Report 5: Chordy - A distributed hash table

Emil Ståhl

October 8, 2019

1 Introduction

This report covers the implementation of a distributed hash table (DHT) based on the Chord scheme. A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A DHT is simply a hash table that has been distributed on multiple nodes in a network. A node will store the values for all the keys for which it is responsible. Chord is one of the four original distributed hash table protocols, along with CAN, Tapestry, and Pastry.

In Chord, the nodes are arranged in a ring structure where each node has a successor who is the next node in the ring and a predecessor that is the previous node. The nodes also has an unique Id. Chord specifies how keys are assigned to nodes and how a node can retrieve a value given a key by locating the node responsible for that key. The main topic covered in this work is to get a deeper understanding of the problems that a DHT results in and how they can be solved.

2 Main problems and solutions

The work was split up into three different phases, each containing its own improvements.

- **node1.erl** introduces a basic ring structure that handles new incoming nodes.
- **node2.erl** implements a storage of values in each node and the possibility to add and lookup key-value pairs in the DHT.
- **node3.erl** implements failure handling.

2.1 node1.erl - Maintaining a ring structure

In this initial implementation of the DHT the main focus was to get a working ring structure up and running. The nodes Id's are generated by the `key.erl` which takes the easy way of just randomize a value between 1 and 1 000 000 000 to use as a Id, this is not optimal but works fine in this case. The key module also holds a function

`between(Key, From, To)` that checks if a key is in between two other keys in the ring. However, the main work is done in the `node1` module where the function `stabilize(Pred, Id, Successor)` is the most important one. It makes sure that a complete ring structure is maintained when a new node enters the ring. The node sends a request message to its successor to get the predecessor of its successor. Depending on the value of `Pred`, the node could suggest to be the new predecessor of the successor. The successor will receive the notify message of the suggestion from the node, call the notify function and decide if the node should be the new predecessor or not. The procedure `schedule_stabilize/1` is called when a node is created or every 100 ms.

In addition to this, to be able to check that we have a complete ring structure a probe message was implemented:

```
{probe, I, Nodes, Time}
```

The probe message is forwarded around the ring until it reaches the node that matches the Id, i.e. it have made a complete iteration. The probe time is then printed along with the visited nodes.

2.2 node2.erl - Storage, add and lookup

This is the module where the DHT actually becomes useful due to the newly added storage module. The storage is implemented as a list of tuples `{Key, Value}`, we can then use the key functions in the lists module to search for entries. A node will take care of all keys from (but not including) the identifier of its predecessor to (and including) the identifier of itself. If the node are not responsible it will simply send an add message to its successor. In addition to this, when a new node is added to the ring it should take over some of the responsibility of storing the key value pairs from its successor. This is handled by the function `handover(Id, Store, Nkey, Npid).`

However, in order to add new key value pairs to the DHT an add function was implemented in the `node2` module that checks if the node is responsible for the given key or if it should be forwarded to its successor, if it is responsible it simply calls the add function found in the storage module that puts the data in its list of tuples. A similar procedure is performed for the `lookup` function in order to retrieve data that is associated with a specific key.

2.2 node3.erl - Handling failures

In order to keep a fully connected ring structure at all times the ring must be able to repair itself if a node crashes. This is as usual done with the built in `monitor` function. If the current successor to a node crashes it must be able to update the successor to the one next in line, i.e. the successor of its successor. A new pointer to the successors successor is introduced and named Next. The `node` and `stabilize` functions are extended with an extra argument `Next`. However, if the predecessor of a node dies things are quite simple. There is no way to find the predecessor of a nodes predecessor but if the predecessor of the node is set to `nil` some node will sooner or later present them self as a possible predecessor. If the successor dies, things are almost as simple. The next-node will be adopted as the successor and then monitor the new successor as well as run the `stabilizing` procedure.

3 Evaluation

A test was performed where five nodes where created in a ring structure:

```
1> P1 = test:start(node3).

<0.80.0>

2> P2 = test:start(node3, P1).

<0.82.0>

3> P3 = test:start(node3, P2).

<0.84.0>

4> P4 = test:start(node3, P3).

<0.86.0>

5> P5 = test:start(node3, P1).

<0.88.0>
```

One node was then killed off.

```
6> test:kill(P3).

true
```

A probe message was then sent to check if the ring had repaired itself.

```
7> P2 ! probe.

Probe time: 30 micro seconds

Nodes: [501490715,311326755,945816365,723040206]
```

As stated, the DHT managed to detect the crash and update the ring accordingly with new predecessor and successor. However, there are many more things that can be done to improve the failure handling of the system. Because even if the ring quickly can repair itself the data that was stored in the crashed node is gone. A solution to this is to replicate the data on multiple nodes. However, this is an implementation that is not covered in this report.

4 Conclusions

This assignment was by far the hardest of them all yet the most interesting. To begin with it took a while to understand the concept of the chord scheme. However, when that was out of the way it was time to implement the key module with the `between` function which required a lot of thinking in order to compare the keys in a correct way. Furthermore, it was also hard to always think from an arbitrary nodes point of view with the predecessor and successor or what node should take care of what keys etc. Moreover, there was also a minor struggle of setting up and testing the failure handling. It took a while to figure out what was going on. The main topics learned was understanding the chord scheme and distributed hash tables in general.