CSU33031
Assignment 2 Report

Lydia MacBride

19333944

November 2021

Contents

1 Introduction

This document outlines my approach and implementation for Assignment 2 for CSU33031 Computer Networks. The aim of this design is to create a method to allow for communication across a network where a central **Controller Device** handles the distribution of packet routing details to a collection of **Endpoints** and **Routers** spread across multiple networks, upon request from each device. Whilst not required in the initial assignment specs, I have implemented that my **Controller Device** generates the master routing table procedurally as devices connect to it and routing information requests are received.

My implementation is built in **Python 3.9** and particularly the `socket` library. All code is presently designed specifically to run in **Docker** containers and is reliant on its internal DNS resolution capabilities.

For demonstration purposes, I have implemented a very basic chat application running on each **Endpoint** which allows users on each endpoint to send simple text messages to each other.

All code is available at my GitHub repo for this project at: `https://github.com/LydiaUwU/CSU33031-Assignment-2`

1.1 Software

- **Python 3.9**
- **Docker**
- **PyCharm**
- **Wireshark**
- **GitHub**
- **EndeavourOS**

1.2 Libraries

- `socket`
- `threading`

2 Design & Implementation

2.1 Routing Tables

Key to my protocol's design is the method for routing table generation. Routing tables are generated in two cases: the first when a device detects neighbours on the same network as it, and secondarily by the **Controller** upon request for routing information.

The former form of routing table generation occurs during the initialisation of **Endpoint** and **Router** devices. During which, a device scans any network it is connected to for hostnames following a specified pattern and upon successfully discovering a device it adds this device to its routing table. It then sends a `who` type packet to any found devices informing them of its hostname and requesting any usernames associated with these devices if they are **Endpoints**. Lastly, the searching device informs the **Controller** of any new connections

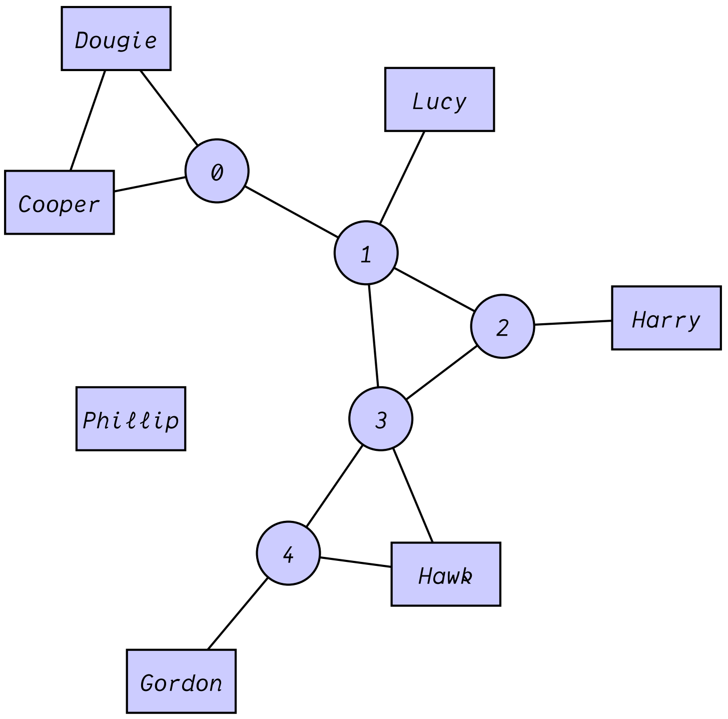
The later form of routing table generation is handled by the **Controller** upon request from any device on the network. When a device wishes to send a packet to a destination that is not within its own routing table, it sends a routing information request to the **Controller**, the controller then checks its master routing table for the requested route and then returns this to the requesting device if it is found. Otherwise, the **Controller** generates a new routing table entry using its `find_route()` method. The results of this is then stored in the master routing table, and the immediate node after the requesting node in the route is returned to the requesting route.

2.1.1 Routing Table Format

Routing tables are stored on each **Endpoint** and **Router** device in such a way that each device only knows the immediately following device in the route to the destination rather than knowing the whole route. Routing tables are stored using **Python's** `dictionary` type. Below is an example of a devices completed routing table for the given network.

For the **Router** `1`:

```python
connections = {
    "Cooper": "0",
    "Dougie": "0",
    "Lucy": "Lucy",
    "Harry": "2",
    "Hawk": "3",
    "Gordon": "3",
    "Phillip": None
}
```



2.2 Components

2.2.1 Controller

The **Controller** plays a core role in the function of this implementation: connected to all devices to the network, it generates routing tables for individual devices upon request. This is accomplished using the **Controller's** internal map of the network, where a simple `Node` object is created to represent each device on the network, and these nodes are then connected to each other based on shared subnets and information shared by a `Node`'s neighbours. Once a device on the network wishes to send a packet to a device that is not listed on its routing tables, it contacts the controller to request a route, which the controller generates using its `find_route()` method, before returning the newly discovered route to the requesting device.

Unfortunately, as the **Controller** is centralised, if it is to crash or otherwise become non-functional, the network as a whole will be unable to generate new routing information.

2.2.2 Endpoint

Endpoints are devices that users interact with, these feature two layers: An **Application** layer that handles input from the user and communicates solely with the **Service** layer that communicates with the external network.

Application

The **Application** layer presently features a simple command line interface (CLI) with a handful of commands the user can run. Most notably is the `send` command which allows the user to send messages to other **Endpoints** by providing the username running on any specified **Endpoint**. Users can then list their received messages with the `ls` command.

When a user runs the `send` command, the resulting packet is sent over `localhost` to the **Service** layer. The **Application** layer does not communicate directly with the external network.

Service

The **Service** layer handles the reception and sending of packets from both the external network and the **Application** layer. The **Service** layer functions as a bridge between the external and internal networks.

The **Service** layer is what is launched upon the start of an **Endpoint** device, and is responsible for then launching the **Application** layer.

When sending packets to the external network the **Service** layer checks for the recipient in its own routing tables, and contacts the **Controller** for routing information in the event that the recipient is not listed.

2.2.3 Router

Routers function very similarly to the **Endpoint's Service** layer, however they do not handle any form of `localhost` communication and are instead solely focused on forwarding packets from one place to another across the network. Similarly to the **Endpoint's Service** layer, **Routers** search for a packets recipient in their routing tables and contact the **Controller** in the event that it is not found.

3 Packet Encoding

All packets are sent using TLV encoding as required by the assignment specs. Packet encoding results in a binary array where the first byte dictates the packet type, the second two bytes the packet size, and any remaining bytes contain the packet data. Two packets can be combined and sent at once using the `combination` packet type, and these can be used recursively such that a `combination` packet can contain additional packets, allowing for packets with indefinite amounts of sub packets.

Decoded packet information is stored in a **Python** `dictionary`, as such, packets are limited to only one of each packet type per packet. However, this allows for very easy processing of every received packet.

3.1 Packet Types

No.	Name	Type	Description
0	`who`	String	”Who-is” packet, used to request the hostname of another device.
1	`new`	String	Sent to controller during device initialisation. Contains the device type.
2	`exit`	None	Sent from the application to the service in order to shutdown an endpoint device.
3	`name`	String	Contains a device’s hostname.
4	`recipient`	String	Contains the destination device’s hostname.
5	`string`	String	Contains a string.
6	`route`	String	Contains the next device’s hostname in the route from the requester to the destination.
7	`combination`	Packet	Contains multiple other packets.
8	`unknown`	String	Any unknown packet is encoded as a string.

3.2 Example Packet

Given two users *alice* and *bob*, lets look at message packet sent from *alice* to *bob* containing the text ”Hello World!”. Given that this packet will require three different parameters: one for the recipient’s name, one for the sender’s name and the last containing the message text, we utilise the `combination` packet type. First off, two packets are created `name:alice` and `string:Hello World!`, these are then wrapped in a `combination` packet. Then another packet is created containing `recipient:bob` which is then wrapped in a second `combination` packet alongside our first, giving us our final packet which will be sent to the external network.

Data												
0x070020040003626f62070017030005616c69636505000c48656c6c6f20576f726c6421												
Type	**Length**	**Value**										
combination	32	0x040003626f62070017030005616c69636505000c48656c6c6f20576f726c6421										
		Type	**Length**	**Value**	**Type**	**Length**	**Value**					
		recipient	3	bob	combination	23	0x030005616c69636505000c48656c6c6f20576f726c6421					
							Type	**Length**	**Value**	**Type**	**Length**	**Value**
							name	5	alice	string	12	Hello World!
0x07	0x0020	0x04	0x0003	0x626f62	0x07	0x0017	0x03	0x0005	0x616c696365	0x05	0x000c	0x48656c6c6f20576f726c6421

4 Reflections & Potential Improvements

Overall, I am fairly satisfied by my design and implementation for this assignment, especially given the present circumstances. As much as I would wish to have time to continue polishing my work, I am very happy with the project's current state. Particularly, I feel that the quality of my code has increased significantly between this project and previous work, especially when it comes to ease of use, readability and reliability. I am very proud that through this project I have pushed myself to explore new features of both **Python** and **Docker**. However, as mentioned, given more time I would have liked to implement the following features.

Firstly, I decided for the purposes of submission that the implementation of package acknowledgement was unnecessary. As this was solely intended to run on **Docker**, I had no concerns of signal degradation or loss, however if this present implementation were to be deployed across a hardware network there are no safeguards against the loss of packets.

Secondly, I was originally planning for the **Application** to store all received messages in a file and allow the user to list them with an additional argument to the `ls` command. This would allow for persistent message logs that would survive reboots and power cycles of the **Endpoint**, which I feel is a pretty core feature to any *real* messaging application.

Thirdly, whilst the routing algorithm used by the **Controller** functions, and correctly generates routes, it is very inefficient. Rather than generating the shortest route between two devices, it generates the last found route. I would like to change this algorithm such that it returns *all* discovered routes between devices, and these could then be used in conjunction with additional data (i.e. ping time for a certain route) by devices on the network to more intelligently choose what route to send a packet along.

Lastly, I believe that with some relatively minor modifications to my implementation, the role of the **Controller** could be made obsolete. This would allow the network to become a lot more robust as the dependency for a central controller would be removed, as currently, if the controller crashes or is otherwise shutdown, the whole network will be unable to generate new routing data. Additionally, removing the controller would lead to an increase in practicality for this protocol, as it would be fairly atypical for a physical network to be structured in such a way that a central controller could be connected to *all* devices.