UNIVERSITY OF CAMBRIDGE

A Multi-Path Bidirectional Layer 3 Proxy

Jake Hillion

Department of Computer Science
University of Cambridge

This dissertation is submitted for the degree of
Bachelor of Arts

Queens' College April 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
April 2021

Proforma

jsh77: Fill in closer to the time.

Table of contents

List of figures

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.

1.1 Existing Work

1.1.1 MultiPath TCP (MPTCP)

MultiPath TCP (Handley et al., 2020) is an extension to the regular Transmission Control Protocol, allowing the creation of subflows. MultiPath TCP 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, MultiPath TCP seems like a solution to the aims of this project. However, it suffers for three reasons: the rise of UDP protocols, device knowledge of interfaces, and legacy devices.

Although many UDP protocols have been around for a long time, using UDP protocols as a replacement for previously TCP protocols is a newer effort. An example of an older

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

protocol is SIP (Schooler et al., 2002), still widely used for VoIP calls. This is an example of a UDP protocol that would benefit particularly from increased resilience to Internet connection outage. For a replacement TCP protocol, HTTP/3 (Bishop, 2021) is one of the largest. Previously, HTTP requests have been sent over TCP connections, but HTTP/3 switches this to a UDP protocol. As such, HTTP requests are moving away from benefiting from MPTCP.

Secondly is the reliance of MPTCP on devices having 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 effectively, as it has knowledge of both Internet connections, and can create subflows appropriately. However, consider instead a tablet with only a WiFi interface, but connected via NAT to a router with two WAN interfaces. In this case, the tablet will believe that it only has one connection to the Internet, while actually being able to take advantage of two. This is a problem that is difficult to solve at the client level, suggesting that it 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 that legacy devices also connect to, the legacy devices see little benefit if they could utilise multiple connections. Providing an infrastructure level solution 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 (Virtual Private Network) 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.

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 suffer many of the same problems as will occur in my software, such as replay attacks, so the Wireguard implementation of such protections will be considered throughout. Finally, Wireguard provides an implementation in Go, which will be a useful reference for Layer 3 networking in Go.

1.2 Aims

This project aims to provide a method of combining multiple Internet connections, possibly heterogeneous, and possibly of dynamic capacity. When combining Internet connections,

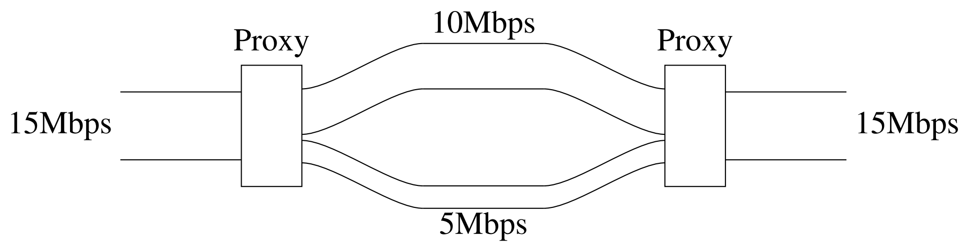


Fig. 1.1 A high level overview of the bottlenecks that are combined in this solution.

there are three main measures that one can prioritise: throughput, resilience, and latency. This project aims to provide throughput and resilience at the cost of latency. By using a layer 3 proxy, 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 insertion of the proxy combining these bottlenecks 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.

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 Risk Analysis

Any connection between two computers presents a set of security risks. A proxy consists of both these risks, and further, which I will present and discuss in this section. Firstly, this focuses on layered security. This is the case of the Local Portal and Remote Portal, with everything in between, being viewed as an Internet connection. The focus is on how the risks 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.

Secondly, this section focuses on the connections between the Local Portal and the Remote Portal. This focuses primarily on the risk of accepting and transmitting a packet that is not intended, or sending packets to an unintended recipient.

These security problems will be considered in the context of the success criteria: provide security no worse than a standard connection. That is, the security should be identical or stronger than the threats in the first case, and provide no additional vectors of attack in the second.

2.1.1 Higher Layer Security

This application proxies entire IP packets, so is a Layer 3 solution. As such, the goal is to maintain the same guarantees that one would normally expect at layer 3, for higher layers to build off of. At layer 3, none of anonymity, integrity, privacy or freshness are guaranteed, so it is up to the application to provide its own security guarantees. As such, maintaining the same level of security for applications can be achieved by ensuring that the the packets which leave one side of a proxy are a subset of the packets that entered the other side.

This ensures that guarantees managed by layers above layer 3 are maintained. Regardless of whether a user is accessing insecure websites over HTTP, running a corporate VPN connection or sending encrypted emails, the security of these applications will be unaltered. Further, this allows other guarantees to be managed, including reliable delivery with TCP.

2.1.2 Portal to Portal Communication

Denial of Service

Proxying packets in this way provides a new method of Denial of Service. If an attacker can convince either portal to send them a portion of the packets due for the other portal, the packet loss of the overall connection from the perspective of the other portal will increase by an equivalent amount. For example, if a bad actor can convince the remote portal to send them 50% of the good packets, and previously the packet loss was at 0.2%, the packet loss will increase to 50.1%.

This is of particular concern for flows carried by the proxy that use loss based congestion control. In such a case, for every 2 packets a TCP flow sends, it will lose one on average. This means that the window size will be unable to grow beyond one with NewReno congestion control. As such, the performance of these flows will be severely negatively impacted.

However, even if only 25% of the packets are lost, NewReno would still fail to increase the window size past 3. This demonstrates that an attacker with even a slower connection than you can have a significant impact on connection performance.

Privacy

Though the packets leaving a modem have no reasonable expectation of privacy, having the packets enter the Internet at two points does present more points at which a packet can be read. For example, if the remote portal lies in a data center, the content and metadata of packets can be sniffed in either the data center or at the physical connections. However, this is equivalent to your packets taking a longer route through the Internet, with more hops. Therefore, comparatively, this is not worse.

Further, if an attacker convinces the Remote Portal that they are a valid connection from the Local Portal, a portion of packets will be sent to them. However, as a fortunate side effect, this method to attempt sniffing would cause a significant Denial of Service to any congestion controlled links based on packet loss, as shown in the previous segment. Therefore, as long as it is ensured that each packet is not sent to multiple places, privacy should be maintained at a similar level to simple Internet access, given that an eavesdropper using this active eavesdropping method will be very easy to detect.

Cost

In many cases, the remote portal will be taking advantage of a cloud instance, for the extremely high bandwidth and well peered connections available at a reasonable price. Cloud instances are often billed per unit of outbound traffic, and as such, an attacker could cause a user to carry a high cost burden by forcing their remote portal to transmit more outbound data. This should be avoided by ensuring that the packets transmitted out are both from the local portal and fresh.

2.2 Security

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 means that the object has not been altered since the last authorised modification, that is the transmission from the other portal. Freshness details that the object has not been used for this purpose before.

2.2.1 Message Authentication

To provide integrity and freshness for each message, I evaluate two choices: Message Authentication Codes (MACs) or Digital Signatures. A MAC combines the data with a shared key using a specific method, before using a one-way hash function to generate a message authentication code, and thus the result is only verifiable by someone with the same private key (Menezes et al., 1997, pp. 352). Producing a digital signature for a message uses the private key in public/private keypair to produce a digital signature for a message, proving that the message was produced by the owner of the private key, which can be verified by anyone with the public key (Anderson, 2008, pp. 147-149). In both cases, the message authentication code is appended to the message, such that the integrity and authenticity of the message can be verified.

The comparison is as such: signatures provide non-repudiation, while MACs do not - one can know the owner of which private key signed a message, while anyone with the shared key could have produced an MAC for a message. The second point is that digital signatures are generally more computationally complex than MACs, and thus, given that the control of both ends lies with the same party, MAC is the message authentication of choice for this project.

2.2.2 IP Authentication Header

The security requirements for this project are equivalent to those provided by the IP Authentication Header (Kent, 2005). The IP authentication header operates between IP and the transport layer, using IP protocol number 51. The authentication header uses a hash function and a secret shared key to provide an Integrity Check Value. This check value covers all immutable parts of the IP header, the authentication header itself, and the data below the authentication header. Combined, this provides connectionless integrity and authenticity, as the IP header is authenticated. Further, the header contains a sequence number, which is used to prevent replay attacks.

Unfortunately, there are two reasons why this solution cannot be used: difficulties with NAT traversal, and inaccessibility for user-space programs. As the IP packet provides integrity for the source and destination addresses, any NAT that alters these addresses violates the integrity of the packet. Although NAT traversal is not an explicit success criteria for this project, it is implicit, as the flexibility of the project for different network structures is a priority, including those where NAT is unavoidable. The second is that IP authentication headers, being an IP protocol and not transport layer, would cause issues interacting with user-space programs. Given that the first implementation of transport is completed using TCP, having IP Authentication Headers would require the user-space program to handle the TCP connection without the aid of the kernel, complicating multiplexing and being an unsupported setup.

Overall, using the IP authentication header would function similarly to running over a VPN, described in section 2.2.3. Although this will be a supported configuration, the shortfalls mean that it will not be the base implementation. However, inspiration can be taken from the header structure, shown in figure 2.1.

It is first important to note the differences between the use of IP authentication headers and the security footers used in this application. Firstly, the next header field is unneces-

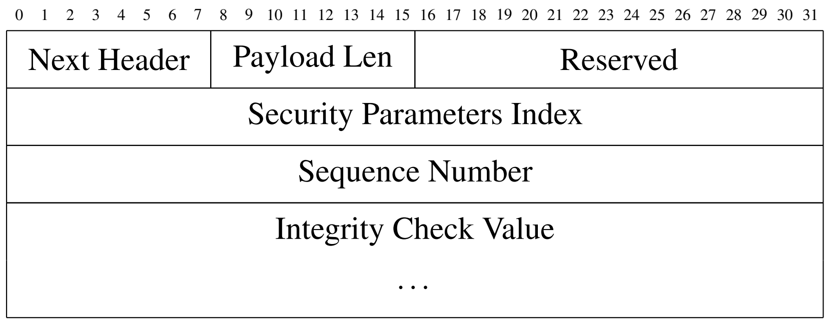


Fig. 2.1 IP authentication header structure

sary, given that headers are not being chained. Secondly, given the portals have a fixed security configuration by static configuration, the payload length field is unnecessary - the payloads will always be of a predetermined length. Similarly, the security parameters index is unnecessary, as the parameters will be equal.

The difference in security arises from the lack of integrity given to the fields above the application layer. That is, the IP header itself, and the TCP or UDP header. However, there is an important distinction between the TCP and UDP cases: TCP congestion control will not be covered by any application provided security, while the UDP congestion control will. That is, this application can do nothing to authenticate the ACKs of a TCP connection, as these are created outside of the control of the application. As such, the TCP implementation provided by the solution should be used in one of two ways: as a baseline test for the performance of other algorithms, or taking advantage of layered security as given in section 2.2.3. The rest of this section will therefore focus on securing the UDP transport.

Further differences arising from the lack of integrity above the application layer still apply to UDP transport. Although the congestion control layer and therefore packet flow is authenticated, the source and destination of packets are not.

Adapting for NAT

To achieve authentication with the IP Authentication Header, one must authenticate the source and destination addresses of the packet. However, these addresses may be altered by NAT devices in transit between the local and remote portals. In the case of source NAT, the source of an outgoing packet is masqueraded to the public address of the NAT router, likely altering the outgoing port as well. For destination NAT, an inbound packet to a NAT router will have its address changed to the internal destination, possibly changing the destination port too.

However, each of these address translations is predictable at the time of packet sending and the time of packet receiving. For a packet that will go through source NAT, the eventual

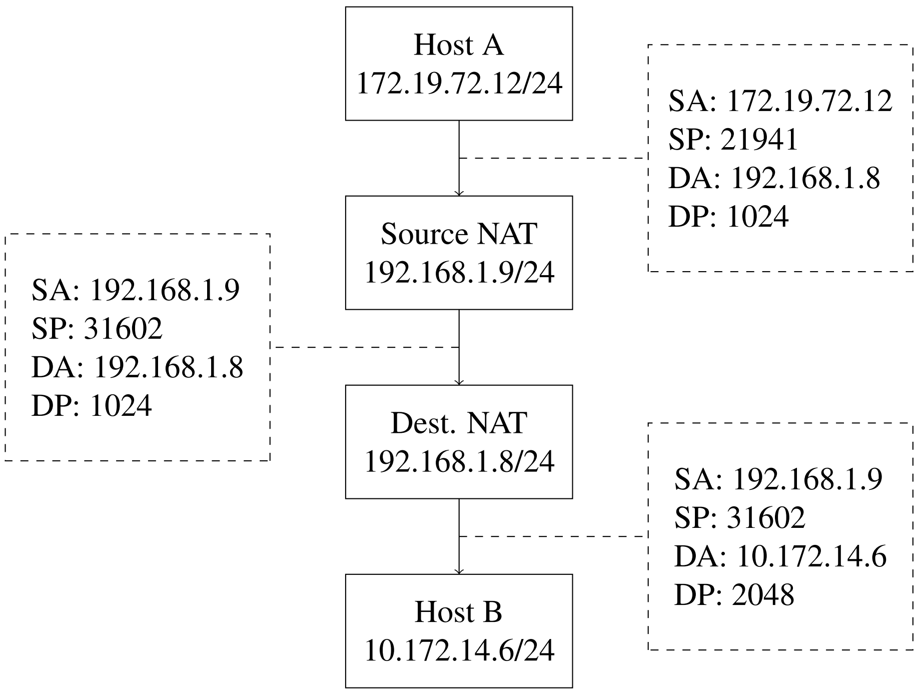


Fig. 2.2 UDP packet passing through source and destination network address translation, and the addresses and ports at each point.

source address is predictable, in that it will be altered from the internal address to the public address of the router. Likewise, with destination NAT, the destination address of the packet will be predictable as the public address of the router that receives it. An example of this is shown in figure 2.2.

Therefore, to authenticate the message's source and destination, the source address and destination address from the period between the NATs will be used. Host A can predict this by using the destination address of the flow transporting the packets and knowledge of its own NATs public IP as the source address. Similarly, host B can predict this by using the source address of the flow transporting the packets and knowledge of its own NATs public IP as the destination address. Although this does mean that the authentication would apply equally to any other device behind both NATs, this is an acceptable compromise for NATs controlled by the user. Achieving sufficient security under a CG-NAT is left as an exercise to the implementer, where the techniques described in section 2.2.3 can be applied.

Replay Attacks

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).

A specific of the multipath nature of this application is requiring the use of a sequence number space that is shared between flows. This is similar to the design pattern of multipath TCP'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).

2.2.3 Layered Security

It was previously mentioned that this solution focuses on maintaing the higher layer security of proxied packets. Further to this, this solution provides transparent security in the other direction. Consider the case of a satellite office that employs both a whole network corporate VPN and this 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.

These two examples are given in figures 2.3 and 2.4, for the VPN Wireguard (Donenfeld, 2017). In figure 2.4, the portals are only accessible via the VPN protected network. It can be seen that the packet in figure 2.4 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 this solution breaks 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 extraneous security guarantees and therefore causes reduced performance. To this end, the security features mentioned will all be configurable. This allows for flexibility in implementation.

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 intended to be 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.

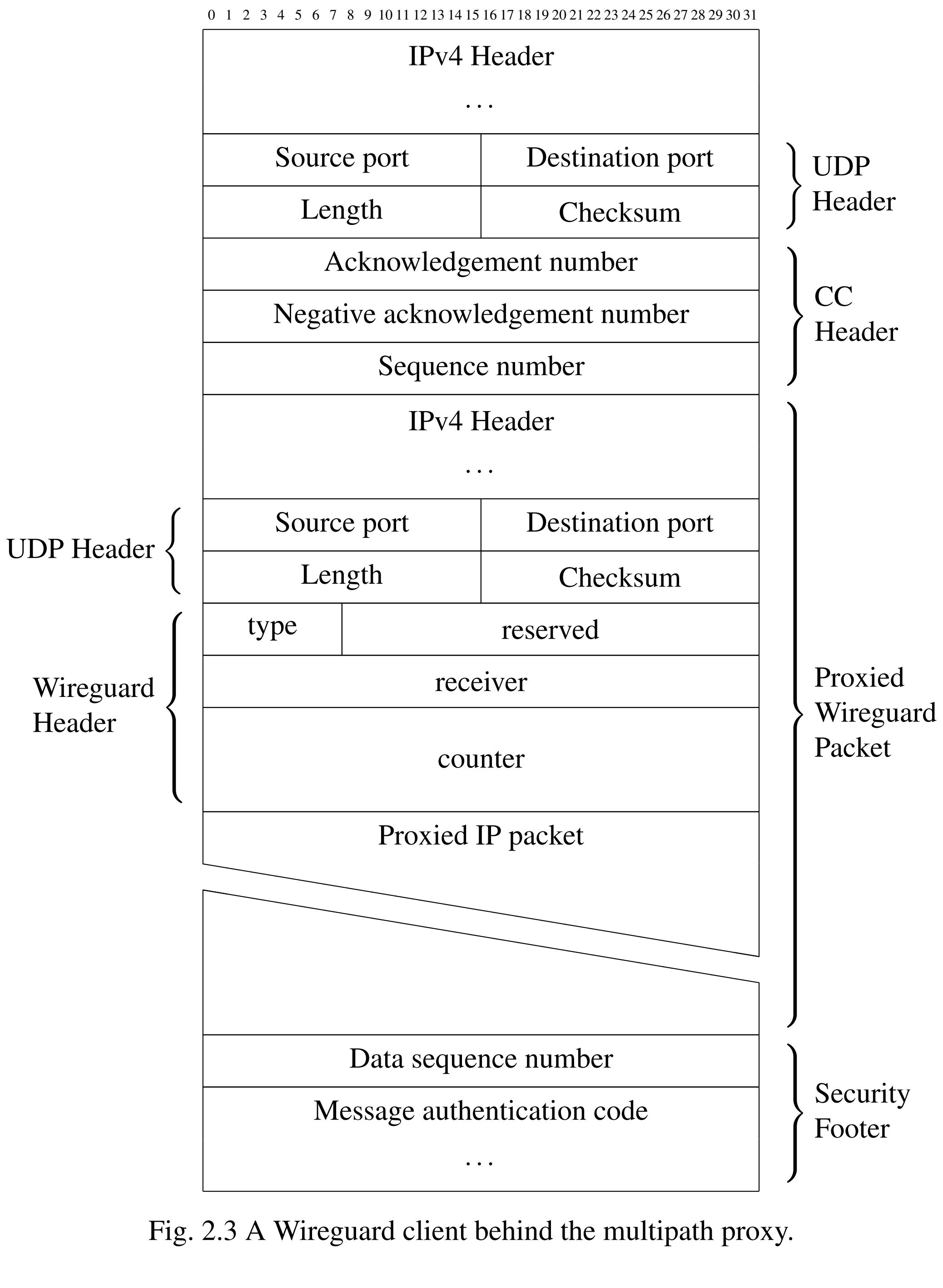


Fig. 2.3 A Wireguard client behind the multipath proxy.

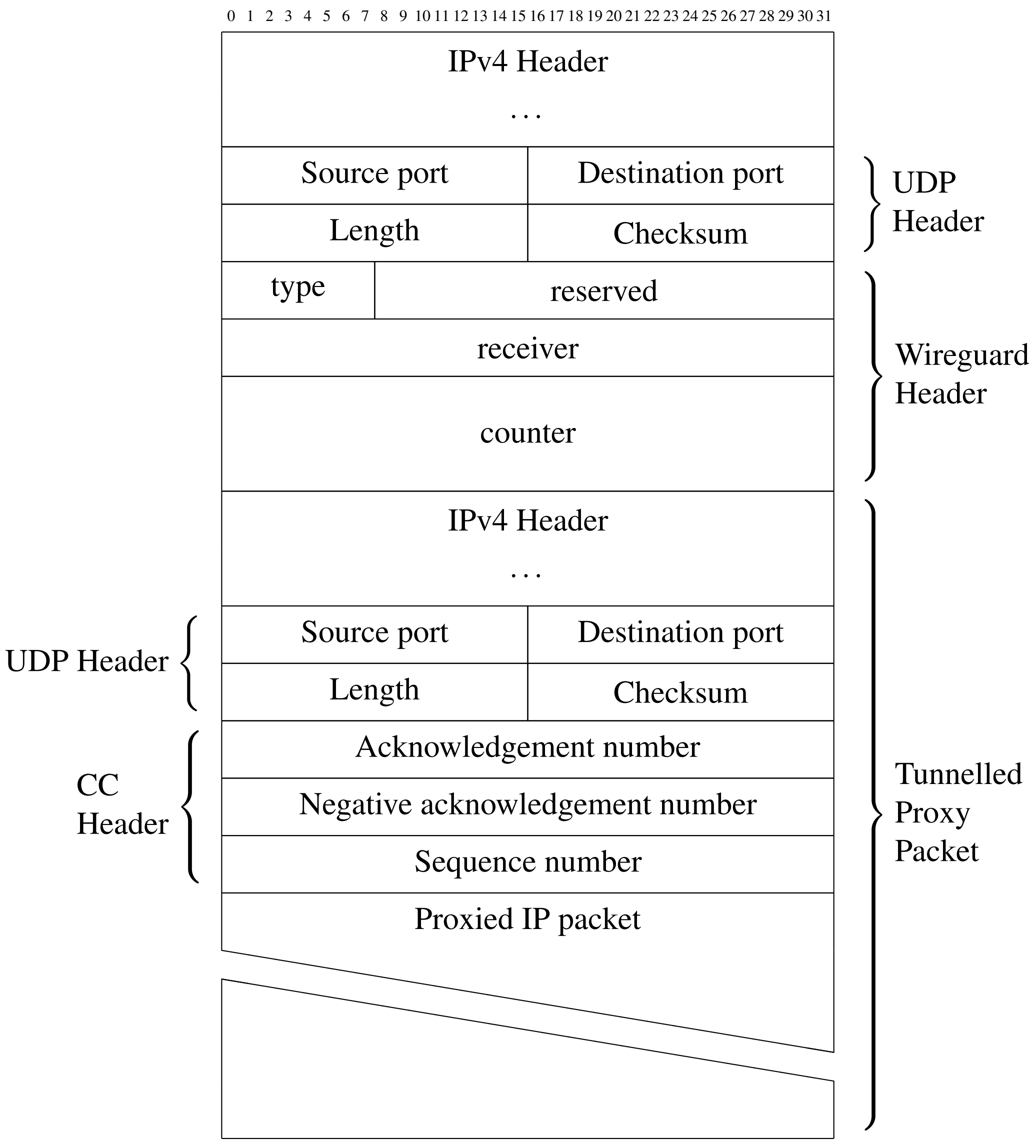


Fig. 2.4 A Wireguard client in front of the multipath proxy.

The primary considerations will be the performance of the language, clarity of code of the style needed to complete this software, and the ecosystem of the language. This culminates in choosing Go for the implementation language.

Alongside the implementation language, a language is chosen to evaluate the implementation. Two potential languages are considered here, Python and Java. Though Python was initially chosen for rapid development and better ecosystem support, the final result 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 these goals. The negatives of using C++ are demonstrated in the sample script, given in figure A.1, where it is immediately obvious that to achieve even the base of this project, the code in C++ is multiple times the length of equivalent code in either Rust or Go, at 93 lines compared to 34 for Rust or 48 for Go. This difference arises from the need to manually implement the required thread safe queue, while it is available as a library for both Rust and Go, and can be handled by the respective package managers. This manual implementation gives rise to additional risk of incorrect implementation, specifically with regards to thread safety, that could cause undefined behaviour and great difficulty debugging.

The lack of memory safety in C++ is a significant negative of the language. Although C++ would provide increased performance over a language such as Go with a runtime, it is avoided due to the massive incidental complexity of manual memory management and the difficulty of manual thread safety.

Rust

Rust is memory safe and thread safe, solving the latter issues with C++. Rust also has no runtime, allowing for similar 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, the downsides of Rust come from its youthfulness. This is two-faceted: IDE support and Crate stability. 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 many other languages. Secondly, the crate available for TUN support (tun-tap[1]) does not yet provide a stable API, which was noticed during the development of even this test program. Between writing the program initially and re-testing it to place in this document, the API of the Crate 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 didn't compile or function. Although writing the API for TUN interaction is not an issue, the safety benefits of Rust would be less pronounced, as the direct systems interaction would require unsafe code, 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. Regardless, it is the language of choice for this project, with a sample 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. The downside of this runtime is that the speed of execution is negatively affected. 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 sample of speeds displayed in section (Ref Needed: Introduction Comments on Speed), and the performance shown in section 4.5, the compromise of using a well-suited high-level language is one worth taking.

2.3.2 Evaluation Languages

Python

Python is a dynamically typed, and was chosen as the initial implementation language. The first reason for this is `matplotlib`[2], a widely used graphing library that can produce

[1]https://docs.rs/tun-tap/
[2]https://matplotlib.org/

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 changing requirements. The lack of static typing limits 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. One of the initial reasons for not choosing Java was the availability of an equivalent library to `proxmoxer`. Although two libraries to interact with Proxmox are available for Java, one was released under an incompatible license, and the other does not have adequate type safety. To this end, to develop in Java, I would need to develop my own Proxmox library. However, after the initial development in Python, it became clear that this was a valuable use of time, and thus development began. By developing a type safe Proxmox API library, and having learnt from the initial development in Python, a clear path to producing the appropriate Java libraries was available.

However, as Python is an incredibly popular language for data processing, the solution was not to use purely Java. Given the graphing existed already in Python and worked perfectly well, a combined solution with Java gathering the data and Python processing it was chosen.

2.4 Requirements Analysis

The requirements of the project are detailed in the Success Criteria of the Project Proposal (Appendix C), 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

[3]https://github.com/proxmoxer/proxmoxer

```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. 2.5 An example of refactoring for changing requirements.

stretch goals, or could have elements, are aimed at increasing performance by reducing overheads, and supporting IPv6 alongside IPv4.

2.5 Engineering Approach

Software Development Model

The development of this software followed the agile methodology. Work was organised into 2-7 day 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 extracted in the correct sized segments. Examples of these sprints are: initial build including configuration, TUN adapters 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 for performance and control.

One of the most important features of any agile methodology is welcoming changing requirements (Beck et al., 2001). As the project grew, it became clear where shortcomings existed, and these could be fixed in short sprints. An example is given in figure 2.5, 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 or DNS names could be impractical with certain setups. The static typing in the chosen language 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. As discussed in section 2.3, IDE support was important to me. My preferred IDEs are those supplied by JetBrains[4], generously provided for education and academic research free of charge. As such, I used GoLand for the Go development of this project, IntelliJ for the Java evaluation development, and PyCharm for the Python evaluation program. Using an intelligent IDE, particularly with the statically typed Go and Java, significantly increases my productivity as a programmer, and thus reduces incidental complexity.

I used Git version control, with a self-hosted Gitea[5] server as the remote. My repositories have a multitude of on- and off-site backups at varying frequencies (2xUSB + 2xDistinct Cloud Storage + NAS + Multiple Computers).

Alongside my self-hosted Gitea server, I have a self hosted Drone by Harness[6] server for continuous integration. This made it simple to add a Drone file to the repository, allowing for the Go tests to be run, formatting verified, and artefacts built. On a push, after the verification, each artefact is built and uploaded to a central repository, where it is saved for the branch name. This is particularly useful for automated testing, as the relevant artefact can be downloaded automatically from a known location for the branch under test. Further, artefacts can be built for multiple architectures, particularly useful when performing real world testing spread between x86_64 and ARMv7 architectures.

Licensing

I have chosen to license this software under the MIT license. The MIT license is simple and permissive, enabling reuse and modification of the code, subject to including the license. Alongside the hopes that the code will receive updated pull requests over time, a permissive license allows others to build upon the given solution. A potential example of a solution that could build from this is a company employing a SaaS (Software as a Service) model to configure remote portals on your behalf, perhaps including the hardware required to convert this fairly involved solution into a plug-and-play option.

[4]https://jetbrains.com/
[5]https://gitea.com/
[6]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

Security is a large area in this project - perhaps more than the single success criteria suggests. This preparation has led to two clear concepts in security: the system must be adaptable in code, and flexible in deployment. Being adaptable allows more options to be provided in the future, while deployment flexibility allows the solution to better fit into a network with special security requirements.

Go has a concurrency structure excellently suited to this project, and the large library reduces incidental complexity. Using a high level language allows for more readable code, which future implementations in a lower level language could effectively build off of. The structure of this project suggests a large initial program base, from which further features can be merged to reach the success criteria.

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 proxy as described. An overview of the software and system is presented in figure 3.1.

This chapter will detail this implementation in three sections. The software will be described in sections 3.1 and 3.2. Section 3.1 explains the software's structure and dataflow. Section 3.2 details the implementation of both TCP and UDP methods of transporting the tunnelled packets between the hosts. The system configuration will be described in section 3.3, along with a discussion of some of the oddities of multipath routing, such that a reader would have enough knowledge to implement the proxy.

3.1 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 flexiblity of the interfaces to future additions, while also describing the concrete implementations available with the software as of this work.

3.1.1 Proxy

The central structure for the operation of the software is the `Proxy` struct. The proxy is defined by its source and sink, and provides methods for `AddConsumer` and `AddProducer`. The proxy coordinates the dispatching of sourced packets to consumers, and the delivery of produced packets to the sink. This follows the packet data path shown in figure 3.2.

The proxy is implemented to take a consistent sink and source and accept consumers and producers that vary over the lifetime. This is due to the nature of producers and consumers, as

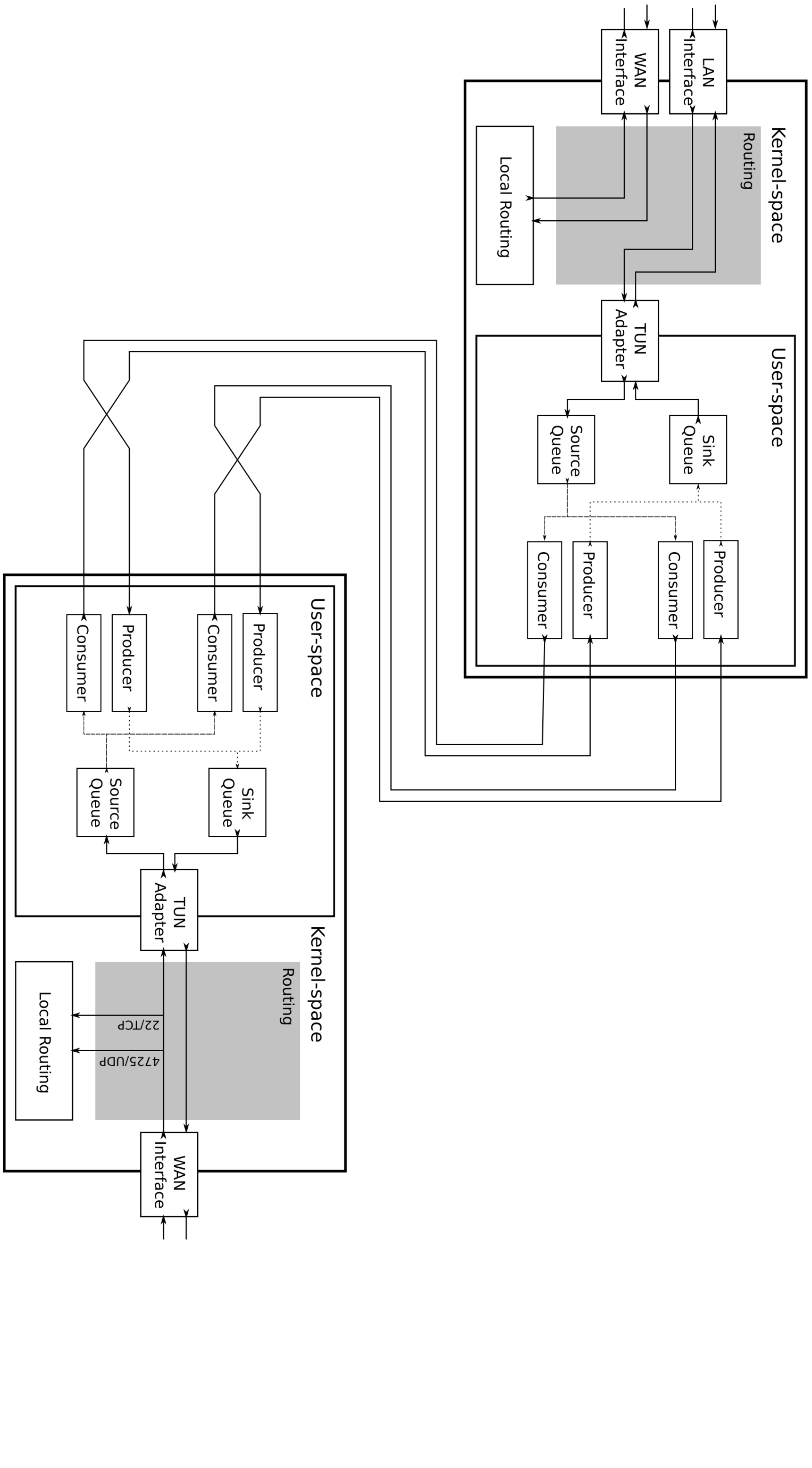


Fig. 3.1 Diagram of the dataflow within the proxy.

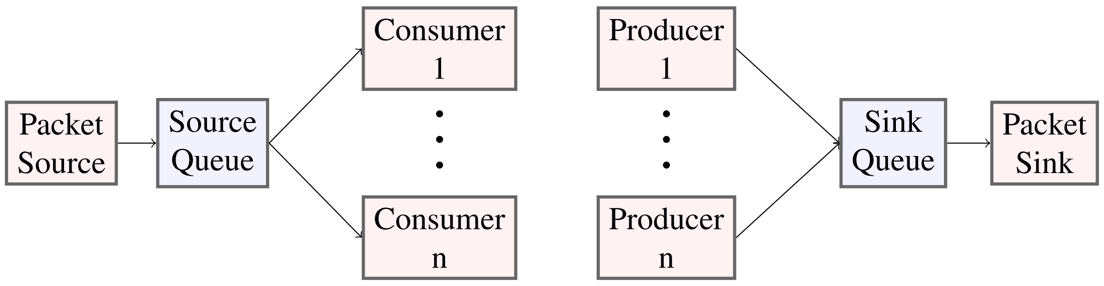


Fig. 3.2 Packet flow within proxy start method.

```
do:
  while is_reconnectable(consumer) and not is_alive(consumer):
    reconnect(consumer)
  while is_alive(consumer):
    packet = source_queue.popOrBlock()
    consumer.consume(packet)
while is_reconnectable(consumer)
```

Fig. 3.3 Pseudocode for a consumer, supporting reconnection.

each may be either ephemeral or persistent, depending on the configuration. An example is a device that accepts TCP connections and makes outbound UDP connections. In such a case, the TCP producers and consumers would be ephemeral, existing only until they are closed by the far side. The UDP producers and consumers are persistent, as control of reconnection is handled by this proxy. As the configuration is deliberately intended to be flexible, both of these can exist within the same proxy instance.

The structure of the proxy is built around the flow graph in figure 3.2. The flow graph demonstrates the four transfers of data that occur: packet source to source queue, source queue to consumer, producer to sink queue, and sink queue to packet sink. For the former and latter, these exist once for an instance of the proxy. The others run once for each consumer or producer. The lifetime of producers and consumers are controlled by the lifetime of these data flow loops and are only referenced within them, such that the garbage collector can collect any producers and consumers for which the loops have exited.

Finally is the aforementioned ability for the central proxy to restart consumers or producers that support it (thus far, those initiated by the proxy in question). Pseudocode for a consumer is shown in figure 3.3. Whenever a producer or consumer terminates, and is found to be restartable, the application attempts to restart it until succeeding and re-entering the work loop.

3.1.2 Configuration

The configuration format chosen was INI, extended with duplicate names. Included is a single Host section, followed by multiple Peer sections specific to a method of communicating with the other side. Processing the configuration file is split into three parts: loading the configuration file into a Go struct, validating the configuration file, and building a proxy from the loaded configuration.

Validation of the configuration file is included to discover configuration errors prior to building an invalid proxy. Firstly, this ensures that all parts of the program built from the configuration are given values which are invalid in context and easily verifiable, such as a TCP port of above 65,535. Secondly, catching errors in configuration before attempting to build the proxy constrains the errors of an invalid configuration to a single location. For a user, this might mean that an error such as `Peer[1].LocalPort invalid: max 65535; given 74523` is shown, as opposed to `tcp: invalid address`, which more clearly explains the user's error.

Once a configuration is validated, the proxy is built. This is a simple case of creating the proxy from the given data and adding the producers and consumers for its successful running, given that the provided configuration can already be built. Whereas other packages function in terms of interfaces, the builder package ties together all of the pieces to produce a working proxy from the configuration.

3.1.3 Running the Application

The software is designed to run 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 correctly configured, after a short pause this command exits with status 0. The system then has an interface named `nc0` which provides a tunnel as configured. To achieve this in a C application, a fork is used. However, the language Go is incompatible with forking, so instead a second process is spawned. Spawning a process and dying is convenient here for two reasons: if a user launches the application it leaves their shell clear, and if an init system launches the application it knows that the TUN is available as soon as the application exits.

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

To exit cleanly after opening the TUN adapter, the application hands over control of the adapter to a second process. To achieve this, a duplicate process is spawned with the TUN's file descriptor in slot 3, and an environment variable stating as such. When the application is started it checks for the presence of this variable, and if it finds it, knows that it is the child process. Once this process is spawned, the parent process may exit, as the child process now controls the TUN adapter and can continue the work of the proxy.

The application expects to find configuration in well known locations, `/etc/netcombiner/%IF` on Linux and `/usr/local/etc/netcombiner/%IF` on FreeBSD. However, the application also supports being provided with a configuration file location on the command line, ensuring flexibility with other systems, such as router operating systems based on Linux or FreeBSD, without code changes.

3.1.4 Sourcing and Sinking Packets

Packets that wish to leave the software leave via a sink, and packets entering arrive via a source. As the application is developed in user space, the solution that is most flexible here is a TUN adapter. A TUN adapter provides a file like interface to the layer 3 networking stack of a system.

Originally it was intended to use the Go library `taptun` for TUN driver interaction, but this library ended up lacking platform compatibility that I was aiming for with this project. Fortunately, the `wireguard-go` project has excellent compatibility for TUN adapters, and is licensed under the MIT-license. This allows me to instead rely on this as a library, increasing the software's compatibility significantly.

3.1.5 Security

The integrated security solution of this software is in three parts: message authentication, repeat protection, and cryptographic exchanges. The interfaces for each of these and their implementations are described in this section.

Message Authenticity Verification

Message authentication is provided by a pair of interfaces, `MacGenerator` and `MacVerifier`. `MacGenerator` provides a method which takes input data and produces a sum as output, while `MacVerifier` confirms that the given sum is valid for the given data.

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

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 produced, 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.

One difference with repeat protection is that it is shared between all producers and consumers. This is in contrast to the message authenticity, which are thus far 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 includes locking for thread safety.

Exchange

The `Exchange` interface provides for a cryptographic exchange, but is flexible enough to be used for other purposes too, as will be described for UDP congestion control. When beginning a flow, an ordered list of the `Exchange` type is supplied. These exchanges are then performed in order until each has completed. If any exchange fails, the process returns to the beginning.

Currently, no cryptographic exchange is necessary, as the methods mentioned above are symmetric. However, the exchange interface is taken advantage of when beginning a UDP flow. As UDP requires an initial exchange to initiate congestion control and establish a connection with the other node, this exchange interface is used. By pairing the congestion controller with the initial UDP exchange, the exchange interacts with the congestion controller, setting up the state correctly for the continuing connection.

This demonstrates the flexibility of combining the exchange interface with other objects. Although the software does not currently implement any key exchange algorithms, this is possible with the interfaces as described. Simply provide a type that implements both `Exchange`

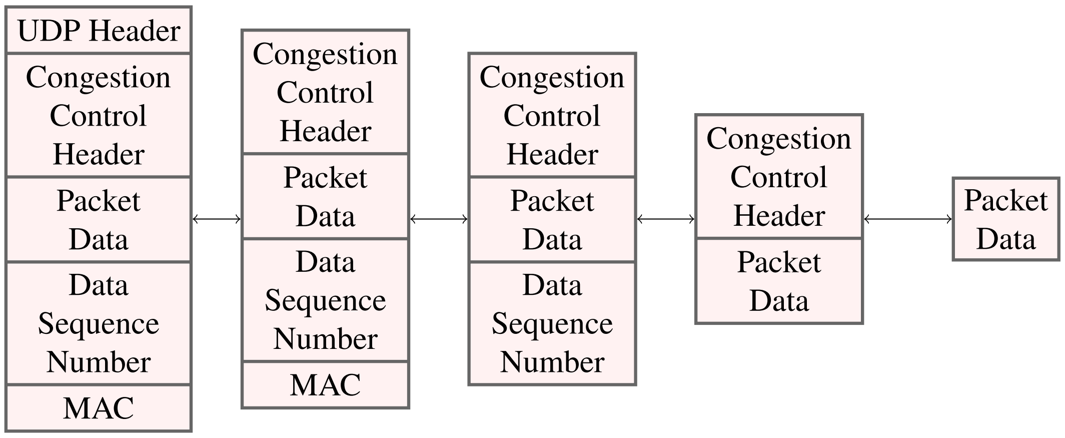


Fig. 3.4 Data flow of a UDP packet through the application.

and `MacGenerator`. During the exchange, the keys needed for message authentication can be inserted directly into the structure, after which it will work for the lifetime of the consumer.

3.1.6 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 into the application binary, and `evaluation` is used to verify the performance characteristics and generate graphs.

3.2 Packet Transport

As shown in figure 3.1 and described in section 3.1, the interfaces through which transport for packets is provided between the two hosts are producers and consumers. A transport pair is then created between a consumer on one host 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 a base for a proof of concept, while UDP expands on this proof of concept to produce a usable solution. This section discusses, in section 3.2.1, the method of transporting discrete packets across the continuous byte stream of a TCP flow. Then, in section 3.2.2, it goes on to discuss adding congestion control to UDP datagrams, while avoiding retransmissions.

```
/
├── code ........................................ Go code for the project
│   ├── config .......................... Configuration management
│   ├── crypto ............................... Cryptographic methods
│   │   ├── exchanges ............... Cryptographic exchange FSMs
│   │   └── sharedkey ........................... Shared key MACs
│   ├── mocks .............................. Mocks to enable testing
│   ├── proxy ............................ The central proxy controller
│   ├── shared ....................................... Shared errors
│   ├── tcp ................................... TCP flow transport
│   ├── tun .............................................. TUN adapter
│   ├── udp ............................... UDP datagram transport
│   │   └── congestion ............ Congestion control methods
│   └── utils ............................ Common data structures
└── evaluation ............ Result gathering and graph generation
    ├── java ................. Java automated result gathering
    └── python .......................... Python graph generation
```

Fig. 3.5 Repository folder structure.

3.2.1 TCP

The base implementation for producers and consumers takes advantage of TCP. The requirements for the load balancing given above 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.

TCP is a stream oriented connection, while the packets to be sent are discrete datagrams. That is, a TCP flow cannot be connected directly to a TUN adapter, as the TUN adapter expects 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. On the sending side, this involves writing the 32-bit length of the packet, followed by the packet itself. For the receiver, first 4 bytes are read to recover the length of the next packet, after which that many bytes are read. This successfully punctuates the stream oriented connection into a packet based connection.

However, using TCP to tunnel TCP packets (known as TCP-over-TCP) can cause a degradation in performance in non-ideal circumstances (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 do not require reliable delivery, such as a live video stream - a live stream does not wish to wait for a packet to be redelivered from a portion that is already played, and thus will spend longer buffering than if it received the up to date packets instead. 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 a download to arrive, increasing latency unnecessarily.

Although the TCP implementation provides an excellent proof of concept and basic implementation, work moved to a second UDP implementation, aiming to solve some of these problems. However, the TCP implementation is functionally correct, so is left as an option, furthering the idea of flexibility maintained throughout this project. 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.

3.2.2 UDP

To resolve the issues seen with TCP, an implementation using UDP was built as an alternative. UDP differs from TCP in that it provides almost no guarantees, and is based on sending discrete datagrams as opposed to a stream of bytes. However, UDP datagrams don't provide the congestion control or flow control required, so this must be built on top of the protocol. As the flow itself can be managed in userspace, opposed to the TCP flow which is managed in kernel space, more flexibility is available in implementation. This allows received packets to be immediately dispatched, with little regard for ordering.

3.2.3 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 TCP congestion control algorithms to apply in an unreliable context. 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 unreliable protocol, cumulative acknowledgements are not as simple. As 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 3.6b. Neither side can progress

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

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.

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. 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.

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 3.7. The congestion control header consists of the sequence number and the ACK and NACK, each 32-bit unsigned integers.

New Reno

The first algorithm to be implemented for UDP Congestion Control is based on TCP New Reno. TCP New Reno is a well understood and powerful congestion control protocol. RTT estimation is performed by applying $RTT_{AVG} = RTT_{AVG} * (1 - x) + RTT_{SAMPLE} * x$ for each newly received packet. Packet loss is measured in two ways: negative acknowledgements when a receiver receives a later packet than expected and has not received the preceding for $0.5 * RTT$, and a sender timeout of $3 * RTT$. The sender timeout exists to ensure that even if the only packet containing a NACK is dropped, the sender does not deadlock, though this case should be rare with a busy connection.

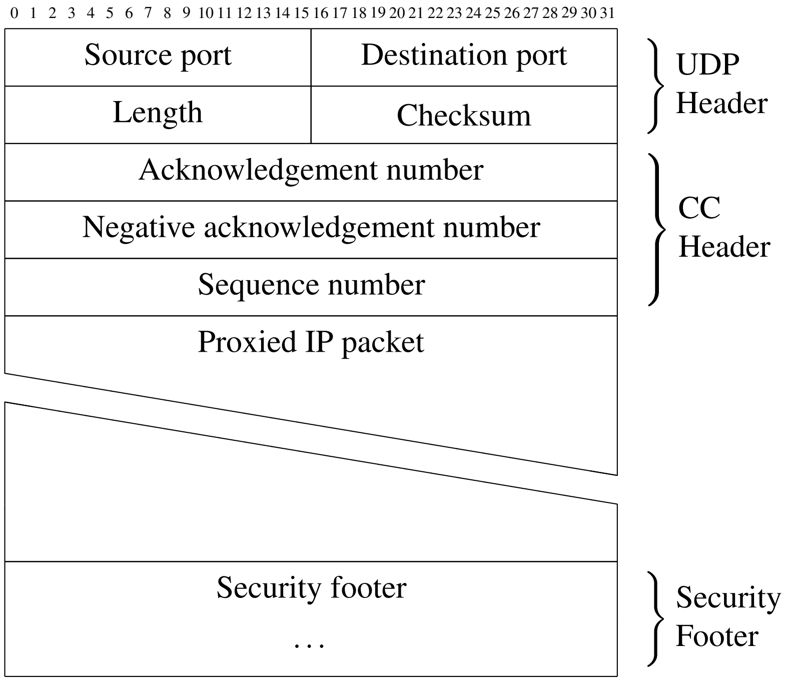


Fig. 3.7 UDP packet structure

To achieve the same curve as New Reno, there are two phases: exponential growth and congestion avoidance. On flow start, using a technique known as slow start, for every packet that is acknowledged, the window size is increased by one. When a packet loss is detected (using either of the two aforementioned methods), slow start ends, and the window size is halved. Now in congestion avoidance, the window size is increased by one for every full window of packets acknowledged without loss, instead of each individual packet. When a packet loss is detected, the window size is half, and congestion avoidance continues.

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 and remote portals, 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 how to route relevant packets via this interface. In sections 3.3.1 and 3.3.2, the configuration for routing the packets for the remote portal and local portal respectively are explained. Finally, in section 3.3.3, some potentially unexpected behaviour of using devices with multiple interfaces is explained, 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 Portal Routing

The common case for remote portals is a cloud 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 portal. 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 portal, 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)
```

jsh77: Test this pass out line.

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 portal side, as dropping packets before saturating the low bandwidth connections between the local and remote portal 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, and is left as an exercise.

3.3.2 Local Portal Routing

Routing within the local portal 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 specific rules are required to achieve this routing. Although this is true in most, the overview diagram avoids the complexity of the kernel routing to this software itself, which will be discussed in more detail here. Therefore, 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 portal 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 portal 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 portal. The routing for the local portal 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 portal address, and the gateway to `192.168.1.1`. Now the local portal can send and receive packets to the remote portal, but some further routing rules are needed to ensure that the packets from the proxy reach the remote portal, 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 portal while also forwarding the packets necessary to the client. The local portal 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
```

The final discovery I made is that many of these problems can be solved by changing the question. In my real world testing, explained in section 4.5.3, the local portal lies behind a dual WAN router. This router allows the same port to be accessible via two WAN IPs, and avoids any routing complication as the router itself handles the NAT perfectly. Prior to this I was attempting to route outbound, similar to the situation described above, with some difficulty. Hence it is worth considering whether an architecture modification can make the routing simpler for the task you are trying to achieve.

Chapter 4

Evaluation

This chapter will discuss the methods used to evaluate my project and the results gained. The results will be discussed in the context of the success criteria laid out in the Project Proposal.

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.

4.1 Evaluation Methodology

I performed my experiments on a local Proxmox[1] server. To encourage frequent and thorough testing, a harness was built in Python, allowing tests to be added easily and repeated with any code changes.

Proxmox was chosen due to its RESTful API, for integration with Python. It provides the required tools to limit connection speeds and disable connections. The server that ran these tests holds only a single other virtual machine which handles routing. This limits the effect of external factors on the tests.

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

[1]https://proxmox.com

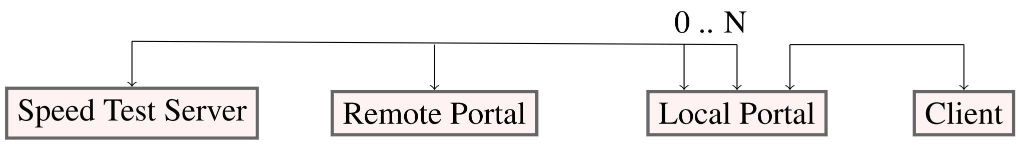


Fig. 4.1 The network structure of standard tests

4.1.1 Data Gathering

To generate these results, a fresh set of VMs (Virtual Machines) are created and the software installed on them. Once this is complete, each test begins, and is repeated a fixed number of times. When visualising the data produced, unless otherwise specified, the error bars will represent the inter-quartile range of the data, and the plotted point the median.

The network structure of all standard tests is shown in figure 4.1. Any deviations from this structure will be mentioned. The Local Portal has as many interfaces as referenced in any test, plus one to connect to the client. All Virtual Machines also have an additional interface for management, but this has no effect on the tests.

4.2 Success Criteria

4.2.1 Flow Maintained

Demonstrating that a flow is maintained under connection loss has two cases: TCP flows and UDP flows. For TCP flows, the success criteria will be met if a TCP flow can continue to provide reliable transmission under a connection loss. UDP flows are less standardised, so will be split into two categories: an artificial iperf3 test, and a less artificial SIP phone call. The iperf3 test can be used to study the packet behaviour at the time of connection loss, while the SIP phone call provides a representative example of a UDP flow that would be dropped under normal conditions, but is not dropped under this proxy.

TCP

To test whether a TCP flow is maintained, a pair of small Python scripts were written. The first creates a TCP server and listens on a port. When it receives a message, it reads the first 12 bytes as a UTF-8 string, replaces the second character with an 'o', and sends back the string. The client sends messages of the form `ping%s`, where `%s` is an 8-byte nonce. It therefore receives back messages of the form `pong%s`, where `%s` is the nonce that it just sent. This allows the client to check that the connection to the server is still responding.

jsh77: Fill in missing graph.

UDP

Firstly, the effect of connection loss on a UDP flow will be judged by the packet loss statistics of an iperf3 test. The bandwidth of the test will be kept sufficiently low that packet loss would not be expected, given that loss is only introduced within the test network when bandwidth exceeds the limit. This bandwidth is chosen as 128KBps, which is sufficiently low as to not hit the bandwidth limits, but far higher than a UDP flow such as a SIP call.

If the proxy can maintain a flow under a connection loss, the expected result is a peak in packet loss of no more than 50%, which rapidly returns to 0%. This represents the small portion of packets needing to be sent to note that the internal flow is offline, and thus stop sending packets to it to be lost. This loss should not exceed 50%, as the connections are of equal bandwidth and thus should receive half of the packets each.

jsh77: Fill in missing graph.

The second test is qualitative, involving making a call and checking for disconnection. Firstly, the call is made and both connections for the local proxy disconnected. This represents a failure scenario, as no connections continue existing. It should be confirmed that the call is dropped. Secondly, a call is made from which only one connection is disconnected. To pass this success criteria, the call should continue, though may experience some minor disruption as the connection is disconnected.

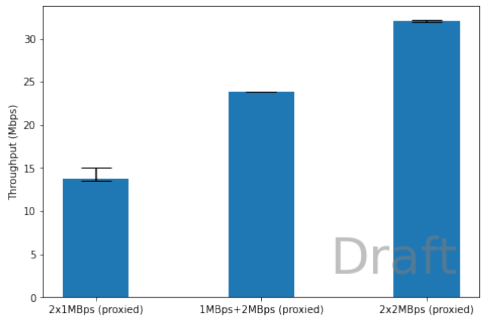
jsh77: Fill in Wireshark traces and analysis.

4.2.2 Bidirectional Performance Gains

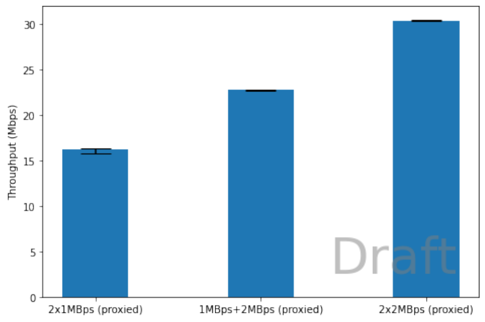
To demonstrate that all performance gains are bidirectional, I will provide graphs both inbound and outbound to the client for each performance test executed in this evaluation. This will sufficiently show the performance gains in each case. Inbound tests occur with the test server running on the proxy client and the test client running outside, while outbound tests occur with the test server running outside of the proxy and reaching in.

To demonstrate this somewhat succinctly, a pair of graphs for the same test in a common case will be shown. To demonstrate that this requirement is satisfied for all cases, for each graph of results presented in this evaluation, the graph for the alternative direction will be provided in appendix B.

Figure 4.2 shows two graphs of the same set of tests - one for the inbound performance and one for the outbound. It can be seen that both graphs show the same shape, satisfying that the performance gains of this proxy apply in both directions.



(a) Throughput of proxied connections inbound to the client.



(b) Throughput of proxied connections outbound from the client.

Fig. 4.2 Throughput results both inbound to the client and outbound from the client.

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

Fig. 4.3 The IP layout of the standard test network structure.

4.2.3 IP Spoofing

To demonstrate that the IP of the client can be set to the IP of the remote portal, the network structure shown in figure 4.1, used for most of these tests, can be examined further. This will demonstrate that it is possible to set the IP as such, as all of the tests in this section did so.

In the given network structure, the speed test server, remote portal and local portal 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 portal to the client. The IP addresses of the interfaces shown in figure 4.1 are listed in 4.3. 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 portal. To achieve this, the client configuration is particularly simple. A static route is added for 192.168.1.1 from the eth0 interface, and this then set as the default gateway. The IP address is set as the IP address of the remote portal. The details of this configuration are provided in 3.3.

Given that the client shares the IP address of the remote portal in these cases, it is demonstrated that this success criteria is met. Sharing the IP of the remote portal allows most routers to be configured behind the local portal as a client, allowing it to act as a standard Internet connection. An alternative approach, where the local portal acts as a router, is detailed in section 4.5.3.

4.2.4 Security

Success in terms of security involves providing security no worse than a standard connection. To demonstrate that this is satisfied, I refer back to section 2.2.3, in which I describe the ability for this proxy to be layered with other security software. Specifically, the ability to run this proxy behind the VPN solution Wireguard. By setting up a Wireguard tunnel for each connection and using a separate IP range in each, configuring the proxy to run behind Wireguard is no more complicated than the IP routing necessary.

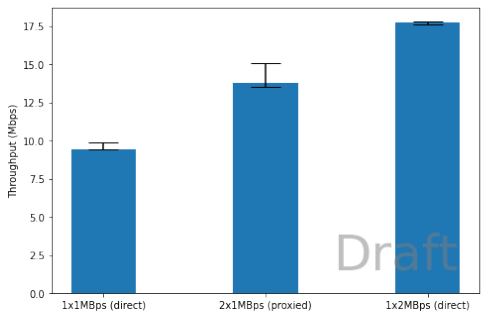
Therefore, to provide security no worse than a standard connection, it is sufficient to show that the security provided is better than a standard connection. If Wireguard provides security better than a standard connection, then it is possible for this proxy to be configured such that it provides security no worse than a standard connection. Further, if any solution which this can be configured behind, such as IPsec Authentication Headers, provides the correct security guarantees, then the security is no worse.

Further, I presented an additional security mechanism that does not rely on other software. However, given the difficulty of proving the comparative security, I will be relying on the ability to improve security with layering to satisfy this success criteria.

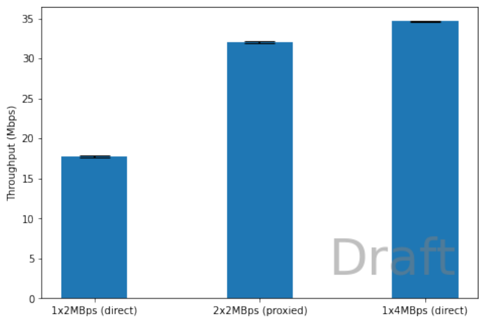
4.2.5 More Bandwidth over Two Equal Connections

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

The results of these tests are given in figure 4.4, for both a pair of 1MBps connections and a pair of 2MBps 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



(a) Throughput of proxied connections inbound to the client.



(b) Throughput of proxied connections outbound from the client.

Fig. 4.4 Graphs demonstrating that the throughput of two connections proxied lie between one connection of the same speed and one connection of double the speed

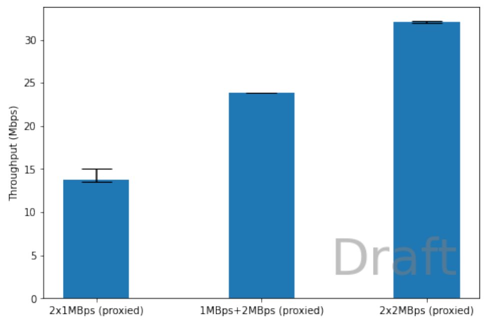
equal bandwidth connection, demonstrating a good portion of the maximum performance is achieved.

4.3 Extended Goals

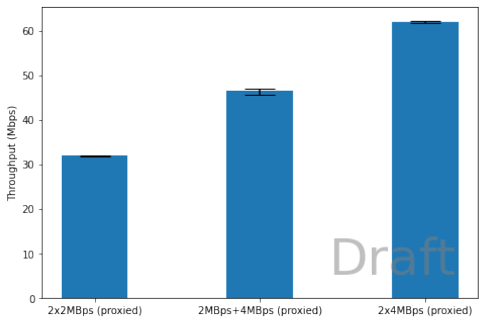
4.3.1 More Bandwidth over Unequal Connections

For showing improved throughput over connections which are not equal, three results will be compared. That of two pairs of equal connections, one lower and one higher, and one of unequal connections, composed of the one of each of the pairs. To show that unequal connections exceed the performance of a pair of lower connections, the results for the unequal proxied connections should lie between the other two. 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 1MBps and 2MBps connections, and another for 2MBps and 4MBps connections. In both cases, it can be seen that the proxy with unequal connections lies between the equal connection proxies. Further, it can be seen that both unequal proxied connections lie approximately halfway between the equal pairs. This suggests that the proxy design is successful in being invariant to the static balance of connection throughput.

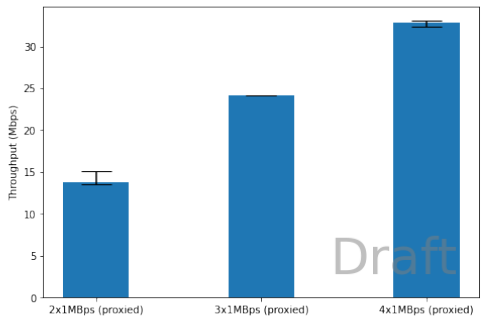


(a) Throughput of proxied connections inbound to the client.

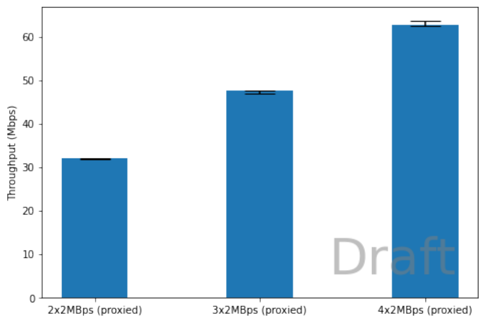


(b) Throughput of proxied connections outbound from the client.

Fig. 4.5 Graphs demonstrating that the throughput of two two connections proxied lie between one connection of the same speed and one connection of double the speed



(a) 1MB connections



(b) 2MB connections

Fig. 4.6 Scaling of equal connections

4.3.2 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 4.5.2.

Provided in figure 4.6 are results for both 1MBps and 2MBps connections. Firstly, it is clear that the proxy consisting of 4 connections exceeds the throughput of the proxy consisting of 3 connections in both cases. Secondly, it appears that a linear trend is forming. This trends will be further evaluated in section 4.5.2, but suggests that the structure of the proxy suffers little loss in performance from adding further connections.

4.3.3 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 bandwidth 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. The error bars on the x-axis represent the range of continuous time results included in each discrete plotted point, while the y-axis error bars again represent the inter-quartile range of the gathered data. The target rates will be plotted as a fixed line for each of the speeds, as opposed to time-series. The error bar for these series will be omitted, as they occlude much of the graph, and are visible in figure (ref needed).

The criteria will be met if the following are true: the throughput begins at the rate of a time constant connection; the throughput stabilises at the altered rate after alteration; the throughput returns to the original rate after the rate is reset.

jsh77: Re-gather data and include graph.

Two graphs are presented here. Figure **??** presents a situation where the speed of a connection decreases, before returning to its original rate. This test begins with two 2MBps connections, changing to 1MBps + 2MBps at $t = 10$, and returning to two 2MBps connections at $t = 20$. Figure **??** presents a situation where the speed of a connection increases, before returning to its original rate. This test begins with two 2MBps connections, changing to 3MBps + 2MBps at $t = 10$, and returning to two 2MBps connections at $t = 20$.

4.3.4 Connection Loss

This criteria judges the ability of the proxy as a whole to handle a complete connection loss while maintaining proportional throughput. 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.

jsh77: Re-evaluate with UDP.

4.3.5 Single Interface Remote Portal

Similarly to section 4.2.3, 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.

4.3.6 Connection Metric Values

The extended goal of connection metric values has not been implemented at this point.

4.4 Stretch Goals

4.4.1 IPv4/IPv6 Support

The project is only tested with IPv4.

4.4.2 UDP Proxy Flows

Although UDP proxy flows are implemented, they are unable to provide improved performance over a TCP connection.

4.4.3 IP Proxy Packets

The project only supports TCP and UDP flows for carrying the proxied data.

4.5 Performance Evaluation

The discussion of success criteria above used relatively slow network connections to test scaling in certain situations. This section will focus on testing how the solution scales, in terms of faster individual connections, and with many more connections. Further, all of the above tests were automated and carried out entirely on virtual hardware. This section will show some 'real-world' data, using a Raspberry Pi 4B and real Internet connections.

4.5.1 Faster Connections Scaling

jsh77: Once automated testing is up, grab the data for this and plot a couple of graphs.

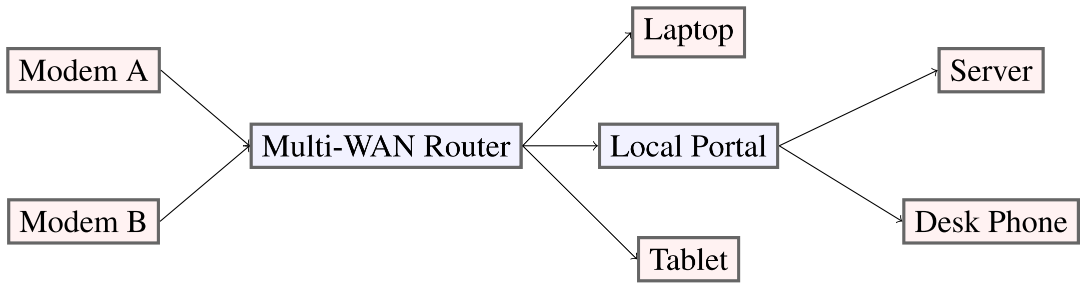


Fig. 4.7 Real world proxy implementation.

4.5.2 Number of Connections Scaling

jsh77: Once automated testing is up, grab the data for this and plot a couple of graphs.

4.5.3 Real World Testing

Although the success criteria of this project revolve around virtual hardware, it extends into the real world. This section will describe the application of this proxy to my network, including the considerations made in network design.

Figure 4.7 presents the real world deployment as setup in my network. It begins with two modems providing two Internet connections to the multi-WAN router. The multi-WAN router uses session based load balancing to achieve usage of both connections simultaneously, across multiple devices. For some devices, such as laptops and tablets, this is sufficient. To avoid the extra cost of proxying traffics, such devices access the Internet directly via the multi-WAN router. Thus far, the configuration is standard for a multi-ISP setup.

Behind the multi-WAN router lies the local portal. Using a multi-WAN router here simplifies the configuration of the local portal. By forwarding a port from both WANs on the router to the local portal's internal host, it can receive connections via both WANs with a single IP address. The remote portal is then configured to connect via both IP addresses. The flexibility of the system allows either portal to listen for or initiate connections, which simplifies setup in this use case.

The local portal in this configuration runs FreeBSD[2] 13 on a Raspberry Pi 4. To make the most use of the proxied IP address, the local portal is configured as a NAT router. Source NAT is configured to translate the source address of outgoing packets to the address of the remote portal, allowing them to be proxied. Destination NAT is used to forward ports from the remote portal's IP address to the devices on the internal network.

[2]https://www.freebsd.org/

Chapter 5

Conclusions

The software produced in this project provides a method of combining multiple Internet connections, prioritising throughput and resilience in the resultant aggregate connection.

The project was a success. I met all of the core success criteria, and many of the extended goals beyond that. This project has produced a piece of software that provides an interesting approach to combining Internet connections. The software is useful and has seen one real world deployment, and it's hoped that others will find it a useful tool in their circumstances.

Though this project provides an approach that can work very well in today's Internet, it's hoped that the modernisation of the Internet will make it largely redundant. At present, the solution presented in this dissertation provides a highly effective solution for combining Internet connections in the following circumstances: when a single flow needs to exceed the capacity of a single connection, when connections are unreliable and flows must be maintained, and when connections have unpredictable performance characteristics. In each of these cases, this solution can currently be implemented to provide benefits to an entire network. However, using this system involves forwarding packets, increasing latency and cost. Some further work will be presented that solves these problems in different ways.

Potential further work focuses on providing the same abilities as this work on a per-device level, as opposed to for the entire network. That is, rather than inserting a middlebox to provide a combined connection, each device would be able to observe those benefits individually. One such solution is MultiPath TCP, discussed previously, which is close to seeing deployment in the Linux kernel[1]. Consider the case where a mobile device, such as a laptop, connects to a WiFi network. This device gains only a single IP from this WiFi network, but the network has access to two Internet connections. To obtain optimal performance, MultiPath TCP should create a flow that takes advantage of each Internet connection. However, work is needed to establish the most effective method to achieve this.

[1]https://kernel.org

Some options are creating multiple exploratory flows and checking if they're split between IPs, or the router informing the device that more IPs are available for flows. This could be taken one step further, with an MPTCP capable router expanding a non-MPTCP capable TCP flow to MPTCP at the router level, and continuing to show it as a standard TCP flow to the end device, allowing older devices to experience the benefits of multipath.

UDP connections such as for phone calls are also seeing improvements for multipath, though less uniformly than TCP. Multipath usage with UDP flows is often highly application specific. Considering a phone call, the tiny bandwidth usage suggests that it might be feasible to simply duplicate the packets for resilience, with appropriate timing. Further work in the UDP space will likely focus on providing modern protocols with specific resilience for existing uses of UDP. Further, QUIC is seeing work on a multipath version, though this shares many similarities with MPTCP.

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.

To conclude, the proxy built in this project provides an effective method to combine dynamic Internet connections, that 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 huge advantage, especially in situations where gaining a single faster link is difficult.

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.

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/.

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.

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. page 60110H, Boston, MA, October 2005. doi: 10.1117/12.630496. URL http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.630496.

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

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.

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.

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

```cpp
#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

```cpp
    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

Outbound Graphs

The graphs shown in the evaluation section are Inbound to the Client (unless otherwise specified). This appendix contains the same tests but Outbound from the client.

Appendix C

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.

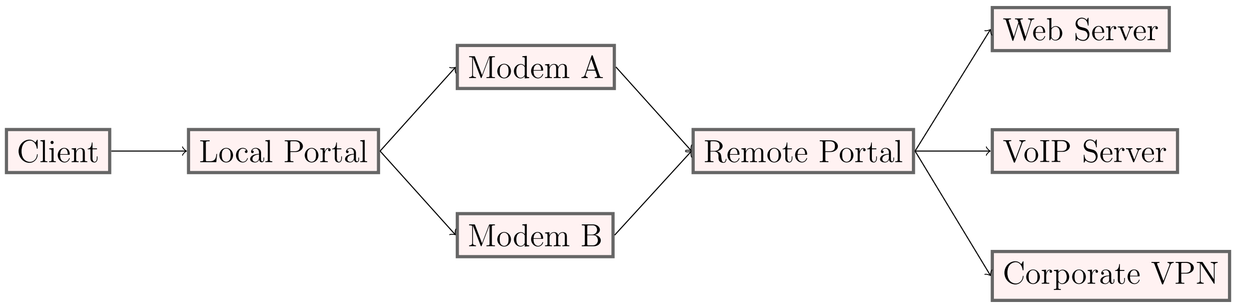


Figure 1: A network applying this proxy

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

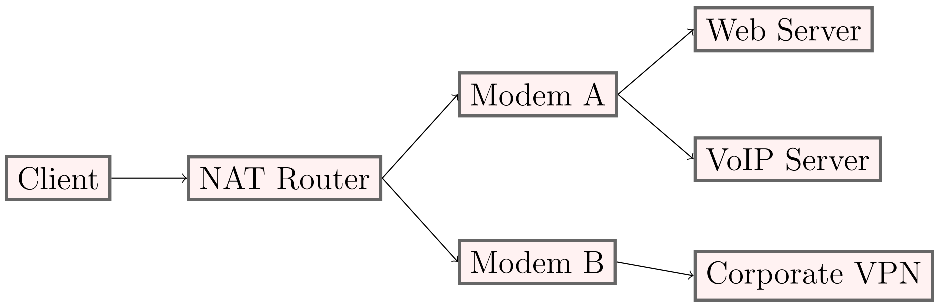


Figure 2: A network with a NAT Router and two modems

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.