k and then it waits for the responses. Whenever $\overline{C}$ reaches R replies, it sends a DidGet to the client with the most recent value among the ones received. Otherwise, $\overline{C}$ timeouts and it sends a DidnGet.

5.1.2 Read

This operation is triggered by a Read on k. The replica receiving such a request checks if it has already granted a write-lock for k. If it has not, it replies with DidRead, complete with the value and the version of the requested entry. Otherwise, it puts the request on hold. The replica will reply to all the request putted on hold as soon as it receives a Write or - when there are no lock request pending - the Unlock for the current granted lock.

5.1.3 Put

The **Put** operation is invoked by a client that wants to store a k-entry. This operation is triggered by a Put on k. The replica receiving such a request will be the coordinator C. $\overline{C}$ will send a Lock to the N replicas responsible for k and then it waits for the responses. Whenever $\overline{C}$ reaches W locks, it sends to all the locked replicas a Write, complete with the new value and the new version. Immediately after, it sends a DidPut to the client. Otherwise, $\overline{C}$ timeouts and it sends a DidnPut.

5.1.4 Lock

This operation is triggered by a Lock on k. The replica receiving such a request schedule an Unlock at the requester's termination and checks if it has already granted a write-lock for k. If it has not, it replies with Ack, complete with the stored version of the requested entry. Otherwise, it puts the request on hold. The replica will reply to a lock request putted on hold as soon as it receives an Unlock for the current granted lock.

5.2 Concurrent data operations

The locking mechanism gives rise to three potential scenarios when concurrent data operations are performed. Let A, B be coordinators and let $N = 4$, $W = 3$, $R = 2$:

- Two concurrent **Put** on k
  In this case there are three possible outcomes. If $\overline{A}$ manages to get W locks before $\overline{B}$, $\overline{A}$ will **put** and $\overline{B}$ will either **put** when $\overline{A}$ finishes or timeout. Similarly, if $\overline{B}$ arrives first the situation will be reversed. Lastly, if both $\overline{A}$ and $\overline{B}$ get 2 locks, the system will transition in a deadlock situation. In this case, both $\overline{A}$ and $\overline{B}$ need another lock to complete the operation but none of them will release any for the other. This situation always ends up in the timeout of one or both $\overline{A}$ and $\overline{B}$

- A concurrent **Put** and **Get** on k
  In this case there are two possible outcomes. Assume wlog that A has to **get**, while B has to **put**. If $\overline{A}$ requests arrives before $\overline{B}$ gets the locks, $\overline{A}$ will **get** the most recent value stored for k. Instead, if $\overline{B}$ gets the locks before $\overline{A}$ can **get**, $\overline{A}$ will have to wait for $\overline{B}$ to finish the operation. When this happens, if $\overline{A}$ did not timeout in the meanwhile, it will **get** the value written by $\overline{B}$.

```
N = 4
10 20 30 40 50

[    N     |   network   ]
20 30 40 50 10 20 30 40 50
 (^^^^^^^^^^^]
    (^^^^^^^^^^^]
       (^^^^^^^^^^^]
          (^^^^^^^^^^^]
             (^^^^^^^^^^^]
```

Figure 3. Range computation example

- Two concurrent **Get** on k
  In this situation the two **Get** will always complete no matter who arrives first. This is because we use write-locks, thus, if there are no **Put** operation currently active on k every **Get** will be satisfied.

5.3 deadlock-freedom

The project comes with an alternative version based on auctions that grants deadlock freedom.

6. Ranges

The range of a node X - denoted with φ_X - is the set of key items for which X is responsible. To compute it, a node counts N replicas starting from itself in anti-clockwise order.

Let Y be the Nth replica before X, then φ_X is the range that goes from Y excluded to X included, $\varphi_X = (Y, X]$.

Figure 3 shows an example of such procedure with $N = 4$ and 5 nodes in the network. Note that, in a minimal network state $\mathfrak{M}$, it is true that

$$\forall X \in \mathfrak{M} \text{ w.h. } \varphi_X = (X, X]$$

In other words, each node of the network is responsible for every k-entry in the network.

7. Network operations

In this section we explain how network operations are implemented, how they work and which messages are exchanged between replicas.

7.1 Join

The **Join** operation is the first operation a newbie node performs to join the network. Such operation requires a support node, say Y. $\overline{X}$ asks Y for the current network topology with Ask4key2node. If Y does not reply, $\overline{X}$ timeouts and the operation aborts with DidntJoin. Otherwise, $\overline{X}$ can compute the current range of each node in the current network (fig. 4) and the φ_X according to the network topology resulting from

```
10 20 30 40 50 60 70 80 90
^]              (^^^^^^^^^^ 10
^^^^]              (^^^^^^^ 20
^^^^^^^]              (^^^^ 30
^^^^^^^^^^]              (^ 40
 (^^^^^^^^^^^]              50
    (^^^^^^^^^^^]           60
       (^^^^^^^^^^^]        70
          (^^^^^^^^^^^]     80
             (^^^^^^^^^^^]  90
```

Figure 4. Network without $X = 45$

```
10 20 30 40 45 50 60 70 80 90
^]                 (^^^^^^^^^^ 10
^^^^]                 (^^^^^^^ 20
^^^^^^^]                 (^^^^ 30
^^^^^^^^^^]                 (^ 40
 (***********]                 45 *
    (^^^^^^^^^^^]              50
       (^^^^^^^^^^^]           60
          (^^^^^^^^^^^]        70
             (^^^^^^^^^^^]     80
                (^^^^^^^^^^^]  90
```

Figure 5. Network with $X = 45$

```
10 20 30 40 50 60 70 80 90
 ]              (           10
  **]              (        20 *
  *****]              (     30 *
  ********]              (  40 *
 (***********]              50 *
    (********   ]           60 *
       (*****      ]        70 *
          (**         ]     80 *
             (           ]  90
```

Figure 6. Node ranges that intersect with φ_X

the current network topology plus X (fig. 5 depicted with asterisks). It does so to contact all and only the replicas which handle entries belonging to φ_X with Ask4key2word (fig. 6). Once each sub-range of φ_X is covered by R responses, X safely joins the network and announce itself sending Announce to each node in the network and DidJoin to the Root. Otherwise, the **Join** operation fails with DidntJoin.

7.2 Leave

The **Leave** operation occurs when a node, say X, has to leave the network. This operation is triggered by a Leave. When this happens X can either be responsible for some entries or not. If not, $\overline{X}$ sends a Ping to every other node and, if at least one replies with Ack, X safely leaves the network sending AnnounceLeaving to every node. Instead, if X is responsible for some entries, $\overline{X}$ has to be sure that those entries will be distributed accordingly to the system specifications. Firstly, $\overline{X}$ computes the range $\hat{\varphi}_Y$ that each node $Y \neq X$ would have in a network topology without X. After that, $\overline{X}$ checks for which node Y it is true that $\hat{\varphi}_Y \cap \varphi_X \neq \emptyset$. It then sends a Ping to such nodes. For every Ack received back, $\overline{X}$ increases the counter of the sub-ranges covered by the Ack sender. If $\overline{X}$ receives at least W Acks for each sub-range before timing-out, it is sure - by **A.2** - that the data can be safely distributed. Hence, $\overline{X}$ sends an AnnounceLeaving complete with X's entries related to the recipient. Eventually, X leaves the network with DidLeave. Instead, if $\overline{X}$ timeouts, the operation fails with DidntLeave

7.3 Crash

The **Crash** operation occurs when a node has to simulate crashing. This operation is triggered by a Crash and simply put the selected node into the crashed state.

7.4 Recover

The **Recover** operation moves a node, say X, from the crashed state to the working network. This operation is triggered by a Recover specifying a support node, say Y. $\overline{X}$ asks for the current network topology to Y with Ask4key2node. If Y does not reply, $\overline{X}$ timeouts and the operation aborts with DidntRecover. Otherwise, $\overline{X}$ computes φ_X and checks if there are entries for which it is not responsible anymore. In such case, X drops those items. After this step, X recovers with DidRecover.

7.5 Clients

The **Clients** operation spawns a list of parallel clients and, for each of them, it specifies the list of **Get** and / or **Put** they have to execute. This operation is triggered by a Clients specifying the list of list of operations. Each client invokes its operations sequentially in the specified order. As soon as the last operation is completed, the client terminates.

8. Consistency

In this section we *informally* prove how each operation preserves the consistency of the data.

Def 8.1 (Consistent network)**.** *Let $\mathfrak{C} = \langle \mathbf{X}, N, W, R \rangle$, where $\mathbf{X}$ is the set of replicas, $N = |\mathbf{X}|$, and W, R respect quorum properties listed in section 4. $\mathfrak{C}$ is a consistent network iff $\forall k \in \mathscr{K}$, at least W replicas, such that each replica range contains k, have latest k-entry.*

Thm 8.1 (Consistency-preserving **Join**)**.** *Given $\mathfrak{C}$ a consistent network. After applying a **Join** operation to $\mathfrak{C}$, we have $\mathfrak{C}'$ a consistent network.*

Proof. If the **Join** operation aborts, it leaves the network unchanged, hence $\mathfrak{C}' = \mathfrak{C}$. By assumption $\mathfrak{C}$ is consistent.

If the **Join** operation completes, it adds a new replica X to the network and reduce the range of the N replicas clockwise-following X by a “relative step”. Thus, $\forall k \in \varphi_X$, X has latest k-entry and at most one replica range does not include k anymore. Thus, $\forall k \in \varphi_X$, we have at least $W + 1 - 1 = W$ replicas containing a latest k-entry. Since this operation does not influence any $k' \notin \varphi_X$ and $\mathfrak{C}$ is consistent, also $\mathfrak{C}'$ is consistent by definition. $\square$

Thm 8.2 (Consistency-preserving **Leave**)**.** *Given $\mathfrak{C}$ a consistent network. After applying a **Leave** operation to $\mathfrak{C}$, we have $\mathfrak{C}'$ a consistent network.*

Proof. If the **Leave** operation aborts, it leaves the network unchanged, hence $\mathfrak{C}' = \mathfrak{C}$. By assumption $\mathfrak{C}$ is consistent.

If the **Leave** operation completes, it removes a replica X from the network. If X contained no latest k-entry, $\mathfrak{C}' = \mathfrak{C} \setminus X$ is still consistent.

$\forall k \in \varphi_X$ such that X contained a latest k-entry, X updated at least W replicas. Thus, $\forall k \in \varphi_X$, at least W replicas contains a latest k-entry. Since this operation does not influence any $k' \notin \varphi_X$ and $\mathfrak{C}$ is consistent, also $\mathfrak{C}'$ is consistent by definition. $\square$

Thm 8.3 (Consistency-preserving **Recover**)**.** *Given $\mathfrak{C}$ a consistent network. After applying a **Recover** operation to $\mathfrak{C}$, we have $\mathfrak{C}'$ a consistent network.*

Proof. The **Recover** operation prunes X’s archive to only the entries belonging to φ_X. Thus, it leaves the network unchanged, hence $\mathfrak{C}' = \mathfrak{C}$. By assumption $\mathfrak{C}$ is consistent. $\square$

Thm 8.4 (Consistency-preserving **Put**)**.** *Given $\mathfrak{C}$ a consistent network. After applying a **Put** operation on k to $\mathfrak{C}$, we have $\mathfrak{C}'$ a consistent network.*

Proof. If the **Put** operation aborts, it leaves the network unchanged, hence $\mathfrak{C}' = \mathfrak{C}$. By assumption $\mathfrak{C}$ is consistent.

If the **Put** operation on k completes, W replicas have the latest k-entry. Since this operation does not influence any $k' \neq k$ and $\mathfrak{C}$ is consistent, also $\mathfrak{C}'$ is consistent by definition. $\square$

Thm 8.5 (Consistency-preserving **Get**)**.** *Given $\mathfrak{C}$ a consistent network. After applying a **Get** operation on k to $\mathfrak{C}$, we have $\mathfrak{C}'$ a consistent network.*

Proof. The **Get** operation leaves the network unchanged, hence $\mathfrak{C}' = \mathfrak{C}$. By assumption $\mathfrak{C}$ is consistent. $\square$

Thm 8.6 (Get safeness)**.** *Given $\mathfrak{C}$ a consistent network. If a **Get** operation on k to $\mathfrak{C}$ completes, it reads a latest k-entry.*

Proof. Assume - towards contradiction - that the theorem 8.6 does not hold. That is, the **Get** operation on k to $\mathfrak{C}$ reads a non-latest k-entry. That is, exist R non-crashed replicas such that none of them contain the latest k-entry. By definition 8.1 exist at least W replicas, such that each replica range contains k, which have the latest k-entry. By the above, we have that $N \geq R + W$. Contradiction with the quorum property $N < R + W$. $\square$

References

[1] Amazon DynamoDB - Fast, flexible NoSQL database service for single-digit millisecond performance at any scale. [Online]. Available: https://aws.amazon.com/dynamodb

[2] R. H. Thomas, “A majority consensus approach to concurrency control for multiple copy databases,” *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, pp. 180–209, 1979.

[3] D. K. Gifford, “Weighted voting for replicated data,” in *Proceedings of the seventh ACM symposium on Operating systems principles*, 1979, pp. 150–162.