Report - Homework 5

Emil Karlsson

October 4, 2022

1 Introduction

This homework introduced the idea of distributing a key-value store between multiple nodes over the network.

2 node1

The first implementation of Chordy was essentially a ring topology network, since no persistent data were stored nor passed. However, some challenges arose along the way. The implementation was based on the idea of continuously stabilizing the network. Each iteration improved some connections and eventually it was a stable match between all the nodes. To assert this, it was vital that every node had a unique identifier. This was implemented as a random integer between one and one billion.

The last part added to the first implementation was a probing mechanism. This essentially worked by entering the network of nodes (ring network) and stepping through the nodes until it arrived back to the same node. Apart from the obvious information that a closed loop was found, it could also be used to print information at each of the nodes, count how many was stepped through along the way, and how long it took (for performance reasons). This proved useful to understand which nodes were connected to the network.

Below is the out where probing was run on a network of five nodes:

```
Nodes: [93,874,620,506,449,333]
Duration (microseconds): 59
```

3 node2

The second implementation added the actual distributed hash tables to the network. This implementation was a bit harder to wrap one's head around, as it tied neatly together with the tools already existing from node1. One of the interesting part was how node identifiers was used to group the items in the distributed hash tables at different nodes. This was an elegant solution because the key of the items in the distributed hash tables was created the same way as the nodes' identifiers.

As for implementation of the storage, it was just a basic wrapper for a list of tuples. However, it proved relatively difficult to implement the *add* and *lookup* methods. These methods required the node to check whether it should add/lookup or in fact forward the request to its predecessor. It was solved by comparing the identifier of the request with the identifier of its predecessor, if it was in the range of responsibility for the node it would add/lookup the item immediately. The neat part of this was how it formed a propagating effect "If I can't do it, my predecessor will" recursively.

Below is the output of probing the network where each node printed the size of its store, as well as a benchmark using the given *test* module In the example, 6 nodes were created and 25 items were added to the distributed hash table.

```
12> Peer ! probe.
Node: <0.108.0> Store size: 3
Node: <0.114.0> Store size: 4
Node: <0.110.0> Store size: 2
Node: <0.113.0> Store size: 12
Node: <0.112.0> Store size: 4
Node: <0.111.0> Store size: 0
-- Probe FINISHED --
Node count: 6
Duration (microseconds): 2110

13> test:check(Keys, Peer).
25 lookup operation in 0 ms
0 lookups failed, 0 caused a timeout
```

4 node3

The third implementation added fault tolerance by adding Erlang's built in monitor functionality. The idea was to make each node monitor both its successor and predecessor in order to be notified when they disconnected (for any reason). By observing *DOWN* messages, the network could use the information to build itself up again.

The difficult part of this implementation was the details; how would a node know what to reconnect to if its successor died? To solve this problem, another node relationship was added called the *Next node*, which was just the successor of the successor. This was not needed backwards since predecessors was allowed to be *nil*. Therefore, it came down to two cases, did the predecessor die or did the successor die? Below is a snippet of code how this was implemented.

```
down(Ref, {_, Ref, _}, Successor, Next) ->
  {nil, Successor, Next};
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
  Nref = monitor(Npid),
  Npid ! stabilize,
  % Hopefully a stabilize will occur before another node crashes...
  {Predecessor, {Nkey, Nref, Npid}, nil}.
```

As stated in the comment, if the continuous stabilization was not done until another crash happened, it would not be possible to recover, since the *Next node* would not have been set again.

This leads to some speculations how it could be improved. As mentioned in the PDF for the Homework, a general solution would be to give every node a list of other nodes as form of back up recovery plan. Another idea would be to minimize the risk of multiple nodes crashing by stabilizing more often. However, that could lead to performance issues as many messages are sent over the network.

Below is the result of testing the fault tolerance of the network. The same setup as the test for *node2* above was used.

```
11> Peer ! probe.
Node: <0.108.0> Store size: 3
Node: <0.110.0> Store size: 3
Node: <0.114.0> Store size: 6
Node: <0.113.0> Store size: 3 <--- We kill node <0.113.0> with 3 items
Node: <0.112.0> Store size: 1
Node: <0.111.0> Store size: 9
-- Probe FINISHED --
Node count: 6
Duration (microseconds): 818

% The connected nodes reported the loss
13> <0.113.0> ! stop. % Killing the node
Received DOWN from : #Ref<0.424320506.4123262980.45400>
Received DOWN from : #Ref<0.424320506.4123262980.45415>

% The gap was closed
Started monitor [By, On]: [<0.114.0>,<0.112.0>]
Started monitor [By, On]: [<0.112.0>,<0.114.0>]

% But 3 items are missing, as expected
14> test:check(Keys, Peer).
```

```
25 lookup operation in 0 ms
3 lookups failed, 0 caused a timeout
```