# Seminar Report: Chordy

Pedro Garcia Mota, Pau Espin Pedrol

June 7, 2013

## 1 Introduction

*Introduce in a couple of sentences the seminar and the main topic related to distributed systems it covers.*

The aim of this seminar is to implement a distributed hash table structure. Once we have a growing ring we will introduce a store where key-value pairs can be added.

## 2 Work done

*Present the new code that you have added or how you have implemented a required functionality by using small Erlang code snippets (you do not need to copy&paste all the code).*

### 2.1 node1.erl

```
-module(node1).
-export([start/1, start/2, init/2, node/3, notify/3, request/2,
    stabilize/3]).

-define(Stabilize, 2000).
-define(Timeout, 5000).

start(MyKey) ->
     start(MyKey, nil).

start(MyKey, PeerPid) ->
  timer:start(),
  spawn(fun() -> init(MyKey, PeerPid) end).

init(MyKey, PeerPid) ->
  Predecessor = nil,
  {ok, Successor} = connect(MyKey, PeerPid),
  schedule_stabilize(),
  node(MyKey, Predecessor, Successor).

connect(MyKey, nil) ->
  {ok, {MyKey , self()}};
```

```
connect(_, PeerPid) ->
  Qref = make_ref(),
  PeerPid ! {key, Qref, self()},
  receive
    {Qref, Skey} ->
      {ok, {Skey , PeerPid}}
  after ?Timeout ->
    io:format("Timeout: no response from ~w~n", [PeerPid])
  end.

node(MyKey, Predecessor, Successor) ->
  receive
    {key, Qref, PeerPid} ->
      PeerPid ! {Qref, MyKey},
      node(MyKey, Predecessor, Successor);
    {notify, New} ->
      Pred = notify(New, MyKey, Predecessor),
      node(MyKey, Pred, Successor);
    {request, Peer} ->
      request(Peer, Predecessor),
      node(MyKey, Predecessor, Successor);
    {status, Pred} ->
      Succ = stabilize(Pred, MyKey, Successor),
      node(MyKey, Predecessor, Succ);
    stabilize ->
      stabilize(Successor),
      node(MyKey, Predecessor, Successor);
    probe ->
      create_probe(MyKey, Successor),
      node(MyKey, Predecessor, Successor);
    {probe, MyKey, Nodes, T} ->
      remove_probe(MyKey, Nodes, T),
      node(MyKey, Predecessor, Successor);
    {probe, RefKey, Nodes, T} ->
      forward_probe(RefKey, [MyKey|Nodes], T, Successor),
      node(MyKey, Predecessor, Successor)
  end.

notify({Nkey, Npid}, MyKey, Predecessor) ->
  case Predecessor of
    nil ->
      {Nkey, Npid};
    {Pkey, _} ->
      case key:between(Nkey, Pkey, MyKey) of
        true ->
          {Nkey, Npid};
        false ->
          Predecessor
```

```
      end
  end.

request(Peer, Predecessor) ->
  case Predecessor of
    nil ->
      Peer ! {status, nil};
    {Pkey, Ppid} ->
      Peer ! {status, {Pkey, Ppid}}
  end.

stabilize(Pred, MyKey, Successor) ->
  {Skey, Spid} = Successor,
  case Pred of
    nil ->
      Spid ! {notify, {MyKey, self()}},
      Successor;
    {MyKey, _} ->
      Successor;
    {Skey, _} ->
      Spid ! {notify, {MyKey, self()}},
      Successor;
    {Xkey, Xpid} ->
      case key:between(Xkey, MyKey, Skey) of
        true ->
          self() ! stabilize,
          {Xkey, Xpid};
        false ->
          Spid ! {notify, {MyKey, self()}},
          Successor
  end
end.

stabilize({_, Spid}) ->
  Spid ! {request, self()}.

create_probe(MyKey, {_, Spid}) ->
  Spid ! {probe, MyKey, [MyKey], erlang:now()},
  io:format("Create probe ~w!~n", [MyKey]).

remove_probe(MyKey, Nodes, T) ->
  Time = timer:now_diff(erlang:now(), T),
  io:format("Received probe ~w in ~w ms Ring: ~w~n", [MyKey, Time,
      Nodes]).

forward_probe(RefKey, Nodes, T, {_, Spid}) ->
  Spid ! {probe, RefKey, Nodes, T},
  io:format("Forward probe ~w!~n", [RefKey]).
```

```
schedule_stabilize() ->
  timer:send_interval(?Stabilize, self(), stabilize).
```

### 2.2 node2.erl

```
-module(node2).
-export([start/1, start/2, init/2, node/4, notify/4, request/2,
    stabilize/3]).

-define(Stabilize, 2000).
-define(Timeout, 5000).

start(MyKey) ->
     start(MyKey, nil).

start(MyKey, PeerPid) ->
  timer:start(),
  spawn(fun() -> init(MyKey, PeerPid) end).

init(MyKey, PeerPid) ->
  Predecessor = nil,
  {ok, Successor} = connect(MyKey, PeerPid),
  schedule_stabilize(),
  node(MyKey, Predecessor, Successor, []).

connect(MyKey, nil) ->
  {ok, {MyKey , self()}};

connect(_, PeerPid) ->
  Qref = make_ref(),
  PeerPid ! {key, Qref, self()},
  receive
    {Qref, Skey} ->
      {ok, {Skey , PeerPid}}
  after ?Timeout ->
    io:format("Timeout: no response from ~w~n", [PeerPid])
  end.

node(MyKey, Predecessor, Successor, Store) ->
  receive
    {key, Qref, PeerPid} ->
      PeerPid ! {Qref, MyKey},
      node(MyKey, Predecessor, Successor, Store);
    {notify, New} ->
      {Pred, St} = notify(New, MyKey, Predecessor, Store),
      node(MyKey, Pred, Successor, St);
    {request, Peer} ->
      request(Peer, Predecessor),
      node(MyKey, Predecessor, Successor, Store);
    {status, Pred} ->
```

```
      Succ = stabilize(Pred, MyKey, Successor),
      node(MyKey, Predecessor, Succ, Store);
    stabilize ->
      stabilize(Successor),
      node(MyKey, Predecessor, Successor, Store);
    {add, Key, Value, Qref, Client} ->
      Added = add(Key, Value, Qref, Client, MyKey, Predecessor,
          Successor, Store),
      node(MyKey, Predecessor, Successor, Added);
    {lookup, Key, Qref, Client} ->
      lookup(Key, Qref, Client, MyKey, Predecessor, Successor, Store
          ),
      node(MyKey, Predecessor, Successor, Store);
    {handover, Elements} ->
      Merged = storage:merge(Store, Elements),
      node(MyKey, Predecessor, Successor, Merged);
    probe ->
      create_probe(MyKey, Store, Successor),
      node(MyKey, Predecessor, Successor, Store);
    {probe, MyKey, Nodes, T} ->
      remove_probe(MyKey, Store, Nodes, T),
      node(MyKey, Predecessor, Successor, Store);
    {probe, RefKey, Nodes, T} ->
      forward_probe(RefKey, [MyKey|Nodes], Store, T, Successor),
      node(MyKey, Predecessor, Successor, Store)
  end.

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

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


notify({Nkey, Npid}, MyKey, Predecessor, Store) ->
```

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

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

request(Peer, Predecessor) ->
    case Predecessor of
      nil ->
        Peer ! {status, nil};
      {Pkey, Ppid} ->
        Peer ! {status, {Pkey, Ppid}}
    end.

stabilize(Pred, MyKey, Successor) ->
    {Skey, Spid} = Successor,
    case Pred of
      nil ->
        Spid ! {notify, {MyKey, self()}},
        Successor;
      {MyKey, _} ->
        Successor;
      {Skey, _} ->
        Spid ! {notify, {MyKey, self()}},
        Successor;
      {Xkey, Xpid} ->
        case key:between(Xkey, MyKey, Skey) of
          true ->
            self() ! stabilize,
            {Xkey, Xpid};
          false ->
            Spid ! {notify, {MyKey, self()}},
            Successor
    end
end.
```

```
stabilize({_, Spid}) ->
  Spid ! {request, self()}.

create_probe(MyKey, Store, {_, Spid}) ->
  Spid ! {probe, MyKey, [MyKey], erlang:now()},
  io:format("Create probe ~w! Store ~w~n", [MyKey, Store]).

remove_probe(MyKey, Store, Nodes, T) ->
  Time = timer:now_diff(erlang:now(), T),
  io:format("Received probe ~w in ~w ms Ring: ~w Store ~w~n", [MyKey
      , Time, Nodes, Store]).

forward_probe(RefKey, Nodes, Store, T, {_, Spid}) ->
  Spid ! {probe, RefKey, Nodes, T},
  io:format("Forward probe ~w! Store ~w~n", [RefKey, Store]).

schedule_stabilize() ->
  timer:send_interval(?Stabilize, self(), stabilize).
```

### 2.3 node3.erl

```
  -module(node3).
-export([start/1, start/2, stop/0, init/2, node/5, notify/4, request
    /3, stabilize/4, monit/1]).

-define(Stabilize, 2000).
-define(Timeout, 5000).

start(MyKey) ->
     start(MyKey, nil).

start(MyKey, PeerPid) ->
  timer:start(),
  spawn(fun() -> init(MyKey, PeerPid) end).

stop() ->
  exit(self()).

init(MyKey, PeerPid) ->
  Predecessor = nil,
  {ok, Successor} = connect(MyKey, PeerPid),
  schedule_stabilize(),
  node(MyKey, Predecessor, Successor, nil, []).

connect(MyKey, nil) ->
  {ok, {MyKey, nil, self()}};

connect(_, PeerPid) ->
  Qref = make_ref(),
  PeerPid ! {key, Qref, self()},
```

```
    receive
      {Qref, Skey} ->
        MonitorRef = monit(PeerPid),
        {ok, {Skey, MonitorRef, PeerPid}}
    after ?Timeout ->
      io:format("Timeout: no response from ~w~n", [PeerPid])
    end.

node(MyKey, Predecessor, Successor, Next, Store) ->
    receive
      {key, Qref, PeerPid} ->
        PeerPid ! {Qref, MyKey},
        node(MyKey, Predecessor, Successor, Next, Store);
      {notify, New} ->
        {Pred, St} = notify(New, MyKey, Predecessor, Store),
        node(MyKey, Pred, Successor, Next, St);
      {request, Peer} ->
        request(Peer, Predecessor, Successor),
        node(MyKey, Predecessor, Successor, Next, Store);
      {status, Pred, Nx} ->
        {Succ, Nxt} = stabilize(Pred, Nx, MyKey, Successor),
        node(MyKey, Predecessor, Succ, Nxt, Store);
      stabilize ->
        stabilize(Successor),
        node(MyKey, Predecessor, Successor, Next, Store);
      {add, Key, Value, Qref, Client} ->
        Added = add(Key, Value, Qref, Client, MyKey, Predecessor,
            Successor, Store),
        node(MyKey, Predecessor, Successor, Next, Added);
      {lookup, Key, Qref, Client} ->
        lookup(Key, Qref, Client, MyKey, Predecessor, Successor, Store
            ),
        node(MyKey, Predecessor, Successor, Next, Store);
      {handover, Elements} ->
        Merged = storage:merge(Store, Elements),
        node(MyKey, Predecessor, Successor, Next, Merged);
      {'DOWN', Ref, process, _, _} ->
        {Pred, Succ, Nxt} = down(Ref, Predecessor, Successor, Next),
        node(MyKey, Pred, Succ, Nxt, Store);
      probe ->
        create_probe(MyKey, Store, Successor),
        node(MyKey, Predecessor, Successor, Next, Store);
      {probe, MyKey, Nodes, T} ->
        remove_probe(MyKey, Store, Nodes, T),
        node(MyKey, Predecessor, Successor, Next, Store);
      {probe, RefKey, Nodes, T} ->
        forward_probe(RefKey, [MyKey|Nodes], Store, T, Successor),
        node(MyKey, Predecessor, Successor, Next, Store);
      stop ->
```

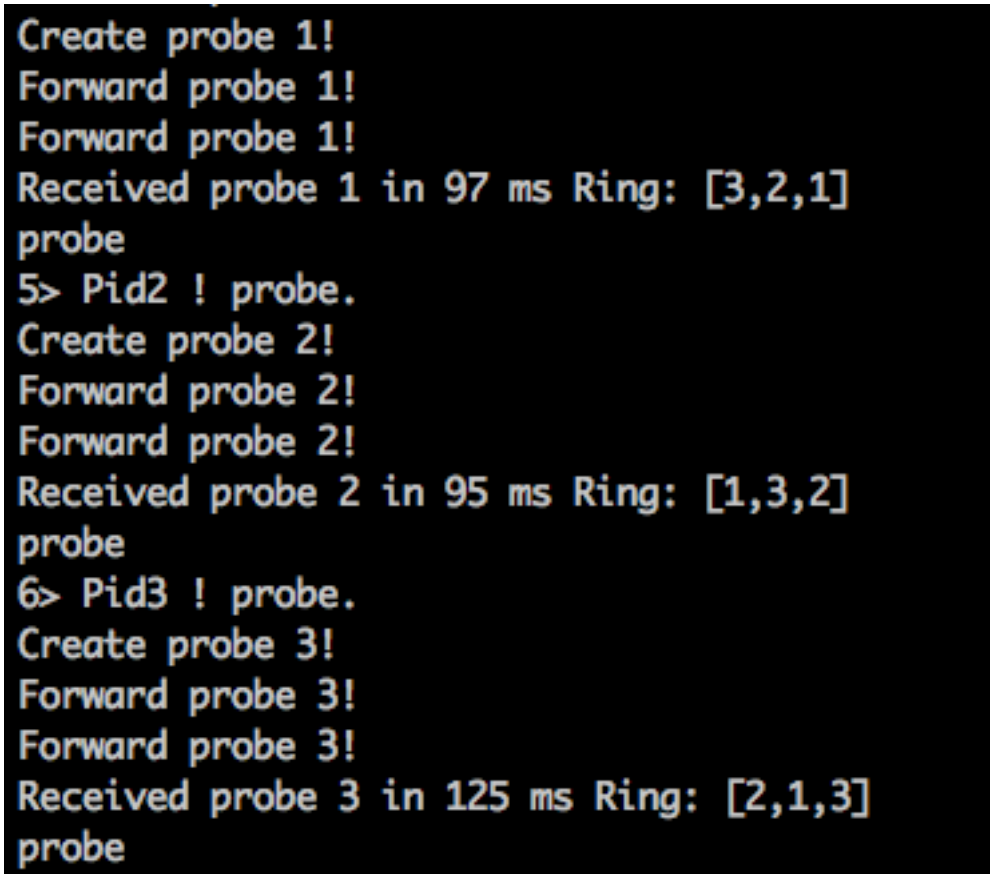
```
      stop()
  end.

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

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


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

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

```
request(Peer, Predecessor, {Skey, Sref, Spid}) ->
  case Predecessor of
    nil ->
      Peer ! {status, nil, {Skey, Sref, Spid}};
    {Pkey, Pref, Ppid} ->
      Peer ! {status, {Pkey, Pref, Ppid}, {Skey, Sref, Spid}}
  end.

stabilize(Pred, Nx, MyKey, Successor) -> %%monitor
  {Skey, Sref, Spid} = Successor,
  case Pred of
    nil ->
      Spid ! {notify, {MyKey, nil, self()}},
      {Successor, Nx};
    {MyKey, _, _} ->
      {Successor, Nx};
    {Skey, _, _} ->
      Spid ! {notify, {MyKey, nil, self()}},
      {Successor, Nx};
    {Xkey, _, Xpid} ->
      case key:between(Xkey, MyKey, Skey) of
        true ->
          Xref = monit(Xpid),
          demonit(Sref),
          self() ! stabilize,
          {{Xkey, Xref, Xpid}, Successor};
        false ->
          Spid ! {notify, {MyKey, nil, self()}},
          {Successor, Nx}
  end
end.

stabilize({_, _,Spid}) ->
  %%io:format("168 Spid ~w~n", [Spid]),
  Spid ! {request, self()}.

monit(Pid) ->
  erlang:monitor(process, Pid).

demonit(nil) ->
  ok;

demonit(MonitorRef) ->
  erlang:demonitor(MonitorRef, [flush]).

down(Ref, {_, Ref, _}, Successor, Next) ->
  {nil, Successor, Next};
```

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

create_probe(MyKey, Store, {_, _, Spid}) ->
  Spid ! {probe, MyKey, [MyKey], erlang:now()},
  io:format("Create probe ~w! Store ~w~n", [MyKey, Store]).

remove_probe(MyKey, Store, Nodes, T) ->
  Time = timer:now_diff(erlang:now(), T),
  io:format("Received probe ~w in ~w ms Ring: ~w Store ~w~n", [MyKey
      , Time, Nodes, Store]).

forward_probe(RefKey, Nodes, Store, T, {_, _, Spid}) ->
  Spid ! {probe, RefKey, Nodes, T},
  io:format("Forward probe ~w! Store ~w~n", [RefKey, Store]).

schedule_stabilize() ->
  timer:send_interval(?Stabilize, self(), stabilize).
```

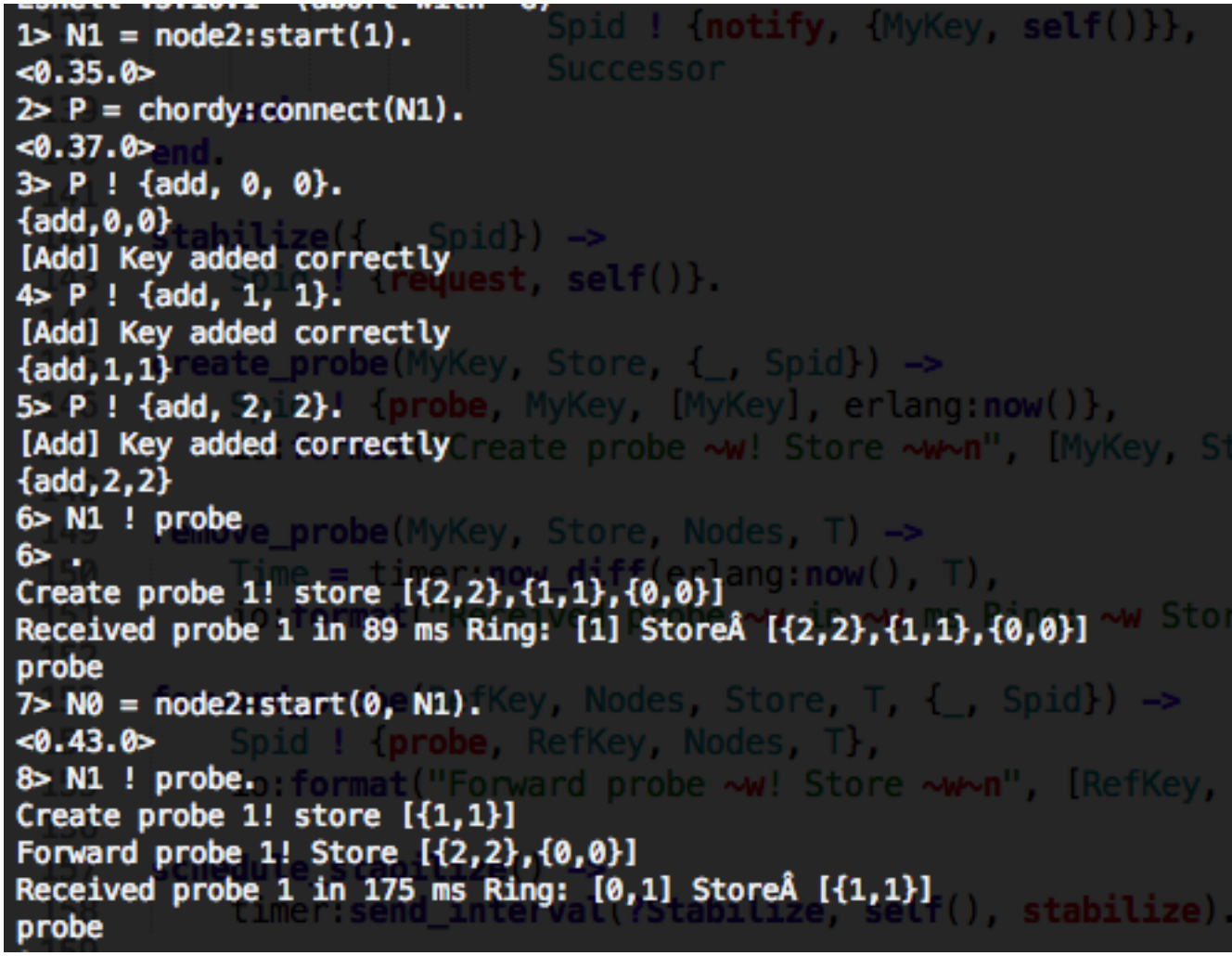
# 3 Experiments

- *Do some small experiments, to start with in one Erlang machine but then in a network or machines. When connecting nodes on different platforms remember to start Erlang in distributed mode (giving a name argument) and make sure that you use the same cookie (-setcookie)*

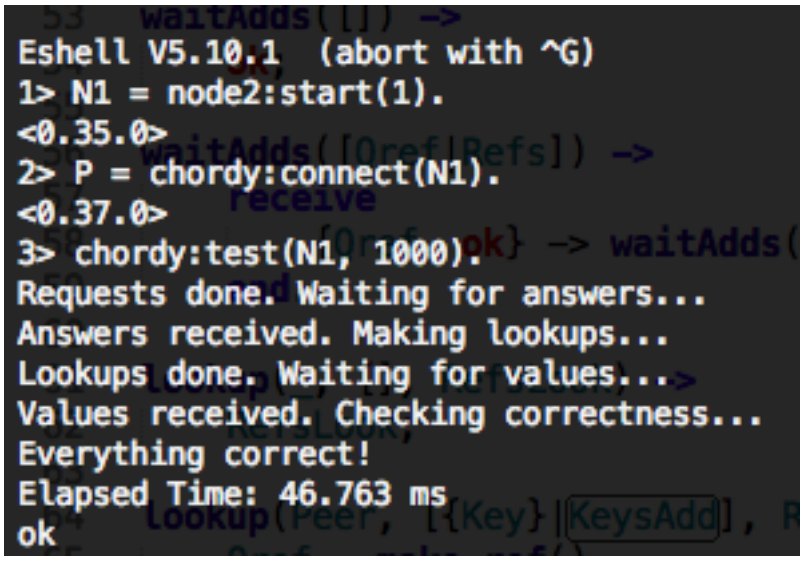



Stabilized vanilla version with 3 nodes and no stores.

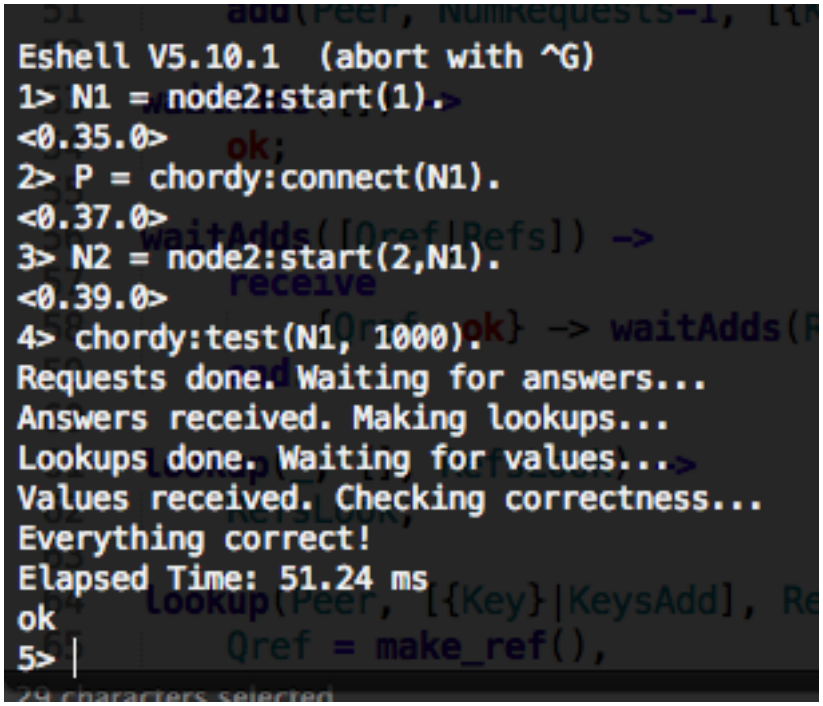
- *If we now have a distributed store that can handle new nodes that are added to the ring we might try some performance testing. You need several machines to do this. Assume that we have eight machines and that we will use four in building the ring and four in testing the performance.*

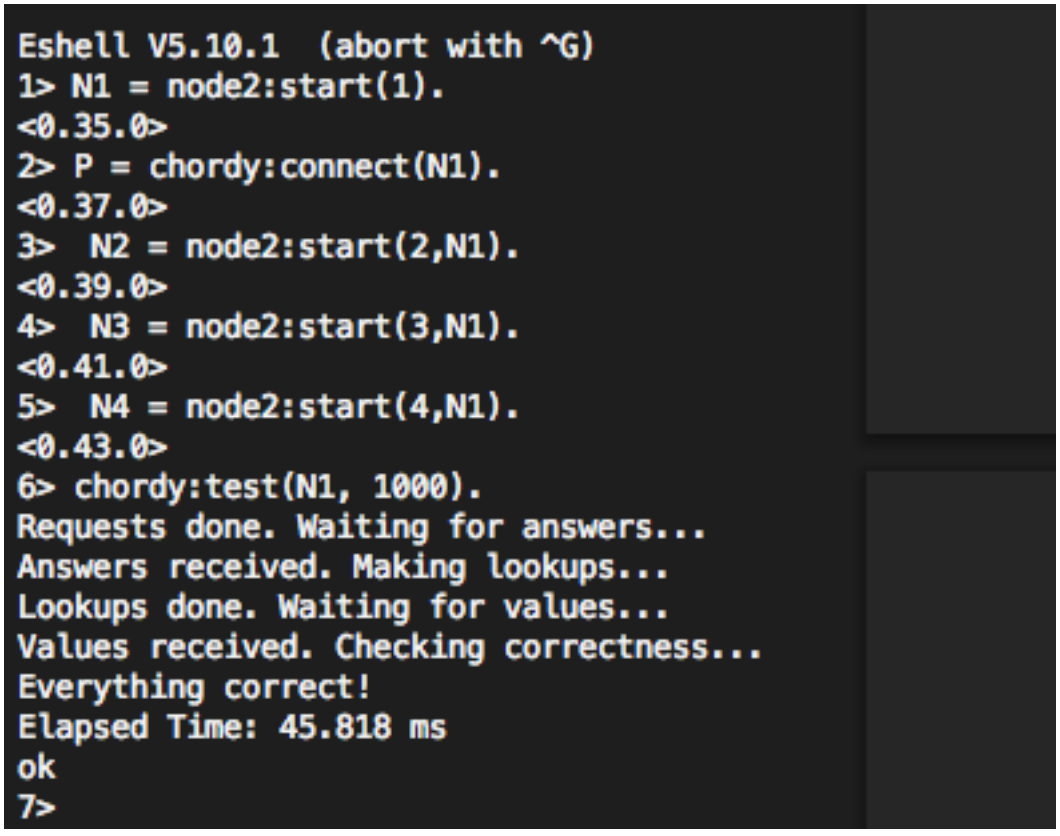



*As a first test, we have one node only in the ring and let the four test machines add a number (e.g., 1000) of random elements (key-value pairs) to the ring and then do a lookup of the elements, measuring the time it takes. You can use the test procedure given in 'Appendix A', which should be given with the name of the node to contact.*

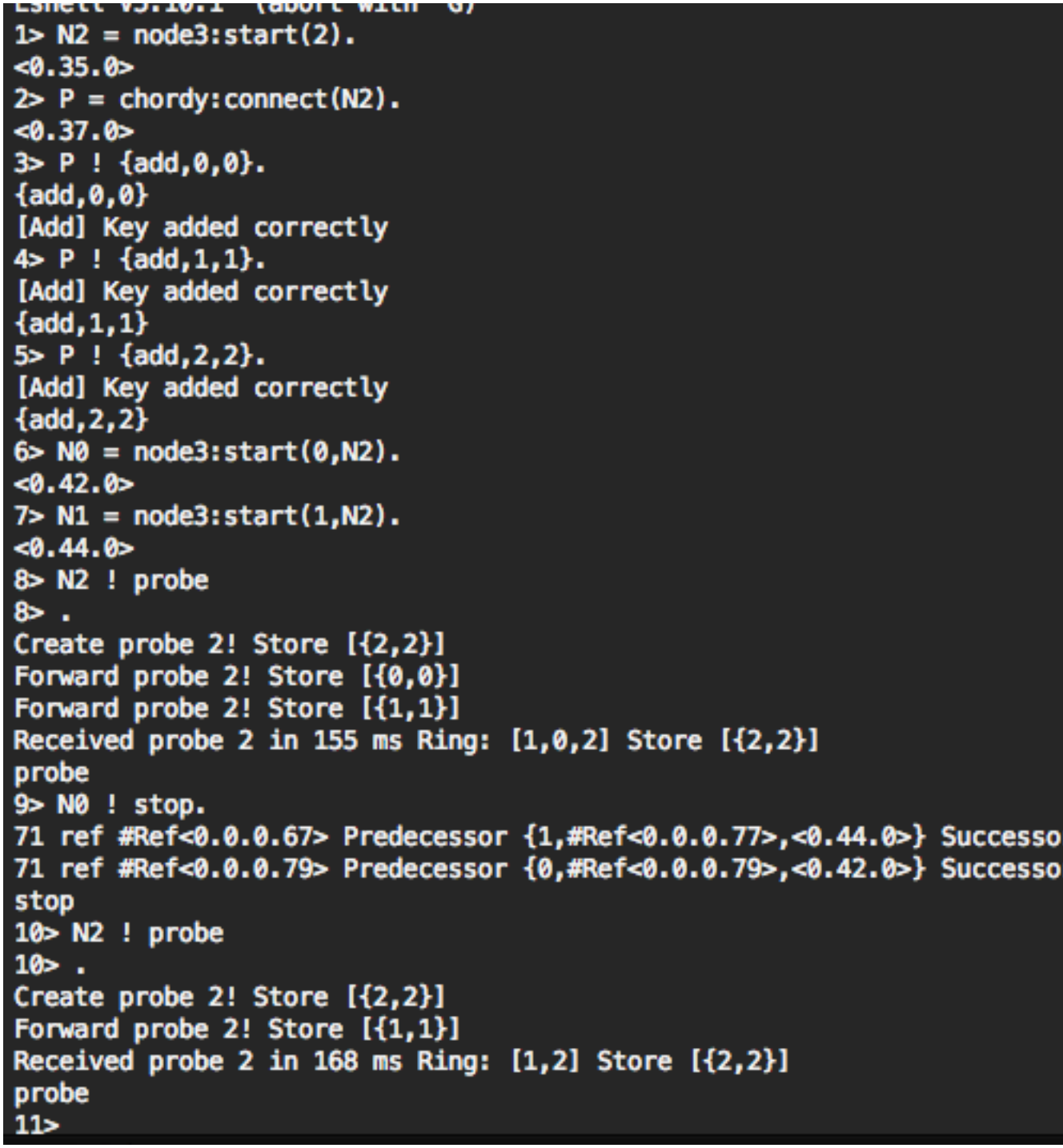


*How does the performance change if we add another node to the ring? Add two more nodes to the ring, any changes? Does it matter if alldemonit erlang test machines access the same node or different nodes?*



- *Do some experiments to evaluate the behavior of your implementation when nodes can fail.*




# 4 Open questions

- Open Questions. What are the pros and cons of a more frequent stabilizing procedure?

  The more frequent de stabilizing procedures are done, the more robust the distributed system will be. On the other hand, frequent stabilizing procedures causes a big overhead

on the sytem because additional messages are sent all the time.

- Do we have to inform the new node about our decision? How will it know if we have discarded its friendly proposal?

  The new node will probably have less information than us, so we have to point him to the right direction in order for it to find its proper successor. As we know more stuff than it, for example, we know our own successor, we can know better than him where can it continue looking for its successor.

- What would happen if we didn't schedule the stabilize procedure? Would things still work?

  Not at all, cause nodes already in which will be the predecessor of new entering nodes wouldn't be aware of those new nodes entries and shouldn't update their information, so the ring would become malformed.

- Open Questions. What will happen if a node is falsely detected of being dead (e.g. it has only been temporally unavailable)?

  It would be discarded and the ring would be temporally change till the apparently dead node would become available again, moment at which the ring would return to its initial topology.

# 5 Personal Opinion

*Your personal opinion of the seminar, including whether it should be included in next year's course or no .*

The seminar should be included in the next course as it expands the concepts of name servers and gives the opportunity to implement a naming distributed system similar to DNS, having a better idea of how these systems are implemented.