Homework 5 Report

Rafat Khan

October 13, 2021

1 Introduction

The purpose of this homework is to understanding distributed hash tables (DHT), a structure used to store data among different nodes in a distributed environment. Our prototype will implement methods to add a new node, insert some key/value data, and get the value for a stored key. Our DHT will be designed as a ring, each node keeping reference of its predecessor and successor. In order to facilitate routing, we want to keep the nodes ordered.

2 Main problems and solutions

First, we have only implemented the DHT ring without the ability of storing and retrieving data. Considering the stabilize operation, each node sends a request message to its successor telling it that it wants to know its predecessor. One specific case is when there is only one node in the ring : at the beginning, the successor of the single node is itself and it has no predecessor. Then, it will begin the stabilizing procedure: it sends a request message to its successor (itself), and then it answers to itself that it has no predecessor, therefore it sends to itself to take itself as a predecessor (notifying). Thus, when there is only one node in the system, it has itself as predecessor and successor (in the code, Xkey is equal to Skey).

Moreover, the frequency of this stabilizing operation is every 100 ms for each node. The pros of a frequent stabilizing procedure are that the system sees very rapidly if another node has entered the system (or if a node has crashed), and then it is able to very rapidly re-build a correct ring. The cons are obviously that it induces a network traffic overload. On the other hand, if there is no stabilizing procedure, the ring can not be re-build after additions or deletions of nodes, so it is important to find the right frequency for this operation.

Secondly, we have added a storage mechanism to the DHT. The tricky part is to correctly split the data store when a new node is added to the system.

3 Evaluation

3.1 Ring

The properties we want for our ring is that the key of any given node is between the keys of its neighbours. And all we know of initialisation is that all the new nodes (except for a center, if any), have their successor set to another node somehow himself linked to the Ring. In order to converge toward the structure we want, we will go for the easiest solution : a node that looks for a place in the ring will check weather it's at the right position already, if yes insert itself, if not move to the next one. This part was actually most of the assignment. The main two methods are stabilize/3, used to examine the Predecessor of our Successor, and determine if we need to take its place, and notify/3, which determines weather a change as to be made when we're told so by another node.

```
Returning the updated successor:

stabilize(Pred, Id, Successor) ->
    {Skey, Spid} = Successor,
    case Pred of
        nil ->
            Spid ! {notify, {Id, self()}},
            Successor;
        {Id, _} ->
            Successor;
        {Skey, _} ->
            Spid ! {notify, {Id, self()}},
            Successor;
        {Xkey, Xpid} ->
            case key:between(Id, Xkey, Skey) of (Xkey)
                false ->
                    Xpid ! {request, self()},
                    ~n",[Id, Skey, Xkey]),
                    Pred;
                true ->
                    Spid ! {notify, {Id, self()}},
                    )~n",[Id, Skey, Xkey]),
                    Successor
            end
    end.
```

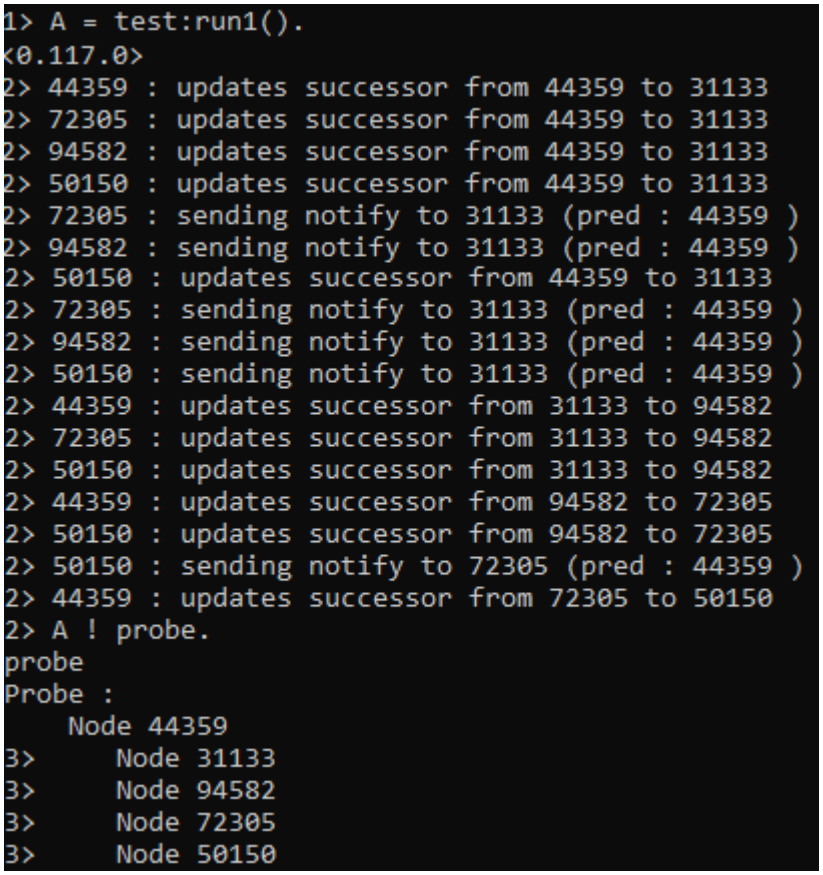

Figure 1: Creating a ring network by updating successors.

Returning the updated predecessor:

```
notify({Nkey, Npid}, Id, Predecessor, Store) ->
    case Predecessor of
        nil ->
            Keep = handover(Store, Nkey, Npid),
            {{Nkey, Npid}, Keep};
        {Pkey, _} ->
            case key:between(Nkey, Pkey, Id) of
                true ->
                    Keep = handover(Store, Nkey, Npid),
                    {{Nkey, Npid}, Keep};
                false ->
                    {Predecessor, Store}
            end
    end.
```

3.2 Storage

Now that our ring is up and running, we will use it to store data. Our storage will be naively implemented as a pair list of key and value. We will keep it sorted at any time so the split operation is easier. Now to actually add a storage to each node, we need to update our method's signatures to take a Store as extra argument and return also a Store.

We will create functions to add, lookup and handover the Store:

```
add(Key, Value, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store) ->
    case key:between(Key, Pkey, Id) of
        true -> Client ! {Qref, ok} ,
                storage:add(Key, Value, Store);
        false -> Spid ! {add, Key, Value, Qref, Client},
                Store
        end.

lookup(Key, Qref, Client, Id, {Pkey, _}, Successor, Store) ->
    case key:between(Key, Pkey, Id) of
        true -> Result = storage:lookup(Key, Store),
                Client ! {Qref, Result};
        false -> {_, Spid} = Successor,
                Spid ! {lookup, Key, Qref, Client}
        end.

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

Also, the notify procedure has to be updated to execute the handover.

Now we will verify that the handover is distributing data while a new node join.

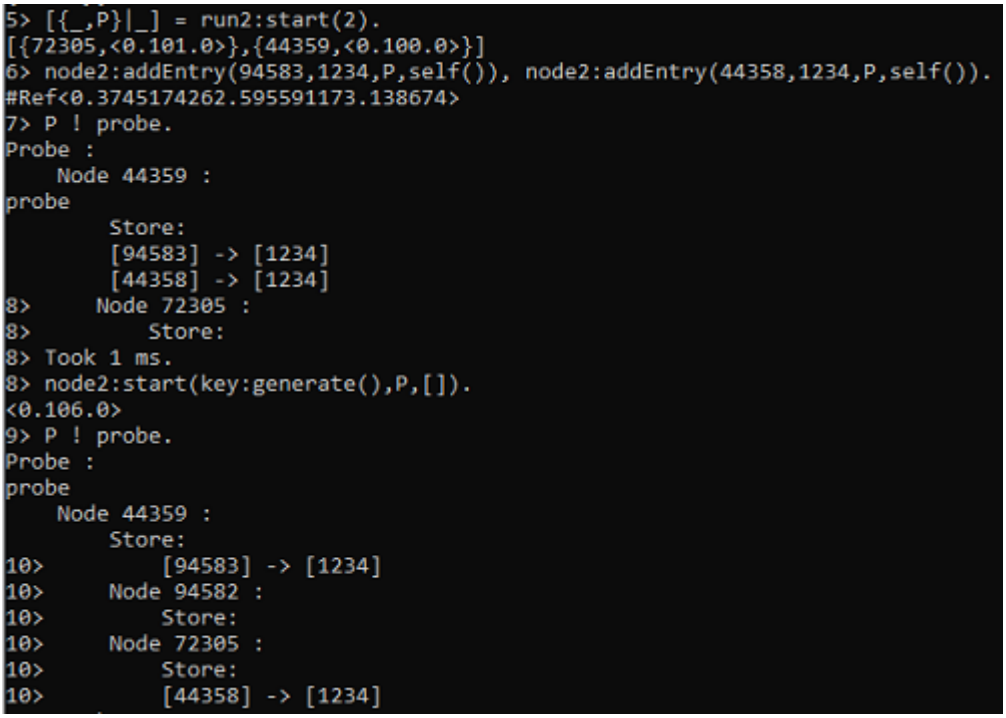

Figure 2: Data handover after joining a new node.

3.3 Handling failure

The erlang moniter finds if the process is down and call the down functions which creates a new successor. When the successor dies, the next node becomes the new successor.

```
{'DOWN',Ref,process,_,_}->
        {Pred, Succ, Nxt} = down(Ref, Predecessor, Successor, Next),
        node(Id, Pred, Succ, Nxt, Store);

down(Ref, Predecessor, {_, Ref, _}, {Nkey,_, Npid}) ->
        self()!stabilize,
        Nref = monitor(Npid),
        {Predecessor, {Nkey, Nref, Npid}, nil}.
```

4 Conclusions

In conclusion, this assignment is based on chord but much more simpler than chord. The core of the assignment is to dealing the new node and dead node. The concept is a bit hard to understand, but during this seminar I have understood the basics of the P2P,Chord and DHT.