Report 2: Routy

Thomas Galliker

September 14, 2011

1 Introduction

In this exercise our task was to develop a very basic router software using a link-state routing algorithm. The aim of this exercise is not only to get a good understanding of how link-state routing protocols like OSPF work, but also to get an idea of how a well-scaling distributed algorithm looks like.

2 Main problems and solutions

This exercise was pretty difficult to solve. I mostly understood what was asked, but as a beginner in Erlang it was difficult to bring all requirements to code. The first part with the network map (map.erl) was the easiest one, what does not mean that I haven't spent much time on it. The most time was used to programm Dijkstra's algorithm (dijkstra.erl). This module contains functions to maintain the routing table. It is the heart of the router.

Erlang List Functions
The lists library of Erlang provided a good source of functions to search, delete and sort keys of tupel lists. A very special function which I have found and used was the fold function "foldl". This function allows to call a certain function for every element in a provided list. The appropriate use of this function can save a lot of time. With all respects to Erlang: The online documentation would be much more useful if there was example code for each function. Most runtime errors happened due to wrongly passed arguments.

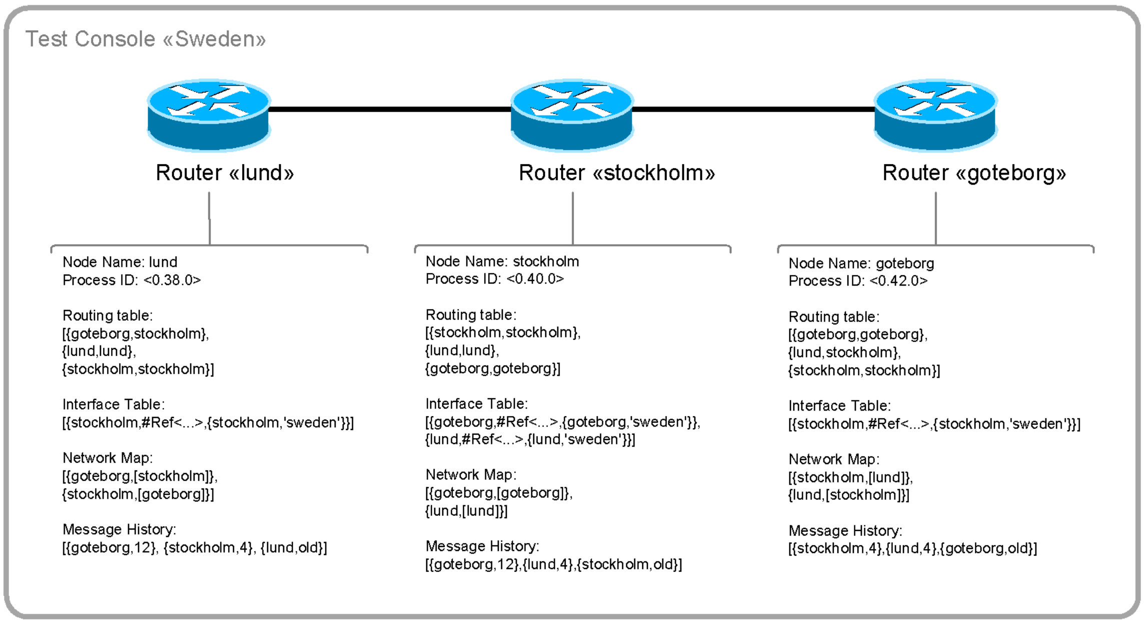
Debugging Erlang
Nobody is perfect in programming. But everyone should get appropriate tools to debug his/her code in an efficient way. This is obviously not the case with Erlang. Troubleshooting Erlang code is a very annoying task. Thus, I have injected lots of "io:format" calls to get out status information and finally get to the point where my code needs correction. Is this the way how professional Erlang developers work? Hopefully not...

Confusion between Network Map and Routing Table
There were confusions about the ambiguity between the network map and the routing table. Hope this explanation is correct:
- A network map provides an overview of all available networks in a particular domain. Each entry consists of two attributes: The target network and the gateway with the shortest path directing to this network.
- A routing table is used to make the decision to which gateway a particular message must be forwarded to reach its target network.

3 Tests & Evaluation

After having finished the main tasks of this exercise, I tried to find simple scenarios in which the code can be checked for errors.

- **Test Scenario 1:** The first test case consists of three routers. Two routers named lund and goteborg are directly connected to a third router named stockholm. I wanted to configure the routers in a way to make it possible to send messages from lund via stockholm to goteborg and vice versa. (Refer to figure 1). The setup instruction can be found in the appendix of this report.



```
Node Name: lund
Process ID: <0.38.0>

Routing table:
[{goteborg,stockholm},
{lund,lund},
{stockholm,stockholm}]

Interface Table:
[{stockholm,#Ref<...>,{stockholm,'sweden'}}]

Network Map:
[{goteborg,[stockholm]},
{stockholm,[goteborg]}]

Message History:
[{goteborg,12}, {stockholm,4}, {lund,old}]
```

```
Node Name: stockholm
Process ID: <0.40.0>

Routing table:
[{stockholm,stockholm},
{lund,lund},
{goteborg,goteborg}]

Interface Table:
[{goteborg,#Ref<...>,{goteborg,'sweden'}},
{lund,#Ref<...>,{lund,'sweden'}}]

Network Map:
[{goteborg,[goteborg]},
{lund,[lund]}]

Message History:
[{goteborg,12},{lund,4},{stockholm,old}]
```

```
Node Name: goteborg
Process ID: <0.42.0>

Routing table:
[{goteborg,goteborg},
{lund,stockholm},
{stockholm,stockholm}]

Interface Table:
[{stockholm,#Ref<...>,{stockholm,'sweden'}}]

Network Map:
[{stockholm,[lund]},
{lund,[stockholm]}]

Message History:
[{stockholm,4},{lund,4},{goteborg,old}]
```

Figure 1: Test scenario 1 with three routers.

Conclusions on Test 1:
- While setting up this test case I experience several problems with my router software. My biggest problem was the lack of information that is displayed to find errors caused by missing route entries or wrong network maps and so on. Therefore I introduced lots of io:format console outputs, including the new "info" command which was used to display the configuration of the router.
- There was also a very strange behavior in my router software. Routers obviously have broadcasted wrong routing information. There were situations where a particular router was told to reach itself via another router . This caused wrong entries in the routing table. For example the router stockholm has had an entry {stockholm,goteborg} in its table, which means that messages destined to stockholm must be sent to gateway goteborg. This is definitely wrong! To my luck this mistake doesn't play a role since the router recognises its own messages before they're forwarded to the routing process.
- To eliminate this errors we could either add a local loop back entry manually...
`stockholm ! {add, stockholm, {stockholm, 'sweden@130.229.190.132'}}.`
... or we could inject some code to do it automatically during the init process of the router:
`self() ! {add, Name, self()},`
- Finally the message could be sent from lund to goteborg and v.v.

- **Test Scenario 2:** This test case is based upon test scenario 1. The plan is to setup six routers on two different computers. On both computers one of the routers holds the role of a border gateway to its peer on the other computer. (Refer to figure 2).

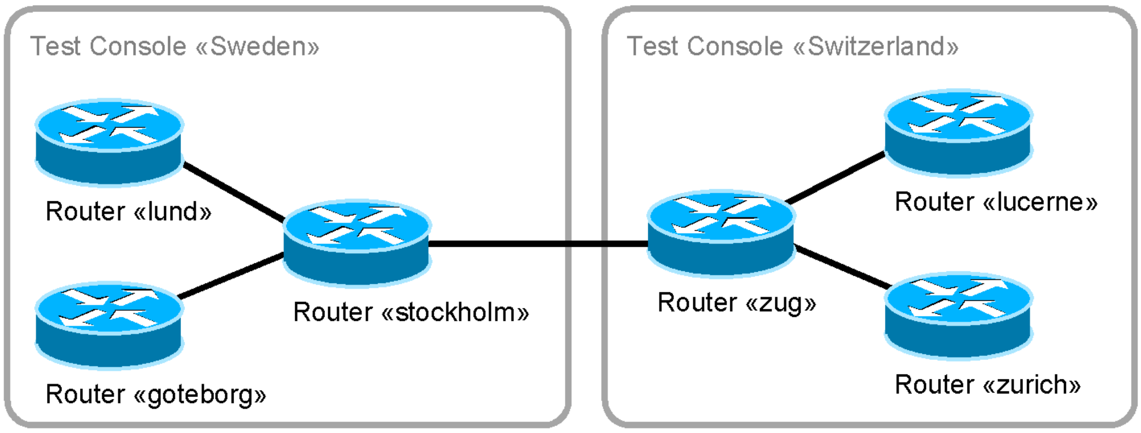


Figure 2: Test scenario 2 with two physically separated domains with three routers each.

Conclusions on Test 1:
- I was surprised how quickly this scenario was up and running. There were just minor issues with missing routes due to omitted update commands... (human error).
- As a consequence of this test I decided to complement my router with a management process which is responsible for automatically broadcasting routing information to neighbors and applying routing updates received from them.

- **Test Scenario 3:** This test was invented by one of my fellows. We tried to build a ring of three routers. Each of them only knew one router so that they had formed a logical ring. Means, every router just had one single gateway. All messages are sent towards the same "direction" as if it was a ring. (Refer to figure 3).

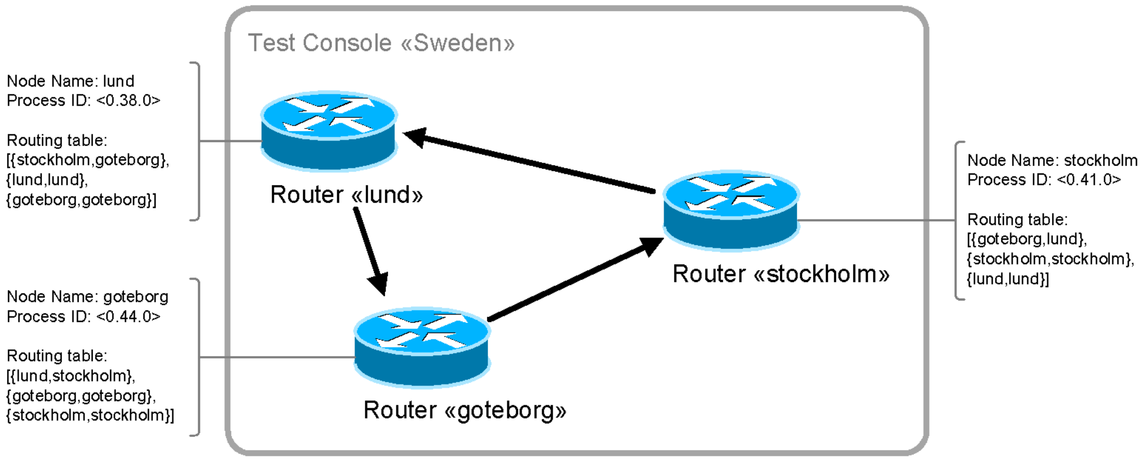


Figure 3: Test scenario 3 illustrates the ability of directing traffic.

Conclusions on Test 1:
- This scenario illustrates that (incorrect) interface configurations can cause traffic bottlenecks. Messages are may not forwarded to the actually shortest path.
- It is possible to establish circular unidirectional communication networks. But what for?

4 Conclusions

As already stated, this exercise was very difficult. I finally brought it to run but just with a huge effort and many late night hours spent. The harder the exercise the better the learning effects. This was definitely the case with this task. I thought I knew how the OSPF principle is used to maintain scalable IP networks since I've already worked through the Cisco CCNA courses but only with the implementation of these algorithms could I get an in-depth understanding of how things really work. We tried to build up fancy multi-router scenarios during the Tuesday's lab as you can see in the sections above. This was fun! Each scenario revealed several minor bugs that were (hopefully completely) solved until Thursday's lab.

What haven't talked about is security. Since we don't make use of secured channels nor an authentication mechanism between router nodes it is of course easy for an attacker to inject wrong routing information that manipulates routing tables.

I have to confess that these kinds of exercises have brought me a bit closer to the mysterious Erlang programming language. Nevertheless it was a hard piece of work.

5 Links

http://www.erlang.org/doc/man/lists.html
http://en.wikipedia.org/wiki/Open_Shortest_Path_First
http://en.wikipedia.org/wiki/Dijkstra's_algorithm
http://bestpractices.wikia.com/wiki/OSPF
http://www.thomasgalliker.ch

6 APPENDIX

Setup of Test Scenario 1:

```
%% Start a new command prompt named ”Sweden@your-current-IP”.
erl -name sweden@130.229.190.132 -setcookie routy -connect_all false

%% Start three routers: lund, stockholm and goteborg.
routy:start(lund).
routy:start(stockholm).
routy:start(goteborg).

%% Add interfaces to each of the routers according to the test map.
lund ! {add, stockholm, {stockholm, ’sweden@130.229.190.132’}}.
stockholm ! {add, lund, {lund, ’sweden@130.229.190.132’}}.
stockholm ! {add, goteborg, {goteborg, ’sweden@130.229.190.132’}}.
goteborg ! {add, stockholm, {stockholm, ’sweden@130.229.190.132’}}.

%% Broadcast routing information to all neighbors. All routers should now receive
%% link-state messages and rebuild their network maps.
lund ! broadcast.
stockholm ! broadcast.
goteborg ! broadcast.

%% After having rebuilt the network maps, we can issue an update on all routers
%% to let the dijkstra algorithm generate new routing tables.
stockholm ! update.
lund ! update.
goteborg ! update.

%% Issue the info command to consult the router configuration.
lund ! info.
stockholm ! info.
goteborg ! info.

%% Finally try to send a message from lund via stockholm to goteborg and vice versa.
lund ! {send, goteborg, ”hello goteborg!”}.
goteborg ! {send, lund, ”hello lund!”}.
```

Setup of Test Scenario 2:

%% Start a new command prompt on the 1st computer and call it "Sweden@<your-IP>".
erl -name sweden@130.229.190.132 -setcookie routy -connect_all false

%% Start three routers: lund, stockholm and goteborg.
routy:start(lund).
routy:start(stockholm).
routy:start(goteborg).

%% Add interfaces to each of the routers according to the test map.
%% The red-marked interface is used to connect to the second computer (see below).
lund ! {add, stockholm, {stockholm, 'sweden@130.229.190.132'}}.
stockholm ! {add, lund, {lund, 'sweden@130.229.190.132'}}.
stockholm ! add, goteborg, {goteborg, 'sweden@130.229.190.132'}}.
stockholm ! {add, zug, {zug, 'switzerland@130.229.118.201'}}.
goteborg ! {add, stockholm, {stockholm, 'sweden@130.229.190.132'}}.

%% Start a new command prompt on the 2nd computer and call it "Switzerland@<your-IP>".
erl -name switzerland@130.229.118.201 -setcookie routy -connect_all false

%% Start three routers: zug, zuerich and luzern.
routy:start(zug).
routy:start(zuerich).
routy:start(luzern).

%% Add interfaces to each of the routers according to the test map.
%% The blue-marked interface is used to connect to the first computer (see above).
luzern ! {add, zug, {zug, 'switzerland@130.229.118.201'}}.
zug ! {add, luzern, {luzern, 'switzerland@130.229.118.201'}}.
zug ! {add, zuerich, {zuerich, 'switzerland@130.229.118.201'}}.
zug ! {add, stockholm, {stockholm, 'sweden@130.229.190.132'}}.
zuerich ! {add, zug, {zug, 'switzerland@130.229.118.201'}}.

Setup of Test Scenario 3:

%% Start a new command prompt named "Sweden@your-current-IP".
erl -name sweden@130.229.190.132 -setcookie routy -connect_all false

routy:start(lund).
routy:start(stockholm).
routy:start(goteborg).

lund ! {add, goteborg, {goteborg, 'sweden@130.229.190.132'}}.
goteborg ! {add, stockholm, {stockholm, 'sweden@130.229.190.132'}}.
stockholm ! {add, lund, {lund, 'sweden@130.229.190.132'}}.