# Report 5: Chordy, a distributed hash table

Pierre Fleitz

October 13, 2015

## 1 Introduction

*Chordy, a distributed hash table in Erlang.*

The main goal of this seminar was to implement a peer-to-peer Distributed Hash Table (DHT) following the Chord protocol. A DHT is a structure used to store data among several nodes in a distributed fashion. During this seminar we 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 refrence of its predecessor and successor. This report will be divided into several parts. First we will talk about the implementation of methods for a Key that will identifie our nodes and allow to sort them. Secondly we will talk about the implementation of nodes so that they can negociate and organize themselves into our ring. Then we will talk about the build storage on top of this ring. And finally, we will discuss the performances of our distributed hash table.

## 2 Main problems and solutions

### 2.1 Key

When we add a new node, we want to insert it in its right position, so we can route easily. In order to do this we will give a numerical ID to each node and ordering them. We do need a key:generate/0 method, but also a methode key:between/3 :

```
between(Key, From, To) ->
  if
    From<To ->
      (Key>From) and (Key=<To);
    From>To ->
      (Key>From) or (Key=<To);
    true ->
      true
  end.
```

### 2.2 Node

This part is the part that took me most of the time I spend on this seminar. Just like previous seminars, we have a skelettic implementation of a node and we need to complete it.

There is two main methods : **stabilize/3** (used to examine the Predecessor of our Successor, in order to know if we need to take its place) and **notify/3** (used to determine if a change as to be made when another node says that we need to).

**stabilize/3 :**

```
stabilize(Pred, Id, Successor) ->
  {Skey, Spid} = Successor,
  case Pred of
    nil -> %% We should inform it about our existence
      Spid ! {notify, {Id, self()}},
      Successor;
    {Id, _} -> %% Pointing back to us => Nothing to do
      Successor;
    {Skey, _} -> %% Pointing to itself => Should notify it about our existence
      Spid ! {notify, {Id, self()}},
      Successor;
    {Xkey, Xpid} -> %% Pointing to another node
      case key:between(Xkey, Id, Skey) of
        true -> %% The key of the predecessor of our succesor (Xkey) is between us a
          Xpid ! {request, self()},
          Pred;
        false -> %% Case where we should be in between the nodes => we inform our su
          Spid ! {notify, {Id, self()}},
          Successor
      end
  end.
```

**notify/3 :**

```
notify({Nkey, Npid}, Id, Predecessor) ->
  case Predecessor of
    nil -> %% Our own predecessor is set to the nil, case closed
      {Nkey, Npid};
    {Pkey, _} -> %% We already have a predecessor :
      case key:between(Nkey, Pkey, Id) of
        true -> %% => We have to check if the new node actually should be our prede
          {Nkey, Npid};
        false -> %% => Or not.
          Predecessor
      end
  end.
```

*What are the pros and cons of a frequent stabilizing procedure ?* 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 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.

## 2.3 Store

At this moment of the seminar, we have our ring working, and we now want to use it to store data. Our store is a simple implementation of a list of key,value. As we want it to be sorted at any time, the **split** operation is not complicated. This part of the seminar was not that hard it was all about using efficiencly the list module.

**storage:lookup/2 and storage:split/3 :**

```
lookup(Key,Store) ->
  lists:keyfind(Key, 1, Store).

split(From, To, Store) ->
  {Updated, Rest} = lists:foldl(
    fun({Key,Value},{AccSplit1,AccSplit2}) ->

      case key:between(Key, To, From) of
          true ->
            {[{Key,Value}|AccSplit1],AccSplit2};
          false ->
            {AccSplit1,[{Key,Value}|AccSplit2]}
      end

    end, {[],[]}, Store).
```

Finally after implemented the storage module we needed to improve the node1 module into a node2 module, especially adding 3 important methods : **add/8, lookup/7, handover/4** :

**add/8 :**

```
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/7 :**

```
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 ! {loojup, Key, Qref, Client}
  end.
```

**handover/4 :**

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

## 2.4 Performance :

*In this part we were supposed to test the performance of our DHT.*
In order to do so, I tried to implement a test module where I can launch either 1 or 4 nodes (with randomly generated keys) and launch either 1 or 4 test machines.
If there is only one test machine, this machine adds 4000 randomly generated keys, and then do a lookup for all of these 4000 keys. If there are 4 test machines, each of these adds 1000 keys and then do a lookup for all of these 1000 keys (all the test machines run concurrently). In my tests, there is no network latency, so these tests may differ from real distributed tests. We will measure the lookup time for each configuration. In the case of multiple test machines, the lookup times of all machines are added together.

*What can I conclude about my tests ?*
What I have been able to see is that the distributed nature of the DHT plays a real important role (in a performance point of view) when the clients act concurrently on the system. But when they are all on the same node, then it changes nothing.

This sounds logic to me.. But during the first seminar I encountered some problems during some tests using **erlang:now()** and **erlang:now diff()**, the results were really incoherent. Here what I obtain looks quite logic.. But again I'm not sure : firstly about my test procedure because I didn't had the time to spend much time on it, and secondly about what my terminal is displaying...

## 3 Conclusions

This seminar really helped to understand better the basics of the P2P Chord DHT. The skelettic code given helped me a lot again and I never felt lost during this seminar wich is quite good. I'm a little bit skeptical about my test procedure and the results I obtained because of previous problems, but it seems quite logic at then end and I really look forward to see and test more all of this during the seminar session.