Report 2: Routy

Fabrizio Demaria

September 24, 2014

1 Introduction

The project is about implementing a link-state routing protocol in Erlang, adopting directional links between routers. The protocol allows to build and check routing tables to determine which gateway is best to forward a message, if any. Every node in the network will be given the name of a city to easily reproduce network tests.

2 Main problems and solutions

This laboratory required better programming skills in Erlang with respect to the "Rudy" exercise. Moreover, the Erlang documentation had to be studied in order to implement the required modules. Apart from the coding difficulties, another tricky part was to understand the relationships among the various components of the network protocol, in particular how to use the sorted list in order to update the routing tables. The function *iterate* in the *dijkstra.erl* module was the hardest. Everything became clear when testing the correct final version of the code, also thanks to many printings adopted for debugging purposes.

3 Evaluation

When performing our tests, first it is required to update the map of the network by broadcasting link-state messages (operation that we do manually each time a new link is added) and second the routing table is updated through the *update* procedure. When all the routing tables are updated it is possible to send messages through the network. If a node is not reachable the message is simply dropped in this basic implementation.

Our test wants to connect three cities: **stockholm**, **gothenburg** and **malmo**. After having initialized the servers, each will have a configuration like this:

```
Status -------------
 name: stockholm
    n: 0
 msgs: [{stockholm,infinite}]
 intf: []
table: []
  map: []
```

Next we adopt the *add* messages to link **stockholm** with **gothenburg**, **gothenburg** with **malmo** and **gothenburg** with **stockholm** (all links are mono-directional). The simple network is shown in Figure 1

Figure 1: Network built for the test.

If now we broadcast a link-state message from **stockholm** we obtain the following map in **malmo** router:

```
map: [{stockholm,[gothenburg]}]
```

If we also broadcast from **gothenburg** the map will be completed (broadcasting from **malmo** wouldn't modify the map since we haven't added any interface to **malmo**):

```
map: [{gothenburg,[stockholm,malmo]},{stockholm,[gothenburg]}]
```

It is interesting to show the status of **gothenburg** at this point:

```
Status -------------
 name: gothenburg
    n: 1
 msgs: [{stockholm,1},{gothenburg,infinite}]
 intf: [{stockholm,#Ref<0.0.0.98>,{stockholm,'sweden@83.179.39.185'}},
        {malmo,#Ref<0.0.0.92>,{malmo,'sweden@83.179.39.185'}}]
table: []
  map: [{stockholm,[gothenburg]}]
```

The map of **gothenburg** doesn't show the links from **gotheburg** itself (as expected), but those are retrievable as "direct" interfaces (section *intf*).

It is clear the importance of the *hist.erl* module: without keeping track of old messages, due to the double link between **stockholm** and **gothenburg** the link-state message from any of the two cities would have been sent back and forth forever.

Now we can build the routing tables for the three cities through the *update* command. Here it is the result table for **stockholm**:

```
table: [{malmo,gothenburg},{stockholm,gothenburg},{gothenburg,gothenburg}]
```

The second tuple **{stockholm,gothenburg}** says that to send a message to **stockholm** from **stockholm** it is necessary to go through **gothenburg**. This is not correct but the fact is that this tuple is never used anyway: as soon as the name of the message's recipient matches the name of the router (i.e. the message has reached the destination), the routing table is not even scanned: sending a message from **stockholm** to **stockholm** doesn't involve routing in

any case since we use a special message entry to handle messages directed to the router itself.

Similar considerations can be made for the updated routing table of **gothenburg**:

```
table: [{gothenburg,stockholm},{malmo,malmo},{stockholm,stockholm}]
```

Everything matches the topology designed at the beginning. If we now send a message from **stockholm** to **malmo** the result is the following:

```
( ... )30> stockholm ! {send, malmo, "Hello World!"}.
stockholm: routing message (Hello World!) from stockholm to malmo
stockholm: forward to gothenburg
gothenburg: routing message (Hello World!) from stockholm to malmo
gothenburg: forward to malmo
malmo: received message (Hello World!) from stockholm
```

4 Conclusions

This laboratory was important to improve Erlang coding capabilities and it required to look at Erlang's documentation in order to be able to manage lists properly. The OSPF-related network protocol developed was important since implementing every component step by step allowed to deeply understand the routing process in all its parts, including the various problems like the cyclic paths and the management of disconnected/unreachable nodes.