Report 2: Routy - a routing network

Yining Hou

September 20, 2023

1 Introduction

In this report, I implement a link-state routing protocol in Erlang. I also connect my router to my group members and communicate with each other.

In this seminar, the main topic is about routing protocols for Internet routers. Routing is important for distributed issues, as nodes need to cooperate and share knowledge with each other.

2 Main problems and solutions

Set up different processes to simulate different routers over the network. Send and receive messages among these routers.

I use different processes to represent different "cities" routers in "the world". The main steps of the implementation includes: construct a map of the network, compute the routing table using Dijkstra's algorithm, manage interfaces and history, and build the router based on these modules.

2.1 The map

The map is a representation of the connection among nodes, in which each entry links to a list of directly connected cities. For example, in `{stockholm, [lund, uppsala...]}`, which means stockholm can forward messages to lund and uppsala, while lund and uppsala cannot send messages to stockholm.

In the function `all_nodes(Map)`, in order to avoid duplication, I find the function `lists:usort(List)` useful, where all except the first element of the elements comparing equal have been deleted.

2.2 Dijkstra

From the maps of cities and their links, the router needs a routing table, so it knows the best gateway to each destination. The table should look like this: `[{stockholm, lund}, {lund, lund}...]`. It says that if we want to send a message to stockholm, we should send it to lund first. Since lund is a gateway, we can send the message to itself directly.The implementation can be divided into two parts.

2.2.1 A sorted list

First, create a sorted list which will be later used to calculate the routing table. Each entry in the list should look like this: `{stockholm, 2, lund}`. It contains the city's name, the path length to the city, and the gateway we should use to reach the city.

In the function `replace(Node, N, Gateway, Sorted)`, the resulting list should also be sorted by length to the city. I use `keymerge` function, which merges two TupleLists on the second element of each tuple.

```
replace(Node, N, Gateway, Sorted) ->
    lists:keymerge(2, [{Node, N, Gateway}],
    lists:keydelete(Node, 1, Sorted)).
```

In the function `entry(Node, Sorted)`, we return 0 if the city is not found. Because if no entry is found, then no new entry is added. In the `update` part, N hops cannot be less than 0, so it can only return the original list.

2.2.2 The iteration

Second, do the iteration. We take a sorted list and a map to construct the routing table. Each element in these 3 data structures should look like:

- Sorted List - [{City, Dist, Gateway}...]
- Map - [{City, [Links]}...]
- Table - [{City, Gateway}...]

In the function `iterate(Sorted, Map, Table)`, if the entry city is valid, I apply update to each city in the map reachable from this city and the sorted list is updated by accumulator `S`.

```
[{Node, Dist, Gateway} | Rest] ->
    NewSorted = lists:foldl(fun(N, S) ->
    update(N, Dist + 1, Gateway, S) end,
    Rest, map:reachable(Node, Map))
```

In the function `table(Gateways, Map)`, I divide cities into gateways and non-gateways and append non-gateways list to the gateways list, so the list is sorted by the length to the city. Pass `Map` and an empty list as `Table` to the `iterate` function I implemented above.

2.3 Interfaces

The router needs to keep track of a set of interfaces. The interface should have a structure of symbolic name (stockholm), a process reference, and a process identifier.

We can add and remove a interface for a city router. The interface's name represents a city's name and the process identifier will be used to send message to that city in the part of `broadcast`.

2.4 The history

The router also needs a way to avoid cyclic paths. We can achieve this by tagging each constructed message with a per router increasing message number.

Every time a router broadcasts a message, it will increase its counter by 1. When other routers receive a message from it, they will compare the tagged number with the number in history. If the tagged number is bigger than record or the entry doesn't exist, they will accept the message, otherwise it's an old message which should be thrown away.

2.5 The router

Build the router based on all implemented above. The commands the router receives can be divided into 3 parts: managing interfaces, link-state messages and routing messages.

Interfaces can be added or removed from one router. Then the router broadcasts link-state messages to its interfaces and increases its counter. Interfaces will recursively receive messages and update their maps. After that, call `update` for each router to use dijkstra algorithm to create routing tables. Now, it's easy for the routers to send and receive messages.

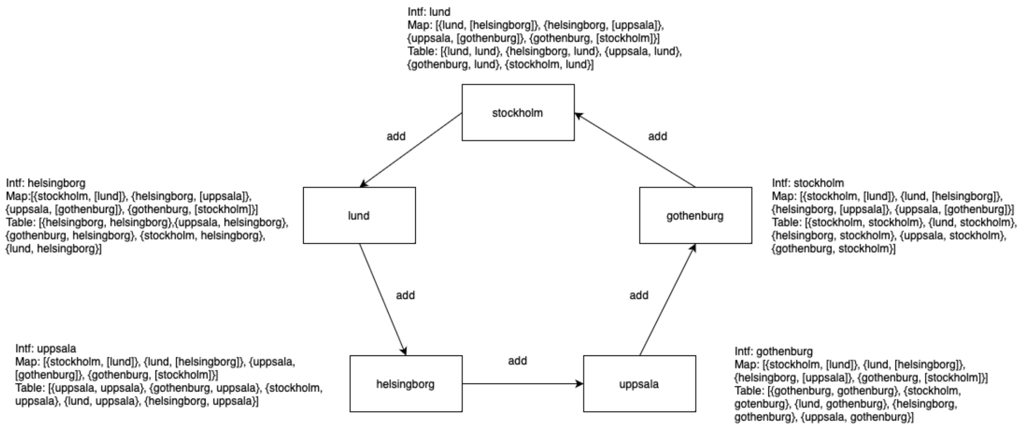
I also implement a function `status(Router)` which helps me see the details of a router clearly.

```
status(Router) ->
    Router ! {status, self()},
    receive
        {status, {Name, N, Hist, Intf, Table, Map}} ->
        io:format("Name : ~w~n
        N : ~w~n
        Hist : ~w~n
        Intf : ~w~n
        Table : ~w~n
        Map : ~w~n",
        [Name, N, Hist, Intf, Table, Map])
    end.
```

3 Tests

Test the protocol by starting several routing processes and letting them connect to each other.

I start 5 different processes to represent 5 different city routers in the network of sweden. The relationship and status details are shown in Figure 1.



Intf: lund
Map: [{lund, [helsingborg]}, {helsingborg, [uppsala]}, {uppsala, [gothenburg]}, {gothenburg, [stockholm]}]
Table: [{lund, lund}, {helsingborg, lund}, {uppsala, lund}, {gothenburg, lund}, {stockholm, lund}]

Intf: helsingborg
Map:[{stockholm, [lund]}, {helsingborg, [uppsala]}, {uppsala, [gothenburg]}, {gothenburg, [stockholm]}]
Table: [{helsingborg, helsingborg},{uppsala, helsingborg}, {gothenburg, helsingborg}, {stockholm, helsingborg}, {lund, helsingborg}]

Intf: stockholm
Map: [{stockholm, [lund]}, {lund, [helsingborg]}, {helsingborg, [uppsala]}, {uppsala, [gothenburg]}]
Table: [{stockholm, stockholm}, {lund, stockholm}, {helsingborg, stockholm}, {uppsala, stockholm}, {gothenburg, stockholm}]

Intf: uppsala
Map: [{stockholm, [lund]}, {lund, [helsingborg]}, {uppsala, [gothenburg]}, {gothenburg, [stockholm]}]
Table: [{uppsala, uppsala}, {gothenburg, uppsala}, {stockholm, uppsala}, {lund, uppsala}, {helsingborg, uppsala}]

Intf: gothenburg
Map: [{stockholm, [lund]}, {lund, [helsingborg]}, {helsingborg, [uppsala]}, {gothenburg, [stockholm]}]
Table: [{gothenburg, gothenburg}, {stockholm, gotenburg}, {lund, gothenburg}, {helsingborg, gothenburg}, {uppsala, gothenburg}]

Figure 1: Routy

Start the test and send a message 'Greet from stockholm' from stockholm(r1) to gothenburg(r5).

```
(sweden@192.168.1.100)6> r1 ! {send, gothenburg, 'Greet from stockholm'}.
stockholm: routing message ('Greet from stockholm')
lund: routing message ('Greet from stockholm')
helsingborg: routing message ('Greet from stockholm')
uppsala: routing message ('Greet from stockholm')
gothenburg: received message 'Greet from stockholm'
```

4 The world

Cooperate in a group and each member represents a region in the world. Create routers and connect to each other.

I will demonstrate this part with my classmates in class. The routers' network is shown in Figure 2.

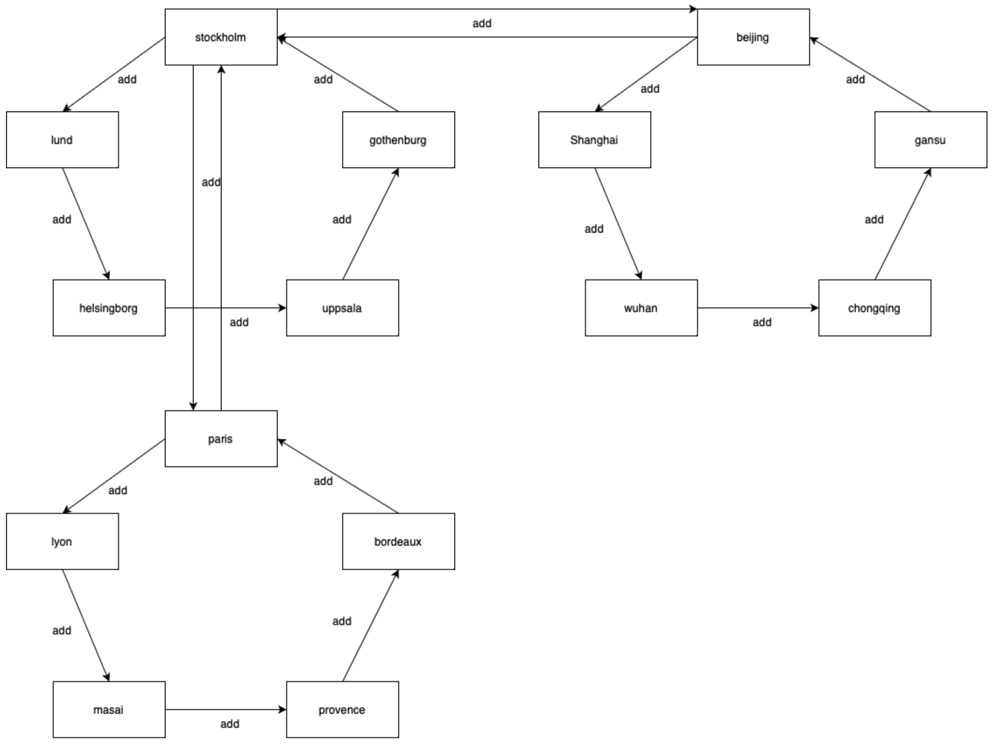


Figure 2: Routy Network

5 Conclusions

Routy works well and connects successfully with other networks' routers.

It looked quite ambiguous when I was designing the basic modules like map and interfaces for the routing protocol. But after the entire development, everything became smooth and every layer of the implementation ended up being clearly understood.

My functional programming skills have also been practiced. I became familiar with the functions in the `List` module like how to use `foldl`, `map` and try to find an elegant way to solve problems.