Oliver Black

Software IPv6 Router in Rust

Computer Science Tripos – Part II

Selwyn College

May 5, 2019

Declaration of Originality

I, Oliver Black of Selwyn College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Oliver Black

Date May 5, 2019

Proforma

Name:	**Oliver Black**
College:	**Selwyn College**
Project Title:	**Software IPv6 Router in Rust**
Examination:	**Computer Science Tripos – Part II, July 2001**
Word Count:	**8000** [1]
Project Originator:	Oliver Black & Richard Watts
Supervisor:	Andrew Moore

Original Aims of the Project

The IPv6 standard contains a large number of complex requirements, complicating understanding. I aim to design and implement a simple IPv6 router (called Luyou) using Rust[2] that behaves as specified in the IPv6 RFCs[7]. This router should implement the minimum functionality required by the relevant standards, yet still be functional, minimal, & stable. Rust is a new programming language that aims to be as fast as C while maintaining memory safety, I wanted to understand how practical it was to develop in.

Work Completed

Despite having initial difficulties setting up my test environment (called Luxing) using Mininet[3], due to its lack of support for IPv6, I successfully implemented a functioning IPv6 router in rust that met almost all of my core requirements. Both the Luyou itself, and the Luxing, are available for public use. TODO mention speedup/code coverage/size/RFC coverage/throughput AND fill rest in when main body done. mention how ambitious original claims were

Special Difficulties

None. TODO fill in

[1] This word count was computed by copying the relevant part of the dissertation into `https://wordcounter.net/`

Contents

Cover Sheet **1**

Declaration of Originality **2**

Proforma **3**

Table of Contents **6**

List of Figures **7**

1 Introduction **9**

2 Preparation **11**
2.1 Starting Point 11
2.2 Research 12
2.3 Analysis 13
2.4 Design 16
2.5 Test Plan 18
2.6 Professional Practice 19

3 Implementation **21**
Repository Overview 21
3.1 Router 22
3.2 Test Bench 25
3.3 Software Engineering 26

4 Evaluation **29**
4.1 Tests 29
4.2 Issues and Observations 30
4.3 Ambitious Case 32
4.4 Other Metrics 32
4.5 Goals 32

5 Conclusion **33**

Bibliography **35**

A Requirements **37**

Project Proposal **43**

List of Figures

2.1 IPv6 Header Format[7] 12
2.2 Router Design 16
2.3 Layer Separation 17

3.1 Code for starting a transmitting thread, returning a thread safe pipe . . . 23
3.2 Excerpt from Ethernet layer: creates and IPv6 packet from a new buffer, lends it to the IPv6 layer, then copies the result into the Ethernet packet . 24

Acknowledgements

Many thanks to:

- My supervisor Andrew Moore for his helpful advice.
- My Director of Studies Dr Richard Watts for his guidance.
- Friends & family for proofreading.

Chapter 1

Introduction

Slowly but surely the internet is making progress towards IPv6, but how do pages of Requests for Comments (RFCs) translate into real world network components? The aim of this project was to develop an IPv6 Router in Rust (Luyou) that explores the functionality of IPv6, and how different parts of the various standards fit together. The project has been a success, I have produced a functioning router and accompanying test suite (Luxing).

The router is called Luyou as 路由器 (Luyouqi) is Chinese for router, and the router was primarily written while travelling in China, and 旅游 (Luyou) is Chinese for travelling. 旅行 (Luxing) is also Chinese for travelling, but could also mean 路行 (Luxing) which could be taken to be an abbreviation for "Router, OK", so Luxing is the name of my test bench. The small test client and test server are called Luxingke and Luxingfu respectively, as 客 (Ke) is client and 服 (Fu) is server.

Due to the popularity of the Internet, there are now not enough IPv4 addresses to go around. IPv6 is the incoming internet addressing standard that solves numerous issues with IPv4. Primarily it increases the number of addresses, however it also fixes many flaws in the IPv4 design, and standardises common non-standard practices. For example, the Time To Live in IPv4 was defined partly in terms of seconds left to live[9], but in practice was just decremented by 1 on every hop between nodes. In IPv6 the field is accurately renamed to Hop Limit, and is now defined in terms of hops between nodes (as opposed to seconds). Many subtle decisions like this have gone into the IPv6 standard, with an aim to making an internet that works well, rather than one that just works.

Rust[2] is an up and coming modern low level programming language. It aims to match the performance of C/C++ without sacrificing memory safety, and avoiding garbage collection. It does this through zero-cost high level abstractions such as *ownership* and *lifetimes*. For example, if you pass something to a function, that function then owns that and everything it owns, with it being inaccessible after the function returns. It is possible for functions to borrow values instead, using '`&`', similar to passing by reference. I chose Rust for my project as it can be easier to debug than C or C++, but mainly because I was interesting in learning Rust.

Mininet[3] is an open source virtual network simulator that was developed at Stanford and until 2016 was used in the Part 1B Computer Networking course, it is written in Python. It creates lightweight virtual networks by making use of Linux's *networked namespaces*, allowing processes to share a kernel, yet be behind different network interfaces. This made it the ideal candidate to build Luyou and Luxing on top of. A simple IPv4 router[14] already exists, and can be ran on top of Mininet, it explores how IPv4 works quite effectively. Seeing this was one of the key inspirations for my project.

Routers are the backbone of the internet, at the most simple level many of them have a *control plane* that deals with addressing, and a *forwarding plane* that deals with actually sending packets. There are many open source routers out there, but almost all of them have lots of IPv4 code. This makes it difficult to isolate and understand how the IPv6 part actually works. Starting from scratch allows you to avoid having to deal with IPv4 at all.

Using the IPv6 standard as a framework, combined with some knowledge about the internals of routers, it is possible to develop and IPv6 router that is stable, small, simple, & fast. Such a router could continue to be developed until it could be deployed on actual hardware, but the implementation and testing required meant this was not an objective of this project. Instead, the aim is to develop a router that implements a sub-set of the IPv6 standard, hopefully including everything an IPv6 router is required by the standards to implement. In the remainder of this dissertation I will discuss the preparation, implementation, and evaluation of this project.

Chapter 2

Preparation

Before starting the implementation lots of research and design needed to be done. Lots of research was done into IPv6 RFCs and which aspects were required to be implemented, and which were not. The router software was then designed to provide a framework within which these aspects could be implemented. Additionally a test plan needed to be made, to enable effective evaluation of the finished product.

2.1 Starting Point

This project was in areas I was interested in and had some experience in. Those areas were all, however, approached from new angles:

- **Low-level Systems Programming:** As well as the Part 1B C course I had done several internships that involved a substantial amount of low-level programming in C. However I had never written any project in Rust before.
- **Network Programming:** I had completed the Part 1B Computer Networking course, so had some theoretical understanding. I had also worked on side assignment of a large networking project during an internship. However I had never worked on a networking project by myself from the ground up before.
- **Testing:** I had obviously written tests for my own code before, and had been exposed to large testing frameworks during internships. However I had never devised my own formal test plan and developed my own test bench before. I also only had brief experience with Python.

In terms of existing software, the following where used or built upon by my project:

- **pnet**[4] the packet and interface libraries in pnet allowed packets to be received, and for individual packets to be manipulated.
- **Mininet**[3] allowed virtual network environments on Linux to be setup with IPv4 addresses.

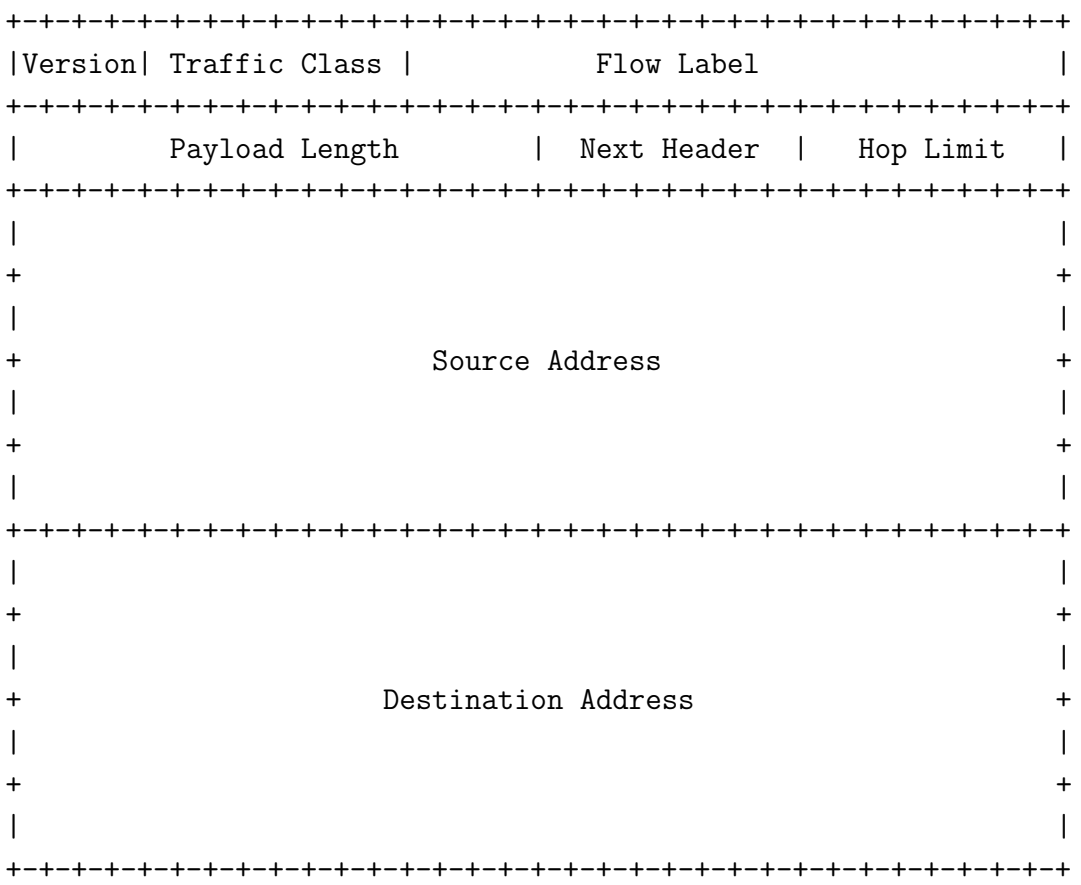

Figure 2.1: IPv6 Header Format[7]

2.2 Research

An analogous project for IPv4 called Simple Router already exists[14] (it used to be a recommended extension task for the Part 1B Computer Networking course). It is implemented in C, but it ran on top of Mininet. The implementation didn't help at all (due to being in C and for IPv4), but it running on Mininet demonstrates that running a router on Mininet is feasible. Additionally the Mininet python code that ran the Simple Router's executable helped me in designing Luxing.

Most of my research time however was spent reading RFCs related to IPv6. The main RFC[7] specifies everything you need to know about IPv6 packets. This describes the contents of the main IPv6 header, Figure 2.1, including how the fields (e.g. *Hop Limit*) are modified for packets in transit. Additionally this includes the extension headers that need to be implemented by a router, when the packet in question is not addressed to the router this turns out to be none.

Another important RFC was the Internet Control Messaging Protocol RFC[10]. This protocol accompanies IPv6 proper, and must be implemented by all IPv6 routers. It allows, among other things, errors about dropped packets to be sent back to the source, and *Echo Request/Reply* (ping) messages to be sent. Alongside this it was important to gain an understanding of how IPv6 related to the link layer below and the transport layer above.

The IPv6 Addressing RFC[8] contains all you could want to know about the various kinds of IPv6 addresses. However, it is mainly just a list of address ranges and whether they have any special behaviour, it doesn't affect the design of a router much. I also consulted a few more RFCs related to features that didn't *need* to be implemented by an IPv6 router, more details on those can be found under Appendix A.

Finally I spent some time researching rust, as it was a new language to me, and I didn't want to make mistakes early on in my implementation that would make things much harder later on. I discovered a library called pnet[4] which implemented low level networking functions and packet abstractions, exactly what I would need.

2.3 Analysis

After finishing my research I needed to do some *requirements analysis* to workout what exactly Luyou (my router) needed to implement, and in what order I should go about implementing them. I divided up the requirements as recommended into *core* and *extension*, where core contained everything an IPv6 router *needed* to do (according to the RFCs), and extension things I thought I would like it to do as well. Core was further divided up into *basic* and *advanced*, with basic being everything a router required to provide some form of basic testable functionality, and extension being everything else that was *needed*.

There are three main areas of requirements:

- Addressing
- Packet inspection & forwarding
- Error reporting & ICMPv6[10]

The requirements for addressing can be divided into two parts, the address discovery mechanism (static, SLAAC[11] or DHCPv6[12]) and different address types (Unicast, Anycast, Multicast).

Although DHCPv6 and SLAAC are both practical and interesting, they aren't *needed* for an RFC router - static addressing is sufficient - so they were put as extension requirements. Static addressing means the relationship between IPv6 addresses and link layer interfaces is defined when the router starts based on a fixed configuration. In order to get the router forwarding packets as quickly as possible an additional requirement of *flooding* addressing was added. This is not defined in the RFC for IPv6, as it means a router would instead by functioning as a link layer switch, sending all incoming packets out on all interfaces. Static addressing was *needed* by the RFCs and I decided flooding alone wouldn't really constitute basic testable functionality (of a router). So static addressing is a basic core requirement, with DHCPv6 and SLAAC being extension requirements.

Address types in IPv6 are well defined by the addressing RFC[8], hand a router *needs* to deal with all of them. However, in order to test basic functionality only Unicast really needs to be implemented, as Anycast and Multicast are just mappings from a 'Unicast'

address to many Unicast addresses. So Unicast is a basic core requirement, with Anycast and Multicast being advanced core requirements. IPv6 also includes scope for local only addresses, as well as a variety of other specific types, these could also have been extension requirements.

Every packet a router receives needs to be sent to the right destination, and any packet for which the destination is unknown should be sent to the default route. This falls under static addressing as my requirements don't differentiate between the control and data plane - see Design. However the payload length must be checked to see if it matches the actual length of the payload - and the packet discarded if not. Additionally the hop limit must be checked, if it is 1 or 0 the packet should be discarded, otherwise it should be decreased by 1. Both of these are basic core requirements, they are *needed* and without them it is hard to test basic functionality. Without hop-limit decrements a router leaves packets unaffected, so it is hard to tell if they went through a router at all.

IPv6 also has many extension headers, but they *need* to be ignored by intermediate nodes (for example, fragmentation can only be done by the source and destination nodes), except for the *hop-by-hop options header* which can be ignored by intermediate nodes. Apart from ICMPv6 packets Luyou does not deal with packets that are encapsulated by IPv6 packets (including transport layer protocols). This means that it does not need to process any headers at all, except for ICMPv6 packets.

ICMPv6 works alongside IPv6 to send informational and error messages between nodes. These messages include destination, packet size, hop limit, & header error messages, and echo request & reply informational messages. One slightly odd requirement relates back to hop-limit in the IPv6 header, if an arriving packet has a hop limit of 1, it should be discarded, unless the destination node is the router in question. This only occurs for ICMPv6 messages with Luyou, as it does not support receiving any other packets

ICMPv6 is *needed* for any IPv6 router. However, in order to test the basic functionality of Luyou it was not required. This is because error reporting functions can be, in part, replaced by log output from Luyou itself. As such, ICMPv6 is an advanced core requirement.

Additionally I thought about what my project did not need to do. I have already discussed briefly in my Introduction that I did not want to require Luyou to necessarily be able to be ran on actual hardware, as this would add needless complexity when my aim is to explore and illuminate the IPv6 requirements.

I did not want to bother with any experimental/unused features, as they don't much help understand the IPv6 requirements. For example the flow label in the IPv6 header can in theory be used to prioritise real time packets, and several other uses have been suggested, however it can just be ignored by the standard, and I believe that is what many routers do. The same can be applied to many IPv6 extension headers, I planned to evaluate all my extension requirements to see if they met this criterion after finishing my core requirements.

I didn't want to get involved with cross-layer optimisations, as these would also not help to understand the IPv6 requirements. Secondly, this would not be particularly useful, as the nature of Mininet creating perfect virtual ethernet links means Luyou has few bottlenecks, but it is often in bottlenecks that such cross-layer optimisations are used.

Mininet does allow for artificially reducing the MTU of links, and through the help of helper applications artificially dropping packets and increasing latency. However, my aim was not to produce a high performance router, so any requirement based on performance was also not what I was looking for.

To summarise, here are the requirements for Luyou, categorised by type, starting with my basic core requirements:

- Send packets to the correct hardware interface in accordance with the static routing rules provided
- Deal with IPv6 headers in accordance with the standard (Hop limit, etc)

My advanced core requirements:

- ICMPv6
- Multicast
- Anycast

My extension requirements:

- IPv6 extension headers
- DHCPv6
- SLAAC

My non-goals, things that would needlessly complicate the project:

- To be stable, complete, fast, & compatible enough to be easily run on real hardware in a real world environment.
- Implementing experimental or unused features - specifically those potentially covered by my extension requirements.
- Higher layer packet inspection
- Performance based goals (throughput, etc)

For a formal list of the requirements I came up with (along with identifiers and associated tests) see Appendix A.

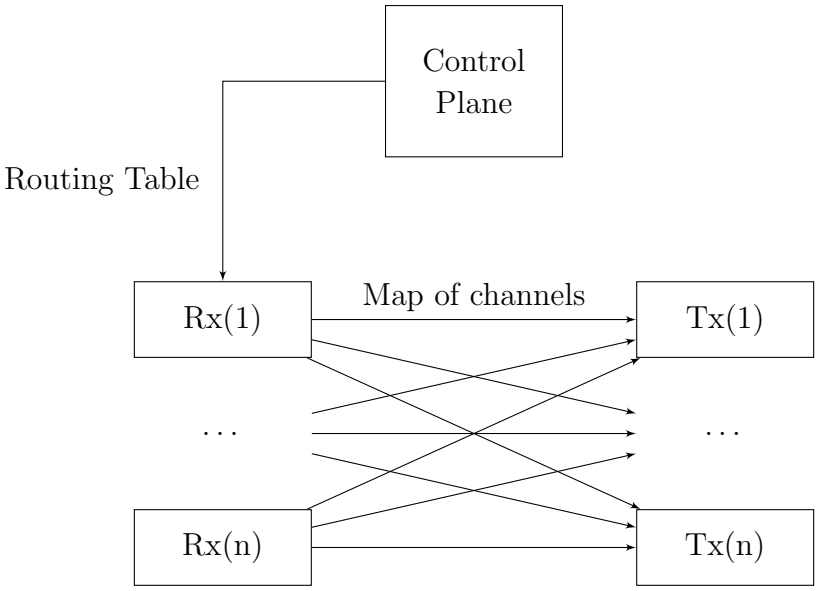

Figure 2.2: Router Design

2.4 Design

Having completed my analysis and produced a structured list of requirements the next step was to come up with a design for Luyou that would enable me to implement these requirements. There were two main ideas in the design Luyou itself itself (the design of the Luxing (the test bench) is discussed in the next section).

The first was the separation of the control and forwarding plane, which by itself is nothing special, but the design in software was slight more complex. As always, the control plane deals with the addressing (along with other aspects and the forwarding plane with link layer interfaces and individual packets. The two communicate through forwarding tables (in the case of non static addressing communication in the other direction is also required). The forwarding plane in my case is made up of a pair of threads for each interface, one receiving and one transmitting. All of the receiving threads have read access to two objects, one is the routing table produced by the control plane, the other is a map of hardware addresses to channels. The channel each hardware address is linked to leads to the transmitting thread for the interface of that hardware address (See Figure 2.2). Read Implementation for specifics.

The second was designing the layer separation inherent in the TCP/IP stack into Luyou. This was achieved by having a different function handle each layer, and only passing necessary information between the functions. There are three layers, Ethernet, IPv6, and ICMPv6 in Luyou, so there are three functions, with only the packets themselves and the relevant addresses being passed between the functions. The Ethernet function sends the IPv6 packet to the IPv6 function, which returns the new IPv6 packet, and the hardware address to send it to. The IPv6 function sends the ICMPv6 packet along with the source and destination address to the ICMPv6 function, which returns the new ICMPv6 packet along with the new source and destination addresses (See Figure 2.3).

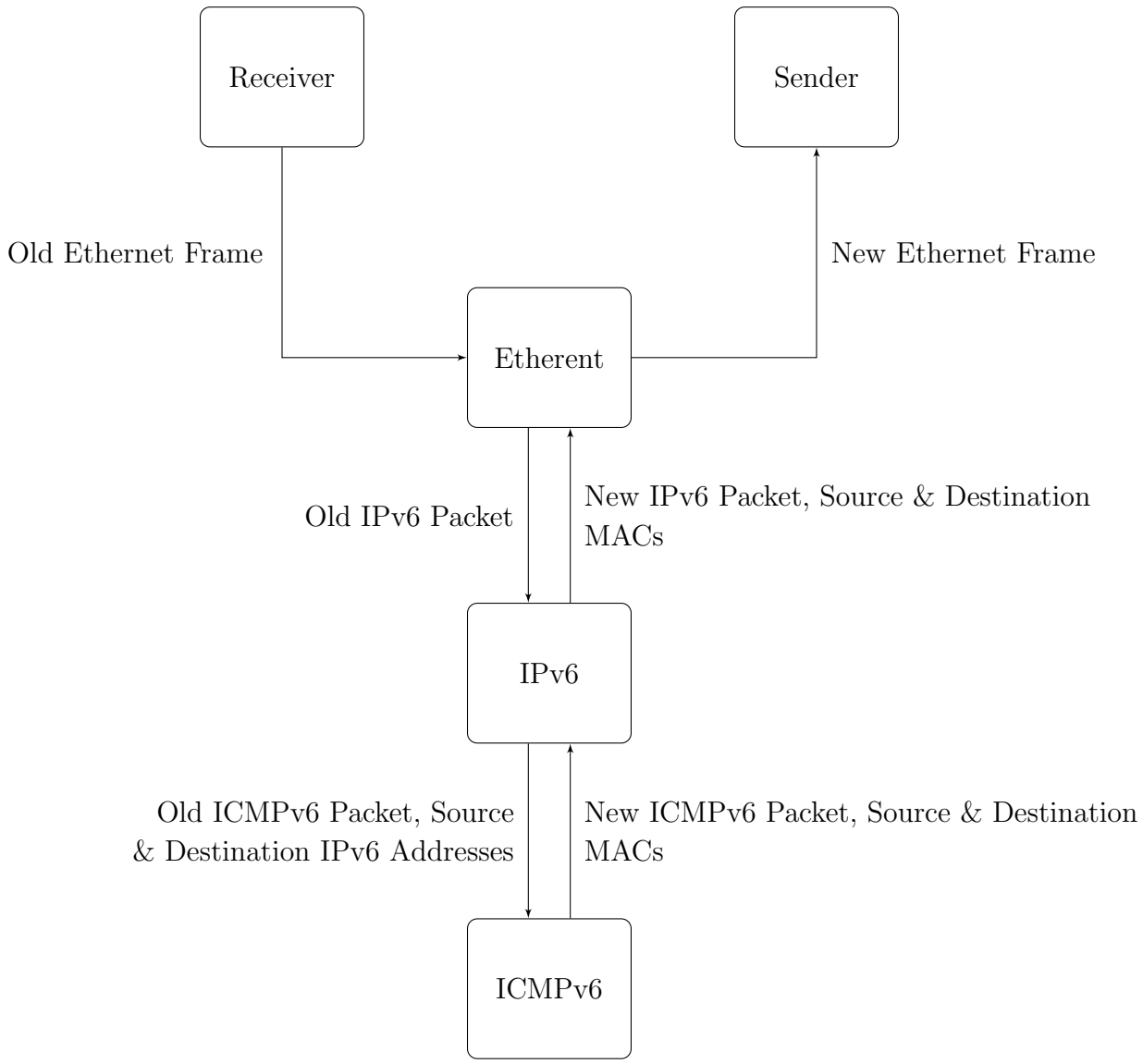

Figure 2.3: Layer Separation

Both of these design choices made the implementation much easier, separating the code into functionally separate sections, reducing the risk of introducing bugs, and most importantly providing a structured framework within which to code.

2.5 Test Plan

With requirements and a design of Luyou completed, I needed to come up with a plan of how I was going to verify that Luyou was functioning as expected. This was divided into two parts, the design of Luxing, and secondly the list of tests to be run on it.

I planned to make Luxing from Mininet and a couple of helper applications (either written in Python or Rust). Some helper applications would be capable of sending packets with specific properties, and outputting any packets they received. Other helper applications would do more complicated end to end testing, such as running a web server and requesting web pages. Unfortunately I found Mininet's IPv6 support was far less complete than I had believed in my Project Proposal. I either had to write my own network emulation environment from scratch, or develop some kind of modification or wrapper for Mininet. I chose to focus on a wrapper that would intercept the functions I needed to use, and change the necessary things to get Mininet working, details of this wrapper can be found in Implementation.

When formalising the results of my analysis into a list of tests there were two key steps. The first, a one, was numbering everything so I wouldn't get lost or forget any requirements. Secondly I had to come up with the tests themselves, this normally included a test for the given functionality when it should happen, a test for it not happening when it shouldn't happen, and any edge case tests I could thing of. For example, hop limit decrement, when a packet passes through a router its hop limit should be decreased by 1 and the packet sent on its way; when the packet has a hop limit of 1 or 0 on arrival it should be dropped and not sent on its way, and if, the packet has a hop limit of 1 and is addressed to the router it should not be dropped, it should instead be handed up a layer.

To avoid all the tests being dependent on an almost complete implementation of Luyou I avoided tests for features relying on other features. If we return to the example hop limit given above Luyou should send an ICMPv6 *Time Exceeded* message when a packet is dropped. However, this would mean any test for hop limit depends on a correct implementation of ICMPv6, as well as the correct implementation of Hop Limit. As a result every test can only rely on functionality already implemented. My numbering system happened to match my planned order of implementation, so just reading off all the previous tests lets you know which features that test potentially relies on. Back to the hop limit example, this means that success of the hop limit test was judged by the packet not continuing and log messages being outputted by Luyou. When it came to test Time Exceeded though it could rely on the already implemented hop limit, so the planned test involved sending a packet with hop limit 0 and testing for a Time Exceeded response.

2.6 Professional Practice

This is a brief section on preparation related to professional skills and practice. Two large aspects are testing and software engineering, both of which are explained in detail in the rest of Preparation and in Implementation.

There could be security exploits in my code, and if anybody would like to adapt my code for use in a real world router I would recommend it is tested extensively.

In terms of licensing my code makes use of Mininet, Rust, the Rust standard library, and pnet. However, none of these are distributed with my project, so there are no licensing requirements. That said, my code is released under the GNU GPL[15], which is compatible with their Apache 2.0/MIT licenses.

I don't believe there are any ethical issues involved in my project, and there is no human testing.

Chapter 3

Implementation

Implementing Luyou was an exciting iterative process, with many challenges, but also a real sense of achievement every time I implemented something I had never done before and again when it passed my tests. There were two main parts to implement, Luyou (the router itself) and Luxing (the test bench). Both had their own different challenges. The relationship between the two is explained in more detail in the third section about Software Engineering practices used.

Repository Overview

Here is the repository overview, the project is made up of a python application (the test bench - Luxing) and three rust applications (a test client - Luxingke, a test server - Luxingfu, and the router itself - Luyou). To shorten and explain the tree `*comments*` have been added inline. All of these files were written from scratch except for the few examples commented as `from documentation`.

```
.
├── dissertation
│   └── *latex files*
├── python
│   ├── luxing
│   │   ├── __init__.py
│   │   ├── test_*test number*.py
│   │   ├── test_example.py
│   │   ├── test_framework.py *mininet wrapper*
│   │   ├── test_playground.py
│   │   └── test_tw_example.py *example mininet application from
│   │                           documentation*
│   └── __init__.py
├── rust
│   ├── luxingfu
│   │   └── src
│   │       └── main.rs *test server*
```

```
    ├── luxingke
    │   └── src
    │       └── main.rs *test client*
    └── luyou
        ├── resource
        │   └── routing.txt *static routing file*
        └── src
            ├── examples
            │   └── *related examples from documentation*
            ├── control.rs *control plane*
            ├── forwarding.rs *forwarding plane*
            └── main.rs *main router file*
├── COPYING.txt
├── LICENSE.txt
├── README.md
└── test.sh *script to start tests*
```

3.1 Router

Luyou was implemented requirement by requirement, as set out in Preparation andAppendix A. Version control was performed using git, with backups committed to an *exploratory* branch, with merges into the *master* branch whenever a specific requirement had been completely implemented.

The first thing I did was write a very simple Ethernet repeater, that connected to two interfaces and forwarded all packets as is, on the other interface. This demonstrated pnet[4] worked as I expected, and allowed me to move on, and satisfied the flooding requirement. The next step was implementing the the shared map of addresses to interfaces to allow forwarding on a specific interface, then a shared routing table to allow working out which interface to forward on, and the separation of layers to allow easily adding IPv6 feautres and ICMPv6 later.

One of the early issues I encountered was implementing a shared map of transmission channels, as described in Design. There is a method in pnet[4] called `channel() -> Channel`, where a `Channel` is a pair of pipes, one receiving and one transmitting. I planned to give each receiving thread a receiving pipe, and give them all a map from MAC address to transmitting pipe for the sending address. Unfortunately the pipe type used by pnet does not support sharing across threads (and this is enforced by the rust type system). So instead I had to have one additional thread for each transmitting pipe, with another thread safe pipe receiver associated with each transmitting thread, and all the corresponding transmitting thread safe pipes in the map. This worked without much issues, partly because the map never changed, so was cloned across threads. See Figure 3.1 for the code that creates a thread safe channel. Also, the packet representations used by pnet can't be sent over a channel used by multiple threads, as they don't implement

```
//SENDER
pub fn start_sender(tx : Box<DataLinkSender>)
  -> (JoinHandle<()>, Sender<Box<[u8]>>) {
    let (sender, receiver) = channel();
    let handle = thread::spawn(move || sender_loop(tx, receiver));
    (handle, (sender))
}

fn sender_loop(mut sender: Box<DataLinkSender>,
   receiver: Receiver<Box<[u8]>>) {
    loop {
        let packet = receiver.recv().unwrap();
        sender.send_to(&packet,None);
    }
}
```

Figure 3.1: Code for starting a transmitting thread, returning a thread safe pipe

the `Clone` trait, however the underlying byte buffer does implement the `Clone` trait. A trait in Rust is similar to an interface in a language like Java.

Sharing the routing table between layers was difficult. I could have just generated a map of IPv6 addresses to MAC addresses on start and shared it analogously to the map of MAC addresses to interfaces. However, my extension goals included implementing DHCPv6 and SLAAC, both of which require the routing table to be editable. To allow this change without having to edit the forwarding layer too much it made sense to have a separate control thread with write access to the routing table, and give all the reading threads read access. I did this even though as it stands the control thread sets up the static routing then exits. Rust enforces safety, so in order to share the structure between threads it needed to be put inside an `Arc` (Atomically Reference Counted)[5] a thread-safer reference counting pointer. This doesn't actually support the multiple reader single writer semantics required for DHCPv6 or SLAAC, but simply swapping it out for a RwLock[6] would be sufficient.

The static routing rules themselves were read from a text file (that was created by Luxing). It had quite a simple format, the first line was the IPv6 address of the default route, the second line that of Luyou. Every following line was an IPv6 address, then by an '`@`' followed by the pair of MAC addresses for the associated interface (separated by a '`,`'). Little validation was performed when parsing this file, as any time spent improving error reporting in the case of invalid files could be better spent working towards implementing DHCPv6 or SLAAC, and rendering the static addressing obsolete. Rust makes reading text files easy, through parsing traits.

Separating each layer was quite simple, the ethernet layer received a packet, called the IPv6 function, which in turn sometimes called the ICMPv6 layer. My plan was to hand a pair of packets up the layers (the received packet and the new packet to be sent), with

```
let mut buffer = vec![0;new_packet.payload().len()];
let mut new_ipv6_packet = MutableIpv6Packet::new(&mut buffer).unwrap();
new_ipv6_packet.set_payload_length(
    (new_packet.payload().len()-40) as u16);

let (source, destination) =
    match transform_ipv6_packet(
        old_ipv6_packet, &mut new_ipv6_packet, routing) {
    Ok(p) => p,
    Err(e) => return Err(e),
};

new_packet.set_destination(destination);
new_packet.set_source(source);
new_packet.set_ethertype(Ipv6);
new_packet.set_payload(new_ipv6_packet.packet());
```

Figure 3.2: Excerpt from Ethernet layer: creates and IPv6 packet from a new buffer, lends it to the IPv6 layer, then copies the result into the Ethernet packet

each layer stripping off the headers and only handing the rest down to the next layer. Unfortunately this was complicated by the design of pnet. A packet in pnet is merely a wrapper for an underlying byte buffer, this is sensible as it reduces copying. Everything in Rust can only have a maximum of one mutable reference at a time. A mutable ethernet packet in pnet refers to the entire of the underlying buffer, both header and payload. This means that if you have a mutable ethernet packet it is impossible to create a mutable IPv6 packet from the payload and then edit both of them. I believe this is a design flaw in pnet, as the solution is to create a fresh buffer for the ipv6 packet and then copy the payload across, resulting in extra copies. Instead creating a packet from a buffer should return a tuple of the header and the payload, allowing them to be owned and mutated separately, with each owning a different `slice` of the underlying buffer. This can be better seen in Figure 3.2.

ICMPv6 often requires packets be sent not to their destination, but back to their source address. Initially this appeared to be a challenge, as it looked like I needed a completely separate path for ICMPv6 packets addressed to Luyou. However due to how I separated the layers this was not necessary. Instead the IPv6 layer passed the old ICMPv6 packet and the old source and destination addresses to the ICMPv6 layer. The ICMPv6 layer then responded with the new ICMPv6 packet, and the new source and destination addresses (for example, with an `Echo Request` ICMPv6 packet, source and destination would be swapped for the `Echo Reply` response). As long as the IPv6 packet honestly copied these into the new IPv6 packet, and worked out the MAC addresses to pass back to the Ethernet based on the new source and destination addresses, it would work perfectly. This demonstrates how the separation of layers allows new features to leverage existing well tested code effectively, reducing code duplication and increasing reliability.

3.2 Test Bench

As outlined in Preparation a key part of this project is Luxing. Without Luxing it is impossible to verify that Luyou works as expected. Below I go into detail about how I implemented Luxing, and issues I had. Solving these issues involved a lot of trial and error, and I didn't have much experience with networking on Linux.

Initially I thought Mininet[3] supported IPv6. However when I set up a network, and tried to use `ping6` between two nodes it didn't work. This is because Mininet does not support IPv6, and doesn't allocate IPv6 addresses to network nodes. As described in Test Plan I chose to write a wrapper for Mininet that added the IPv6 functionality I required. This wrapper is not as resilient as Mininet itself, it requires operations to be performed in a certain order, that said, it does work, and effectively wraps around Mininet to support IPv6 in a subset of Mininet's use cases.

Almost all the tests run on a 4 node network. Luyou itself is one node, and is in the centre of the network. The leaves of the network are made up of 2 client nodes, these are the nodes on which the tests are run (using Luxingke and Luxingfu). The final leaf is the sink, the default route, which represents the rest of the internet, in your home network it would be the phone or fibre line.

The first issue I faced was allocating IPv6 addresses to nodes. This was achieved by calling `ifconfig [interface-name] inet6 add [address]` every time a node was added. I then stored the node name and IPv6 address in a two way map, to allow me to easily use the addresses later. My wrapper does not support auto allocating IPv6 addresses, you need to specify an address when you create a node, unlike Mininet which allows IPv4 nodes to be created without an address being specified.

The next big issue came when linking the network up. In general it worked fine, as Mininet links nodes together with Veth (Virtual Ethernet) links, which operate at the link layer. However, in order to make it easy to debug I wanted to be able to choose mac addresses for each node. As for IPv6 addresses I just had them specified on node creation. However this didn't work for the router as it had multiple interfaces, so required multiple mac addresses. This was resolved by requiring a MAC address for all links but the first involving the router.

Another issue was adding the default route to all nodes. In order for ping to work effectively the default route needed to be set to the router for all nodes. This is easily achievable using the `ip -6 route add default via [default-address] src [node-address]` command, but this didn't work. This is because when allocated in linux IPv6 addresses take a while before shifting to the `UP` state. Once I had discovered this I added a loop that slept for 100 milliseconds while the output of `ip -6 addr` contained "`TENTATIVE`" before trying to update the default route, this fixed the issue.

As I was using static routing Luxing needed to generate a routing file for the router describing the network it was in. This was relatively simple, on router start a text file

is created, and filled in according to the format described in Router. In order to gather the required information whenever a link was added the relevant IPv6 address and source and destination MAC addresses were stored in a map. Luyou's and the default gateway's addresses also need to be included in the routing file, which required they not be added as ordinary nodes, and instead be added as their own method. For a long time I couldn't get `ping6` to work via my router, through debugging I discovered this was due to `ping6` making use of *Solicited Multicast Addresses* to send requests and responses. A *Solicited Node Multicast* address is most recognisably used by the [**?**][13]. Luxing resolves this by adding the addresses to the static routing file, generating them from the already allocated IPv6 addresses.

I also wrote a small test client and server in Rust (called Luxingke and Luxingfu). Both take an argument that determines which test is being run, the client then produces packets and optionally waits for responses, with the server waiting for packets and optionally producing responses. By using Rust rather than Python for this I could use pnet, which I understood relatively well by this point, avoiding having to learn a new low level networking library.

3.3 Software Engineering

Due to both the nature of my project - having to produce a functioning router - and my own personal goals - wanting to learn useful real world skills - I was extremely aware of how I was applying software engineering techniques throughout.

Overall I began by creating a complete list of requirements, then a complete design, then I implemented my router, and then finally I ran all my tests on it. This is very similar to the *Waterfall* methodology. This was appropriate (compared to more flexible software engineering methodologies) as my requirements were well defined by RFCs. This meant that I did not need to explore and refine requirements as my development progressed, as would be necessary if I was reacting to user feedback or my own improved understanding.

However, I didn't really follow the Waterfall methodology as it was not particularly appropriate or sensible to complete the implementation stage as one large chunk. Instead I completed several prototypes, each implementing a larger set of the requirements than the last:

- **Flooding:** Packets forwarded on all interfaces.
- **Static Routing:** Packets forwarded on specific interfaces.
- **IPv6 validation:** Invalid IPv6 packets dropped and logged.
- **ICMPv6:** Invalid IPv6 packets responded to with ICMPv6 and echo request/reply implemented.

For each of these stages I first refined the requirements (partly based on the successes and failings of the last stage), and at the end of each iteration I would produce a more complete prototype, this is similar to the *Spiral* software development methodology. Doing all of the implementation in one would have prevented me from effectively testing parts of the router independently from each other, and made the final testing much more complex.

Every time I implemented an individual requirement I would write tests for that requirement and run them (ensuring they pass) before proceeding. Once they passed I would merge my work (on the *exploratory* branch) into the *master* branch, before continuing with the next requirement back on the *exploratory* branch. Before merging I would also ensure all previous tests passed. This would involve either fixing Luyou, or in some cases modifying the test itself (for example if it relied on log output that was no longer there due to the implementation of the ICMPv6 response). By merging often and ensuring my router continued passing all tests before every stage I ensured that I was continually making progress, and not shooting myself in the foot by breaking old features, this is similar to *Continuous Integration*.

Chapter 4

Evaluation

Evaluating Luyou was relatively complex. Working from the tests I had written and that Luyou passed was relatively easy. However working out how this fit in relation to the real world was challenging and analysing the success and failure of my project relative to my original goals was even more challenging.

4.1 Tests

As I have already mentioned, I wrote many individual tests on Luxing (my test bench) as I was implementing each requirement. Due to this, when it came to testing Luyou I merely needed to run all the tests and every requirement would be checked.

An issue with such tests is whether or not passing the tests means Luyou actually works. Put another way, passing the tests means Luyou does something, but is that 'something' what I wanted Luyou to be. The tests were defined alongside the requirements, and matched the requirements precisely. So, as long as the requirements are correct the tests would be. Additionally the tests themselves were implemented after each feature was implemented (as opposed to concurrently). This prevents tests being passed by partially hard coding the test itself, and so passing the specific test, but not satisfying the requirement the test is meant to verify. Also the tests deal with many edge cases (see below for more detail). Overall I believe Luyou passing the tests results in a router that matches my requirements.

An example of a test would be test 11212 which is testing for the correct hop limit, it involves sending four packets, two with hop limits 1 and 0, and two more with hop limits 10 and 2. The test passes if the packets with hop limits 10 and 2 are received by the server (Luxingfu) with hop limits of 9 and 1, and if the packets with hop limits 1 and 0 are dropped. A later test, test 1215, deals with a hop limit of 1 or 0 and the destination being the router - where the packet should not be dropped immediately. In the case that the test passes this has tested the general behaviour, the hop limit should be decremented. But more importantly it has tested the edge case behaviour, the two cases where a packet should be dropped, and the case where the packet shouldn't be dropped (but only just).

Some of the tests defined at the requirements stage will fail on the final build of Luyou, this is because the requirement was obsoleted. Earlier I talked about how ICMPv6 being implemented would obsolete tests that rely on log output for invalid packets (as an ICMPv6 message is sent instead). However in reality I left those log messages in, allowing the original test to be used, to reduce the amount of test reworking that needed to be done. However, the test for flooding was removed, due to it being superseded by static routing.

To conclude, apart from the issues explained in the next section, my tests verify that I met my goals. My goal to implement a router that satisfied all my core requirements, and although I didn't implement all of these, for those I did implement the tests effectively verify that I met them. Overall I implemented 10 tests, with each sending and receiving between 1-5 packets. Most importantly for me however, implementing these tests taught me a lot, and I certainly feel better prepared when I have to do something similar in the future.

4.2 Issues and Observations

As with almost all projects, my project didn't go completely smoothly. I had many issues implementing Luxing (my test bench) due to Mininet not supporting IPv6 to the extent I had expected. The time spent on this took away from time that could be spent on Luyou, so I didn't implement as many requirements as I would have liked to.

When I wrote my project proposal I believed Mininet supported IPv6 with minimal or no changes. I discovered this was not the case when I started implementing Luyou. In reality Mininet only supports IPv6 insofar as any generic ethernet network supports IPv6, you can send IPv6 packets over the network, but it won't easily setup addressing at each node for you. Obviously addressing at Luyou's node was handled by Luyou, but each node still needed to be allocated an address. Allocating these addresses and setting up an IPv6 network took a long time, as it was very difficult to test. This was partly because I wasn't particularly knowledgeable in this area, and I was learning a lot about Linux networking and networking tools as I was going along. It was also because all the issues I was facing were as a result of multiple variables interacted (interface configuration, router configuration, python configuration, environment variables configuration), and it was very hard to work out which one was wrong. However I eventually managed to get it working sufficiently that I could work on Luyou.

Whilst I managed to implement every basic core requirement set out in my project proposal. I unfortunately failed to implement some of the advanced core requirements. These include some aspects of ICMPv6, and the Multicast and Anycast requirements. Any other aspect of ICMPv6 not mentioned in the implementation or in the source code that *needs* to be implemented according to the RFC logically cannot occur within my core requirements (for example, it is impossible for most Destination Unreachable messages to be created due to my router having a default route, having no firewall, also the requirement is only a *should*).

All IPv6 extension headers can be ignored by every intermediate node, however they need to be examined by destination nodes. Luyou is only a destination for ICMPv6 packets, so only needs to examine the headers in this case. Luyou doesn't step through headers, so doesn't support any extension headers being used in the case of ICMPv6 packets (but works find with other packets) Routing headers should be checked, and if the *routing type* is unknown the header should be ignored if the *segments left* field is 0, and a ICMPv6 *Parameter Problem needs* to be sent if it is non-zero, my router does not send the ICMPv6 message, so doesn't comply with the standard. Fragment headers should also be checked, my router doesn't support fragmented ICMPv6 packets. Neither of these should be a problem, as the only ICMPv6 packets my router supports receiving are Echo Request packets, which rarely have a Routing Options header, and shouldn't need to be fragmented (they are considerably less than one MTU). In addition to not supporting fragmentation in IPv6 packets, Luyou also doesn't support sending Time Exceeded messages with code 1, i.e. Fragment Reassembly Time Exceeded.

Part of the ICMPv6 requirements is that a router must rate limit the number of ICMPv6 packets it sends out, my router does not do this for two reasons. It would require a whole different set of test tools to look at rate of packet's sent and perform load testing. Also, my router exists in a perfect software environment, so performing rate limiting when no other rate limiting exists seems a bit artificial.

The largest part of my core requirements that went unimplemented was IPv6 Multicast and Anycast addressing. This was purely due to time constraints, and due to the requirements being relatively self contained, not relied on by anything, nor affecting anything. This also included checking the scope of addresses, not allowing link-local addresses to be sent to the gateway (and sending an ICMPv6 message in this case). Implementing this would be relatively simple, it would require changes to the control plane to validate and support all the different types, and then the addition of methods that can be called by the forwarding plane, triggering packets to be dropped and messages to be sent as appropriate. Multicast and Anycast also wouldn't be too complex under a static routing scheme, with Anycast being the same as Unicast (due to the network being static), and Multicast requiring packets be sent on multiple interfaces.

I didn't implement any of my extension requirements. I was very interested in implementing SLAAC or DHCPv6, but unfortunately didn't have time. I also would have implemented extension headers, many of which are sparsely used, but it would have been extremely interesting to see how they work and interact with each other.

I didn't perform much unit testing as the simplest of my end to end tests passed without too much effort. This was in part due to Rust and how it prevents C-like behaviour where the wrong memory location is unexpectedly edited.

4.3 Ambitious Case

Alongside the tests in Luxing I also attempted testing some more ambitious cases. First I tried 'pinging' between all the nodes, then secondly I tried running a web server on one node and serving a web page.

Mininet did not have an method that pinged all nodes using *ping6* (for IPv6), so I had to implement one myself. This worked on my flooding network, however when I add the hop limit decrement to Luyou it stopped working. No matter how much I tried I couldn't understand why this was the case, it worked fine with the `-1;` commented out, but didn't work with it in. Only one octet of the header was changed, I checked it was the correct one by outputting the packets before and after at the router. This octet was not included in the ICMPv6 checksum either (it takes the ICMPv6 packet and the destination and source addresses). This remains unsolved, but the final version of the router, with hop limit decrement turned off, does allow all nodes to see each other.

Secondly I attempted to run a web server on one node and request a web page from it with another node. Although I think got the web server to work I couldn't successfully get the web page request to work. This could be due to any of the following reasons. The web server didn't bind to the right interface or address. The client didn't have the right destination or source address (it didn't match the one Luxing wanted it to use). Or for a reason related to why ping6 fails (see previous paragraph). Either way, I attempted this quite late in my project, so even though I was disappointed it didn't just work, I wasn't too surprised either.

I believe the failures in both of these cases were down to quirks of networking in Mininet and Linux combined with the slightly shaky implementation of Luxing, and not due to failings in Luyou itself.

4.4 Other Metrics

I also looked at some other metrics to evaluate my router.

The total length of Luyou's source code is only 740 lines, which is really a testament to how concise Rust makes error handling and parsing code.

In terms of test coverage, the tests included in Luxingfu cover approximately 95% of the code

code coverage Code size 736 lines How much of RFC (ex references) - IPv6:; Addr: ; ICMPv6: 14 of 18 - unsure Test coverage - 95

evaluate test bench - ease of use

4.5 Goals

Were they achieved?

Chapter 5

Conclusion

lack of REAL WORLD end to end testing

What would I do better - earlier prototype to check existing tech

time on mininet - but still did most of core

goals

tests

Capitilisation, grammar, spelling, test bench or suite, forwarding or data plane, capital Rust, lower case pnet, ethernet caps? https://www.cl.cam.ac.uk/teaching/projects/pinkbook/node18.html Search for identifying information - EDIT PROFORMA PER PINK BOOK/GROUP CHAT Capitalise MAC - edit declaration Document of notes and pdf from how to write a dissertation

HOW TO BUILD AND RUN

literature review

add personal goals?

link basic requirements to project proposal

Update appendices

TODO remove TODO

Bibliography

[1] Software IPv6 Router in Rust repository, the code that accompanies this dissertation, https://github.com/ollie299792458/dissertation-rust-ipv6-router

[2] Rust, a modern low level programming language, https://www.rust-lang.org/

[3] Mininet, a network virtualisation library in Python, http://mininet.org/

[4] pnet, a low-level networking API for rust, https://docs.rs/pnet

[5] Arc, a thread safe reference counted pointer for Rust, https://doc.rust-lang.org/std/sync/struct.Arc.html

[6] RwLock, a multiple reader single writer lock for Rust, https://doc.rust-lang.org/std/sync/struct.RwLock.html

[7] Internet Protocol, Version 6 (IPv6) Specification, RFC 8200, July 2017

[8] IP Version 6 Addressing Architecture, RFC 4291, February 2006

[9] INTERNET PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION, RFC 791, September 1981

[10] Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, RFC 4443, March 2006

[11] IPv6 Stateless Address Autoconfiguration, RFC 4862, September 2007

[12] Dynamic Host Configuration Protocol for IPv6 (DHCPv6), RFC 3315, July 2003

[13] Neighbour Discovery Protocol, Neighbor Discovery for IP version 6 (IPv6), RFC 4861, September 2007

[14] Simple Router, implementing an IPv4 router in C on Mininet, https://github.com/mininet/mininet/wiki/Simple-Router

[15] GNU General Purpose License, https://www.gnu.org/licenses/quick-guide-gplv3.html

Appendix A

Requirements

Requirements

By Oliver Black
The requirements below have been taken from the RFC, each requirement includes a description an example test, and in some cases a list of edge cases tests, tests and descriptions for extensions have not been given if no work has been done on their implementation.

Core - 1

Basic - 1

Control Plane - 1

Flooding - 1111

Every packet received should be forwarded to every interface but the originating one. Overridden by static.
Test: Given a packet, check every other interface receives it, and the sending interface does not
Edge cases: No other interfaces

Static - 1112

Read a configuration file with known addresses & interfaces, forward as per the file, else send to default route (also defined in the file)
Test: Given packets from every known address to every known address sent correctly, with no additional sends
Edge cases: Address not known, address very similar (last/first bit different), loopback address, multicast address, unspecified address, loopback address, link local addresses

Data Plane - 2

Header - 1

Payload Length - 11211

https://tools.ietf.org/html/rfc8200#section-3
Must correctly forward the correct amount of data based on the length in this field
Test: Given a packet containing X in this field must forward X bytes
Edge cases: Packet length 0, Packet length = packet size, packet length > packet size, packet length < packet size

Hop Limit - 11212

https://tools.ietf.org/html/rfc8200#section-3

Must correctly discard all packets to be forwarded with a hop limit of 0, and decrement the hop limit of all packets
Test: Given a packet with a hop limit of X, hop limit should be X-1 when forwarded, or not forwarded if X-1 = 0
Edge cases: Hop limit = 0, Hop limit < 0, Hop limit = 0 but router is recipient, Hop limit < 0 but router is recipient.

Addressing (Unicast) - 3

Internal Structure - 1131

Assume addresses have no internal structure - simplest
Test: Random destination addresses that are not in the immediate network are always treated the same

Advanced - 2

ICMPv6 - 1

https://tools.ietf.org/html/rfc4443

Checksum 1211

ICMPv6 packets with incorrect checksums should be dropped (only if router is destination)
Test: Random bit changes to packets so that checksum is incorrect

Unknown 1212

ICMPv6 packets of unknown type must be silently discarded (if router is destination)
Test: Packets with a variety of unknown types that are otherwise valid

Rate limit 1213

Router must apply some form of rate limiting
Test: Try and trigger more than limit of responses

Packet too big 1214

Sent to sender when a packet cannot be forwarded due to size
Test: Send a packet that is too big to be forwarded

Time exceeded 1215

Sent to a sender when a packet's hop limit is decremented to 0
Test: Send a packet with hop_limit of 0 or 1

Echo 1216

Must reply to echo request messages
Test: Send an echo request message

Parameter Problem 1217

Sent to a sender when there is an issue with an ipv6_header
Test: Send packets with an erroneous header, unrecognised next header type, and unrecognised ipv6 option

Uniquely Identify 1218

Do not send ICMPv6 responses if a packets source address is the unspecified address, a multicast address or an anycast address.
Test: Send response triggering messages with the above source addresses

Multicast - 2

https://tools.ietf.org/html/rfc4291#section-2.7
Solicited node address

(e.6) A packet whose source address does not uniquely identify a
single node -- e.g., the IPv6 Unspecified Address, an IPv6
multicast address, or an address known by the ICMP message
originator to be an IPv6 anycast address.

Anycast - 3

https://tools.ietf.org/html/rfc4291#section-2.6

Extension - 3

Optional Requirements from Core - 1

Traffic Class

https://tools.ietf.org/html/rfc8200#section-7

Flow Label

https://tools.ietf.org/html/rfc8200#section-6

Next Header

Header field, may be used to optimise

Text Representation of Addresses and Prefixes

https://tools.ietf.org/html/rfc4291#section-2.2
For logging

More Unicast

https://tools.ietf.org/html/rfc4291#section-2.5

Extension Headers - 2

https://tools.ietf.org/html/rfc8200#section-4

- Hop-by-Hop Options - https://tools.ietf.org/html/rfc8200#appendix-A
- Fragment
- Destination Options
- Routing - https://tools.ietf.org/html/rfc8200#section-8.4
- Authentication
- Encapsulating Security Payload

And order checks
Check IPv6 parameters for full list

DHCPv6 - 3

https://tools.ietf.org/html/rfc3315

SLAAC - 4

https://tools.ietf.org/html/rfc4862

TCP & UDP - 5

https://tools.ietf.org/html/rfc8200#section-8 but not 8.4

Optimisation - 6

IPSec - 7

Security - https://tools.ietf.org/html/rfc4301
Scanning - https://tools.ietf.org/html/rfc4301
Privacy - https://tools.ietf.org/html/rfc7721

Project Proposal

Part II Project Proposal

Software IPv6 Router in Rust

2018-10-11

Oliver Black (olb22) - *Selwyn College*

Overseers - *Jean Bacon / Ross Anderson / Amanda Prorok*
Supervisor - *Andrew Moore*
Director of Studies - *Richard Watts*

Originator - *Oliver Black / Richard Watts*

Introduction & Description of Work

IPv6 is the next generation internet addressing standard, it solves numerous problems with IPv4, such as the shortage of IPv4 addresses. It does more than just increase the number of addresses though, many advanced features not in IPv4 are added with IPv6, full details of the current IPv6 standard can be found in IETF RFCs [1.1]. Mininet [1.2] is a virtual network simulator (supporting IPv6) that was developed at Stanford and until 2016 was used in the 1B Computer Networking Course [1.3] (the year before I attended it).

This project will make use of Mininet to write a software implementation of an IPv6 router in Rust [1.4]. This will begin with gaining an understanding of how Mininet works, then doing research into the IPv6 specification, and finally developing a router and a test bench. Initially that router will implement a minimum amount of IPv6 functionality (as per the IPv6 standards), moving on to more advanced functionality if time permits.

Resource Declaration

No special resources will be required. Work will be undertaken on a personal laptop, with git used for source control, and github for cloud backup. This will enable work to be seamlessly continued on an MCS machine if the laptop is taken out of action.

Starting Point

I took the 2017 1B Computer Networking course, so have a good overview of what a router is meant to do. I have spent around 1 hour fiddling with Mininet, and reading up on the 2016 Computer Networking course, to check what I want to do is feasible. An implementation of an extremely simple router already exists as an optional extension of that course, and I have looked at it briefly [3.1]. I have attended a Rust talk at the CL, and have done my own brief research into the programming language. I have used continuous integration testing methodologies during my summer internship at Solarflare.

Substance & Structure of the Project

The aim of this project is to write a software IPv6 router that complies with the IPv6 RFC standard [4.1]. It'll begin by complying with all the 'must's mentioned in the main IPv6 RFC standard, with other aspects of the standard being extensions, see success criteria for more details.

The first stage of the project will be research and refining requirements, this will be two fold. On a practical level, understanding how the Simple Router [4.2] project works and interfaces with Mininet, also gaining a working knowledge of Rust. On a non-practical level, researching IPv6 and gaining a deeper understanding of the requirements listed in success

criteria. From these requirements a detailed plan will be written, including the router's system architecture.

The next stage of the project will be development and testing. A test bench will be created as the project proceeds, with tests being added for each new requirement that is implemented. The test bench will take the form of scripts that set up a Mininet environment, and then have IPv6 nodes sending packets to each other and the router. All the tests will be run every time a feature is added, the result being a rudimentary form of manual continuous integration.

The development itself will primarily be in Rust [4.3], with bits of C and Python. Rust is a safe modern low level language, and will be an interesting learning experience, the project can always be completed entirely in C if there are insufficient libraries available, or the learning curve is too steep. Interfacing with Mininet at a high level (for the test bench) will be done in Python, as that is the language the API is written in. Exploratory work will be done in C due to preexisting code bases, such as the Simple Router example, being in C.

The final stage will be the verification of the test bench, with additional end-to-end tests being performed. This will be followed by an evaluation to demonstrate the implementation has been successful, and then by the writing of the dissertation.

Success Criteria

To have a software IPv6 router and accompanying test bench running on top of Mininet. The router will comply with all of the core requirements of an IPv6 router, and the test bench will test each of these requirements.

The core requirements are divided into two parts: basic and advanced. The basic requirements are an implementation of a control and data plane in software that can forward packets to the correct unicast addresses, with static address allocation. The advanced requirements are an implementation of ICMPv6 (i.e. proper error handling), and handling anycast and multicast addresses.

Testing is required for all of these, and will meet the success criteria if there is a unit test for each implemented bit of functionality and the relevant requirements listed in the specification, and if the router passes all these tests. This includes tests for edge cases, for example when TTL is 1.

The extension requirements are implementing IPv6 extension headers (both those included in the main IPv6 RFC, and those with their own RFC), implementing SLAAC & DHCPv6 (stateless and stateful), compatibility features with IPv4 addresses, and checking addresses comply with IPv6. Other stretch goals include optimisation, these will be tested through load testing and comparisons with non-optimised versions. Additionally any optional minor features encompassed by the core requirements are extension requirements. A further potential extension would be to pull the software router out of mininet, and get it running on a switch, testing it with real machines.

All of these requirements are based on the relevant RFCs[1.1]. Where applicable, the core requirements only include aspects of the RFC prefaced with ‘must’, whereas the extension requirements include all functionality specified.

Plan of Work

Note: Throughout, any work on extension goals can be replaced with work from a previous 2 week slot, making the plan more flexible and responsive to unexpected changes.

20th October

Start of project - Time since last entry/special events
<> - Work that is completed/stopped on this date
Kick off, begin research - Work that is begun on this date
Milestones: - List of milestones expected by this date

3rd November

2 weeks
Research completed.
Start implementation of core requirements.
Milestones:
List of core and extension requirements mapped out in detail from IPv6 standards and documentation, including system architecture.
Simple Router implementation and interface with Mininet understood.
Basic Rust concepts understood.

17th November

2 weeks
Basic routing framework completed.
Start working on the advanced core requirements.
Milestones:
Router software that can perform basic packet forwarding.
Test bench that can perform basic tests for packet forwarding.

1st December

2 weeks - End of Michaelmas term
Core criteria all met.
End-to-end testing begins, more comprehensive test cases to be added to test bench, small problems fixed as testing continues, big problems documented and listed. Implementation evaluated based on success criteria.
Milestones
Router software that meets all of the core success criteria, with accompanying test bench.

15th December

1 week - 1 week holiday
List of big problems completed.
Big problems fixed in order of severity.
Milestones:
List of remaining identified problems with core functionality.

29th December

2 weeks
All major issues with core functionality resolved.
Begin work on extension goals.
Milestones:
Stable router with comprehensive test bench covering all core functionality.

12th January

2 weeks - Start of Lent Term
Extension work suspended.
Begin evaluation and writing dissertation.
Milestones:
List of implemented extension functionality, with accompanying router software and test benches.

26th January

2 weeks - 1st February: Progress Report Deadline
Evaluation completed.
Continue writing dissertation, write a progress report for presentation.
Milestones:
Evaluation and test report.
(midway) Progress report completed and submitted

9th February

2 weeks
Draft dissertation completed, dissertation work suspended.
Resume work on extension goals.
Milestones:
Completed draft dissertation.

23rd February

2 weeks
Extension work suspended.
Resume work on dissertation - get friends to read.

Milestones:
List of implemented extension functionality, with accompanying router software and test benches.

9th March

2 weeks
<>
Based on feedback received begin work on weaknesses in dissertation.
Milestones:
Improved dissertation based on feedback received.
List of areas of weakness to be worked on.

23rd March

2 weeks - End of Lent Term
Areas of weakness resolved/explained.
Exam revision.
Milestones:
Dissertation submitted to overseers and supervisors for review

6th April

2 weeks
<>
Exam revision.

20th April

2 weeks - Start of Easter Term
<>
Alter dissertation based on comments from from overseers and supervisor
Milestones:
Completed Dissertation

4th May

2 weeks
<>
Exam revision

17th May

2 weeks - 17th May: Dissertation Deadline
<>
Dissertation reread and then submitted.
Milestones:
Submitted dissertation and source code.

Links

1.1: IPv6 https://tools.ietf.org/html/rfc8200
- Addressing: https://tools.ietf.org/html/rfc4291
- ICMPv6: https://tools.ietf.org/html/rfc4443
- DHCPv6: https://tools.ietf.org/html/rfc3315
- SLAAC: https://tools.ietf.org/html/rfc4862
- Authentication Header: https://tools.ietf.org/html/rfc4302
- Encapsulating security payload: https://tools.ietf.org/html/rfc4303

1.2: http://mininet.org/
1.3: https://www.cl.cam.ac.uk/teaching/1617/CompNet/handson/
1.4: https://www.rust-lang.org/en-US/
3.1: https://github.com/mininet/mininet/wiki/Simple-Router
4.1: 1.1
4.2: 3.1
4.3: 1.4