Chordy: a distributed hash table

Joakim Öberg

October 8, 2020

1 Introduction

Distributed hash tables have four original protocols; Chord, Tapestry, Pastry and CAN. In this assignment we implemented a distributed hash table (DHT) following the Chord protocol.

The first implementation **node1** will only maintain a ring structure for holding nodes, but we can not add any data elements to it. In the second implementation **node2** we add a storage to the nodes, letting the nodes to keep key value pairs of data. The implementation will also be able to update the ring and divide up the keys between nodes when more nodes are added to the circle. The files in this assignment are *test.erl*, *storage.erl*, *node1.erl*, *node2.erl* and *key.erl*

2 Main problems and solutions

2.1 key

The first task was to implement the **key** module with function generate/0 and between/3. A problem that was found is that the Erlang random:uniform/1 does not really give fully randomized values. When running it on two different machines the output is exactly the same.

The other function, between/3 was one of the fundamental functions of the program and was really hard to implement, no help in code was given only some small hints in text.

2.2 node1, keep the circle intact

The stabilize function is also a really critical function for the program, it is the function that keeps the circle of nodes intact. In init the function schedule_stabilize/0 is called that send a scheduled stabilize message to our own PID(node). When a node receives a stabilize message it checks what the node in front of it think have behind it, in java we would say that we look at the next.prev. Depending on what next.prev is we take different actions, keeping the circle intact.

Another function that is needed for the circle to stay intact is notify/3 that notifies the node in front of it that it is its new predecessor.

2.3 node2, adding a store

To be able to handle elements we add an argument store that can keep key value pairs. For it to function we implemented a new module **storage** that have the functions create/0, add/3, lookup/2, split/3 and merge/2. In node2 we also added the argument store. When a new node is added we do the notify as before but now we also have a function call **handover/4** that look at the key values, for each key that we have in our store if a key value that our node holds is closer to the new nodes id, then it going to be held by the new node instead.

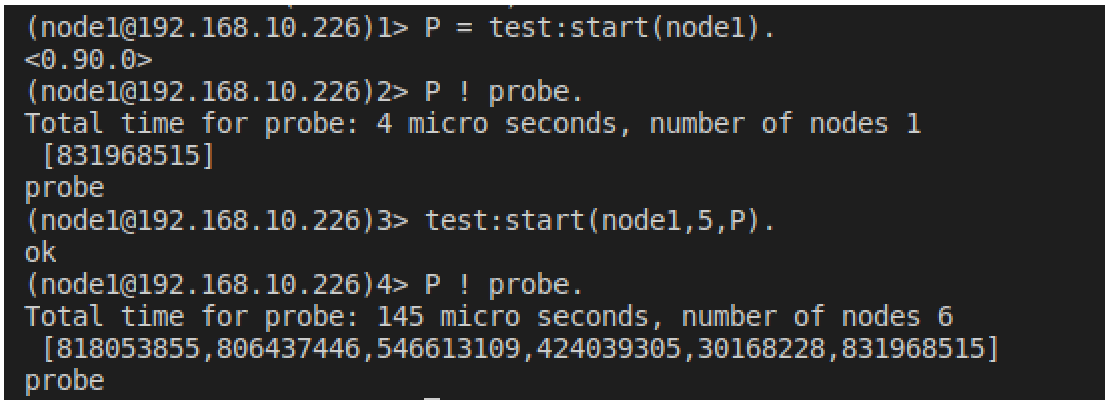
I also added a print out in the handover function so that every time the function is called, the printout tells which node it was, how many key it stored before and how many keys it have now. Sometimes the prints can be misleading because if all keys is still closest to the same node, it will not update the storage. I also added a "status" message which I can use to check how many keys is held in that specific storage.

3 Evaluation

To test **node1** and **node2** a test program *test.erl* was used, from which functions to start nodes, create N random keys, adding keys, do lookups and measure time to iterate nodes could be executed.

3.1 node1, keep the circle intact

To test that node1 fully worked I started one node, added some nodes to it and then sent a probe message to the first node to see that they where added in the correct way, the execution can be seen in figure 1.



```
(node1@192.168.10.226)1> P = test:start(node1).
<0.90.0>
(node1@192.168.10.226)2> P ! probe.
Total time for probe: 4 micro seconds, number of nodes 1
 [831968515]
probe
(node1@192.168.10.226)3> test:start(node1,5,P).
ok
(node1@192.168.10.226)4> P ! probe.
Total time for probe: 145 micro seconds, number of nodes 6
 [818053855,806437446,546613109,424039305,30168228,831968515]
probe
```

Figure 1: Test that node1 builds the circular list correctly

3.2 node2, adding a store

What we want with this solution is that the iteration through the nodes should be insignificant and what takes time is to look for the key in the storage. If we have a lot of data with very few keys then the performance can become better by increasing the number of nodes. But if we add to many nodes than the time to find the right node could take almost the same amount of time that it takes to then find the key and the advantage of the greater amount of nodes is lost, this can we see in table 1.

# Nodes	Time(ms)	# Lookups per ms
1	35	114
4	26	153
8	29	137

Table 1: Test performed with 4000 keys, one terminal

3.3 What we can improve

One of the biggest problem with this implementation is that failures can not be handled, if a node goes down the circle is no longer complete and we have no idea that it is broken. If a node goes down some node should also have a copy of that storage so that the circle can be kept intact without data getting lost. We can also implement so that one node keep more than just one other nodes copy of storage. How many copies that is suitable for the implementation depends on the application, the number of total nodes and how often we think nodes will go down. To implement this is a bit tricky because to be sure that no data will get lost we need to be sure that our successor knows what data we have if our predecessor goes down and our storage is updated, otherwise when we go down if the successor does not know that we had a new storage it will think we only had the older one with fewer elements.

4 Conclusions

In this assignment a distributed hash table was implemented following the Chord protocol. The assignment was very interesting because the implementation of node1 is basically a circular double linked list, something I have heard is hard but possible to implement in Erlang. In node1 we implemented the circular list, in node2 we also added the functionality to hold key value pairs. Node2 also distributes the added keys by consistent hashing, meaning that a key with value closest to some nodes Id will be stored under that node. The biggest problems in this implementation is that we can not handle cases when nodes goes down and the circle is broken.