Seminar Report : Chordy

Vincent Delaunay

October 22, 2014

1 Introduction

Chordy is a distributed hash table in Erlang.

This seminar aims to offer a grasp on distributed hash tables, a structure used to store data among severl nodes in a distributed fashion. 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 precedessor and successor. In order to facilitate routing, we want to keep the nodes ördered". The implementation starts with implementing methods for a Key that will identifie our nodes and allow to sort them. Next, we will implement nodes so that they can negociate and organize themselves into our ring. Then we will build the storage on top of this ring. Finally, we will try and discuss the performances of our distributed hash table.

2 Key

When we add a new node, we want to insert it in its r̈ight position", so we can route easily. This is achieved by giving a numerical id to each node and ordering them.

Aside from key:generate/0, we need a method key:between/3 :

```
% From == To -> complete range
between(_, From, From) ->
    true;
% From < To --> between is the inner range
between(Key, From, To) when From < To ->
    ((From < Key) and (Key < To));
% From > To --> between is the outer range
between(Key, From, To) when From > To->
    ((From < Key) or (Key < To)).
```

Instead of wasting time thinking about off-by-one errors, let's use a test-driven approach to this problem. Using the same minimalistic test framework as in other seminars :

```
"key:between 20 in 10,30 ?" : pass
"key:between 20 not in 30,10 ?" : pass
%% ... 8 more edge / non-edge cases
```

3 Ring

The propertie 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 (execpt 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 wether 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. We are given a skelettic implementation of a node, which we need to complete. 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, wich determines wether a change as to be made when we're told so by another node.

```
% Returns 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
                            true -> Spid ! {notify, {Id, self()}},
                                    Successor;
                            false ->  Xpid ! {request, self()},
                                      Pred
                        end
    end.

% Returns the updated predecessor.
notify({Nkey, Npid}, Id, Predecessor) ->
    case Predecessor of
        nil ->  {Nkey, Npid};
        {Pkey, _} ->
              case key:between(Nkey, Pkey, Id) of
                true ->
                    {Nkey, Npid};
```

```
                false ->
                    Predecessor
            end
    end.
```

As a side note, I spent most of my time being confused about argument orders. I find some method signatures we are given to be really counter-intuitive (mostly node2:stabilize/3, and the calls to key:between/3)..

All we have left to do is to start and link several nodes. I experimented with it with two methods : run2:sartStar and run2:startLine. They initiate the network of nodes in different layouts, a star or a line. More on this later.

We can follow the negociations thanks to some prints now and there, and view the final state using the mentioned probe message, that I implemented but did not present here.

```
% Starting 4 nodes.
% The key range has been altered for this example so it is easier to figure out.
2> A = tests:run1().
45 : updates successor from 45 to 95
%% ... a dozen updates...
45 : updates successor from 73 to 51
3> A ! probe.
Probe :
    Node 45
    Node 95
    Node 73
    Node 51
Done in: 1654 ms
```

4 Short observations

I measured the time taken by the probe with 30, 300 nodes, and it was linear as expected.

Testing the influence of the starting network layout was interesting. While performances were the same for a small number of nodes, differences start to appear with bigger networks. I did not measure the time needed to converge but the number of notify messages. Non-rigorous, non-statistically fiable numbers below :

	5	30	300	(nodes)
Line	50	250	23000	(notifies)
Star	50	250	5000	(notifies)

We can see here that the star design is clearly winning. It seems to be linear where the line one seems at least quadratic.

5 Store

Now that we have our ring up and running, we want to use it to store data.

Our store will be naively implemented as a list of key,value. We will want to keep it sorted at any time so the split operation is easier. All of this is trivial, really the s̈toragem̈odule is just used an interface of the l̈istsm̈odule.

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.

The actual work comes into play when it comes 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.

6 Evaluation

How does our DHT perform ?.

First, let's see how it behaves when in the best conditions. By b̈est conditions̈, I mean the ring is stabilized before we start adding to it (we wait, and we have very few nodes). More on this later.

So, I have implemented some test methods, wich look roughtly as follow (slightly edited) :

```
run2() ->
    Nodes = run2:start(5),
    % select a node as our entry point for the rest of the tests
    [{_,Pid} |_] = Nodes,
    timer:apply_after(6000, tests, test_one_item, [Pid]),
    timer:send_after(7000, Pid, probe),
    timer:apply_after(5000, tests, test_a_list_of_items, [1000,Pid]),
    Nodes.

test_one_item(Npid) ->
    addEntry(test, 12345, Npid),  Entry = getEntry(test, Npid),
    test("test_one_item : ",Entry=={test,12345}).

test_a_list_of_items(N,Npid) ->
    % Generate a list of random number
    Keys = [random:uniform(1000000) || _ <- lists:seq(1, N)],
    % Add entries of the form {N,N}.
    lists:foreach( fun(I) -> addEntry(I,I,Npid) end, Keys),
    % Lookup the value for every key.
    Values = lists:map( fun(I) -> {_,V} = getEntry(I,Npid), V end, Keys ),
    % Assert.
    test("test_a_list_of_items : ",Keys==Values),
    ok.
```

Since the tests pass, it looks that our DHT can handle a small amount of key/value pairs (1000 here). That's it already.

Then, we will simply verify that the handover works well in a no-stress situation. Here is a manipulation from the Erlang console (thanks to not seeding the RNG, I already knew what to insert to make the demonstration):

```
1> [{_,P}|_] = run2:start(2).
2> node2:addEntry(94583,1234,P,self()), node2:addEntry(44358,1234,P,self()).
3> P ! probe.
    Node 44359 :
        [44358] -> [1234]
        [94583] -> [1234]
    Node 94582 :
4> node2:start(key:generate(),P,[]).
5> P ! probe.
    Node 44359 :
        [44358] -> [1234]
    Node 31133 :            %% New node
        [94583] -> [1234]
    Node 94582 :
```

Finally, what is left to test is the behavior when the ring is expanding : adding an entry just after starting stabilization over a large number of nodes, and/or adding a new node just after starting a massive addition of entries. Sadly, due to the code having absolutely no protection, nodes will currently crash when they have to send a message over a nil link. This could of course be fixed, but I didn't look into it. We can also work around crashes and make the DHT stand, and this is the purpose of part 3 of this seminar.

7 Conclusions

Regarding performance, the biggest issue faced by chordy is the repartition of the entries. There is actually no mechanism to make it remotely uniform. Almost the same issue, but once such a l̈oad balancerm̈echanism is implemented, we will want it to be able to take into consideration newcomers.

This seminar was interesting. Distributed storage / file system / history is a field that have interested me for some times, as they are plenty of neat features to look after : integrity (git, bitcoin), availability (AFS ? is there nothing new here ?), confidentiality, anonimity & deniability (freenet), least authority (tahoe), ... a lot of thought have already been put into it, and a lot more will again.