Routy — A small routing protocol

Anton Bothin

August 31, 2019

1 Introduction

The assignment was to implement a link-state routing protocol in Erlang.

The goal of the assignment was to gain knowledge of the structure of a link-state routing protocol, to understand how a consistent view is maintained, and to reflect on some of the problems that relates to network failures.

2 Main problems, solutions and evaluation

2.1 Map

The map is used to represent the network. The structure is implemented as a list of tuples containing two elements: the name of a node, and a list of nodes reachable from that particular node.

The map module consists of the following functions: `new/0`, `update/3`, `reachable/2`, `all_nodes/1`.

- `new()` simply creates an empty list.
- `update(Node, Links, Map)` removes the previous entry for `Node` (if it exists) and appends a new entry with the new list of reachable nodes.
- `reachable(Node, Map)` returns a list of all nodes reachable from `Node`.
- `all_nodes(Map)` returns a list of all nodes in the Map. This includes nodes that do not have their own entry, that is to say nodes that can be reached but can themselves not reach any other node. A small problem encountered during the implementation of this function was that I needed to make sure that no duplicate entries were returned. This was solved crudely by checking `lists:member/2` before adding a node to the list.

2.2 Dijkstra

The dijkstra module is used to create the routing table. It contains four help functions: `entry/2, replace/4, update/4, iterate/3`, and two exported functions: `table/2, route/2`.

- `entry(Node, Sorted)` return the shortest amount of hops to reach `Node`. If there is no entry with `Node` then 0 is returned instead, this later guarantees that `update/4` won't add a node that is not already in the routing table.
- `replace(Node, N, Gateway, Sorted)` replaces an already existing node in `Sorted` with a new entry with the updated `N` and `Gateway` values. The returned list will still be sorted by inserting the value in its correct place straight away. Technically `replace/4` will still work even if the node entry doesn't already exist. This doesn't matter in the implementation through, since `replace/4` is only called if `entry/2` returns a greater `N`.
- `update(Node, N, Gateway, Sorted)` uses the previously described `entry/2` and `replace/4` to update an entry only if the new entry has a lesser `N` than the previous.
- `iterate(Sorted, Map, Table)` goes through `Sorted`, during each iteration it takes the first entry on the form {`Node, N, Gateway`} and uses `Map` to see which nodes can be reached from `Node`. `Sorted` is then updated for each reachable node with a new value of `N + 1`, because of how `update/4` is implemented this new entry will only be added if `N + 1` is less than the previous length. The built in function `lists:foldl/3` was useful here since I could avoid unnecessary help functions. The routing table is returned either once there are no elements left `Sorted` or once an entry with an infinite length is encountered.
- `table(Gateways, Map)` is the function that constructs the routing table. The first step is to generate `Sorted`, for each gateway an entry on the form {`gateway, 0, gateway`} is added, and for all nodes in `Map` an entry on the form {`node, inf, unknown`} is added. After this `iterate/3` is called with an empty routing table.
- `route(Node, Table)` checks if the routing table contains an entry for `Node`. If it does the gateway for that node is returned, otherwise `notfound` is returned.

2.3 Interface

Each router need to keep track of its current interfaces. These are stored in a list containing tuples with the structure {`Name, Ref, Pid`}. The intf module contains useful functions for looking up values and broadcasting messages. The implementation of these functions was very straightforward and didn't pose any sort of problem.

2.4 History

Each router also has to keep track of its history, this is to prevent circular loops. It would however be a waste of resources to store each message, therefor messages are marked with an incremental number. If a router receives a message with lower number and from the same node as an entry in the history, this message is disgarded. The hist module contains two functions: `new/1`, and `update/3`.

- `new(Name)` returns a list containing a single entry: {`Name, inf`}. This is because a router should discard all messages that have looped back to itself.
- `update(Node, N, History)` should return `old` if `N` is less than the length stored in the history. If no entry exists or if `N` is greater, {`new, Updated`} should be returned, where `Updated` is the new history with an updated entry for `Node`. A problem I had creating this function was the fact that it should return a tuple, making it hard to append values during the recursion. This was solved by using `lists:keyfind/3` and `lists:keydelete/3`, but this solution resulted in an increased runtime.

2.5 Router

The code for the routy module was already provided, all that was left was putting it together and testing the application.

3 Conclusions

I think this assignment did a great job of showing how a network of routers is built up and how routers communicate with each other. It was easy to understand how routers propegate information they have received throughout the network to maintain a consistent view. This was however not the case when a router received a 'DOWN' message, this information was not propegated which resulted in messages not getting to their destination when testing.