HW Report 5: Chordy – A Distributed Hash Table

Ayushman Khazanchi

October 13th, 2021

1 Introduction

The aim of this homework assignment was to create a simple distributed hash table based closely on the Chord scheme. Initially we only create and maintain the ring structure of the hash table but eventually we also add functionality to store key-value pairs.

2 Main Problems and Solutions

2.1 Building the Ring

We start out by building the ring structure based on each node identifying its relative position in the hash table. Our nodes keep a track of their successor as well as their predecessor. We implement this with the use of an identifying "key" that we assign to each node. For the sake of simplicity we will use integers as our keys.

When we want to add a new node we use our *key* module to do a lookup of our position. The new node's key is checked to see whether it falls between the predecessor and successor of the node being informed. If the key falls between the two, we expect the new node to become the predecessor of the node being informed. If the key doesn't fall between the predecessor and successor the notified node simply diverts the new node to the next successor. This logic is implemented by the *key:between/3* function as well as the *stabilize/3* function in node2.erl.

With the help of stabilize and notify functions, as the ring grows, we are able to stabilize the structure and have it in the right order. In the case that a node does not get added in a particular location it simply moves on to the next node until it finds its right spot.

Below you can see the nodes getting added successfully. Further down with the help of the probe output we will see that they are in the right order.

```
9> node1:start(1).
<0.105.0>
10> node1:start(2, <0.105.0>).
<0.107.0>
```

```
11> node1:start(3, <0.107.0>).
<0.109.0>
12> node1:start(4, <0.105.0>).
<0.111.0>
```

We also add a probe in order to check that our ring is actually forming correctly. The probe function helps us traverse our whole ring from where it's probed until it reaches back to the same node. Below you see the probe being invoked on the PID of key 1 and key 4 and it returns the whole ring.

```
13> <0.111.0> ! probe.

 Time =22 --- Nodes = [4,1,2,3]
probe

14> <0.105.0> ! probe.

 Time =23 --- Nodes = [1,2,3,4]
probe
```

2.2 Adding a Store

Implementing a store requires adding a few more things to our implementation. First we need our nodes to have a reference of a local store of their own. For our purposes we use a list of tuples as our "Store" and when our nodes are initialized we start them with empty stores. When a new node is added we have to decide how they will not only join the ring structure but also split some part of the key-value store and take over responsibility for it. On the other hand, the node that is giving up a part of the store has to decide how to handover the store and which part of the store it has to hand over. This is handled by our *handover* function shown below.

```
handover(Store, Id, Nkey, Npid) ->
   {Leave, Keep} = storage:split(Id, Nkey, Store),
   Npid ! {handover, Leave},
   Keep.
```

Below you can see an implementation of the nodes getting added with the node2.erl file which also handles the store implementation in this ring structure.

```
2> node2:start(1).
<0.87.0>
3> node2:start(2, <0.87.0>).
<0.89.0>
4> node2:start(5, <0.87.0>).
<0.91.0>
5> node2:start(10, <0.89.0>).
<0.93.0>
6> node2:start(20, <0.93.0>).
<0.95.0>
7> node2:start(24, <0.93.0>).
<0.97.0>
8> <0.97.0> ! probe.
24 probe
1 2 5 10 20
 Time = 319
9> <0.87.0> ! probe.
1 probe
2 5 10 20 24
 Time = 2810
```

Additionally we also have to have functions for adding and looking up data in our new key value store. The add function simply does a look up of where the data needs to be stored similar to how we add nodes to our ring. If the notified node is responsible for the data, it will take responsibility, else pass the request on to its successor. The same principal works for the lookup function as well.

```
add(Key, Value, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store) ->
    % if client key is between pred key and our key (id) then we take care of request else we pass to successor
    case key:between(Key, Pkey, Id) of
        true ->
            Client ! {Qref, ok},
            Added = storage:add(Key, Value, Store),
            Added;
        false ->
            Spid ! {add, Key, Value, Qref, Client},
            Store
    end.
```

With the add and lookup function in place we can see an example of the data being added to the ring structure and then a successful lookup taking place on the same ring.

```
4> node2:start(1).
<0.96.0>
5> node2:start(2, <0.96.0>).
<0.98.0>
6> node2:start(5, <0.96.0>).
<0.100.0>
7> node2:start(10, <0.96.0>).
<0.102.0>
8> node2:start(15, <0.96.0>).
<0.104.0>
9> <0.96.0> ! probe.
1 probe
2 5 10 15
 Time = 2510>
10> test:add(12, "hello", <0.96.0>).
ok
11> test:lookup(12, <0.96.0>).
{12,"hello"}
```

3 Evaluation

So we can see our DHT ring works quite well with the key value store as well. After doing some rough time measurements I found the time to be increasing mostly linearly. I would think this would affect the implementation of the same method over a large number of nodes although I did not get to try this. Additionally, I also tested adding data rapidly after creating nodes in order to simulate data being added before the ring is stabilized and I ran into a few crashes so the nodes are not currently fault-tolerant.

4 Conclusions

I found this assignment quite interesting in terms of how simplified it is and helpful in understanding the principles of distributed hash tables. However, I found it quite confusing that some of the parameter names and function calls are not explained well and therefore very hard to follow (for example, introducing the parameter Client suddenly without much context).