Routy: a small routing protocol

Chuan Su

September 18, 2018

1 Introduction

Link-state routing protocols are one of the main classs of routing protocols used in packet switching networks for computer communications.

Implementing a link-state routing algorithm in general involves major stages :

1. Link state flooding and network topology;
2. Routing table and shorteast path algorithm;

Furthermore, maintaining a consitent view of the network topology for every router node plays an essential role in link-state protocol implementation. This report will present our approach in each stage of the implementation of a link-state routing protocol.

2 Link state flooding and network topology

Link-state message is the medium that link-state router nodes utilize to exchange network connectivity information with their neighbours - directly connected nodes, which allows each router to learn the entire network topology.

A link-state message is composed of:

- The identifier of the publisher node of the link-state message;
- The identifiers of nodes to which the publisher directly connects.
- A sequence number that identifies the message verison.

Each router node periodically broadcast a link-state message to notify every connected node of its network change.

Link-state message broadcasting is illustrated in the code snippet below (Full implementaion can be found in `routy.erl` *module):*

```
%% Name is the identifier of itself, the publisher node;
%% N is the sequene number;
Interfaces = intf:list(Intf), %% Interfaces to all its neighbour nodes.
Message = { links, Name, N, Interfaces },
intf:broadcast(Message, Intf).
```

Upon receiving the link-state message node updates its own network map and propagates the same message to all adjacent nodes.

The code snippetsblow illustrates our approach to handle link-state message:

```
reveive
  {links, Node, N, Links } ->
    %% broadcast the received link-state message to all neighbour nodes.
    intf:broadcast({links, Node, N, Links}, Intf),
    %% update network map.
    NewMap = map:update(Node, Links, Map).
  end
end
```

Reading here, you may have noticed that both sending and receiveing end of link-state message involves interaction with its adjacent nodes. Without each node knowing about its neighbour nodes, neither would publisher nodes be able to publish link-state message nor be able receipiant nodes to update their network map. You may curious about how each router nodes determines the connection betwenn their neighbours.

Neighbour discovery is the first step to get a link state enviroment up and running. Link-state router determins its neighbours it is connected to, over fully working links, through exchanging a *HELLO* message periodically on all of its interfaces. This message contains the router's address. As its neigbouring routers also send *HELLO* message, the router automcatically discovers to which neighbours it is connected.

However in `routy` implementaion, we skips the implementation of *neighbour discovery* algorithm instead provides an interface to enable a node to connect a neighbour node mannualy:

```
receive
  {add, Node, Pid} ->
    Ref = erlang:monitor(process, Pid),
    Intf1 = intf:add(Node, Ref, Pis, Intf).
end
```

Read Computer Networking: Principles, Protocols and Practice for a detailed description of the *Neighbour discovery algorithm.*

Imagin every nodes in the world sends outs link-state messages to each of its connected adjacent nodes. In turn, each recipient node learns the publisher's network map and forwards to every neighbour execpt the one that sent the link-state message. Eventually every nodes will be able to construct an entire network topology of the world!

3 Routing table and shortest path algorithm

As previously mentioned that the second major stage in implementing the ink-state protocol is to construct a routing table to enable nodes to route message to the desired destination node.

In link-state environment each router node independently runs an algorithm over the network map built from link-state messages to determine the shortest path from itself to every other node in the network.

Each router maintains two data structures which are:

1. a list of gateway nodes that are proved to be reachable
2. a list of candidates nodes.

These two data structures are the input to the calucation of the shortest path in link-state network graph.

In `routy`, a sorted list is used to implement the candidate list data structure. The initial nodes of the sorted list are produced from the following two steps:

1. All gateway/neighbour nodes which are directly connected to the node are just added the sorted list with path length setting to zero.
2. All the other nodes from the map are appended to the sorted list with path length setting to infinite.

The snippet below illustrates the approach in building the initial candiate list and our interation process:

```
table(Gateways, Map) ->
  %% build the candidate node list
  Nodes = map:all_node(Map).

  NonNeighbourEntries = lists:map(fun(Node) ->
     {Node, inf, unknown } end, Nodes -- Gateways),
  NeighbourEntries = lists:map(fun(Gateway) ->
     {Gateway, 0, Gateway} end, Gateways),

  Candidates = NeighbourEntries ++ NonNeighbourEntries,
  %% iterate candidate list to construct routing table.
  iterate(Candidates, Map, []).
```

We then iterate each node in the sorted list as well as each of its adjacent nodes to compare with the ones in routing table in terms of path length. The nodes with shorter length of path are added to the routing table in replacement of the previous added nodes (with the same identifier).

The iteration procedure stops when the end of the candidate list or a node with infinite path length is reached.

```
%% stop when reach the end of table
iterate([], _Map, Table) ->
  Table;
%% stop when reach unknown entry
iterate([{_, inf, unknown}|_Rest], _Map, Table) ->
  Table;
%% interate each node as well as the nodes it directly linked to.
iterate([{Node, N, Gateway}|Rest], Map, Table) ->
  ReachableNodes =  map:reachable(Node, Map),
  NSorted = lists:foldl(fun(Curr, Prev) -> update(Curr, N + 1, Gateway, Prev) end, Rest
  iterate(NSorted, Map, [{Node, Gateway} | Table]).
```

Now we have successfully constructed the routing table which can be used to locate the gateway node to which the message should be routed.

```
route(Node, Table) ->
  case lists:keyfind(Node, 1, Table) of
    {_, Gateway} ->
      {ok, Gateway};
    false  ->
      notfound
  end.
```

4 Maintain a consistent view of the network topology

Maintaining a consistent view of the network map requires each link-state router node to

- filter historically recieved link-state messages to avoid updating its network map with invalid/outdated network connectivity information.
- detect 'DOWN' nodes or nodes that are no longer reachable.

In link-state protocol implementation, link-state message contains a sequence number indicating the version of the message and each router node maintains the latest version number it previously received and the identifier of conresponding sender node.

Historical link-state mesasge detection is implemented in update/3 function in `hist.erl` module:

```
update(Node, N, History) ->
  case lists:keyfind(Node, 1, History) of
    { _, Counter } when Counter < N ->
        {new, lists:keyreplace(Node, 1, History, {Node, N})};
    { _, _Counter } -> old;
    false -> {new, [{Node, N} | History]}
end.
```

Next subsidiary step is to detect the `'DOWN'` or unreachable nodes in the link-state network.

In `routy`, upon connecting to a neigbour node the node is automatically added to Erlang process monitor. Erlang monitor detects the previously connected nodes not only in case they should crash, but also in case they should become unreachable. Erlang monitors notifies the failure of a connected nodes through sending a `'DOWN'` message with the node identifier to allow us to remove it from the node's network map and recompute the routing table.

'DOWN' message notification handling in ***`routy`*** *is illustrated in the code snippet below:*

```
receive
  {add, Node, Pid} ->
    Ref = erlang:monitor(process, Pid),
```

```
  {'DOWN', Ref, process, _, _} ->
    {ok, Down} = intf:name(Ref, Intf),
    io:format("~w: exit received from ~w~n", [Name, Down]),
    Intf1 = intf:remove(Down, Intf),
end
```

5 Evaluation

The Evaluation of our link-state routing algorithm is performed over a graph of nodes running on several independant Erlang VMs (to simulate the link-state network). The image below presents the network graph built from our evaluation:

First, we start four erlang VMs to present four different regions/countries in the world.

```
erl -sname germany -setcookie routy -connect_all false
erl -sname norway -setcookie routy -connect_all false
erl -sname sweden -setcookie routy -connect_all fals
erl -sname spain -setcookie routy -connect_all false
```

The node on each region are activated through `routy:start(node_name)` command:

```
routy:start(gothenburg).
routy:start(lund).
%% norway
routy:start(oslo).
....
```

We then mannually interconnect each node with its directly linked nodes in the graph:

```
gothenburg ! {add, lund, {lund, 'sweden@Chuans-MBP'}}.
lund ! {add, gothenburg, {gothenburg, 'sweden@Chuans-MBP'}}.
lund ! {add, stockholm, {stockholm, 'sweden@Chuans-MBP'}}.
stockholm ! {add, lund, {lund, 'sweden@Chuans-MBP'}}.

berlin ! {add, gothenburg, {gothenburg, 'sweden@Chuans-MBP'}}.
gothenburg ! {add, berlin, {berlin, 'germany@Chuans-MBP'}}.
```

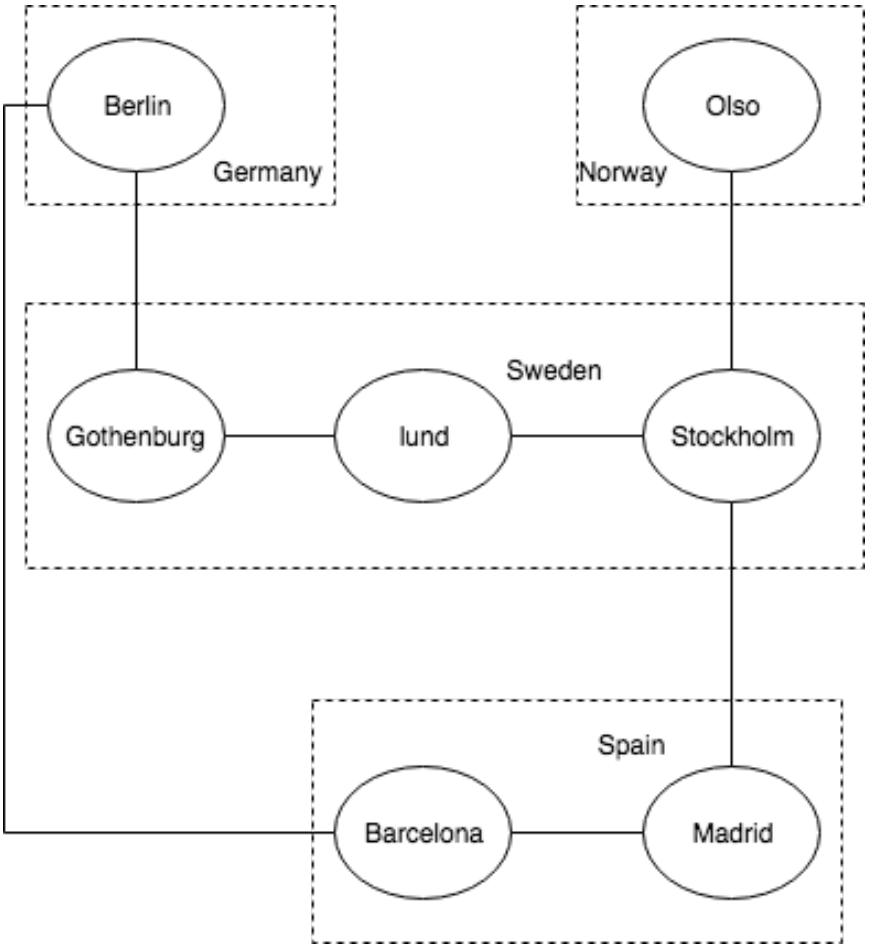

Figure 1: Link-state network graph

```
stockholm ! {add, oslo, {oslo, 'norway@Chuans-MBP'}}.
oslo ! {add, stockholm, {stockholm, 'sweden@Chuans-MBP'}}.

barcelona ! {add, madrid, {madrid, 'spain@Chuans-MBP'}}.
madrid ! {add, barcelona, {barcelona, 'spain@Chuans-MBP'}}.

stockholm ! {add, madrid, {madrid, 'spain@Chuans-MBP'}}.
madrid ! {add, stockholm, {stockholm, 'sweden@Chuans-MBP'}}.

barcelona ! {add, berlin, {berlin, 'germany@Chuans-MBP'}}.
berlin ! {add, barcelona, {barcelona, 'spain@Chuans-MBP'}}.
```

If we send `node!status` message to each node, we will obtain an overview of gatways connected to each node:

Node	Gatways / Links
Berline	Barcelona, Gothenburg
Oslo	Stockholm
Barcelona	Berlin,Madrid
Madrid	Stockholm, Barcelona
Lund	Stockholm, Gothenburg
Gothenburg	Berlin, Lund
Stockholm	Madrid, Oslo, Lund

We then mannully trigger link-state message broadcasting on each node in the graph to construct the routing table for each node.

```
berlin ! broadcast.
oslo ! broadcast.
...
%% construct the routing table
lund ! update.
....
%% view map and routing table information of node Lund.
lund ! status.
....
```

Now we should be able to route a message to a destination node to verify our algorithm:

```
madrid ! { send, berlin, 'Hello from Madrid' }.
```

And we are able to see that node *Berlin* successfully received the message. The route is *Madrid -> Barcelona -> Berlin*, which is the shortest path in the graph for the message to arrive *Berlin* from *Madrid*

6 Conclusion

In this semniar I have gained a comprehensive understanding of *Link State Routing Protocol and Algorithms* and practiced erlarng's map and reduce APIs - `lists:map` and `lists:foldl`.