Report 2: Routy - a routing network

Emile Le Gallic

September 18, 2024

1 Introduction

The main topic of this seminar is the implementation of a link-state routing protocol in a distributed system, using Erlang. This type of protocol is crucial in network systems such as OSPF (Open Shortest Path First), a widely used routing protocol for Internet routers.

During this seminar, we focus on several key concepts in distributed systems:

- **Link-state routing**: How routing tables are built based on the topology of the network, and how routers exchange information to construct a global view of the network.
- **Consistency in distributed systems**: How a consistent view of the network is maintained across multiple routers, ensuring correct packet delivery.
- **Network failures and fault tolerance**: Understanding and handling scenarios where routers or links fail, and reflecting on how to recover and reroute traffic to maintain connectivity.

This assignment emphasizes the importance of efficient communication and fault tolerance in distributed systems, which is critical in real-world networks where data must be reliably transmitted even under challenging conditions.

2 Main problems and solutions

During the development of the solution, one of the key challenges was efficiently processing lists to implement the functions required for the router's operations. I utilized the `foldl` function extensively, as it allowed me to accumulate values while traversing the list, which was crucial for processing large sets of data, like in the `table/2` and `all_nodes/1` functions. For example, in the `table/2` function, I used `foldl` to iterate over all

nodes, checking if they were gateways and initializing their routing values accordingly. Similarly, in `all_nodes/1`, `foldl` was essential in building a complete list of nodes, even those without outgoing links, by checking each node and its connections.

The implementation of Dijkstra was fairly straightforward because the required components (such as maps, lists, and helper functions) were already in place. The primary task was combining these "building blocks" into more complex operations like constructing the routing table and handling the network map.

I also took time to deeply understand the link-state routing protocol, particularly how routers communicate and spread link-state messages. This required examining how messages are tagged with counters and how routers determine whether a received message is old or new. Understanding this process was key to implementing the proper workflow for spreading information across the network, ensuring the protocol worked as expected. Overall, the problem-solving approach relied heavily on `foldl` for list processing and careful attention to the network protocol's message-handling mechanisms.

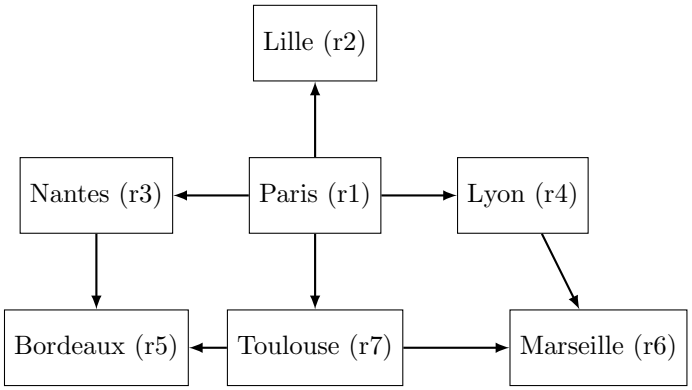
3 Evaluation

3.1 A first network

In this simulation, I have implemented a network of routers representing cities in France. The network consists of at least five routers, with each router communicating its links to the others and updating its routing table using the Dijkstra algorithm. The following cities are represented as routers:

- Paris
- Lille
- Lyon
- Nantes
- Bordeaux
- Toulouse
- Marseille

And the links are as follows :



The purpose of this test is to demonstrate how the routing tables are computed and updated dynamically when a router is shut down, showing the robustness of the network's path-finding algorithm.

To start the test, execute the following command in your Erlang console to initialize the nodes and set up the network:

```
erl -name france -setcookie routy -connect_all false
```

Then compile the test module and set up the network by running:

```
c(test).
test:setup('france@{YOUR_HOST}').
```

where {YOUR_HOST} should be replaced with the hostname of your Erlang console.

3.1.1 Network Initialization

During initialization, each router will broadcast its links to its neighbors. This process is necessary to ensure that all routers have complete knowledge of the network topology. Here is an example of the link updates exchanged between routers:

```
paris: adding lille
lille: adding paris
lyon: adding paris
nantes: adding paris
bordeaux: adding nantes
toulouse: adding bordeaux
marseille: adding toulouse
paris: adding lyon
lyon: adding marseille
nantes: adding bordeaux
bordeaux: adding toulouse
toulouse: adding marseille
marseille: adding lyon
paris: adding nantes
paris: adding toulouse
```

After all the links are established, the routers update their routing tables using the Dijkstra algorithm.

3.1.2 Routing Table Update

Each router maintains a routing table that is updated dynamically based on the received link information. Here is an example of the routing table update of the Paris router:

```
(france@MBP-Emile.local)3> r1 ! {status, self()}.
paris: received status from <0.91.0>
{status,<0.91.0>}
******
status of paris:
N: 1
Hist: [{toulouse,0},{marseille,0},{bordeaux,0},{paris,0},{lyon,0},{nantes
    ,0},{lille,0}]
Intf: [{toulouse,#Ref<0.851052405.2216689665.148593>,{r7,'france@MBP-Emile
    .local'}},{nantes,#Ref<0.851052405.2216689665.148589>,{r3,'france@MBP-
    Emile.local'}},{lyon,#Ref<0.851052405.2216689665.148576>,{r4,'
    france@MBP-Emile.local'}},{lille,#Ref<0.851052405.2216689665.148550>,{
    r2,'france@MBP-Emile.local'}}]
Table: [{marseille,toulouse},{bordeaux,toulouse},{paris,lyon},{lille,lille
    },{nantes,nantes},{lyon,lyon},{toulouse,toulouse}]
Map: [{toulouse,[marseille,bordeaux]},{marseille,[lyon,toulouse]},{
    bordeaux,[toulouse,nantes]},{paris,[toulouse,nantes,lyon,lille]},{lyon
    ,[marseille,paris]},{nantes,[bordeaux,paris]},{lille,[paris]}]
******
```

The routing table of Paris reflects paths to all the routers in the network. Messages can now be routed efficiently across the network.

Example :

```
(france@MBP-Emile.local)4> r6 ! {send, nantes, "Bonjour"}.
marseille: routing message ([66,111,110,106,111,117,114])
{send,nantes,"Bonjour"}
lyon: routing message ([66,111,110,106,111,117,114])
paris: routing message ([66,111,110,106,111,117,114])
nantes: received message [66,111,110,106,111,117,114]
```

3.1.3 Simulating Router Shutdown

One key part of the test is to prove that the network can handle the shutdown of a router and recompute the shortest paths. This is done by stopping one of the routers, in this case, Lyon (router 4):

```
routy:stop(r4).
```

The other routers receive this shutdown information and update their routing tables accordingly. Below is an example of how the network reacts to the shutdown of Lyon:

```
(france@MBP-Emile.local)5> routy:stop(r4).
marseille: exit received from lyon
paris: exit received from lyon
paris: updating table
toulouse: updating table
bordeaux: updating table
nantes: updating table
lille: updating table
```

The network successfully recomputes the new shortest paths and the routers continue to route messages without issues.

3.1.4 Routing a Message

After the network has stabilized, we can route messages from one router to another. Below is an example where Marseille sends a message to Nantes:

```
(france@MBP-Emile.local)6> r6 ! {send, nantes, "Bonjour"}.
marseille: routing message ([66,111,110,106,111,117,114])
lyon: routing message ([66,111,110,106,111,117,114])
paris: routing message ([66,111,110,106,111,117,114])
nantes: received message [66,111,110,106,111,117,114]
```

As shown, the message successfully routes through multiple routers and reaches its destination (Nantes).

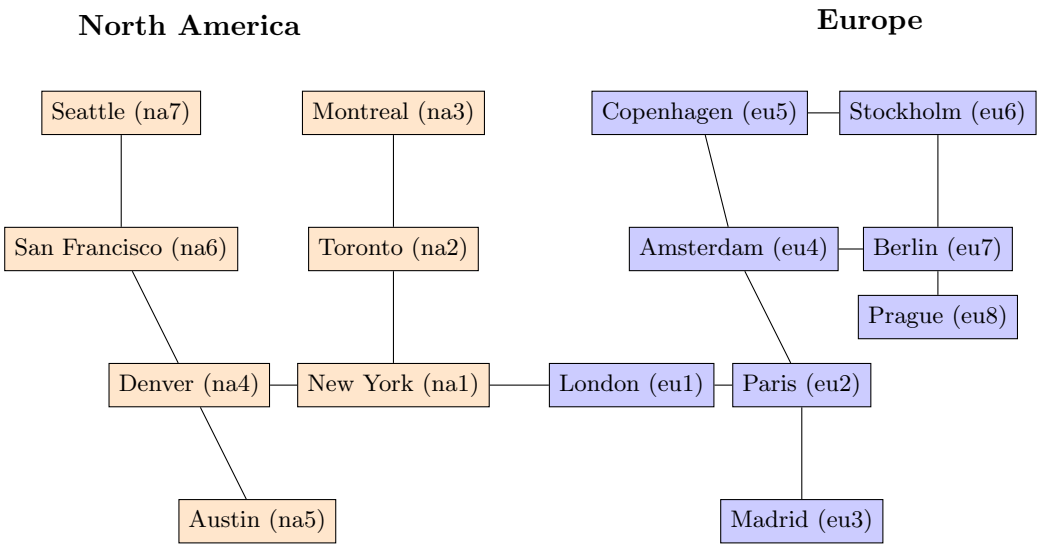
3.1.5 Importance of Sleep Between Phases

It is important to note that sleeps are required between certain phases of the test, such as after broadcasting and before updating the routing tables. This ensures that all routers have received the necessary link information before attempting to compute new paths. Without these sleeps, race conditions may occur, causing incomplete link information and inaccurate routing tables.

3.2 Bonus part : connecting North America and Europe

3.2.1 Network Design

In collaboration with a classmate, we constructed two networks: one for North America and one for Europe. In each region, we added different routers corresponding to various cities. Below is the network diagram:



3.2.2 Connecting the two regions

We created two new test programs to set up the respective networks of the two regions on our individual computers. These programs are `test_eu.erl` and `test_na.erl`.

In each of these programs, we had a function:

```
setup_remote(RemoteHost, RemoteCity, RemoteRegister) ->
   na1 ! {add, RemoteCity, {RemoteRegister, RemoteHost}}.
```

This function allowed us to connect our computers.
The procedure was as follows:

1. We launched the Erlang terminal using the command:

   ```
   erl -name eu -setcookie routy -connect_all false
   ```

2. We noted the host displayed in the console, for example:

   ```
   (eu@n141-p26.eduroam.kth.se)1>
   ```

   In this case, the host is `n141-p26.eduroam.kth.se`.

3. The other person, who was in charge of the North American region, would then run:

   ```
   test_na:setup_remote('eu@n141-p26.eduroam.kth.se', london, eu1).
   ```

   This established the connection between the regions. Afterward, we could broadcast and update to synchronize the tables.

We tested sending messages between regions, deliberately cutting off a router, and then refreshing the tables. All of these tests worked successfully.

4 Conclusions

The problem helped me gain a deeper understanding of link-state routing protocols, particularly how routers exchange and update network information to maintain accurate routing tables. I learned how networks can dynamically adapt to router failures, ensuring continuous communication. This assignment helped to understand how real-world distributed systems manage routing and fault tolerance efficiently.