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

CSU33031 Computer Networks
Assignment 2: Flow Forwarding

Liam Junkermann, 19300141

December 4, 2022

Contents

Abstract

This report will discuss the implementation of a flow forwarding software defined wide area system (SD-WAN). Starting with a brief overview of the theory needed to build and execute such a system, the components of the final implementation, a demonstration of a network topology employing this system, a brief discussion of potential improvements, and finally a summary and reflection of the assignment.

1 Introduction

This assignment focused on the development and deployment of a flow forwarding mechanism to allow two clients (`End nodes`) on separate networks to interact as though they have a share network by employing a SD-WAN. A centralised `Controller` keeps track of the `Routers` and in turn the `End nodes` attached to them. Messages are sent through an interactive console on the `End node` and then, with help from the `Routers` and `Controller`, directed to the appropriate `End node`. This report will discuss the theory which led to the final implementation.

2 Theory

This flow forwarding implementation took inspiration from the OpenFlow [1] protocol. Understanding this protocol has enabled me to lay the foundation for my implementation.

2.1 OpenFlow

This protocol was proposed as a way of allowing researchers to better test experimental protocols. Researchers found that the barrier to entry for new networking ideas was quite difficult as it is difficult to simulate the scale of a full-sized network. Therefore most new networking ideas go untested and therefore cannot make their way into production. OpenFlow addresses this by creating a standard for switches and communications which allow for programmable flow tables allowing the testing of new networking ideas. In practice OpenFlow leverages existing flow-tables built into ethernet switches and routers, this allows new protocols to be tested on a production network, while also allowing existing traffic to continue as normal.

3 Implementation

3.1 Overview

This implementation of the network uses some basic pieces from the proposed OpenFlow method. Specifically, `Routers` connect to the `Controller` with a `HELLO` packet, expecting a response in the form of a flow table (`FWD_MOD`) update packet. This table is implemented statically for the purposes of this assignment. In the virtual network for this assignment I have implemented one `Controller` node and five `Router` nodes, these are started automatically by the `docker-compose.yml` file and run headlessly for the interactive `End Node` nodes to connect to. There are four `End Node` nodes also defined in the `docker-compose.yml` file, however these processes need to be started manually.

3.2 Packet Overview

All packets have a two byte header, where byte index 0 is the packet type, and byte index one is the destination length. There are some cases where no destination is supplied, for example in the initial `Router` setup so this destination length packet allows data to be passed efficiently. This also allows for variable length destination names which is used in this implementation to mock different potential destinations in a humanly readable format. The packet types and their respective value are listed below:

```
// Packet Types
static final byte HELLO = 0;
static final byte PACKET_IN = 1;
static final byte FWD_MOD = 2;
static final byte NETWORK_ID = 3;
```

Listing 1: Code snippet from `Node` with the encoded packet types

All packets packets transmitted through this network follow the packet structure detailed in Figure 1

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------------------------------------------------------+
|                 |                |                              |
|   PACKET TYPE   |  DEST LENGTH   |     DESTINATION/PAYLOAD      |
|                 |                |                              |
+----------------------------------+                              |
|                                                                 |
|                       DESTINATION/PAYLOAD                       |
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 1: Standard Packet

3.2.1 Operation overview

Figure 2 Shows the steps needed for the potential message to be sent from one end node to another. The network deployed for this assignment is illustrated in Figure 3. A controller is connected to each router as well.

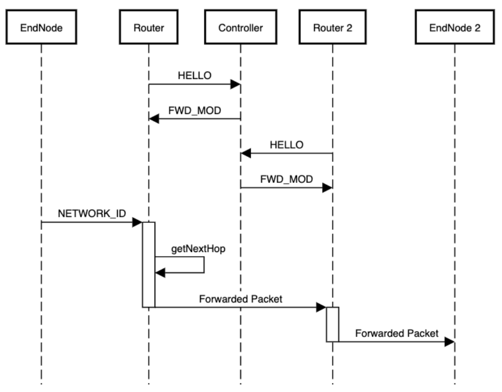


Figure 2: A sequence diagram illustrating the setup and example message sharing between two end nodes

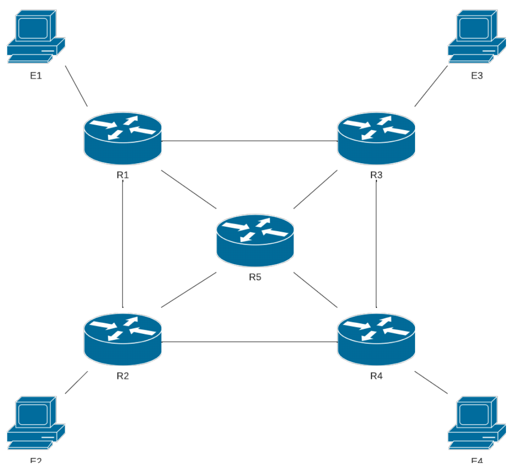


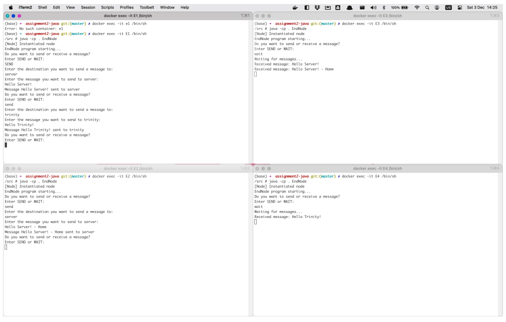
Figure 3: An example network topology illustrating the interactions between nodes.

This implementation only requires the destination to be included in the header as the flow table manages the output of each router based on the destination, source information would therefore be redundant for the purposes of forwarding messages. This decision will be discussed further in Future Enhancements. Figure 4 and Figure 5 show the client interface and packet capture of an example interaction in the virtual network.

3.3 Network Components

3.3.1 Node

In practice each of these network nodes implement a `Node` class which handles most of the boilerplate Datagram behaviour and sets constants for header length, values, and positions, as well as providing some worker functions to generate packets and packet data appropriately. This `Node` also handles the multithreading needed to successfully multiplex within the `Controller`.



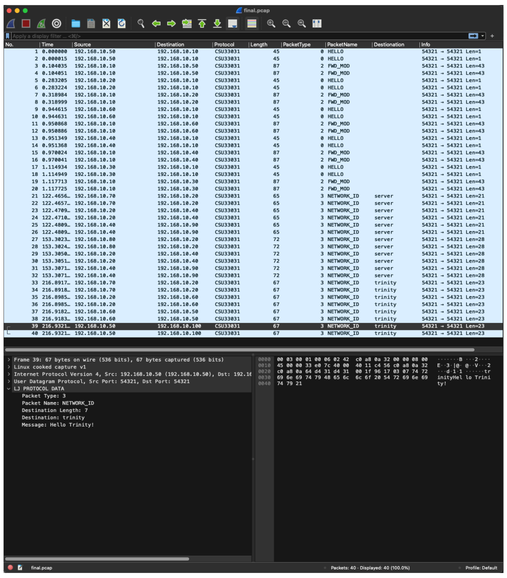
```
docker exec -it E1 /bin/sh
(base) → assignment2-java git:(master) ✗ docker exec -it e1 /bin/sh
Error: No such container: e1
(base) → assignment2-java git:(master) ✗ docker exec -it E1 /bin/sh
/src # java -cp . EndNode
[Node] Instantiated node
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
SEND
Enter the destination you want to send a message to:
server
Enter the message you want to send to server:
Hello Server!
Message Hello Server! sent to server
Do you want to send or receive a message?
Enter SEND or WAIT:
send
Enter the destination you want to send a message to:
trinity
Enter the message you want to send to trinity:
Hello Trinity!
Message Hello Trinity! sent to trinity
Do you want to send or receive a message?
Enter SEND or WAIT:
```

```
docker exec -it E3 /bin/sh
(base) → assignment2-java git:(master) ✗ docker exec -it E3 /bin/sh
/src # java -cp . EndNode
[Node] Instantiated node
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
wait
Waiting for messages...
Received message: Hello Server!
Received message: Hello Server! - Home
```

```
docker exec -it E2 /bin/sh
(base) → assignment2-java git:(master) ✗ docker exec -it E2 /bin/sh
/src # java -cp . EndNode
[Node] Instantiated node
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
send
Enter the destination you want to send a message to:
server
Enter the message you want to send to server:
Hello Server! - Home
Message Hello Server! - Home sent to server
Do you want to send or receive a message?
Enter SEND or WAIT:
```

```
docker exec -it E4 /bin/sh
(base) → assignment2-java git:(master) ✗ docker exec -it E4 /bin/sh
/src # java -cp . EndNode
[Node] Instantiated node
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
wait
Waiting for messages...
Received message: Hello Trinity!
```

Figure 4: Four terminals showing an example of the network usage with four mocked clients



```
No. Time      Source         Destination     Protocol Length PacketType PacketName Destination Info
  1 0.000000  192.168.10.50  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
  2 0.000015  192.168.10.50  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
  3 0.104035  192.168.10.10  192.168.10.50   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
  4 0.104051  192.168.10.10  192.168.10.50   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
  5 0.283205  192.168.10.20  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
  6 0.283224  192.168.10.20  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
  7 0.318984  192.168.10.10  192.168.10.20   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
  8 0.318999  192.168.10.10  192.168.10.20   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
  9 0.944615  192.168.10.60  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 10 0.944631  192.168.10.60  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 11 0.950868  192.168.10.10  192.168.10.60   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 12 0.950886  192.168.10.10  192.168.10.60   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 13 0.951349  192.168.10.40  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 14 0.951368  192.168.10.40  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 15 0.970024  192.168.10.10  192.168.10.40   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 16 0.970041  192.168.10.10  192.168.10.40   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 17 1.114934  192.168.10.30  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 18 1.114949  192.168.10.30  192.168.10.10   CSU33031 45     0 HELLO                  54321 → 54321 Len=1
 19 1.117713  192.168.10.10  192.168.10.30   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 20 1.117725  192.168.10.10  192.168.10.30   CSU33031 87     2 FWD_MOD                54321 → 54321 Len=43
 21 122.4656… 192.168.10.70  192.168.10.20   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 22 122.4657… 192.168.10.70  192.168.10.20   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 23 122.4709… 192.168.10.20  192.168.10.40   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 24 122.4710… 192.168.10.20  192.168.10.40   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 25 122.4809… 192.168.10.40  192.168.10.90   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 26 122.4809… 192.168.10.40  192.168.10.90   CSU33031 65     3 NETWORK_ID server      54321 → 54321 Len=21
 27 153.3023… 192.168.10.80  192.168.10.20   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 28 153.3024… 192.168.10.80  192.168.10.20   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 29 153.3050… 192.168.10.20  192.168.10.40   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 30 153.3051… 192.168.10.20  192.168.10.40   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 31 153.3071… 192.168.10.40  192.168.10.90   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 32 153.3071… 192.168.10.40  192.168.10.90   CSU33031 72     3 NETWORK_ID server      54321 → 54321 Len=28
 33 216.8917… 192.168.10.70  192.168.10.20   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 34 216.8918… 192.168.10.70  192.168.10.20   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 35 216.8985… 192.168.10.20  192.168.10.60   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 36 216.8985… 192.168.10.20  192.168.10.60   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 37 216.9182… 192.168.10.60  192.168.10.50   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 38 216.9183… 192.168.10.60  192.168.10.50   CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 39 216.9321… 192.168.10.50  192.168.10.100  CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23
 40 216.9321… 192.168.10.50  192.168.10.100  CSU33031 67     3 NETWORK_ID trinity     54321 → 54321 Len=23

> Frame 39: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.10.50 (192.168.10.50), Dst: 192.16
> User Datagram Protocol, Src Port: 54321, Dst Port: 54321
v LJ PROTOCOL DATA
    Packet Type: 3
    Packet Name: NETWORK_ID
    Destination Length: 7
    Destination: trinity
    Message: Hello Trinity!

0000  00 03 00 01 00 06 02 42  c0 a8 0a 32 00 00 08 00   ·······B ···2····
0010  45 00 00 33 e0 7c 40 00  40 11 c4 56 c0 a8 0a 32   E··3·|@· @··V···2
0020  c0 a8 0a 64 d4 31 d4 31  00 1f 96 17 03 07 74 72   ···d·1·1 ······tr
0030  69 6e 69 74 79 48 65 6c  6c 6f 20 54 72 69 6e 69   inityHel lo Trini
0040  74 79 21                                           ty!
```

Figure 5: The packet capture of the example network usage

3.3.2 Controller

The `Controller` manages the forwarding table and helps routers resolve where packets should go when an entry may not be in the forwarding table. The `Controller` is initialised first in order to register `Router` nodes when receiving `HELLO` packets. The `Controller` also accepts `PACKET_IN` packets which are also sent from the `Router`. This will be discussed further in the Router section. The forwarding table ahs been implemented statically as two dimensional array in the current implementation of the controller. This table outlines how each `Router` should direct a packet based on the destination value. The `Controller` is a headless node and is the first node started in the Docker Compose environment.

3.3.3 Router

The `Router` node does most of the heavy lifting in this network, handling inputs from both `End Node` and other `Router` nodes and directing it appropriately. The `Router` handles two packet types:

NETWORK ID The `NETWORK_ID` packet is sent from the `End Node` nodes with messages to forward to other nodes. The `Router` takes the data from the incoming packet and repackages it in a new packet with the next hop destination as described in the forwarding table which is received after the `Router` node starts up and makes contact with the `Controller`

FWD MOD The `FWD_MOD` packet is sent from the `Controller` and updates the forwarding table of the `Router` node. This `Router` can also receive this packet after forwarding a packet with an unresolved destination to the controller. This will be discussed further in Future Enhancements.

3.3.4 End Node

The `End Node` handles the user input as well as displaying packets sent. As such, the `End Node` is fairly simple as it only expects to receive `PACKET_IN` packets with messages. In this implementation, because all input is taken from a terminal based scanner, a client must opt to `WAIT` for messages, a feature which will be discussed further in Future Enhancements. Ordinarily, the user will enter whether they would like to send a message or wait for messages, if they choose to send messages they will be prompted to enter the destination, and the message. This interaction is exhibited in Figure 4.

4 Future Enhancements

This project is a simplified version of what a production ready release would look like. Therefore, some decisions were made to make the work more manageable. Firstly, the static forwarding table. Because I knew what the network topology would look like for this implementation I was able to create a static forwarding table. The first enhancement was I to continue this project would be to enable `Router` nodes to register with one another, and the `Controller` to allow new nodes to be added to the WAN. `End node` nodes would connect to the router and be added to the forwarding table dynamically allowing for a more flexible system. Next, the user experience would need to improve allowing clients to both view messages received and sent. This is more easily done when an application can handle both user input and network traffic separately as opposed to within the same terminal. Finally I would add a source header field to allow easier packet acknowledgement and response, but this would be built based on the needs of a given application.

5 Summary

In this report discussed the requirements for the assignment, the real-world version of a solution from which I drew inspiration, and finally my implementation of a flow forwarding mechanism. Then, an exploration of the basic functionality and an example topology of the completed network, as well as some limitations and potential improvements to this implementation.

6 Reflection

Like the first assignment, this project had quite a large scope. Unfortunately during the period this assignment ran I fell somewhat behind in college work as a result of the sheer volume so I found if difficult to dedicate the time needed to execute this project with the same persistence and quality as the first assignment, this is reflected in the quality of this report. The expectations for each part of the assignment were clearer in this assignment though, which allowed me to be more organised in the final execution, and I should have used those opportunities more effectively.

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008.