Routy – a small routing protocol

Óttar Guðmundsson
14 September, 2017

1 Introduction

This week's assignment was to develop and test out a so called *Link-state* routing protocol and test it out to gain insight how nodes in the network can maintain consistent views, even though we have to deal with node failures and other things.

Our code had to be able to send messages to other processes that were named after cities, e.g. 'stockholm' or 'lund' but I preferred to use the Icelandic airport codes 'rvk' and 'aki'. But the sending method wasn't just so simple because our router needed to figure out which path it should take before sending it for fastest delivery.

All of the code was written in Erlang. This felt a little bit easier than prior assignment since I understood the syntax better. New functions were introduced that helped with the development.

2 Main problems and solutions

As last time, we were given instructions on how we should structure our code and files. Aside from receiving names of functions and modules, we were also given new functions to use such *as foldl/3, map/2* and *keyfind/3*. They were easy to understand, because they worked just like the same functions in similar functional programing langue called F#.

2.1 Map

The first code we wrote towards our Routy implementation. We were given the functions as seen above which worked quite well and weren't hard to implement. I only had a little bit of trouble with the last function *all_nodes*. After reading through the documentation I found a class called **sets** which got me rid of duplicate nodes in the list.

```
all_nodes(Map) ->
  EmptyList = [],
  List = lists:foldl(fun({Node,Links}, Final) ->
    [Node|Links] ++ Final end, EmptyList, Map),
  SetOfSingles = sets:from_list(List),
  sets:to_list(SetOfSingles).
```

Image 1: Code from Map.erl

2.2 Dijkstra

By far the hardest part. I've known the algorithm and how it works for quite some time, but implementing it in Erlang was a bit frustrating. From the skeletal structure provided, writing *entry, replace* and *update* was not so hard because the code was similar to the map file.

At first I started writing the *iterate* function by using a big switch/case for all possible problems. I decided to make my code more clean using pattern matching as we did in the http parser from Rudy.

```
%Pattern match for empty list
iterate([],_Map,Table) ->
  Table;
%Pattern match for dummy list. Check if first node has inf
% dont care about RestOfList
iterate([{_Node, inf, _Gateway} | _RestOfInfiniteList],_Map,Table) ->
  Table;
%recursive for values needed to find
iterate([{Node, Length, Gateway} | RestOfList],Map,Table) ->
```

Image 2: Code from dijkstra.erl

The *table* function seemed like a hard part to code but after reading the description from the assignment paper, the only thing needed to do was to set each node to starting stage (that is, infinite length and destination unknown) , and then add the gateways to the same list with 0 length.

2.3 History

The history was easy to write. I tried something new in Erlang which I hadn't done before but that was the use of an *if* sentence in the code. The code never compiled for me but after using adding a *true* statement to it, it worked as expected.

```
if N > LastNumber ->
  % return new, list where the old one is
  {new, lists:append(lists:keydelete(Node,
true ->
  %else the message is old
old
```

Image 3: Code from history.erl

2.4 Interface

Adding interface to the router was needed to handle simplified actions such as finding a process by its Pid or Name. This was by far the easiest part of the assignment and I had no problems implementing them. But as I thought this was the easiest part, I made some typos that gave us error in the extra point demonstration.

```
  % This was wrong, used index of 1 instead of 2
name(Ref, Intf) ->
  SearchInterface = lists:keyfind(Ref, 2, Intf),
  case SearchInterface of
    false ->
      notfound;
    {Name, _Ref, _Pid} ->
      {ok, Name}
    end.
```

Image 4: Code from interface.erl

2.5 Routy

Not much to say here, except this was the hardest reading part. I didn't implement any of my code into the router itself except a *status* function to see my routers table and map, since the whole skeletal code was provided with the functions as well. The router had to take care of the most important things though, like keeping track of its interfaces and managing the link-state messages sent from other routers.

The most important part was to manually broadcast and update the routers as they were connected. In a more sophisticated manner this could be done automatically.

3 Evaluation

After some refactoring, tests and recompiling I managed to get the nodes talking without errors, so I guess this worked out okay. For this assignment we were not asked to measure anything. Also, there was nothing given to measure our code performance or measure any data or statistics regarding the routers efficiency. There were some ideas about attacking the network by overflooding it with requests or trying to send in some kind of malicious code to try and crash our node collection but I will leave that out until I've gained better knowledge about Erlang.

But to really test out the code, me and two other students in my program got together to test it out. Not only for the bonus points, but also to see if the code worked as it was supposed to. Sadly, in our demonstration we noticed that the code was very fragile and if any errors came up in the midst of the session, the whole network crashed and we had to start all over.

3.1 The world (Extra points)

Me, Max and Xin grouped up to try out the optional assignment. We had trouble starting since there were some troubles with connection to my computer. We didn't change the code from the day before and my partners couldn't even ping my computer. Thus, I had to restart and we were delayed a bit. After that, everything worked again (cons of going to school with corporate laptops).

After the restart, we started. I was North America, Xin was Asia and Max was Europe. In our demo, North America had NewYork as a city, Asia had Beijing and Shanghai and Europe had Stockholm. The test demonstrated was simple – NewYork was connected to Beijing and Shanghai, Shanghai and Beijing were connected to Stockholm. After broadcasting and updating, I would send a message to Stockholm which worked perfectly. Now in our first try, Asia killed Beijing and we expected the network to hold but sadly, my shell crashed and we didn't understand why. After looking at the code in my *interface* file, the *name* function was looking for the wrong tuple index when using *keyfind*. After fixing that, the code worked as expected.

The live demonstration can be found at chapter ***5 Demonstrated code.***

4 Conclusions

As I started the assignment it looked like a big step up from Hello World and Rudy. It really was a step up, but not as intense as it first looked. By delegating methods and files to the guidelines provided, the implementation made sense in a way. That is, after reading through it carefully over and over.

This assignment really showed how a complex system of networks might grow and how big it can get. Managing it like this takes a lot of work but I guess nowadays there exists a better way for handling overlaying networks and more.

5 Demonstrated code

Started by opening up the Erlang shell with name

```
werl –name northa@130.229.139.30 –setcookie routy –connect_all false
```

After that, the shell opened and we started the live demonstration

```
Erlang/OTP 20 [erts-9.0] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]

Eshell V9.0  (abort with ^G)
(northa@130.229.139.30)1> cd("C:/Users/ottarg/Documents/erl/routy").
C:/Users/ottarg/Documents/erl/routy
ok
(northa@130.229.139.30)2> c(history).
{ok,history}

(northa@130.229.139.30)26> routy:start(n1, newyork).
(northa@130.229.139.30)27> n1 ! {add, beijing, {a1, 'asia@130.229.183.88'}}.

(northa@130.229.139.30)30> n1 ! update.
Table [{beijing,beijing}]

(northa@130.229.139.30)31> n1 ! broadcast.
broadcast {links,newyork,1,[beijing]}

(northa@130.229.139.30)35> n1 ! broadcast.
broadcast {links,newyork,2,[beijing]}

(northa@130.229.139.30)39> n1 ! update.
Table [{newyork,beijing},{stockholm,beijing},{beijing,beijing}]
(northa@130.229.139.30)40> n1 ! {send, stockholm, 'hejdo'}.
newyork: routing message (hejdo){send,stockholm,hejdo

(northa@130.229.139.30)41>
=ERROR REPORT==== 13-Sep-2017::16:02:28 ===
Error in process <0.113.0> on node 'northa@130.229.139.30' with exit value:
{{case_clause,{beijing,#Ref<0.170562145.197918721.55155>,
                       {a1,'asia@130.229.183.88'}}},
 [{interface,name,2,[{file,"interface.erl"},{line,37}]},
  {routy,router,6,[{file,"routy.erl"},{line,66}]}]}
```

```
(northa@130.229.139.30)50> c(interface).
{ok,interface}

(northa@130.229.139.30)53> routy:start(n1, newyork).
(northa@130.229.139.30)54> n1 ! {add, beijing, {a1, 'asia@130.229.183.88'}}.
(northa@130.229.139.30)55> n1 ! {add, shanghai, {a2, 'asia@130.229.183.88'}}.

(northa@130.229.139.30)58> n1 ! update.
Table [{shanghai,shanghai},{beijing,beijing}]

(northa@130.229.139.30)59> n1 ! broadcast.
broadcast {links,newyork,1,[beijing,shanghai]}

(northa@130.229.139.30)60> n1 ! update.
Table [{newyork,beijing},{stockholm,beijing},{shanghai,shanghai},{beijing,beijing}]

(northa@130.229.139.30)61> n1 ! {send, stockholm, 'hejdo'}.
newyork: routing message (hejdo){send,stockholm,hejdo}
newyork: exit received from beijing

(northa@130.229.139.30)62> n1 ! update.
Update Table [{newyork,shanghai},{stockholm,shanghai},{shanghai,shanghai}]

(northa@130.229.139.30)63> n1 ! {send, stockholm, 'hejdo'}.
newyork: routing message (hejdo){send,stockholm,hejdo}
newyork: exit received from shanghai
```