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

CSU33031 Computer Networks
Assignment 2 - Flow Forwarding

Liam Junkermann - 19300141

December 3, 2021

Introduction

The goal of this assignment was to explore the use of flow control in networks, and the decisions made by routers and services to forward packet flow to the necessary clients and applications. This report will outline the design, implementation, and learnings associated with the development of this Flow Forwarding assignment. In outlining the design and implementation, this report will also analyse the strengths and weaknesses of the developed system.

Contents

1 Design and Implementation

The design of this system is quite simple. There is a centralised `Controller` which manages connections to, and from routers. Multiple `Router`s are set up to handle traffic from various endpoints (`Application`). The `Controller` relies on the use of a forwarding table to direct traffic appropriately. Forwarding tables store the possible endpoints (transmitting and receiving), and the correct route for the querying `Router` to correctly route traffic. This use of a centralised Forwarding Table in the `Controller` also allows for the `Router` to use fewer resources storing route information for routes it will never use.

1.1 The Packet Structure

Much like the previous assignment, all the packets in this system are the same size, with Type-Length-Value (TLV) headers. This allows `Router`s (and the `Controller`), to determine where the packet is originating from, and in turn where the value should be sent.

1.2 Network Components

This system is built using four basic components – `Application`, `Controller`, `ForwardingService`, and `Router` – which are deployed in different numbers and configurations across the network topology. These four components work together to demonstrate Flow Forwarding.

`Application` This is the Client of the network - an interactive user terminal to send and recieve messages from the other client (or other parts of the network). The `Application` asks the user where they would like to send their message, then what content they would like to send. This information is then compiled into a packet sent to the `ForwardingService`. The `Application` then waits for a response packet before determining what it will do next. The `Application` then repeats this process of waiting for input or a packet indefinitely.

`Controller` This is effectively the master router of the network. When the `Router` has not saved where to send an unknown packet, it will check with the `Controller` where to send the packet. The controller initialises the Forwarding Table and address table on load. Then awaits contact from a `Router` before then responding with the needed route – repeating this process until shutdown.

`ForwardingService` is the bridge between the `Applications`s and the `Router` network. It routes traffic from the querying `Application` to the correct `Router` where the `Router` logic takes over to route the packet the rest of the way. In a larger system the `ForwardingService` may also be used to format or augment data in order to send it into a larger array of routers or another third party system.

`Router` is the work horse of the whole system. An array of `Router`s is deployed in order to manage traffic from various `Application` instances. The `Router` leverages the `Controller` to inform where it sends packages, and the distributed nature of the `Router`s in the network topology allows for resource efficient routing and flow forwarding.
Packets are sent to the `Router` and if it does not recognise the packet it will consult the `Controller`. This then prompts an update of the local flow table for future use. The `Router` then sends the necessary packets and awaits the next incoming packet.

2 Deployment Approach and Topology

The deployment of this network is managed with Docker's `Docker Compose`. This allows for one command to create the demonstration network of components rather than many (or chained) commands and allows for quicker tweaks to the entire system. Docker compose relies on a `yml` file to properly set up the network:

```
version: "3"

services:
  Controller:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    command: java -cp /code Controller
    networks:
      - flowctrlNet
  ForwardingService:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    command: java -cp /code ForwardingService
    networks:
      - flowctrlNet

  RouterOne:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    networks:
      - flowctrlNet
    command: java -cp /code Router 1
  RouterTwo:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    networks:
      - flowctrlNet
    command: java -cp /code Router 2
  RouterThree:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    networks:
      - flowctrlNet
    command: java -cp /code Router 3
  RouterFour:
    image: flowctrltest
    volumes:
      - $pwd/src:/code
    networks:
      - flowctrlNet
    command: java -cp /code Router 4
  RouterFive:
    image: flowctrltest
    volumes:
```

```
50        - $pwd/src:/code
51      networks:
52        - flowctrlNet
53      command: java -cp /code Router 5
54    RouterSix:
55      image: flowctrltest
56      volumes:
57        - $pwd/src:/code
58      networks:
59        - flowctrlNet
60      command: java -cp /code Router 6
61    EndpointOne:
62      image: flowctrltest
63      volumes:
64        - $pwd/src:/code
65      command: java -cp /code Application EndpointOne
66      networks:
67        - flowctrlNet
68    EndpointTwo:
69      image: flowctrltest
70      volumes:
71        - $pwd/src:/code
72      command: java -cp /code Application EndpointTwo
73      networks:
74        - flowctrlNet

76    wireshark:
77      image: wiresharkcontainer
78      command: wireshark --display host.docker.internal:0
79      networks:
80        - flowctrlNet
81  networks:
82    flowctrlNet:
83      driver: bridge
```

This docker-compose file setups all the relevant services, provided that the host machine has properly built the code and is running the suitable x11 host. The `docker-compose` file also relies on a custom docker image "flowctrltest" defined below:

```
1 FROM adoptopenjdk/openjdk14:latest
2 ENV DISPLAY=host.docker.internal:0
3 RUN apt-get update -y && apt-get install libxrender1 libxtst6 libxi6 -y
4 CMD [ "/bin/bash" ]
```

This creates a docker container with all the necessary dependencies to run the services required, as well as the host command which allows the terminal to run on the host machine.

3 Summary and Reflection

3.1 The systems strengths and weaknesses

The general structure of this solution results in an effecient and scalable solution to the problem. In practice, though, the current approach to generating services and the setup required is a bit cumbersome. Each node needs to be manually created and added to the static `forwardingTable` and `addressTable` respectively. In a more developed solution the code could be changed to utilise the `Controller` to manage and spin up `Router`s as needed, and have the endpoint `Application`s responsible for connecting to the correct `Router`, who would then manage the `Controller` table updating through some kind of initialisation process.

3.2 Learnings from Assignment 2

The biggest extra learning gained from this assignment was learning to utilising `docker-compose` to more efficiently generate and test services. Using this solution allowed me to see the logs and manage of all the services from one terminal in contrast to the ever-growing open terminals I had before. Learning to use `docker-compose` was crucial in being able to more efficiently test and ensure the solution worked. Again, I ran into troubles with getting wireshark working due to the macOS "security containers" and this is something perhaps using another machine might resolve which is something I will explore in the future.