Jake Hillion

UNIVERSITY OF CAMBRIDGE

A Multi-Path Bidirectional Layer 3 Proxy

Department of Computer Science
University of Cambridge

This dissertation is submitted for the degree of
Bachelor of Arts

Queens' College

May 2021

Declaration

I, Jake Hillion of Queens' 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. I am content for my dissertation to be made available to the students and staff of the University.

Jake Hillion
May 2021

Proforma

Candidate Number:	2373A
Project Title:	A Multi-Path Bidirectional Layer 3 Proxy
Examination:	Computer Science Tripos - Part II, 2021
Word Count:	11894
Line Count:	3564[1]
Project Originator:	The dissertation author
Supervisor:	Michael Dodson

Original Aims of the Project

This project aimed to produce a multipath proxy, which combines multiple, heterogeneous connections and utilises congestion control to dynamically monitor and adapt to changing link conditions. Using congestion control to dynamically monitor link capacity allows a wider variety of links to be combined, such as those on shared mediums or impacted by environmental factors. The core project aimed to implement such a proxy with links over TCP, with extensions relating to improved performance. The stretch goals of the project were to implement the proxy using other transport mechanisms.

Work Completed

The project fulfilled its core success criteria and most of its extended criteria. The proxy supports transport over UDP, which provides increased performance and flexibility over TCP. The performance gains of the proxy over standard connections is tangible, demonstrating the proxy is effective at providing increased bandwidth in many cases. The proxy solution also improves the resilience of the Internet connection, such that if any one connection remains up, all carried flows are maintained.

Special Difficulties

None.

[1]Gathered using `cat **/*.go | wc -l`

Table of contents

Chapter 1

Introduction

The advertised broadband download speed of most UK residences lies between 30Mbps and 100Mbps (Ofcom, 2020), which is often the highest available speed. However, in most cases, more of these connections can be installed at a price linear in the number of connections. More generally, a wider variety of Internet connections for fixed locations are becoming available with time. These include: DSL, Fibre To The Premises, 4G, 5G, Wireless ISPs such as LARIAT[1] and Low Earth Orbit ISPs such as Starlink.[2]

Though multiple low bandwidth, low cost connections may be accessible, a mechanism to combine multiple connections to present a single high speed, highly available connection to a user is unavailable. This work focuses on providing such a mechanism, taking multiple, distinct connections and providing a single, aggregate connection via a proxy.

Using a proxy to combine connections provides three significant benefits: immediate failover of a single flow, exceeding the bandwidth of each individual connection with a single flow, and balancing of inbound connections. For failover, this means that a flow between a user of this proxy and an external user, such as a SIP call, is maintained when one Internet connection is lost. Exceeding the bandwidth of a single connection means that an application which utilises a single flow can take advantage of higher bandwidth than is available over a single connection. This is useful in cases such as a CCTV system, where viewing a live stream from a camera remotely is possible in a higher resolution with the increased bandwidth available. Finally, although methods such as load balancing routers can balance outgoing flows effectively in many cases, inbound flows cannot be handled so simply. Balancing inbound flows involves complex solutions, which rely on client support. The proxy presented here returns control to the network architect, and hides the complexity from the client and server on either side of the proxy, providing a practical mechanism for obtaining all three benefits.

1.1 Existing Work

Three pieces of existing work that will be examined for their usefulness are MultiPath TCP (MPTCP), Wireguard, and Cloudflare. MPTCP is an effort to expand TCP (Transmission Control Protocol) connections to multiple paths, and is implemented at the kernel layer such that applications which already use TCP can immediately take advantage of the multipath benefits. Wireguard is a state of the art Virtual Private Network (VPN), providing an excellent example for transmitting packets securely over the Internet. Finally, Cloudflare shows examples of how a high bandwidth network can be used to supplement multiple smaller networks, but in a different context to this project. This section focuses

[1]http://lariat.net
[2]https://starlink.com

on how these examples do not satisfy the aims of this project, and how they provide useful initial steps and considerations for this project.

1.1.1 MultiPath TCP (MPTCP)

MPTCP (Handley et al., 2020) is an extension to the regular Transmission Control Protocol, allowing for the creation of subflows. MPTCP was designed with two purposes: increasing resiliency and throughput for multi-homed mobile devices, and providing multi-homed servers with better control over balancing flows between their interfaces. Initially, MPTCP seems like a solution to the aims of this project. However, it suffers for three reasons: the rise of User Datagram Protocol (UDP)-based protocols, device knowledge of interfaces, and legacy devices.

Although many UDP-based protocols have been around for a long time, using UDP-based protocols in applications to replace TCP-based protocols is a newer effort. An example of an older UDP-based protocol is SIP (Schooler et al., 2002), still widely used for VoIP, which would benefit particularly from increased resilience to single Internet connection outages. For a more recent UDP-based protocol intended to replace a TCP-based protocol, HTTP/3 (Bishop, 2021), also known as HTTP-over-QUIC, is one of the largest. HTTP/3 is enabled by default in Google Chrome (Govindan, 2020) and its derivatives, soon to be enabled by default in Mozilla Firefox (Damjanovic, 2021), and available behind an experimental flag in Apple's Safari (Kinnear, 2020). Previously, HTTP requests have been sent over TCP connections, but HTTP/3 switches this to a UDP-based protocol, reducing the benefit of MPTCP.

Secondly, devices using MPTCP must have knowledge of their network infrastructure. Consider the example of a phone with a WiFi and 4G interface reaching out to a voice assistant. The phone in this case can utilise MPTCP, as it has knowledge of both Internet connections. However, consider instead a tablet with only a WiFi interface, but behind a router with two Wide Area Network (WAN) interfaces using Network Address Translation (NAT). In this case, the tablet only sees one connection to the Internet, but could take advantage of two. This problem is difficult to solve at the client level, suggesting that solving the problem of combining multiple Internet connections is better suited to network infrastructure.

Finally, it is important to remember legacy devices. Often, these legacy devices will benefit the most from resilience improvements, and they are the least likely to receive updates to new networking technologies such as MPTCP. Although MPTCP can still provide a significant balancing benefit to the servers to which legacy devices connect, the legacy devices see little benefit from the availability of multiple connections. In contrast, providing an infrastructure-level solution, such as the proxy presented here, benefits all devices behind it equally, regardless of their legacy status.

1.1.2 Wireguard

Wireguard (Donenfeld, 2017) is a state of the art VPN solution. Though Wireguard does not serve to combine multiple network connections, it is widely considered an excellent method of transmitting packets securely via the Internet, demonstrated by its inclusion in the Linux kernel (Torvalds, 2020), use by commercial providers of overlay networks (Pennarun, 2020), a security audit (Donenfeld, 2020), and ongoing efforts for formal verification (Donenfeld and Milner, Dowling and Paterson, 2018).

For each Layer 3 packet that Wireguard transports, it generates and sends a single UDP datagram. This is a pattern that will be followed in the UDP implementation of my software. These UDP packets present many of the same challenges as will occur in my software, such as a vulnerability to replay attacks, so the Wireguard implementation overcoming these challenges will be considered throughout. Finally, Wireguard provides an implementation in Go, which will be a useful reference for the Layer 3 networking in Go used in my project.

1.1.3 Cloudflare

Cloudflare uses a global network of servers to provide a variety of infrastructure products, mostly pertaining to websites and security (Cloudflare). Two of the products offered by Cloudflare are of particular interest to this project: load balancing and magic WAN.

Cloudflare provides the option to proxy HTTPS traffic via their global network of servers to your origin server. This layer 7 (application layer) proxy operates on the level of HTTP requests themselves, and takes advantage of its knowledge of connections to load balance between origin servers. Cloudflare can use knowledge of origin server responsiveness to alter the load balancing. This is a similar use case to my proxy, where items (HTTP requests / IP packets) hit one high-bandwidth server (one of Cloudflare's edge servers / the remote proxy), and this server decides the path through which to proxy the item (a chosen origin server / a connection to the local proxy).

Unlike Cloudflare load balancing, the proxy presented in this work operates on layer 3. Cloudflare receives a stream of HTTPS requests and forwards each to a chosen origin server, while my remote proxy receives a stream of IP packets and forwards them via a chosen path to my local proxy. Though these achieve different goals, Cloudflare load balancing provides an example of using a high-bandwidth edge server to manage balancing between multiple low-bandwidth endpoints.

Cloudflare Magic WAN provides a fully software-defined WAN over their global network. That is, their anycast infrastructure will accept traffic to your network at any edge server in their global infrastructure before forwarding it to you. This supports DDoS mitigation and firewalling at a far higher capacity than on your origin servers. When a DDoS attack or violation of firewall policies occur, offending connections are cut off at the Cloudflare edge, without reaching the limited bandwidth of your local system.

Magic WAN demonstrates that there can be security benefits to moving your network edge to the cloud. By configuring to block bad traffic at the edge, the limited bandwidth connections at your origin are protected. It further demonstrates that WAN-as-a-Service is possible at a large scale, which is the same class of products as my proxy.

Though neither of these Cloudflare products address the aims of my proxy, specifically the multipath problem, they show how cloud infrastructure can be leveraged to support the Internet connections of services in different capacities.

1.2 Aims

This project aims to produce proxy software that uses congestion control to manage transporting packets across multiple paths of data flow, including across discrete Internet connections. When combining Internet connections, there are three main measures that one can prioritise: throughput, resilience, and latency. This project aims to improve throughput and resilience at the cost of latency. By using a layer 3 proxy for entire IP packets, connections are combined in a way that is transparent to devices on both sides of the proxy, overcoming the throughput and availability limitations of each individual connection. The basic structure of this proxy system is shown in Figure 1.1.

The approach presented in this work achieves throughput superior to a single connection by using congestion control to split packets appropriately between each available connection. Further, resilience increases, as a connection loss results in decreased throughout, but does not lose any connection state. Latency, however, increases, as packets must travel via a proxy server. Fortunately, the wide availability of well-peered cloud servers allows for this latency increase to be kept minimal, affecting only the most latency sensitive applications.

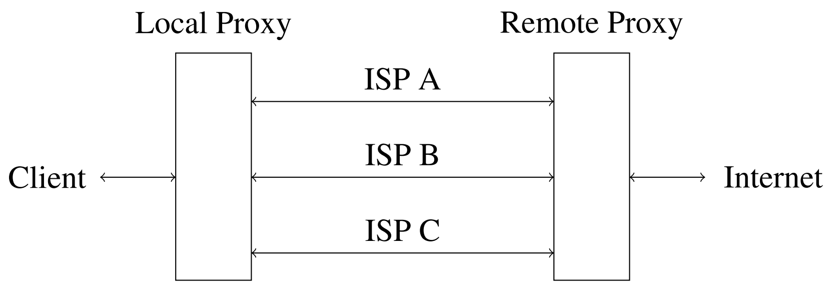


Fig. 1.1 The basic components of this proxy.

Chapter 2

Preparation

Proxying packets is the process of taking packets that arrive at one location and transporting them to leave at another. This chapter focuses on the preparatory work to achieve this practically and securely, given the design outlined in the previous chapter, in which the proxy consolidates multiple connections to appear as one to both the wider Internet and devices on the local network. In Sections 2.1 and 2.2, I discuss the security risks and plans to confront them. In Section 2.3, I present three languages: Go, Rust and C++, and provide context for choosing Go as the implementation language. Finally, in sections 2.4 and 2.5, I present a requirements analysis and a description of the engineering approach for the project.

2.1 Security Analysis

Any connection between two computers presents a set of security risks. A proxy adds some further risks to this, as additional attack vectors are created by the proxy itself. Firstly, this section focuses on layered security. If we consider the Local Proxy and Remote Proxy, with everything in between, as a single Internet connection, layered security focuses on how the risks of this combined connection compare to that of a standard Internet connection, and what guarantees must be made to achieve the same risks for a proxied connection as for a standard connection.

The transportation of packets is in three sections, as shown in Figure 2.1. The first segment of the figure is Client-to-Proxy, which occurs physically in the local zone. The second section is Proxy-to-Proxy, which occurs across the Internet. Finally is Proxy-to-Server, which also occurs across the Internet. With the goal of providing security equivalent to a standard connection, the Client-to-Proxy communication can be considered safe - it is equivalent to connecting a client directly to a modem. Therefore, this section will focus on the transports of Proxy-to-Proxy, and Proxy-to-Server communication. The threat model for this analysis will now be described.

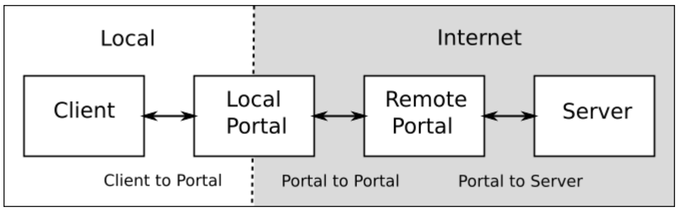


Fig. 2.1 A summary of the three different transportation zones in this proxy, with grey shading indicating an adversarial network.

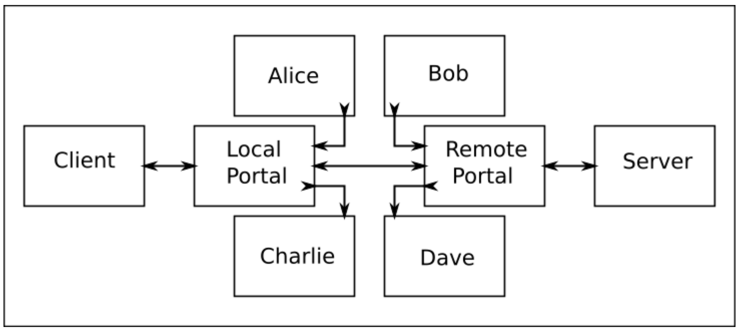


Fig. 2.2 Four potential directions of attack for Proxy-to-Proxy communication.

2.1.1 Threat Model

The threat model considered here will be that packets can be injected, read, and black-holed at any point in the Internet. This is the model employed by Dolev and Yao (1983), in which the attacker has full control of the message while it's transmitting over the Internet. Private networks will be considered safe, covering both the connection between from client to local proxy, and any connections within a VPN (Virtual Private Network).

2.1.2 Proxy-to-Proxy Communication

There are four locations to insert or remove packets in the transport between the two proxies, as shown in Figure 2.2. In this case, Alice can insert packets to the local proxy to be sent to the client, Bob can insert packets to the remote proxy to be sent to the world, Charlie can steal packets from the local proxy destined for the remote proxy, and Dave can steal packets from the remote proxy destined for the local proxy. Each of these will be examined for the impact that it would cause.

The impact of Alice inserting packets of their choosing to the local proxy is none. Considering a client connected directly to a modem, any device on the Internet is able to send packets to this modem. In the case of Alice inserting these packets, they could simply send the packets to the remote proxy instead, achieving the same effect. As such, inserting packets destined for the client presents no additional risk.

The impact of Bob inserting packets of their choosing to the remote proxy creates a legal risk to the user, and further cost. For example, Bob may be inserting packets destined for Frank, of an illegal nature. As the machine running the remote proxy is your responsibility, these packets would appear to have come from you. Similarly, if using a metered service such as many cloud providers, all traffic inserted by Bob will be billed to you. Thus it is highly important to prevent attackers such as Bob from inserting packets that will be forwarded as if from you.

Charlie and Dave black-holing packets provides the same risk in either direction, which is denial of service. Even if only a small percentage of packets are able to be stolen, the increase in packet loss has a significant effect on any loss based congestion control mechanisms. This applies whether to the tunnelled flows, or to the congestion controlled flows between proxies themselves. Mitigations for this will focus on opportunities for stealing packets unique to this proxy setup, such as convincing one proxy to send you a portion of its outgoing packets. As stated in Section 2.1.1, attackers are able to black-hole packets going to a server on the Internet regardless of the presence of this proxy, so this will not be mitigated.

2.1.3 Proxy-to-Server Communication

Packets between the proxy and server are transmitted openly across the Internet. As this proxy transports entire IP packets at layer 3, no security guarantees need be maintained once the IP packet has left the remote proxy, as it is the responsibility of the application to provide its own security guarantees. Maintaining the same level of security as a standard connection can therefore be achieved by ensuring that the packets which leave one side of a proxy are a subset of the packets that entered the other side.

2.2 Security Solutions

This section provides means of alleviating the risks given in Section 2.1. To achieve this goal, the authenticity of packets will be verified. Authenticity in this context means two properties of the object hold: integrity and freshness (Anderson, 2008, pp. 14). Integrity guarantees that any modification between the sending and receiving proxies can be detected, while freshness guarantees that reuse of a previously transmitted packet can be detected.

2.2.1 Message Authentication

To provide integrity and freshness for each message, I evaluate two choices: Message Authentication Codes (MACs) and Digital Signatures. A MAC is a hash digest generated from a concatenation of data and a secret key. The hash digest is appended to the data before transmission. Anyone sharing the secret key can perform an identical operation to verify the hash and, therefore, the integrity of the data (Menezes et al., 1997, pp. 352). Producing a digital signature for a message uses the private key in a public/private keypair to generate a digital signature for a message, proving that the message was signed by the owner of the private key, which can be verified by anyone with the corresponding public key (Anderson, 2008, pp. 147-149). In each case, a code is appended to the message, such that the integrity and authenticity of the message can be verified.

As both proxy servers are controlled by the same party, non-repudiation - the knowledge of which side of the proxy provided an authenticity guarantee for the message - is not necessary. This leaves MAC as the message authentication of choice for this project, as producing MACs is less computationally complex than digital signatures, while not providing non-repudiation.

2.2.2 Connection Authentication

Beyond authenticating messages themselves, the connection built between the two proxies must be authenticated. Consider a person-in-the-middle attack, where an attacker forwards the packets between the two proxies. Then, the attacker stops forwarding packets, and instead black holes them. This creates the denial of service mentioned in the previous section.

To prevent such forwarding attacks, the connection itself must be authenticated. I present two methods to solve this, the first being address allow-lists, and the second authenticating the IP address and port of each sent packet. The first solution is static, and simply states that the listening proxy may only respond to new communications when the IP address of the communication is in an approved set. This verifies that the connection is from an approved party, as they must control that IP to create a two-way communication from it.

The second is a more dynamic solution. The IP authentication header (Kent, 2005) achieves this by protecting all immutable parts of the IP header with an authentication code. In the case of this software, authenticating the source IP address, source port, destination IP address, and destination port ensures connection authenticity. By authenticating these addresses, which can be checked easily at both ends, it can be confirmed that both devices knew with whom they were talking, and from where the connection was initiated. That is, an owner of the shared key authorised this communication path.

However, both of these solutions have some shortfalls when Network Address Translation (NAT) is involved. The second solution, authenticating addresses, fails with any form of NAT. This is because the IPs and ports of the packets sent by the sending proxy are different to when they will be received by the receiving proxy, and therefore cannot be authenticated. The first solution, providing a set of addresses, fails with Carrier Grade NAT (CG-NAT), as many users share the same IP address, and hence anyone under the same IP could perform an attack. In most cases one of these solutions will work, else one can fail over to the security layering presented in Section 2.2.4.

2.2.3 Freshness

To ensure freshness of received packets, an anti-replay algorithm is employed. Replay protection in IP authentication headers is achieved by using a sequence number on each packet. This sequence number is monotonically and strictly increasing. The algorithm that I have chosen to implement for this is *IPsec Anti-Replay Algorithm without Bit Shifting* (Tsou and Zhang, 2012), also employed in Wireguard (Donenfeld, 2017).

When applying message authentication, it was sufficient to authenticate messages individually to their flow. However, replay protection must be applied across all flows connected to the proxy, otherwise, a packet could be repeated but appearing as a different connection, and therefore remain undetected. This is similar to the design pattern of MPTCP's congestion control, where there is a separation between the sequence number of individual subflows and the sequence number of the data transport as a whole (Wischik et al., 2011, pp. 11).

2.2.4 Layered Security

It was previously mentioned that my solution is transparent to the higher layer security encapsulated by proxied packets. Further to this, my solution provides transparent security in the other direction, where proxied packets are encapsulated by another security solution. Consider the case of a satellite office that employs both a whole network corporate VPN and my solution. The network can be configured in each of two cases: the multipath proxy runs behind the VPN, or the VPN runs behind the multipath proxy.

Packet structures for proxied packets in each of these cases are given in Appendix B, as Figure B.1 and Figure B.2 for the VPN Wireguard (Donenfeld, 2017). In Figure B.2, the proxies are only accessible via the VPN protected network. It can be seen that the packet in Figure B.2 is shorter, given the removal of the message authentication code and the data sequence number. The data sequence number is unnecessary, given that Wireguard uses the same anti-replay algorithm, and thus replayed packets would have been caught entering the secure network. Further, the message authentication code is unnecessary, as the authenticity of packets is now guaranteed by Wireguard.

Supporting and encouraging this layering of protocols provides a second benefit: if the security in my solution degrades with time, there are two options to repair it. One can either fix the open source application, or compose it with a security solution that is not broken, but perhaps provides redundant security guarantees, translating to additional overhead. To this end, the security features mentioned are all configurable. This allows for flexibility in implementation.

The benefits of using a VPN tunnel between the two proxies are shown in Figure 2.3. Whereas in Figure 2.1 the proxy-to-proxy communication is across the unprotected Internet, in Figure 2.3 this communication occurs across a secure overlay network. This allows the packet transport to be trusted, and avoids the need for additional verification. Further, it allows the application to remain secure in any

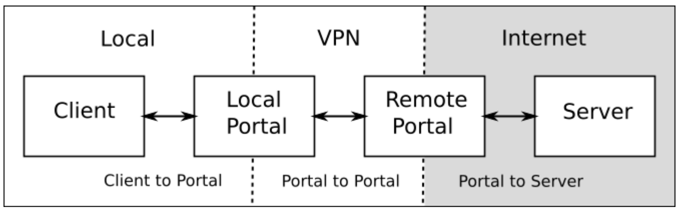


Fig. 2.3 A summary of the three different transportation zones in this proxy behind a VPN, with grey shading indicating an adversarial network.

situation where a VPN will work. Home users, in most cases, would use this solution with the inbuilt authentication mechanisms. Business users, who already have a need for a corporate VPN, would benefit from running my solution across VPN tunnels, avoiding the need to complete authentication work multiple times.

2.3 Language Selection

In this section, I evaluate three potential languages (C++, Rust and Go) for the implementation of this software. To support this evaluation, I have provided a sample program in each language. The sample program is a minimal example of reading packets from a TUN interface, placing them in a queue from a single thread, and consuming the packets from the queue with multiple threads. These examples are given in Figures A.1 through A.3, in Appendix A. For each language, I considered the performance, code clarity, and the language ecosystem. This culminated in choosing Go for the implementation language.

I similarly evaluated two languages for the test suite: Python and Java. Though Python was initially chosen for rapid development and better ecosystem support, the final test suite is a combination of both Python and Java - Python for data processing, and Java for systems interaction.

2.3.1 Implementation Languages

C++

There are two primary advantages to completing this project in C++: speed of execution, and C++ being low level enough to achieve this project's goals (which turned out to be true for all considered languages).

The negatives of using C++ are demonstrated in the sample script, given in Figure A.1: achieving even the base functionality of this project requires multiple times more code than Rust or Go (93 lines compared to 34 for Rust or 48 for Go). This arises from the need to manually implement the required thread safe queue, which is available as a library for Rust, and included in the Go runtime. This manual implementation gives rise to additional risk of incorrect implementation, specifically with regards to thread safety, that could cause undefined behaviour, security vulnerabilities, and great difficulty debugging. Further, although open source queues are available, they are not handled by a package manager, and thus security updates would have to be manual, risking the introduction of bugs. Finally, C++ does not provide any memory safety guarantees.

Rust

Rust is memory safe and thread safe, solving the latter issues with C++. Rust also has a minimal runtime, allowing for an execution speed comparable to C or C++. The Rust sample is given in Figure A.2, and is pleasantly concise.

For the purposes of this project, Rust's youthfulness is a negative. This is two-faceted: Integrated Development Environment (IDE) support and crate stability (crates are the Rust mechanism for package management). Firstly, the IDE support for Rust in my IDEs of choice is provided via a plugin to IntelliJ, and is not as well supported as the other languages. Secondly, the crate available for TUN support (tun-tap[1]) does not yet provide a stable Application Programming Interface (API), which was noticed during test program development. Between writing the program initially and re-testing when documenting it, the crate API had changed to the point where my script no longer type checked. Further, the old version had disappeared, and thus I was left with a program that did not compile or function. Although I could write the API for TUN interaction myself, the safety benefits of Rust would be less pronounced, as the direct systems interaction requires `unsafe` code, which bypasses parts of the type-checker and borrow-checker, leading to an increased potential for bugs.

Go

The final language to evaluate is Go, often written as GoLang. The primary difference between Go and the other two evaluated languages is the presence of a runtime. The code sample is provided in Figure A.3. Go is significantly higher level than the other two languages mentioned, and provides a memory management model that is both simpler than C++ and more standard than Rust.

For the greedy structure of this project, Go's focus on concurrency is extremely beneficial. Go has channels in the standard runtime, which support any number of both producers and consumers. In this project, both SPMC (Single Producer Multi Consumer) and MPSC (Multi Producer Single Consumer) queues are required, so having these provided as a first class feature of the language is beneficial.

Garbage collection and first order concurrency come together to make the code produced for this project highly readable, but relies on a more complex runtime than the other two languages evaluated. The downside is that the speed of execution is negatively affected by this runtime. However, for the purposes of this first production, that compromise is acceptable. By producing code that makes the functionality of the application clear, future implementations could more easily be built to mirror it. Given the performance shown in Section 4.4, the benefits of the compromise of using a well-suited, high-level language are clearly evident.

2.3.2 Evaluation Languages

Python

Python is a dynamically typed language, and it was chosen as the initial implementation language for the test suite. The first reason for this is `matplotlib`,[2] a widely used graphing library that can produce the graphs needed for this evaluation. The second reason is `proxmoxer`[3], a fluent API for interacting with a Proxmox server.

Having the required modules available allowed for a swift initial development sprint. This showed that the method of evaluation was viable and effective. However, the requirements of evaluation changed with the growth of the software, and an important part of an agile process is adapting to

[1]https://docs.rs/tun-tap/
[2]https://matplotlib.org/
[3]https://github.com/proxmoxer/proxmoxer

changing requirements. The lack of static typing complicates the refactorability of Python, and becomes increasingly challenging as the project grows. Therefore, after the initial proof of concept, it became necessary to explore another language for the Proxmox interaction.

Java

Java is statically typed and became the implementation language for all external interaction within the test suite. The initial reason for not choosing Java was the lack of availability of an equivalent library to `proxmoxer`. However, as the implementation size grew in Python, the lack of static typing meant that making changes to the system without adding bugs became particularly difficulty. Further, productivity was reduced by lack of code suggestions provided by `proxmoxer` without type hints, as much API documentation had to be read for each implemented piece of code.

To this end, I developed a library in Java with an almost identical interface, but providing a high degree of type-safety. This allowed for much safer changes to the program, while also encouraging the use of IDE hints for quickly generating code. Although the data gathering was much improved by switching to Java, the code for generating graphs was perfectly manageable in Python. As such, a hybrid solution with Java for data gathering and Python for data processing was employed.

2.4 Requirements Analysis

The requirements of the project are detailed in the Success Criteria of the Project Proposal (Appendix D), and are the primary method of evaluation for project success. They are split into three categories: success criteria, extended goals and stretch goals.

The three categories of success criteria can be summarised as follows. The success criteria, or must have elements, are to provide a multi-path proxy that is functional, secure and improves speed and resilience in specific cases. The extended goals, or should have elements, are focused on increasing the performance and flexibility of the solution. The stretch goals, or could have elements, are aimed at increasing performance by reducing overheads, and supporting IPv6 alongside IPv4.

Beyond the success criteria, I wanted to demonstrate the practicality of my software on prototypic networking equipment; therefore, continuous integration testing and evaluation will run on Linux and FreeBSD.

2.5 Engineering Approach

Software Development Model

The development of this software followed the agile methodology. Work was organised into weekly sprints, aiming for increased functionality in the software each time. By focusing on sufficient but not excessive planning, a minimum viable product was quickly established. From there, the remaining features could be implemented in the correct sized segments. Examples of these sprints are: initial build including configuration, TUN adaptors and main program; TCP transport, enabling an end-to-end connection between the two parts; repeatable testing, providing the data to evaluate each iteration of the project against its success criteria; UDP transport for performance and control.

The agile methodology welcomse changing requirements (Beck et al., 2001), and as the project grew, it became clear where shortcomings existed, and these could be fixed in very quick pull requests. An example is given in Figure 2.4, in which the type of a variable was changed from `string` to `func() string`. This allowed for lazy evaluation, when it became clear that configuring fixed IP addresses

```go
type InitiatedFlow struct {
        Local  string
        Remote string

        mu sync.RWMutex

        Flow
}
```

(a) The structure with a fixed local address.

```go
type InitiatedFlow struct {
        Local  func() string
        Remote string

        mu sync.RWMutex

        Flow
}
```

(b) The structure with a dynamic local address.

Fig. 2.4 An example of refactoring for changing requirements.

or DNS names could be impractical. Static typing enables refactors like this to be completed with ease, particularly with the development tools mentioned in the next section, reducing the incidental complexity of the agile methodology.

Development Tools

A large part of the language choice focused on development tools, particularly IDE support. I used GoLand (Go), IntelliJ (Java), and PyCharm (Python). Using intelligent IDEs, particularly with the statically-typed Go and Java, significantly increases programming productivity. They provide code suggestions and automated code generation for repetitive sections to reduce keystrokes, syntax highlighting for ease of reading, near-instant type checking without interaction, and many other features. Each reduce incidental complexity.

I used Git version control, with a self-hosted Gitea[4] server as the remote. The repository contains over 180 commits, committed at regular intervals while programming. I maintained several on- and off-site backups (Multiple Computers + Git Remote + NAS + 2xCloud + 2xUSB). The Git remote was updated with every commit, the NAS and Cloud providers daily, with one USB updated every time significant work was added and the other a few days after. Having some automated and some manual backups, along with a variety of backup locations, minimises any potential data loss in the event of any failure. The backups are regularly checked for consistency, to ensure no data loss goes unnoticed.

Alongside my Gitea server, I have a self-hosted Drone[5] server for continuous integration: running Go tests, verifying formatting, and building artefacts. On a push, after verification, each artefact is built, uploaded to a central repository, and saved under the branch name. This dovetailed with my automated testing, which downloaded the relevant artefact automatically for the branch under test. I also built artefacts for multiple architectures to support real world testing on `AMD64` and `ARM64` architectures.

Continuous integration and Git are used in tandem to ensure that each pull request meet certain standards before merging, reducing the possibility of accidentally causing performance regressions. Pull requests also provide an opportunity to review submitted code, even with the same set of eyes, in an attempt to detect any glaring errors. Twenty-four pull requests were submitted to the repository for this project.

Licensing

I chose to license this software under the MIT license, which is simple and permissive.

[4]https://gitea.com/
[5]http://drone.io/

2.6 Starting Point

I had significant experience with the language Go before the start of this project, though not formally taught. My knowledge of networking is limited to that of a user, and the content of the Part IB Tripos courses *Computer Networking* and *Principles of Communication* (the latter given after the start of this project). The security analysis drew from the Part IA course *Software and Security Engineering* and the Part IB course *Security*. As the software is highly concurrent, the Part IB course *Concurrent and Distributed Systems* and the Part II Unit of Assessment *Multicore Semantics and Programming* were applied.

2.7 Summary

In this Chapter, I described my preparation for developing, testing and securing my proxy application. I chose to implement MACs, authenticated headers, and IP allow-lists for security, while maintaining composability with other solutions such as VPNs. I will be using Go as the implementation language for its high-level features that are well suited to this project, and Python and Java for evaluation for programming speed and type-safety, respectively. I have prepared a set of development tools, including IDEs, version control and continuous integration, to encourage productivity as both a developer and a project manager.

Chapter 3

Implementation

Implementation of the proxy is in two parts: software that provides a multipath layer 3 tunnel between two hosts, and the system configuration necessary to utilise this tunnel as a proxy. An overview of the software and system is presented in Figure 3.1.

This chapter details this implementation in three sections. The software will be described in Sections 3.1 and 3.2. Section 3.1 details the implementation of both TCP and UDP methods of transporting the tunnelled packets between the hosts. Section 3.2 explains the software's structure and dataflow. The system configuration will be described in Section 3.3. Figure 3.1 shows the path of packets within the proxy, and it will be referenced throughout these sections.

3.1 Packet Transport

As shown in Figure 3.1, the interfaces through which transport for packets is provided between the two hosts are producers and consumers. A transport pair is between a consumer on one proxy and a producer on the other, where packets enter the consumer and exit the corresponding producer. Two methods for producers and consumers are implemented: TCP and UDP. As the greedy load balancing of this proxy relies on congestion control, TCP provided an initial proof-of-concept, while UDP expands on this proof-of-concept to remove unnecessary overhead and improve performance in the case of TCP-over-TCP tunnelling. Section 3.1.1 discusses the method of transporting discrete packets across the continuous byte stream of a TCP flow, before describing why this solution is not ideal. Then, Section 3.1.2 goes on to discuss adding congestion control to UDP datagrams, while avoiding retransmitting a proxied packet.

3.1.1 TCP

The requirements for greedy load balancing to function are simple: flow control and congestion control. TCP provides both of these, so was an obvious initial solution. However, TCP also provides unnecessary overhead, which will go on to be discussed further.

A TCP flow cannot be connected directly to a TUN adaptor, as the TUN adaptor accepts and outputs discrete and formatted IP packets while the TCP connection sends a stream of bytes. To resolve this, each packet sent across a TCP flow is prefixed with the length of the packet. When a TCP consumer is given a packet to send, it first sends the 32-bit length of the packet across the TCP flow, before sending the packet itself. The corresponding TCP producer then reads these 4 bytes from the TCP flow, before reading the number of bytes specified by the received number. This enables punctuation of the stream-oriented TCP flow into a packet-carrying connection.

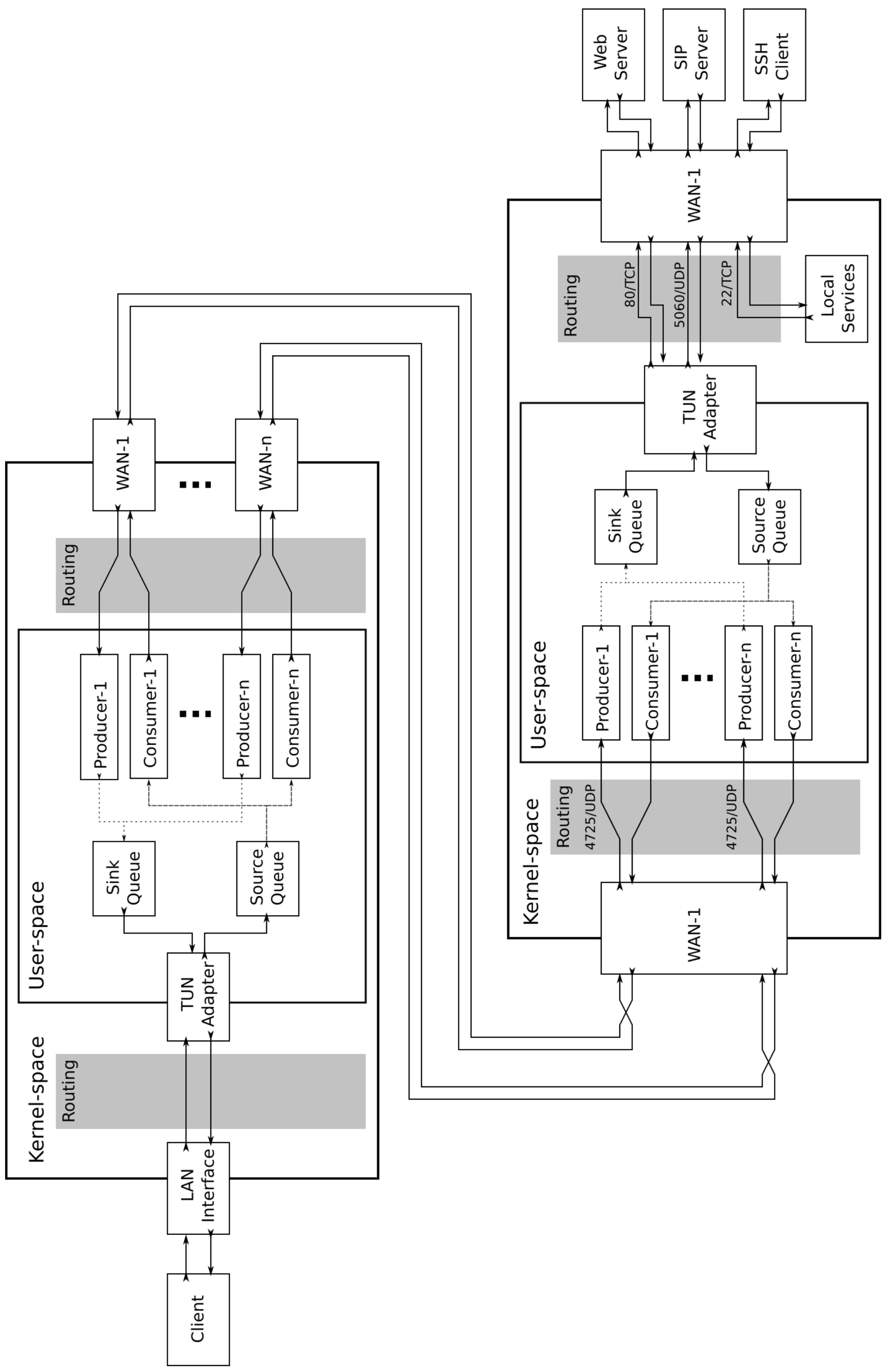


Fig. 3.1 Diagram of packet path from a client behind the proxy to a server on the Internet.

However, using TCP to tunnel TCP packets (TCP-over-TCP) can cause a degradation in performance (Honda et al., 2005). Further, using TCP to tunnel IP packets provides a superset of the required guarantees, in that reliable delivery and ordering are guaranteed. Reliable delivery can cause a decrease in performance for tunnelled flows which may not require reliable delivery, such as a live video stream. Ordering can limit performance when tunnelling multiple streams, as a packet for a phone call could already be received, but instead has to wait in a buffer for a packet for an unrelated download to arrive.

Although the TCP implementation provides an excellent proof-of-concept, work moved to a second UDP implementation, aiming to solve some of these problems. However, the TCP implementation is functionally correct; in cases where a connection that suffers particularly high packet loss is combined with one which is more stable, TCP could be employed on the high loss connection to limit overall packet loss. The effectiveness of such a solution would be implementation specific, so is left for the architect to decide.

3.1.2 UDP

After initial success with the TCP proof-of-concept, work moved to developing a UDP protocol for transporting the proxied packets. UDP differs from TCP in providing a more basic mechanism for sending discrete messages, while TCP provides a stream of bytes. Implementing a UDP datagram proxy solution returns control from the kernel to the application itself, allowing much more fine-grained management of congestion control. Further, UDP provides increased performance over TCP by removing ordering guarantees, and improving the quality of TCP tunnelling compared to TCP-over-TCP. This allows maximum flexibility, as application developers should not have to avoid using TCP to maintain compatibility with my proxy.

This section first describes the special purpose congestion control mechanism designed, which uses negative acknowledgements to avoid retransmissions. This design informs the design of the UDP packet structure. Finally, this section discusses the initial implementation of congestion control, which is based on the characteristic curve of TCP New Reno (Henderson et al., 2012).

3.1.3 Congestion Control

Congestion control is most commonly applied in the context of reliable delivery. This provides a significant benefit to TCP congestion control protocols: cumulative acknowledgements. As all of the bytes should always arrive eventually, unless the connection has faulted, the acknowledgement number (ACK) can simply be set to the highest received byte. Therefore, some adaptations are necessary for such a congestion control algorithm to apply in an context where reliable delivery is not expected. Firstly, for a packet based connection, ACKing specific bytes makes little sense - a packet is atomic, and is lost as a whole unit. To account for this, sequence numbers and their respective acknowledgements will be for entire packets, as opposed to per byte.

Secondly, for an protocol that does not guarantee reliable delivery, cumulative acknowledgements are not as simple. As tunnelled packets are now allowed to never arrive within the correct function of the flow, a situation where a packet is never received would cause deadlock with an ACK that is simply set to the highest received sequence number, demonstrated in Figure 3.2b. Neither side can progress once the window is full, as the sender will not receive an ACK to free up space within the window, and the receiver will not receive the missing packet to increase the ACK. In TCP, one would expect the missing packet (one above the received ACK) to be retransmitted, which allows the ACK to catch up in only one RTT. However, as retransmissions are to be avoided, the UDP solution presented here would become deadlocked - the sending side knows that the far side has not received the packet, but must not retransmit.

SEQ	ACK
1	0
2	0
3	2
4	2
5	2
6	5
6	6

(a) ACKs only responding to in order sequence numbers

SEQ	ACK
1	0
2	0
3	2
5	3
6	3
7	3
7	3

(b) ACKs only responding to a missing sequence number

SEQ	ACK	NACK
1	0	0
2	0	0
3	2	0
5	2	0
6	2	0
7	6	4
7	7	4

(c) ACKs and NACKs responding to a missing sequence number

Fig. 3.2 Congestion control responding to correct and missing sequence numbers of packets.

I present a solution based on Negative Acknowledgements (NACKs). When the receiver believes that it will never receive a packet, it increases the NACK to the highest missing sequence number, and sets the ACK to one above the NACK. This occurs after a timeout that is presently set at $3*RTT$ (Round Trip Time). The ACK algorithm is then performed to grow the ACK as high as possible. This is simplified to any change in NACK representing at least one lost packet, which can be used by the specific congestion control algorithms to react. Though this usage of the NACK appears to provide a close approximation to ACKs on reliable delivery, the choice of how to use the ACK and NACK fields is delegated to the congestion controller implementation, allowing for different implementations if they better suit the method of congestion control. Using NACKs, the deadlock in Figure 3.2c can be avoided, with the case in Figure 3.2 occurring instead. The NACK is used to inform the far side that a packet was lost, and therefore allow it to continue sending fresh packets. In contrast, TCP would retransmit the missing packet, which can be avoided with this NACK-based solution.

Given the decision to use ACKs and NACKs, the packet structure for UDP datagrams can now be designed. The chosen structure is given in Figure 3.3. The congestion control header consists of the sequence number and the ACK and NACK, each 32-bit unsigned integers.

New Reno

TCP New Reno (Henderson et al., 2012) is widely known for its sawtooth pattern of throughput. New Reno is an RTT-based congestion control mechanism, which, in the steady state, increases the window size (number of packets in flight at a time) by 1 for each successful window. In the case of a retransmission, this quantity halves. The window size is the quantity of packets that can be in flight at one time, which depends on the round trip time, as a longer round trip time requires a larger window size to transmit the same amount of packets. For a freshly started New Reno connection, slow start occurs, which increases the window size by 1 for each packet transmitted successfully, as opposed to each full window of packets. This creates an exponential curve, which stops on the first transmission failure.

An algorithm that performs similarly but takes advantage of NACKs works identically for a flawless connection. That is, if no packets are lost, the implementation is identical. This includes increasing the window size by one for each successfully transmitted packet initially, and dropping to increasing by one for each window size later in the process. The difference from TCP's mechanisms arises when packets are lost, and more specifically, how that is detected. This is the NACK mechanism, which sets the NACK to the missing packet if a packet has been waiting for more than $0.5*RTT$ to be acknowledged. This occurs when packet 4 arrives before packet 3, and packet 3 has still not arrived

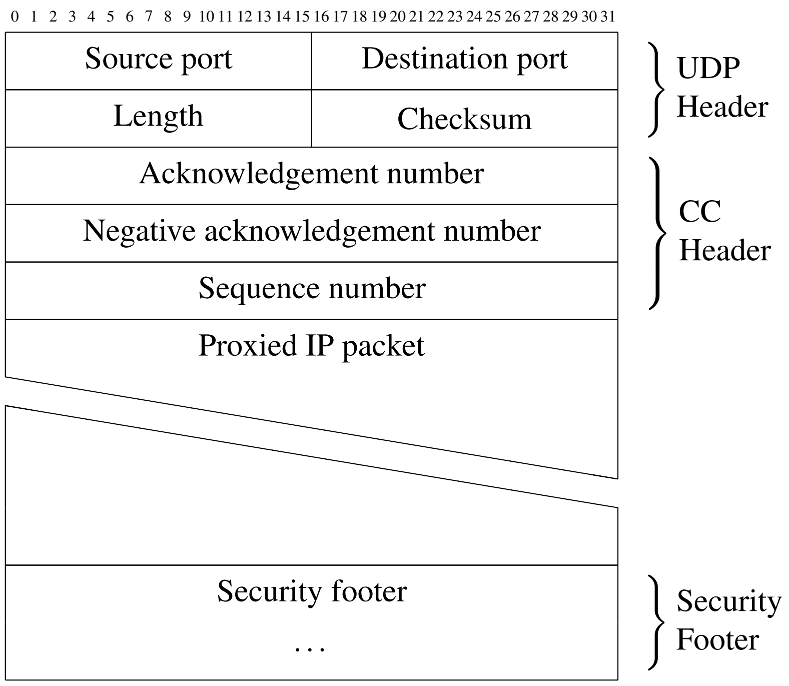


Fig. 3.3 UDP packet structure

after an additional half of the round trip time (the entire time expected for the packet to arrive), and would cause the NACK field on the next packet to be set to 3, with the ACK field set to 4. When the sender receives this NACK response, it affects the window size as TCP would (halving the size, and stopping slow start).

The congestion control algorithm has multiple threads accessing it at any one time, so uses a mixture of atomic operations and fine-grained locking to remain consistent. The `ack`, `nack` and `windowSize` fields all use atomic operations, such that they can be read immediately and allow a packet to almost be sent without gaining a lock. However, the `inFlight` and `awaitingAck` fields are each protected by a mutex, ensuring that they remain consistent. This is a compromise between performance and correctness, limiting code complexity while allowing more performance than coarse-grained locks. Further, high-level data structures (specifically, growable lists) are used, which reduce programming complexity at the cost of some performance. This allows for good readability, and increases the likelihood of writing correct code.

Congestion control is one of the main point for tests in the repository. The New Reno controller was developed mostly with test-driven development, due to the complicated interactions between threads. Though the testing of multithreaded code can be extremely challenging due to the risk of deadlock when the code is incorrect, large timeouts and a CI environment made this quite manageable.

3.2 Software Structure

This section details the design decisions behind the application structure, and how it fits into the systems where it will be used. Much of the focus is on the flexibility of the interfaces to future additions, while also describing the concrete implementations available with the software as of this work.

3.2.1 Running the Application

Initially, the application suffered from a significant race condition when starting. The application followed a standard flow, where it created a TUN adaptor to receive IP packets and then began proxying the packets from/to it. However, when running the application, no notification was received when this TUN adaptor became available. As such, any configuration completed on the TUN adaptor was racing with the TUN adaptor's creation, resulting in many start failures.

The software now runs in much the same way as other daemons you would launch, leading to a similar experience as other applications. The primary inspiration for the functionality of the application is Wireguard (Donenfeld, 2017), specifically `wireguard-go`[1]. To launch the application, the following shell command is used:

```
netcombiner nc0
```

When the program is executed as such, the following control flow occurs:

```
if not child process:
    c = validate_configuration()
    t = new_tun(nc0)
    child_process = new_process(this, c, t)
    return

proxy = new_proxy(c, t)
proxy.run()
```

Firstly, the application validates the configuration, allowing an early exit if misconfigured. Then the TUN adaptor is created. This TUN adaptor and the configuration are handed to a duplicate of the process, which sees them and begins running the given proxy. This allows the parent process to exit, while the background process continues running as a daemon.

By exiting cleanly and running the proxy in the background, the race condition is avoided. The exit is a notice to the launcher that the TUN adaptor is up and ready, allowing for further configuration steps to occur. Otherwise, an implementation specific signal would be necessary to allow the launcher of the application to move on, which conflicts with the requirement of easy future platform compatibility.

3.2.2 Security

The integrated security solution of this software is in two parts: message authentication and repeat protection. The interface for these is shared, as they perform the same action from the perspective of the producer or consumer.

Message Authenticity Verification

Message authentication is provided by a pair of interfaces, `MacGenerator` and `MacVerifier`, which add bytes at consumers and remove bytes at producers respectively. `MacGenerator` provides a method which takes input data and produces a list of bytes as output, to be appended to the message. `MacVerifier` takes the appended bytes to the message, and confirms whether they are valid for that message.

The provided implementation for message authenticity uses the BLAKE2s (Aumasson et al., 2013) algorithm. By using library functions, the implementation is achieved simply by matching the

[1]https://github.com/WireGuard/wireguard-go

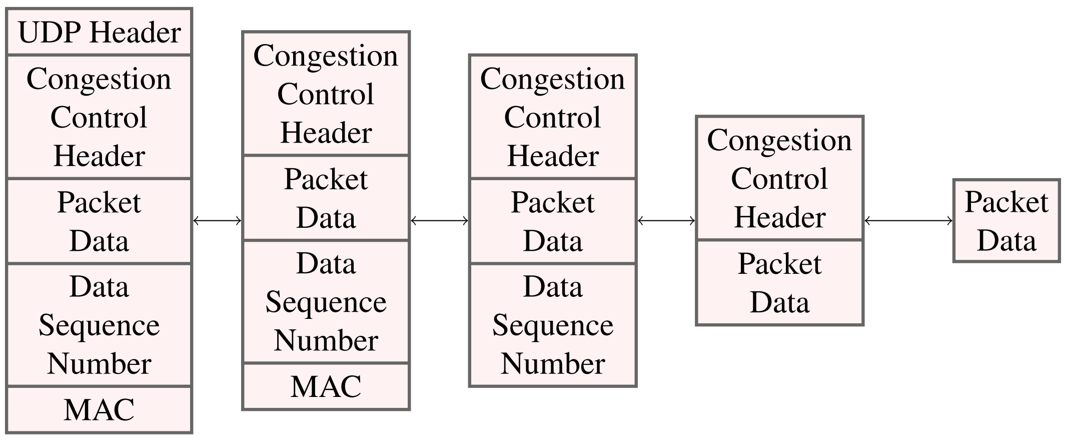


Fig. 3.4 Expansion of a UDP packet through a consumer/producer.

interface provided by the library and the interface mentioned here. This ensures clarity, and reduces the likelihood of introducing a bug.

Key exchange is presently implemented by using a secure and external channel. For example, one might configure their proxies using the Secure Shell Protocol (SSH), and would transmit the shared key over this secure channel. In future, this could be extended with external software that manages the tunnel for you, by using its own secure channel to configure the proxies with a shared key.

Repeat Protection

Repeat protection takes advantage of the same two interfaces already mentioned. To allow this to be implemented, each consumer or producer takes an ordered list of `MacGenerators` or `MacVerifiers`. When a packet is consumed, each of the generators is run in order, operating on the data of the last. When called by a producer, this operation is completed in reverse, with each `MacVerifier` stripping off the corresponding generator. An example of this is shown in Figure 3.4. Firstly, the data sequence number is generated, before the MAC. When receiving the packet, the MAC is first stripped, before the data sequence number. This means that the data sequence number is protected by the MAC.

One difference between repeat protection and MAC generation is that repeat protection is shared between all producers and consumers. This is in contrast to the message authenticity, which are, as implemented, specific to a producer or consumer. The currently implemented repeat protection is that of Tsou and Zhang (2012). The code sample is provided with a BSD license, so is compatible with this project, and hence was simply adapted from C to Go. This is created at a host level when building the proxy, and the same shared amongst all producers, so has to be thread safe. Producing the sequence numbers is achieved with a single atomic operation, avoiding the need to lock at all. Verifying the sequences requires altering multiple elements of an array of bytes, so uses locking to ensure consistency. Ensuring that locks are only taken when necessary makes the calls as efficient as possible.

3.2.3 Repository Overview

A directory tree of the repository is provided in Figure 3.5. The top level is split between `code` and `evaluation`, where `code` is compiled to produce the application binary, and `evaluation` is used to verify the performance characteristics and generate graphs. The Go code is built with the Go modules system, the Java code built with Gradle, and the Python code runs in an iPython notebook. Go tests are

```
/
├── code ................................................ Go code for the project
│   ├── config ...................................... Configuration management
│   ├── crypto ......................................... Cryptographic methods
│   │   └── sharedkey ...................................... Shared key MACs
│   ├── flags ..................................... Command line flag processing
│   ├── mocks ........................................ Mocks to enable testing
│   ├── proxy ..................................... The central proxy controller
│   ├── replay ............................................. Replay protection
│   ├── shared ................................................. Shared errors
│   ├── tcp ............................................... TCP flow transport
│   ├── tun ................................................... TUN adaptor
│   ├── udp ........................................... UDP datagram transport
│   │   └── congestion ............................. Congestion control methods
│   └── .drone.yml ........................................... CI specification
└── evaluation ............................ Result gathering and graph generation
    ├── java ................................... Java automated result gathering
    └── python ........................................... Python graph generation
```

Fig. 3.5 Repository folder structure.

interspersed with the code, for example in a file named `flow_test.go`, providing tests for `flow.go` in the same directory.

3.3 System Configuration

The software portion of this proxy is entirely symmetric, as can be seen in Figure 3.1. However, the system configuration diverges, as each side of the proxy serves a different role. Referring to Figure 3.1, it can be seen that the kernel routing differs between the two nodes. Throughout, these two sides have been referred to as the local proxy and the remote proxy, with the local in the top left and the remote in the bottom right.

As the software portion of this application is implemented in user-space, it has no control over the routing of packets. Instead, a virtual interface is provided, and the kernel is instructed to route relevant packets to/from this interface. In sections 3.3.1 and 3.3.2, the configuration for routing the packets for the remote proxy and local proxy respectively are explained. Finally, in Section 3.3.3, some potentially unexpected behaviour of using devices with multiple interfaces is discussed, such that the reader can avoid some of these pitfalls. Throughout this section, examples will be given for both Linux and FreeBSD. Though these examples are provided, they are one of many methods of achieving the same results.

3.3.1 Remote Proxy Routing

The common case for remote proxies is a cloud Virtual Private Server (VPS) with one public network interface. As such, some configuration is required to both proxy bidirectionally via that interface, and also use it for communication with the local proxy. Firstly, packet forwarding must be enabled for the device. On Linux this is achieved as follows:

```
sysctl -w net.ipv4.ip_forward=1
```

Or on FreeBSD via:

```
echo 'GATEWAY_ENABLE="YES"' >> /etc/rc.conf
```

These instruct the kernel in each case to forward packets. However, more instructions are necessary to ensure packets are routed correctly once forwarded. For the remote proxy, this involves two things: routing the communication for the proxy to the software side, and routing items necessary to the local system to the relevant application. Both of these are achieved in the same way, involving adjustments to the local routing table on Linux, and using `pf(4)` rules on FreeBSD.

Linux:

```
# Add a new rule to the local table at a lower priority
ip rule add from all table local priority 20
# Delete the existing lowest priority rule (always to the local table)
ip rule del priority 0
# Forward SSH traffic to the host
ip rule add to "$REMOTE_PORTAL_ADDRESS" dport 22 table local priority 1
# Forward proxy traffic to the host
ip rule add to "$REMOTE_PORTAL_ADDRESS" dport 4725 table local priority 1
# Create a new routing table and route for crossing the TUN
ip route add table 19 to "$REMOTE_PORTAL_ADDRESS" via 172.19.152.3 dev nc0
# Route all packets not already caught via the TUN
ip rule add to "$REMOTE_PORTAL_ADDRESS" table 19 priority 19
```

FreeBSD:

```
# Forward SSH traffic to the host
pass in quick on $ext_if inet proto tcp to ($ext_if) port { 22 }
# Forward proxy traffic to the host
pass in quick on $ext_if inet proto udp to ($ext_if) port { 4725 }
# Forward everything via the netcombiner interface
pass out quick on $nc_if inet to ($ext_if)
```

These settings combined will provide the proxying effect via the TUN interface configured in software. It is also likely worth firewalling much more aggressively at the remote proxy side, as dropping packets before saturating the low bandwidth connections between the local and remote proxy improves resilience to denial of service attacks. This can be completed either with similar routing and firewall rules to those above, or externally with many cloud providers.

3.3.2 Local Proxy Routing

Routing within the local proxy expects $1+N$ interfaces: one connected to the client device expecting the public IP, and N connected to the wider Internet for communication with the other node. Referring to Figure 3.1, it can be seen that no complex rules are required to achieve this routing, as each interface serves a different role. As such, there are three goals: ensure the packets for the remote IP are routed from the TUN to the client device and vice versa, ensuring that packets destined for the remote proxy are not routed to the client, and ensuring each connection is routed via the correct WAN connection. The first two will be covered in this section, with a discussion on the latter in the next section.

Routing the packets from/for the local proxy is pleasantly easy. Firstly, enable IP forwarding for Linux or gateway mode for FreeBSD, as seen previously. Secondly, routes must be setup. Fortunately, these routes are far simpler than those for the remote proxy. The routing for the local proxy client interface is as follows on Linux:

```
ip addr add 192.168.1.1 dev "$CLIENT_INTERFACE"
ip route add "$REMOTE_PORTAL_ADDR" dev "$CLIENT_INTERFACE"
```

Or on FreeBSD:

```
ifconfig "$CLIENT_INTERFACE" 192.168.1.1 netmask 255.255.255.255
route add "$REMOTE_PORTAL_ADDR" -interface "$CLIENT_INTERFACE"
```

Then, on the client device, simply set the IP address statically to the remote proxy address, and the gateway to `192.168.1.1`. Now the local proxy can send and receive packets to the remote proxy, but some further routing rules are needed to ensure that the packets from the proxy reach the remote proxy, and that forwarding works correctly. This falls to routing tables and `pf(4)`, so for Linux:

```
# The local table has priority, so packets for the proxy will be routed correctly
# Add a default route via the other node via the tunnel
ip route add table 20 default via 172.19.152.2 dev nc0
# Use this default route for outbound client packets
ip rule add from "$REMOTE_PORTAL_ADDRESS" iif "$CLIENT_INTERFACE" table 20 priority 20
# Add a route to the client
ip route add table 21 to "$REMOTE_PORTAL_ADDRESS" dev "$CLIENT_INTERFACE"
# Use this route for packets to the remote portal from the tunnel
# Note: there must be a higher priority table for proxy packets
ip rule add to "$REMOTE_PORTAL_ADDRESS" table 21 priority 21
```

FreeBSD:

```
# Route packets due to the other node via the WAN interface
pass out quick on $ext_if to $rp_ip port { 4725 }
# Else route these packets to the client
pass out quick on $cl_if to $rp_ip
# Route packets due to this node locally
pass in quick on $ext_if from $rp_ip port { 4725 }
# Else route these packets via the tunnel
pass out quick on $nc_if from $rp_ip
```

These rules achieve both the listed criteria, of communicating with the remote proxy while also forwarding the packets necessary to the client. The local proxy can be extended with more functionality, such as NAT and DHCP. This allows plug and play for the client, while also allowing multiple clients to take advantage of the connection without another router present.

3.3.3 Multi-Homed Behaviour

During testing, I discovered behaviour that I found surprising when it came to multi-homed hosts. Here I will detail some of this behaviour, and workarounds found to enable the software to still work well regardless.

The first piece of surprising behaviour comes from a device which has multiple interfaces lying on the same subnet. Consider a device with two Ethernet interfaces, each of which gains a DHCP IPv4 address from the same network. The first interface `eth0` takes the IP `10.10.0.2` and the second `eth1` takes the IP `10.10.0.3`, each with a subnet mask of `/24`. If a packet originates from userspace with source address `10.10.0.2` and destination address `10.10.0.1`, it may leave via either `eth0` or `eth1`. I initially found this behaviour very surprising, as it seems clear that the packet should be delivered from `eth0`, as that is the interface which has the given IP. However, as the routing is completed by the source subnet, each of these interfaces match.

Although this may seem like a contrived use case, consider this: a dual WAN router lies in front of a server, which uses these two interfaces to take two IPs. Policy routing is used on the dual WAN router to allow this device control over choice of WAN, by using either of its LAN IPs. In this case, this default routing would mean that the userspace software has no control over the WAN, as one will be selected seemingly arbitrarily. The solution to this problem is manipulation of routing tables. By creating a high priority routing table for each interface, and routing packets more specifically than the default routes, the correct packets can be routed outbound via the correct interface.

The second issue follows a similar theme of IP addresses being owned by the host and not the interface which has that IP set, as Linux hosts respond to ARP requests for any of their IP addresses on all interfaces by default. This problem is known as ARP flux. Going back to our prior example of `eth0` and `eth1` on the same subnet, ARP flux means that if another host sends packets to `10.10.0.2`, they may arrive at either `eth0` or `eth1`, and this changes with time. Once again, this is rather contrived, but also means that, for example, a private VPN IP will be responded to from the LAN a computer is on. Although this is desirable in some cases, it continues to seem like surprising default behaviour. The solution to this is also simple, a pair of kernel parameters, set by the following, resolve the issue.

```
sysctl -w net.ipv4.conf.all.arp_announce=1
sysctl -w net.ipv4.conf.all.arp_ignore=1
```

Chapter 4

Evaluation

This chapter will discuss the methods used to evaluate my project and the results obtained. The results will be discussed in the context of the success criteria laid out in the Project Proposal (Appendix D). This evaluation shows that a network using my method of combining Internet connections can see vastly superior network performance to one without. It will show the benefits to throughput, availability, and adaptability. The tests are performed on a Dell R710 Server with the following specifications:

CPU(s)	16 x Intel(R) Xeon(R) CPU X5667 @ 3.07GHz (2 Sockets)
Memory	6 x 2GB DDR3 ECC RDIMMS
Kernel	Linux 5.4 LTS

When presenting data, error bars are given of the Inter-Quartile Range (IQR) of the data, with the plotted point being the median.

4.1 Success Criteria

4.1.1 Flow Maintained

The results for whether a flow can be maintained during a single connection loss are achieved using an iperf3 UDP test. The UDP test runs at a fixed bitrate, and measures the quantity of datagrams lost in transit. Three tests will be performed on a proxy with two connections: both connections remain up, one connection remains up, and both connections are lost. To satisfy this success criteria, the single connection lost may have a small amount of loss, while losing both connections should terminate the test.

These results are given in figures 4.1, 4.2 and 4.3 respectively. The results are as expected: no connection loss handles the 1MB/s stream with no problems, and therefore no packets are lost, one connection loss causes slight packet loss (0.88%) but the test is able to continue, and a complete connection loss stalls the test. Given the consistent external IP, this shows that a flow can be maintained through a single connection loss, with only a small loss of packets. This level of packet loss represents some loss on a phone call that lasts approximately 45ms, after which the call continues gracefully. This satisfies the success criteria.

4.1.2 Bidirectional Performance Gains

To demonstrate that all performance gains are bidirectional, I will provide graphs both inbound and outbound to the client for each performance test of the core success criteria. This will sufficiently

```
Connecting to host X.X.X.X, port 5201
[  5] local X.X.X.Y port 43039 connected to X.X.X.X port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   4.00-5.00   sec   129 KBytes  1.05 Mbits/sec  91
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-5.00   sec   641 KBytes  1.05 Mbits/sec  0.000 ms  0/453 (0%)  sender
[  5]   0.00-5.04   sec   641 KBytes  1.04 Mbits/sec  0.092 ms  0/453 (0%)  receiver
```

Fig. 4.1 iperf3 UDP results with two stable connections (inbound).

```
Connecting to host X.X.X.X, port 5201
[  5] local X.X.X.Y port 49929 connected to X.X.X.X port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   4.00-5.00   sec   129 KBytes  1.05 Mbits/sec  91
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-5.00   sec   641 KBytes  1.05 Mbits/sec  0.000 ms  0/453 (0%)  sender
[  5]   0.00-5.04   sec   635 KBytes  1.03 Mbits/sec  0.115 ms  4/453 (0.88%)  receiver
```

Fig. 4.2 iperf3 UDP results with a single connections loss (inbound).

```
Connecting to host X.X.X.X, port 5201
[  5] local X.X.X.Y port 51581 connected to X.X.X.X port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   129 KBytes  1.05 Mbits/sec  91
```

Fig. 4.3 iperf3 UDP results with a total connection loss (inbound).

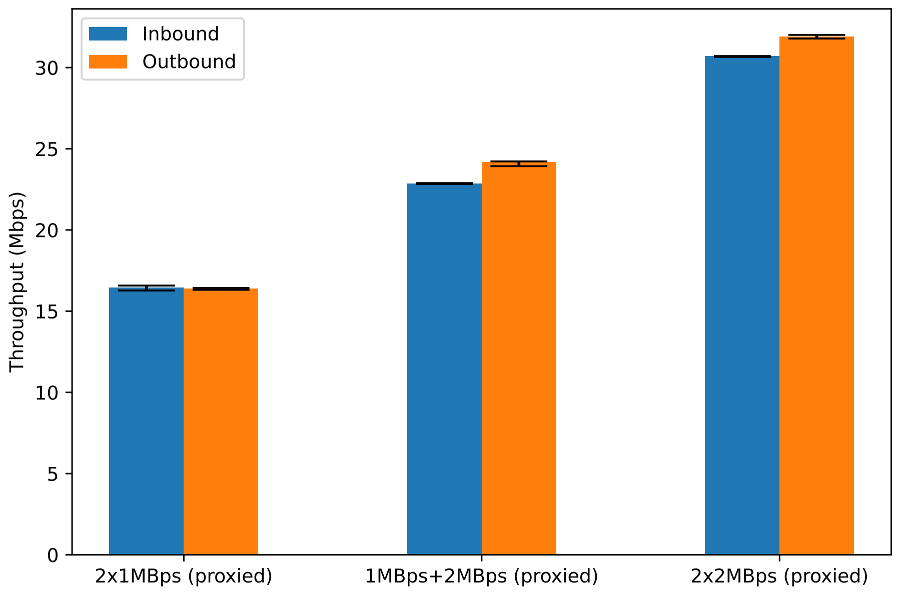


Fig. 4.4 Comparing the performance of packets inbound to the client to outbound from the client in three different test conditions.

show the performance gains in each case. Inbound tests occur with the test server running on the proxy client and the test client running outside, while outbound tests place the test server outside and the test client on the proxy client.

To demonstrate this somewhat succinctly, the same test will be executed both inbound and outbound, with each plotted as a series on a graph. To demonstrate that this requirement is satisfied for all cases, for each graph of results presented for the basic success criteria, the graph for the alternative direction will be provided in appendix C.

Figure 4.4 has two series for the same set of tests - one for the inbound (reaching in to the client, or download) performance and one for the outbound (the client reaching out, or upload) performance. The trend is consistent within a direction, however, there is a slight preference to outbound flows. This is due to the outbound flows being spread between interfaces, which avoids waiting for the kernel to finish locking an interface quite as often. In each case, both inbound and outbound performance satisfy the success criteria, so this is satisfied.

4.1.3 IP Spoofing

Demonstrating that the IP of the client can be set to the IP of the remote proxy is achieved, as each test in this evaluation relies on this fact. When allocating virtual machines to test on, the client is given the IP of the remote proxy. In the given network structure, the speed test server, remote proxy and local proxy are each connected to one virtual switch, which acts as a mock Internet. There is then a separate virtual switch, which connects an additional interface of the local proxy to the client. The IP addresses of the interfaces used in these tests are listed in Figure 4.5. The IP addresses of the public interfaces are represented by letters, as they use arbitrary public IP addresses to ensure no local network firewall rules impact the configuration.

It is shown that the client in this testing setup shares an IP address with the remote proxy. The details of this configuration are provided in Section 3.3. This satisfies the success criteria.

Machine	Interface	IP Address
Speed Test Server	eth0	*A*
Remote Proxy	eth0	*B*
Local Proxy	eth0	*C0*
	eth1	*C1*
	⋮	⋮
	ethN	*CN*
	eth{N+1}	192.168.1.1
Client	eth0	*B*

Fig. 4.5 The IP layout of the test network structure.

4.1.4 Security

Success for security involves providing security no worse than a standard connection. This is achieved by using Message Authentication Codes, Replay Protection and extra authenticated information for connection authentication, described in detail in Section 2.2. Further, Section 2.2.4 provides an argument that the proxying of packets can be made secure by operating in a secure overlay network, such as a VPN. This ensures that security can be maintained, regardless of changes in the security landscape, by composing my proxy with additional security software.

4.1.5 More Bandwidth over Two Equal Connections

To demonstrate that more bandwidth is available over two equal connections through this proxy than one without, I will compare the iperf3 throughput between the two cases. Further, I will provide a comparison point against a single connection of the combined bandwidth, as this is the maximum theoretical performance of combining the two lower bandwidth connections.

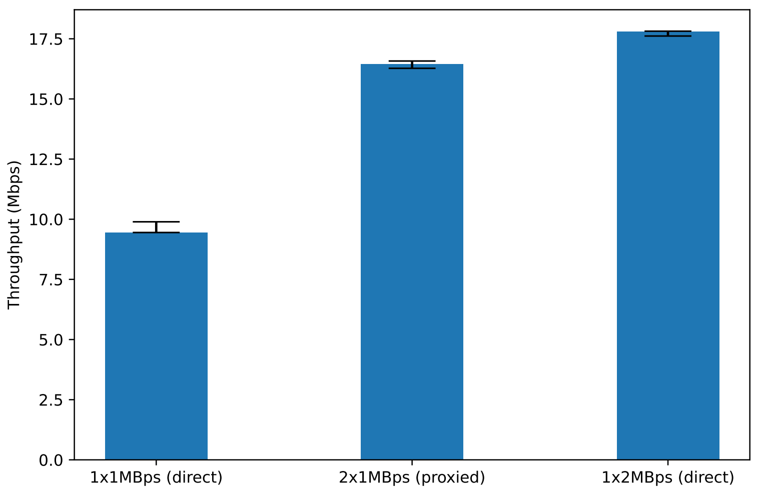
The results of these tests are given in Figure 4.6, for both a pair of 1MB/s connections and a pair of 2MB/s connections. To satisfy this success criteria, the proxied bar on each graph should exceed the throughput of the direct bar of equal bandwidth. It can be seen in both cases that this occurs, and thus the success criteria is met. The throughput far exceeds the single direct connection, and is closer to the single double bandwidth connection than the single equal bandwidth connection, demonstrating a good portion of the maximum performance is achieved (92.5% for the 1+1MB/s proxy, and 88.5% for the 2+2MB/s proxy).

4.2 Extended Goals

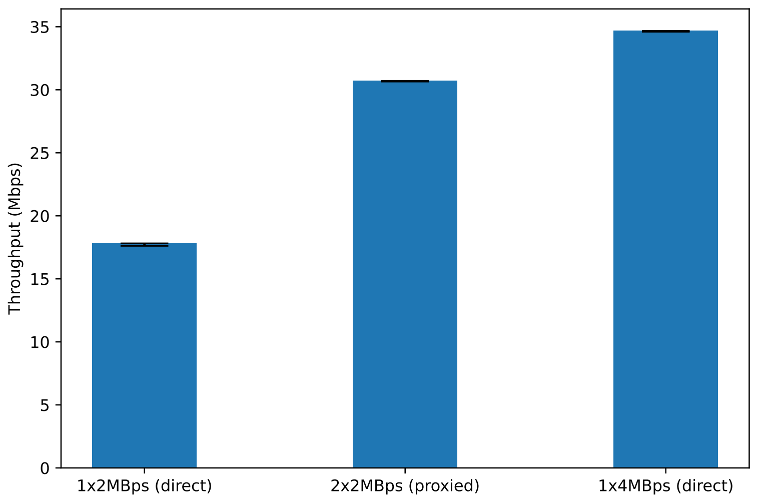
4.2.1 More Bandwidth over Unequal Connections

For showing improved throughput over connections which are not equal, three results will be compared. Connections of speed $x+x$, speeds $x+y$, and speeds $y+y$ will be shown, where $x<y$. To show that unequal connections exceed the performance of a pair of slower connections, the results for speeds $x+y$ should lie between $x+x$ and $y+y$. Further, to show that percentage throughput is invariant to the balance of connection throughput, the unequal connections should lie halfway between the two equal connection results.

Two sets of results are provided - one for 1MB/s and 2MB/s connections, and another for 2MB/s and 4MB/s connections. In both cases, it can be seen that the proxy with unequal connections lies

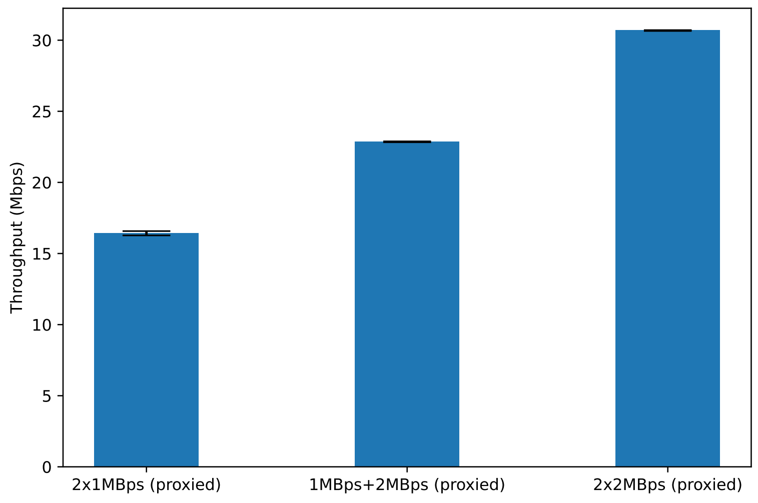


(a) Throughput of 1+1MB/s connections compared with 1MB/s and 2MB/s (inbound).

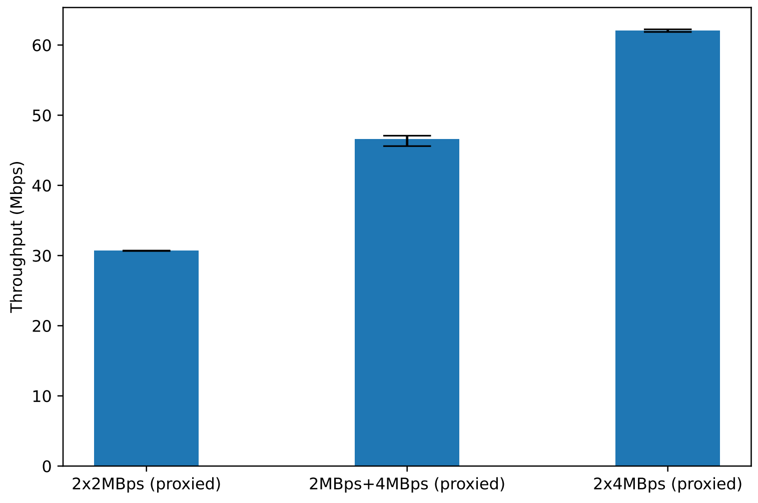


(b) Throughput of 2+2MB/s connections compared with 2MB/s and 4MB/s (inbound).

Fig. 4.6 Graphs showing that the throughput of two connections proxied lie between one connection of the same speed and one connection of double the speed



(a) Bandwidth of 1+2MB/s connections compared to 1+1MB/s connections and 2+2MB/s connections.



(b) Bandwidth of 2+4MB/s connections compared to 2+2MB/s connections and 4+4MB/s connections.

Fig. 4.7 Graphs to demonstrate that the proxy appropriately balances between imbalanced connections, resulting in near-maximal throughput.

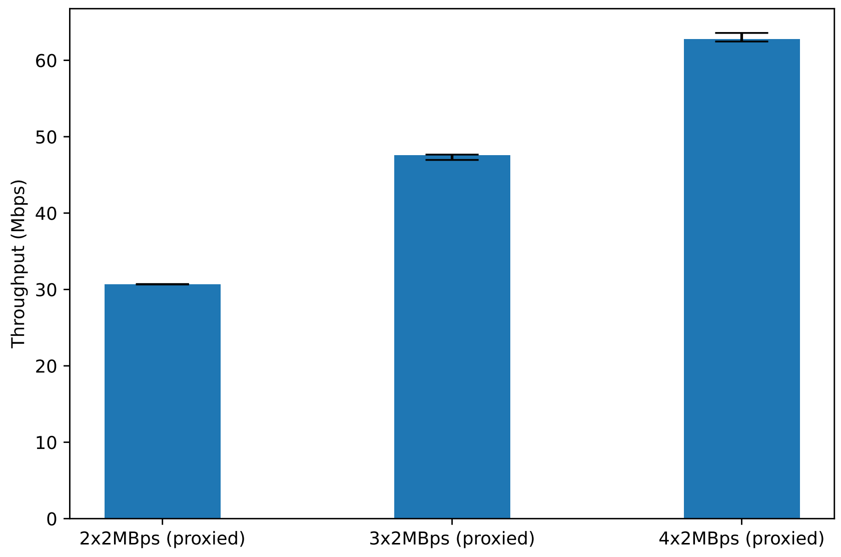


Fig. 4.8 Scaling of 2-4 equal bandwidth connections when combined.

between the equal connection proxies. Further, it can be seen that both unequal proxied connections lie approximately halfway between the equal pairs (74.4% of the maximum for 1+2MB/s, and 75.1% of the maximum for 2+4MB/s). This suggests that the proxy design is successful in being invariant to the static balance of connection throughput.

4.2.2 More Bandwidth over Four Equal Connections

This criteria expands on the scalability in terms of number of connections of the proxy. Specifically, comparing the performance of three connections against four. To fulfil this, the results for each of two, three and four connections are included on each graph. This allows the trend of performance with an increasing number of connections to begin being visualised, which is expanded upon further in Section 4.4.

Provided in Figure 4.8 are results for each of two, three and four combined 2MB/s connections. Firstly, it is clear that the proxy consisting of 4 connections exceeds the throughput of the proxy consisting of 3 connections. Secondly, it appears that a linear trend is forming. This trend will be further evaluated in Section 4.4, but suggests that the structure of the proxy suffers little loss in efficiency from adding further connections.

4.2.3 Bandwidth Variation

This criteria judges the adaptability of the congestion control system in changing network conditions. To test this, the bandwidth of one of the local portal's connections is varied during an iperf3 throughput test. Thus far, bar graphs have been sufficient to show the results of each test. In this case, as the performance should now be time sensitive, I will be presenting a line graph. Due to the nature of the time in these tests, producing consistent enough results to produce error bars was not feasible. The data is also smoothed across the x-axis with a 5-point moving average, to avoid intense fluctuations caused by the interface rate limiting.

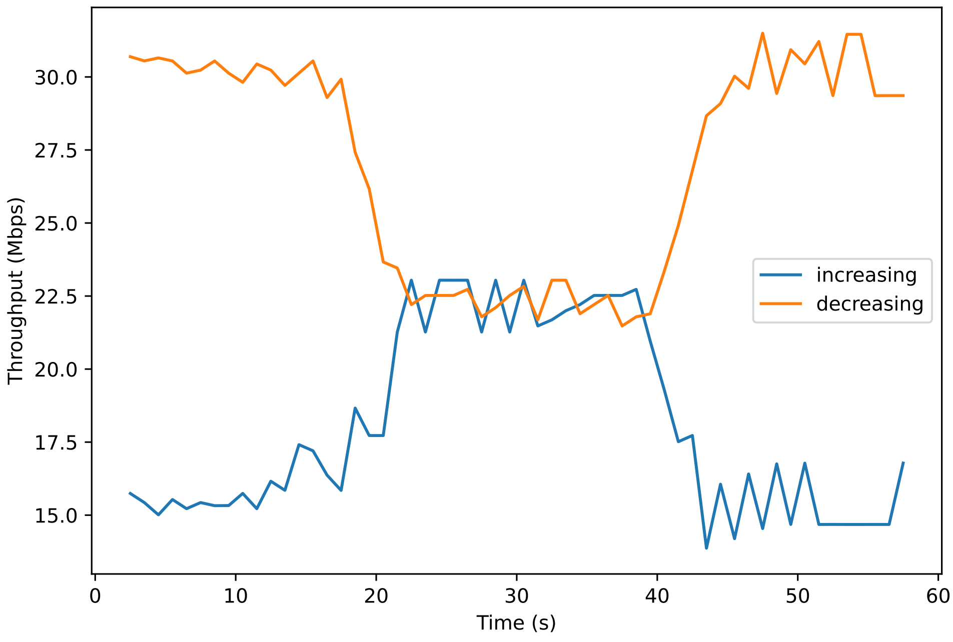


Fig. 4.9 Connection capacity increasing and decreasing over time. The decrease is from 2+2MB/s connections to 1+2MB/s connections, and the increase from 1+1MB/s connections to 1+2MB/s connections.

The criteria will be met if the following are true: the throughput begins at a constant rate; the throughput stabilises at a lower rate after the increase/decrease; the throughput returns to the original rate after the bandwidth returns.

The results are given in Figure 4.9. The decreasing series drops from 2+2MB/s connections, with a maximum throughput of 32Mbps, to 1+2MB/s connections, with a maximum throughput of 24Mbps. The increasing series increases from 1+1MB/s connections, with a maximum throughput of 16Mbps, to 1+2MB/s connections, with a maximum throughput of 24Mbps. The events occur at approximately the same time. The graph displays each series beginning at their constant rate, before converging at approximately 24Mbps in the center of the graph. Once the connection change is reversed, each series returns to its original throughput. This satisfies the success criteria for connection capacity changes.

4.2.4 Connection Loss

This criteria judges the ability of the proxy as a whole to handle a complete connection loss while maintaining proportional throughput, and later regaining that capacity when the connection becomes available again. As the proxy has redundant connections, it is feasible for this to cause a minimal loss of service. Unfortunately, losing a connection causes significant instability with the proxy, so this extended goal has not been met. This is due to the interactions between the proxy and the system kernel, where the proxy has very little control of the underlying TCP connection. With future work on UDP I am hopeful that this will be eventually satisfied, but it is not with the current implementation.

4.2.5 Single Interface Remote Portal

Similarly to Section 4.1.3, a remote portal with a single interface is employed within the standard testing structure for this section, using techniques detailed in Section 3.3. By altering the routing tables such that all local traffic for the remote portal is sent to the local portal via the proxy, excluding the traffic for the proxy itself, the packets can be further forwarded from the local portal to the client which holds that IP address. As the standard testing structure employs a remote portal with a single interface, it is shown in each test result that this is a supported configuration, and thus this success criteria is met.

4.2.6 Connection Metric Values

The extended goal of connection metric values has not been implemented. Instead, peers which only transfer data in one direction were implemented, which covers some of the use cases for metric values. Though metric values for connections would have been useful in some cases, they do not represent the standard usage of the software, and the added complexity of managing live peers was deemed unnecessary for the core software. Instead, I would consider providing a better interface to control the software externally, which would allow a separate piece of software to manage live peers. This has not been completed at this time.

4.3 Stretch Goals

4.3.1 UDP Proxy Flows

UDP flows are implemented, and provide a solid base for UDP testing and development. The present implementation of a New Reno imitating congestion control mechanism still has some implementation flaws, meaning that UDP is not yet feasible for use. However, the API for writing congestion control mechanisms is strong, and some of the future work suggested in Section 5.2 could be developed on this base, so that much is a success.

4.4 Performance Evaluation

The discussion of success criteria above used relatively slow network connections to test scaling in certain situations, while ensuring that hardware limitations have no impact on the tests. This section provides a brief analysis of how this solution would scale to providing a higher bandwidth connection, specifically by adding network connections.

The results of these tests are shown in Figure 4.10. Each of $1MB/s$, $2MB/s$ and $4MB/s$ capacity links are tested with 1 to 8 connections. The throughput demonstrated is largely linear, with a suggestion that eight $4MB/s$ connections are approaching the software's limits. This result is very promising, as it shows that the software can handle a large number of connections. While this is quite limiting for some higher download speed connections, many upload speeds are far slower, and would benefit from this quantity of links.

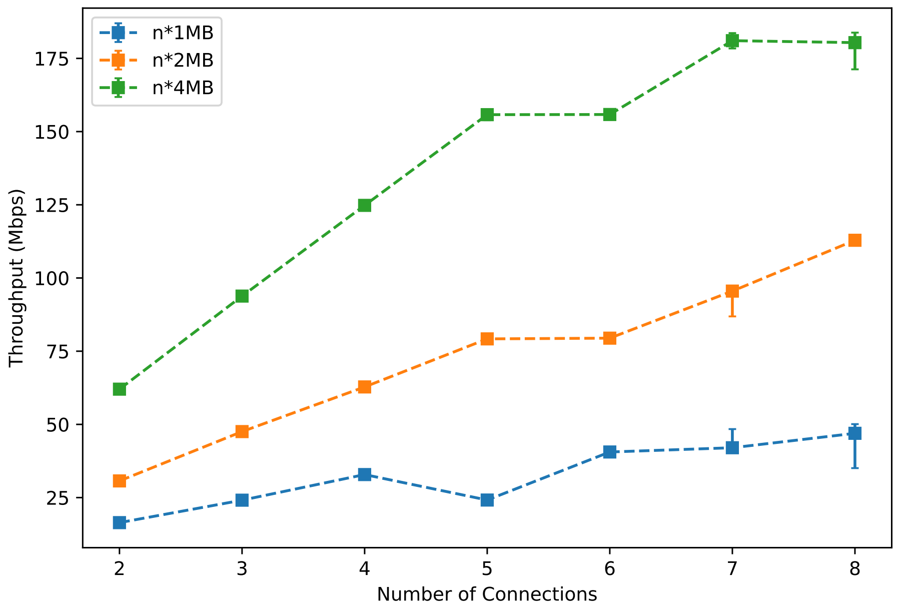


Fig. 4.10 Scaling of proxy throughput based on number of connections, for three speeds of connection.

Chapter 5

Conclusions

The software produced in this project provides a method of combining multiple Internet connections via a proxy, prioritising throughput and resilience in the resultant aggregate connection. The proxy provides a novel approach to combining Internet connections, responding well to very dynamic Internet connections. All of the core success criteria were met, along with many of the extended goals.

The multipath proxy built in this project provides an effective method to combine dynamic Internet connections, and it works in today's Internet. Though future work may make much of this redundant, the performance gains seen today are useful in many situations. As it becomes more common to see a variety of connections in homes, such as 5G, Low Earth Orbit and DSL, a method to combine these that dynamically adapts to the variability of the wireless connections can be a significant, practical benefit, especially in situations where gaining a single faster link is difficult.

5.1 Lessons Learnt

The lessons learned in this project fall in two classes: personal reflections on running this project, and lessons learnt if this proxy sees another implementation.

I learnt throughout this project the importance of producing a minimum viable product. Very early in the project, I produced a working proof of concept. Nearing the end of the project, once the design was mostly settled and with a view of how the program would be deployed, the code was refactored to produce a user friendly piece of software. This approach of fast development that did not commit early to a usage pattern served me very well with this project, as details of the deployment only became clear after some use.

Further, lessons were learnt on the quality of packages. A package being a part of the standard library for a language does not imply support or a full feature set, while packages from respected software companies can be superior.

On re-implementation of this work, more considerations should be made for the interface of the software. In particular, monitoring the current connections without a debugger is particularly difficult, and monitoring long term statistics is presently impossible. This compromise was made for code readability and clarity, increasing the likelihood of correct code, but does raise some issues for the network architects who implement this software.

Many of the lessons learnt relating to IP routing are detailed in Section 3.3, which would aid future implementations significantly, allowing the developer to focus only on what needs to occur in the application itself. Similarly, Figure 3.1 provides a comprehensive overview of the particularly complex dataflow within this application. These tools provide an effective summary of the information needed to implement this software again, reducing the complexity of such a new implementation, and allowing the developer to focus on the important features.

5.2 Future Work

Alternative methods of load balancing could take multipath proxies further. Having control of both proxies allows for a variety of load balancing mechanisms, of which congestion control is only one. An alternative method is to monitor packet loss, and use this to infer the maximum capacity of each link. These capacities can then be used to load balance packets by proportion as opposed to greedily with congestion control. This could provide performance benefits over congestion control by allowing the congestion control mechanisms of underlying flows to be better employed, while also having trade-offs with slower reaction to connection changes.

To increase performance, a kernel implementation of the proxy could be written. Kernel implementations avoid copying the packets between kernel- and user-space, as well as removing the cost of syscalls. This can increase maximum performance significantly, as well as reducing latency. These combine to make the software useful in more places, though enforcing platform compatibility to only that which uses the compatible kernel. Therefore, having kernel implementations maintain compatibility with a user-space implementation allows more systems to take advantage of the proxy.

Bibliography

Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. Wiley Pub, Indianapolis, IN, 2nd ed edition, 2008. ISBN 978-0-470-06852-6. OCLC: ocn192045774.

Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954, pages 119–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38979-5 978-3-642-38980-1. doi: 10.1007/978-3-642-38980-1_8. URL http://link.springer.com/10.1007/978-3-642-38980-1_8. Series Title: Lecture Notes in Computer Science.

Kent Beck, Mike Beedle, Arie van Bekkenum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001. URL http://agilemanifesto.org/.

Mike Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3), February 2021. URL https://tools.ietf.org/html/draft-ietf-quic-http-34.

Inc. Cloudflare. Cloudflare - The Web Performance & Security Company. URL https://www.cloudflare.com/.

Dragana Damjanovic. QUIC and HTTP/3 Support now in Firefox Nightly and Beta – Mozilla Hacks - the Web developer blog, April 2021. URL https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta.

Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. *IEEE TRANSACTIONS ON INFORMATION THEORY*, (2):11, 1983.

Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. ISBN 978-1-891562-46-4. doi: 10.14722/ndss.2017.23160. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

Jason A Donenfeld. wireguard fixes for 5.6-rc7, March 2020. URL https://lore.kernel.org/netdev/20200319003047.113501-1-Jason@zx2c4.com/.

Jason A Donenfeld and Kevin Milner. Formal Verification of the WireGuard Protocol. page 11.

Benjamin Dowling and Kenneth G. Paterson. A Cryptographic Analysis of the WireGuard Protocol. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, volume 10892, pages 3–21. Springer International Publishing, Cham, 2018. ISBN 978-3-319-93386-3 978-3-319-93387-0. doi: 10.1007/978-3-319-93387-0_1. URL http://link.springer.com/10.1007/978-3-319-93387-0_1. Series Title: Lecture Notes in Computer Science.

Dharani Govindan. Enabling QUIC in tip-of-tree, April 2020. URL https://groups.google.com/a/chromium.org/g/net-dev/c/5M9Z5mtvg_Y/m/iw9co1VrBQAJ?pli=1.

Mark Handley, Olivier Bonaventure, Costin Raiciu, Alan Ford, and Christoph Paasch. TCP Extensions for Multipath Operation with Multiple Addresses, March 2020. URL https://tools.ietf.org/html/rfc8684.

T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm, April 2012. URL https://tools.ietf.org/html/rfc6582.

Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency. In *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, volume 6011, page 60110H. International Society for Optics and Photonics, October 2005. doi: 10.1117/12.630496. URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/6011/60110H/Understanding-TCP-over-TCP--effects-of-TCP-tunneling-on/10.1117/12.630496.short.

Stephen Kent. IP Authentication Header, December 2005. URL https://tools.ietf.org/html/rfc4302.

Eric Kinnear. Boost performance and security with modern networking - WWDC 2020 - Videos, June 2020. URL https://developer.apple.com/videos/play/wwdc2020/10111/?time=644.

A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press series on discrete mathematics and its applications. CRC Press, Boca Raton, 1997. ISBN 978-0-8493-8523-0.

Ofcom. The performance of fixed-line broadband delivered to UK residential customers, May 2020. URL https://www.ofcom.org.uk/research-and-data/telecoms-research/broadband-research/home-broadband-performance-2019.

Avery Pennarun. How Tailscale works, March 2020. URL https://tailscale.com/blog/how-tailscale-works/.

Eve Schooler, Gonzalo Camarillo, Mark Handley, Jon Peterson, Jonathan Rosenberg, Alan Johnston, Henning Schulzrinne, and Robert Sparks. SIP: Session Initiation Protocol, June 2002. URL https://tools.ietf.org/html/rfc3261.

Linus Torvalds. Linux 5.6 - Linus Torvalds, March 2020. URL https://lore.kernel.org/lkml/CAHk-=wi9ZT7Stg-uSpX0UWQzam6OP9Jzz6Xu1CkYu1cicpD5OA@mail.gmail.com/.

Tina Tsou and Xiangyang Zhang. IPsec Anti-Replay Algorithm without Bit Shifting, January 2012. URL https://tools.ietf.org/html/rfc6479.

Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 99–112, USA, March 2011. USENIX Association.

Appendix A

Language Samples

```
#include...
struct Packet {
    size_t len; uint8_t* data;

    Packet(const uint8_t *input, size_t num_bytes) {
        len = num_bytes; data = new uint8_t[len];
        std::memcpy(data, input, len);
    };

    ~Packet() { delete[] data; }

    [[nodiscard]] std::string print() const {
        std::stringstream out;
        for (size_t i = 0; i < len; i++) {
            int temp = data[i];
            out << std::hex << temp << " ";
        }
        return out.str();
    }
};
template <class T> class ThreadSafeQueue {
    std::queue<T> _queue = std::queue<T>();
    std::mutex _mutex; std::condition_variable _cond;
public:
    ThreadSafeQueue() = default;
    void push(T item) {
        _mutex.lock(); _queue.push(item); _mutex.unlock();
        _cond.notify_one();
    }
    T pop() {
        while (true) {
            std::unique_lock<std::mutex> unique(_mutex);
            _cond.wait(unique);
            if (!_queue.empty()) {
                T out = _queue.front();
                _queue.pop();
                return out;
            }
        }
    }
};
int tun_alloc(const char *dev, short flags) {
    struct ifreq ifr{};
    int fd, err;
    if( (fd = open("/dev/net/tun" , O_RDWR)) < 0 ) {
        perror("Opening /dev/net/tun");
        return fd;
    }
```

Fig. A.1 A sample script written in C++ to collect packets from a TUN interface and print them from multiple threads

```
    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = flags;
    strncpy(ifr.ifr_name, dev, IFNAMSIZ);
    if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
        perror("ioctl(TUNSETIFF)");
        close(fd);
        return err;
    }
    return fd;
}
std::mutex print_lock;
void consumer(const int index, ThreadSafeQueue<Packet*> *queue) {
    std::cout << "thread " << index << "starting" << std::endl;

    while (!stop) {
        Packet *p = queue->pop();

        print_lock.lock();
        std::cout << "thread " << index << " received a packet with content `" << p->print()
         ↪  << "`" << std::endl;
        print_lock.unlock();

        delete p;
    }
}
int main() {
    int tun = tun_alloc("nc%d", IFF_TUN);
    auto queue = new ThreadSafeQueue<Packet*>();
    std::thread threads[10];
    for (int i = 0; i < 10; i++) {
        const int i_safe = i;
        threads[i] = std::thread ([i_safe, queue]() {
            consumer(i_safe, queue);
        });
    }
    std::thread reader([tun, queue]() {
        uint8_t buffer[1500];
        while (true) {
            int num_bytes = read(tun, &buffer, 1500);
            if (num_bytes != 0) {
                auto *packet = new Packet(buffer, num_bytes);
                queue->push(packet);
            }
        }
    });
}
```

```rust
use std::thread;
use tun_tap::{Iface, Mode};

#[derive(Debug)]
struct Packet {
    data: [u8; 1504],
}

fn main() {
    let (mut tx, rx) = spmc::channel();

    let iface = Iface::new("nc%d", Mode::Tun).expect("failed to create TUN device");

    let mut buffer = vec![0; 1504];

    for i in 0..10 {
        let rx = rx.clone();
        thread::spawn(move || {
            let packet: Packet = rx.recv().unwrap();
            println!("Thread {}: {:?}", i, packet);
        });
    }

    for _ in 0..500 {
        iface.recv(&mut buffer).unwrap();
        let mut packet = Packet{ data: [0; 1504] };

        for i in 0..1504 {
            packet.data[i] = buffer[i];
        }

        tx.send(packet).unwrap();
    }
}
```

Fig. A.2 A sample script written in Rust to collect packets from a TUN interface and print them from multiple threads

```go
package main

import (
        "fmt"
        "github.com/pkg/taptun"
        "os"
        "os/signal"
        "syscall"
)

type Packet struct {
        Data []byte
}

func main() {
        tun, err := taptun.NewTun("nc%d")
        if err != nil { panic(err) }

        inboundPackets := make(chan Packet, 128)

        go func() {
                bufferSize := 1500
                buffer := make([]byte, bufferSize)

                for {
                        read, err := tun.Read(buffer)
                        if err != nil { panic(err) }

                        if read == 0 { panic("0 bytes read!") }

                        p := Packet{}
                        p.Data = make([]byte, read)
                        copy(p.Data, buffer)

                        inboundPackets <- p
                }
        }()

        for i := 0; i < 10; i++ {
                i := i
                go func() {
                        for {
                                p := <-inboundPackets
                                fmt.Printf("Reader %d: %v\n", i, p)
                        }
                }()
        }
}
```

Fig. A.3 A sample script written in Go to collect packets from a TUN interface and print them from multiple threads

Appendix B

Layered Security Packet Diagrams

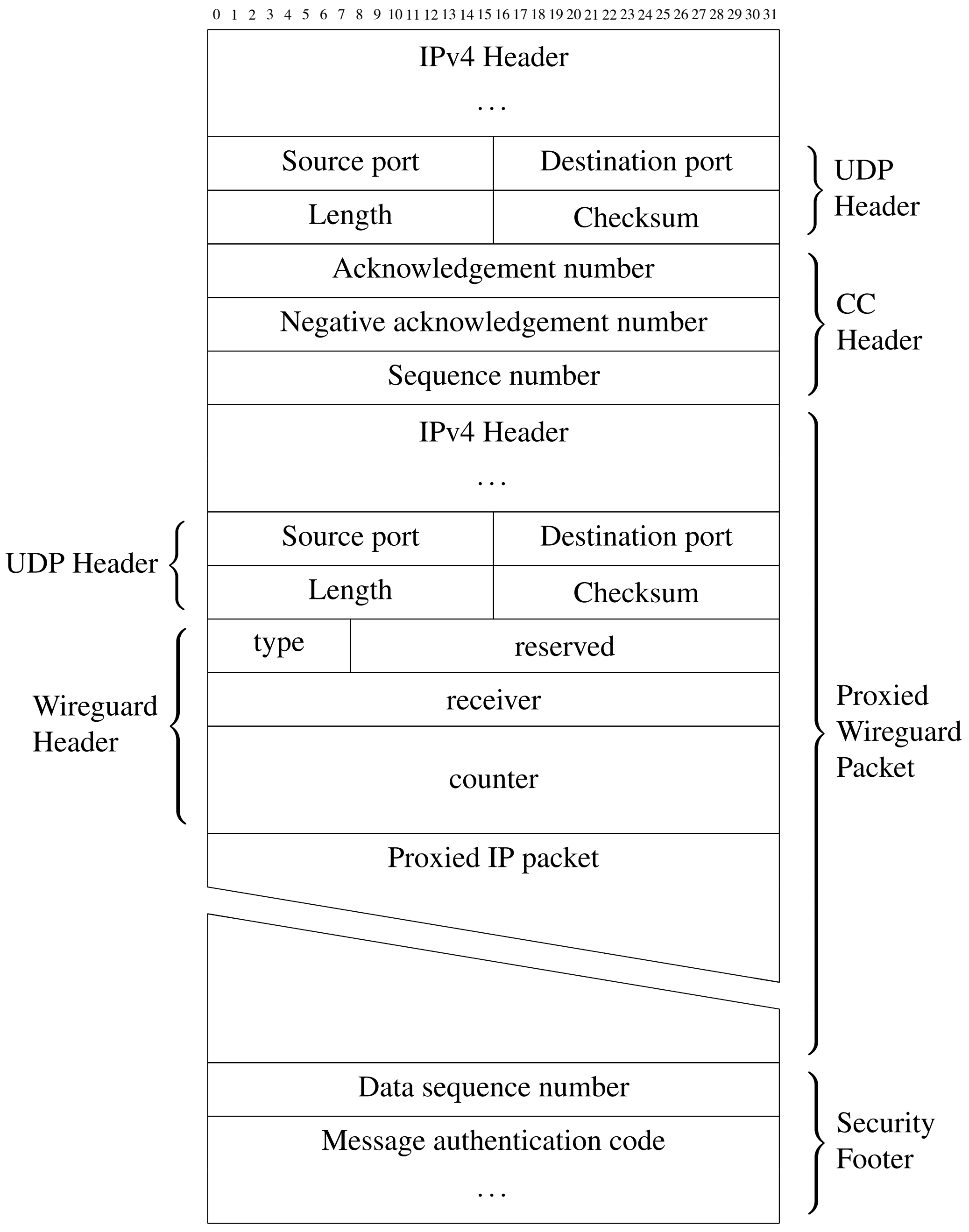


Fig. B.1 Packet structure for a configuration with a Wireguard client behind my multipath proxy.

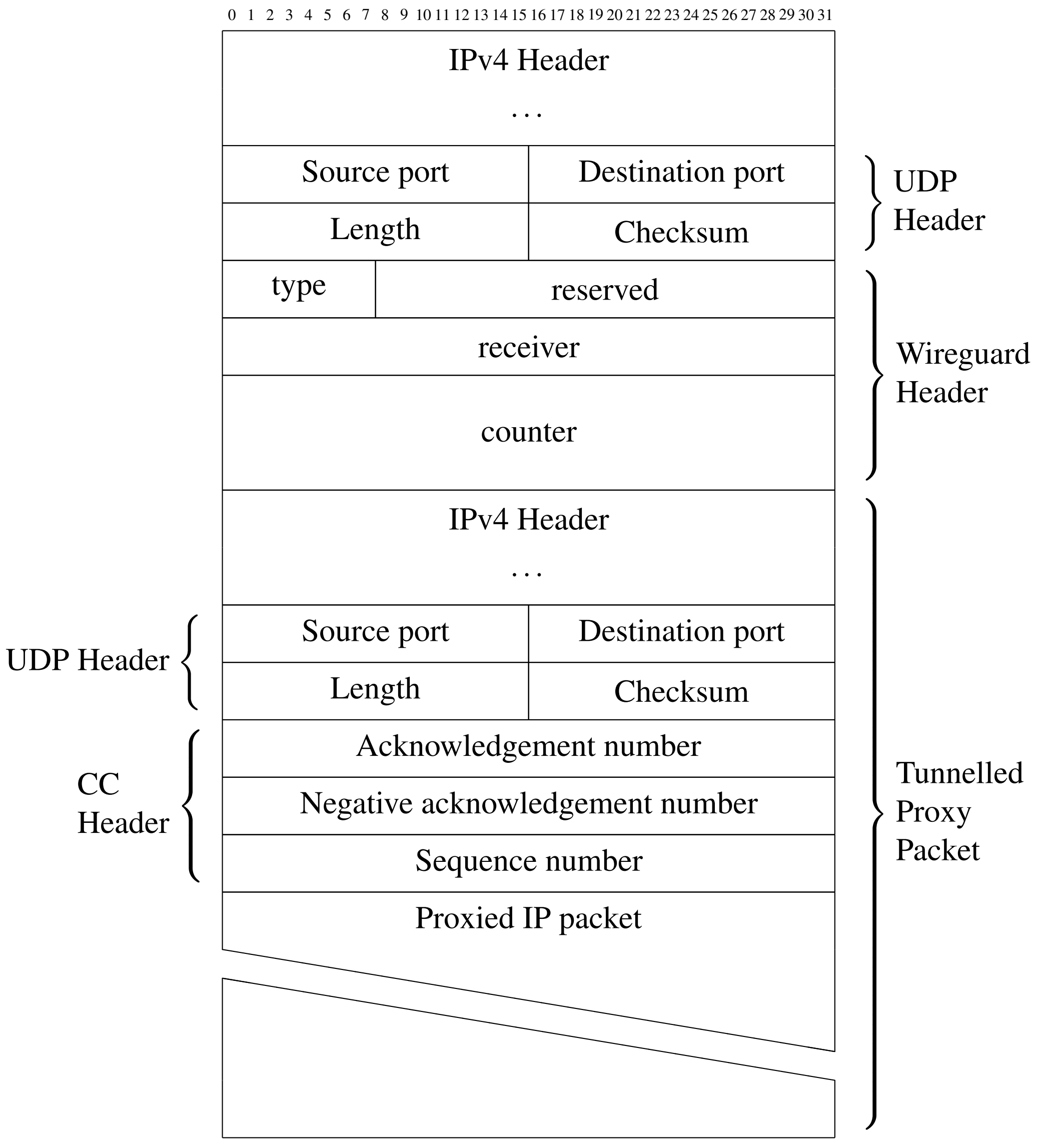


Fig. B.2 Packet structure for a configuration with a Wireguard client in front of my multipath proxy.

Appendix C

Outbound Graphs

```
Connecting to host X.X.X.Y, port 5201
[  5] local X.X.X.X port 53587 connected to X.X.X.Y port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   4.00-5.00   sec   129 KBytes  1.05 Mbits/sec  91
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-5.00   sec   641 KBytes  1.05 Mbits/sec  0.000 ms  0/453 (0%)  sender
[  5]   0.00-5.04   sec   641 KBytes  1.04 Mbits/sec  0.070 ms  0/453 (0%)  receiver
```

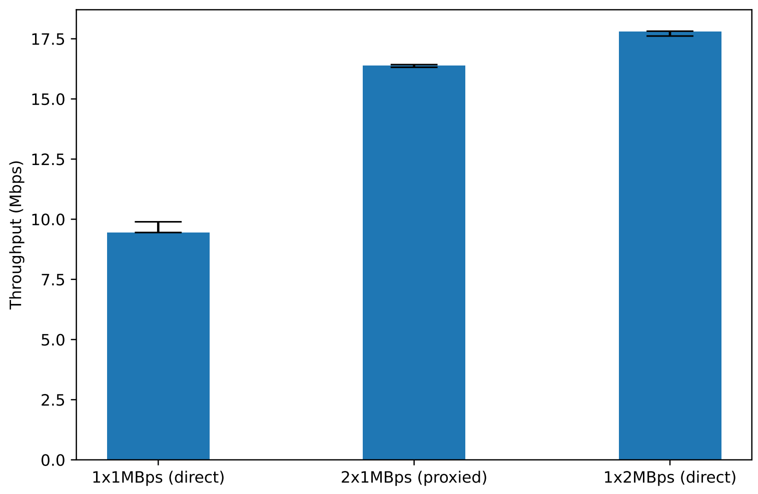
Fig. C.1 iperf3 UDP results with two stable connections (outbound).

```
Connecting to host X.X.X.Y, port 5201
[  5] local X.X.X.X port 38793 connected to X.X.X.Y port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   4.00-5.00   sec   127 KBytes  1.04 Mbits/sec  90
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-5.00   sec   641 KBytes  1.05 Mbits/sec  0.000 ms  0/453 (0%)  sender
[  5]   0.00-5.04   sec   635 KBytes  1.03 Mbits/sec  0.086 ms  4/453 (0.88%)  receiver
```

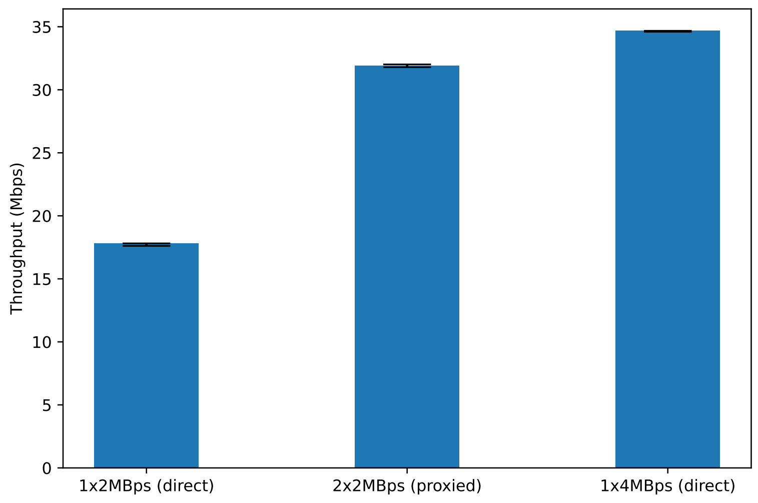
Fig. C.2 iperf3 UDP results with a single connections loss (outbound).

```
Connecting to host X.X.X.Y, port 5201
[  5] local X.X.X.X port 35549 connected to X.X.X.Y port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   1.00-2.00   sec   127 KBytes  1.04 Mbits/sec  90
[  5]   2.00-3.00   sec   129 KBytes  1.05 Mbits/sec  91
[  5]   3.00-4.00   sec   127 KBytes  1.04 Mbits/sec  90
```

Fig. C.3 iperf3 UDP results with a total connection loss (outbound).



(a) Throughput of 1+1MB/s connections compared with 1MB/s and 2MB/s (outbound).



(b) Throughput of 2+2MB/s connections compared with 2MB/s and 4MB/s (outbound).

Fig. C.6 Graphs showing that the throughput of two connections proxied lie between one connection of the same speed and one connection of double the speed

Appendix D

Project Proposal

Computer Science Tripos

Part II Project Proposal Coversheet

Please fill in Part 1 of this form and attach it to the front of your Project Proposal.

Part 1

Name:	Jake Hillion	CRSID:	jsh77
College:	Queens'	Overseers: (Initials)	AWM & AV
Title of Project:	A Multi-Path Bidirectional Layer 3 Proxy		
Date of submission:	22/10/2020	Will Human Participants be used?	No
Project Originator:	Jake Hillion	Signature:	--
Project Supervisor:	Mike Dodson	Signature:	--
Directors of Studies:	Neil Lawrence	Signature:	--
Special Resource Sponsor:		Signature:	--
Special Resource Sponsor:		Signature:	--

Above signatures to be obtained by the Student

Part 2

Overseer Signature 1: ---

Overseer Signature 2: ---

Overseers signatures to be obtained by Student Administration.

Overseers Notes:

Part 3

SA Date Received:		SA Signature Approved:	

Introduction and Description of the Work

This project attempts to combine multiple heterogeneous network connections into a single virtual connection, which has both the combined speed and the maximum resilience of the original connections. This will be achieved by inserting a Local Portal and a Remote Portal into the network path, as shown in Figure 1. While there are existing solutions that combine multiple connections, they prioritise one of resilience or speed over the other; this project will attempt to show that this trade-off can be avoided.

The speed focus of this software is achieved by providing a single virtual connection which aggregates the speed of the individual connections. As this single connection is all that's made visible to the client, all applications and protocols can benefit from the speed benefits, as they require no knowledge of how their packets are being split. As an example, a live video stream that only uses one flow will be able to use the full capacity of the virtual connection.

The resilience focus provides similar benefits, in that the virtual connection conceals the failing of any individual network connections from the client and applications. This again means that applications and protocols not built to handle a network failover can benefit from the resilience provided by this solution. An example is a SIP call continuing without a redial.

This system is useful in areas where multiple low bandwidth connections are available, but not a single higher bandwidth connection. This is often the case in rural areas in the UK. It will also be useful in areas with diverse connections of varying reliability, such as a home with both DSL and wireless connections, which may become more common with the advent of 5G and LEO systems such as Starlink. The lack of requirement for vendor support allows for this mixture of connections to be supported.

Some existing attempts to solve these problems, and the shortfalls of each solution, are summarized below:

- Failover: All existing flows must be restarted when failover occurs. There is no speed benefit over having a single connection.

- Session Based Load Balancing: All flows on a failed connection must be restarted. Speed benefit varies between applications, but is excellent in ideal circumstances. This solution is less effective when parameters of the connections vary with time, as with wireless connections. Further, advanced policies can be required on an application level to achieve the best speed.

- Application Support: Many modern protocols that are designed with mobile devices in mind can already handle IP changes (e.g. switching from WiFi to 4G). This allows these applications to handle situations such as Failover (above), as they treat it like any other network change. The downside of requiring application support is older protocols, such as SIP, for which resilience needs to be gained at a higher level.

- MultiPath TCP: MPTCP works best with multiple interfaces on each device that is using it, e.g. a 4G and WiFi connection on a mobile device. This is due to a device on a NAT with access to two WAN connections having no direct knowledge of this. It also requires support on both ends, which isn't common yet (MPTCP is not yet mainlined in the Linux kernel). Further, many modern applications are moving away from TCP in favour of lighter UDP protocols, which wouldn't benefit from MPTCP support.

- OpenVPN over MultiPath TCP: This allows both non-TCP based protocols, and clients that don't support MPTCP to benefit (if it's implemented network wide). Head of line blocking becomes more of an issue when passing multiple entirely different applications over a VPN, as any application can block any other. OpenVPN also adds a lot of unnecessary overhead if a network wide VPN would not otherwise be used.

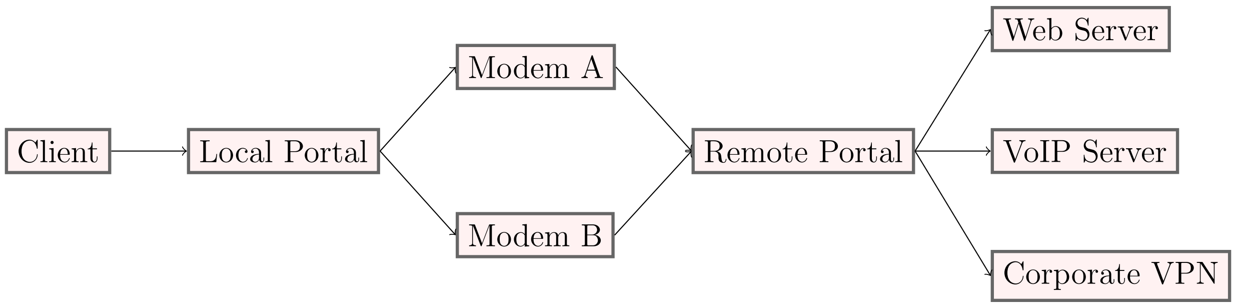


Figure 1: A network applying this proxy

By providing congestion control over each interface and therefore being able to share packets without bias between connections, this project should provide a superior solution for load balancing across heterogeneous and volatile network connections. An example of a client using this is shown in Figure 1. This solution is highly flexible, allowing the client to be a NAT Router with more devices behind it, or the flows from the Local Portal to the Remote Portal being tunnelled over a VPN.

Starting Point

I have spent some time looking into the shortfalls and benefits of the available methods for combining multiple Internet connections. The Part IB course *Computer Networking* has provided the background information for this project. I have significant experience with Go, though none with lower level networking. I have no experience with Rust, and my C++ experience is limited to the Part IB course *Programming in C and C++*.

While I am not aware of any existing software that accomplishes the task that I propose, Wireguard performs a similar task of tunnelling between a local and remote node, has a well regarded interface, and is a well structured project, providing both inspiration and an initial model for the structure of my project.

Substance and Structure of the Project

The system will involve load balancing multiple congestion controlled flows between the Local Portal and the Remote Portal. The Local Portal will receive packets from the client, and use load balancing and congestion control algorithms to send individual packets along one of the multiple available connections to the Remote Portal, which will extract the original packets and forward them along a high bandwidth connection to the wider network.

To achieve this congestion control, I will initially use TCP flows, which include congestion control. However, TCP also provides other guarantees, which will not benefit this task. For this reason, the application should be structured in such a way that it can support alternative protocols to TCP. An improved alternative is using UDP datagrams with a custom congestion control protocol, that only guarantees congestion control as opposed to packet delivery. Another alternative solution would be a custom IP packet with modified source and destination addresses and a custom preamble. Having a variety of techniques available would be very useful, as each of these has less overhead than the last, while also being less likely to work with more complicated network setups.

When the Local Portal has a packet it wishes to send outbound, it will place the packet and some additional security data in a queue. The multiple congestion controlled links will each be consuming from this queue when they are not congested. This will cause greedy load balancing, where each connection takes all that it can get from the packet queue. As congestion control algorithms adapt

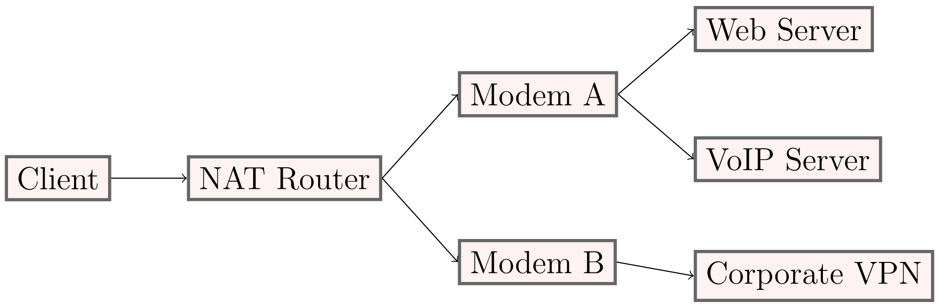


Figure 2: A network with a NAT Router and two modems

to the present network conditions, this load balancing will alter the balance between links as the capacity of each link changes.

Security is an important consideration in this project. Creating a multipath connection and proxies in general can create additional attack vectors, so I will perform a review of some existing security literature for each of these. However, as the tunnel created here transports entire IP packets, any security added by the application or transport layer will be maintained by my solution.

Examples are provided showing the path of a packet with standard session based load balancing, and with this solution applied:

Session Based Load Balancing

A sample network is provided in Figure 2.

1. NAT Router receives the packet from the client.
2. NAT Router uses packet details and Layer 4 knowledge in an attempt to find an established connection. If there is an established connection, the NAT Router allocates this packet to that WAN interface. Else, it selects one using a defined load balancing algorithm.
3. NAT Router masquerades the source IP of the packet as that of the selected WAN interface.
4. NAT Router dispatches the packet via the chosen WAN interface.
5. Destination server receives the packet.

This Solution

A sample network is provided in Figure 1.

1. Local Portal receives the packet from the client.
2. Local Portal wraps the packet with additional information.
3. Local Portal sends the wrapped packet along whichever connection has available capacity.
4. Wrapped packet travels across the Internet to the Remote Portal.
5. Remote Portal receives the packet.
6. Remote Portal dispatches the unwrapped packet via its high speed WAN interface.
7. Destination receives the packet.

Success Criteria

1. Demonstrate that a flow can be maintained over two connections of equal bandwidth with this solution if one of the connections becomes unavailable.

2. Any and all performance gains stated below should function bidirectionally (inbound/outbound to/from the client).

3. Allow the network client behind the main client to treat its IP address on the link to the Local Portal as the IP of the Remote Portal.

4. Provide security that is no worse than not using this solution at all.

5. Demonstrate that more bandwidth is available over two connections of equal bandwidth with this solution than is available over one connection without.

Extended Goals

1. Demonstrate that more bandwidth is available over two connections of unequal bandwidth than is available over two connections of equal bandwidth, where this bandwidth is the minimum of the unequal connections.

2. Demonstrate that more bandwidth is available over four connections of equal bandwidth than is available over three connections of equal bandwidth.

3. Demonstrate that if the bandwidth of one of two connections increases/decreases, the bandwidth available adapts accordingly.

4. Demonstrate that if one of two connections is lost and then regained, the bandwidth available reaches the levels of before the connection was lost.

5. My initial design requires the Remote Portal to have two interfaces: one for communicating with the Local Portal, and one for communicating with the wider network. This criteria is achieved by supporting both of these actions over one interface.

6. Support a metric value for connections, such that connections with higher metrics are only used for load balancing if no connection with a lower metric is available.

Stretch Goals

1. Provide full support for both IPv4 and IPv6. This includes reaching the Remote Portal over IPv6 but proxying IPv4 packets, and vice versa.

2. Provide a UDP based solution of tunnelling the IP packets which exceeds the performance of the TCP solution in the above bandwidth tests.

3. Provide an IP based solution of forwarding the IP packets which exceeds the performance of the UDP solution in the above bandwidth tests.

Although these tests will be performed predominantly on virtual hardware, I will endeavour to replicate some of them in a non-virtual environment, though this will not be a part of the success criteria.

Timetable and Milestones

12/10/2020 - 1/11/2020 (Weeks 1-3)

Study Go, Rust and C++'s abilities to read all packets from an interface and place them into some form of concurrent queue. Research the positives and negatives of each language's SPMC and MPSC queues.

Milestone: Example programs in each language that read all packets from a specific interface and place them into a queue, or a reason why this isn't feasible. A decision of which language to use for the rest of the project, based on these code segments and the status of SPMC queues in the language.

02/11/2020 - 15/11/2020 (Weeks 4-5)

Set up the infrastructure to effectively test any produced work from this point onwards.

Milestone: A virtual router acting as a virtual Internet for these tests. 3 standard VMs below this level for each: the Local Portal, the Remote Portal and a speed test server to host iPerf3. Behind the Local Portal should be another virtual machine, acting as the client to test the speed from. Backups of this setup should also have been made.

16/11/2020 - 29/11/2020 (Weeks 6-7)

This section should focus on the security of the application. This would include the ability for someone to maliciously use a Remote Portal to perform a DoS attack. Draft the introduction chapter.

Milestone: An analysis of how the security of this solution compares, both with other multipath solutions and a network without any multipath solution applied. A drafted introduction chapter.

30/11/2020 - 20/12/2020 (Weeks 8-10)

Implementation of the transport aspect of the Local Portal and Remote Portal. The first data structure for transport should also be created. This does not include the load sharing between connections - it is for a single connection. To enable testing, this will also require the setup of configuration options for each side. At this stage, it would be reasonable for the Remote Portal to require two different IPs - one for server communication, and one as the public IP of the Local Router. The initial implementation should use TCP, but if time is available, UDP with a custom datagram should be explored for reduced overhead.

Milestone: A piece of software that can act either as the Local Portal or Remote Portal based on configuration. Any IP packets sent to the Local Portal should emerge from the Remote Portal.

21/12/2020 - 10/01/2021 (Weeks 11-13)

Create mock connections for tests that support variable speeds, a list of packet numbers to lose and a number of packets to stop handling packets after. Finalise the introduction chapter. Produce the first draft of the preparation chapter.

Milestone: Mock connections and tests for the existing single transport. A finalised introduction chapter. A draft of the preparation chapter.

11/01/2021 - 07/02/2021 (Weeks 14-17)

Implement the load balancing between multiple connections for both servers. At this point, connection losses should be tested too. The progress report is due soon after this work segment, so that should be completed in here.

Milestone: The Local Portal and Remote Portal are capable of balancing load between multiple connections. They can also suffer a network failure of all but one connection with minimal packet loss. The progress report should be prepared.

08/02/2021 - 21/02/2021 (Weeks 18-19)

Finalise the drafted preparation chapter. Draft the implementation chapter. Produce a non-exhaustive list of graphs and tests that should be included in the evaluation section.

Milestone: Completed preparation chapter. Drafted implementation chapter. A plan of data to gather to back up the evaluation section.

22/02/2021 - 21/03/2021 (Weeks 20-23)

Finalise the implementation chapter. Gather the data required for graphs. Draft the evaluation chapter. Draft the conclusions chapter.

Milestone: Finalised implementation chapter. Benchmarks and graphs for non-extended success criteria complete and added. First complete dissertation draft handed to DoS and supervisor for feedback.

22/03/2021 - 25/04/2021 (Weeks 24-28)

Flexible time: divide between re-drafting dissertation and adding additional extended success criteria features, with priority given to re-drafting the dissertation.

Milestone: A finished dissertation and any extended success criteria that have been completed.

26/04/2021 - 09/05/2021 (Weeks 29-30)

New additions freeze. Nothing new should be added to either the dissertation or code at this point.

Milestone: Bug fixes and polishing.

10/05/2021 - 14/05/2021 (Week 31)

The project should already be submitted a week clear of the deadline, so this week has no planned activity.

Resources Required

- Personal Computer (AMD R9 3950X, 32GB RAM)
- Personal Laptop (AMD i7-8550U, 16GB RAM)

Used for development without requiring the lab. Testing this application will require extended capabilities, which would not be readily available on shared systems.

- Virtualisation Server (2x Intel Xeon X5667, 12GB RAM)
- Backup Virtualisation Server (2x Intel Xeon X5570, 48GB RAM)

A virtualisation server allows controlled testing of the application, without any packets leaving the physical interfaces of the server.

I accept full responsibility for the above 4 machines and I have made contingency plans to protect myself against hardware and/or software failure. All resources will be backed up according to the 3-2-1 rule. This would allow me to migrate development and/or testing to the cloud if needed.

Go(Lang) code written will use a version later than that available on the MCS, as the version currently on the MCS (1.10) does not support Go Modules. Rust is not available on the MCS at the time of writing. This can be managed by using personal machines or cloud machines accessed via the MCS.