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

CS2031 Telecommunication II
Assignment #2: OpenFlow

Jevgenijus Cistiakovas,

December 22, 2018

Contents

1 Introduction

This report describes the design and implementation of a version of an OpenFlow protocol.

2 Theory of Topic

This section describes the theory behind the design and implementation. It also talks about the decision and assumptions made during the design phase.

2.1 OpenFlow

The task is to implement an OpenFlow protocol. This includes the protocol itself, the actors that use the protocol, and everything else that is needed for protocol to operate correctly. At the start, I should look into specifications of the OpenFlow and recognise everything that needs to be considered.

OpenFlow is a protocol that allows for an alternative to traditional routing. Specifically, it is used to implement Software Defined Networking (SDN). The key idea behind SDN is to separate routing process(control plane) from a forwarding process(data plane). That is the task of path finding is given to a network controller, while traditionally it is done by a router. Network controller and router are different entities. A network controller determines the paths that packets should take, while router simply routes packets based on information provided by the controller. OpenFlow protocol defines how controller and router should exchange messages. A controller uses OpenFlow to ensures that all the routers in the network follow correct routing logic.
Normally, a router is connected to a controller via a separate logical connection.

2.2 Types of Messages

The task is to design an implementation of OpenFlow protocol that will allow communication between a controller and routers. I used provided sample message types as a starting point. However, there were more than 20 message types provided, while I need much less for my simplified protocol. I identified message types that my protocol needs to support:

- HELLO message that is used by router to initiate communication with a controller
- FEATURE_REQUEST that is used by controller to request information about the router such as who are its neighbours and what are its interfaces.
- FEATURE_REPLY that is used by a router to sends its features to a controller in response to a FEATURE_REQUEST message.
- SET_CONFIG that is used by a controller to update flow table of a router.
- GET_CONFIG_REQUEST that is used by a controller to request full flow table from a router.
- GET_CONFIG_REPLY that is used by a router to send a full flow table to a controller in response to a request.
- ECHO_REQUEST that is used by one party to ping another party.
- ECHO_REPLY that is used to reply to ECHO_REQUEST.

- PACKET_OUT that is used by a router when it does not know how to route a packet and requires help from a controller.
- PACKET_IN message that is sent by controller to a router that previously sent a PACKET_OUT. This message should only be sent after the controller has found a path to destination and has updated all the routers on that path.
- ACK message that is used to acknowledge a successful delivery of a message. Reply messages can serve this role for request messages. However, acknowledgement is required for reply messages themselves.
- ERROR message that can be used to notify about any errors.

It should be noted that I decided to use a generic ACK message, since its function is to only acknowledge the delivery of a message.

2.3 Layers of abstraction

OpenFlow protocol normally operates on network devices such as routers. Routers have a forwarding table that operates on levels as low as link level. However, my implementation will not run on an actual router, but will simulate a router on an application level. Hence, I do not have access to physical interfaces that are used for sending out packets to the neighbours. I decided to implement an Interface class to simulate a physical link between the router and its neighbours. Interface class contains an ip address and port of the neighbouring routers to which a router can send packages. While a real table will have actual interface addresses.

The protocol is to be run over UDP(User Datagram Protocol). That means nodes will use UDP packets inside of IP(Internet Protocol) packets for communication. Using standard methods I can only specify IP address and port number of the final destination. I have no control over how the packet will reach that destination. But I would like to pretend that I have control over routing. For that reason I decided to wrap a datagram into another packet that will store both the address of a next hop and of final destination.

2.4 Table Format

Based on the example provided, I decided to implement a simplified flow table. I do not include statistics section in my table and I do not include action section. Actions are selected based on whether an entry for a specific input exists. If an entry exists, then the action is to forward the packet to an interface port specified in the table. Otherwise, the router attempts to get such an interface port from the controller. A third option could be to just drop the packet, but I decided not to implement it.

2.5 Link State Routing - Graphs and Dijkstra

I decided to implement a version of link state routing. Initial plan was to have each router to send all the pre-defined information that it has about its neighbours, so that the controller would be able to construct a topology og the network. Knowing topology of the network, the controller would be able to calculate the best path between each end node and routers.

I identified this to be a typical graph problem, where I would need to build a graph and then find the best path. As the algorithm to calculate the best path, I decided to use Dijkstra's shortest

path algorithm as it should be relative easy to implement, while it is also relatively efficient. I realise that there are more efficient algorithms that use adjacency matrix representation of a graph, however, for this task Dijkstra's algorithm should be good enough.
There are other algorithms available, but I chose to use the simplest one. Since efficiency is not a concern here.

2.6 Flow and Error Control

I decided not to implement any ingenious flow control. I use buffers at application level that simulate transport layer buffers. I use two buffers: one for sending and one for receiving.
The OpenFlow protocol works on top of transport level. In my implementation I use UDP as a transport layer protocol, which itself uses one of the IP protocols as network layer protocol. None of these protocols as well as a link layer protocol implement any error control other than checksums that ensures integrity of a packet. I decided to implement support for Stop-and-Wait ARQ with sequence and acknowledgement numbers. Acknowledgements packets are not needed since most messages are request messages and have a corresponding reply message that can also be viewed as an acknowledgement. This allows a sender to know whether a packet was delivered correctly. I included support sequence numbers to ensure that packets will be processed in an expected order, since UDP is a connectionless protocol. It should be noted that I decided not to implement flow and error control as it is not essential to concept of OpenFlow and SDN. However, the protocol I designed supports addition of flow control if needed.

2.7 Extra Material

This subsection talks about possible extensions to the current design that were not implemented.

2.7.1 Multiple controllers

For this assignment I assumed that a network only has one controller device. However, it is also possible to have multiple controllers. Moreover, multiple controllers enable for better SDN network.[1] It removes the issue of single point of failure as well as it allows for better scalability and availability.

3 Implemenation

This section consists of an overview and explanation of the implementation. Overall, the project consists of five main parts: packets, flow table, controller, switch, end node.

3.1 Packets

I chose to implement packets in an Object Oriented way. That is each unique packet is represented as a class and is processed at a node as a class. As a starting point I used PacketContent and AckPacketContent classes provided. I extended and modified the classes to be used for the project.

3.1.1 Open Flow packet

The classes for OpenFlow protocol create a hierarchy as on 1.
All specific message type classes extend from abstract class OFPacket. It defines the general attributes of a protocol's header such as version of a protocol, connection id to identify the sender, sequence number to allow for flow control and type to identify the type of packet. It also defines

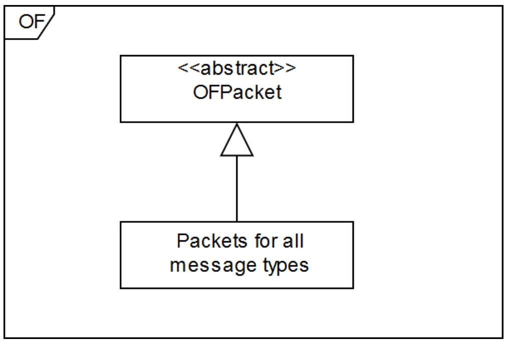


Figure 1: OpenFlow packets hierarchy

tow important methods: toDatagramPacket() and fromDatagramPacket() that are used for conversion between DatagramPacket class and OFPacket subclasses. This is required as sockets in Java operate on DatagramPackets. Each message type is implemented as a subclass of OFPacket. They define attributes specific to the message type and provide methods for packing and extracting this data from an object stream that will be inserted into a DatagramPacket. The description of packet types can be found in section 2.2.

The attributes for each individual message type can be found in the source code, but I will elaborate on few of them. Specifically, FEATURE_REPLY message carries information about all the connections of a specific switch. This is implemented by packing an array of Interface objects into a packet. SET_CONFIG packet carries an entry that is to be placed into the switch's own flow table. This is implemented by packing an array of RouterFlowTableEntry objects. More on RouterFlowTableEntry in section 3.2. PACKET_IN and PACKET_OUT carry a full packet that is to be forwarded. However, DatagramPacket cannot be serialized in Java, so all the atomic components that define it are extracted and packet into a message.

3.1.2 Generic packet

Encapsulation is one of the key principles in current networking model. Encapsulation is essential for routing as it allows to differentiate between the final destination address and the address of the next hop that is directly connected. However, in this project communication occurs at transport layer. Specifically, UDP socket in Java only accepts the final destination IP address and performs the routing for you. That is why I created a GenericPacket class that is to encapsulate any DatagramPacket and at the same time have an attribute of the final destination address. A lot of encapsulation and wrapping occurs here that certainly does not improve the efficiency. The idea is that a node can send a DatagramPacket to the next hop, where this DatagramPacket will encapsulate GenericPacket with final destination address and the actual payload. Diagram at 2 demonstrates the idea.

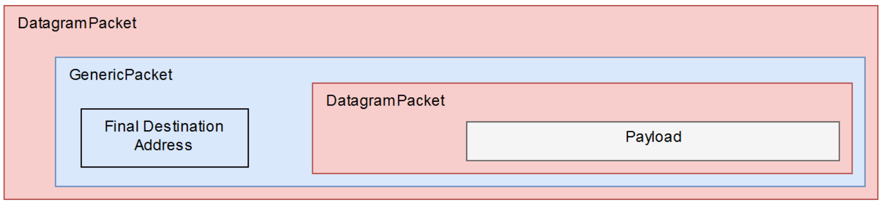
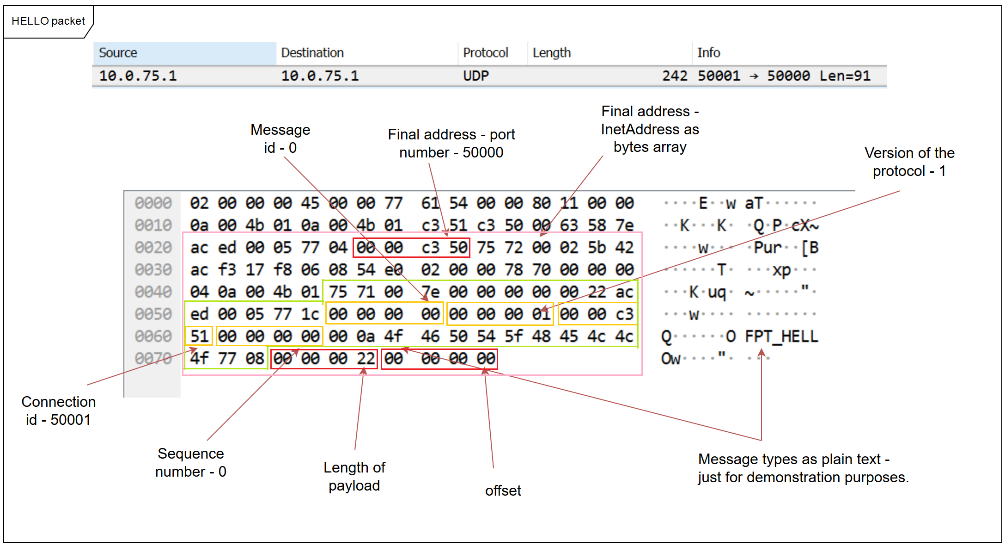


Figure 2: Encapsulation of packets

3.1.3 Demonstration

The shapshots of some actual packets exchanged are presented here. Snapshots were obtained via Wireshark [2].


HELLO packet
Message id - 0
Final address - port number - 50000
Final address - InetAddress as bytes array
Version of the protocol - 1
Connection id - 50001
Sequence number - 0
Length of payload
offset
Message types as plain text - just for demonstration purposes.

```
Source        Destination   Protocol  Length  Info
10.0.75.1     10.0.75.1     UDP          242  50001 → 50000 Len=91
```

```
0000  02 00 00 00 45 00 00 77  61 54 00 00 80 11 00 00   ····E··w aT······
0010  0a 00 4b 01 0a 00 4b 01  c3 51 c3 50 00 63 58 7e   ··K···K· ·Q·P·cX~
0020  ac ed 00 05 77 04 00 00  c3 50 75 72 00 02 5b 42   ····w··· ·Pur··[B
0030  ac f3 17 f8 06 08 54 e0  02 00 00 78 70 00 00 00   ······T· ···xp···
0040  04 0a 00 4b 01 75 71 00  7e 00 00 00 00 00 22 ac   ···K·uq· ~·····".
0050  ed 00 05 77 1c 00 00 00  00 00 00 00 01 00 00 c3   ···w···· ········
0060  51 00 00 00 00 00 0a 4f  46 50 54 5f 48 45 4c 4c   Q······O FPT_HELL
0070  4f 77 08 00 00 00 22 00  00 00 00                  Ow····". ···
```

Figure 3: Insides of HELLO packet. In pink is the payload of UDP packet. Final address - port number in red shows the start of GenericPacket. In green is the OFPacket. HelloPacket class constitutes no data here.

3.2 Flow Table

Flow table is a routing table present in both switch and controller. However, flow table for switch is not the same as for controller. Specifically, switch's flow table is much simpler as a switch only needs to know where to output a packet based on the final destination, whereas a controller needs to keep a full path for any combination of source and destination. Thus I created a generic class FlowTable that can operate on any type of entries and that provides methods addEntry() and getEntry()
As a starting point for my flow tables I used the sample tables provided in a description of the assignment. Below are the examples of structure of the tables used.

Table 1: General format of Flow Table

Source	Destination	Entry

Table 2: Router Flow Table Entry

Destination	In Interface Id	Out Interface Id

Table 3: Controller Flow Table Entry - each entry consists of multiple entries depicted in the table that define a full path

Router ID	Next hop address	In Interface ID	Out Interface ID

3.3 Controller

Controller is represented by a Controller class and its three helper classes ControllerListeningThread, ControllerProcessingThread, ControllerSendingThread. Controller does not interact with a user and quietly runs as a background process.

Controller has different attributes required for its operation. It stores a whole network as a directed weighted graph with the help of a Graph class that I created. Controller stores its own flow table in FlowTable class with FlowTableEntry that acts as a container for ControllerFlowTableEntry. It stores two hash tables to store information about the routers it is managing, as well as it stores some other attributes such as protocol version it uses and its own ID, network address, etc.

I identified three functions that controller needs to perform and separated them into three threads.

ControllerListeningThread listens for incoming OpenFlow packets, it discard any other packets. It also performs some checks and data manipulations before putting the packet into a processQueue that is implemented as a LinkedBlockingQueue object.

ControllerProcessingThread is the thread that does the main work in Controller. It identifies the type of an OpenFlow packet received and acts based on that. Specifically, when it receives HELLO packet, it sends a FEATURE_REQUEST. When it receives FEATURE_REPLY packet it processes the payload and inserts/updates the sender node, sender node's neighbours and the edges into a network graph as well as storing some other information. When it receives PACKET_OUT from a router, it updates its own flow table if it is out of date and then uses this flow table to update the flow tables in routers that are on the path to the destination specified in PACKET_OUT. Controller SET_CONFIG packets to issue updates to routers. Finally, when routers on a path are updated and ready, it sends a PACKET_IN packet back to the initial sender.At this stage initial sender and all the other routers on a path should be able to forward the packet without contacting the controller. The process of finding a path is described in 3.3.1. I only implemented support for these packets as they are the ones without which protocol would not work. Other packet types defined in Message Types (2.2) can also be implemented if needed. It should be noted that no explicit flow control is used, but it can be added as protocol supports it.

ControllerSendingThread continuously takes any packet that is put into a sendQueue, performs necessary encapsulation into a DatagramPacket and sends the packet to the receiver, which in this case should be a router.

3.3.1 Link State Routing

One of the features of my implementation is that the flow tables are not preconfigured, but rather are calculated dynamically. I decided to use link state routing as each router knows its own link state and can easily share it with a controller. The difference to traditional link state routing is that full network topology is only built at the controller and it is the controller that performs best path calculations.

As I stated earlier the network is stored as a directed weighted graph in a Graph object. I decided to use Dijkstra algorithm to build a single source shortest path tree for each node in a graph and based on that update the entries in a flow table. builFlowTable() method of the Controller class does this calculation. As this is one of the key components of the program, I created a JUnit test

GraphTest for Graph class that also tests Dijkstra's algorithm.

3.4 Switch

Switch is represented by a Router class and three helper/thread classes ListeningThread, ControllingThread and SendingThread. The structure is very similar to Controller class. Functions are split up between three threads that communicate with the use of LinkedBlockingQueue.

Main attributes of a Router class apart from its own id, address and protocol version are default controller address, flow table stored in a FlowTable object with RouterFlowTableEntry as entries and interfaces stored in a hash table of Interface objects.

ListeningThread listens for incoming packets. If a packet has router itself as its final destination and is an OpenFlow packet that means it was sent by controller and it needs to be processed by a controlling thread. If a packet is just a GenericPacket with final destination not being the router then it needs to forward this packet to next hop. The thread calls a hasRoute() method of a Router that checks whether there is rule in a flow table that tells what is the next hop. If a route is known then the packet is simply transferred to sendingThread, otherwise it is sent to ControllingThread that should make the routing possible.

ControllingThread is the most important thread as it is what implements OpenFlow switch. When ControllingThread is stated it first attempts to connect to default controller by sending HELLO packet. It then waits for FEATURE_REQUEST from controller and replies with FEATURE_REPLY packet. After this connection is considered to be established and the threads goes into resolving incoming packets. The controlling thread sends a PACKET_OUT for each packet that it does not know how to forward and it processes all the packets received from the controller. One of the most important packet types is SET_CONFIG which makes the router to update its flow table.

SendingThread has a simple function. It takes packet that are to be sent from sendQueue, performs encapsulation, finds the out interface and sends the packet using that interface.
Intially, router was implementing stop-and-wait flow control. However, it was later removed as to make the operation of a router easier to understand.

3.5 End-Node

End-node is represented by a class EndPoint and its helper classes/threads EndPointInputThread and EndPointOutputThread. EndPoint spawns and uses two terminals for interaction with a user. Terminal class from tcdIO is used.

End-node is the simplest out of three nodes implemented. Its high level functions include sending a packet to other end-node and receiving packets from other end-nodes. In my implementation end-node is not a switch and vice versa. Thus it does not perform any routing. Each end-node has a single associated default switch, to which it forwards all the packets. This default switch is specified in a constructor of the EndPoint class.

Endpoint has two helper threads that perform the two main functions. EndPointInputThread gets a message and an address of receiver from a terminal, performs required encapsulation by first creating a DatgramPacket, then packing it into a GenericPacket and then sending it to default router as a

DatagramPacket. The receiver's address is recorded in a GenericPacket. EndPointOutputThread listens for incoming GenericPackets. On the receive, it extracts payload and source address from the packet and outputs this in a terminal.

3.6 Discussion

The program successfully demonstrates the operation of SDN and OpenFlow. However, this is only a demonstration of a concept and in no way this can be used for actual routing. Many simplifications were made to allow finish the program within a given time frame and to keep it relatively simple to understand. Flow control was left out as it is not essential to the concept of SDN. Programs communicate over a link local network and use ports for addresses as this allows for simple development, testing and demonstration. Efficiency of the protocol was not the dominating factor in development of the protocol. I prioritized ease of use and understanding, and readability over efficiency. That is why object oriented approach was used for packets, where bit manipulations could have been used if efficiency was a concern. The multiple encapsulation of DatagramPackets in GenericPackets and again into DatagramPackets definitely is not efficient and adds a lot of overhead. Efficiency of algorithms was taken into account, but not much emphasis was put on it. I acknowledge that dynamic path calculation is not implemented in a most efficient way.

I did not implement support for all the packet types that I defined at the start as well I intentionally left out most of the packet types provided in specification of the assignment. These packet types allow for flow control, error control, debugging and more. For example, ECHO request and reply packets can be used for polling of router and controller to determine whether it is alive, which can be used to determine failure in a network. GET_CONFIG packets can be used for debugging as it allows to inspect a routing table of a router. ERROR message can be used for debugging.

4 Demonstration

A demonstration of operation of all the classes together is provided in an OpenFlowDemo class. It has two predefined network topologies. The class creates all the objects and starts them. The network defined by difficulty 2 is used for demonstration (4).

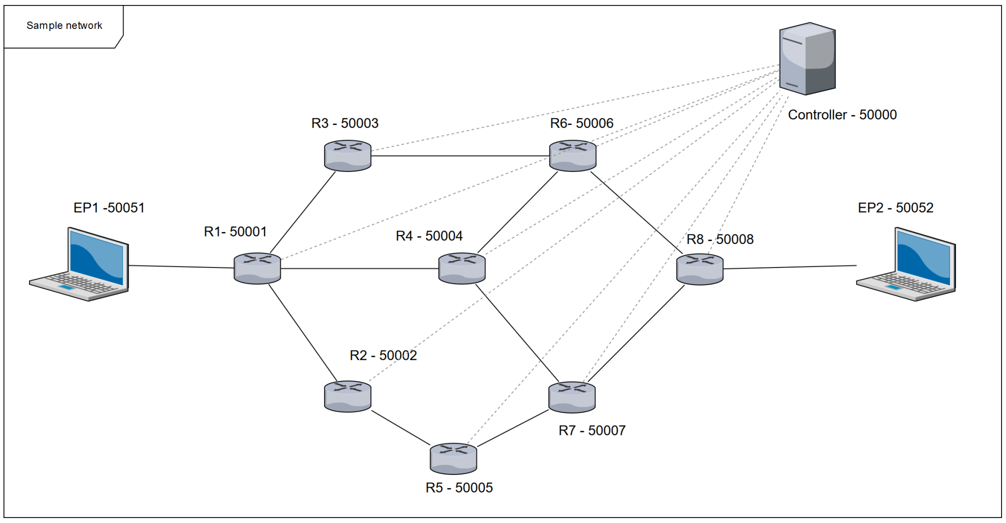
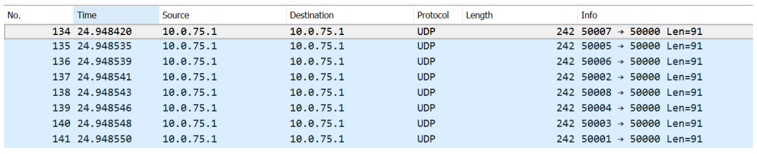


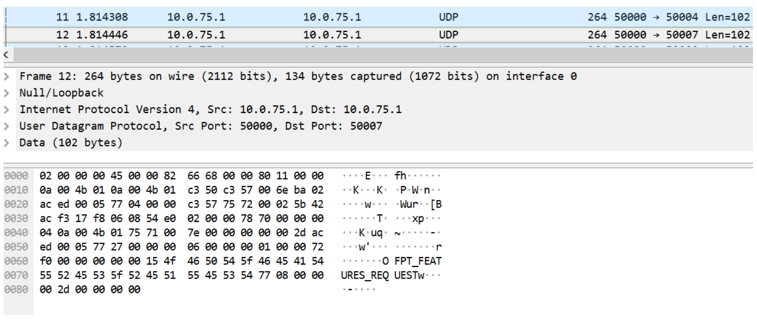
Figure 4: Sample network with 8 routers and 2 endpoints. Number beside names are port addresses.

When demonstration is started all routers send HELLO pakcet to controller as can be seen in (5). Controller then sends FEATURE_REQUEST to each router as can be seen in (6).



```
No.        Time         Source         Destination    Protocol  Length   Info
       134 24.948420    10.0.75.1      10.0.75.1      UDP            242 50007 → 50000 Len=91
       135 24.948535    10.0.75.1      10.0.75.1      UDP            242 50005 → 50000 Len=91
       136 24.948539    10.0.75.1      10.0.75.1      UDP            242 50006 → 50000 Len=91
       137 24.948541    10.0.75.1      10.0.75.1      UDP            242 50002 → 50000 Len=91
       138 24.948543    10.0.75.1      10.0.75.1      UDP            242 50008 → 50000 Len=91
       139 24.948546    10.0.75.1      10.0.75.1      UDP            242 50004 → 50000 Len=91
       140 24.948548    10.0.75.1      10.0.75.1      UDP            242 50003 → 50000 Len=91
       141 24.948550    10.0.75.1      10.0.75.1      UDP            242 50001 → 50000 Len=91
```

Figure 5: Hello phase.



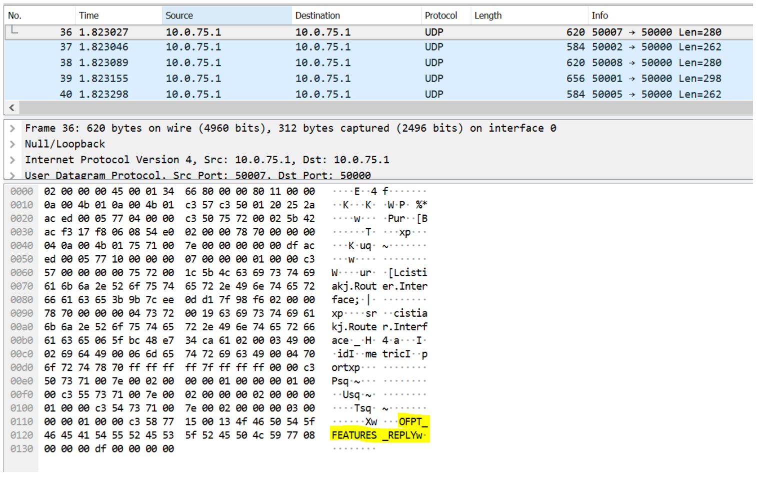
```
        11 1.814308     10.0.75.1      10.0.75.1      UDP            264 50000 → 50004 Len=102
        12 1.814446     10.0.75.1      10.0.75.1      UDP            264 50000 → 50007 Len=102

> Frame 12: 264 bytes on wire (2112 bits), 134 bytes captured (1072 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 10.0.75.1, Dst: 10.0.75.1
> User Datagram Protocol, Src Port: 50000, Dst Port: 50007
> Data (102 bytes)

0000  02 00 00 00 45 00 00 82  66 68 00 00 80 11 00 00   ····E···  fh······
0010  0a 00 4b 01 0a 00 4b 01  c3 50 c3 57 00 6e ba 02   ··K···K·  ·P·W·n··
0020  ac ed 00 05 77 04 00 00  c3 57 75 72 00 02 5b 42   ····w···  ·Wur··[B
0030  ac f3 17 f8 06 08 54 e0  02 00 00 78 70 00 00 00   ······T·  ···xp···
0040  04 0a 00 4b 01 75 71 00  7e 00 00 00 00 00 2d ac   ···K·uq·  ~·····-·
0050  ed 00 05 77 27 00 00 00  06 00 00 00 01 00 00 72   ···w'···  ·······r
0060  f0 00 00 00 00 00 15 4f  46 50 54 5f 46 45 41 54   ·······O  FPT_FEAT
0070  55 52 45 53 5f 52 45 51  55 45 53 54 77 08 00 00   URES_REQ  UESTw···
0080  00 2d 00 00 00 00                                  ·-····
```

Figure 6: Feature request phase

Router reply with FEATURE_REPLY and sending all the link states to the controller with port 50000 as can be seen in (7). User can then use terminals to exchange messages between the



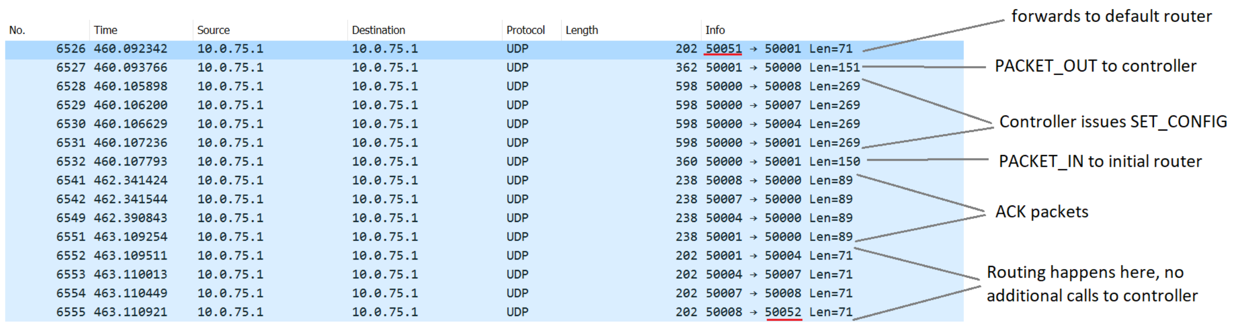
```
No.        Time         Source         Destination    Protocol  Length   Info
        36 1.823027     10.0.75.1      10.0.75.1      UDP            620 50007 → 50000 Len=280
        37 1.823046     10.0.75.1      10.0.75.1      UDP            584 50002 → 50000 Len=262
        38 1.823089     10.0.75.1      10.0.75.1      UDP            620 50008 → 50000 Len=280
        39 1.823155     10.0.75.1      10.0.75.1      UDP            656 50001 → 50000 Len=298
        40 1.823298     10.0.75.1      10.0.75.1      UDP            584 50005 → 50000 Len=262

> Frame 36: 620 bytes on wire (4960 bits), 312 bytes captured (2496 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 10.0.75.1, Dst: 10.0.75.1
> User Datagram Protocol, Src Port: 50007, Dst Port: 50000

0000  02 00 00 00 45 00 01 34  66 80 00 00 80 11 00 00   ····E··4  f·······
0010  0a 00 4b 01 0a 00 4b 01  c3 57 c3 50 01 20 25 2a   ··K···K·  ·W·P· %*
0020  ac ed 00 05 77 04 00 00  c3 50 75 72 00 02 5b 42   ····w···  ·Pur··[B
0030  ac f3 17 f8 06 08 54 e0  02 00 00 78 70 00 00 00   ······T·  ···xp···
0040  04 0a 00 4b 01 75 71 00  7e 00 00 00 00 00 df ac   ···K·uq·  ~·······
0050  ed 00 05 77 10 00 00 00  07 00 00 00 01 00 00 c3   ···w····  ········
0060  57 00 00 00 00 75 72 00  1c 5b 4c 63 69 73 74 69   W····ur·  ·[Lcisti
0070  61 6b 6a 2e 52 6f 75 74  65 72 2e 49 6e 74 65 72   akj.Rout  er.Inter
0080  66 61 63 65 3b 9b 7c ee  0d d1 7f 98 f6 02 00 00   face;·|·  ········
0090  78 70 00 00 00 04 73 72  00 19 63 69 73 74 69 61   xp····sr  ··cistia
00a0  6b 6a 2e 52 6f 75 74 65  72 2e 49 6e 74 65 72 66   kj.Route  r.Interf
00b0  61 63 65 06 5f bc 48 e7  34 ca 61 02 00 03 49 00   ace·_·H·  4·a···I·
00c0  02 69 64 49 00 06 6d 65  74 72 69 63 49 00 04 70   ·idI··me  tricI··p
00d0  6f 72 74 78 70 ff ff ff  ff 7f ff ff ff 00 00 c3   ortxp···  ········
00e0  50 73 71 00 7e 00 02 00  00 00 01 00 00 00 01 00   Psq·~···  ········
00f0  00 c3 55 73 71 00 7e 00  02 00 00 00 02 00 00 00   ··Usq·~·  ········
0100  01 00 00 c3 54 73 71 00  7e 00 02 00 00 00 03 00   ····Tsq·  ~·······
0110  00 00 01 00 00 c3 58 77  15 00 13 4f 46 50 54 5f   ······Xw  ···OFPT_
0120  46 45 41 54 55 52 45 53  5f 52 45 50 4c 59 77 08   FEATURES  _REPLYw·
0130  00 00 00 df 00 00 00 00                            ········
```

Figure 7: Feature reply phase

endpoints. For example endpoints 50051 sends a message to endpoint 50052. It first sends a packet to its default router at 50001, but router R1 does not know how to reach endpoint 2 at 50052, so it asks controller for help and then routing procedure happens as can be seen in (8).
After that if EP1 wants to send to EP2, there would be no need to contact the controller as R1 would know the route to EP2.



No.	Time	Source	Destination	Protocol	Length	Info
6526	460.092342	10.0.75.1	10.0.75.1	UDP	202	50051 → 50001 Len=71
6527	460.093766	10.0.75.1	10.0.75.1	UDP	362	50001 → 50000 Len=151
6528	460.105898	10.0.75.1	10.0.75.1	UDP	598	50000 → 50008 Len=269
6529	460.106200	10.0.75.1	10.0.75.1	UDP	598	50000 → 50007 Len=269
6530	460.106629	10.0.75.1	10.0.75.1	UDP	598	50000 → 50004 Len=269
6531	460.107236	10.0.75.1	10.0.75.1	UDP	598	50000 → 50001 Len=269
6532	460.107793	10.0.75.1	10.0.75.1	UDP	360	50000 → 50001 Len=150
6541	462.341424	10.0.75.1	10.0.75.1	UDP	238	50008 → 50000 Len=89
6542	462.341544	10.0.75.1	10.0.75.1	UDP	238	50007 → 50000 Len=89
6549	462.390843	10.0.75.1	10.0.75.1	UDP	238	50004 → 50000 Len=89
6551	463.109254	10.0.75.1	10.0.75.1	UDP	238	50001 → 50000 Len=89
6552	463.109511	10.0.75.1	10.0.75.1	UDP	202	50001 → 50004 Len=71
6553	463.110013	10.0.75.1	10.0.75.1	UDP	202	50004 → 50007 Len=71
6554	463.110449	10.0.75.1	10.0.75.1	UDP	202	50007 → 50008 Len=71
6555	463.110921	10.0.75.1	10.0.75.1	UDP	202	50008 → 50052 Len=71

forwards to default router
PACKET_OUT to controller
Controller issues SET_CONFIG
PACKET_IN to initial router
ACK packets
Routing happens here, no additional calls to controller

Figure 8: Routing of packet.

5 Summary

This report explained the design and implementation of my own version of OpenFlow protocol. It talked about the decisions made and how these decisions affected the final result. The goal of the assignment was to design and implement programs that simulate SDN and which use a home-made version of OpenFlow protocol to communicate with each other. The goal was successfully achieved as reflected in the report.

6 Reflection

Overall, I am happy with how I did in this assignment. I got a better understanding of SDN and OpenFlow as well as I got to practice working on a programming project. It took me a month to do the assignment. However, I was not working continuously on it, but rather only when I had spare time. According to github I committed almost every week and the project has over 2000 lines of code. I enjoyed doing this assignment much more than the first one as got more experienced in concurrent programming and I feel that I structured my project much better.

References

[1] Othmane Blial, Mouad Ben Mamoun, and Benaini Redouane. An overview on sdn architectures with multiple controllers. *Journal of Computer Networks and Communications*, 2016:1–8, 01 2016.

[2] Wireshark. Wireshark, 2018.