Routy - a small routing protocol - Report HW2

Malte Berg

September 17, 2025

1 Introduction

In this homework assignment, the topics covered have been the following:

- Understanding the structure of a link-state routing protocol
- Utilizing Dijkstra's algorithm for routing messages
- Learning more about reference handling in Erlang
- Programming with the lists library

2 Main problems and solutions

Following the instructions in the beginning was fairly easy. The tip to study the lists library, especially `keyfind/3`, `keydelete/3`, `map/2`, `foldl/3`, as well as the unmentioned `keyreplace/4`, was great as it was necessary to understand them during the implementation of the different modules.

The first task was to implement the `map` module. `new/0` and `reachable/2` were easily implemented, with new retuning a simple empty list, and reachable only needing a `case` expression catching {`Node, Links`} from the map and returning the links, otherwise returning an empty list if the node is not found.

Implementing `update/3` seemed easy at first. Returning only the node with its links as a tuple if the map was empty worked fine as the base case. The implementation for when the map had entries was done with `keyreplace/4` at first, which worked when the node already existed in the map, but failed if it did not.

This problem was solved by adding a case expression first with a `keyfind/3` to see if the node exists before replacing, and appending the tuple to the existing map if the node does not exist. The problem was found very late in the process of solving the task, and could have been found earlier if more edge-cases had been tested.

Next the `dijkstra` module was to be implemented. `entry/2` and `update/4` were constructed without problems. The `replace/4` function was however later in the process discovered to have been implemented faulty. The problem was found in the sorting of the list, where an error in the self-written expression for the `sort/2` function in the lists library sorted with the first item being the smallest, and the rest being in reverse order. The `keysort/2` function from the `lists` library was used instead, which was sorted on the amount of hops as the key.

For the `iterate/3` function, it was easy to solve, but another tiny mistake proved to cause major problems in the final stages of finishing the task.

Returning the supplied table without change when either no more entries or the entry has infinite hops was done with simple catching in the parameters. When an actual iteration was to be done, the updated sorted list was created with the help of the `foldl/3` function doing an update with a link of the node, a hop of N+1 and the gateway. The problem was that the update was called with the link and the node itself instead of the gateway, which was incorrect logic of Dijkstra's algorithm. When the problem was fixed, the function worked properly.

The `table/2` function uses two `foldl/3` functions. The first generates an initial list with all the nodes of the map having infinite hops to an unknown destination, the second to update the initial list with the gateways having zero hops to itself. The new sorted list is sent to iterate to compute a route table.

The `route/2` function was solved with a simple `keyfind/3` on the tuple {`Node, Gateway`}.

The `intf` module was the quickest to implement. `new/0` returns an empty list. `add/4` appends to the interface list, `remove/2` uses `keydelete/3` from the `lists` library on the interface list. `lookup/2`, `ref/2`, and `name/2` all use `keyfind/3` on different keys. `list/1` uses `foldl/3` on the first element of the tuples in the interface list appending it to the accumulator, starting with an empty list. Finally, `broadcast/2` uses `foreach/2` from the `lists` library to send the message to the third element in every tuple of the interface list, that being the process ID of the specific router.

The `hist` module for the message history has two functions. `new/1` returns a list with a tuple of the name given as a parameter, and a counter value of `inf` to make any message from itself appear old. As taught earlier in the instructions, `inf` will always be bigger than any integer it's compared to, proving useful when doing an update comparison.

The function `update/3` contains a `keyfind/3` which results in an update if not found, and a comparison of the message index to the new update, re-

sulting in an update if greater than the current counter, and an `old` response if less than or equal to the current counter.

Finally the `routy` module was implemented. The functions were all implemented according to instructions. The status function was implemented in two ways. The first way was to implement it as a functions that could be called through the `routy` module. This was done by having the parameter be the node to get the status from, and having the first line be sending the node a message of {`status, self()`}. The response is then caught with a receive, and is printed in the shell using `io:format/2`.

The second solution, that was quicker to type when debugging, was adding a receive guard with the line status, that prints the specifics of the node, and then loops to `router/6` again. This meant, that for router `r1`, the difference would be: `routy:status(r1).` compared to `r1 ! status.`.

A test module `test` was written which can be called with `test:start(Node).`, where `Node` is the Erlang shell where it should be executed in, for example `'sweden@192.168.1.10'`.

3 Evaluation

The task took approximately 20 hours to complete, with a large portion of that time spent on debugging and testing. The solutions done have been looked over and attempted to be improved where it could be.

The result is a program that can successfully route messages between different processes in the shell, but also route between shells.

The test module shows of the capability of the program, linking 5 routers together with messages being sent, and showing that it can successfully detach two nodes from each other and update the routing tables.

The program would absolutely benefit from automatic broadcasting and updating of route tables, as especially when adding multiple routers to each other, a router link is easily forgotten. With the way the broadcast is implemented with the interfaces, an improvement could be made to allow one-way connections, as all links now have to be bi-directional to function. The implementation also makes it so the routers' routing tables are updated, but their maps keep the disconnected node still on there.