Chordy - a distributed hash table - Report HW5

Malte Berg

October 8, 2025

1 Introduction

In this homework assignment, the primary topics covered were understanding the Chord protocol, as well as implementing a similar version of the protocol to create a distributed hash table.

2 Main problems and solutions

This task has by far been the hardest to implement. Reading through the paper and understanding it fully took a while, and the concept was hard to grasp, but is a very clever way to create a distributed hash table once it was understood more.

The stabilize/3 was quite hard to figure out what code was supposed to fill in the blanks of the given skeleton code. Mistakes were made here where some of the cases did not return back the successor as it should have if nothing was updated, making the program throw errors, and taking a while to understand what had gone wrong. It also took some trial and error to get the different cases to handle the correct situation, and to understand what each case meant.

The `request` part of the code was simple to implement, and just followed the instructions.

`notify` was implemented in a similar fashion to stabilize, but looking at the predecessor instead. After having had issues implementing the stabilize part, it was easier to understand how to implement notify.

Next, the functions `start`, `init`, and `connect` were created, following the instructions and filling in the gaps in the skeleton code. This took some testing and thinking, but got there in the end.

Trying to test before writing the probe handling proved to not be that giving, as the only thing that was noticed was that there were no errors when the code was correct, although it did throw some errors when trying to connect that were fixed.

The probe handling showed that the ring actually was created and that the message was sent the whole way around, but once again did not do it fully at first, which meant some more bug fixing.

The hardest, and at the same time easiest problem that was faced was when trying to create the nodes of the ring on different Erlang shells. Due to the fact that the key generator had not gotten a seeding function before creating the IDs, they gave the same ID when creating the nodes, meaning that the first node did not connect, but subsequent ones did. This took some real head scratching, and a lot of probing, to finally figure out, but seeding was implemented when creating a key to make the generating a bit more random.

Storage was quickly implemented, mostly using functions found in the lists library, where the most tricky part was handling the split correctly and which to keep and which to hand over.

However, a major issue that took the longest time to find and fix, due to no other fault than my own, was that the key:between function was not used in the split, but instead just checking if it was between From and To, which mean that if To ¡ From, it completely failed. This was a real pain to find, and when it was fixed, the program finally functioned just like it should, handing over keys when a new node was added with an ID that meant keys should be moved to the new node.

3 Evaluation

Clients	Nodes	Time (ms)
1	1	100
1	2	127
1	4	189
4	1	38
4	2	44
4	4	65

Table 1: 4000 keys lookup times, with varying amount of clients and nodes in the ring, in ms

As can be seen in both Table 1 and Table 2, there is significant improvement when looking up keys from multiple clients compared to just looking from one. There is a small increase in the time taken when more nodes are

Clients	Nodes	Time (ms)
1	1	292
1	2	288
1	4	344
4	1	103
4	2	126
4	4	136

Table 2: 10000 keys lookup times, with varying amount of clients and nodes in the ring, in ms

present, which seems logical as the nodes need to pass on the lookup if the key does not exist in a nodes store, meaning messages will be sent that will take some additional time. When 10000 keys are used instead of 4000, we see an increase of time taken, of course.

The times seen in the table are from connecting between different Erlang shells on the same machine, instead of on different machines, but is sending with the form of node, 'address' when doing lookups on nodes from different shells.

Testing was also done with 4 clients on 4 nodes, with 4000 keys, where each client does lookup on each separate node. This however did take 83 ms, which is greater than the 65 ms seen in Table 1.

This program could be extended greatly by implementing the bonus tasks given in the instructions, as the program as of right now is not implemented in a way where it can fix itself from a node drop. It can neither fix its successors or predecessors, nor can it recover keys that might be lost when a node drops. This is of course, not ideal, and implementing it would greatly increase the durability of the hash table.