Routy: a small routing protocol

Joakim Öberg

September 17, 2020

1 Introduction

*This report describe the main problems and solutions that was found when I implemented a small routing protocol. The code files for this lab was map.erl, interface.erl, hist.erl, dijkstra.erl and routy.erl. In **Main problems and solutions** I explain what I thought was hard and my solutions to the faced problems.In **Evaluation** I go thorough some of the problems my router can face and what parts of the code that could be improved.*

2 Main problems and solutions

2.1 Map

For the implementation of the functions **update** and **reachable** I used the library function *lists:keyfind* to easily find a specific tuple in the map. In **update** I also used *lists:keydelete* to delete the Node that was to be updated. Another function that was a bit tricky was **all_nodes**, to remove the tuples and add the nodes in a list was straight forward but because the links was already in a list, the final list ended up to be a list of lists. This was solved with *lists:flatten* which does exactly this, removes lists within lists.

2.2 Dijkstra

The function **entry** and **replace** both iterates through the lists of sorted to find if the searched node is present. In **entry** if found we simply return the path length and in **replace** we insert a new path length and sort the list before returning it. **Update** is simply checking if the new path length is shorted than the old one and if it is shorter updates the path length to the new.

The hardest function to implement was by far **iterate**, mostly because I thought it was hard to understand how I could iterate through a list and in the same time update the same list without being stuck in an infinite loop.

This is also where for each of the nodes in the list the paths is increased by one. This was as said in the instruction *The heart of the algorithm* and was probably therefor it was harder to implement than the rest of the functions. The exercise when Klas went through it helped a lot for my understanding how to implement it.

Another difficult function was **table** that I first implemented without sorting out duplicates in the concatenated list of nodes of the map and gateways, this resulted in that a node both could have the length set to infinity, the gateway set to unknown and also have an entry where the length was zero and gateway set to itself. This problem was something I found when the program was completed and I made my own network. It was solved by sorting the list of duplicates then map all the lengths to infinity and gateway to unknown, then they passed through a function **isGatewayOrNot** that took the list of tupes with nodes, length inf and gateway to unknown, and the list of gateways. For each of the tuples the node was searched for in gateways if it was present or not, if it was we called **update** to set length to zero and gateway to itself.

route simply searches the Table with *lists:keyfind* if the node is present or not found.

2.3 Interfaces

The functions in the interfaces was easy to implement but when I first did **remove** it was a bit wrong and was not discovered until I tried to remove a link in a network I played aroud with. The problem was that it was not only removing the one wanted but removing the whole list, always returning an empty list.

```
remove(Name,Intf)
remove(_Name,[]) -> [];
remove(Name,[{Name,_,_}|Rest]) -> Rest;
remove(Name,[H|Rest]) -> remove(Name,Rest).
```

It was doing it because on the last row, it was not saving the Head. For it to work it should looked like

```
[H|remove(Name,Rest)]
```

But instead I solved it by using the library function *lists:keydelete* which returns either the same list if not found or the list without the one that was deleted.

The function **add** did first not check if the name already was present, and we could add an infinite amount of links to the same city, making the interface longer than needed. In **broadcast** I first was unsure if Erlang would understand if a message was sent directly to a PID. I thought not and was only sending the reference to a process reference, for example r1 in PID:

```
{r1,'sweden@130.123.112.23'}
```

This became an error when I started routers on different machines and tried to connect to them. Without the IP address my computer always looked for a process registered on itself, where it obviously did not found it. Some searches on the internet gave me the answer that Erlang does understand if a message is sent directly to a PID and I changed the code.

2.4 History

My first Idea when create a new history was to simply create it with a tuple and of course I thought that the history shoud be set to zero, even though the instructions said *where messages from Name will always be seen as old.*

```
new(Name) -> {Name,0}.
```

This solution had two problems; 1. because the history was set to zero, one could receive messages from it self and therefor create cycles. 2. My function to update history **hist:update**:

```
update(Node, N, History) ->
    case lists:keyfind(Node, 1, History) of
        {Node,Sofar} ->
             case N =< Sofar of
                true -> old;
                false ->
                    UpdatedHistory = lists:keydelete(Node, 1, History),
                    {new,[{Node,N}|UpdatedHistory]}
end;
false -> {new, [{Node,N}|History]}
end.
```

Tries with *lists:keyfind* to search for node in a list of tuples where the first element should match against node and the history should be a list. Because I implemented a new history as a tuple this will miss match the first time and throw an error. These problems was solved by change the history for itself to infinity and set the function **new** to return a tuple inside a list.

```
new(Name) -> [{Name,inf}].
```

Summarize your problems, proposed solutions, etc. You do not need to copy&paste your code. Only if needed, you may write down small code snipeds to show how you have solved a specific problem/question.

2.5 Router(routy)

How the router works was already implemented and the code was given from the instructions. Three messages the router could receive *update*, *status* and *broadcast* was given outside the first code skeleton that was given, but was just added into it. For the message status, I created a function **status** to make a pretty print of the status of the node. It works by calling *routy:status(Node)*, code for *status*:

```
status(Reg) ->
    Reg ! {status, self()},
    receive
        {status, {Name, N, Hist, Intf, Table, Map}} ->
            io:format(" Status recieved from: ~w~n Name: ~w~n N: ~w~n Hist: ~w~n Int
end.
```

Status gets the name registered for a node and sends a status message to the registered name, with the PID to itself (self()).

```
{status,From} ->
    From ! {status, {Name, N, Hist, Intf, Table, Map}},
    router(Name, N, Hist, Intf, Table, Map);
```

The registered name, whose router is listening for messages receives the status message and returns the information about itself which **status** receives and prints it. The function **status** was also exported.

2.6 Bonus point: The world

The hardest part for the bonus point was to set up the network, simple enough to set up but complex enough so that a node could be stopped and the network took the longer path.

3 Evaluation

For this router we have to do the broadcast and update manually, to improve this we could set the router to send a broadcast message after it have received for example an add or remove message and update itself. This could also be done by setting a timer on how often a broadcast and update should be sent. The biggest problem however is if we have a directed network where we sometimes can reach a node, but it can not reach us back. Then the routing table is complete with all necessary information in the last node, but because the first node have not received any broadcasts it does not know who it can reach other than its gateways. In this router we need to add each directed node one at a time, maybe we could after receiving an add message send back a reversed add message to from who we got it

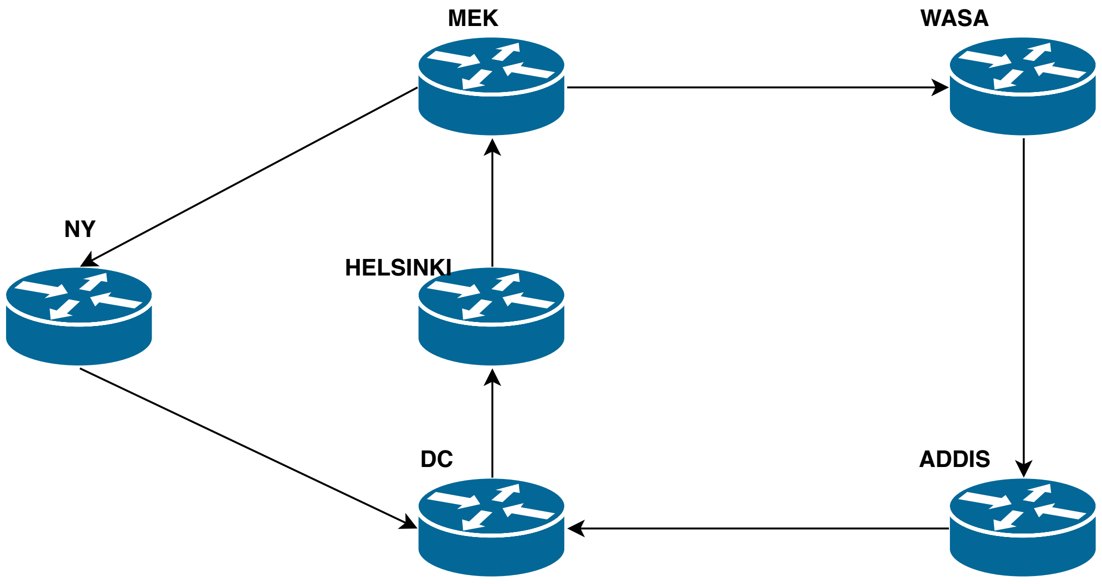

Figure 1: Overview of the bonus point network

from creating a path that both nodes can use to reach each other. Another problem with messages is that you does not know if it was received or not, and if you received one you does not from whom it was.

One thing that could be improved that I tried to think about was time complexity, most of my functions is linear in time and could probably be solved in a better way if I gave it some more time. The code could also be refactored by for example learning foldl and do some of the functions in one line instead of in a new function.

3.1 Bonus point: The world

The network me and two friends created contained the three countries with cities **Ethiopia:** *MEK, ADDIS*, **Finland:** *HELSINKI, WASA* and **USA:** *DC, NY*.

The network was set up and tested so that one could send messages to each others. When Helsinki sent a message to DC it was routed via MEK and then routed via NY and then arrived in DC, hence it took the fastest route. Then we stopped the NY node, and MEK received an exit message *mek: exit recived from ny.* If we not update MEK, MEK still think it can send to NY, which results in a dropped message if someone tries to send to DC. But if we update MEK and broadcast and do that on the other nodes to. Then when we sent a message again from Helsinki to DC and now Wasa and addis started to route messages *wasa: routing message ('helo from helsinki')* exactly as it should work. Then we did another test, set everything up in the exact same way but instead of sending a stop to NY we disabled the network card for the computer holding USA network. A message was sent once again from helsinki to DC, and after ca 60 seconds the computer holding Ethiopia and the computer holding Finland received:

```
(finland@130.229.183.123)130> =ERROR REPORT==== 17-Sep-2020::16:47:57.989285 ===
** Node 'usa@130.229.190.98' not responding **
** Removing (timedout) connection **
```

4 Conclusions

In this assignment we implemented a small routing protocol. This lab got me into thinking how modern routers work and how much of the network that has to be filed with messages of adding, removing and broadcasting information. The router that we implemented is very basic and could be better improved by automate broadcast, update and make links directed in both ways. In the bonus point assignment we connected three different countries on different computers, each of them having two cities. We sent messages from them to see that they indeed sent via the fastest route then we ran two different tests, one where we stopped a node and the other where we disconnected the network card for one of the countries. From the test we saw that the routers do recalculate a new route if the fastest one disappears.