Chordy – a distributed hash table

Óttar Guðmundsson
11 Oktober, 2017

1 Introduction

The final assignment was to implement a distributed has table by using the Chord protocol, described a paper provided. Chord can be described as a ring of nodes that all have a random id to lookup other nodes and store values. By reading through the paper and following given instructions, the plan is to gain the understanding of the protocol, how distributed hash tables work and its implementation. The paper did also describe other complex entities like a finger table, which we did not implement.

2 Main problems and solutions

Key

The first task was to implement a simple module called key which had two functions. One of them was to generate an id for each node and also an id for a key value, a random number of 1.000.000.000 and the second one checked if a given key was between two values.

Node 1

In the first implementation, nodes were structured in a ring where each node held a pointer to either its predecessor or its successor. In a custom made heartbeat of each node (timed interval) each node would send a request message to its successor to call for the function *stabilize* which was called to keep track of the pointers. The function would send the requestor his own predecessor so it could then check out the sent predecessor to see where it was currently positioned. Now we had to finish a long case command. If the predecessor was simply nil then the node should just set itself as its own predecessor with a notify message sent to its successor. The same thing could be done if it was already pointing to its successor. If it was pointing to itself, we just return our successor. The tricky part was what to do if we had another node, the predecessor of our successor, than ourselves or our successor. If the other node was between us and our successor, we send a request for stabilization to the other node with our own id which repeats the whole process. If not, we are between the nodes and notify our successor about it.

Whenever the node received suggestion regarding a new predecessor it would have to check it before positioning it in the ring. We adapted the node if it didn't have any predecessor before but if it had a predecessor before, we had to check if its id would be in between the current predecessor and itself and if so it will add it as a predecessor. There is no reason to inform the new node about the current decision. On a timed interval, it will call for stabilization where it will discover its successor by itself. To test this out, we were given a simply function to test out the ring and see if it would work. After a couple of tries and a lot of refactoring, a ring was assembled! Connecting usually took around 1ms.

Storage

Now since the nodes could be connected to a ring it was time to make a use of it. All nodes would now hold a store with a value (not sure why every value should be a "gurka") and a key reference to it. Every time we needed to add a value to the ring or look it up for later usage, we could simply check if the key was between our predecessor's id and the current node we were positioned in. If so, the key and its value would be stored / found in that node. If not, the key request was sent to its successor.

I had some problems finishing this part since I had written my between function incorrectly. After debugging for some time I found out where the problem was and I could store and forward requests correctly. Another tricky thing to implement was the handover part when a new node entered the ring. The new node would receive all of the keys that were lower than the new predecessor key. That was done by splitting up the key values where the fit into the between function.

Node 2

Adding the storage to the node 2 was also tricky, don't want to say how long I was to get that working. It introduced a new variable for the nodes to take care of, the storage itself, and new messages and functions to use. Both the Add and Lookup were troublesome to implement. After finally getting them right, adding list of keys and values through the given test module to a ring finally worked as expected.

Node3 (Optional task)

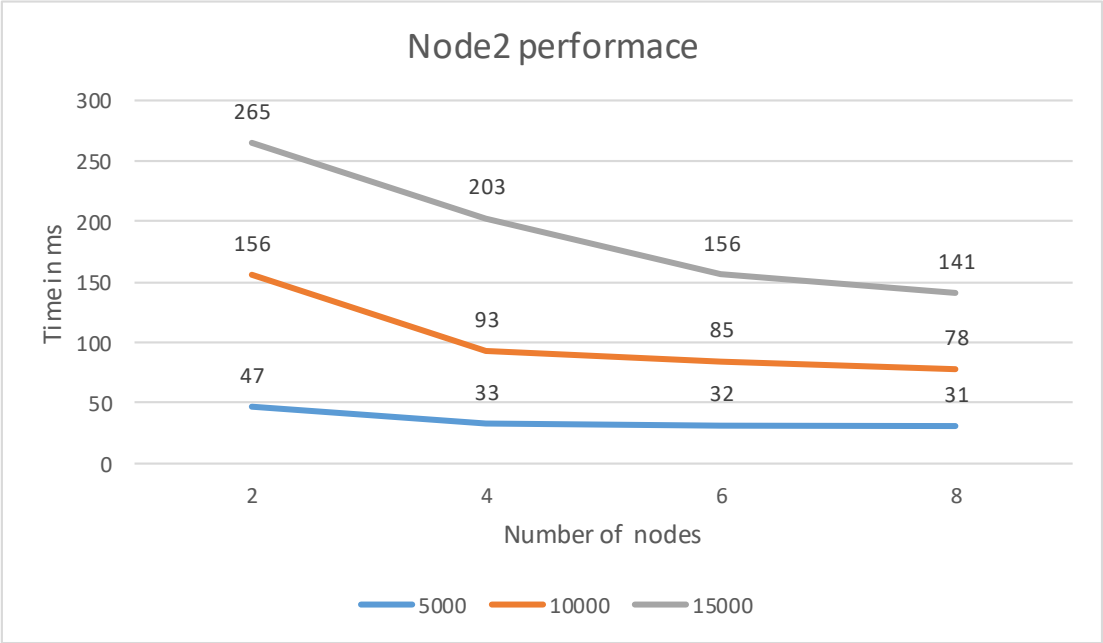
I must admit, I was not really excited to do the optional part since I had just got rid of the problems of node1 and node2. But so far I've done all of the optional parts, why stop here?

A new variable was introduced to the optional task, *Next*, which simply pointed to the successor of the nodes successor. This required a lot of change in the code. Firstly, the stabilize function now had to return the successor and its own successor in a tuple message. Thus the next successor should be set as the previous one before adapting a new one. We used the inbuilt erlang monitor function to keep track of nodes. We had to set the monitor function every time a node changed its predecessor or its successor. That meant also demonitor the previous ones we were currently monitoring. If we received the classic 'down' message we had to implement a pattern match to find out if it was coming from our predecessor or our successor. In the former case, all we had to do was to remove the reference by setting it as nil which the stabilize function should then take care of in the next beat. If it was the latter case, simply set the next as our successor. This took me way more time than I anticipated, because my pattern matching was incorrect since the Next Successor that was returned from the *stabilize* function always included the reference. After removing it, it finally worked.

There was another optional task to replicate the data when a node was disconnected. I did not implement that part since I had already put so much time into this project. Thus, I will accept the loss of data in my ring. However, the first idea on how to handle things is that every node also keeps and extra storage in their successor. That way we will always have a backup if only one node dies. Now if the predecessor would die we could merge the data in the current node with the replica and if we detected that our successor died we simply send the replica storage to the new successor and replicate it there. This also needs to be handled in the handover function when a new node joins the ring, which does require a lot of extra refactoring

3 Evaluation

Trying out the performance was a little bit problematic for me. I had access to two different computers at my house but I had trouble connecting them together since my own laptop has some corporate restrictions regarding the firewall. Thus I only tested it on my own machine.



I did several tests with different amount of nodes and different amounts of items in the list. Here it is clear that by adding more nodes to the ring the performance was getting better. It seemed like searching for the key values in a single, long list with the built in *keyfind* takes a lot of time while it is faster to send messages around. I guess these findings may vary based on the network of computers with limited transfer rates, jitter and processing power.

4 Conclusions

After being one week ahead of all projects in the course, I'm happy I had two weeks before this one. I spent way more time on it than I thought it would cost me since it was really tough and confusing on parts. I've never heard or used distributed hash tables before so it was nice to learn about what it was and what it was capable of. Seeing the speed increase in the search was also an experience. Nevertheless, I would never use my own implementation since data might be lost and I'm not sure how to fix other faulty problems of the nodes.

5 Data from table

	2	**4**	**6**	**8**
5000	47	33	32	31
10000	156	93	85	78
15000	265	203	156	141