

CSU33031 Computer Networks: Assignment 2 Report

Cian Mawhinney, 19335893

Contents

1 Introduction	1
2 Protocol Design	2
2.1 Applications	2
2.2 Forwarders	2
2.3 Controller	2
2.4 Communication	3
2.5 Packet Description	4
2.5.1 Message Type	4
2.5.2 Field Count	5
2.5.3 Fields	6
3 Implementation	6
3.1 Packet Encoding & Decoding	6
3.2 Network components	6
3.2.1 Demo Application	7
3.2.2 Forwarder	8
3.2.3 Controller	8
3.3 Network Topology	9
4 Discussion	11
5 Reflection	12

1 Introduction

The goal of this assignment is to gain an understanding into the design, development and implementation of a routing protocol. The guidance set out in the assignment description specifies that the protocol designed should identify source and destination nodes by a collection of strings, and to reduce the processing requirements on the forwarders in the network, a central controller should determine the routing tables for each forwarder.

As with any protocol, a balance must be struck between the overhead of the header and the functionality enabled by that overhead. During this report, the details of the protocol along with the design decisions made will be documented and explained.

2 Protocol Design

This section of the report documents the design phase of the protocol and how the various components on the network should interact with each other. Subsections [2.1](#), [2.2](#), [2.3](#) explain the role of each type of distinct network component, their role within the network, and their design goals. Subsections [2.4](#), [2.5](#) focuses on how the protocol should enable communication between each of the network components, and how the packets themselves should be constructed.

Throughout this assignment, the approach to the design of the protocol was to place an emphasis on simplicity, ensuring extraneous features were not added into the design, yet still to provide clear paths for extensibility.

2.1 Applications

In this report, the term ‘application’ refers to a program that wishes to communicate with another application using the network.

Each application is able to pick their own string encoded address that should uniquely identify that particular application to the rest of the network. Applications have a direct connection to a forwarding service, whether that is a separate machine or a service implemented locally on the same machine the application is running on. This means that the task of routing is offloaded to the forwarder as early as possible, reducing the complexity of the application.

2.2 Forwarders

The term ‘forwarder’ is used in this report to refer to a component on the network that is capable of inspecting the packet header and determining where that packet should be sent to next. As part of a network, forwarders may be connected to applications, the controller, and other forwarders.

The main duty of a forwarder is to forward packets. This process involves inspecting the packet to determine the destination, consulting the routing table for the next hop and sending the packet there. Should the destination not be present in its routing table, the packet should be dropped, and no further action should be taken. A key detail of the forwarder is that it receives its routing table from the controller, and is updated every time there is a change on the network.

2.3 Controller

The term ‘controller’ refers to a service that determines the routing tables for the forwarders on the network. Key features of the controller are that it should be connected to every forwarder by a dedicated management network and that it should have a complete view of the network so that it can calculate the best possible paths for packets to take.

Since the controller is centralised, making changes to the way traffic flows through the network is easier than a system in which forwarders are responsible for their routing table since there is just a single service to update.

2.4 Communication

This section sets out the communication mechanisms which are necessary for the network to function correctly, as set out in sections [2.1](#), [2.2](#) and [2.3](#). Most of these mechanisms stem from the need for the controller to have a complete view of the network so that it can determine the routing tables for each of the forwarders.

The first requirement is that the controller needs to know when forwarders leave and join the network. This is so that other forwarders do not try to pass traffic to a node which is not actually present in the network and also so that a connection is established that the controller can use to send it updated routing tables in the future. Therefore when a forwarder starts up, it should send a 'forwarder registration' packet to the controller, and before it leaves the network should send a 'forwarder deregistration' packet.

For similar reasons to the forwarders, applications also need to be known to the controller. So when an application starts, it should send an 'application registration' packet to the forwarder it is directly connected to, which will then forward that packet to the controller. It should then also send an 'application deregistration' packet whenever it wants to stop receiving packets.

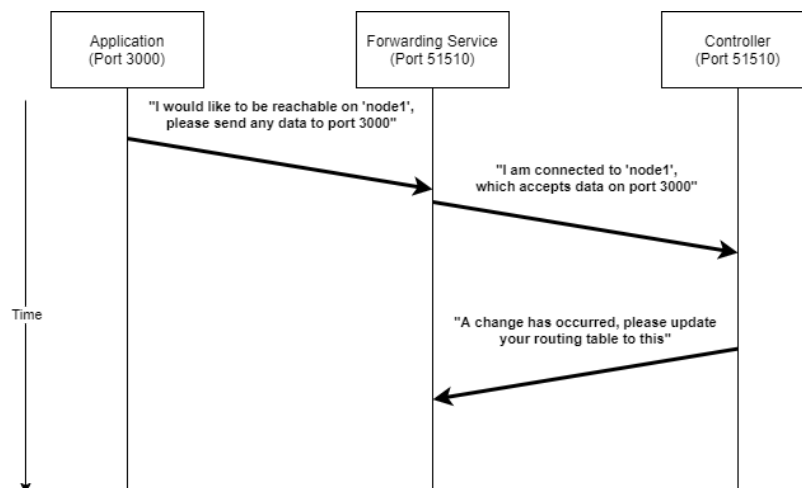


Figure 1: A diagram showing the 'application registration' communication mechanism. The application first sends its registration data to the forwarder which is local to it, which then passes it on to the controller. Since the controller has been informed of a change to the network, the forwarder is sent a new routing table.

Another essential communication mechanism is how forwarders receive their routing tables. Every time there is a change on the network, the controller should send updated tables reflecting those changes, and to do that a 'route change' packet should be sent with the new table in the payload of the packet.

Lastly, the most important requirement is that packets should be able to be passed between forwarders. To achieve this, a 'forwarded packet' packet type should be used.

2.5 Packet Description

One of the requirements of the assignment was that fields should use a TLV (Type-Length-Value) encoding. Within the header there are two sections, one containing fixed size fields, and the other containing the TLV fields, the number and count of which can vary depending on the message type.

As a general overview to the packet structure, the first byte of the packet is the message type, the second is the number of fields encoded in TLV format, the third section is the list of TLV fields, and lastly, the payload is appended to the end of the header to form the packet.

Order in packet	1	2	3	4
Length	1 byte	1 byte	Variable	Variable
Name	Message type	Fields Count	TLV fields	Payload

As an example demonstrating this structure, Figure 2 below shows how a packet for forwarding is constructed.

Category	Header								Body
	Fixed length fields		TLV Encoded Fields						
Description	Message type	Fields Count	Source Field			Destination Field			Payload
			Field Type - Source	Field Value Length	Source Address	Field Type - Destination	Field Value Length	Destination Address	
Binary Data	00	02	01	05	61 70 70 5f 31	02	05	61 70 70 5f 32	00 00 5f 12

Figure 2: A diagram dissecting an example packet being sent from 'app_1' to 'app_2' with an integer payload of 24338. Since this is a packet intended to be forwarded, it has a message type of 0 and two fields specifying the source and destination addresses.

2.5.1 Message Type

The message type is a byte sized value, and is the first field in the header. As only 6 of the 256 possible types have been allocated, there is plenty of room for expansion of the protocol to account for changing use cases. A table detailing all of the assigned message types can be found below.

Value	Name	Notes	Required Fields (2.5.3)
0	Forwarded packet	Packets with this type are used for application-to-application communication. When a forwarder receives a packet of this type, it will be sent on to the next hop, according to its routing table.	Source Address, Destination Address
1	Application Registration	Sent when an application starts up.	Source Address, Application Port
2	Application Deregistration	Sent when an application should no longer receive packets	Source Address
3	Forwarder Registration	Sent when a forwarder joins the network	Source Address
4	Forwarder Deregistration	Sent when a forwarder leaves the network	Source Address
5	Route Change	Sent by the controller when there is a change on the network. The new routing table will be present in the body of the packet	Destination Address
6-256	Unassigned	N/A	N/A

2.5.2 Field Count

The field count is a byte sized value, and is the second field in the header. It's important to note that the value counts the number of fields in the header, and not the number of bytes they collectively occupy. When parsing the header, the parser should continue to interpret sequences of bytes as TLV fields until the counter matches the number of TLV fields discovered.

2.5.3 Fields

Fields within the header are encoded in TLV (Type-Length-Value) format. This allows for fields to be placed in any order within the TLV section and be of variable length.

Type Identifier	Name	Data Type	Description
1	Source	String	The name of the source node. Necessary for applications receiving a packet to be able to reply.
2	Destination	String	The name of the destination application. Necessary for forwarders to be able to route packets to their intended destination.
3	Application Port	16-bit integer	The port the application listens on for data. Necessary so the forwarder local to the application can deliver packets to where it is listening for them.
4 - 256	Unassigned	N/A	N/A

3 Implementation

The process of implementing the protocol was relatively simple due to the design not including extraneous functionality. For convenience, UDP (User Datagram Protocol) sockets were used for communication instead of dealing with raw frames and hardware addresses. This allowed for the implementation to be done in a high level language like JavaScript, which does not have native support for interacting with raw sockets.

Docker compose was used to orchestrate the setup, giving the advantage of being able to automate the startup of every container and network connection. Each network component was built into its own container image, and then multiple copies of the images could be started according to the parameters specified in the docker compose file.

3.1 Packet Encoding & Decoding

To be able to send and receive packets consistently across multiple types of network component, a common library was written to encode and decode packets conforming to the protocol specification. This itself made heavy use of the [binary-parser-encoder](#) library, a fork of [binary-parser](#), which was a great help to simplify the codebase.

3.2 Network components

The setup for the assignment required 3 types of network components, the applications, forwarders and the controller. All network components were implemented using JavaScript

and Node.js making heavy use of the event-based programming paradigm to respond to incoming packets. The dgram library was used to interact with UDP sockets.

3.2.1 Demo Application

The program written to demonstrate that communication was taking place between two applications is unsophisticated, as the focus of the assignment was on packet forwarding. Therefore, all the program did was log any data it received from forwarded packets to the screen and send random integers that it generated.

```
registerApplication(hostname)

network.on('forwardedPacket', packet => {
  print(packet)
})

while(true) {
  sendRandomInteger(destination)
  sleep(5s)
}
```

Figure 3: Pseudocode of the demo application, showing the registration step, an event handler for when a packet is received, and a loop which sends a random integer over the network every 5 seconds.

Despite being very barebones, this approach is enough to demonstrate that every packet is indeed being sent through the network of forwarders.

assignment2-application_2-1		Sending <Buffer 00 01 63 0f>
assignment2-application_1-1		Received <Buffer 00 01 63 0f>
assignment2-application_1-1		Sending <Buffer 00 00 dc bf>
assignment2-application_2-1		Received <Buffer 00 00 dc bf>
assignment2-application_2-1		Sending <Buffer 00 00 05 45>
assignment2-application_1-1		Received <Buffer 00 00 05 45>
assignment2-application_1-1		Sending <Buffer 00 01 2c fd>
assignment2-application_2-1		Received <Buffer 00 01 2c fd>
assignment2-application_2-1		Sending <Buffer 00 00 33 c2>
assignment2-application_1-1		Received <Buffer 00 00 33 c2>
assignment2-application_1-1		Sending <Buffer 00 01 6c c8>
assignment2-application_2-1		Received <Buffer 00 01 6c c8>

Figure 4: Screenshot of output from the demo applications, indicating that traffic is being sent and received between two separate applications by transmitting packets over the network

3.2.2 Forwarder

The forwarders within the network all listen on UDP port 51510, a port that is well known in advance to the rest of the network. The main piece of logic within the forwarder is how it determines where to forward packets to. After receiving its routing table from the controller, the forwarder will look up the destination in its routing table for the next hop, and then the packet there. Also included in the functionality of the forwarder is that it passes control packets from any locally attached applications to the controller.

```
routingTable = []

registerForwarder(hostname)

network.on('forwardedPacket', packet => {
  nextHop = routingTable[packet.destination]
  if (nextHop) {
    sendPacket(packet, nextHop)
  } else {
    // do nothing
  }
})

network.on('applicationRegistration', packet => {
  sendPacket(packet, controller)
})

network.on('applicationDeregistration', packet => {
  sendPacket(packet, controller)
})

network.on('routeChange', packet => {
  routingTable = packet.payload
})
```

Figure 5: Pseudocode for a forwarder showing the registration that happens at the beginning and the events it must be able to deal with in response to a packet being received

3.2.3 Controller

Similarly to the forwarders, the controller listens on UDP port 51510. So that it keeps its view of the network in sync with reality, it must listen for any types of packets that signal that a change in the network has occurred. Those events are 'forwarder registration', 'application registration', 'forwarder deregistration', and 'application deregistration'.


```

globalNetworkView = []

network.on('forwarderRegistration', packet => {
  addForwarderToNetwork(packet)
  publishNewRouteTables(globalNetworkView)
})

network.on('applicationRegistration', packet => {
  addApplicationToNetwork(packet)
  publishNewRouteTables(globalNetworkView)
})

network.on('forwarderDeregistration', packet => {
  removeForwarderFromNetwork(packet)
  publishNewRouteTables(globalNetworkView)
})

network.on('applicationDeregistration', packet => {
  removeApplicationFromNetwork(packet)
  publishNewRouteTables(globalNetworkView)
})

```

Figure 6: Pseudocode for the controller, showing a high level version of the program implementation.

The controller ended up being the most complex component to implement due to the calculation of routing tables. Every time a change on the network is reported to the controller, every forwarder's routing table is recalculated and preemptively pushed out to them. To do this the controller uses Dijkstra's algorithm to calculate the shortest path between every forwarder and application combination, allowing the most efficient next hop to be used by each of the forwarders. Once the routing table is calculated, it is encoded into JSON and placed in the payload of a 'route change' packet.

For this implementation, the connections between the forwarders were hard coded into the controller, though since the IP addresses assigned to containers by docker are dynamically allocated, the full network view could not be hard coded, and therefore still required applications and forwarders to register with the controller.

3.3 Network Topology

During the assignment, the topology for the network evolved from being relatively simple, with 3 forwarders using hard coded routing tables, shown in Figure 7, to a more complex topology, making use of 9 forwarders which each received their routing table from the central controller, shown in Figure 8. At this stage, a separate management network was introduced to facilitate communication between the controller and the forwarders.

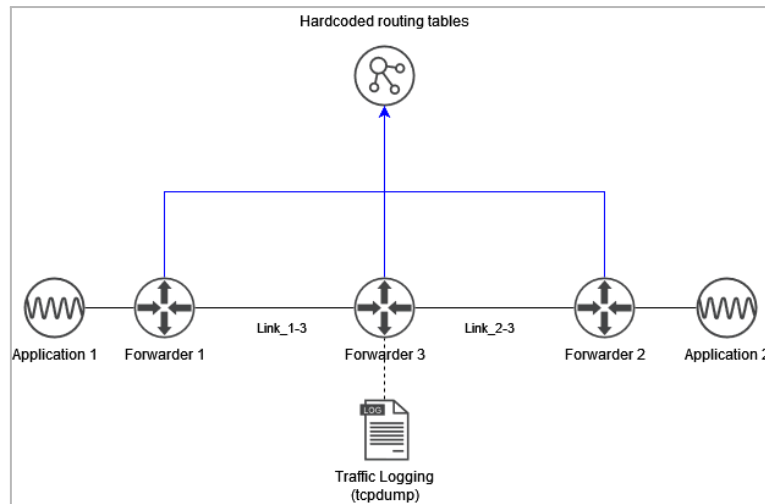


Figure 7: A diagram showing forwarders connected in a basic configuration. 3 forwarders are responsible for forwarding packets between the 2 applications. At this stage, routing tables were hardcoded, and traffic was captured at the middle forwarder.

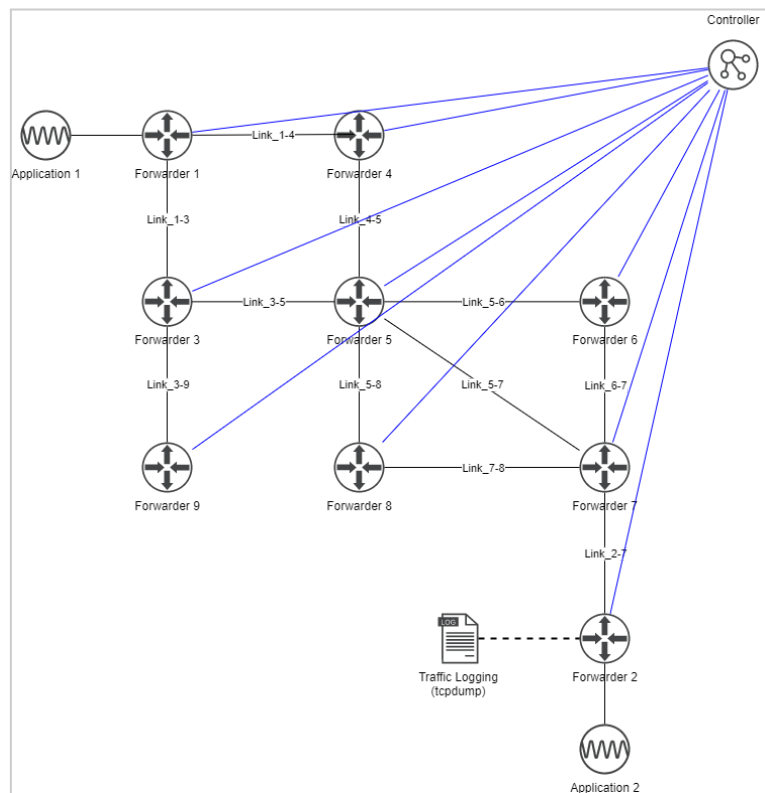
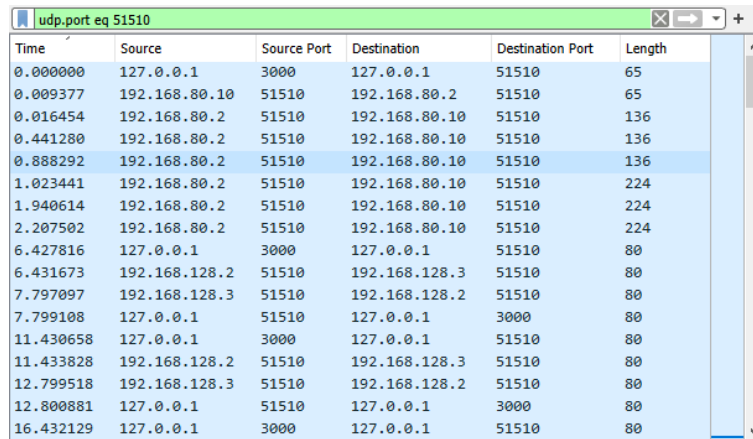


Figure 8: A diagram showing forwarders interconnected with a topology of increased complexity. Shown are 2 applications, 9 forwarders and a controller. A separate management network shown in blue connects the controller to the forwarders, and traffic is captured at a forwarder labelled 'forwarder 2'.

Starting with a simpler topology allowed a more incremental approach to the development of the individual components, as the forwarders and application could be verified to behave as expected before introducing more complexity.

Links between forwarders were implemented using separate Docker networks, so that the source and destination forwarders could not directly communicate with each other, meaning the packet forwarding functionality was definitely being tested.

In the updated, more complex topology, traffic was captured from the point of view of the forwarder labelled 'Forwarder 2' in Figure 8, by a tcpdump container which piggybacked the network stack of the forwarder. Note that with this approach, the forwarder registration is missed since the registration happens immediately upon startup and the traffic capture container can only be started when the forwarder is running. Below is the result of this traffic capture.



Time	Source	Source Port	Destination	Destination Port	Length
0.000000	127.0.0.1	3000	127.0.0.1	51510	65
0.009377	192.168.80.10	51510	192.168.80.2	51510	65
0.016454	192.168.80.2	51510	192.168.80.10	51510	136
0.441280	192.168.80.2	51510	192.168.80.10	51510	136
0.888292	192.168.80.2	51510	192.168.80.10	51510	136
1.023441	192.168.80.2	51510	192.168.80.10	51510	224
1.940614	192.168.80.2	51510	192.168.80.10	51510	224
2.207502	192.168.80.2	51510	192.168.80.10	51510	224
6.427816	127.0.0.1	3000	127.0.0.1	51510	80
6.431673	192.168.128.2	51510	192.168.128.3	51510	80
7.797097	192.168.128.3	51510	192.168.128.2	51510	80
7.799108	127.0.0.1	51510	127.0.0.1	3000	80
11.430658	127.0.0.1	3000	127.0.0.1	51510	80
11.433828	192.168.128.2	51510	192.168.128.3	51510	80
12.799518	192.168.128.3	51510	192.168.128.2	51510	80
12.800881	127.0.0.1	51510	127.0.0.1	3000	80
16.432129	127.0.0.1	3000	127.0.0.1	51510	80

Figure 9: A screenshot of a wireshark analysis of traffic captures from the topology shown in Figure 8. This capture shows an application registering with the controller via a forwarder, the routing tables being updated several times as changes are made to the network, before data starts being sent between the applications over the network.

4 Discussion

A major source of difference between this protocol and the likes of IPv4 (Internet Protocol Version 4) or IPv6 (Internet Protocol Version 6) is how the addresses are encoded.

In one sense the main strengths and weaknesses of the protocol are derived from this change to how addressing is done. Since addresses are strings they can become human readable, a shortfall in IP addresses, which must rely on external name resolution protocols such as DNS (Domain Name System) to map names to addresses. This leads to dependencies on the name resolution service, which should that service suffer an outage, the rest of the network is effectively offline for a significant portion of end users. This protocol does not have any external dependencies for name resolution, since there is no translation required to make addresses human readable.

A further advantage is that field format is very flexible, with expansion of the header relatively easy through the assignment of additional message or field types, should they become necessary in the future. This is possible since out of a possible 256 message types, the maximum allowed with a 1 field, only 6 have been allocated.

This approach of string based addresses taken by this protocol does have its downsides, though. Since it is desirable to be able to type any address manually, addresses are effectively limited to 64 characters (52 characters of the upper and lower case alphabet, 10 digits, the underscore and the hyphen), each byte of an address only makes use of 6 of the

possible 8 bits. IPv4 on the other hand is much more efficient, making use of all of the bits allocated to the address.

5 Reflection

In comparison to the previous PubSub assignment, I feel this assignment went a lot smoother. As the general structure of the assignments were roughly the same, in that they were both assignments requiring the design, implementation and documentation of a protocol, I was able to apply lessons learned from the previous assignment, and take different approaches to overcome difficulties more easily.

A key learning point from last time which helped hugely was to keep designs simple until there's a genuine need to introduce any kind of complexity. For example, with this assignment, the packet format was designed with specific use cases and mechanisms in mind, which is in contrast to the PubSub assignment, where I added fields which I thought would be helpful, then tried to work out what functionality was possible. This led to some unnecessary bloat in the specification for the header, which did not happen this time around.

A further simplification came within the implementation. This time around, after noticing how complex the packet decoding was, even for a protocol with fixed length fields, I decided to use a parsing and encoding library.

Overall I feel that I have learned a lot from completing this assignment, and am pleased with how it has gone.