Computer Networking I

Switch assignment

Martin Sehnoutka

November 9, 2015

Contents

1. Messages format

For this assignment I defined three different types of messages. First one is for address request, because unique address for every node is one of requirements for the switch. As a consequence, there is also address respond. In this message switch accepts or denies the requested address. Last message is for normal communication between nodes. This message does not have any acknowledge because TCP takes care of this.

Every message starts with header because the switch is reading lines from an input and compare them to this header. After header is found in the input, it reads N more lines (where N is number of lines in given message type) and then check the format and content of the message.

The whole communication protocol is in plain text to preserve simplicity.

Address request from client

```
SWITCH REQUEST
Address: <node_address>
```

Switch respond to address request

```
SWITCH RESPOND
Address accepted OR not accepted
```

Message format

```
SWITCH MESSAGE
Source: <node_address>
Destination: <node_address>
<message_body>
```

2. Software architecture

In this section architecture of the whole program is very briefly described. The aim of this section is to make reading source code easier, by providing so called "big picture".

2.1 Language used

I chose Haskell language. Haskell is purely functional language with wide range of packages and great support for concurrent programming[1]. Haskell platform also contain automated build system, that is used to download all necessary dependencies and install both switch and client software.

2.2 Server side - switch

The switch program starts with initial sequence of opening network socket (localhost, port 4242) and creating necessary communication channels. Then the program flow is forked into two threads. The first one is "acceptLoop":

```
acceptLoop socket channels = do
  (handle,hostname,port) <- accept socket --waiting for incoming connections
```

In this loop program is waiting for incoming connections. The second thread is "switch":

```
switch channels clients = do
  let (controlChannel,readChannel) = channels
  input <- atomically $ select readChannel controlChannel --waiting for input
```

This functions performs the message switching between clients and also manages list of connected users.

Before a new client is allowed to use switch, it must provide its address and this address must be accepted by switch thread. For this purpose another thread is created. It is called "obtainAddress". After the address is obtained, this thread send control message to switch thread. The switch thread reads this control message and if the node address is not used, it creates another two threads for client. One is for reading from client (readFromClient) and second one is to write messages to client (writeToClient).

2.3 Client side

Clients side is much simpler. First the client opens socket to connect with switch, then sends address request and waits for a respond. If the respond is accepted, program flow is again forked into two threads: read and write. The client's side contain a lot of randomness, e.g. number of messages, payload, delays.

The client program takes 3 command line arguments. First one is ip address of switch, second one is port for incoming connections and the last one is node address.

[1] Source: www.haskell.org

```
main :: IO()
main =  do
  arguments <- getArgs
  if length arguments >= 3
    then do
      let [host, portStr, addr] = take 3 arguments
```

3. Installation on Lintula machines

First step is to add ghc [2] and cabal [3] into your path variable:

```
$ PATH=$PATH:/usr/local/lang/haskell/ghc-7.10.2/bin
```

Then it is necessary to update cabal package database. These packages will be installed into ~/.cabal directory, thus it does not need root privileges.

```
$ cabal update
```

Now it is possible to install both switch and client using cabal installer.

```
$ cabal sandbox init
$ cabal install -j
```

3.1 Clean up

After running update command, cabal directory will become really big so I always run

```
$ rm -rf ~/.cabal/
```

after I'm done with testing.

4. Testing

I wrote little script for testing purposes. This script runs switch and then in loop starts 8 clients. Output of each process is stored in different file: switch.log, client0.log, client1.log, etc.. By exploring these file I can compare messages that arrived to each client and switch.

```
./.cabal-sandbox/bin/switch &> log/switch.log &
switchpid=$!
echo "Starting switch with PID:$switchpid"

for i in `seq 0 7`;
do
    echo "Starting client $i"
    ./.cabal-sandbox/bin/client localhost 4242 $i &> log/client$i.log &
done
```

[2] Glasgow Haskell Compiler

[3] Common Architecture for Building Applications and Libraries

For example I can explore messages that arrived into switch with destination address 6 and compare them with client 6 output:

```
$ cat switch.log | grep --regexp='(Unicast) [0-9]->6' | head --lines=6
t: 17:52:36 | [client] (Unicast) 3->6: loigsbbrjisvy
t: 17:52:39 | [client] (Unicast) 7->6: rckoveje
t: 17:52:40 | [client] (Unicast) 6->6: qhtfwyygvfj
t: 17:52:40 | [client] (Unicast) 3->6: klhkyqe
t: 17:52:43 | [client] (Unicast) 1->6: jreoqbrmrukxegx
t: 17:52:44 | [client] (Unicast) 4->6: rifyndnwlkrym

$ cat client6.log | grep --regexp='For me' | head --lines=6
t: 17:52:36 | [For me    ] 3->6: loigsbbrjisvy
t: 17:52:39 | [For me    ] 7->6: rckoveje
t: 17:52:40 | [For me    ] 3->6: klhkyqe
t: 17:52:43 | [For me    ] 1->6: jreoqbrmrukxegx
t: 17:52:44 | [For me    ] 4->6: rifyndnwlkrym
t: 17:52:44 | [For me    ] 2->6: faxdvxqb
```

In this case I can see, that switch does not send messages back to their source address because the third message is from client 6, but is not present in client's log file.

I can do the same for broadcast messages:

```
$ cat switch.log | grep Broadcast | head --lines=5
t: 17:52:33 | [client] (Broadcast) 5->8: fnmyvnbu
t: 17:52:33 | [client] (Broadcast) 0->10: kokeiym
t: 17:52:34 | [client] (Broadcast) 6->10: ksonkagrgeuzwvt
t: 17:52:35 | [client] (Broadcast) 0->9: cowmv
t: 17:52:35 | [client] (Broadcast) 6->9: erfozavxllymd

$ cat client7.log | grep Broadcast | head --lines=5
t: 17:52:33 | [Broadcast ] 5->8: fnmyvnbu
t: 17:52:33 | [Broadcast ] 0->10: kokeiym
t: 17:52:34 | [Broadcast ] 6->10: ksonkagrgeuzwvt
t: 17:52:35 | [Broadcast ] 0->9: cowmv
t: 17:52:35 | [Broadcast ] 6->9: erfozavxllymd
```

I tried this project on Lintula machine:

```
$ uname -a
Linux pikkukorppi 2.6.32-573.7.1.el6.x86_64 #1 SMP Thu Sep 10 13:42:16
EDT 2015 x86_64 x86_64 x86_64 GNU/Linux
```