Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 2
Netchange

Trevor L. McDonell & Ivo Gabe de Wolff

Deadline: Saturday, 21 December, 23:59

The internet is a large network consisting of many connected computers. You can find all intercontinental connections on this map:

`https://submarine-cable-map-2016.telegeography.com`

Not all pairs of countries have a direct connection, so some packets may need to travel through other countries. More generally, packets usually need to travel via many computers. For performance it is thus important to find the fastest route (fewest number of hops). Furthermore, the structure of the network may change frequently, when links between computers are created or broken. The Netchange algorithm can discover the structure of the network. The algorithm is adaptive: when links are created or broken, the algorithm will propagate this information through the network. This way, each computer in the network will know the best routes to all other computers.

Network

A network contains of a set V of processes (the computers). A direct connection between two neighbouring processes u and v implies that u can send messages to v and the other way around. If u is connected with v and v with w, then u can send messages to w via v. Computer v will then relay the message from u to w. We can also construct longer paths, but we want to find as short as possible paths (with the fewest number of hops) between two processes. We can model such graph as an undirected graph.

For efficient routing, processes must keep track of a *routing table*, denoted by Nb_u. For each process v, we should have that the first link of the path from u to v is $(u, Nb_u(v))$, if such path exists. Each process u will also keep track of the estimated distance to each other process v, denoted by $D_u(v)$. Note that $(u, Nb_u(v))$ should be a link from the network. It is thus only equal to (u, v) if there is a direct link between processes u and v.

When process u wants to send a message b to process v (where $v \neq u$), he sends the message to neighbour $y = Nb_u(v)$ on the direct connection (u, y). When y receives this message, he will send it to $Nb_y(v)$, unless $y = v$ of course. This is repeated until the message arrives at the recipient. Note that the network may change, while processes are still busy updating their routing tables.

Each process only has partial information. A process u will only know its own routing table, namely Nb_u and D_u.

In this assignment, you will write a program which uses the Netchange algorithm to construct a network with other instances of this program. You can create or remove links by sending commands in the console. You can also let processes send messages to other processes through the console.

Sockets

All processes will communicate with each other through sockets. It would thus be possible to let different processes of your program run on different machines, but for testing we will always run the programs on the same machine. You can find example code for sockets in the template and in the sockets library and in System.IO.

Threads

Your program must be multithreaded. You should have threads for:

- Interaction with the user
- Accepting new incoming connections
- Receiving messages from neighbouring processes (one thread per neighbour)

These threads need to make use of shared resources, like the routing table Nb_u. You will need some form of synchronisation for this. Part of your grade is based on how fine-grained the synchronisation is, that is, allowing multiple threads to simultaneously update parts of the routing table. However, it is of course more important to have a working implementation.

You can achieve such fine-grained synchronisation using software transactional memory (STM). It is not required to use this; you may also use MVars and/or IORefs as you wish.

Part 1: Setting up the network

Each instance of the application is one process. The processes in the network will communicate using sockets. The parameters which the program gets when starting the program, contain its port number and the port numbers of the neighbours with which it has a direct connection. All processes execute the exact same program; the programs differ in the arguments which are passed to these processes.

Initialisation

When starting the program, some command line arguments are passed which give the port number of the program, and the port numbers of its neighbours. The syntax is as follows:

```
stack run -- op np1 np2 ...
```

The arguments have the following meaning:

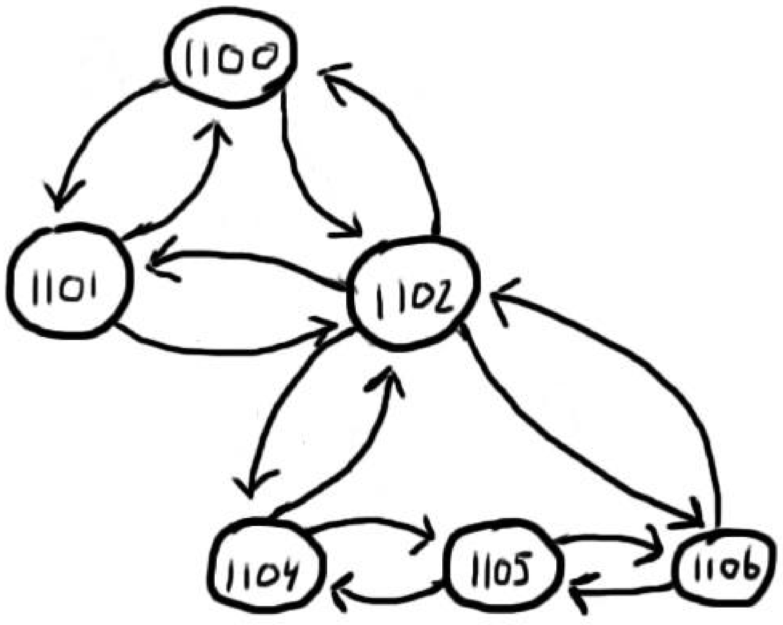
- `op`: This is the unique port number for the current process. This process thus has to listen to this port. Other processes will use this number to connect to this program. Port numbers are integral numbers in the range 1-65535; we will try to not use port numbers of frequently used applications to avoid conflicts.
- $\texttt{np}_i$: The port number of a neighbour, meaning that the process should make a connection with this port at the start of the program. This is always symmetric, which means that if $\texttt{np}_i$ occurs in the list of arguments of the process on port `op`, then `ep` will be in the list of neighbours of process on port $\texttt{np}_i$.

A single process without any neighbours is also a valid network, so the list of $\texttt{np}_i$s can thus be empty. We will never let a process connect to itself.

Example

You can create an example network using the following commands:

```
stack run -- 1100 1102 1101
stack run -- 1101 1100 1102
stack run -- 1102 1100 1101 1106 1104
stack run -- 1104 1102 1105
stack run -- 1105 1104 1106
stack run -- 1106 1105 1102
```



Part 2: Input

Each process must provide a console interface to communicate with the user. The following commands should be provided:

- **Show routing table** - `R`: Show the current routing table Nb_u. The table should contain all processes (identified by port number) which are reachable from this process, the current estimate of distance and the neighbouring process through which the shortest path goes, all separated by spaces.

```
1100 0 local
1101 1 1101
1102 1 1102
1103 2 1101
```

- **Send message** - `B portnumber string`: sends *string* to *portnumber*, after which the process on that port should print the message to the console. The string may contain spaces. If *portnumber* is not a known port number, then you should print an error message.

- **Make connection** - `C portnumber`: makes the process on port *portnumber* a direct neighbour of u. This command is only given on one of the two processes. That is, if u gets command `C` v, then v will NOT get command `C` u. Process u will thus have to communicate that with process v.

- **Disconnect** - `D portnumber`: destroys the connetion between *u* and *portnumber*. Just as with command `C`, this is only given on one of the two processes of a link. Note that this command may cause a network partition. If a port number is given, with which the process does not have a direct link, an error is printed.

We advise you to implement the commands in the same order as they were introduced here.

Part 3: Output

To enable automatic testing using TomJudge, you must follow the output specification precisely. Messages must be printed to the console on the following events:

- When the distance estimate is updated:

  `Distance to` *v* `is now` *d* `via` *h*

- When a new direct connection is constructed with the process on port *p*:

  `Connected:` *p*

- When a direct connection is removed:

  `Disconnected:` *p*

- When a process on port *p* becomes unreachable:

  `Unreachable:` *p*

- If an unknown port number is given in the `B` or `D` command:

  `Port` *p* `is not known`

- If a message is relayed:

  `Message for` *v* `is relayed to` *h*

- You can print debug information by preceding it with `//`. TomJudge will ignore these lines.

Hints

- You have to create one thread per direct neighbour. You should let these threads directly update the routing table. It is *not* allowed to place the messages on a central queue and handle them one by one.

- You have to synchronise the access to the routing table. Having fine grained synchronisation will improve your grade. We advise to use software transactional memory (STM), but it is also allowed to MVars and/or IORefs as you see fit.

- It can happen, during startup, that some of the neighbour processes hasn't started yet. In that case, you should wait with a small delay and try again.

- We will decrease your grade if you use make unnecessary use of spin locks. For a started, idle network, the CPU usage should be low. If your program is using 100% CPU then we cannot grade your submission.
- When connections are broken, a network partition may be formed. This causes the Netchange algorithm to enter an infinite loop, repeatedly increasing the distance estimate. You must prevent this and end the loop. (Hint: the program only has to handle networks of at most 20 nodes, but there are more intelligent solutions!)

General remarks

- Make sure your program compiles using `stack build`.
- Include *useful* comments in your code. Do not paraphrase code but describe the structure of your program, special cases, preconditions, et cetera.
- Try to write readable and idiomatic Haskell. Style influences the grade!
- Copying solutions from the internet is not allowed.
- You may use external packages such as `containers` or `vector` which provide *non-concurrent* functions or data structures. Any concurrency related functions or data structures you must create yourself.
- We *strongly* suggest you work in a team of size two, but individual submissions are allowed. A team must submit a single assignment and put both names on it. Submission is done using the groups facility of Blackboard.

Change Log

2019-12-03: Add some general comments

2019-12-03: Initial release