ID2201 Distributed Systems, basic course

Routy: A Small Routing Protocol

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
July 7, 2023

Introduction

Your task is to implement a link-state routing protocol in Erlang. The link-state protocol is used in, for example OSPF, the most used routing protocol for Internet routers. The aim of this exercise is that you should be able to:

- describe the structure of a link-state routing protocol;
- describe how a consistent view is maintained;
- reflect on the problems related to network failures.

We will implement routing processes with logical names such as `london`, `berlin`, `paris` etc. Routers can be connected to each other with directional (one-way) links and can only communicate with the routers they are directly connected to.

The routing processes should be able to receive a message of the form `{route, london, berlin, "Hello"}` and determine that it is a message from `berlin` that should be routed to `london`. A routing process should consult its routing table and determine which *gateway* (a routing process to which it has a direct connection) is best suited to deliver the message. If a message arrives at its destination (the router called `london`), it is printed on the screen. Messages for which paths are not found are thrown away, and no control messages are sent back to the sender.

During the seminar, you will be divided into groups representing regions of the world (Europe, Asia, Africa, etc). Each Erlang engine you run will have a name of a country in that region (Sweden, UK, France, etc.). Assume that the Erlang shell named `sweden` runs on a machine with the IP address 130.123.112.23. Routing processes that we create will be registered as `r1`, `r2` etc. The address of a routing process is thus, for example:

```
    {r1, 'sweden@130.123.112.23'}
```

The routing process will also have a logical name, for example, `stockolm`, but this is a name on a different level.

Your task before the seminar will be to have a router up and running. At the seminar, we will connect the routers.

Before implementing the operations, we advise you to study the `lists` library and learn how `keyfind/3`, `keydelete/3`, `map/2` and `foldl/3` works. It will make your life easier, but if you don't understand what `foldl/3` does, don't even try to use it.

1 The map

Think of a good representation of a directional map where you can easily update the map and find nodes directly connected to a given node. We could represent it as a list of entries where each entry consists of a city with a list of directly connected cities. This will give us a very quick way of updating the map; replace an entry with a new entry. For our purposes, this is fine; in other situations, one might want other operations to be efficient and therefore need another representation.

In a module `map`, implement and export the following functions:

- **new():** returns an empty map (an empty list)
- **update(Node, Links, Map):** updates the Map to reflect that Node has directional links to all nodes in the list Links. The old entry is removed.
- **reachable(Node, Map):** returns the list of nodes directly reachable from Node.
- **all_nodes(Map):** returns a list of all nodes in the Map, also the ones without outgoing links. So if `berlin` is linked to `london` but `london` does not have any outgoing links (and thus no entry in the list), `london` should still be in the returned list.

Before going further, make sure that your implementation of the map works. In the tests below, the map is represented as a list of entries holding the node and the links. Try the following tests:

```
 > map:new().
[]

> map:update(berlin, [london, paris], []).
[{berlin,[london,paris]}]
```

```
 > map:reachable(berlin, [{berlin,[london,paris]}]).
[london,paris]

>map:reachable(london, [{berlin,[london,paris]}]).
[]

>map:all_nodes([{berlin,[london,paris]}]).
[paris,london,berlin]

>map:update(berlin, [madrid], [{berlin,[london,paris]}]).
[{berlin, [madrid]}]
```

Note that the users of a map should not know the representation of the map. A module using a map should only use the four functions described above.

2 Dijkstra

The Dijkstra algorithm will compute a *routing table*. The table is represented by a list with one entry per node where the entry describes which gateway city should be used to reach the node. The input to the algorithm is:

- a map
- a list of *gateways* to which we have direct access.

An example of a routing table is:

```
[{berlin,madrid},{rome,paris},{madrid,madrid},{paris,paris}]
```

This table says that if we want to send something to `berlin`, we should send it to `madrid`. Note that we also include information that to reach `madrid`, we should send the message to `madrid`.
A router will know its name, a set of gateways, a network map, and hopefully not an old routing table.

The map will describe how all other nodes, including the gateways, are connected. The map will not include the router itself. When building the router, we should see that the map is updated frequently. The routing table is, however, only updated once in a while (when we say so).

2.1 A sorted list

The algorithm will use a sorted list to calculate a new routing table. We will start by implementing operations on a sorted list and then look at the algorithm.

Each entry in the list will hold the node's name, the path length to the node, and the gateway we should use to reach the node. An entry showing that `berlin` could be reached in 2 hops using `paris` as a gateway could look like follows:

```
{berlin, 2, paris}
```

The list is sorted based on the length of the path. We should be able to update the list to give a node a new length and a new gateway, but when we do an update, it is important that we update an existing entry and that we actually have an entry in the list to update.

To implement the update procedure, it could be an advantage first to implement two procedures that will help us. In a module `dijkstra`, implement the two procedures:

- **entry(Node, Sorted):** returns the length of the shortest path to the node or 0 if the node is not found.
- **replace(Node, N, Gateway, Sorted):** replaces the entry for Node in Sorted with a new entry having a new length N and Gateway. The resulting list should, of course, be sorted.

Note that in `replace/4`, we require an entry for the node to be present in the sorted list. Be careful and ensure the resulting list is sorted based on the new entry.
Now, when we have these two procedures, it is easier to implement the update procedure.

- **update(Node, N, Gateway, Sorted)**: update the list Sorted given the information that Node can be reached in N hops using Gateway. If no entry is found, then no new entry is added. Only if we have a better (shorter) path should we replace the existing entry.

The procedure is implemented simply by first calling the `entry/2` procedure to get the length of the existing path. If we have a better (shorter) path, then we use the `replace/4` procedure. Why did we make `entry/2` return 0 if the node is not found?

```
> dijkstra:update(london, 2, amsterdam, []).
[]

> dijkstra:update(london, 2, amsterdam, [{london, 2, paris}]).
[{london,2,paris}]

> dijkstra:update(london, 1, stockholm,
      [{berlin, 2, paris}, {london, 3, paris}]).
[{london,1,stockholm}, {berlin, 2, paris}]
```

2.2 The iteration

This is the heart of the algorithm. We will take a sorted list of entries, a map, and a table we have constructed so far. We have three cases:

- If there are no more entries in the sorted list, then we are done, and the given routing table is complete.
- If the first entry is a dummy entry with an infinite path to a city, we know that the rest of the sorted list is also of infinite length, and the given routing table is complete.
- Otherwise, take the first entry in the sorted list, find the nodes in the map reachable from this entry, and for each of these nodes, update the Sorted list. The entry that you took from the sorted list is added to the routing table.

Iterate this until we have no more entries in the sorted list - the table is then complete.

What is happening here? If the entry says that `berlin` can be reached in three hops by going through `paris` and the map says that berlin is directly linked to `copenhagen`, then `copenhagen` is reachable in four hops going through `paris`. We might already have an entry for `copenhagen` using only three hops over `amsterdam`, and then nothing is done, but if we have an entry with more than four hops, we will update the list.

If we have an entry for `copenhagen` with less than three hops, this entry has already been processed and removed from the list. This explains why we do not want to add another entry for `copenhagen`.

Note that since our network is connected by directional links, it could be the case that some nodes in our map are unreachable. If `ulanbator` has a link to `beijing` but there is no link from `beijing` to `ulanbator`, then the world will have `ulanbator` in the map. If all cities in the map are chosen to be part of the original sorted list that we try to iterate over, we will, in the end, find an entry:

```
{ulanbator, inf, unknown}
```

as the first element in the list. If we have this situation, we can conclude that our routing table is complete and contains all reachable cities.

- **iterate(Sorted, Map, Table)**: construct a table given a sorted list of nodes, a map, and a table constructed so far.

The second case is to handle the situation when nodes in the map are not reachable. To capture this, we look at the first node in the sorted list. If we have a node with the length set to infinity, `inf`, then this node (nor any other node after it since the list is sorted) cannot be reached and need not be part of the final table.

This is a test of the iterate procedure:

```
> dijkstra:iterate([{paris, 0, paris}, {berlin, inf, unknown}],
       [{paris, [berlin]}], []).
[{paris, paris},{berlin,paris}]
```

Now in the same module, implement the function `table/2` that should take a list of gateways and a map and produce a routing table with one entry per node in the map. The table could be a list of entries where each entry states which gateway to use to find the shortest path to a node (if we have a path). Follow the outline below, and your program will run in no-time.

- **table(Gateways, Map)**: construct a routing table given the gateways and a map.

List the nodes of the map and construct an initial sorted list. This list should have dummy entries for all nodes with the length set to infinity, `inf`, and the gateway to `unknown`. The entries of the gateways should have a length of zero and the gateway set to itself. Note that `inf` is greater than any integer (try). When you have constructed this list, you can call iterate with an empty table. This is a test of the table procedure:

```
> dijkstra:table([paris, madrid], [{madrid,[berlin]}, {paris, [rome,madrid]}]).
[{berlin,madrid},{rome,paris},{madrid,madrid},{paris,paris}]
```

To complete the dijkstra module, we need one more procedure.

- **route(Node, Table)** search the routing table and return the gateway suitable to route messages to a node. If a gateway is found, we should return {`ok, Gateway`}; otherwise, we return `notfound`.

The `table/2` and `route/2` are the only procedures that we need to export. No one outside the module knows how the table is represented, so you can re-implement it and make it even more efficient.

3 Interfaces

A router will also need to keep track of a set of interfaces. An interface is described by the symbolic name (`london`), a *process reference*, and a *process identifier*. When you implement the router, it will be clear what a process reference is. Implement the following procedures:

- **new()** return an empty set of interfaces.

- **add(Name, Ref, Pid, Intf)** add a new entry to the set and return the new set of interfaces.
- **remove(Name, Intf)** remove an entry given a name of an interface, return a new set of interfaces.
- **lookup(Name, Intf)** find the process identifier given a name, return {`ok, Pid`} if found otherwise `notfound`.
- **ref(Name, Intf)** find the reference given a name and return {`ok, Ref`} or `notfound`.
- **name(Ref, Intf)** find the name of an entry given a reference and return {`ok, Name`} or `notfound`.
- **list(Intf)** return a list with all names.
- **broadcast(Message, Intf)** send the message to all interface processes.

It should be quite straightforward to implement this.

4 The history

When we send link-state messages around, we must avoid cyclic paths; if we are not careful, we will resend messages forever. We can solve this in two ways: we set a counter on each message and decrement the counter in each hop, hoping it will reach all routers before the counter reaches zero, or we keep track of what messages we have seen so far.

We will try the latter strategy, but to avoid keeping a copy of all messages, we will tag each constructed message with a per router increasing message number. If we know that we have seen message `15` from `london`, then we know that messages from `london` with a lower number are old and can be thrown away. This strategy avoids circular loops and prevents old messages from being delayed and later allowed to change our view of the network.

Implement a data structure called history that keeps track of what messages we have seen. In module `hist`, implement two procedures.

new(Name) Return a new history, where messages from Name will always be seen as old.

update(Node, N, History) Check if message number N from the Node is old or new. If it is old, then return `old`, but if it is new, return {`new, Updated`} where Updated is the updated history.

To determine if a link-state message is old or new, one needs not to store the message itself nor all previously received messages. We only have to keep track of the highest counter value received from each node. Can you create an entry for a node that will make any message look old?

5 The router

The router should be able to route messages through a network of connected nodes, maintain a view of the network, and construct optimal routing tables. Each routing process will have a state:

- a symbolic name such as `london`,
- a counter,
- history of received messages,
- a set of interfaces,
- a routing table,
- a map of the network.

When a new router process is created, it will initially set all its parameters to initially empty values. We will also register the router process under a unique name (unique for the Erlang machine it runs on, for example, `r1`, `r2`, etc.).

```
-module(routy).

-export([start/2, stop/1, ...]).

start(Reg, Name) ->
    register(Reg, spawn(fun() -> init(Name) end)).

stop(Node) ->
    Node ! stop,
    unregister(Node).

init(Name) ->
    Intf = intf:new(),
    Map = map:new(),
    Table = dijkstra:table(Intf, Map),
    Hist = hist:new(Name),
    router(Name, 0, Msgs, Intf, Table, Map).
```

To route a message to a node, the router will consult the routing table to find the best gateway and then find the pid of that gateway, given the list of interfaces. This is the easy part; the hard part is to maintain a consistent view of the networks as interfaces are added and removed. The algorithm of a links-state protocol is as follows:

- determine which nodes that you are connected to;
- tell all neighbors in a *link-state message*;
- if you receive a link-state message that you have not seen before, pass it along to your neighbors

A node will thus collect link-state messages from all other routers in the network. The link-state messages are exactly what we need to build a map. Since we also know which nodes we can reach directly, our gateways, we can use Dijkstra's algorithm to generate a routing table.

In our first effort, we will, however, only implement a process that can connect or disconnect to other nodes in the system and update its set of interfaces.

5.1 Adding interfaces

We will use *monitors* to detect if a node is unreachable; a monitor will send an `'DOWN'` message to the process, and we can then remove links to the node. A skeleton code for the router process could look as follows.

```
router(Name, N, Hist, Intf, Table, Map) ->
    receive
%            :
%            :
        {add, Node, Pid} ->
            Ref = erlang:monitor(process,Pid),
            Intf1 = intf:add(Node, Ref, Pid, Intf),
            router(Name, N, Hist, Intf1, Table, Map);

        {remove, Node} ->
            {ok, Ref} = intf:ref(Node, Intf),
            erlang:demonitor(Ref),
            Intf1 = intf:remove(Node, Intf),
            router(Name, N, Hist, Intf1, Table, Map);

        {'DOWN', Ref, process, _, _}  ->
            {ok, Down} = intf:name(Ref, Intf),
            io:format("~w: exit recived from ~w~n", [Name, Down]),
            Intf1 = intf:remove(Down, Intf),
            router(Name, N, Hist, Intf1, Table, Map);

%            :
%            :
        {status, From} ->
            From ! {status, {Name, N, Hist, Intf, Table, Map}},
            router(Name, N, Hist, Intf, Table, Map);

        stop ->
            ok
    end.
```

Note that creating a monitor for a process that does not exist will fail nor throw an exception. What will happen is that you're immediately sent a down message. The behavior is thus the same if you add a monitor to a process that dies or if you add a monitor to a process that died 10 milliseconds ago.

The {status, From} message can be used to do a pretty-print of the state. Add a function that sends a status message to a process, receives the reply, and displays the information.

When we start Erlang shells, we will all have to use the same magic cookie, so let's agree on routy. We could also use a flag to reduce the underlying network traffic. The default behavior for distributed Erlang is to try to connect to all nodes available in the network. Connecting A with B, where B is already connected to C, will create a connection between A and C. We can turn this feature off since we will allow our nodes to crash.

```
erl -name sweden@130.123.112.23 -setcookie routy -connect_all false
```

To try to keep things under control, we name Erlang nodes after countries and routers after names in that country. Start two routers and send them messages so that they connect to each other. Terminate one of them and see that things work.

5.2 Link-state messages

Link State（链路状态）指
的就是路由器的接口状态。

Next, we need to implement the link-state message. When this is sent, it is tagged with the counter value. The counter is then updated so subsequent messages will have a higher value. When receiving a links-state message, a router must check if this is an old or new message. The handling of link-state messages can be implemented as follows:

```
        {links, Node, R, Links} ->
           case hist:update(Node, R, Hist) of
            {new, Hist1} ->
                  intf:broadcast({links, Node, R, Links}, Intf),
                  Map1 = map:update(Node, Links, Map),
                  router(Name, N, Hist1, Intf, Table, Map1);
            old ->
                 router(Name, N, Hist, Intf, Table, Map)
        end;
```

When we have updated our map, we should also update the routing table. This is where we invoke the Dijkstra algorithm. We should do it periodically, every time we receive a link-state message or, better yet, every time the map changes. In our experiment, we will do it manually. We add a method `update` that we will send to order the router to update its routing table.

```
        update ->
          Table1 = dijkstra:table(intf:list(Intf), Map),
          router(Name, N, Hist, Intf, Table1, Map);
```

We also add a message so that we can manually order our router to broadcast a link-state message. This should be done periodically or every time a link is added, but we want to experiment with inconsistent maps, so we keep this as a manual procedure.

```
        broadcast ->
          Message = {links, Name, N, intf:list(Intf)},
          intf:broadcast(Message, Intf),
          router(Name, N+1, Hist, Intf, Table, Map);
```

5.3 Testing what we have

We can now test our protocol by starting several routing processes and letting them connect to each other. Let's call Erlang machines for countries and routers for cities. So start an Erlang node with the command:

```
erl -name sweden@130.123.112.23 -setcookie routy -connect_all false
```

Load the `routy` and `dijkstra` module and then start routers for different cities in Sweden. Then connect the routers by manually sending them `add` messages. Note that the add message contains both the logical name (`stockholm`) and the process identifier of the router (for example `{r1, 'sweden@130.123.112.23'}`).

```
(sweden@130.123.112.23)>routy:start(r1, stockholm).

(sweden@130.123.112.23)>routy:start(r2, lund).

(sweden@130.123.112.23)>~~lund~~ ! {add, stockholm, {r1, 'sweden@130.123.112.23'}}.
                        r2
true
```

If everything works out ok, you should be able to build a network of routers. When you send the message `broadcast` to a router, the link-state messages should be generated, and after a `update` message, the routing table should be computed. Try it with some Erlang nodes running on one machine. If you have problems with the long network names, you could start Erlang using short node names `-sname` or have all routers in the same Erlang process.

5.4 Routing a message

It's now time to implement the actual routing. We have one easy case: when a message has arrived at the final destination.

```
{route, Name, From, Message} ->
     io:format("~w: received message ~w ~n", [Name, Message]),
     router(Name, N, Hist, Intf, Table, Map);
```

If the message is not ours, we should forward it. If we find a suitable gateway in the routing table, we forward the message to the gateway. If we do not find a routing entry or do not find an interface of a gateway, we have a problem; drop the packet and keep smiling.

```
{route, To, From, Message} ->
     io:format("~w: routing message (~w)", [Name, Message]),
     case dijkstra:route(To, Table) of
        {ok, Gw} ->
           case intf:lookup(Gw, Intf) of
              {ok, Pid} ->
                  Pid ! {route, To, From, Message};
              notfound ->
                  ok
           end;
        notfound ->
           ok
     end,
     router(Name, N, Hist, Intf, Table, Map);
```

In the implementation, we use the fact that the routing table contains entries even for our own gateways. Could we also have a dummy entry for the node itself so that we would not need to have a special message entry to handle messages directed to the router itself? yes, having s dummy would be better.

We also add a message so that a local user can initiate the routing of a message without knowing the name of the local router.

```
    {send, To, Message} ->
         self() ! {route, To, Name, Message},
         router(Name, N, Hist, Intf, Table, Map);
```

Routy should now be fully operational. Write a 2-3 page report on what was difficult and how you solved it. Discuss if any potential improvements to Routy could be made. Before the reporting seminar, prepare an automated test that sets up a network containing at least five routers and demonstrates Routy's functionality.

6 Optional task for extra bonus: The world

This optional task can be done either before or at the reporting seminar: form a group with your classmates and connect as many routers as possible, start killing nodes, and watch how the network can still route messages. See more details below. If you have done it before the reporting seminar, describe the results of your experiments in the report, including screenshots, if possible. Optionally, you can demonstrate such experiments with your partners at the seminar.

Form a group with your classmates and be responsible for a region in the world (Europe, Africa, South America, etc.); coordinate with your partners so each partner/group has its own region. Then start a set of Erlang nodes on each machine where you give each Erlang node the name of a country (that is in your region).

In each Erlang node, you can now create one or more routers with registered names of cities in that country. Then start to connect the routers to each other. Note that all cities in the world must have unique names, so even if there is a Paris in Texas, the network will only allow one node to be called `paris`. Start to send messages to other nodes and see that it works. Note that you must do this manually since you have not implemented automatic broadcast and update functionality.

When things are working in your region, choose two or more routers that should connect to other parts of the world. Make sensible connections to make it easier to understand what the network looks like. Can we send messages from Sydney to Oslo? Yes

If everything works ok, you can try to either stop routers, close Erlang nodes, or disable the network card. Will the routing functionality still work? How long time does it take between a disabled network card and the delivery of a 'DOWN' message to the other nodes?

If I close the sweden erlang node, usa erlang node will receive the DOWN message immediately, reason: noconnection

When you try to send messages to the sweden erlang node after it is dead, nothing happens and no errors in the world erlang terminal.