Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing
Bachelors in Informatics and Computer Engineering

3L.EIC01_G5
Joel Fernandes up201904977@up.pt
Mário Travassos up201905871@up.pt
Tiago Rodrigues up201907021@up.pt

June 3, 2022

1 Membership service

1.1 Messages

The messages mostly follows the "Generic Char-Based Message Syntax" protocol recommended by the professor on the project specifications at `https://web.fe.up.pt/~pfs/aulas/cpd2122/projs/proj2/proj2.html#MessageSyntax`.

The main differences are in the way the header is structured, as well as the body data. First of all, it is important to highlight that the header is made up of a single line. Each header field is separated by a single space character, and each field is set up as a key value-pair, with the exception of the first one, which is just the type of message that was sent.

The key acts as an identifier for the field, like "id" for the id of the sender, or "membership" for his membership counter. Each message carries with it the relevant key value pairs. The specific values are shown below. It is to note that only the membership messages of the protocol need to carry a body, as the others do not carry data that is relevant to be in the body of the message.

1.1.1 JOIN Message

For the JOIN Message, according to the specification, it was relevant to include the senders ID, his current membership counter (which should be an even number every message), as well as the senders TCP port. For this implementation, it was decided to also include the senders IP address, as it cannot be guaranteed that it will be the same as the ID.

The `composeMessage` method present in the `JoinMessage` class delivers a byte array with the message in the appropriate format, ready to be sent to the remaining nodes in the cluster.

An example JOIN message is shown below. Note that the values are arbitrary:

```
1    JOIN id:31 membership:4 port:8080 ip:193.136.33.132
```

1.1.2 LEAVE Message

Again, for the LEAVE Message, and according to the specification, it was relevant to include the senders ID, and his current membership counter. For this implementation, it was decided to stay true to the specification, as it was not deemed necessary to include any extra fields.

The `composeMessage` method present in the `LeaveMessage` class delivers a byte array with the message in the appropriate format, ready to be sent to the remaining nodes in the cluster.

An example LEAVE message is shown below. Note that the values are arbitrary:

```
1    LEAVE id:31 membership:5
```

1.1.3 MEMBERSHIP Message

Finally, for the MEMBERSHIP Message, the specification mentions that it should include the current members **as seen by the sender**, as well as his 32 most recent log messages. For this implementation, it was also decided to stick to the specification, since it didn't need any extra fields.

The members are represented in the header, as a set of dash-separated strings, with each string being an ID of a node. The body is sent as a list of strings, each representing a log line, with the newline character "\n" separating them.

An example MEMBERSHIP message is shown below. Note that the values are arbitrary:

```
1    MEMBERSHIP id:31 members:1-4-15-31
2
3    31 JOIN 0
4    1 JOIN 0
5    2 JOIN 0
6    2 LEAVE 1
7    15 JOIN 0
8    4 JOIN 0
```

1.2 Other details

1.2.1 Use of RMI

The Store implements a RMI interface that can be found in

`src/communication/RMI.java`

1.2.2 Preventing Stale Information in Membership Messages

According to the specification, in order to prevent stale information, a node shall multicast a membership message periodically. In this project, this node is chosen based on the sorted list of cluster members. The first node in the list is chosen as the "leader", and will periodically multicast the his view of the membership. Since the information is being sent to everyone periodically, there is only a small chance of delivering stale information, as over time, even if the node misses some events, he will be updated by the newer ones who will replace him as "leader".

2 Key-Value Storage Service

2.1 Messages

The messages follow the same format as the membership service messages, but include different fields. In the header, every message includes the type of message (which can be either PUT, GET or DELETE), the IP and the Port of the Client that made the request. Finally, the body of the message includes the operand for each method (the key for GET and DELETE, and the value for PUT).

The information above is passed from node to node as well, since this way, when the node that received the request does not have the information, he does not need to be involved in handling that request (for example, by acting as a middle man between the correct node and the client), and can move on to newer requests.

2.1.1 PUT Message

Message format:

```
1    PUT ip:193.136.33.132 port:4321
2
3    This is the content of the passed to the client
```

2.1.2 GET Message

Message format:

```
GET ip:193.136.33.132 port:4321

0c3392045342bf3019bf8eedb0a7c015d0e33772cc21c6ef533ccb
1ceddbd77b
```

2.1.3 DELETE Message

Message format:

```
DELETE ip:193.136.33.132 port:4321

dfc5721126b16938a935678b9ee60e7463d0065561543e266fbac4
fbb487fcc5
```

3 Fault tolerance

3.1 Use of Periodic Membership Messages

Even though the periodic membership messages are a bit rudimentary and only send the 32 most recent messages, they scale really well as they are simple for all systems, and in a large cluster, even if one node is down, the others will capture them and store them for when the node comes back up, and update him properly after he is back in the cluster.

4 Concurrency

4.1 Use of Thread Pools

Thread pools are used in the nodes to increase the availability for receiving messages. There are 2 main thread pools that are created. One for handling Multicast messages, in the `MulticastDispatcher` class, and another for TCP messages, in the `TCPDispatcher` class.

Each one of these methods has a loop that waits to receive a message and upon getting one, immediately launches a thread to handle it, so that the main thread can keep listening for new requests. this way, it won't block when listening to other nodes, potentially failing to detect important messages.

This thread pools are cached, to allow the OS to do load balancing and automatically manage them, adding more to one or another as needed, creating a more streamlined process that wouldnt be possible with, for example, scheduled thread pools.

There is also a third thread, in the `PeriodicMulticastMessageSender` class, whose job is to handle the periodic MEMBERSHIP messages. It is single threaded because it only sends messages, and does not need to be in a tight loop waiting for messages or responses, or launch any threads to handle any remaining logic.