Bachelor in Informatics and Computing Engineering

Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing

David Preda - up201904726
Fernando Rego - up201905951
Miguel Amorim - up201907756

June 3, 2022

Contents

Chapter 1

Problem Description

A key-value store is a simple storage system that stores arbitrary data objects, the values, each of which is accessed by means of a key, very much like in a hash table. To ensure persistency, the data items and their keys must be stored in persistent storage, e.g. a hard disk drive (HDD) or a solid state disk (SSD), rather than in RAM.

By distributed, we mean that the data items in the key-value store are partitioned among different cluster nodes.

Our design is loosely based on Amazon's Dynamo, in that it uses consistent-hashing to partition the key-value pairs among the different nodes.

Chapter 2

Message Format

The communication is the most essential part of this project. The distributed system need constant communication and data exchanges between nodes. This communication does not work without a constant and reliable message format to easily send and read information contained in every type of message.

All message types are composed of a header and a body:

2.1 Header

The header consists in a sequence of ASCII lines started and terminated by special characters (CR: *0xD*; LF: *0xA*). Each header line represents a message field and its value separated by a space. There are a variaty of message fields that can be present in the header of the message like the message type, indicating the operation associated to the message (join, leave, put,get, delete), or indicating that it refeers to a membership message, as well the size of the contents to easily read all messages.

2.2 Body

The body contains the data, or by another words, the content of the message that is supposed to arrive at the destination point. The content differs according to the different message types, it can be a key to access and make operation in a stored value or a file to store in a node, as also can be some node information like the address and port, or the cluster view.

Chapter 3

Membership Service

The message format is detailed described in the Chapter 2

The membership service is responsible for ensuring that the distributed network is well managed and that each node is aware of the existence of others. To achieve this purpose, a class representing the service, MembershipService, is integrated into the Node class so it communicates in an efficient fashion.

Let us now break down how this class handles

3.1 Tasks

Every time a MembershipService-related message is received, either through TCP or UDP (depending on the message type), the service handles it in another thread, in the form of a Task. A Task is a family of classes that extends the Thread class, responsible for handling one and only one specific type of message.

After creation, every Task is passed to the membership service's ThreadPool, which executes it when one of its Threads is available. This is a simple and generic way of handling all kinds of messages while keeping the program very scalable, which made the integration of new features easier.

This implementation uses a custom ThreadPool implementation rather than the ThreadPoolExecutor class provided by Java. This allowed the authors to learn more about multithreaded-programming and to have a larger control over what is executing and what isn't.

3.2 View of the Membership

Each Node needs to have the network easily represented to it. Therefore, the MembershipService also contains a View, a class responsible for keeping record of the state of the membership.

The view class keeps two hash maps:

- entries - a map relating hashed node IDs to an entry with each node ID. Each entry contains the address, the store port, the membership counter and the seconds since epoch for the moment when the information was received;
- upEntries - the same as the previous map, but only for online nodes. It allows for a better performance on certain communication needs, specially for the leader (explained in the Leader section).

When transmitted via TCP, the view keeps each entry in the following format:
$\mathrm{hashed\text{–}ID}\,;\,\mathrm{address}\,;\,\mathrm{port}\,;\,\mathrm{counter}\,;\,\mathrm{seconds\text{–}since\text{–}epoch}$

3.3 Updating Everyone's Information

A distributed network requires that for everyone to know everyone (or, at least, a subset of everyone). Therefore, it is required that this information is transmitted by some node, hopefully the most knowledgeable one. This problem is a common distributed network problem, known as the leadership election problem.

Our algorithm takes advantage of the consistent-hashing approach to the database. Every 10 seconds, all nodes decide they want to become the leader. Therefore, they try to pass that message through everyone through the consistent-hashing ring, until the message comes back to themselves.

If a node receives the intent of another node becoming leader, it compares both their Views of the membership. If the wannabe-leader has a more up to date View of the membership, it's message continues along the ring. Otherwise, it is stopped, never reaching itself.

If the intent message reaches the the node who, originally, sent it, that means that all other nodes have an inferior or equal View of the membership. The node then becomes the leader, updating everyone else with its View of the membership for, at least, the next 10 seconds.

The way the comparison is made, it is guaranteed that only one node becomes the leader. The following code snippet illustrates how this process is initialized:

```
public class LeaderSearch implements Runnable {
        private final String nodeId;
        private final MembershipService membershipService;

        public LeaderSearch(String nodeId, MembershipService membershipService) {
                this.nodeId = nodeId;
                this.membershipService = membershipService;
        }

        @Override
        public void run() {
                View view = this.membershipService.getView();
                LeadershipMessage leadershipMessage = new LeadershipMessage(nodeId, nodeId, view);

                while (true) {
                        ViewEntry nextNode = view.getNextUpEntry(UtilsHash.hashSHA256(nodeId));
                        if (nextNode == null) {
                                return;
                        }
                        try {
                                Socket socket = new Socket(nextNode.getAddress(), nextNode.getPort());

                                UtilsTCP.sendTCPMessage(socket.getOutputStream(), leadershipMessage);
                                System.out.println("[M] Asking to become the leader. Redirected to "
                                 + nextNode.getAddress());
                                break;
                        } catch (Exception ignored) {
                                this.membershipService.flagNodeDown(nextNode.getAddress());
                        }
                }
        }
}
```

With this, each node then receives a LeadershipMessage, processed by on the aforementioned Tasks:

```
public class LeadershipTask extends Thread {
	private final View view;
	private final String nodeId;
	private final String leadershipMessageString;

	public LeadershipTask(View view, String nodeId, String leadershipMessageString) {
		this.view = view;
		this.leadershipMessageString = leadershipMessageString;
		this.nodeId = nodeId;
	}

	@Override
	public void run() {
		LeadershipMessage leadershipMessage = new LeadershipMessage(leadershipMessageString);
		String wannabeLeaderId = leadershipMessage.getLeaderId();

		if (wannabeLeaderId != null) {
			Message message;

			if (wannabeLeaderId.equals(nodeId)) {
				message = new CoupMessage(this.nodeId, this.nodeId);
				System.out.println("[M] Informing everyone that "
				+ this.nodeId + " is the new leader.");
			} else {
				int comparison = this.view.compareTo(leadershipMessage.getView());

				if (comparison > 0
				|| (comparison == 0 && this.nodeId.compareTo(wannabeLeaderId) < 0)) {
					System.out.println("[M] Rejecting "
					+ wannabeLeaderId + " proposal as the new leader.");
					return;
				}

				message = new LeadershipMessage(this.nodeId,
				wannabeLeaderId, leadershipMessage.getView());

				System.out.println("[M] Approved "
				+ wannabeLeaderId + " as the new leader.");
			}

			ViewEntry nextNodeInfo =
			this.view.getNextUpEntry(UtilsHash.hashSHA256(this.nodeId));

			try {
				Socket socket = new Socket(nextNodeInfo.getAddress(),
				nextNodeInfo.getPort());
				socket.setSoTimeout(3000);
				UtilsTCP.sendTCPMessage(socket.getOutputStream(), message);
				System.out.println("[M] Informing "
				+ nextNodeInfo.getAddress() + " about the leadership decision.");
			} catch (Exception ignored) {
				ignored.printStackTrace();
			}

		}
```

Finally, it is worth noting that the search for the leader, as well the leader updates, are executed periodically with the help of the ScheduleExecutorService class. The search is executed every 10 seconds by every node, while the leader updates are given by the leader every second.

3.4 RMI

The RMI (Remote Method Invocation) was used for both membership operations, join and leave. The file with the definition of the remote interface can be found at the following path:

`g05/assign2/src/client/Services.java`

Chapter 4

Key-value Store

The message format is detailed described in the Chapter 2

The key-value store is a distributed partitioned hash table in which each node stores the key-value pairs. To partition the key-value pairs among the nodes, the store uses a hashing technique, *consistent hashing*. This technique allows to easily resize the hash table without remapping the present keys in the table.

4.1 Implementation

The store has three major operations:

4.1.1 Put

The put operation send a value to the cluster and generate a key related to the value with the goal of add a new key-value pair to the store

Upon a put operation, a message of the type *PutMessage* is sent with the content to store in a node of the cluster. To find the responsible node to store the new value, the hash (key) of the new pair is calculated and, according to the consistent hashing technique and a simple binary search, the node can easily be found. Then the message is sent to the that node where the new key-value pair is stored.

After the node store the new key-value pair, an acknowledge message of the type *PutMessageReply* will be sent back to the client with the new generated key so that the client can access the stored value.

4.1.2 Get

The get operation send a key to the cluster retrieves the pair associated value.

Upon a get operation, a message of the type *GetMessage* is sent with a key. This key is associated with a value and the pair is stored in some node. The responsible node for this pair is calculated according to the consistent hashing, and then, a request of the value is sent to obtain the pretended value.

When the pretended value is accessed, a reply message of the type *GetMessageReply* will be sent back to the client with the pretended value which will be saved in the client's folder at the disk.

4.1.3 Delete

The delete operation send a key to the cluster which deletes a key-value pair and retrieves an acknowledge message.

Upon a delete operation, a message of the type *DeleteMessage* is sent with a key. This key, like in the get operation, is associated with a value and the pair is stored in some node. The responsible node for this pair is calculated according to the consistent hashing, and then, a request to delete this key-value pair is sent to that node.

When the pretended value is accessed, an acknowledge message of the type *DeleteMessageReply* will be sent back to the client with the final state of operation, in other words, if the operation was successful or failed.

Chapter 5

Replication

Replication is an important factor for distributed computing as it increases the availability of a service, in this case a file, throughout the distributed network.

5.1 Implementation

Implementing replication consists of keeping a factor of 3, by another words, each key-value pair must be on 3 nodes of the cluster. This allow the client to always obtain the key-value pair, even if one of the nodes is down, because there will be another node with the pretended key-value.

Therefore, for each operation, the following tools for keeping the replication factor were implemented:

5.1.1 Put

Upon a put operation, the value associated to the operation is stored in the first active node according to the consistent hashing, and then, if possible, the file is replicated to the next two active nodes in the cluster.

5.1.2 Get

Upon a get operation, the responsible node is calculated according to the consistent hashing, and then, a request of the value is sent to obtain the pretended value. If, for some reason, the value can't be obtained from that node, the same request is made to one of the next two nodes that follows the responsible node.

5.1.3 Delete

Upon a delete operation, the responsible node is calculated according to the consistent hashing, and then, a request to delete a key-value pair is sent to responsible node as well to the two next active nodes in the cluster.

5.1.4 Join and Leave

Upon a join or a leave membership operation, for each key-value pair, the nodes that will store the pair or the replicated pairs are recalculated to make the necessary changes of key-value pairs between the nodes to mantain the replication factor of 3 key-value pairs.

5.2 Implications on membership and storage devices

The implementation of the replication involved both membership and storage devices and forced some significant changes to this devices.

To reach the full potential of the replication, the membership should always be updated to minimize any error when the nodes are calculated to perform any operation of the storage. This had some implications of operation synchronization in order to keep the replication factor of 3 and to store the key-value pairs in the right nodes and minimize the probability of errors occurring to the maximum.

Chapter 6

Fault-Tolerance

Distributed networks must have mechanisms to deal with faults. This project contains some, briefly described below.

6.1 Connection Crash

In case a node crashes, it has no way to save whatever information it had before crashing. If some information was kept, it is brought back upon reinitialization of the program - the latest view of the membership is restored and the replication factor is kept in place for all the locally saved files.

In case the node tries to reconnect with an outdated membership counter, all the other nodes inform it that it is outdated and tell them the latest counter it had sent. The node can then recover its latest membership counter, allowing it to enter the network and resume its work.

6.2 Leader Crash

A worse version of the previous problem is if the node that crashes is the leader. However, the leadership election algorithm handles this problem quite easily.

Given its periodic nature, if the leader crashes, the algorithm will elect, in at most 10 seconds, a new leader. However, dead nodes are a problem for this algorithm - they will not be reachable and will stop messages from going full circle (or ring).

In order to solve this, the leadership election algorithm marks the node as down and tries to find the node after the crashed one. This processed is repeated

until a node is found. In an extreme case, the node just reports back to itself, as the last node remaining in the network.

Therefore, the leadership election algorithm solves two problems at once - it handles dead nodes and assures an updated leader is online for most of the time.

Chapter 7

Thread-pools

Our program uses a custom ThreadPool implementation, as aforementioned. The following code snippets illustrate how the ThreadPool was implemented. Its use is explained in the relevant section.

```
public class ThreadPool {
        private BlockingQueue<Thread> tasks;
        private List<PoolThread> threads;
        private boolean closed;

        public ThreadPool(int threadsNum, int maxTasksNum) {
                this.tasks = (BlockingQueue<Thread>) new ArrayBlockingQueue<Thread>(maxTasksNum);
                this.threads = new ArrayList<>();
                this.closed = false;

                for (int i = 0; i < threadsNum; i++) {
                        threads.add(new PoolThread(this.tasks));
                }

                for (PoolThread thread: threads) {
                        thread.start();
                }
        }

        public synchronized void execute(Thread task) {
                if (!closed) {
                        tasks.add(task);
                }
        }

        public synchronized void stop() {
                if (!closed) {
                        /*for (PoolThread thread: threads) {
                                thread.stopThread();
                        }*/
                        tasks.clear();

                        this.closed = true;
                }

        }

        public synchronized void reopen() {
                if (closed) {
                        for (PoolThread thread: threads) {
                                thread.start();
                        }

                        this.closed = false;
                }
        }

        public void waitForTasks() {
                while (tasks.size() > 0) {
                        try {
                                Thread.sleep(1);
                        } catch (InterruptedException ignored) {
                        }
                }
        }
}
```

```
public class PoolThread extends Thread {
        private final BlockingQueue<Thread> tasks;
        private Thread thread = null;
        private boolean stopped;

        public PoolThread(BlockingQueue<Thread> tasks) {
                this.tasks = tasks;
        }

        @Override
        public void run() {
                this.thread = Thread.currentThread();
                while (!stopped) {
                        try {
                                Thread task = tasks.take();
                                task.run();
                        } catch (Exception ignored) {
                        }
                }
                super.run();
        }

        public synchronized void stopThread() {
                if (thread != null && !stopped) {
                        this.thread.interrupt();
                        this.stopped = true;
                }
        }

        public synchronized boolean isStopped() {
                return stopped;
        }
}
```

Chapter 8

Conclusions

This project allowed us to deeply understand some computer concepts related to distributed systems and force us to solve several different type of problems.

The development of this project provided us important knowledge about communication protocols, such as TCP and UDP, as well the implementation and the usage of these protocols. Allied to the communication, the synchronization problems that appear during the development of this project, force us to think *out of the box* to solve an variety of different problems. Finally, this project gave us a little idea of how large distributed systems are implemented in real world scenarios.

8.1 Work Distribution

- David Preda - MembershipService, Fault Tolerance, ThreadPools and respective parts of the report;
- Fernando Rego - Key-Value-Store, Replication and respective parts of the report;
- Miguel Amorim - Remaining part of the report and local file saving functions;