Oliver Black

Software IPv6 Router in Rust

Computer Science Tripos – Part II

Selwyn College

May 16, 2019

Declarations

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 16, 2019

I, Oliver Black of Selwyn College, am content for my dissertation to be made available to the students and staff of the University.

Signed Oliver Black

Date May 16, 2019

Acknowledgements

Many thanks to:

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

Proforma

Candidate Number:	**2340C**
Project Title:	**Software IPv6 Router in Rust**
Tripos:	**Computer Science Tripos**
Examination:	**Part II Dissertation, July 2019**
Word Count (dissertation):	**11092**[1]
Line Count (software):	**1998**[2]
Project Originator:	**The Dissertation Author & Richard Watts**
Project Supervisor:	**Andrew Moore**

Original Aims of the Project

The IPv6 standard[2] contains a large number of complex requirements, making it at times difficult to understand. I aim to design and implement a simple IPv6 router (called Luyou) using Rust[3] that behaves as specified in the IPv6 RFCs. Luyou should implement the minimum functionality required by the relevant standards, and be functional & stable. These goals were quite ambitious, as even the core requirements of the IPv6 standard are quite broad - my project would have to go very smoothly in order to succeed in implementing all of them.

Work Completed

Despite having initial difficulties setting up my test environment (called Luxing) using Mininet[4], 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 (called Luyou). Both Luyou itself, and Luxing, are available for public use[1]. The final router was 575 lines of code with 95% test coverage (by statement), with the core requirements making up 40%-70% of the text of their respective RFCs.

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

[2]This line count was computed by running `find . -name '*.rs/.py/.sh' | xargs wc -l` on the relevant files, breakdown: 995 Python, 575 router Rust, 379 other Rust, 49 Bash shell

Special Difficulties

Cloud Computing took up more of my time in Michaelmas than other Units of Assessment. I had no other special difficulties.

Contents

List of Figures

Chapter 1

Introduction

Slowly but surely the internet is switching to IPv6[2], 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 both 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 bench (Luxing).

The router is called Luyou as 路由器 (Luyouqi) is Chinese for router. Also Luyou was partly written while in China over both vacations, 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. Luxing also includes a small test client and test server, both written in Rust[3], called Luxingke and Luxingfu respectively, as 客 (Ke) is client and 服 (Fu) is server.

Mininet[4] 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 *network namespaces*, allowing processes to share a kernel, but run with separate network contexts. This made it an ideal candidate to build Luyou and Luxing on top of. A simple IPv4 router[5] already exists, and can be run on top of Mininet, it effectively explores how IPv4 works.

My project has three main motivations, I was interested in:

- Writing a router to improve my computer networking skills.
- Providing a clear example of an IPv6 router to illustrate the IPv6 standard.
- Experimenting with Rust to enable me to use it in future projects.

These are explained in more detail in the following paragraphs.

Routers are the backbone of the internet. A common abstraction for working with routers is to separate their functions into a *control plane* that deals with the routing logic and a *forwarding plane* that performs packet routing. There are many open source routing packages[6], but many of them are dual-mode IPv4 & IPv6. This makes it difficult to

isolate and understand how the IPv6 part actually works. Starting from scratch allows a clear IPv6 router to be produced, avoiding the compromises that come with supporting dual IPv4 & IPv6 operation. Implementing Luyou will improve my knowledge of common networking abstractions.

Due to the popularity of the Internet, there have long been not enough IPv4[7] addresses to go around. IPv6 addresses numerous issues in IPv4, including the lack of addresses. It also standardises many common non-standard practices in IPv4. These include the fact that Time To Live in IPv4 was defined partly in terms of seconds left to live, 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. Luyou should provide a clear example of how the decisions made in the IPv6 standard actually work - analogously to the *simple IPv4 router*[5] mentioned above.

Rust[3] is a modern low level programming language that aims to match the performance of C/C++ without sacrificing memory safety, while avoiding garbage collection.

Rust does this through zero-cost high level abstractions such as *ownership* and *lifetimes*. For example, if you pass a reference or a primitive to a function, that function then owns that and everything it owns, the passed in value thus becomes inaccessible to the calling program from the point of the call. It is also possible for functions to 'borrow' values, avoiding transferring ownership, this is similar to pass 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.

Using the IPv6 standard as a framework, combined with some knowledge about the internals of routers, it is possible to develop an IPv6 router that is stable, small, simple, & fast. Such a router could continue to be developed until it was ready to 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 subset 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

First I completed research into three areas: IPv6, Rust, & Mininet. I then began by annotating the IPv6 RFCs[2][8][9] to divide my implementation into a minimal core and optional extensions. Next I designed a framework in which these could be implemented, and developed a test plan to enable effective evaluation of the finished product.

2.1 Starting Point

I had some experience in some of these areas, but this project approaches them 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 a project in Rust before, but was very interested in learning Rust.
- **Network Programming:** I had completed the Part 1B Computer Networking course, so had some theoretical understanding. I had also worked on an assignment related to a large networking project during an internship. However I had never worked on a networking project itself by myself from the ground up before.
- **Testing:** I had written tests for code previously 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 had only brief experience with Python.

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

- **pnet**[10] the packet and interface libraries in pnet allowed packets to be received, and for individual packets to be manipulated.
- **Mininet**[4] allowed virtual network environments on Linux to be setup with IPv4 addresses.

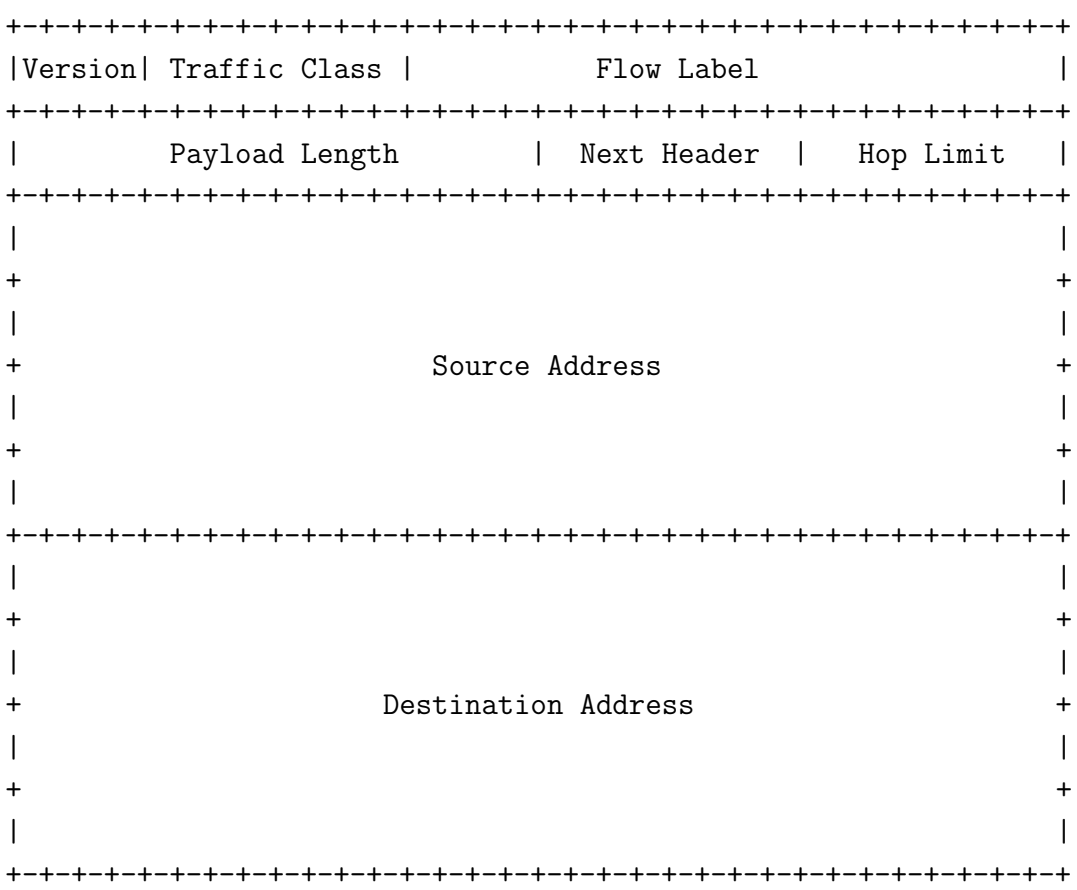

Figure 2.1: IPv6 Header Format[2]

2.2 Research

An analogous project for IPv4 called Simple Router already exists[5] (partially implementing it used to be a recommended extension task for the Part 1B Computer Networking course). It is implemented in C and it runs on top of Mininet. The implementation itself didn't help at all (due to being in C and for IPv4), but by 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 provide some hints for the design of Luxing. I did complete additional research into Mininet, which is detailed under section 2.5 - Test Plan.

Most of my research time was spent reading IPv6 RFCs - Requests For Comment (RFC) are published by the Internet Engineering Task Force and contain technical and organisational notes about the internet[11]. The main RFC[2] specifies how IPv6 packets are constructed and handled. It 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. The RFC also specified the required extension headers and how these are handled. If the router is not the packets destination the router does not need to handle any extension headers.

Another important RFC was the Internet Control Messaging Protocol RFC[9]. This protocol accompanies IPv6 proper, and must be implemented by all IPv6 routers. It

provides the control channel for IPv6 Internet traffic, this includes, among other things, errors about dropped packets being sent back to the source, and *Echo Request/Reply* ("ping") messages being sent and responded to. In order to understand ICMPv6 it was important learn how IPv6 related to the link layer below and the transport layer above.

The IPv6 Addressing RFC[8] contains details on how IPv6 addresses are allocated and how they should behave. It is mainly just a list of address ranges and whether they have any special behaviour.

I also consulted other RFCs related to features that didn't need to be implemented by an IPv6 router[12][13][14]. More details on those can be found under Appendix A.

Finally I spent some time researching and experimenting in Rust[3], 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[10] 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 work out what exactly Luyou 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 advanced being everything else that required by the IPv6 RFCs. This classification the implementation priority of each requirement (with critical requirements coming first), and is not related to the type of functionality each requirement provides.

I used the Basic, Advanced & Extension classification extensively when implementing Luyou to ensure I implemented the most important functionality first. However the following division - which is based on type of functionality - is more appropriate for explaining the process by which I found and classified the requirements. The relationship between the two can be seen in Figure 2.2.

- Addressing
- Packet inspection & forwarding
- Error reporting & ICMPv6[9]

Figure 2.2 gives a list of requirements, below I discuss how I classified each into Basic, Advanced & Extension - with requirement numbers in '()'.

The requirements for addressing can be divided into two parts, the address discovery mechanism (static (1112), DHCPv6[12] (23) or SLAAC[13] (24)) and different address types (Unicast (113), Multicast (122), & Anycast (123)).

```
Key
B : Basic (Core)
A : Advanced (Core)
E : Extension
T : Temporary requirement
XX : Requirement number
RFCXXX : Where the requirement can be found
8200 - IPv6 RFC[2]
4443 - ICMPv6 RFC[9] 4291 - IPv6 Addressing[8] 3315 - DHCPv6[12] 4826 -
SLAAC[13]
```

```
Addressing
├── Discovery
│   ├── Flooding B T 1111
│   ├── Static B RFC8200 1112
│   ├── DHCPv6 B RFC3315 23
│   └── SLAAC E RFC4826 24
└── Types
    ├── Unicast B RFC4291 113
    ├── Anycast E RFC4291 122
    ├── Multicast E RFC4291 123
    └── Scopes E RFC4291 124
```

```
Packets
├── Inspection
│   ├── Payload Length B RFC8200 11211
│   ├── Hop Limit B RFC8200 11212
│   ├── Traffic Class E RFC8200 21
│   ├── Flow Label E RFC8200 21
│   ├── Next Header E RFC8200 21
│   └── Extension Headers E RFC8200 (and other RFCs) 22
└── Forwarding B RFC8200 1112
```

```
Reporting
├── Log Output B 1
└── ICMPv6
    ├── Checksum A RFC4443 1211
    ├── Unknown A RFC4443 1212
    ├── Rate limit A RFC4443 1213
    ├── Packet too big A RFC4443 1214
    ├── Time exceeded A RFC4443 1215
    ├── Echo A RFC4443 1216
    └── Parameter Problem A RFC4443 1217
```

Figure 2.2: Requirements from Appendix A by type of functionality

In order to get the router forwarding packets as quickly as possible I added the requirement of *flooding* addressing. This is not defined in the RFC for IPv6, as it means a router would instead be functioning as a broadcasting switch, sending all incoming packets out on all interfaces.

Static addressing means the relationship between IPv6 addresses and link layer interfaces is defined when the router starts based on a fixed configuration. Static addressing was *needed* by the RFCs and I decided flooding alone wouldn't constitute basic testable IPv6 functionality. So static addressing is a basic core requirement, and supersedes flooding addressing (which is also a basic core requirement).

Although DHCPv6 (23) and SLAAC (24) are both practical and interesting, they aren't *needed* for an RFC router - static addressing is sufficient - so they were classified as extension requirements.

Address types in IPv6 are well defined by the addressing RFC[8], and a router *needs* to deal with all of them. However, in order to test basic functionality only Unicast (113) really needs to be implemented, as Multicast (122) and Anycast (123) 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 local only addresses, as well as a variety of other specific scopes, these are also all advanced core requirements.

Every packet the router receives must have its payload length field checked (11211) to see if it matches the actual length of the payload - and the packet discarded if not. Additionally the hop limit (11212) must be checked: if it is 1 or 0 the packet should be discarded, otherwise it should be decreased by 1. Hop limit reduction must be implemented to routing loops. As a result these are basic core requirements, they are *needed* and without them it is hard to test basic functionality (and the router would barely provide any functionality).

IPv6 also has many extension headers (22), 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 intermediate nodes do not need to process[2]. Since Luyou receives ICMPv6 packets, I need to process IPv6 extension headers, but only to the extent necessary to properly interpret the encapsulated ICMPv6 (as Luyou has no reason to deal with any other packet types). The only headers required for this are the options headers (RFC8200 4.3 & 4.6[2]), the routing header (RFC8200 4.4[2]), and the fragment header (RFC8200 4.5[2]).

ICMPv6 (121) works on top of IPv6 to send informational and error messages between nodes. These messages are destination, packet size (1214), hop limit (1215) & header (1217) error messages, and echo request & reply (1216) informational messages.

ICMPv6 is *needed* for any IPv6 router. However, the basic functionality of Luyou could be verified using log output, so ICMPv6 was not required to test basic functionality. 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 chapter 1 - Introduction that I did not want to Luyou to be run on

actual hardware, as this would add needless complexity when my aim is to explore and illuminate the IPv6 requirements.

For the sake of limiting the core part of the project to something that could be implemented in the time available, I also removed any features that were experimental or not in general use in IPv6 networking. This included cross-layer optimisations (inspecting TCP & UDP packets and making decisions based on their contents).

There were also no performance related requirements, as my aim was not to produce a high performance router.

Here are the requirements for Luyou, now in implementation order, starting with the basic core requirements:

- Send packets to the correct hardware interface in accordance with the static routing rules provided (RFC8200 2[2])
- Handle IPv6 headers in accordance with the standard - Hop limit, etc (RFC8200 3[2])

Advanced core requirements:

- ICMPv6[9]
- Multicast[8]
- Anycast[8]

My extension requirements:

- IPv6 extension headers[2]
- DHCPv6[12]
- SLAAC[13]

See Figure 2.2 for a complete list of requirements, and for more detail (along with associated tests) see Appendix A. Final implementation and test status can be seen in Figure 4.1. Note, these requirements match those described in my Project Proposal.

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 of Luyou itself, control and forwarding plane separation, and protocol layer separation (the design of Luxing - the test bench - is discussed in the next section 2.5 - Test Plan).

The first was the separation of the control and forwarding plane. The control plane deals with addressing and the forwarding plane with link layer interfaces and individual packets. The two communicate through a routing data structure - the routing table. At

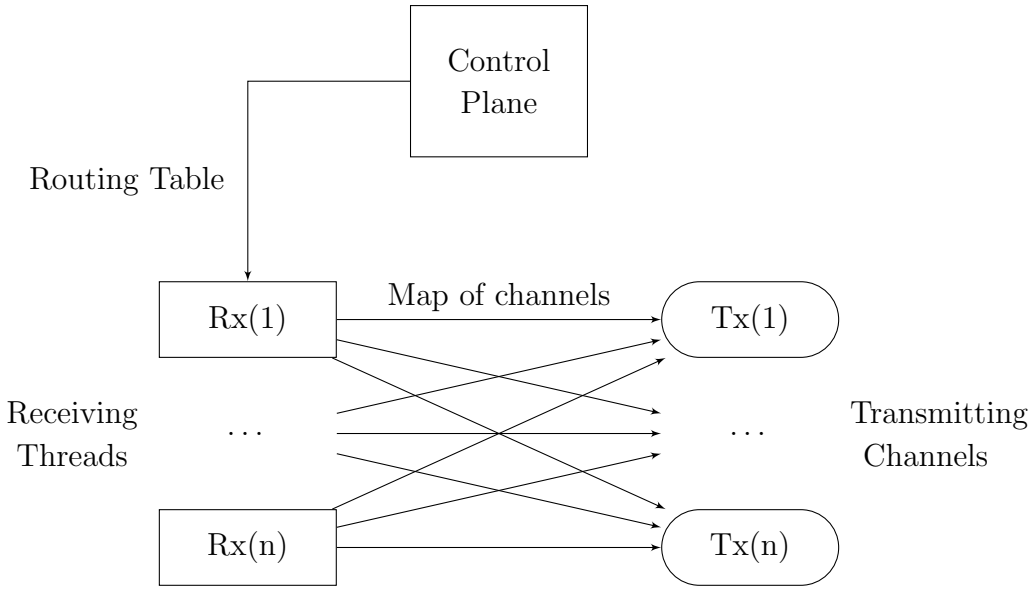

Figure 2.3: Router Design: Control plane with routing table and forwarding plane with forwarding fabric

its most basic level the routing table is a map from IPv6 addresses to pairs of MAC addresses (representing interfaces). However, this map is contained within a larger struct (the routing table) to limit access by the forwarding plane and provide other control related information.

For static addressing, the forwarding plane only requires read-only access to the routing table. However for DHCPv6 or SLAAC the forwarding plane would need to be able to trigger modifications of the routing rules in the control plane.

The forwarding plane itself is made up of a receiving thread for each interface. All of the receiving threads have access to a map of hardware (MAC) addresses to transmitting channels (one for each interface), this is the *forwarding fabric*. This map is given to each receiving thread on creation. Each MAC address is mapped to the transmit channel for that MAC address.

The forwarding fabric is the complete collection of all possible paths packets can take between interfaces. At runtime the specific path to be taken is determined by consulting the routing table. The receiving thread then forwards the packet along that specific path, and it is transmitted by the correct interface.

For an overview see Figure 2.3.

The second idea was designing the layer separation inherent in the TCP/IP stack into Luyou. 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 layer sends the IPv6 packet to the IPv6 layer, which returns the new IPv6 packets, and the hardware addresses to send them to. The IPv6 layer, if necessary, sends the ICMPv6 packet along with the source and destination address to the ICMPv6 layer, which returns the new ICMPv6 packet along with the new source and destination addresses.

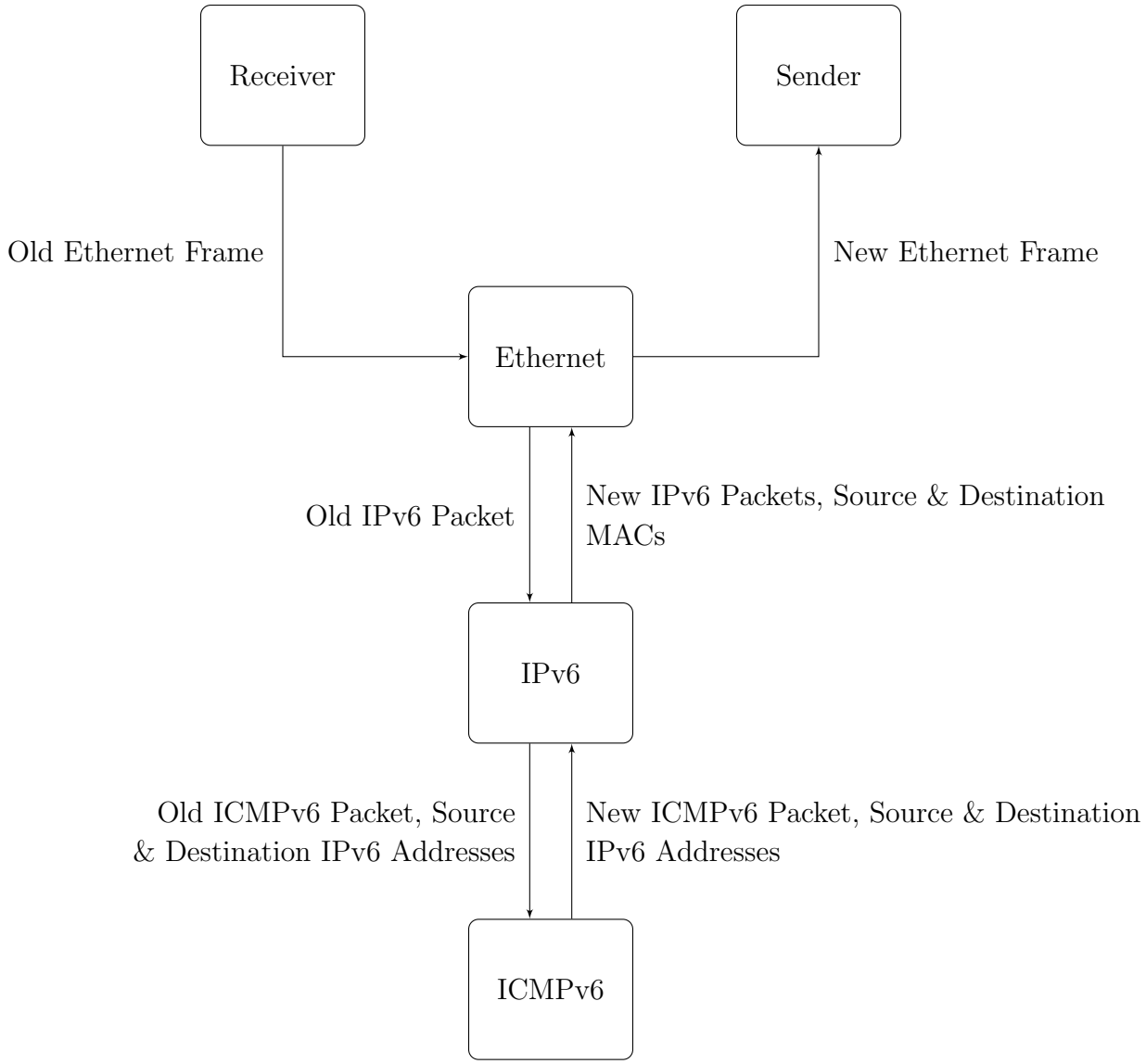

Figure 2.4: Layer Separation

These layers were all implemented within the receiving thread, with each layer being a function, and the Ethernet function sending the packet to the correct transmitting thread (as described in the last paragraph). See Figure 2.4.

Luyou should drop any Ethernet packets that don't contain an IPv6 packet. Additionally most IPv6 packets do not contain an ICMPv6 packet - calling the ICMPv6 function is the exception not the rule.

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

2.5 Test Plan

With requirements and a design of Luyou completed, I needed to plan how to verify that Luyou was functioning as expected. This was divided into two parts, the design of Luxing, and the list of white box tests.

I planned to make Luxing from Mininet and a couple of helper applications (either written in Python or Rust). A couple of helper applications would be capable of sending packets with specific properties, and logging the contents of 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. Although the underlying Linux abstractions used by Mininet supported IPv6, Mininet itself did not allocate IPv6 addresses, enable IPv6 on nodes, or provide a way to remove IPv4 addresses from nodes. I faced either writing my own network emulation environment from scratch, or developing some kind of modification or wrapper for Mininet. I chose to focus on a wrapper that would intercept the Mininet functions I needed to use, and then modify the network environment created by Mininet to work with IPv6. Details of this wrapper can be found in chapter 3 - Implementation.

When formalising the results of my analysis into a list of tests I did two things. First I numbered everything so I wouldn't forget any requirements. Second 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 edge case test. For example, the hop limit tests are:

- `when <incoming packet> then <outgoing packet or "action">`
- `when hop_limit = x then hop_limit=x-1`
- `when hop_limit = 0 then "Packet Dropped"`
- `when hop_limit = 1 && destination != router then "Packet Dropped"`
- `when hop_limit = 1 && destination == router then "Passed up layer"`

To avoid all the tests being dependent on an almost complete implementation of Luyou I implemented my tests progressively so that tests could be run at the point of implementation of a feature. 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 decrement depends on a correct implementation of ICMPv6, as well as the correct implementation of hop limit decrement, so I instead only required a log message to verify success. My numbering system matched my planned order of implementation so just reading off all the previous tests allowed me to know which features the next test was allowed to rely on.

Alongside my requirement based white-box testing I intended to test some more ambitious end-to-end test cases, these were:

- `ping6` - pinging in IPv6 (ICMPv6) between all pairs of nodes.
- Web Server - run a web server on the right node and try to request a web page from the left.

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 this chapter 2 - Preparation and in the next chapter 3 - 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. It would put peoples personal data at risk if this is not done, incurring significant liabilities.

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. My code is released under the GNU GPL[15], which is compatible with the Mininet, Rust & pnet 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 iterative process. There were two main parts to implement, Luyou (the router itself) and Luxing (the test bench).

Repository Overview

See Figure 3.1 for the repository overview, the project is made up of a Python application (the main body of 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 [19]*
│   └── __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*
```

Figure 3.1: Repository Overview

3.1 Router

Luyou was implemented by creating the skeleton framework as set out in 2.4-Design. Then a series of prototypes were completed, each adding more functionality (this is explained in more detail in 3.3 - Analysis). Version control was performed using git[16], with backups committed to an *exploratory* branch and merges into the *master* branch whenever a specific requirement had been completely implemented and tested. See Figure 3.2 for an overview of data structures and threads in the implementation.

I implemented the following prototypes in turn, each covers a larger subset of my requirements, with the final prototype covering all of my basic core requirements and ICMPv6:

- **Flooding:** Packets forwarded on all interfaces - this confirmed that the pnet and the skeleton framework work as expected.
- **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.

Below I describe the implementation of each individual aspect of the router: data structures, threads & channels, and layer separation. Each aspect is followed by related difficulties, it is relatively self explanatory at which prototype each difficulty was encountered.

First I designed the data structures needed. I needed a routing table, a map from IPv6 addresses to pairs of MAC addresses. Each pair represents an interface, and the forwarding plane requires both to fill in both the source and the destination fields of an outgoing Ethernet packet. This map would need to support a one to many relationship when I implemented Multicast, but initially a one to one map would be sufficient. I also needed the forwarding fabric, a map from MAC addresses to transmitting interfaces, used by the forwarding plane to direct packets to the correct interface.

The resulting data structures had the following types:

- Routing Table: `HashMap<Ipv6Addr, (MacAddr, MacAddr)>`, if Multicast implemented: `HashMap<Ipv6Addr, HashSet<(MacAddr, MacAddr)>>`
- Forwarding Fabric: `HashMap<MacAddr, Sender<Box[u8]>>` (was initially: `HashMap<MacAddr, DataLinkSender>`)

I decided the routing table itself should be private to the control plane, an be a member of a `Routing` struct. The routing struct exposed `get_route()` - which either returned from the routing table or the default route(`(Ipv6Addr) -> (MacAddr, MacAddr)`), and `get_router_address()` (`() -> Ipv6Addr`) - which returned the IPv6 address that was statically allocated to the router. In order to share the routing struct between layers I put it inside an `Arc` (Atomically Reference Counter[17]), a thread safe reference counting

```
Main Thread
├── Control Thread - starts
└── Forwarding Threads - starts a pair per interface
    ├── Receiving Thread
    └── Transmitting Thread

Control Thread
└── Routing struct : Routing
    ├── Routing Table : HashMap<Ipv6Addr, (MacAddr, MacAddr)>
    ├── Get Route : (Ipv6Addr) -> (MacAddr, MacAddr)
    ├── Get Router Address : () -> Ipv6Addr
    └── Get MTU : MacAddr -> Ipv6Addr


Receiving Thread
├── Receiving Channel : DataLinkReceiver
├── Transmitting Channels (forwarding fabric) :
        HashMap<MacAddr, Sender<EthernetPacket>>
├── Routing struct (control thread) : Arc<Routing>
├── Ethernet Function :
        (EthernetPacket, &mut MutableEthernetPacket, Arc<Routing>)
         -> (Result<(MacAddr), String>)
├── IPv6 Function :
        (IPv6Packet, &mut MutableIPv6Packet, Arc<Routing>)
         -> (Result<(MacAddr, MacAddr), String>)
└── ICMPv6 Function :
        ((Icmpv6Type, u32), IPv6Packet, &mut MutableIcmpv6Packet,
            Arc<Routing>)
         -> (Result<(MacAddr, MacAddr), (Ipv6Addr, Ipv6Addr)>)

Transmitting Thread
├── Receiving Channel : Receiver<Box[u8]>
└── Transmitting Channel : DataLinkSender

Key:
& - Borrowing, value remains valid on function return
&mut - Mutable borrowing, can be mutated by the function
(,) - Tuple
<> - Generic
Result<A,B> - Either Ok(A) or Err(B)
```

Figure 3.2: Implementation Overview: Data structures and threads

reference (this was enforced by Rust's type system, Rust needs to know when a struct can be deallocated to prevent potential memory leaks). For the forwarding threads to trigger modifications in the routing table (as would be required for DHCPv6 or SLAAC) some form of write lock is required, but swapping the `Arc` for a `RwLock`[18] should be sufficient.

Finally I needed a representation for the different types of packets I would be dealing with, pnet[10] includes structures like `MutableEthernetPacket` & `IPv6Packet` that wrap around a slice of the byte buffer received from the network interface and allow the packet fields to be easily accessed and manipulated with methods like `get_version : () -> u4`.

Then I implemented the threads that make up the forwarding fabric. A network interface in pnet is made up of a channel - a sender receiver pair (`(DataLinkReceiver, DataLinkSender`). I set up a skeleton receiving thread for each receiver, and created a map of MAC addresses to transmitters (`HashMap<MacAddr, DataLinkSender>`). I then shared this map with the receiving threads. The map didn't change so it could be safely copied across threads. See Figure 3.3 for the code excerpt that creates sending thread safe channels and starts the sending threads, the receiving channels are created in a similar way.

Unfortunately the channel type used by pnet does not support being shared 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 (`HashMap<MacAddr, Sender<EthernetPacket>>`).

Additionally the packet representations used by pnet can't be sent over a channel, as they don't implement the `Clone` trait that would have allowed them to be copied and sent (a trait is similar to an interface in a language like Java). However, the underlying byte buffer does implement the `Clone` trait, so the thread safe channels are of type `Box<[u8]>` (the `Box` is basically a reference) not of type `EthernetPacket`.

Next I implemented the layer abstraction explained in 2.4-Design. This involved implementing three functions. Each function takes a new and an old packet, updating the new packet appropriately.

The functions don't return the new packet, they mutably borrow it (`&mut`), this is analogous to passing by reference. The functions return a `Result<A,B>`, in the event of an error the return value is `Err(b)`, and in the case of success is `Ok(a)` (where '`a`' & '`b`' are of type '`A`' & '`B`' respectively). Errors are returned in cases where a packet should be dropped, when an error is returned no packet is sent.

Each function potentially calls the 'higher layer' function in its body, passing the relevant part of old and new packets it itself received, and using the resultant new packet as the payload for its new packet - see Figure 3.4 for an example. The functions were as follows, each one potentially calls its successor in its body:

- **Ethernet:** Passes the routing struct up. Returns the new destination MAC address for look up in the forwarding fabric.

```
(EthernetPacket, &mut MutableEthernetPacket, Arc<Routing>)
        -> (Result<(MacAddr), String>)
```

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

fn sender_loop(mut sender: Box<DataLinkSender>,
                           receiver: Receiver<Box<[u8]>>) {
    loop {
        //get a packet from the thread safe receiver
        let packet = receiver.recv().unwrap();
        //send it on the DataLinkSender
        sender.send_to(&packet,None);
    }
}
```

Figure 3.3: Code excerpt: Starting a transmitting thread, returning a thread safe pipe

- **IPv6:** Passes the routing struct up (if the encapsulated packet is ICMPv6). Returns the pair of MAC addresses (new source and new destination), to put in the new Ethernet packet, and to be passed down again.

```
(IPv6Packet, &mut MutableIPv6Packet, Arc<Routing>)
        -> (Result<(MacAddr, MacAddr), String>)
```

- **ICMPv6:** Receives the IPv6Packet as it needs the source and destination address (they form the ICMPv6 pseudo-header), also receives the type of ICMPv6 message to generate (with an argument - usually the ICMPv6 code). Returns a pair of IPv6 addresses and MAC addresses (both new source and new destination) to be used in the new IPv6 packet and by the IPv6 function.

```
((Icmpv6Type, u32), IPv6Packet, &mut MutableIcmpv6Packet, Arc<Routing>)
         -> (Result<(MacAddr, MacAddr), (Ipv6Addr, Ipv6Addr)>)
```

Initially I planned to allocate a mutable buffer in the main loop of the receiver thread, then pass this as a `MutableEtherentPacket` to the Ethernet function, the Ethernet Function would then pass the payload of the `MutableEthernetPacket` as a `MutableIpv6Packet` to the IPv6 function. 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. The solution is to create a fresh buffer for the IPv6 packet and then copy the payload across. This can be better seen in Figure 3.4.

I don't agree with this design choice in pnet; it results in extra copies. This could have been avoided if creating a packet from a buffer in pnet returned 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. However this is not the case.

The ICMPv6 method returns a pair of MAC addresses because it requires a MAC address to determine the MTU of a outgoing link (which it uses to limit the size of *EchoReply* messages). If the MAC address of the destination IP changes between the MTU check in the ICMPv6 function and the get route call in the IPv6 function the packet could be unnecessarily trimmed, or worse, be sent down a link with too low an MTU. This is partially resolved by performing the call to get route in the ICMPv6 method, so that both calls occur in the same function. However, to fully resolve this issue the calls would need to happen atomically. Under static addressing such an inconsistency cannot occur, so the atomicity has not been implemented.

The static routing rules were read from a text file. These consisted of the default route, the router address, then a list of routing rules. Each routing rule either related an IPv6 address to an interface or an MTU with a MAC address. See Figure 3.5 for details and an example of the file format.

```
//create old IPv6 packet from Etherent payload
let old_ipv6_packet = match Ipv6Packet::new(
                                old_packet.payload()) {
        Some(p) => p,
        None => return Err(format!("Invalid Packet")),
};

//allocate new mutable IPv6 Packet
let mut buffer = vec![0;new_packet.payload().len()];
let mut new_ipv6_packet =
    MutableIpv6Packet::new(&mut buffer).unwrap();

//set the length of the new IPv6 packet to match
// the new Ethernet packet
new_ipv6_packet.set_payload_length(
    (new_packet.payload().len()-40) as u16);

//call IPv6 layer function and match on result
// passing on errors
let (source, destination) =
    match transform_ipv6_packet(
        old_ipv6_packet, &mut new_ipv6_packet, routing) {
    Ok(p) => p,
    Err(e) => return Err(e),
};

//update new mutable Ethernet packet
new_packet.set_destination(destination);
new_packet.set_source(source);
new_packet.set_ethertype(Ipv6);
new_packet.set_payload(new_ipv6_packet.packet());
```

Figure 3.4: Code excerpt: Ethernet layer calling IPv6 layer

```
DEFINITION:
<Default Route IPv6 Address>
<Router IPv6 Address>
[Routing Rules]
...

Routing Rule:
<IPv6 Address>@<Source MAC Address>,<Destination MAC Address>
OR
mtu<MTU>@<Destination MAC Address>

EXAMPLE:
fc00::
fc00::3
fc00::@00:00:00:00:03:00,ff:00:00:00:00:00
fc00::1@00:00:00:00:03:01,00:00:00:00:01:00
fc00::2@00:00:00:00:03:02,00:00:00:00:02:00
ff02::1:ff00:0@00:00:00:00:03:00,ff:00:00:00:00:00
ff02::1:ff00:1@00:00:00:00:03:01,00:00:00:00:01:00
ff02::1:ff00:2@00:00:00:00:03:02,00:00:00:00:02:00
mtu1300@00:00:00:00:02:00
```

Figure 3.5: Static addressing configuration file format

3.2 Test Bench

As outlined in chapter 2 - Preparation a key part of this project is the test bench (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. In the end Luxing took the form of a test framework (`test_framework.py`) which wrapped around Mininet adding rudimentary IPv6 support and the tests themselves which were separate Python files that made use of the framework. The test framework was not part of my initial plan for the project, but I successfully implemented an IPv6 wrapper for Mininet.

Initially I thought Mininet[4] 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 all interfaces of network nodes (even though this example[19] had led me to believe it did when writing my project proposal). As described in Test Plan I chose to write a wrapper for Mininet that added the IPv6 functionality I required.

The wrapper is a python class (`test_framework.py`) which extends the main `Mininet` python class (the class used to initialise and manipulate networks in Mininet). It overrides some public methods, such as `addNode` (used to add a node to the network), and adds

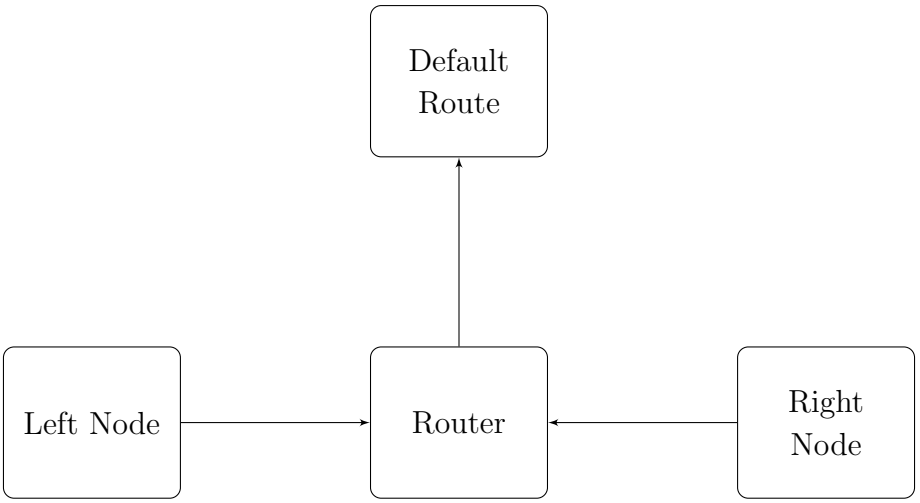

Figure 3.6: Test Network Topology: arrows in directions of default router

some new public methods like `addRouter`. It also includes several private methods that implement functionality like allocating IPv6 addresses and adding default routes.

The wrapper aims to support the necessary functionality to run tests, it does not aim to be a complete IPv6 rewrite of Mininet. Also, not as much design went into the wrapper as went into Luyou. This results in a wrapper that 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, see Figure 3.6. Luyou itself is one node, and is in the centre of the network. The leaves of the network are made up of two 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. Running tests on this single simple network topology prevents scaling and interactions with other routers from being tested. However, with static addressing such tests aren't particularly interesting, as interactions with other routers are predefined in the static routing file, with the other router appearing to Luyou as a node with more addresses.

Luxing is run through a simple shell script - `test.sh [Test/Requirement number]` - which selects the correct Python script to run based on the requirement number. Each test python script (e.g. `test_example.py`) creates an instance of the wrapper, sets up the network topology, tells the wrapper to start Luyou, then runs the specified test. The wrapper starts Luyou by starting the compiled Rust binary in the network context of the node, and passing the path to the static address configuration file as an argument (Luxing does this by running `luyou static_routing.txt`).

Pass or failure of the test is verified by the user; the test will log the expected behaviour (e.g. "3 packets dropped, one received with the 4th octet being 7"), and the user then reads the previous log messages to confirm this is the case. See Figure 3.7 for an example.

The first issue I faced was getting Luxing to allocate IPv6 addresses to nodes. This was eventually resolved by adding a `__add_ipv6_address` method which runs

```
Luxingke - Client starting
Packet(s) sent
Packet dropped, hop limit, and imcpv6 response sent
Packet dropped, hop limit, and imcpv6 response sent
fc00::1:[96, 0, 0, 0, 0, 0, 59, 9, 252, 0, ...]
fc00::1:[96, 0, 0, 0, 0, 0, 59, 1, 252, 0, ...]
Test completed: if two packets dropped due to hop limit then two with 9
    and 1 as 8th octect, then success
```

Figure 3.7: Test Example: Log output and success state

```
def __add_default_route(self, node, gateway_address):
        while "tentative" in node.cmd("ip -6 addr"):
            sleep(0.1)
        node.cmd("ip -6 route add default via " + gateway_address
                 +" src "+self.address(node.name))
```

Figure 3.8: Waiting for IPv6 addresses to be up before updating default route

`ifconfig [interface-name] inet6 add [address]` and then calling it every time a node was added. Luxing then stores the node name and IPv6 address in a two way map, allowing it to easily find the addresses later. Luxing 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.

Mininet links nodes with *Veth* (Virtual Ethernet) links. However, this does not allow you to choose MAC addresses for each node, which is necessary to construct the static routing configuration file. As with IPv6 addresses I just specified MAC addresses on node creation.

Another issue was adding the default route to all nodes, I discovered this when implementing the flooding Ethernet repeater version of Luyou. I'm unsure exactly why, but, without the default route set to point to the router `ping6` would not work. This is easily achievable using the `ip -6 route add default via [default-address] src [node-address]` command, but this seem 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, this can be seen in Figure 3.8. The default routes can also be seen in Figure 3.6.

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 3.1-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 (generated by intercepting calls to `addLink`). Luyou's address and the default gateway's address also need to be included in the routing file, which was gathered when they were added using `addRouter` and `addDefault`.

After implementing static addressing 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 *Neighbour Discovery Protocol*[14]. Luxing resolves this by adding these 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.

The hardest part of implementing Luxing was working out which linux commands and in which configurations was needed to get Mininet working with IPv6. For example, Luxing barely work at all until IPv6 addresses were custom allocated to every interface and all the default routes set. I had no way of knowing in advance that the combination of these two commands was what was required. So even though almost all of the above fixes seem obvious in hindsight, they took a lot of careful trial and error to discover.

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. I successfully applied theoretical knowledge from the Part 1A Software & Security Engineering course.

Overall I began by creating a complete list of requirements, then a complete design (as described in chapter 2 - Preparation), 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 of a problem.

However, I didn't strictly 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, Static Routing, IPv6 Validation, & ICMPv6 - see section 3.1 - Router for more details).

For each of these stages I first tweaked and 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, making 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 stable 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. My goal was to pass my tests, so verifying I met my goals was also quite easy. However verifying my tests matched up to the IPv6 requirements was more complex. Full details on how to build and run my project can be found in the `README.md` file in the root directory of my project.

4.1 Tests

As I have already mentioned, I wrote 9 individual requirement 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 do. The tests were defined alongside the requirements, and matched the requirements precisely. So, as long as the requirements were 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 avoiding the case were Luyou passes the specific test, but does not satisfy 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 verifies that it passes 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).

To conclude, apart from the issues explained in the next section, my tests verify that I met my goals. My goal was 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 successfully. Overall I implemented 11 tests (9 of which were requirements tests), with each sending and receiving between 1-5 packets.

- example: Pings between all nodes.
- 1112: Pings from left and right nodes.
- 11211: Tests IPv6 payload_length, sends two packets.
- 11212: Tests IPv6 hop_limit, sends 4 packets, see Figure 3.6.
- 1211: Tests ICMPv6 checksum, sends two packets.
- 1212: Tests unknown ICMPv6 type, sends one packet.
- 1214: Tests packet too big ICMPv6 response, sends one packet.
- 1215: Tests hop_limit ICMPv6 response, sends one packet
- 1216: Tests Echo Request/Reply, sends one packet
- 1217: Tests parameter problem, sends one packet
- playground: Runs a web server on right node, requests page from left node

Most importantly for me however, implementing these tests taught me a lot, and I certainly feel better prepared when I have to test something extensively in 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. I spent around 30%-60% of my implementation time on Luxing, which was far more than I had anticipated. The time spent on Luxing took away from time that could be spent on Luyou, so I didn't implement as many requirements as I would have liked to.

As discussed in chapter 2 - Preparation when I wrote my project proposal I believed Mininet supported IPv6 with minimal or no changes[19]. 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 interacting (interface configuration, router configuration, Python configuration, environment variables configuration), without already having deep knowledge of all the tools it took a while to work out what was wrong.

Personally I wanted to create a test bench that was easy to use (this was inspired by witnessing first hand during one of my internships the issues a hard to use test bench can cause), unfortunately time constraints combined with having to write far more testing framework code than I would have liked to made this unattainable. I did separate out framework code from specific test code to avoid code duplication, presenting a good base on which to build a stable easy to use test framework, but Luxing isn't quite there yet. But Luxing is stable enough to run Luyou effectively.

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 has not been implemented because it 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 and having no firewall. For a complete list of requirements implemented and tested, please see Figure 4.1.

Luyou also doesn't implement the following correctly:

- Options headers in Echo Request packets addressed to the router
- Fragmentation of Echo Request packets addressed to the router

4.3 Ambitious Case

Alongside the tests in Luxing I also attempted testing some more ambitious cases (as described in chapter 2 - Preparation. 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 only the line that decrements the hop limit removed. Only one octet of the header was changed, I checked it was the correct one by outputting the packets before and after the change at Luyou. This octet was not included in the ICMPv6 checksum either (it takes the ICMPv6 packet and the destination and source IPv6 addresses). This remains unsolved, but the final version of the router, with hop limit decrement turned off, does allow all nodes to ping each other.

Identifier	Requirement	Defined	Implemented	Tests Pass
1111	Routing - Flood	TRUE	FALSE	TRUE
1112	Routing - Static	TRUE	TRUE	TRUE
11211	Payload Length - IPv6	TRUE	TRUE	TRUE
11212	Hop Limit - IPv6	TRUE	TRUE	TRUE
1131	Internal Structure - IPv6	TRUE	TRUE	FALSE
1211	Checksum - ICMPv6	TRUE	TRUE	TRUE
1212	Unknown Type - ICMPv6	TRUE	TRUE	TRUE
1213	Rate limit - ICMPv6	TRUE	FALSE	
1214	Packet too big - ICMPv6	TRUE	TRUE	TRUE
1215	Time exceeded - ICMPv6	TRUE	TRUE	TRUE
1216	Echo reply - ICMPv6	TRUE	TRUE	TRUE
1217	Parameter Problem - ICMPv6	TRUE	TRUE	TRUE
1218	Uniquely Identify - ICMPv6	TRUE	FALSE	
1221	Multicast sending	TRUE		
12221	Avoid reserved - Multicast	TRUE		
12222	Implement reserved - Multicast	TRUE		
1223	All routers multicast	TRUE		
1224	Never send ICMPv6 multicast	TRUE		
1225	Not source field - Multicast	TRUE		
1231	Anycast	TRUE		
1232	Subnet router anycast	TRUE		
124	Address scopes			
21	Optional IPv6 requirements			
22	Extension IPv6 Headers			
23	DHCPv6			
24	SLAAC			
25	TCP & UDP			
26	Optimisations			
27	IPSec			

Figure 4.1: Requirements by implementation and test status

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. I had a variety of hypotheses as to why this could be the case, but I didn't have time to test them.

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 575 lines, which is really a testament to how concise Rust makes error handling and parsing code.

In terms of code coverage, the tests included in Luxingfu cover approximately 95% of Luyou's code statements (with the remaining 5% being error handling code for obscure and unexpected errors).

Finally, the requirements based off of the aspects of the RFCs that *need* to be implemented end up covering around 40%-70% of the body of the relevant RFCs (my basic requirements). I actually implemented around 20%-50% of the body of the relevant RFCs.

4.5 Goals

My goal was to have a Software IPv6 Router and test bench running on top of Mininet, with the router complying with all of the core requirements of an IPv6 router, each verified by the test bench. As described under issues I didn't implement all of core requirements, however I did produce a verifying set of tests for every requirement I did meet. Overall, although I didn't fully meet my formal goals, I met a good proportion of them despite spending a considerable amount of time being held back by Mininet not working as well as expected.

In terms of my personal goals I am extremely happy with everything I learnt from this project. I learnt a significant amount about coding in Rust, and now have a good understanding of several of the abstract constructs Rust makes use of. I greatly improved my knowledge of Python and of Linux networking through developing a comprehensive test bench for the first time.

Chapter 5

Conclusion

As set out in my introduction, my aim was to produce a stable, small, simple & fast IPv6 router that could act as a practical example of what the IPv6 standard set out in various RFCs means in practice. Overall I feel I achieved this, I met the most important of my core requirements.

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, as 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 (for example, where the wrong memory location is unexpectedly accessed).

If I attempted a similar project again I would make sure that the existing software I intend to rely on actually does what I think it does. This can be done by producing a minimal prototype very early on in development. Either the software is shown to do all that you require it to do, or you work out where it doesn't work, allowing you to plan on how to fill in the gaps through development work. This would have prevented my project being held up by test bench development, and enabled me to complete all of my core requirements.

Another thing I would have done differently is increasing the amount of realistic end to end testing. My end to end tests that verify individual features were good from the standpoint of testing my requirements, but they didn't provide a comparison with real world use. Had I started testing a web server and client from much earlier I may have been able to get it working, giving me an easier to understand demonstration of success.

That said I am very happy with what I achieved in my project. I have certainly learnt a lot about professionalism and software engineering, and believe it will set me up well for the future.

Bibliography

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

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

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

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

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

[6] DD-WRT vs. Tomato vs. OpenWrt: Which Router Firmware Is the Best?, https://www.maketecheasier.com/dd-wrt-vs-tomato-vs-openwrt-router-firmware/

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

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

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

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

[11] IETF RFCs, what is an RFC, https://www.ietf.org/standards/rfcs/

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

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

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

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

[16] Git, version control system, https://git-scm.com/

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

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

[19] IPv6 working in Mininet, http://csie.nqu.edu.tw/smallko/sdn/ipv6_test.htm

Appendix A

Requirements

Requirements

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.

Progress on requirements here

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

Scoped Addresses - 4

Extension - 2

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

2340C (crsid) - *Selwyn College*

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

Originator - *2340C / 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