Distributed and Partitioned Key-Value Store

Report of the second project of CPD

Class 9 Group 5
Maria José Valente da Silva Carneiro (up201907726)
Rodrigo Tuna de Andrade (up201904967)

June 2022

Contents

1 Problem Description

The problem proposed consisted of developing a distributed key-value store system for a large cluster, based on `Amazon's Dynamo` and its' use of consistent-hashing to store arbitrary data objects, accessed by a key, and partitioned among different cluster nodes.

Nodes can join and leave the cluster, and data can be added to the system, retrieved from it and deleted, using 3 different key-value store operations (`put`, `get` and `delete`, respectively). The system handles concurrent requests and tolerates network node crashes and message loss.

The full detailed description can be found in the `Project Proposal`.

2 Structure and Organization

We structured our system in two different sections: client and server side operations. The service nodes can be invoked using the following structure:

```
java Store <IP_mcast_addr> <IP_mcast_port> <node_id> <store_port>
```

The `Store` creates a new node in the cluster using the arguments provided: the `IP_mcast_addr` and the `IP_mcast_port` initialize the multicast membership socket, with the respective membership group address and the port of the connection; the `node_id` and the `store_port` initialize the node access point address, with the node's `ID` - it's unique `IP` address -, and the port used by the storage service.

Each node holds a `MembershipLog`, a storage `Ring` and a `FileSystem`, that configure how the node communicates and interacts with other nodes in the cluster:

- `MembershipLog`: creates and updates logs that report the membership operations in the cluster.
- `Ring`: circular key-value storage that keeps each node's `ID` and access point hashed address, in order to determine the responsible nodes for in-cluster operations.
- `FileSystem`: visual representation of the cluster data that stores each node's files in their respective folders, to evaluate the system's behaviour using only one computer.

When a node effectively joins the cluster, both the `MembershipLog` and the `Ring` are merged with the values of the other nodes in the cluster, so each node has the updated cluster information.

The `TestClient` interface tests the key-value store by allowing membership operations (join or leave) and key-value operations (`put`, `get` or `delete`). The `TestClient` can be invoked using the following structure:

```
java TestClient <node_ap> <operation> [<opnd>]
```

The `TestClient` parses each argument and applies the `operation` in the node with it's respective access point (`node_ap`). The `opnd` is the argument of the key-value operations (the filename for `put`, and the file hashed `ID` for `get` and `delete`).

3 Membership Service

3.1 Membership Interface

The `Membership Interface` contains the different membership operations which are implemented by the `Node`, the operations are triggered by the `TestClient` and are called via RMI (for more details see RMI) and are the following:

- `JOIN` : The join operation allows the Node where it is called to join the cluster, making an effective part of the Key-Value Store, the invocation can be done as such:

```
java TestClient <node_ap> join
```

- `LEAVE` : The leave operation allows the Node where it is called to leave the cluster, making it no longer a part of the Key-Value Store, the invocation can be done as such:

```
java TestClient <node_ap> leave
```

3.2 Membership Log

Every Node contains a Membership Log, contains its view of the cluster, the log is stored in non-volatile memory in order to persist node crashes and the state of a Node, as well as its knowledge of the cluster is recoverable upon reading the file.

In this implementation of the Membership Log, the file is named `<node_ap>.log` and the file is either created, indicating that the node has not yet participated on the Store; or read, given that the file existed, in order to restore its configuration.

The file is organized as set of entries, one per each line, where a entry is of the form:

```
<node_ap>;<membership counter>
```

The changes to the log are done via the class `MembershipLog`.

3.3 Cluster Initialization

3.4 Periodic Membership Message

3.5 Avoidance of Stale Information

3.6 RMI

4 Storage Service

4.1 Storage Ring

As mentioned in the Structure and Organization section, the `Ring` represents our storage service and the distributed partitioned hash table that holds each cluster node. The `TreeMap` keeps each node ordered by their hashed `ID`, that identifies it and holds the respective access point of the node.

We use consistent hashing to find the node responsible for the hashed value we want to add or map to our cluster: the next node inclusive closest to the hashed value to add is the one that stores it, taking into account the circular nature of the key-value storage (`circular upper bound`).

The `getResponsible` method retrieves a list containing the responsible node of the hashed value at index 0 and the next two following nodes (see Replication), used to deal with the key-value store operations (`put`, `get` and `delete`).

4.2 Message Structure

The key-value operation messages (`PutMessage`, (`GetMessage` and (`DeleteMessage`) extend a more general `TCPMessage` that defines the `TCP` messages generic structure: the header holds the message type and the hash key of the value to add (`PUT`), retrieve (`GET`) or delete (`DELETE`). Besides the key, `PUT` and `DELETE` messages also require a factor (see Replication).

4.3 Execution Flow

Communication between the client and the cluster is done by messages that follow the `TCP` protocol and are sent through a `TCP` server socket created when a node joins the cluster (`startTCPSocket`).

4.3.1 Client Communication

The key-value store operations are triggered by the `TestClient` that sends requests to the cluster, depending on the operation type:

- `PUT`: reads and hashes the file content and sends a `PutMessage` to the server. If the client access node is the one responsible for storing the file, it receives a code `200` from the server and it sends the file content back for it to store it, and prints the hash of the file in the client's console. If not, it receives a code `300` and it redirects the operation to another client with an access point given by the server.
- `GET`: sends a `GetMessage` to the server. If the client access node is the one responsible for retrieving the file, it receives a code `200` from the server and the file content to retrieve is read from the socket, creating a new file with that content in the current directory of the client. If not, it receives a code `300` and it redirects the operation to another Client with an access point given by the server.
- `DELETE`: sends a `DeleteMessage` to the server. If the client access node is not the one responsible for deleting the file, it receives a code `300` and it redirects the operation to another client with an access point given by the server.

4.3.2 Server Communication

Server-side operations are handled by the `TCPSocketHandler`, particularly by the `TCPMessageHandler`, in the case of the key-value service operations. The message received by the test client requesting a operation is parsed by the `MessageParser` and the operation is handled according to it's type:

- `PUT`: if the current node is responsible to store the file (taking into account it's replication factor - see Replication), the server sends the client a code `200` to indicate that. It then reads the file content from the socket and adds it to the `FileSystem`, in the correspondent node folder. Performs file replication (see Replication).
- `GET`: if the current node is one of the responsible nodes for the file (see Replication), the server sends the client a code `200` to indicate that as well as the file content from the requested file, retrieved from the `FileSystem` in the correspondent node folder.
- `DELETE`: if the current node is responsible to delete the file (taking into account it's replication factor - see Replication), the server sends the client a code `200` to indicate that and the file is not in fact deleted but is "thombstoned" from the `FileSystem` and performs deletion replication (see Replication).

In all cases, if the current node is not the correct one, the server sends a code `300` to the client to indicate that and the operation is redirected to another node, taking into account it's replication factor (see Replication), if the operation is of types `PUT` or `GET`.

4.4 Membership Events Behaviour

When a node joins or leaves the cluster, in order to guarantee that no data is lost or in the incorrect place, the key-value store should rearrange the file organization between the nodes, so that there is no need to remap all data, which would have significant efficiency set backs in the system.

Upon a `Join`, the hashed node `ID` gets all the files from the next node with keys lower or equal to its hash value. In the case of a `Leave`, he node next to the node that left gets all its files.

All the file transfers are handled by the `FileTransferHandler` that given a file and a access point node, makes a `PUT` operation of the file to the correct node after the given membership event, and taking into account the replication factor (see Replication). The file can also be deleted from the previous node if a flag is set to true.

Since the membership events behaviour closely relate to the filesystem structure implemented according to the Replication, there are more details in that section about this topic.

5 Replication

In order to increase availability, each file in the key-value store should be replicated in other two nodes, to guarantee that the file is still available even if a node goes down.

To do that, instead of getting only one node responsible for a file, upon any key-value operation in the `TCPSocketHandler`, we get an list of 3 node access points that consist of the node responsible for the hashed value file and it's next two nodes in the key-value store, by this order.

5.1 Filesystem Organization

Instead of storing all the files that belong to a certain node all together with the backup files from other nodes, each node folder contains three folders that have the replication factor of the files they hold: folder `0` holds the files that the node is actually responsible for and folder `1` and `2` hold the backup of the first and second replicated files.

6 Fault-tolerance

7 Concurrency

7.1 Thread-pools

8 Conclusion