Routy: un pequeño protocolo de routing*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

19 de mayo de 2016

Introducción

La tarea es implementar un protocolo de ruteo *link-state* en Erlang. El protocolo link-state es usado por ejemplo en *OSPF*, el protocolo de ruteo más usado por los routers de Internet. El objetivo del ejercicio es ser capaces de:

- describir la estructura general del protocolo de ruteo link-state
- describir como se mantiene una vista consistente
- reflexionar sobre los problemas relacionados a los fallos de red

Vamos a implementar un proceso de ruteo con nombres lógicos tales como `london, berlin, paris`, etc. Los routers puede estar conectado a otros con links dirigidos (1-vía) y solo pueden comunicarse con routers a los que están conectados directamente.

El proceso de ruteo debería poder recibir un mensaje del la forma:

```
{route, london, berlin, "Hello"}
```

y determinar que es un mensaje de `berlin` que debe ser ruteado a `london`. Un proceso de ruteo debería consultar su tabla de ruteo y determinar cual es el gateway (un proceso de ruteo al cual tiene conexión directa) es el más indicado para entregar el mensaje. Si el mensaje llega a su destino (el router llamado `london`) el mismo se imprime en pantalla. Los mensajes para los que no se encuentra una ruta simplemente son descartados, no se envían mensajes de control al proceso que envía.

Durante el curso nos dividiremos en grupos representado distintos continentes (América, Europa, Asia, etc). Cada nodo Erlang que corremos va a tener el nombre del país en la región (Argentina, UK, Mexico, etc). Asumiendo que el shell de Erlang llamado `argentina` corre en la máquina con IP `130.123.112.23`, el proceso de ruteo `rosario` que corre en dicho nodo va a tener la dirección:

*Adaptado al español del material original de Johan Morellius (`https://people.kth.se/~johanmon/dse.html`)

```
{argentina, 'rosario@130.123.112.23'}
```

La tarea antes de terminar el el proyecto es tener un router ejecutando. Antes de implementar las operaciones recomendamos estudiar la librería de listas y como funcionan `keyfind/3`, `keydelete/2`, `map/2` y `foldl/3`.

1. El Mapa

Debemos pensar una buena representación de un mapa dirigido que debe ser fácil para actualizar y para encontrar nodos directamente conectados a otros nodos. Podemos representar este mapa como una lista de nodos de entradas donde cada entrada consiste de un ciudad con una lista de ciudades directamente conectadas. Esto nos va a dar una forma muy rápida the actualizar el mapa, simplemente reemplazamos una entrada con la nueva. Para nuestro propósito esto es suficiente, en otra situación podríamos necesitar que las operaciones sean eficientes y en tal caso otra representación.

En un módulo `map`, implementar y exportar las siguientes funciones:

- `new()`: retorna un mapa vacío (una lista vacía).
- `update(Node, Links, Map)`: actualiza el mapa para reflejar que un nodo tiene conexiones dirigidas a todos los nodos de la lista de links. La entrada vieja se elimina.
- `reachable(Node, Map)`: retorna la lista de nodos directamente alcanzables desde un Node.
- `all_nodes(Map)`: retorna la lista de todos los nodos del mapa, también aquellos sin conexiones de salida. Por ejemplo, si `berlin` está conectado con `london` pero `london` no tiene conexiones salientes (y por lo tanto ninguna entrada en la lista), `london` debería ser retornado de todas formas en la lista.

Antes de avanzar revisemos que nuestra implementación de `map` funciona. En los test siguientes el mapa es representado como una lista de entradas que contiene los nodos y conexiones. Probar los siguiente:

```
> 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]}]
```

Un modulo que usa `map` no debería conocer la representación interna y solo usar las 4 funciones descriptas anteriormente.

2. Dijkstra

El algoritmo de Dijkstra va a calcular la tabla de ruteo. La tabla va a estar representada como una lista con una entrada por cada nodo donde la entrada describe que gateway (ciudad) debe ser usado para alcanzar un nodo.

La entrada del algoritmo es la siguiente:

- un `map`.
- una lista de gateways a los cuales tenemos acceso directo.

Un ejemplo de de la tabla de ruteo es:

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

La tabla dice que si queremos enviar algo a `berlin` debemos enviarlo a `madrid`. Ver que también incluimos información de que si queremos alcanzar `madrid` debemos enviar un mensaje a `madrid`.

Un router conoce su propio nombre, una lista de gateways, un mapa de la red y una tabla de routeo esperemos no tan vieja.

El mapa describe como están conectados el resto de los nodos, incluidos los gateways. El map no incluye al propio router. Cuando construimos un router debemos ver que mapa se actualiza seguido. La tabla de routeo sin embargo se actualiza infrecuentemente (cuando lo indicamos).

2.1. Lista Ordenada

En el algoritmo usaremos una lista ordenada cuando calculemos la tabla de ruteo. Vamos a empezar implementando operaciones sobre una lista ordenada y luego veremos el algoritmo en si.

Cada entrada en la lista va a tener el nombre de un nodo, la longitud del camino al nodo y el gateway que debemos usar para alcanzar el nodo. Una entrada mostrando que `berlin` está conectado en 2 saltos (hops) usando paris como gateway se ve de la siguiente forma:

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

La lista está ordenada en base a la longitud del camino. Deberíamos ser capaces de actualizar la lista para darle a un nodo una nueva longitud y un nuevo gateway pero es importante que cuando hacemos la actualización que actualicemos la entrada existente y que tenemos dicha entrada en la lista.

Para implementar la función de update podemos valer de 2 funciones de ayuda. En el módulo `dijkstra` implementar las siguientes funciones:

- `entry(Node, Sorted)`: devuelve la longitud del camino más corto a un nodo o 0 si no se encuentra el nodo.
- `replace(Node, N, Gateway, Sorted)`: reemplaza la entrada por `Node` en `Sorted` con una nueva entrada de longitud `N` y `Gateway`. La lista resultante oviamente debe estar ordenada.

Notar que `replace/4` necesita que la entrada para `Node` este presente en la lista ordenada.

Ahora teniendo estas 2 funciones es fácil implementar la función update es más fácil de implementar.

- `update(Node, N, Gateway, Sorted)`: actualiza la lista `Sorted` con la información de que `Node` puede ser alcanzado en `N` saltos usando `Gateway`. Si no se encuentra una entrada, no se agrega la nueva entrada. Solo si tenemos un mejor camino (más corto) reemplazamos la entrada existente.

La función se implementa simplemente llamando primero a `entry/2` para obtener la longitud del camino existente. Si tenemos un mejor camino entonces usamos `replace/4`. ¿Por qué hicimos que `entry/2` devuelva 0 si no encuentra el nodo?

```
> 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. La Iteración

Esta es la parte más importante del algoritmo. Vamos a tomar una lista ordenada de entradas, un mapa y una tabla que es lo que hemos construido hasta el momento. Tenemos 3 casos:

- Si no hay más entradas en la lista ordenada entonces terminamos y la tabla de ruteo está completa.
- Si la primer entrada es una entrada dummy con un path infinito a una ciudad sabemos que el resto de la lista ordenada también tiene paths infinitos y la tabla de ruteo está completa.
- En caso contrario, tomar la primer entrada de la lista ordenada, encontrar los nodos en el mapa que son alcanzables desde dicha entrada y por cada nodo actualizar la lista ordenada. La entrada tomada de la lista ordenada se agrega a la tabla de ruteo.

Iterar hasta que no tengamos más entradas en la lista ordenada – entonces la tabla está completa.

¿Qué sucedió? Si una entrada dice que `berlin` puede ser alcanzada en 3 saltos yendo por `paris` y el mapa dice que `berlin` está directamente conectada con `copenhagen`, entonces `copenhagen` es alcanzable en 4 saltos vía `paris`. Podríamos ya tener una entrada para `copenhagen` usando solo 3 saltos vía `amsterdam` y en tal caso no hacemos nada, pero si tenemos una entrad con más de 4 saltos entonces actualizamos la lista.

Si tenemos una entrada para `copenhagen` con menos de 3 saltos, esta entrada ya ha sido procesada y removida de la lista. Esto explica por qué no vamos a agregar otra entrada para `copenhagen`.

Notar que dado que nuestra red está conectada por links direccionales podría darse el caso que algunos nodos del mapa no sean alcanzables de ninguna forma. Por ejemplo, si `ulaanbaatar` tiene una conexión con `beijing` pero no hay una conexión de `beijing` a `ulaanbaatar` entonces entonces el mapa igual tendría a `ulaanbaatar`. Si todas las ciudades en el mapa son parte de la lista ordenada que estamos iterando encontraríamos finalmente una entrada:

```
{ulaanbaatar, inf, unknown}
```

como primer elemento de la lista. Si nos encontramos esta situación podemos concluir que la tabla de ruteo está completa y contiene todas las ciudades alcanzables.

- `iterate(Sorted, Map, Table)`: construye una tabla dada una lista ordenada de nodos, un mapa y la tabla construida hasta el momento.

El segundo caso es manejar la situación cuando los nodos en el mapa no son alcanzables. Para poder capturar este caso mirar con detenimiento el primer nodo en la lista ordenada. Si tenemos un nodo con longitud infinita, `inf`, entonces el nodo (ni ningún otro después de es dado que la lista está ordenada) puede ser alcanzado y no tiene que ser parte de la tabla final.

Este es un test sobre la función `iterate`:

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

Ahora en el mismo módulo implementar la función `table/2` que toma una lista de gateways y un mapa y produce una tabla de ruteo con una entrada por nodo en el mapa. La tabla puede ser una lista de entradas donde cada entrada dice que gateway usar para encontrar el camino más corto a un nodo (si lo hay). Seguir las siguientes instrucciones y tendremos el programa corriendo.

- `table(Gateways, Map)`: construye una tabla de ruteo dada una lista de gateways y un mapa.

Listar los nosdos del mapa y construir una lista ordenada inicial. Esta lista debe tener entradas dummy para todos los nodos con longitud infinita, `inf`, y gateway `unknown`. Las entradas de los gateways deben tener longitud 0 y como gateway ellos mismos. Notar que `inf` es mayor que cualquier entero (probar!). Cuando tengamos construida la lista podemos iterar con una tabla vacía. Este es un test para la función `table`:

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

Para completar el módulo `dijkstra` necesitamos 1 función más.

- `route(Node, Table)`: busca en la tabla de ruteo y devuelve el gateway adecuado para rutear mensajes a un nodo. Si se encuentra un gateway debemos devolver `ok, Gateway` de otra forma retornamos `notfound`.

Las funciones `table/2` y `route/2` son las únicas funciones que necesitamos exportar. Fuera del módulo nadie debería conocer como está implementada la tabla de forma tal de poder cambiarla y hacerla más eficiente.

3. Interfases

Un router también necesita mantener registro de un conjunto de interfases. Una interfase se describe como un nombre simbólico (`london`), una referencia y un identificador de proceso. Cuando implementemos el router vamos a darnos cuenta que es la referencia a un proceso. Implementar las siguientes funciones:

- `new()`: devuelve una conjunto vacío de interfases.
- `add(Name, Ref, Pid, Intf)`: agrega una nueva entrada a al conjunto de interfases y devuelve el nuevo conjunto.
- `remove(Name, Intf)` remueve una entrada dado el nombre de una interfase y devuelve el nuevo conjunto de interfases.
- `lookup(Name, Intf)` busca el identificador de proceso para un nombre dado, retorna `{ok, Ref}` o `notfound`.
- `ref(Name, Intf)` busca una referencia para un nombre dado y devuelve `{ok, Ref}` o `notfound`
- `name(Ref, Intf)` busca el nombre de una entrada dada una referencia, devuelve `{ok, Name}` o `notfound`.
- `list(Intf)` retorna la lista de todos los nombres.
- `broadcast(Message, Intf)` envía un mensaje a todos los proceso de la interfase.

Implementar estas funciones debería ser bastante directo.

4. La Historia

Cuando enviamos mensajes link-state tenemos que evitar caminos cíclicos; si no somos cuidadosos vamos a reenviar mensajes por siempre. Podemos resolver esto de 2 formas, o bien asignando un contador a cada mensaje y decrementarlo en cada salto, esperando que alcance todos los routers antes de llegar a 0, o podemos mantener registro de que mensajes hemos visto hasta el momento.

Vamos a intentar seguir la segunda estrategia pero para evitar tener la copia de todos los mensajes vamos a tagear cada mensaje construido por un router con un número incremental de mensaje. Si sabemos que vimos un mensaje 15 desde `london` entonces sabemos que los mensajes desde `london` con numeración menor pueden ser descartados. Esta estrategia evita caminos circulares pero además previene que mensajes viejos sean retenidos y cambiar posteriormente la vista de la red.

Implementar una estructura de datos llamada `history` que mantiene registro de que mensajes han sido vistos. En el módulo `hist` implementar 2 funciones:

- `new(Name)`: retorna una nueva historia, donde los mensajes de `Name` siempre son vistos como viejo.
- `update(Node, N, History)`: chequea si el mensaje con número `N` desde `Node` es viejo o nuevo, si es nuevo devuelve `{new, Updated}` donde `Updated` es la historia actualizada.

Para determinar si un mensaje link-state es viejo o nuevo necesitamos que no se guarde el mensaje o los anteriores. Lo único que necesitamos mantener es registro del contador más alto recibido de cada nodo. ¿Podemos crear una entrada para cada nodo que haga ver a cualquier mensaje como viejo?

5. El Router

El router debe ser capaz de no solo routear mensajes a través de una red de nodos conectados, sino también mantener una vista de la red y construir tablas de ruteo óptimas. Cada proceso de ruteo debe tener un estado:

- un nombre simbólico como por ejemplo `london`
- un contador
- la historia de mensajes recibidos
- un conjunto de interfases
- una tabla de ruteo
- un mapa de la red

Cuando un nuevo proceso router es creado debe setear todos los parámetros a valores iniciales. También debemos registrar el proceso router bajo un nombre único (único por nodo Erlang en que está corriendo, por ejemplo `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).
```

Para rutear un mensaje a un nodo, el router simplemente va a consultar la tabla de ruteo para encontrar el mejor gateway y luego pedir el `pid` de ese gateway en la lista de interfases. Esta es la parte fácil; la parte difícil es mantener una vista consistente de las redes en la medida que se agregan y se quitan interfases. El algoritmo the un protocolo link-state es el siguiente:

- determinar a que nodos estoy conectado
- avisarle a todos los vecinos en un mensaje link-state
- si recibimos un mensaje link-state que no hemos visto debemos pasarlo a los vecinos

Un nodo de esta forma va a recolectar mensajes link-state de todos los otros routers en la red. Los mensajes link state son exactamente lo que necesitamos para construir el mapa. Dado que conocemos que nodos podemos alcanzar en forma directa, nuestros gateways, podemos usar el algoritmo de Dijkstra para generar la tabla de ruteo.

En un primer intento solo vamos a implementar un proceso que se pueda conectar y desconectar a otros nodos del sistema y actualizar sus interfases.

5.1. Agregando Interfases

Vamos a usar *monitores*[1] para detectar si un nodo es inalcanzable; un monitor va a enviar un mensaje `'DOWN'` al proceso y entonces podemos remover los links a

[1] `http://erlang.org/doc/man/erlang.html#monitor-2`

un nodo. El esqueleto de código para el proceso router puede verse de la siguiente manera.

```
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.
```

Notar que al crear un monitor para un proceso que no existe no va a fallar ni tirar una excepción. Lo que va a suceder es que inmediatamente se envía el mensaje down. Este comportamiento es igual si agregamos un monitor a un proceso que muere que si agregamos un monitor a un proceso que murió hace 10 segundos.

El mensaje `{status, From}` puede ser usado para imprimir el estado. Agregar una función que envía mensajes de `status` a un proceso, reciba la respuesta y muestre la información.

Cuando iniciamos los shells de Erlang vamos a tener que usar misma cookie, así que usaremos `routy`. Podemos usar también un flag para disminuir el tráfico de red. El comportamiento por defecto de Erlang distribuido es tratar de conectar todos los nodos disponibles en la red. Si conectamos A con B y B ya estaba conectado con C esto va a crear una conexión entre A y C. Dado que vamos a hacer que nuestros nodos se caigan vamos a apagar esta propiedad.

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

Para tratar de mantener las cosas bajo control vamos a ponerle a los nodos nombres de países. Iniciar 2 routers y enviar mensajes de forma tal que se conecten entre si. Terminar uno de ellos y ver que todo funciona.

5.2. Mensajes link-state

A continuación necesitamos implementar mensajes link-state, Cuando se envía un mensaje es taggeado con con el valor del contador. El contador se incrementa de forma tal que los siguientes mensajes tenga un valor mayor. Cuando recibimos un mensaje link-state el router debe verificar si es un mensaje nuevo o viejo. El manejo de los mensajes link-state es implementado de la siguiente forma:

```
{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;
```

Cuando actualizamos nuestro mapa también debemos actualizar la tabla de ruteo. Aquí es cuando invocamos el algoritmo de Dijkstra. Debemos hacerlo periódicamente, tal vez siempre que recibimos un mensaje link-state o mejor cada vez que cambia el mapa. En nuestro experimento lo vamos a hacer manualmente. Vamos a agregar un mensaje `update` que va a enviar la orden de actualizar la tabla de ruteo al router.

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

También necesitamos un mensaje para ordenar en forma manual a nuestro router que haga broadcast de un mensaje link-state. Esto debe hacerse periódicamente o cada vez que se agrega una conexión pero vamos a querer experimentar con mapas inconsistentes así que lo vamos a exponer para poder usarlo manualmente.

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

5.3. 1, 2, 3, Probando...

Ahora podemos probar nuestro protocolo levantando varios procesos de ruteo y conectándolos entre si. Llamaremos a los nodos Erlang con nombres de países

y los routers de ciudades. Empecemos levantando un nodo Erlang de la siguiente forma:

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

Cargar los módulos `routy` y `dijkstra` y iniciar los routers para diferentes ciudades. Ahora conectar los routers mandando mensages `add`. Notar que el mensaje `add` contiene el nombre lógico (`rosario`) y el identificador de proceso del router (por ejemplo `{r1, 'argentina@130.123.112.23'}`).

```
(argentina@130.123.112.23)> routy:start(r1, rosario).
(argentina@130.123.112.23)> routy:start(r2, buenos_aires).
(argentina@130.123.112.23)> buenos_aires ! {add, rosario, {r1,
                                'argentina@130.123.112.23'}}.
true
```

Si todo funciona correctamente, deberíamos ser capaces de construir una red de routers. Cuando enviamos un mensaje de `broadcast` a un router debería generarse un mensaje link-state y después de un mensaje `update` se debería calcular la tabla de ruteo. Probarlo con algunos nodos Erlang corriendo en una máquina. Si tenemos problemas por el uso de nombres largos podemos usar nombres cortos `-sname` o simplemente levantar todos los routers en el mismo nodo Erlang.

5.4. Ruteando un Mesaje

Es tiempo de implementar el routeo en si. Tenemos un caso fácil y es cuando un mensaje llega al destino final.

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

Si el mensaje no es para nosotros debemos forwardearlo. Si encontramos un gateway adecuado en la tabla de ruteo simplemente forwardeamos el mensaje al gateway. Si no encontramos un entrada en la tabla de ruteo o no encontramos una interfaz de un gateway tenemos un problema, simplemente descartamos el paquete...

```
{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);
```

En la implementación nos basamos en el hecho de que la tabla de ruteo contiene entradas aún para nuestro propio gateway. ¿Podemos tener una entrada dummy para el nodo también así no necesitamos un mensaje especial para manejar mensajes dirigidos al router?

Agregamos un mensaje también para que un usuario local pueda iniciar el ruteo de un mensaje sin conocer el nombre del router local.

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

Hasta acá hemos llegado, escribir un informe contando cuales fueron las partes más difíciles y como se resolvieron. Vamos a conectar tantos routers como podamos, y a comenzar a matar nodos para ver como la red aún puede routear los mensajes.

6. El Mundo

Cada grupo será responsable de un país en distintas regiones del mundo (Europa, Africa, Sudamérica, etc); coordinar entre los grupos para cada uno tener un país distinto. Iniciar un conjunto de nodos en cada máquina de forma tal que uno tenga el nombre de un país.

En cada nodo Erlang se pueden crear uno o más routers con nombres registrados de ciudades en el país y comenzar a conectar los routers entre si. Notar que todas las ciudades de "nuestro mundo" tienen que tener nombres distintos, aún si tengo una ciudad Rosario en Argentina y en Uruguay la red solo permite que haya un `rosario` registrado. Notar que dado que no implementamos `broadcast` y `update` automáticos, debemos hacer esto en forma manual.

Cuando las cosas están funcionando en un país entonces elegir uno o dos routers para que se conecten a otros países del mundo. Elegir conexiones realistas (y anotarlas) para entender que forma tiene la red. ¿Podemos mensajes de Sydney a Oslo?

Si todo funciona correctamente, podemos intentar bajar routers, bajando nodos Erlang o simplemente desconectando la red. ¿Sigue funcionando el ruteo? ¿Cuánto tarda entre que un nodo se desconecta y el envío del mensaje `'DOWN'` a los otros nodos?