Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CS2031 Telecommunication II
Assignment #2:OpenFlow

Jerzy Jaskuc, Std# 17341645

December 29, 2018

Contents

1 Introduction

The main task was to design and implement a model of an OpenFlow system, where the Controller is responsible for sending routing information to *Router*s when needed as well as managing the flow of packets inside of the network. The network topology was left to our decision.

2 Theory of Topic

In OpenFlow, most of the routing work is done by the *Controller*. The *Controller* can either be connected to one of the *Router*s in the network, which is the usual case, or have a separate connection to every *Router* in the network. The problem with one connection is that if that *Router* fails, none of the *Router*s in the network can access the *Controller*. Futhermore, if the rout is lengthy, there's more chance that a packet will be lost somewhere on the way. Thats why i decided to use the case where every *Router* has a connection to the *Controller*. In reality it would be harder to implement, but as a programmed model it's actually easier.

2.1 Routers

In OpenFlow, *Routers* don't have much to do. They keep their routing table where they can see the next destination, when they need to forward the packet with a given *End User* destination. They also send information requests to the *Controller* when routing information is needed. In the *Link State Routing* they also have information about their neighbours, which they send to the *Controller* when they get online.

2.2 Controller

Another question is, which routing system to use on the *Controller*? There are 3 ways to do it:

- To use *Preconfigured Routing Table* on the *Controller*.
- To use *Distance Vector Routing*.
- To use *Link State Routing*.

In the first case, the *Controller* have hardcoded information about the whole network topology and predefined routes from and to each of the *End User*s. This one requires a lot of hardcoding but do not require any complicated algorithms. Adding extra *Router*s and *End User*s would be problematic too. In *Distance Vector Routing*, every *Router* would exchange the information about his neighbours with other *Router*s and the *Controller*. After a few exchange Cycles, *Router*s and *Controller*s would converge. At this point *Controller* also has the paths from one point to another. However, when the network grows bigger and when adding extra *Router*s or *End User*s, the information might take a while to spread across the network. Also the need in *Controller* is questionable, since after exchange of information with the neighbours, *Router*s would also have the routing tables in them. The last way, which is *Link State Routing* and which is the one i've chosen, works in a way where every *Router* has an information about it's neighbours. When they go online, they would send their information to the *Controller*s, which will store it for rout calculation when needed. With that, new *Router*s and *End User*s can be added without much difficulty, since they would just send their information to the *Controller*, which would use it when calculating routs.

2.3 End Users

End Users are nodes that are interacting with the *Users* by taking their input, and then sending the message to the neighbouring *Router*, which forwards it until it reaches the destination. They are the only part of the system, that interacts with the *User*. In my implementation, they have pre-configured neighbouring *Router*.

3 Implemenation

As for my implementation, I decided to do it entirely in **Java**, on a **localhost** where every node has it's own port occupied. This avoids any troubles coming from setting up the system on containers or virtual machines.
Since the topology was left to us, i decided to have 7 *Routers* and 4 *End Users*. The connections between them are as shown on figure 1.
The whole Topology is started when the program `Network` is started. Every router then exchanges `HELLO`s and `FEATURE`s. When the network setup is complete, it is indicated with a message. Example network console output during setup process is shown on figure 2
Some of more crucial exchanges use Stop & Wait ARQ, but some less important don't use any flow control. To implement the flow control `Timer` and `TimeoutTimer` classes are used.

3.1 Router Implementation

In this implementation, *Routers* keep the information about their neighbours in `RoutingInfo` class designed for that. When going online *Router* will send `HELLO` message to the controller and will keep resending it until it gets `FEATURE_REQUEST`. After that is will send information about its neighbours and wait for any incoming packets. The routing information is stored in `HashMap` where every final destination is connected to the address of the next hop, that current *Router* should send to. If the routing information is required for

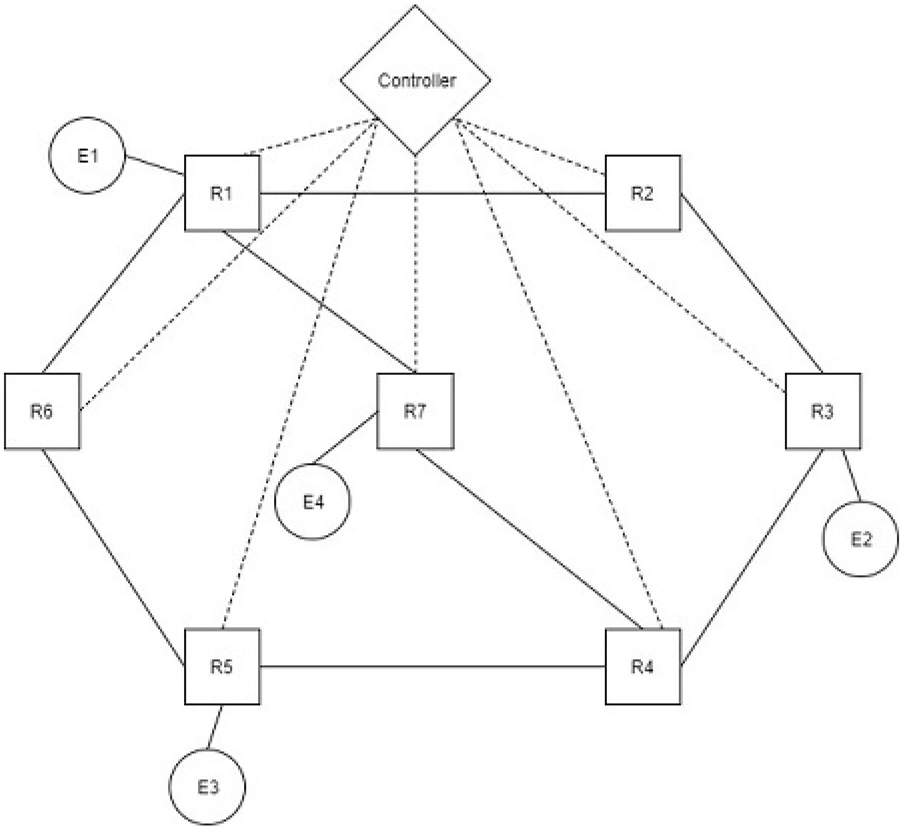


Figure 1: This figure shows the network topology of implemented network

recieved packet, the *Router* will send the information request to *Controller*. Otherwise it will just forward the packet and send ackknowledgement to the sender.
This acknowledgement is crucial, since the downside of my implementation is that if no routing information is found, the request will be sent but the original packet will be dropped. This is due to the problem of being unable to stop the `Thread` and wait for information, because then the `onReciept` method will be locked. With acknowledgement not recieved, it will be resent and forwarded successfully, since routing information, if theres any, will arrive by that time.

3.2 Controller Implementation

The *Controller* is the main thing of this system. It recieves all of the information from the *Routers* and then uses it to calculate routs and sends out routing information to the *Routers*. All of the routing information is stored in `HashMap` where the key is the *Router* that sent it and the value is the `ArrayList` of it's neighbours. The *Controller* uses modified BFS based on the `Queue` ADT to calculate the rout. Modified BFS is used instead of Dijksta, since the distances between routers in this case are unweighted, which makes Dijkstra obsolete. When the rout is calculated, the routing information is sent not only to requesting router, but also to every *Router* on the path. This saves recalculating same rout again and again.
Figure 3 shows the sample for exchanging the information packets between *Router* and *Controller*.
And figure 4 shows the example forwarding process.

3.3 End Users

End Users are the nodes that take input from a user, and then send the message to the neighbouring *Router* that will forward it until it reaches desired destination. *End User* uses separate thread to get an input, so it won't interrupt getting an acknowledgement back. *End user* have to be started manually, but as many

```
Network [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\jav
Controller online!
Connection request sent!
Connection with router: R1, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R2, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R3, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R4, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R5, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R6, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Connection request sent!
Connection with router: R7, established!
Feature request sent!
Feature from a router recieved!
Feature exchange completed succesfully!
Network setup completed!
```

Figure 2: This figure shows the example console output during network setup process

End Users can be started as needed. However, every *End User* has a predefined *Router* address, which in my implementation is every second *Router*.

3.4 Used Packets

There are total of **10** packet types used, therefore **4 bits** of overhead was needed. The packet types are as follows:

- `HELLO` packet - **0000** - used to establish connection.
- `HELLACK` packet - **1111** - acknowledgement of connection, sent only by *Router* to *End User*.
- `FEATURE_REQUEST` packet - **0001** - request of information about neighbours from *Controller* to *Router*.
- `FEATURE` packet - **0010** - information about neighbours.
- `FEATACK` packet - **0100** - acknowledgent of reciept of the `FEATURE` packet.

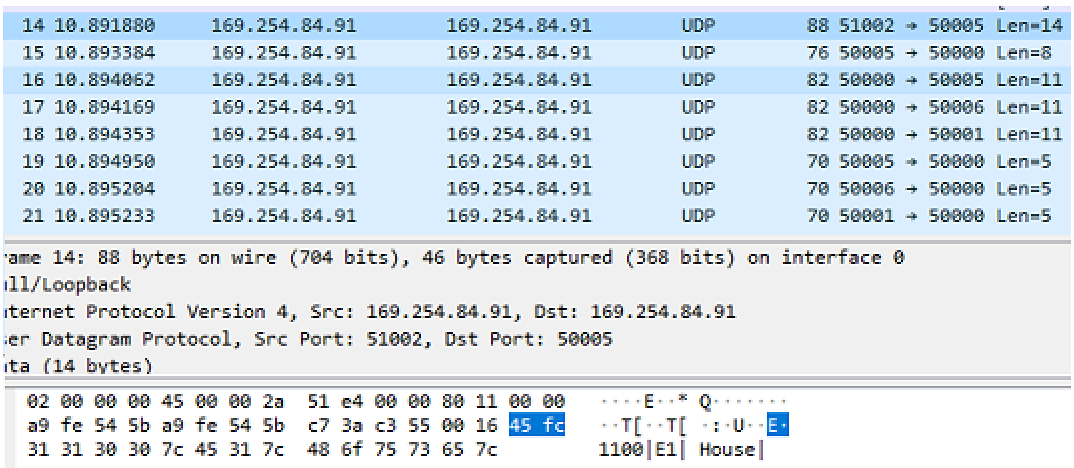


Figure 3: This is an example of packet exchange from Wireshark view. End User with port **51002** sends a message to Router **50005**. Router then sends information request to Controller **50000**, which then sends routing information to all routers on the rout (**50005,50006,50001**) and get acknowledgements back.

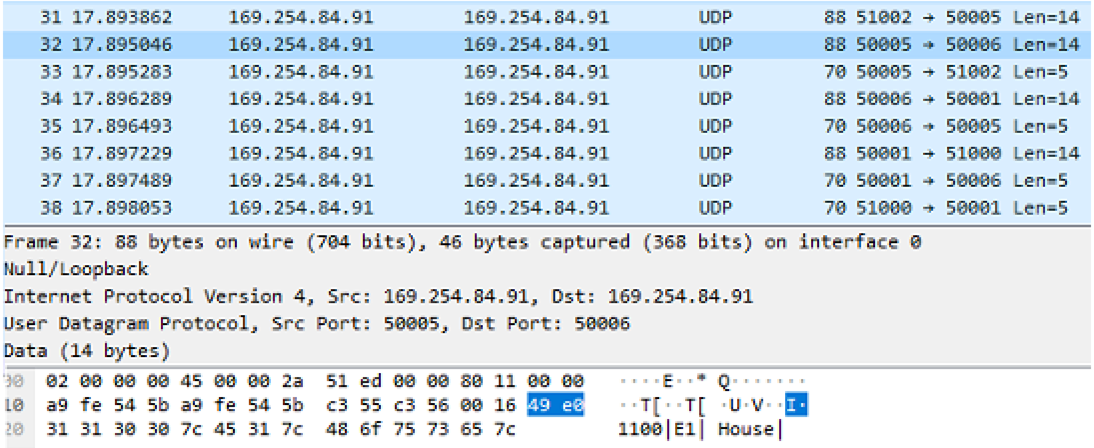
- `INFOREQUEST` packet - **1000** - request for routing information.
- `INFO` packet - **0011** - routing information.
- `INFOACK` packet - **0110** - acknowledgement of reciept of the routing information.
- `SEND` packet - **1100** - packet with a message from the *End User*.
- `SENDACK` packet - **0111** - acknowledgement for `SEND` packet.

4 Summary

Summarizing, my implementation of OpenFlow uses *Link State Routing* to run. The *Controller* uses modified BFS to calcualte the shortest rout and spread the routing information to all relevant routers. Thanks to that, the code does not contain much of the hardcoded elements. The only pre-configured parts are the neighbours of each *Router*, so basically the topology of the network. This also allows to dynamically add more *Routers* or *End Users*. Overall, the system is pretty stable, quite intuitive, and models the OpenFlow quite well. However, as usual, there's still room for improvement.

5 Reflection

Overall, I'm really happy with my design. I wanted 2nd Assignment to be improvement in my coding skills and large projects management. And i can see the improvements. I used `Constants` a lot more, commented the important or less understandable parts of the code more and tried to hard code as least as possible. I've spent around 50-55h hours on this assignment, spread across around 3 months, but i very much enjoyed doing it. I feel like i learned a lot about telecomms in general on a conceptual level by doing this assignment as well as learned some useful tricks in Java. I've spent more time designing than in my 1st assignment as I planned and it helped a lot. The hardest part of the assignment was the proper exchange of information. I had problems with deciding on the format of encoding. The other hard part was in creating different instances of the same class, like *Routers*. Implementing the routing algorithm however, was easier than I expected. It took me a lot of time to design it but it was worth it. I also skipped some design decisions in the report, that are the same as in Assignment 1, since i didn't really want to repeat it all and i reused a lot of code from Assignment 1. I also had to not include some of the console output figures since they were too big even when cut in size.



31 17.893862	169.254.84.91	169.254.84.91	UDP	88 51002 → 50005 Len=14
32 17.895046	169.254.84.91	169.254.84.91	UDP	88 50005 → 50006 Len=14
33 17.895283	169.254.84.91	169.254.84.91	UDP	70 50005 → 51002 Len=5
34 17.896289	169.254.84.91	169.254.84.91	UDP	88 50006 → 50001 Len=14
35 17.896493	169.254.84.91	169.254.84.91	UDP	70 50006 → 50005 Len=5
36 17.897229	169.254.84.91	169.254.84.91	UDP	88 50001 → 51000 Len=14
37 17.897489	169.254.84.91	169.254.84.91	UDP	70 50001 → 50006 Len=5
38 17.898053	169.254.84.91	169.254.84.91	UDP	70 51000 → 50001 Len=5

```
Frame 32: 88 bytes on wire (704 bits), 46 bytes captured (368 bits) on interface 0
Null/Loopback
Internet Protocol Version 4, Src: 169.254.84.91, Dst: 169.254.84.91
User Datagram Protocol, Src Port: 50005, Dst Port: 50006
Data (14 bytes)
00  02 00 00 00 45 00 00 2a  51 ed 00 00 80 11 00 00   ····E··* Q·······
10  a9 fe 54 5b a9 fe 54 5b  c3 55 c3 56 00 16 49 e0   ··T[··T[ ·U·V··I·
20  31 31 30 30 7c 45 31 7c  48 6f 75 73 65 7c         1100|E1| House|
```

Figure 4: This is an example of forwarding process from Wireshark view. For every forwarded packet, acknowledgement is sent back to the sender. This activity is repeated until packet reaches it's destination.

Of course, there are still ways to improve the system, like: include the check for dead routers, include some additional flow control or fix some socket issues. But all in all, the program is pretty realiable, compact and stable.