# Report 5: Chordy Seminar Report

Alberto Lorente Leal

5 October 2011

## 1 Introduction

In this seminar, we implement a distributed hash table based in the documentation of a paper. In this case it is based in the chord scheme, which defines a protocol the creation and storage of keys in a ring node distributed structure. It tries to simplify the design of peer to peer systems trying to address a series of problems which we will not get into much detail about them. In this case the assignment guides us on how to implement this system in Erlang using the core of the Chord protocol but in which some simplifications are introduced to make life easier. We will implement basicly the how nodes are defined and structured in the ring, and also introduce a storage system to simulate how the keys are shared between the nodes.

## 2 Task:

The main task we had to implement chordy until sections 1-2, so the distributed hash table should be able to define the ring with the nodes and also introduce a storage system.

### 2.1 First Implementation:

The first implementation of chordy should be able to grow the ring while we add new nodes into the system. Before that in order to help us on how to arrange the nodes in the ring, we implemented the key module which contained a function named between(Key,From,To) which will tell us if a key is in the range (From,To].We would like to remark that we need to treat carefully this function because, the ring will probably close at some point and we could have a key with value 90 having a successor with value 5. In that case we should check carefully if the condition is met. Also we introduced a function to generate a random key between 1-10000000.

```
between(Key, From, To)->
case From==To of
        true->
```

```
                  true;
         false->
                  if From>To ->
                  ((From<Key)and(Key=<1000000000))or((0<Key)and(Key=<To));
                  To>From ->
                  (From<Key)and(Key<To)
                  end
end.
```

The next part was to implement the stabilize/3 procedure which will be in charge of making the ring stable when it knows the predecessor and successor in order to notify the nodes that need to be updated with their successors. The first 3 cases we simply notified the node and returned back our original successor. In the last case we needed to check if the key is between us and our successor or not.If true we update our successor and send a request for the node to take us as his predecessor. On the other side, we return our original successor and notify.

```
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(Xkey, Id, Skey) of
                           true ->
                                    Xpid ! {request, self()},
                                    {Xkey,Xpid};
                           false ->
                                    Spid ! {notify, {Id,self()}},
                                    Successor
                           end
         end.
```

What will happen if Skey = Xkey?, in this case it will not matter very much because that case is already handled in the pattern matching and so it will never reach the case in which we use the between function.The next part was to implement the notify function as described in the assignment. Following the guidelines we managed to finish it. We should remark that although we don't notify the original node about our decision in their proposal, there would be not too much issues. This is due to the periodic stabilize we have introduced which will update the nodes successor and will see what decision has been made on his proposal. If we didnt had this scheduled stabilize procedure, the system will not work correctly as we can see from the previous case as we could have nodes in which they seem to have successor that they shouldnt be for them.

Next we did some testing with the following commands with a small ring made with 3 nodes:

```
A=node:start(10).
B=node:start(30,A).
C=node:start(80,B).
```

Which gave the following results, showing that the ring was completely closed:

```
Sucessor:{80,<0.497.0>}  Predecesor:{10,<0.222.0>}  Id: 30
Sucessor:{10,<0.222.0>}  Predecesor:{30,<0.318.0>}  Id: 80
Sucessor:{30,<0.318.0>}  Predecesor:{80,<0.497.0>}  Id: 10
Sucessor:{80,<0.497.0>}  Predecesor:{10,<0.222.0>}  Id: 30
Sucessor:{10,<0.222.0>}  Predecesor:{30,<0.318.0>}  Id: 80
```

### 2.2 Second Implementation:

In this second implementation we had to add a new module which simulate will simulate the key store for the system and will include the splitting and merge of the stores when we added or took a node from the ring. So we also needed to introduce extra modifications to the original module. The firs thing we do is to define the storage module which will simulate the local storage of the nodes. To keep it easy we will use a tuple pair {Key,Value} and we will store them in lists.Also we will try to keep them sorted each time we add a new pair in the list. Although the methods are quite simple we would like stand out the methods for splitting and merging as follows:

```
merge(Store1,Store2) ->
        lists:merge(Store1,Store2).
split(Nkey,Store) ->
        lists:partition(fun({Key,Value}) -> Key=<Nkey end, Store).
```

In order to do the merge, we call back the merge function from the lists module. For the splitting we use the partition function from the lists library. This function lets us split the list in two lists, one which satisfies the given function and the other list which doesnt met the function given. This way we can easily implement a condition that will split the list until a certain key. The next step once we implemented the module was to modify the node module adding the add and lookup procedure which are as follows.

```
add(Key, Value, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store) ->
```

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

The next part was to modify the notify procedure, we simply had to take in mind that when we get a notification of the new predecessor, we might need to split our key store. In this case, our split condition simply checks which keys in the store are less or equal to the splitting key. The keys that meet this condition will be the ones we will need to hand over and the rest we will keep them. Also the modification we need to do in notify is simply to return the store that the node will need to keep and send the other list to the predecessor node. This should complete the changes we need to have the implementation up and running.

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

## 3 Conclusions

Thanks to this Chordy assignment, we get in touch with a real world implementation of a distributed system problem which is described in the paper given. It gives a new perspective about the world of the distributed systems field, this lets us see the importance this field and the impact is having right now. This problems shows us the potential that is behind the field of distributed systems which is clearly a top notch research area with a lot of future to come. It also introduces us the concept of distributed hash tables and how are treated actually, as these kind of systems have to address problems related to load balance, decentralization, scalability, availability and flexible naming during the desing of peer-to-peer systems. It also helps us to understand much better how the chord protocol works in real life, although the implementation we have done is much simpler than the one described in the real paper; but this is enough for us as if we get in depth with the development of the protocol it will take us much more time to implement chord protocol.