HW2: Routy - A Routing Network

Jian Wen

September 20, 2023

1 Introduction

The complexity of this assignment is much higher compared with the previous one. In this assignment, I recaptured some basic knowledge about Dijkstra algorithm and its implementation in a routing network. This algorithm is also deployed in routing protocols like OSPF.

This assignment consists of realizing the update mechanism for map, routing table, interfaces and so on, as well as the communication between routers. One thing to be learnt from this assignment is that, in a distributed system, I have to realize how problem is solved on each nodes, while through nodes' communicating with each other the whole system is complete. I think the core concept of a distributed system is to achieve a greater goal by working on each nodes, which is something like divide-and-conquer.

2 Main problems and solutions

2.1 The map

Map is a database storing all the directly reachable nodes of a known node. The update() and reachable() can all be achieved easily using lists:keystore() and lists:keyfind(). For all_nodes(), problem is more tricky because duplicate entries should be removed. By using lists:foldl() and list comprehension, I am able to tell whether an entry is already in the output before adding it to the output.

Implementation of all_nodes():

```
all_nodes(Map) ->
  lists:foldl(fun(Elem, AccIn) -> accumulator(Elem, AccIn) end, [], Map).
  % Use lists:foldl as accumulator to get all the elements

accumulator({Node, Links}, Acc) ->
  EntryList = [Elem || Elem <- [Node | Links], not lists:member(Elem, Acc)],
  % Use list comprehension to remove duplicate entry
  lists:append(EntryList, Acc).
```

2.2 Dijkstra

The dijkstra module is where the core function of this assignment is implemented. In this part, a database Table will be iterated and updated. Table is used to store the gateway (next node) to reach a known node (destination). Sorted list is used to rank the distance and gateway to known nodes.

The entry(), replace(), update() can be achieved using lists:keyfind(), keyreplace(), sort(). iterate() is the most important one, because it will update the routing table using information about the map. For each node, it retrieve all the adjacent nodes and use the distance stored in Sorted to update the rest of the list and added to the table. Since distance is sorted by increasing order in Sorted, then it is guaranteed that the path chosen for the router's routing table is always the shortest one.

In this part, I have encountered with some problem in iterate(). At first, I updated Sorted initializing with the complete Sorted without deleting the first entry after using, and this cause a problem: when there are multiple gateways at a node, the routing will fail. It cost me some time to debug the program and finally locate the problem.

Implementation of iterate():

```
iterate([], _, Table) ->
  Table;
iterate([{_, inf, _} | _], _, Table) ->
  Table;
iterate([{Node, N, Gateway} | RestSorted], Map, Table) ->
  case map:reachable(Node, Map) of
  % Retrieve all the adjacent nodes (neighbors) from map
    [] ->  % If there is nothing, go on with the rest nodes
      iterate(RestSorted, Map, [{Node, Gateway} | Table]);
    Reachable ->  % For each adjacent nodes, use itself update the rest nodes
      NewSorted = lists:foldl(
            fun(Elem, AccIn) -> update(Elem, N+1, Gateway, AccIn) end,
            RestSorted,
            Reachable),
      iterate(NewSorted, Map, [{Node, Gateway} | Table])
      % Go on with the rest nodes
  end.
```

2.3 Interfaces

This part is quite easy since most usable functions, like lists:keyfind(), keystore(), keydelete(), are already used in the precious modules. Two new functions are used here: lists:map(), foreach(). lists:map() is used to return a mapping of a list and foreach() is used to take some action to each element, such as sending messages.

One thing to notice is that there are really a lot of lists in this project, the best way to remember the structure and content of a list is to understand it.

2.4 The history

In a distributed system, each node will have to deal with messages and interactions from other nodes. For a router, there have to be a monitor to keep a record of processed updates. Only when a NEW update arrived will it be forwarded to the others. This is to avoid eternal forwarding in case of loops in the network.

To deal with this, each node will be bound to a counter N as message sequence number. Only the message with number higher than the counter will be processed.

Implementation of update():

```
update(Node, N, History) ->
% Used to identify whether a received update is new or already known.
  case lists:keyfind(Node, 1, History) of
    {Node, OldNum} ->
      if
        N > OldNum ->  % If new (but known before), update History.
          Updated = lists:keyreplace(Node, 1, History, {Node, N}),
          {new, Updated};
        true ->  % Otherwise, return old.
          old
```

```
      end;
    false ->  % If new (never known before), update History.
      {new, [{Node, 0} | History]}
  end.
```

2.5 The router

Most of codes of routy is given, however, I modified some initialization so that I can customize the Pid to build a network in an easier way. When initialize a new interface, itself is added the list to avoid sending out.

```
...
start(Pid, Name) ->
% Modified start and init to customize Pid
  % {r1, 'sweden@192.168.50.59'}, stockholm
  % <process id> Pid = {r1, 'sweden@192.168.50.59'},
  % <router register> Reg = r1,
  % <symbolic name> Name = stockholm
  {Reg, _} = Pid,
  register(Reg, spawn(fun() -> init(Name, Pid) end)).

init(Name, Pid) ->
  Intf = intf:new(),
  Ref = erlang:monitor(process, Pid),
  NewIntf = intf:add(Name, Ref, Pid, Intf),
  % Initialize with itself added as interface
  ...
```

3 Test and evaluation

An automatic test file to construct a network and get router status is implemented. The test file will start 5 nodes representing 5 cities: stockholm, lund, uppsala, malmo, goteborg. Then in intfconf(), interfaces will be configured for each node, so that a network is built. To test the routing algorithm under different schemes, I have constructed 2 different maps, marked as Graph A and Graph B in the Figure 1.

The test start with running init(), confintf(), broadcast() and finally update(). The status of each router can be retrieved by calling getstatus().

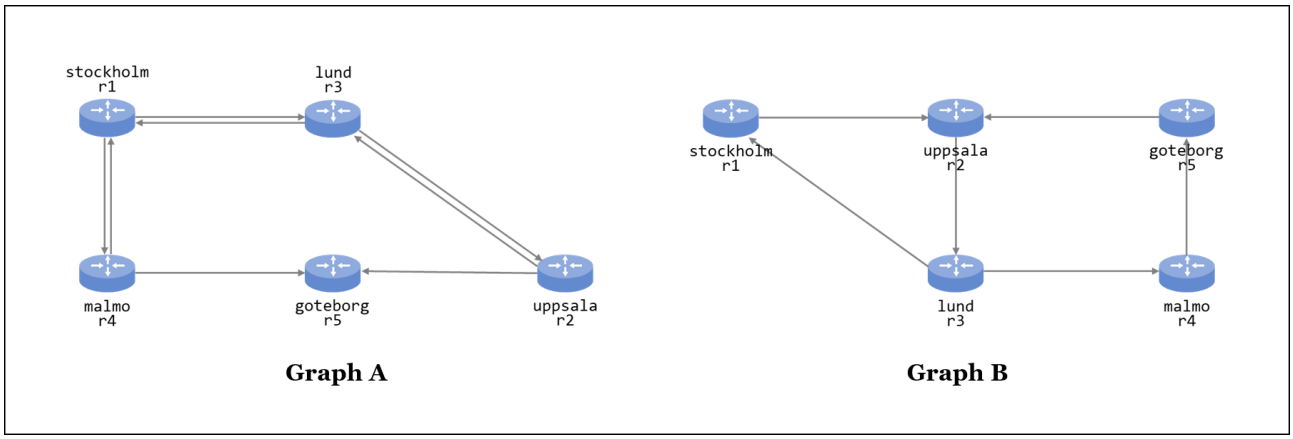


Figure 1: Maps

3.1 Test for Graph A

Graph A is a ring consists of some bi-directional links and unidirectional links. I tried to test by sending messages from different pairs of routers and examining the route. Some io outputs helped me better understand what is going on at each router. I have also looked into Table, Map by receiving status on any router. The result indicates that the algorithm performs well.

Some example outputs are shown in Figure 2, 3.

```
(sweden@130.229.135.154)6> r1 ! {send, goteborg, 'Hello world!'}.
stockholm: routing message ('Hello world!') from stockholm
{send,goteborg,'Hello world!'}
stockholm: sent to {r4,'sweden@130.229.135.154'}
malmo: routing message ('Hello world!') from stockholm
malmo: sent to {r5,'sweden@130.229.135.154'}
goteborg: received message 'Hello world!' from stockholm
```

Figure 2: output: sending from r1 to r5

```
(sweden@130.229.135.154)7> test:getstatus(r2).
 Node = uppsala
 N = 1
 Hist = [{malmo,0},{stockholm,0},{lund,0},{uppsala,0}]
 Intf = [{uppsala,#Ref<0.1610064.630980613.16562>,{r2,'sweden@130.229.135.154'}},{lund,#Ref<0.1610064.630980613.16576>,
{r3,'sweden@130.229.135.154'}},{goteborg,#Ref<0.1610064.630980613.16577>,{r5,'sweden@130.229.135.154'}}]
 Table = [{malmo,lund},{lund,lund},{stockholm,lund},{lund,lund},{uppsala,uppsala},{goteborg,goteborg}]
 Map = [{lund,[lund,uppsala,stockholm]},{stockholm,[stockholm,lund,malmo]},{malmo,[malmo,stockholm,goteborg]}]
ok
```

Figure 3: output: receiving status from r2

3.2 Test for Graph B

Graph B consists of multiple rings with unidirectional links. A test using similar method as Graph A is carried out. The result indicates that the algorithm performs well.

Some example outputs are shown in Figure 4, 5.

```
(sweden@130.229.135.154)6> r1 ! {send, malmo, 'Hello world again!'}.
stockholm: routing message ('Hello world again!') from stockholm
{send,malmo,'Hello world again!'}
stockholm: sent to {r2,'sweden@130.229.135.154'}
uppsala: routing message ('Hello world again!') from stockholm
uppsala: sent to {r3,'sweden@130.229.135.154'}
lund: routing message ('Hello world again!') from stockholm
lund: sent to {r4,'sweden@130.229.135.154'}
malmo: received message 'Hello world again!' from stockholm
```

Figure 4: output: sending from r1 to r4

```
(sweden@130.229.135.154)7> test:getstatus(r3).
 Node = lund
 N = 1
 Hist = [{malmo,0},{goteborg,0},{stockholm,0},{uppsala,0},{lund,0}]
 Intf = [{lund,#Ref<0.1790056762.3854041090.140022>,{r3,'sweden@130.229.135.154'}},{stockholm,#Ref<0.1790056762.3854041090.140035>,
{r1,'sweden@130.229.135.154'}},{malmo,#Ref<0.1790056762.3854041090.140036>,{r4,'sweden@130.229.135.154'}}]
 Table = [{uppsala,stockholm},{goteborg,malmo},{goteborg,malmo},{malmo,malmo},{stockholm,stockholm},{uppsala,stockholm},{stockholm,
stockholm},{lund,lund},{malmo,malmo}]
 Map = [{uppsala,[uppsala,lund]},{stockholm,[stockholm,uppsala]},{goteborg,[goteborg,uppsala]},{malmo,[malmo,goteborg]}]
ok
```

Figure 5: output: receiving status from r3

4 Optional test: The world

I grouped with Zeyang Lyu and Heng Kang to form a global network for Teyvat (a virtual world in Genshin Impact). Each of us acted as one region with 5 routers, so we build a Teyvat network with 15 routers. The network topology is shown in Figure 6.

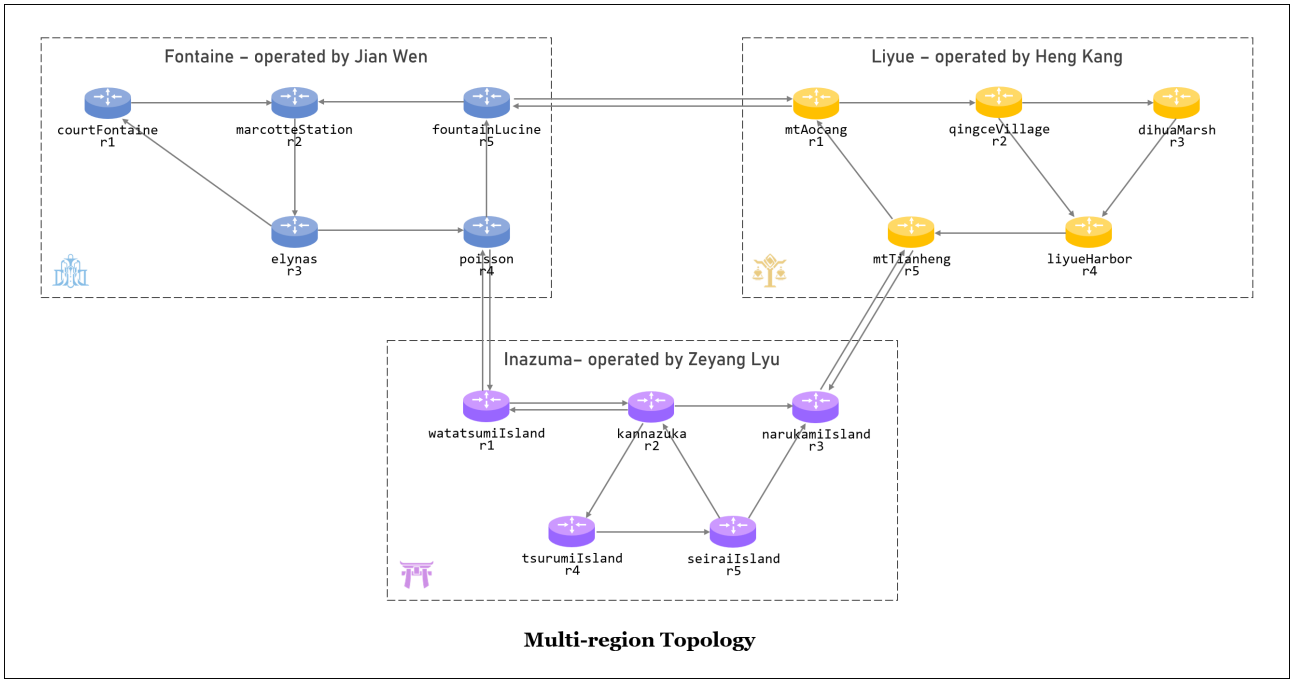


Figure 6: Teyvat network topology

4.1 Connecting with each other

Connecting the networks requires all nodes running under the same sub-net, so our networks are running under the same WiFi. Then we have connect our border router with the outside network according to the structure shown in Figure 6. After successfully configuring the interfaces, we broadcast and update routing table. Now, the whole network is operational! Result of this stage is shown in Figure 7.

4.2 Test with our network

We have tested our Teyvat network with multiple send-and-receive tests. For each test, we examined the IO output in each node's terminal and check whether the path is correct. For my network, I have tested with sending, receiving and forwarding messages. All results, as shown in Figure 8, is as expected.

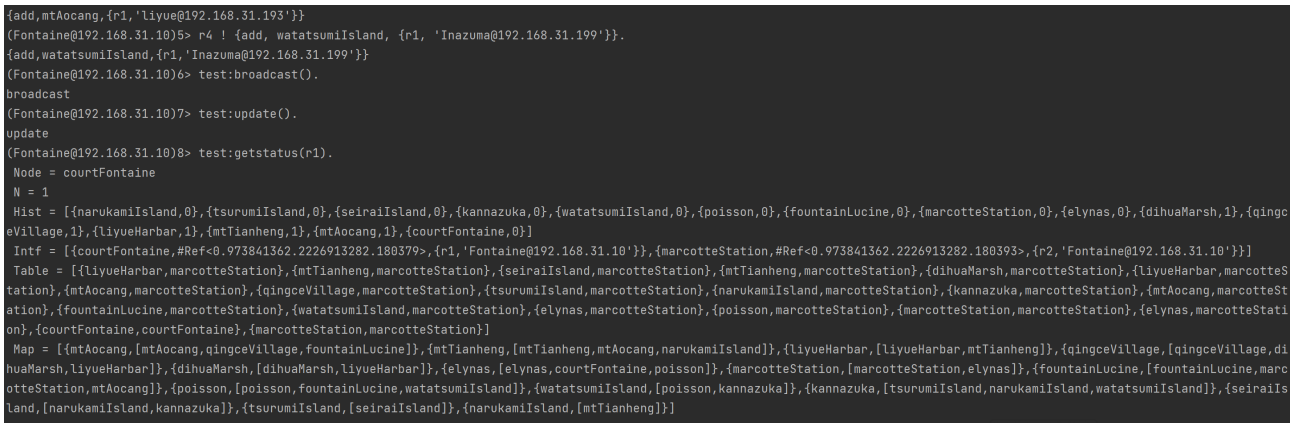


Figure 7: network initialization and result

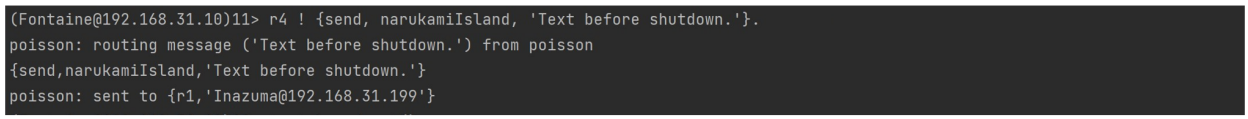
Figure 8: network test results

4.3 Shutdown test

Our final step is to test the network behavior before and after we shutdown certain router. The test case with r2 in Inazuma@192.168.31.199 shutdown is given in this report.

According to the network topology, when routing a message from poisson (r4 in Fontaine) to narukamiIsland (r3 in Inazuma), the path is poisson - watatsumiIsland - kannazuka - narukamiIsland. This process is shown in Figure 9.

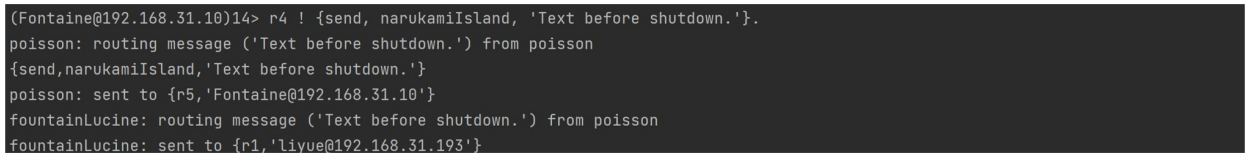
Then we close the router at kannazuka (r2 in Inazuma@192.168.31.199), and broadcast and update again. The routing table are expected to notice the change and choose a new path going through liyue@192.168.31.193. The test result is shown in Figure 10. The result shows that our network can react to node changes by broadcasting and updating the information.



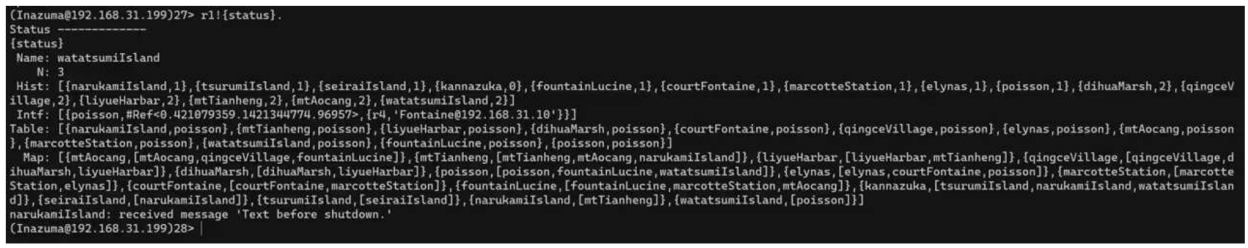
IO output at *Fontaine@192.168.31.10*

IO output at *Inazuma@192.168.31.199*

Figure 9: test before shutdown



IO output at *Fontaine@192.168.31.10*



IO output at *Inazuma@192.168.31.199*

Figure 10: test after shutdown

5 Conclusion

In this assignment, I have implemented a more complex distributed system with multiple nodes. During programming and testing, I have learnt the concept of designing such a distributed system, that focusing on the functions of each nodes and achieve a greater goal by interacting with the other nodes. Besides, this assignment provides me another view looking at the Dijkstra algorithm. In the previous implementation using C++, things are quite different because I am basically implementing everything on a single node (like a centralized server or computer). However, in this project each nodes is updating the map and routing table based on updates from other nodes thanks to Erlang's concurrent oriented and functional features. After this assignment, Erlang development seems to be much more clear.