Distributed and Partitioned Key-Value Store

Parallel and Distributed Computation - Project 2

Ana Rita Antunes Ramada - up201904565
Luísa Maria Pereira Araújo - up201904996
Maria Sofia Diogo Figueiredo - up201904675

Index

1 Introduction

The main goal of this project is to develop a distributed key-value persistent store for a large cluster.
As a distributed system, the data content in the key-value store is divided and stored in the cluster's nodes using consistent-hashing.
There are many challenges but the main one is to build a solid service that is able to handle concurrent requests and deal with possible errors, such as node crashes and message loss.

2 Service

Each node implements a remote interface that the client is able to use via RMI.
This interface includes the functions: join, leave, get, put, delete.

3 Important aspects of node functionality

Each node has three threads that run while the node is active:

1. Listens for multicast messages
2. Listens for unicast membership messages
3. Listens for the rest of unicast messages (put, get, delete)

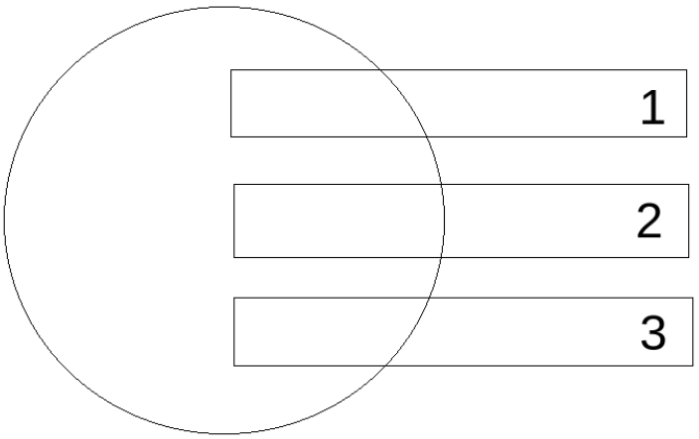

Fig 1. Each number represents a port for a listener thread

4 Membership Service

The membership service is multi-threaded, using a thread pool and it includes 2 main actions: **JOIN** and **LEAVE**.

Each node has a thread dedicated to listening for multicast messages, using a ServerSocketChannel. When it accepts a connection, a SocketChannel is created and a task

is added to the ScheduledExecutorService - from now on referred to as SES - to process the message.
In order to send messages, the node adds “send” tasks to the SES, which creates a SocketChannel to the desired node and sends what needs to be sent.

4.1 Membership messages and corresponding actions

Sending JOIN: When a node joins the cluster it sends a JOIN message to the multicast group, and waits to receive three membership messages. If it doesn’t receive said membership messages, the node will retransmit the JOIN message up to two times, sending a total of three JOIN messages if necessary. If the node never receives three membership messages, it will assume the cluster has less than three other nodes.

Sending LEAVE: When a node leaves the cluster it sends a LEAVE message to the multicast group.

Receiving JOIN: When a node receives a JOIN message, it saves the new node’s information in its view of the cluster members and adds a log entry. Both the members' information and the log are saved in non volatile storage.

Receiving LEAVE: When a node receives a LEAVE message, it removes the new node’s information from its view of the cluster members and updates the log entry.

4.2 Message format and description

The recommended message format in the section “Appendix A: Generic Char-Based Message Syntax” from the project specification was used in our implementation of the key-value store. All messages are char based and human readable.

Here’s an example of a **JOIN** message:

ACTION JOIN
ID 1edd62868f2767a1fff68df0a4cb3c23448e45100715768db9310b5e719536a1
COUNTER 0
ADDRESS 127.0.0.1
MEMBERSHIP_PORT 5000
PORT 6000

BODY

Here’s an example of a **LEAVE** message:

ACTION JOIN
ID 1edd62868f2767a1fff68df0a4cb3c23448e45100715768db9310b5e719536a1
COUNTER 0

BODY

The **BODY** of most messages is empty, but is used to send file content in **PUT** messages.

4.3 Periodic log messages

Each node in the cluster will send periodic LOG messages. In order to minimise the propagation with stale information, nodes with less members in their view of the membership will have longer periods between sending LOG messages.

Nodes with higher sum of its log counters are more likely to not have stale information when compared to nodes with lower sum. Let's look at an example:

Node A	**Node B**
1edd62 3 asfbj1 2 fksdsj 1 1edd62 2 asfbj1 2 fksdsj 1 1edd62 1 asfbj1 0 1edd62 0	asfbj1 2 fksdsj 1 1edd62 1 asfbj1 0 1edd62 0

For these two nodes, we calculate the sum of the counter, for A it is 12 and for B it is 4. In order to prevent B from multicasting stale information, we increase the time between B's log messages.
The expression for the time between sending log messages will be: T=1 + 1/sum (seconds).
This makes it so A sends logs every 1.08 seconds and B sends it every 1.25.
This concept for avoiding propagation of state information was not implemented due to time restrictions.

5 Storage Service

5.1 Finding nodes responsible for pairs

In order to find the nodes responsible for key-value pairs, binary search is used, following consistent hashing logic. Since replication is implemented, there is always one node that owns a pair, and two other nodes that replicate it, as described in 6. Replication

5.2 Pair transfer on membership changes

When a node joins or leaves the cluster, the key value pairs are redistributed. This means that upon membership changes, nodes recalculate key ownership and transfer pairs to the new owners. Since replication is implemented, as described in section 6. Replication, this pair transfer on membership changes, includes transfer to replicator nodes.

5.3 When the key-value pair owner node crashes

The store is resistant to this type of failure, since there is a replication factor of 3. If a node receives a GET request from the client it will contact the owner of the key-value pair, and in case it doesn't answer or it doesn't have the pair anymore, it will contact the replicators one by one until it gets the value needed.

5.4 Message format and description

Here's an example of a **GET** message:

ACTION GET
KEY 1edd62868f2767a1fff68df0a4cb3c23448e45100715768db9310b5e719536a1

BODY

Here's an example of a **PUT** message:

ACTION PUT
KEY 1edd62868f2767a1fff68df0a4cb3c23448e45100715768db9310b5e719536a1

BODY
Hello World!

Here's an example of a **DELETE** message:

ACTION DELETE
KEY 1edd62868f2767a1fff68df0a4cb3c23448e45100715768db9310b5e719536a1

BODY

6 Replication

To increase availability, key-value pairs are replicated with a replication factor of 3, i.e. each key-value pair is stored in 3 different cluster nodes. This means that, at any given time, if the client requests a value, only one of the nodes that stores the pair will need to be available.
For replication we opted to replicate on the next two successors of the key owner.
Let's look at an example:

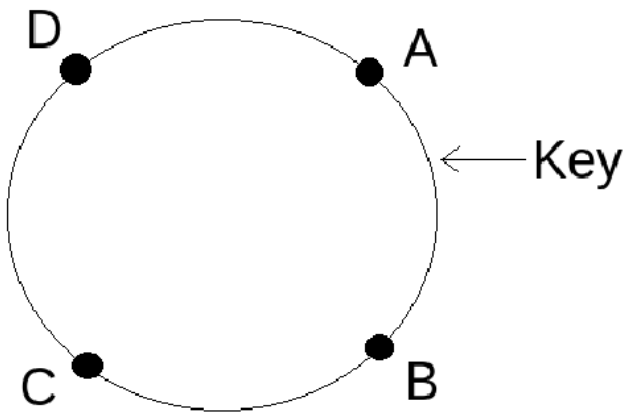

Fig 2. Each represents a node in a consistent hashing system.
For this example, the key value pair would belong to B, while C and D would replicate it.

6.1 Ensuring that the number of copies is equal to the replication factor

If, at any moment, there are less than three nodes in the cluster, and a PUT operation occurs, the node that receives the PUT from the client will redirect the PUT message to the replicator nodes, and check if the replication factor is three. In case the replication is less than desired, the node will wait an arbitrary amount of time (i.e. 10 seconds) before checking if any more nodes joined the cluster and sending a PUT message to the replicator nodes. This will be done infinitely until the replication factor of three is guaranteed.

7 Concurrency

7.1 Thread-pools

This key value store takes advantage of a Scheduled Executor Service, that allows tasks to be run in parallel while having no need to waste time creating and deleting threads because all threads are created at the startup of the node. This makes the store more efficient because creating and deleting threads would have a big overhead, which is not desirable on a high scale key-value store.
This means that the node is able to handle simultaneous requests, both from clients and other nodes.

7.2 Asynchronous I/O

There was an attempt to implement asynchronous I/O with a Selector and Selectable Channels, but because of time constraints and difficulty conceiving a solution based on a Selector, we were unable to implement it.

Asynchronous I/O would make the store faster as the threads connecting, writing or reading to/from a socket wouldn't need to block. The selector would look through the Selectable Channels and in the case it was writable, a thread would be assigned to write what was needed to it, in the case it was readable, a thread would be assigned to read from it, and so on.

The usage of a Selector would lead to less thread time wasted, as no thread would be blocked while waiting to connect, read or write. With a Selector there would be no need to have several threads listening to sockets as we implemented. Therefore "minimising" the number of threads.

8 Remote Method Invocation

We are using Remote Method Invocation, RMI, between the Client and Nodes, allowing the Client application to execute remote methods. RMI is often used to implement Java distributed systems since it allows Java applications to write and access remote objects in a distributed system.

RMI Advantages
The main advantages of using this mechanism are being able to pass full objects as arguments and return values (for example, instead of only being able to pass predefined types you can also pass Java hashtable objects), moving classes implementations from client to server and vice versa and, at last, using Java built-in security mechanisms (these make the distributed system safe in situations where its users are downloading content).

Why use RMI instead of other alternatives, such as TCP Sockets?

TCP sockets allow distributed components to communicate with each other but there is no such thing as a reference to a remote object.

9 Conclusion

This assignment was definitely challenging in various ways. In the development process we had a few obstacles. One aspect worth highlighting is the Selector implementation. As explained above, we wanted to use this mechanism to make our application more efficient and faster.
Overall, with the help of what we learned during our theoretical and practical lessons we were able to conclude this assignment, implementing most of the features planned and acquiring more knowledge of distributed systems.