Netsim

Simulating distance-vector routing with synchronised periodic exchances

Uma Zalakain
2423394z@student.gla.ac.uk

2019–03–01

1 Design overview

Netsim is a Haskell package made of a library containing all the logic (`src/Language/Netsim.hs`) and an executable interface using that library (`src/Main.hs`).

Instead of declaring new types, the library makes an extensive use of type synonyms. Type synonyms are viewed as equivalent by the type-checker, but are useful to convey information.

All information is contained within three types. All three types are indexed first by a *source* node and then by a *destination* node: they represent information regarding a relation $A \to B$.

`Network` constains the cost of every direct link from A to B: static information about the network's toponomy. This information remains unchanged during periodic exchanges – unless the user changes it to simulate broken links or cost changes.

The function `bidir` is used to create undirected networks, where $A \to B \iff B \to A$. This is done by taking the union of the given network and its swapped copy – where $A \to B \to cost$ gets transformed into $B \to A \to cost$.

`State` contains the cost of the best route from A to B together with the gateway for such route: all the runtime state. The gateway of a node must be directly linked to that node.

`Ads` contains advertisment messages sent from A to B. These messages the periodic exchanges in the network. They contain distance vectors – $destination \to cost$ maps.

On each *tick* of the network, nodes first create advertisements (containing a source, a destination and a distance vector) and then update their tables with advertisements sent to them. Stale broken links are cleared from routing tables. Distance vectors created as part of route advertisement messages can have split-horizon toggled.

All features related to execution are found in `src/Main.hs`. Networks are declared by the user through a simple syntax and then parsed. If the `-d` switch is present, links will be considered directed, otherwise they will be made bidirectional. Finally, the user is presented with a simple shell where the main functionality is offered: running the simulation, querying the state, modifying the underlying network, activating split-horizon.

2 User manual

Note: an executable capable of running on Windows 10 x64 could not be provided, as cross-compilation in Haskell can be tricky and the author does not have a Windows 10 x64 machine.

This project is written in Haskell and uses Cabal to build itself. Both can be installed by following the steps found at `https://www.haskell.org/platform/windows.html`.

2.1 Compilation

Assuming a shell with the project's directory as the current directory, the project can be built as follows:

```
# Updates the list of packages available online
cabal update
# Installs dependencies
cabal install --dependencies-only
# Builds the project
cabal build
```

The resulting binary will be found in `dist/build/netsim/netsim`.

2.2 Installation

Assuming a shell with the project's directory as the current directory, the project can be installed as follows:

```
# Updates the list of packages available online
# Only necessary the first time
cabal update
# Install the project and its dependencies
cabal install
```

The generated output will specify the location of the generated binary. The shell's path list should point to the directory in which the binary resides, in which case it will be executable by running `netsim`.

2.3 Execution

Executing netsim without any arguments will output a brief help text. Netsim expects a network file as its first argument. Unless netsim is ran without the `-d` switch, network links are undirected.

Network files must have one link declaration per line (should support Windows' carriage returns). Let `<SOURCE>` and `<DESTINATION>` be arbitrary identifiers with no space in them, and let `<COST>` be an integer, then a link is declared as follows:

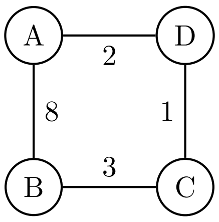
```
<SOURCE> <DESTINATION> <COST>
```

Once a network file is successfully parsed, netsim will present an interactive shell. Unrecognised commands will result in help text being printed. The list of commands follows:

3 Example walkthrough

3.1 Normal convergence

The figure below shows a network in which the direct link $A - B$ is more costly than the route $A - D - C - B$. The file describing the network can be loaded with `netsim examples/ex5.netsim`.



Initially, each node only knows about its neighbours:

```
> table A
<D, 2, D>
<B, 8, B>
```

After the first exchange, each node knows about nodes 2 hops away. The route $A - B$ is still suboptimal:

```
> tick
> table A
<D, 2, D>
<B, 8, B>
<C, 3, D>
```

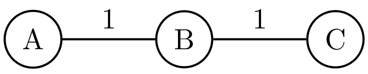
After another exchange, each node knows about nodes 3 hops away, and thus A knows about the optimal route to B:

```
> tick
> table A
<D, 2, D>
<B, 6, D>
<C, 3, D>
```

3.2 Slow convergence

The figure below shows a network in which A and C need to talk through B. The file describing the network can be loaded with `netsim examples/ex4.netsim`.

After the initial exchanges that make the network stabilise, the link between A and B will be broken. Depending on whether split-horizon is activated or not, the subsequent exchanges might enter a loop and the network might never stabilise.



3.2.1 Without split-horizon

First, periodic exchanges occur until the network is stabilised:

```
> run
> table A
<B, 1, B>
<C, 2, B>
```

Then the link between A and B is broken in both directions:

```
> fail A B
> fail B A
```

When the periodic exchanges are resumed, C tells B that it has a route to A through B, and B tells C that it has a route to A through C. This exchanges go on in a loop, and the cost to A keeps forever increasing.

```
> run
> table A
^C
```

Note: Haskell is a lazy language where computation only happens when the resulting data is needed. If evaluation was strict, the `run` command would enter a loop. But because the resulting information is not needed until the routing table for A is displayed, the looping occurs on the `table A` command.

Defining a low *infinity* would prevent the looping behaviour, which would terminate as soon as the cost of the route to A reaches this number.

3.2.2 With split-horizon

As in the previous case, periodic exchanges are ran until the network is stabilised and then the link between A and B is broken:

```
> run
> fail A B
> fail B A
```

Before resuming the periodic exchange, the split-horizon feature is activated:

```
> split-horizon
```

This prevents C from sending to B routes that have B as gateway. Likewise, it prevents B from sending to C routes that have C as gateway. The looping behaviour is therefore avoided and A is successfully purged from the routing tables:

```
> run
> table A

> table B
<C, 1, C>

> table C
<B, 1, B>
```

4 Status report

Netsim contains no *known* limitations or bugs. The functional requirements have been fulfilled as follows:

- *Compute routing tables for any preset number of exchanges or until stability is achieved.*

  The command `tick` will simulate a single exchange. The command `run` will simulate exchanges until stability is reached. The command `run N` will simulate `N` exchanges or stop when stability is reached.

- *Preset any link to change cost or fail after any chosen exchange.*

  The command `fail S D` will make the link `S` $\rightarrow$ `D` fail. The command `cost S D N` will set the cost of the link `S` $\rightarrow$ `D` to `N`. Note that the links are assumed to be directional. These commands will have to be ran once in each direction to simulate changes in both directions.

- *View the best route between any source and destination after any chosen iteration.*

  At any point during the simulation, the command `route S D` can be used to query for the best current route from `S` to `D`.

- *Trace the routing tables of any set of nodes for any specified number of iterations in a way that can be easily viewed.*

  At any point during the simulation, the command `table S` can be used to view the routing table of node `S`.

- *Engage, on request, a split-horizon capability to help combat slow convergence.*

  At any point during the simulation, the command `split-horizon` can be used so that all nodes generate distance vectors with split-horizon activated.