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 `MembershipInterface` 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 `MembershipLog` and 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 `MembershipLog`, 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>
```

Where the `membership counter` is the sum of `join` and `leave` operations minus one, which means that if a nodes' membership counter is even, then it is part of the cluster and if it is odd it is not part of the cluster, because a node can not join or leave the cluster more than one time consecutively. This also means that a node's membership counter is more recent if it is higher, this means that only one entry per node is saved (the highest one) in order to save disk space, if a node perceives that a node has a higher membership counter than the on it is on its log it replaces th previous entry with the newer one.

The changes to the log and the retrieval of its information are done via the class `MembershipLog`.

3.3 Cluster Initialization

3.3.1 As the joining Node

When a node joins the cluster, it sends a multicast message to all nodes already in the cluster, said message contains only the header, the fields on the header are the type of message(”JOIN”), the node's access point for communication, the node's id, the nod's membership counter, and the node's port for connections to receive membership messages. Upon sending the message, the node opens a `TCP` socket for connections on the port sent in the message. It then waits for 3 Membership Messages, if it has not yet received the 3 messages, and a second has without receiving any new messages it then re-sends its join message and tries again to receive 3 messages, without discarding any message that might have previously received. The node trying to join sends the join message a maximum of 3 times, after that it learns the configuration via the membership messages received, how many they may be. If the node receives 0 membership messages it assumes that it is the only node in the cluster. The implementation of such protocol was done in the classes `TCPMembershipSocketHandler` and `JoinHandler`, where the socket handler is responsible for accepting connections and receiving the messages which then stores in a `BlockingQueue`, the join handler sends the join message and tries to retrieve from the queue, counting how many messages it has received.

3.3.2 As a node already on the cluster

If a node receives a join message, it then updates its view of the cluster and tries to connect to the port that was sent on the message, it then sends a membership message, which contains a header whose fields are its type (`MEMBERSHIP`), the sender's id, and a body with the 32 most recent entries on its Membership Log as well as the members of the cluster which are transmitted like:

```
<node_hash_id>;<node_ap>
```

If the joining node re-transmits its join message than the node only responds if the configuration has changed in the mean time, the way this is assured is by saving the last node this node responded to and does not respond to messages of that node.

The implementation is done on the class `JoinMessageHandler`.

3.4 Periodic Membership Message

In order to account for possible failures and lost of messages, a node multicasts a membership messages periodically, with an interval of 1 second. The message contains an header whose fields are its type (`MEMBERSHIP`), the sender's id, and the id of the next node to send the membership message (explanation will follow), and a body composed by the last 32 entries on the node's membership log.

The implementation of the protocol is done in a *Round-Robin* fashion, the node that is responsible for sending the membership message also chooses the node that will send the next one, the chosen node is the node with the smallest id that is higher than node the node that is sending the message, if that node does not exist, then the node with the smallest id is chosen.

When the receiving the the membership message if a node realizes that its id is equal to the id of the next node to send, it then schedules to send a membership message with a delay of 1 second.

In order to ensure that the messages are being sent and that the chain has not been broken, periodically every node checks that the protocol is running, by checking a boolean variable that is set to true when a membership message is received and to false when that variable is checked. If the variable is false the the node with the smallest id restarts the protocol.

The classes where the protocol is implemented are `MembershipMessageHandler` to receive the messages and merge the logs, `PeriodicMembership` to send the message and `PeriodicMembershipRecovery` to ensure that the protocol is running.

3.5 Avoidance of Stale Information

In order to avoid stale information from being sent on the membership messages, a variable *penalty* is adopted which is the number of times that and entry of the node's log was missing or was not updated.

When a node receives a join message, it is made to wait *penalty* milliseconds before sending its membership message, this will allow the nodes with smallest penalty to respond first and so increase the chance of the log sent being up-to-date.

As for the periodic message, every node sends it despite its penalty, in order to keep the *Round-Robin* structure running, however the node only sends its 32 most recent logs with probability of $e^{-penalty}$ otherwise it will send a body-less message, avoiding that the nodes have to process a log that is not up-to-date.

3.6 RMI

The client calls the functions of the node via `RMI`, the remote Interface is the defined in The `MembershipInterface`.

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 Enhancement

In order to avoid long wait times from operations that require the transfer of large files or with clusters that have a large amount of keys, we implemented an acknowledgement code verification in the server that sends code values to indicate if the operation request made by the client was fulfilled. The client first sends the operation header and only the message content after that verification was successful, in order to avoid having to send the same files and values twice to the wrong node, if in the case of a redirect. This contributes to lower wait times and a more efficient system. The inner workings of this protocol for each control sequence and operation are described in the next two sections.

4.3.2 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.

4.3.3 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.

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 subfolders 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.

5.2 Key-Value Store Operations

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.

Both `PutMessage` and `DeleteMessage` require a replication factor that represents which copy of the file is being sent. Therefore, when verifying if the current node is the one that should store the file, the current node should be equal to the responsible node at the index of the message replication factor. In the case of a `GetMessage`, we only need to verify if the current node is any of the responsible nodes, since any of them have the file to retrieve, and there is no need for further replication.

When the `PUT` and `DELETE` operations are done for the current request, if the file is the original (replication factor is 0), the operation must be replicated for the other two replica nodes. For the `PUT` operation, the `FileTransferHandler` transfers the files to the other nodes and for the `DELETE` operation the `DeleteFileOtherNode` handles the removal of the files, by sending a `DeleteMessage` to the node from where the file should be deleted.

When redirecting, the access point of the new node to access should always be the responsible node with the correspondent replication factor specified in the message.

5.3 Membership Operations

When joining or leaving the cluster, the files are reorganized by the remaining nodes, and so are their replicas.

5.3.1 Join

In the `JoinMessageHandler`, to decide which node should send their files to the new node, we get the list of responsible nodes for the hashed node `ID` and for each of them we transfer the files according to the replication factor of the folder:

- `Original File Node`: The second replicated files from this node now belong in other nodes, so they are simply transferred to their respective nodes with a replication factor of 2. The first replicated files from this node are now the second replicated files of the new node, so we change the folder name to 2 and copy the files with a replication factor of 1 to other nodes. For the original files, we transfer to the new node only the files which the new node is responsible for (the ones that have a lower or equal hashed `ID` than the new node).

- `First Replica Node`: The second replicated files from this node are all deleted since they are already being created in the original file node and would be the new node's second replicas. The first replicated files of this node that belong to the new node are deleted, since they are no longer the original file node's first replica's.

- `Second Replica Node`: The second replicated files from this node that belong to the new node are deleted, since they are no longer the original file node's second replica's.

5.3.2 Leave

In the `LeaveMessageHandler`, to decide which node should send their files to the new node, we get the list of responsible nodes for the hashed node ID and for each of them we delete the files according to the replication factor of the folder:

- `Original File Node`: The second replicated files from this node now belong in other nodes, so they are simply transferred to their respective nodes with a replication factor of 2. The first replicated files from this node are now the second replicated files of the node that's leaving, so we delete all the first replicated files from this node.

- `First Replica Node`: The second replicated files from this node are all deleted since they are the leaving node second replicated files.

- `Second Replica Node`: There are no replicated files from the leaving node in this node, so this node stays the same.

6 Fault-tolerance

The system implemented tries to maintain its consistency and availability, to do so it uses a few mechanisms to make sure that every node has the same cluster configuration and that said configuration is the correct one, for example the periodic membership messages exchanged between nodes will allow the configuration the stabilize on all nodes eventually if inconsistencies exist. And the fact that the membership log is kept in non-volatile allows a node that crashed to restore its state and its view of the cluster when re-initiating.

It is also made sure that every key-value operation is done to the correct node, as well as when the transferring of files is done when the cluster is reoriented (because of a join/leave), which allows for problems that arise from the concurrent nature of the system to be solved, i.e. two nodes have different views of the cluster momentarily.

The replication implemented also allows for a file to be retrieved even when the node responsible by it is down.

7 Concurrency

7.1 Thread-pools

A thread pool offers a solution to high thread cycle overheads and wasting resources since it reuses previously created threads to run current tasks: when the request arrives the thread is already existing so the overhead introduced by creating a new thread is eliminated and the system is more efficient and responsive.

Each node holds a scheduled thread pool that keeps 10 threads, since usually the number of cores is 8 and that seems a fair amount considering the operations the system has to execute. They are used in various occasions like transferring or deleting files between nodes and handling the different messages execution. Besides that, the thread pool allows scheduling which is useful when sending periodic membership messages.

8 Conclusion

During the development of the project we were able to learn more about many concepts related to distributed systems, such as multicasting communication and client/server processing. Understanding and exploring the challenges in building a simple yet detailed distributed system made us aware of the complexity and efficiency of many of the communication protocols we use in our daily lives. The project had a challenging nature, exacerbated by the fact that our 2-member group did a 3-person job. Overall, this project had a positive impact on us as it allowed us to deepen our knowledge to hopefully become better future engineers.