Report 4: groupy

Gabriele Morello

October 2022

1 Introduction

In this laboratory we implemented a simplified version of the Chord protocol, a protocol for a peer-to-peer distributed hashing table. In this protocol we will have a ring structure where each node know his successor and his predecessor, and of course his own data, we will not have a full routing table as specified by the Chord protocol. Data is stored with a key and a value, each node is responsible for a segment of keys.

2 Main problems and solutions

We have four modules: key, storage, node and test. key manages keys utilities like creation and checking if a keys belongs in an interval, storage manages the store field in the node: create a storage, lookup of a value given the key, splitting and merging of storage. The node module has 4 versions that add the functionalities described below:
The first module allows the creation of a network with no failure handling where we can only add nodes.
The second module node2 add the possibility to store data in the DHT, we introduce there the Store variable and two commands: add and lookup, respectively to add and retrieve a value in the DHT, at this stage I also creates the storage module to provide storage functions.
The third module node3 covers the first optional task, the goal is to handle failures, it does so by adding a monitor to each node for the successor and a new variable to store the successor of the successor, in case of failure of the successor a node will connect to the successor of the successor. More details will be explained later.

2.1 node1

In node1 I had to implement the stabilize procedure to update links, it works by sending a request to a possible successor and expecting a status response that states its Precedent node, based on what the response is the node can either notify the successor to add him as predecessor, or send a new request to the predecessor of the predecessor.

The check is done by key:between. A stabilize call is made every second. The request function was given. notify is called when another node wants to be the predecessor, before adding it we still have to check that the request is valid, in case the predecessor in null or its Id is lower (in module) than the requesting node we add the new node otherwise we keep the old predecessor. start and init are given but in connect I had to specify the returned value.

To make this module work I had to create a key module with key:generate() and key:between(Key, From, To), the first returns a random key, the second checks if a key is inside an interval (From, To] but interval are modular because we are operating on a ring.

2.2 node2

This module extends node1 by adding a storage in each node and the possibility to add and retrieve values by means of their key.

To achieve this I created the storage module, there we can find add to add a key-value pair, lookup to retrieve a value by means of its key, split to separate a storage given a interval of keys, merge to append two storage.

In the node2 module we were given two new messages add and lookup. I had to implement the add function, this function adds a value to the storage but before it checks if the value belongs to the node with keys:between, in case it does we add it to the storage otherwise we say to the successor to take care of it.

lookup retrieves a value from the storage and it is quite similar to add, we check if we are in charge of that value if we are not we tell to the successor to lookup in its storage.

The hard part of this module was the handover function, what this function does is to give responsibility of keys to a new node that joined the network. The message was given I had to implement handover and update notify with some new features: it has to return a list of values that the new node has to take care of. We select this values with the handover function. handover uses storage:split to make two lists one stays in the node and it will be returned, the other will be sent with a handover message and will be merged with store in the called node.

2.3 node3

This module extends node2 and covers the first extra point: handling failures, each has to be monitored and we have to restore links to avoid interruptions, we need to keep track of the successor of the successor so in case of failure we know who will be out successor(Next variable), note that if two subsequent node crash our DHT will stop working, we can update Next with the status message. In stabilize we need to always return the successor and Next, we also have to take care of the case when a new node should be our successor, we need to demonitor the old node and start to monitoring the new one. In the rest of the program we add monitor during the connection phase when we receive back the reference, in notify after calling handover in both the nil case and the existing predecessor case. The drop function is called also in notify in case we had an old predecessor.

When we receive the DOWN message we handle it with a down function that cover the two cases of predecessor crash and successor crash, in the first case it just leave predecessor as nil while in the second case it starts monitoring Next, it calls stabilize and Next becomes the new successor.

3 Conclusions

This last homework was tricky not because of the Erlang code that was quite guided thanks to the skeleton but because it required a big effort to understand the protocol and how all the parts work, it was also very easy to make mistakes between variables because it was hard to keep track of the flow. Thanks to this assignment I really understood how a DHT works and I hope that this will help me for the exam.