Distributed and Partitioned Key-Value Store

Bachelors in Informatics and Computing Engineering

Paralel and Distributed Computation - Project 2

Class 6 Group 1

up201906086@up.pt Marcelo Henriques Couto
up201907361@up.pt Francisco Pinto de Oliveira
up201906355@up.pt Rui Pedro Mendes Moreira

July, 2022

Contents

1 Membership Service

The membership service is composed by JOIN, LEAVE and MEMBERSHIP messages.
The classes where the membership service is implemented are divided into:

- **Message structure:** package **requests.multicast**
- **Request handling:** package **store.handlers.membership**
- **RMI:** packages **rmi** and **store.coms.client.rmi**
- **Periodic message sending and main service prodivers**: packages **store.service.periodic** and **store.service**

1.1 Join Message

Test client sends a join through RMI, the sever acknowledges it and starts the join protocol. The join protocol can be splitted in two phases.

- The multicast message sent by the joining node
- TCP connections sent by the other nodes in the cluster to a private port in the joining node

1.1.1 Multicast Join Message

1. A join message is sent through the multicast address for a specific cluster.
2. Before sending the multicast Join message, a private port is opened in a different thread and afterwards the Join multicast message is sent. The other nodes in the cluster after receiving the multicast, send through the node's private port, a membership message containing the logs history of the sender node. This can be found at: **store.rmi.MembershipProtocolRemote::join**
3. The thread listening to the cluster nodes responses, has a timeout protocol set and if at least 3 connections aren't established through the TCP private port, the protocol is executed again, until the maximum retry attempts are reached. **store.service.JoinServiceThread::run**
4. The thread only listens to the port until three connections are established, or a timeout is reached as specified.
    - If after all the retries are performed no connections are established, the node assumes that he is alone in the cluster
    - If only one/two connection is/are established, the node assumes that the cluster has one/two node(s) , starting the multicast and merging the logs information that are sent periodically to the multicast
    - If three connections are established, the node joins the protocol correctly and starts listening to the multicast

1.2 Leave Message

If the clients issues a leave request for a node that has successfully joined the cluster, the node after receiving that request issues a multicast message that notifies all the nodes in the cluster, that then register in their logs the counter of the node that left. During this process there is no need to distribute the files, since our project has a daemon job that checks if there are 3 samples of a given key/value pair and if not issues a replication process, not being necessary to distribute the files upon leave request. More details are provided in the replication section

After the multicast leave message is issued, the node is not deleted, but instead the state is set to waiting for the client. And the thread responsible for listening to the multicast for that node is stopped. This behaviour can be found at: **store.rmi.MembershipProtocolRemote::leave**

1.3 Membership Message

The membership messages are sent periodically by a scheduled task assigned to a thread in a thread-pool. The code for this can be found in the package **store.service.periodic**. The membership message is also sent in response to a JOIN request.

1.4 Structure

We use CR-LF to divide the header of the messages from the rest of the body. This is used so that we can analyze the type of message being received by executing the same function (package: **requests**, class: **NetworkSerializable**, function: **getHeader**). This allows us to then use different handlers for each type of message/request.

JOIN\r\n NodeID\r\n PrivatePort\r\n MembershipCounter\r\n\r\n	LEAVE\r\n NodeID\r\n MembershipCounter\r\n\r\n
MEMBERSHIP\r\n\r\n LOG\r\n -log entries- \r\n ACTIVE\r\n -active nodes- \r\n	

1.4.1 JOIN

The join message has no body structure. It contains, besides its identifier:

- **NodeID:** id of the node to join
- **PrivatePort:** port where the new node should receive the membership messages
- **MembershipCounter:** the membership counter of the node joining

1.4.2 LEAVE

The leave message has no body structure as well. Its contents are the same as the join message, lacking only the private port, as it does not need to receive a membership message.

1.4.3 MEMBERSHIP

The membership message header only contains its identifier. Its body is composed by two sections:

- **LOG:** followed by the 32 last occurrences in the log
- **ACTIVE NODES:** followed by the nodes that are active as perceived by the node sending the message

1.5 RMI

The path to the RMI remote interface: **src/store/rmi/MembershipProtocolRemote**

1.6 Fault Tolerance

When a we try to contact a node and the contact fails we consider that it left. Because that node needs to perform JOIN again it will update it's membership view. This part is not implemented

2 Storage Service

Our storage service consists of four request types: GET, PUT, DELETE, SEEK. As said in the handout the client is only able to send GET, PUT and DELETE requests.Regarding message format we used the message format that was recomended in appendix A.

2.1 Anatomy of Storage Service messages

2.1.1 Requests

GET\r\n Key\r\n Replicate(Boolean: must be replicated)\r\n \r\n	PUT\r\n Key\r\n ValueSize(in bytes)\r\n Replicate(Boolean: must be replicated)\r\n \r\n ValueBody
DELETE\r\n Key\r\n Replicate(Boolean: must be replicated)\r\n \r\n	SEEK\r\n Key\r\n \r\n

The need for sending the key is obvious for all the requests but the need for the replicate flag and file size in the case of PUT might not be obvious. We use the value size to determine how many bytes should we read from the message body because we didn't want to implement a complicated character escaping algorithm because it wasn't the focus of the project and because we have done it before. The need for the replicate flag will be explained late in the replication section. The implementstion details can be seen at **src/requests/store** package. Each class in this package is a data class that represents a message

2.1.2 Responses

GET

- ERROR: Key not found
- Value

PUT

- SUCCESS: File stored
- ERROR: Couldn't send the file
- ERROR: This node doesn't handle this key

DELETE

- SUCCESS: File deleted
- ERROR: Couldn't delete file

SEEK

- EXISTS
- `NOT_FOUND`

We used a text based message system because it was required by the handout and because it was easier to debug.

2.1.3 Message flow

All message flow will be described disconsidering replication. The implications of replication on this protocol will be discussed later. Because the protocol specified that the node id is the node ip, the nodes assume that they all use the same port to receive TCP requests. That port is the port that was specified in the Store command line arguments. The methods that apply the consistent hashing algorithm are located at **src/utils/algorithms/NeighbourhoodAlgorithms** and are called **findRequestDest** and **findHeir**. The code to handle requests can be found at the replication subsection because it needed to be greatly adapted to support replication

2.1.4 Store a pair

When a user wants to store a pair it sends a PUT request that was described above to the node specified in the command line arguments of the TestClient. Then by using membership information to know which nodes are available and the key we compute the node that is the destiny of the sent pair. If the pair belongs to the node the client contacted it is stored there. If it dosen't belong to the node the client contacted it will open a socket to the right node and send a similar PUT request to that node.

2.1.5 Get a value

When a user wants to retrieve a value it sends a GET request that was described above to the node specified in the command line arguments of the TestClient. If the file is available at the current node, send it. If by using the consistent hashing algorithm we find that the key belongs to another node in the cluster we send the GET request to that node and pipe the response into the client socket.

2.1.6 Delete a pair

When a user wants to delete a pair it sends a DELETE request that was described above to the node specified in the command line arguments of the TestClient. If the key is handled at the current node, delete it. If by using the consistent hashing algorithm we find that the key belongs to another node in the cluster we send the DELETE request to that node and pipe the response into the client socket.

3 Concurrency

In order to increase the efficiency of our system, we implemented some features related to concurrency, enabling nodes to perform multiple tasks in parallel.

Java offers many different ways of obtaining concurrency in a program. Even within Threads, multiple different mechanisms are available, allowing the programmer to choose between the ones suiting the situation the best. In our system, we use more than one of these tools.

3.1 Thread-Pools

3.1.1 Why?

Parallel execution allows programs to perform multiple tasks at the same time, having the potential to increase the program's performance greatly. Although threads can be phenomenal, they are hard to manage and can bring various problems when it comes to resource management. Thread creation and destruction is a heavy operation, consuming many resources. For short lived threads, the thread's creation and destruction alone can be more resource and time consuming than the tasks it will carry out.

Java thread-pools allow us to assign parallel tasks to different threads, while letting us control the number of threads and their life cycle in a simplistic manner. However, the main advantage of using thread-pools in comparison to regular Java threads lies in resource management and thread creation. Thread-pools reuse previously terminated system-threads to execute new tasks, reducing the overhead from thread creation and deletion and thus improving the system's performance and safety.

3.1.2 How?

We used thread-pools in our project to assign to different threads each request coming from the sockets. In **store.service**, both **StoreServiceThread** and **MembershipServiceThread** classes have a thread-pool as a property. These classes are responsible for listening to the multicast and server sockets. When a request is received, the thread-pool is used to assign a new task to a thread. This task will be of the class **DispatchMulticastMessage**, in the package **store.handlers.membership** for multicast or **DispatchStoreRequest**, in the package **store.handlers.store**. The objects from these classes will decifer the request's header and dispatch the rest of it to the proper handler. In sum, we use thread-pools to assign a task representing the handling of a received request to a different thread.

We also use thread-pools to assign to thread certain tasks that need to be repeated with a certain step, such as the membership message broadcast. The code for this section is in package **store.service.periodic**.

3.2 Normal Threads

Our program also uses normal java threads in some situations where the thread's lifetime is longer. Both **StoreServiceThread** and **MembershipServiceThread** class implement a java thread. These services are responsible to handle the requests coming from the store port and multicast port respectively. As such, they will be alive as long as the node is in the cluster.

4 Replication

4.1 File ownership

To get a replicaiton factor of tree we use the node that was determined by using the legacy consistent hashing algorithm plus the two nodes that succeeded that node in the ring. The implementation of these new consistent hashing algorithms can be seen at **src/utils/algorithms/NeighbourhoodAlgorithms** and are called **findReplicationNodes** and **findReplicationHeirs**

4.2 Implications at the storage services

4.2.1 Retrieving a value

Now the node that receives the get request gets a list of all the nodes that must have the file that the client requested. Then the node that the client contacted will establish a socket with each node of that list until a node that contains the value is found. This flow of execution can be observed at **src/store/handlers/store/GetRequestHandler execute** method

4.2.2 Storing a value

When the client asks the node to store a value. The node determines, by using the plain consistent hashing algorithm if the key belongs to that node. If so it stores the value and replicates that value to the next two nodes. If the file dosen't belong to that node the node creates a connection with the node that was supposed to store the file and that node handles the file storing like described above if the connection fails it moves to the next node in the line. The replication flag here is used to say if a node that received a PUT must try to replicate that value to the other nodes. Usually this value is true for requests sent by the client and false otherwise. This flow of execution can be observed at **src/store/handlers/store/PutRequestHandler execute** method

4.2.3 Deleting a pair

When the client wants to delete a pair the contacted node determines, by using the plain consistent hashing algorithm if the request must be handled by that node. If so, it replaces the pair by a tombstone that is represented by a file whose name is the former key contatenated with `"_DEL"` the content of that file is the time in millisecond of when the pair was deleted. If the node dosen't handle that request it sends the request to the correct node that then replicates the request for all the nodes that handle that pair. This flow of execution can be observed at **src/store/handlers/store/DeleteRequestHandler execute** method.

4.3 Periodic Tasks

We created periodic tasks that ran in threads to both mantain the replication factor and to delete old tombstones. The subclass that is the parent of every task is located at **src/service/periodic/PeriodicActor**

4.3.1 Maintaining the replication factor

To maintain the replication factor we run a method that in all nodes iterate over the pairs that it possesses, computes the nodes that must hold that pair, asks the nodes if they have that pair using a SEEK request that is similar to a GET request but only sends a boolean response that says if the node has a given file or not, and if the node dosen't have that pair it sends it. This task runs every 15 seconds. The code that runs every time can be found at **src/service/periodic/CheckReplicationFactor::run**

4.3.2 Deleting old tombstones

A user might have deleted a file by mistake. In order to prevent backlisting files forever this task deletes all tombstones that have more than 5 seconds. This task runs every 5 seconds. **src/service/periodic/PeriodicDeleteTombstones::run**

Fault Tolerance We thought about sending a Leave request using multicast every time a node tried to comunicate with another node and the comunication failed so that the membership would be updated and acknowledged that that node left. This part was not tested