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 December 2020

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
December 2020

Proforma

TODO

Table of contents

List of figures **ix**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Existing Work . . . 1

1.2.1 MultiPath TCP . . . 1

1.3 Aims . . . 2

2 Preparation **3**

2.1 Threat Model . . . 3

2.1.1 Transparent Security . . . 3

2.1.2 Portal to Portal Communication . . . 4

2.1.3 Privacy . . . 5

3 Implementation **7**

3.1 TCP . . . 7

3.2 UDP . . . 7

3.2.1 Packet Structure . . . 9

3.2.2 Congestion Control . . . 11

3.3 Security . . . 15

3.3.1 Symmetric Key Cryptography . . . 15

4 Evaluation **17**

4.1 Evaluation Methodology . . . 17

4.2 Line Graphs . . . 18

4.3 Success Criteria . . . 19

4.3.1 Flow Maintained . . . 19

4.3.2 Bidirectional Performance Gains . . . 19

4.3.3 IP Spoofing . . . 19

4.3.4 Security 19
4.3.5 More Bandwidth over Two Equal Connections 19
4.4 Extended Goals 20
4.4.1 More Bandwidth over Unequal Connections 20
4.4.2 More Bandwidth over Four Equal Connections 20
4.4.3 Bandwidth Variation 20
4.4.4 Connection Loss 20
4.4.5 Single Interface Remote Portal 22
4.4.6 Connection Metric Values 22
4.5 Stretch Goals 22
4.5.1 IPv4/IPv6 Support 22
4.5.2 UDP Proxy Datagrams 22
4.5.3 IP Proxy Packets 22
4.6 Performance Evaluation 23
4.6.1 Faster Connections Scaling 23
4.6.2 Number of Connections Scaling 23
4.6.3 Real World Testing 23

5 Conclusions **25**
5.1 Future Work 25

Bibliography **27**

Appendix A Graph Generation Script **29**

Appendix B Outbound Graphs **31**

Project Proposal **33**

List of figures

2.1 Bad actors taking a percentage of packets based on their network speed. . . 4
2.2 TCP Throughput Equation (New Reno) 5

3.1 TCP packet structure 8
3.2 UDP packet structure 10
3.3 ACKs and NACKs responding to sequence numbers 11
3.4 Wireshark dissector 12
3.5 Congestion controller interface 13
3.6 UDP New Reno pseudocode 14
3.7 Message authenticator interface 15

4.1 The structure of graphs throughout this section 18
4.2 The network structure of standard tests 18
4.3 The same test performed both inbound and outbound 19
4.4 The script necessary for the Local Portal to accept packets from a client with the spoofed IP. 20
4.5 Unequal connections compared against equal of both sides 21
4.6 Scaling of equal connections 21
4.7 The scripts necessary to allow the Remote Portal to only use a single interface. 22

Chapter 1

Introduction

1.1 Motivation

Most UK residential broadband speeds receive broadband speeds advertised at between 30Mbps and 100Mbps download (Ofcom 2020, [2]). However, it is often possible to have multiple low bandwidth connections installed. 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 and Low Earth Orbit ISPs such as Starlink.

1.2 Existing Work

1.2.1 MultiPath TCP

MultiPath TCP (Wischik et al. 2011, [3]) 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.

The first reason that MPTCP does not satisfy the motivation for this project is temporal. MPTCP is most effective at creating flows when the device has distinct interfaces or IP addresses. In the case of an IPv4 home connection, it is often the case that a single IPv4 address is provided to the home. This leads to the use of NAT for IPv4 addresses behind the router. If an MPTCP capable device lies behind a NAT router which has two external IPv4 addresses, the device itself will have no knowledge of this.

TODO: IPv6 autoconf wrt. multihoming

Further, it is important to remember legacy devices. Many legacy devices will never support IPv6, and certainly will never support MPTCP. Though it is possible that these devices will not require the performance benefits available from multiple Internet connections, it is likely that they would particularly benefit from a more reliable connection. Being able to apply speed benefits to an entire network without control over every device on it is a significant benefit to the solution provided in this dissertation.

The second reason that MPTCP may not provide the change to the Internet that was once hoped is the UDP based protocols that are coming into existence. Although MPTCP is now making its way into the Linux kernel, many services are switching to lighter UDP protocols such as QUIC. The most interesting example of this is HTTP/3, which was previously known as HTTP over QUIC. This shift to application controlled network connections which do not contain unnecessary overhead for each specific application seems to be the direction that the Internet is going in, but suggests that it will be a very long time before even modern applications can benefit from multipathing.

TODO: Find a study on how many of the connections on the Internet are TCP or UDP, particularly over time

1.3 Aims

This project aimed to provide a method of combining a variety of Internet connections, such as the situations listed above.

When combining Internet connections, there are three main measures that one can prioritise: throughput, resilience and latency. This project aimed to provide throughput and resilience at the cost of latency.

Chapter 2

Preparation

2.1 Threat Model

Proxying a network connection via a Remote Portal creates an expanded set of security threats than connecting directly to the Internet via a modem. In this section, I will discuss my analysis of these threats, in both isolation, and compared to the case of connecting directly.

The first focus of this analysis is the transparent security. That is, if the Local Portal is treated as a modem, what security would normally be expected? And for servers communicating with the Remote Portal, what guarantees can they expect of the packets sent and received?

The second focus is the direct interaction between the Local Portal and the Remote Portal. Questions like, does having this system make it easier for someone to perform a Denial of Service attack on the principal?

These security problems will be considered in the context of the success criteria: provide security no worse than not using this solution at all. 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 Transparent Security

A convenient factor of the Internet being an interconnected set of smaller networks is that there are very few guarantees of security. At layer 3, none of anonymity, integrity, privacy or freshness are provided once the packet leaves private ranges, so it is up to the application to ensure its own security on top of this lack of guarantees. For the purposes of this software, this is very useful: if there are no guarantees to maintain, applications can be expected to act correctly regardless of how easy it is for these cases to occur.

Downlink Capacity	Percentage of Packets
25 Mbps	5%
25 Mbps	5%
25 Mbps	5%
(BAD) 425 Mbps	85%

(a) A bad actor with a fast connection taking a percentage of packets.

Downlink Capacity	Percentage of Packets
25 Mbps	25%
25 Mbps	25%
25 Mbps	25%
(BAD) 25 Mbps	25%

(b) A bad actor with an equally slow connection to you taking a percentage of packets.

Fig. 2.1 Bad actors taking a percentage of packets based on their network speed.

Therefore, to maintain the same level of security for applications, this project can simply guarantee that the packets which leave the Remote Portal are the same as those that came in. By doing this, all of the security implemented above Layer 3 will be maintained. This means that 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.

2.1.2 Portal to Portal Communication

Cost

Many Internet connections have caps or cost for additional bandwidth. In a standard network, the control of your cap is physical, in that, if someone wished to increase the load, they would have to physically connect to the modem.

Due to this, it is important that care is taken with regards to cost. The difference is that rather than needing physical access to send data through your connection, all one needs is an Internet connection. A conceivable threat is for someone to send packets to your Remote Portal from their own connection, causing the Portal to forward these packets, and thus using your limited or costly bandwidth.

Denial of Service

If a malicious actor can fool the Remote Portal into sending them a portion of your packets, they are immediately performing an effective Denial of Service on any tunnelled flows relying on loss based congestion control. In figure 2.1a, it can be seen that a bad actor, with a significantly faster connection than you, can cause huge packet loss if the Remote Portal would accept them as a valid Local Portal connection.

However, of much more relevance is 2.1b. Given the TCP throughput equation, shown in figure 2.2, there is an inverse relation between packet loss and throughput of any TCP connections. Assuming a Round Trip Time of $20ms$ and Maximum Segment Size of 1460,

$$Throughput = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}} \tag{2.1}$$

Fig. 2.2 TCP Throughput Equation (New Reno)

packet loss of 25% limits the maximum TCP throughput to approximately $1.17Mbps$. In fact, due to this relation, a packet loss of even 1% leads to a maximum throughput of approximately $5.84Mbps$. This means that even a small packet loss can have a drastic effect on the performance of the connection as a whole, and thus makes Remote Portals an effective target for Denial of Service attacks. Care must be taken that all Local Portal connections are from the intended subject.

2.1.3 Privacy

Though the packets leaving a modem have no reasonable expectation of privacy, having the packets enter the Internet at two points does increase this vector. For example, if a malicious actor 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, due to the amount of packet loss caused. 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.

Chapter 3

Implementation

3.1 TCP

The base implementation is built on TCP. TCP provides congestion control and flow control, which are all that is necessary for this form of greedy load balancing, and therefore solves almost all of the issues given here. To implement such a solution on TCP, the only difference that needs to be made is punctuating the connection. As TCP provides a byte stream and not distinct datagrams, a distinction must be made between the packets. One option is to use a punctuating character, though this would reduce the character set of the packets, and therefore require escape sequences in the packets. The second option is to read the length of the packets and then read the correct amount of data from the stream.

My implementation uses the second option, of punctuating the stream by providing the length of each packet. Although the IP packets do provide their length internally, I kept the TCP flow as flexible as possible. That is, it is kept as simple as possible, so that it doesn't have to be updated for transmitting any other sort of packets. Therefore, the TCP flow is punctuated by sending the length of the packet before the packet itself within the stream. Then, this number of bytes can be read.

3.2 UDP

To increase the performance of the system, I implemented a UDP method of tunnelling packets, available alongside the TCP method discussed earlier. Using UDP datagrams instead of a TCP flow is a two front approach to increasing performance. Firstly, it removes the issue of head of line blocking, as the protocol does not resend packets when they are not received.

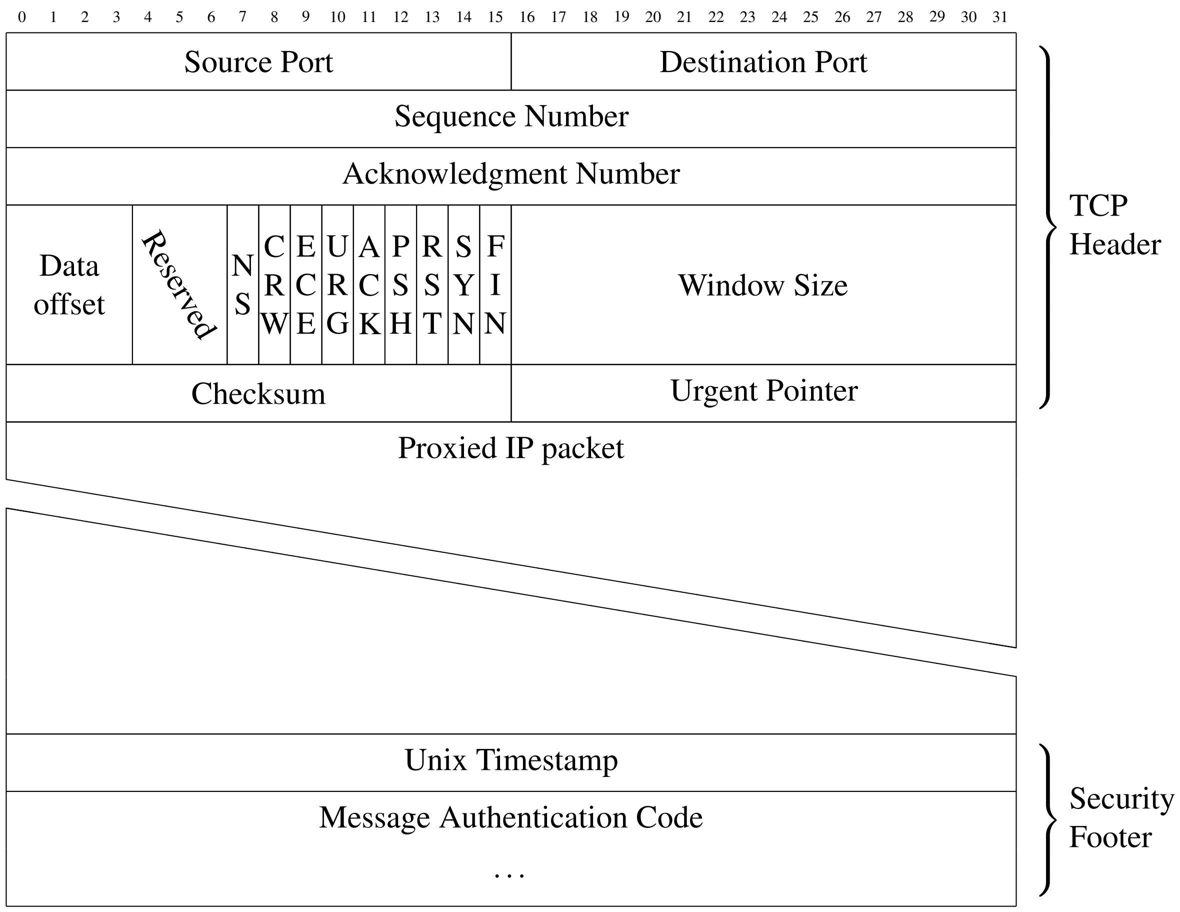


Fig. 3.1 TCP packet structure

Secondly, the datagram design can include less per packet overhead in the form of a header, increasing the efficiency of transmitting packets.

The goal was to create a UDP packet structure that allows for congestion control (and implicit flow control), without the other benefits that TCP provides. This is as the other features of TCP are unnecessary for this project, due to being covered by protocols above Layer 3, which function regardless of the tunnelling.

3.2.1 Packet Structure

The packet structure was decided to allow for effective congestion control and nothing else. This is achieved with a simple 3 part, 12 byte header (shown in figure 3.2). Similarly to TCP, each packet contains an acknowledgement number (ACK) and a sequence number (SEQ). These serve the same purpose as in TCP: providing a method for a congestion controller to know which packets have been received by their partner. However, they are implemented slightly differently. TCP sequence numbers are based on bytes, and as such the sequence number of a packet is the sequence number of the first byte that it contains. As this protocol is designed for transmitting packets, losing part of a packet does not make sense. They will also never be split, as this protocol does not support partial transmission, and as such are atomic. This means that the sequence number can safely represent an individual packet, as opposed to a byte.

In addition to these two fields, a further Negative Acknowledgement (NACK) field is required. Due to TCP's promise of reliable transmission, negative acknowledgements can never occur. Either the sender must resend the packet in question, or the flow is terminated. In my protocol, however, it is necessary that the receiver has a method to provide a discontinuous stream of acknowledgements. If this was attempted without a separate NACK number, it would be required that each ACK number is sent and received individually. This decreases the efficiency and correctness of ACKs, both in terms of missing packets, and having to send at least one packet for every packet received.

The benefit of a NACK is demonstrated in figure 3.3. Figure 3.3a shows a series of ACKs for a perfect set of sequence numbers. This is rather pointless, as there is no point to ACKing packets if you never intend to lose any, but is a situation that can occur for large portions of a flow, given good congestion control and reliable networking. Figure 3.3b shows the same ACK system for a stream of sequence numbers with one missing. It can be seen that the sender and receiver reach an impasse: the receiver cannot increase its ACK number, as it has not received packet 5, and the sender cannot send more packets, as its window is full. The only move is for the receiver to increase its ACK number and rely on the sender realising that it took too long to acknowledge the missing packet, though this is unreliable at best.

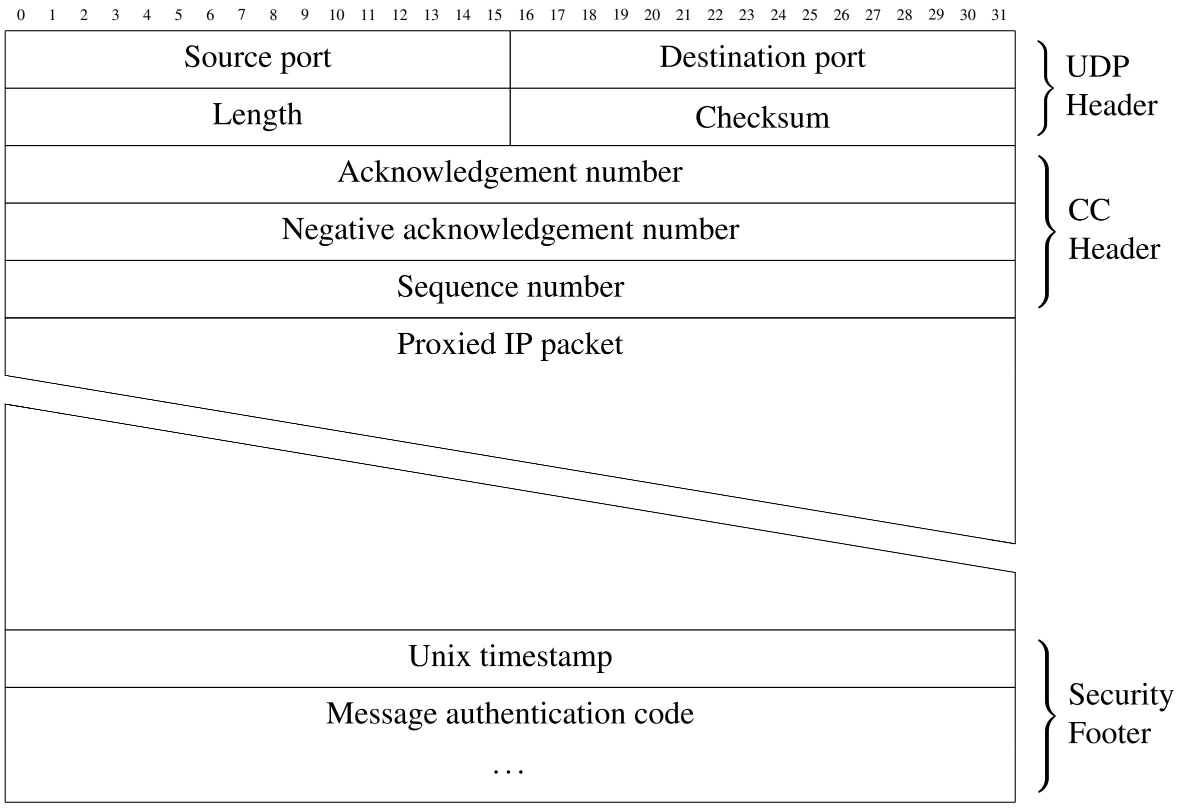


Fig. 3.2 UDP packet structure

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

(a) ACKs responding to in order sequence numbers

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

(b) ACKs responding to a missing sequence number

Sequence	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. 3.3 ACKs and NACKs responding to sequence numbers

Figure 3.3c shows how this same situation can be responded to with a NACK field. After the receiver has concluded that the intermediate packet(s) were lost in transit (a function of RTT, to be discussed further later), it updates the NACK field to the highest lost packet, allowing the ACK field to be increased from one after the lost packet. This solution resolves the deadlock of not being able to increase the ACK number without requiring reliable delivery.

In implementing the UDP based protocol, I spent some time reading packet data in Wireshark[1]. After attempting this with simply the RAW byte data, I wrote a dissector for Wireshark for my protocol. This can be seen in figure 3.4. This is a Lua script that requests Wireshark use the given dissector function for UDP traffic on port 1234 (a port chosen for testing). This extracts the three congestion control numbers from the UDP datagram, showing them in a far easier to read format and allowing more efficient debugging of congestion control protocols.

3.2.2 Congestion Control

To allow for flexibility in congestion control, I started by building an interface (shown in figure 3.5) for congestion controllers. The aim of the interface is to provide the controller with every update that could be used for congestion control, while also providing it every opportunity to set an ACK or NACK on a packet.

A benefit of the chosen language (Go[2] is the powerful management of threads of execution, or Goroutines. This is demonstrated in the interface, particularly the method `Sequence() uint32`. This method expects a congestion controller to block until it can provide the packet

[1]https://wireshark.org
[2]https://golang.org

```
mpbl3p_udp = Proto("mpbl3p_udp", "Multi Path Proxy Custom UDP")

ack_F = ProtoField.uint32("mpbl3p_udp.ack", "Acknowledgement")
nack_F = ProtoField.uint32("mpbl3p_udp.nack", "Negative Acknowledgement")
seq_F = ProtoField.uint32("mpbl3p_udp.seq", "Sequence Number")
time_F = ProtoField.absolute_time("mpbl3p_udp.time", "Timestamp")
proxied_F = ProtoField.bytes("mpbl3p_udp.data", "Proxied Data")

mpbl3p_udp.fields = { ack_F, nack_F, seq_F, time_F, proxied_F }

function mpbl3p_udp.dissector(buffer, pinfo, tree)
    if buffer:len() < 20 then
        return
    end

    pinfo.cols.protocol = "MPBL3P_UDP"

    local ack = buffer(0, 4):le_uint()
    local nack = buffer(4, 4):le_uint()
    local seq = buffer(8, 4):le_uint()

    local unix_time = buffer(buffer:len() - 8, 8):le_uint64()

    local subtree = tree:add(mpbl3p_udp, buffer(), "Multi Path Proxy Header, SEQ: "
    ↪  .. seq .. " ACK: " .. ack .. " NACK: " .. nack)

    subtree:add(ack_F, ack)
    subtree:add(nack_F, nack)
    subtree:add(seq_F, seq)
    subtree:add(time_F, NSTime.new(unix_time:tonumber()))
    if buffer:len() > 20 then
        subtree:add(proxied_F, buffer(12, buffer:len() - 12 - 8))
    end
end

DissectorTable.get("udp.port"):add(1234, mpbl3p_udp)
```

Fig. 3.4 Wireshark dissector

```go
package udp

import "time"

type Congestion interface {
	Sequence() uint32
	ReceivedPacket(seq uint32)

	ReceivedAck(uint32)
	NextAck() uint32

	ReceivedNack(uint32)
	NextNack() uint32

	AwaitEarlyUpdate(keepalive time.Duration) uint32
	Reset()
}
```

Fig. 3.5 Congestion controller interface

with a sequence number for dispatch. Given that the design runs each producer and consumer in a separate Goroutine, this is an effective way to synchronise the packet sending with the congestion controller, and should be effective for any potential method of congestion control.

New Reno

The first congestion control protocol I implemented is based on TCP New Reno. It is a well understood and powerful congestion control protocol. The pseudocode for the two most interesting functions are shown in figure 3.6.

My implementation of New Reno functions differently to the TCP version, given that it responds with NACKs instead of retransmits. In TCP, updating the ACK is similar - the ACK sent is the highest ACK available that remains a continuous stream. The interesting part is visible when the controller decides to send a NACK. Whenever a hole is seen in the packets waiting to be acknowledged, the delay of the minimum packet waiting to be sent is checked. If the packet has been waiting for more than a multiple of the round trip time, chosen presently to be $3 * RTT$, the NACK is updated to one below the next packet that can be sent, indicating that a packet has been missed. The ACK can then be incremented from the next available.

A point of interest is the `acksToSend` data structure. It can be seen that three methods are required: `Min()`, `PopMin()` and `Insert()` (in a section of code not shown in the pseudocode). A data structure that implements these methods particularly efficiently is the binary heap, pro-

```
def findAck(start):
  ack = start
  while acksToSend.Min() == ack+1:
          ack = acksToSend.PopMin()
  return ack

def updateAckNack(lastAck, lastNack):
  nack = lastNack
  ack = findAck(lastAck, acksToSend)
  if ack == lastAck:
          if acksToSend.Min().IsDelayedMoreThan(NackTimeout):
                  nack = acksToSend.Min() - 1
                  ack = findAck(acksToSend.PopMin(), acksToSend)
  return ack, nack

def ReceivedNack(nack):
  if !nack.IsFresh():
           return
  windowSize /= 2

def ReceivedAck(ack):
  if !ack.IsFresh():
          return
  if slowStart:
          windowSize += numberAcked
  else:
          windowCount += numberAcked
          if windowCount >= windowSize:
                  windowSize += 1
                  windowCount -= windowSize
```

Fig. 3.6 UDP New Reno pseudocode

```go
package proxy

type MacGenerator interface {
	CodeLength() int
	Generate([]byte) []byte
}

type MacVerifier interface {
	CodeLength() int
	Verify(data []byte, sum []byte) error
}
```

Fig. 3.7 Message authenticator interface

viding Min in $O(1)$ time, with Insert and PopMin in $O(logn)$ time. Therefore, I implemented a binary heap to store the ACKs to send.

3.3 Security

The security in this solution is achieved by providing a set of interfaces for potential cryptographic systems to implement. This can be seen in figure 3.7. As with all interfaces, the goal here was to create a flexible but minimal interface.

As far as is possible, the security of the application relies on external libraries. Although an interesting exercise, implementing security algorithms directly from papers is far more likely to result in errors and thus security flaws. Due to this, I will be using trusted and open source libraries for the scheme I have chosen.

3.3.1 Symmetric Key Cryptography

When providing integrity and authentication for a message, there are two main choices: a Message Authentication Code (MAC) or signing.

TODO: Finish this section.

BLAKE2s

The shared key algorithm I chose to implement is BLAKE2s[1]. It is extremely fast (comparable to MD5) while remaining cryptographically secure. Further to this, BLAKE2s is available in the Go crypto library[3], which is a trusted and open source implementation.

[3]https://github.com/golang/crypto

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

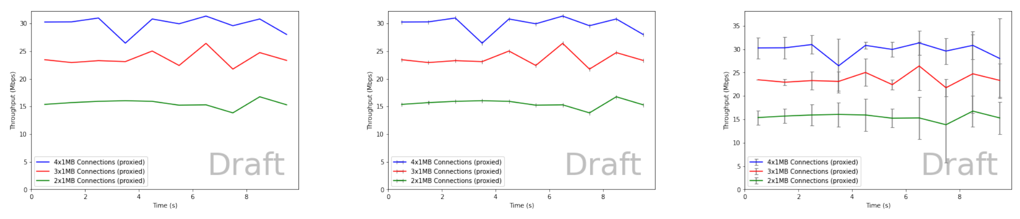
I performed my experiments on a local Proxmox[1] server. To encourage frequent and thorough testing, a harness was built in Python, allowing tests to be added easily and repeated with any code changes.

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



(a) No error bars (b) X error bars (c) Y error bars

Fig. 4.1 The structure of graphs throughout this section

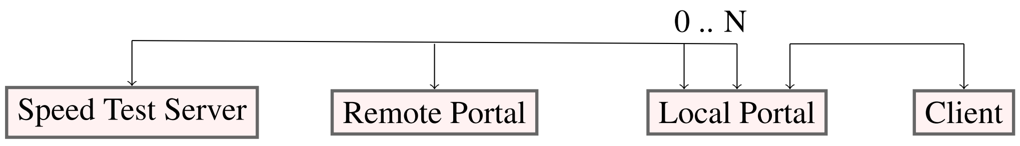


Fig. 4.2 The network structure of standard tests

4.2 Line Graphs

The majority of data presented in this section will be in the form of line graphs. These are generated in a consistent format, using a script found in appendix A.

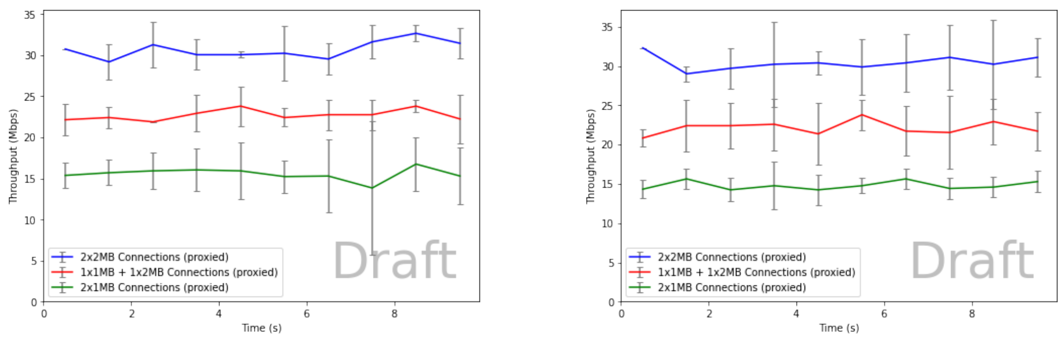
In figure 4.1, examples are shown of the same graph without any error bars, with error bars on the X axis, and with error bars on the Y axis. Error bars for the X axis are plotted as the range of all of the results, while error bars on the Y axis are plotted as $1.5 * \sigma$, where σ represents the standard deviation of the results.

In figure 4.1b, it is shown that the range of the timestamps provided is incredibly tight. For this reason, I will not be including error bars in the X axis on the graphs shown from this point onwards.

In figure 4.1c, it can be seen that the error bars on the Y axis are far more significant. Thus, error bars will continue to be included in the Y axis.

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 is repeated until the coefficient of variance (σ/μ, where μ is the arithmetic mean and σ the standard deviation) is below a desired level, or too many attempts have been completed. The number of attempts taken for each series will be shown in the legend of each graph.

The network structure of all standard tests is shown in figure 4.2. 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.



(a) The inbound graph

(b) The outbound graph

Fig. 4.3 The same test performed both inbound and outbound

4.3 Success Criteria

4.3.1 Flow Maintained

TODO.

4.3.2 Bidirectional Performance Gains

The performance gains measured are visible in both directions (inbound and outbound to the client). The graphs shown in this evaluation section are inbound unless stated otherwise, with the outbound graphs being available in Appendix B.

Figure 4.3 shows two graphs of the same test - one for the inbound performance and one for the outbound. It can be seen that both graphs show the same shape.

4.3.3 IP Spoofing

This goal was to ensure the Client could use its network interface as if it really had that IP. This is achieved through Policy Based Routing. Example scripts are shown in figure 4.4. Linux also requires the kernel parameter `net.ipv4.ip_forward` to be set to 1.

4.3.4 Security

TODO.

4.3.5 More Bandwidth over Two Equal Connections

TODO.

```
#IPv4 Forwarding
sysctl -w net.ipv4.ip_forward=1

# Route packets from the remote portal address on the client interface via the
↪  tunnel
ip route flush 12
ip route add table 12 to 1.1.1.0/24 via 172.19.152.2 dev nc0
ip rule add from 1.1.1.3 iif eth3 table 12 priority 12

# Route packets to the remote portal address out of the client interface
ip route flush 13
ip route add table 13 to 1.1.1.3 dev eth3
ip rule add to 1.1.1.3 table 13 priority 13
```

Fig. 4.4 The script necessary for the Local Portal to accept packets from a client with the spoofed IP.

4.4 Extended Goals

4.4.1 More Bandwidth over Unequal Connections

This is demonstrated by showing that $1x1MB + 1x2MB$ connections can exceed the performance of $2x1MB$ connections. The results for this can be seen in figure 4.5, compared against $2x2MB$ and $1x2MB$. It can be seen that the uneven connections fall between the two, which is as expected.

4.4.2 More Bandwidth over Four Equal Connections

This criteria is about throughput increasing with the number of equal connections added. It is demonstrated by comparing the throughput of $2x1MB$, $3x1MB$ and $4x1MB$ connections. This can be seen in figure 4.6a. A further example is provided of $2x2MB$, $3x2MB$ and $4x2MB$ in figure 4.6b.

4.4.3 Bandwidth Variation

TODO.

4.4.4 Connection Loss

TODO.

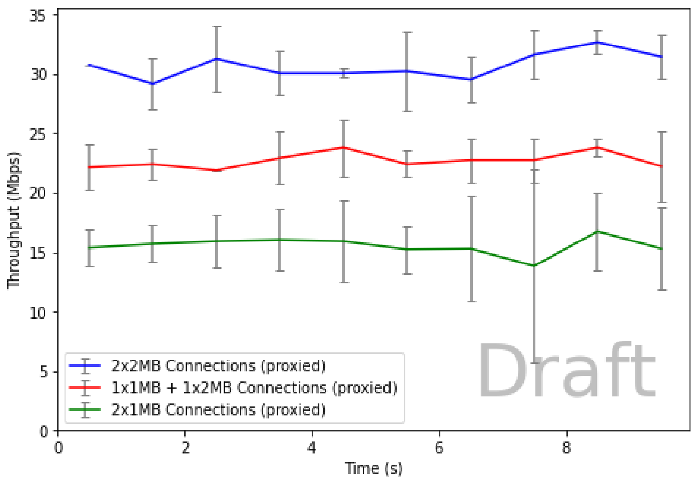
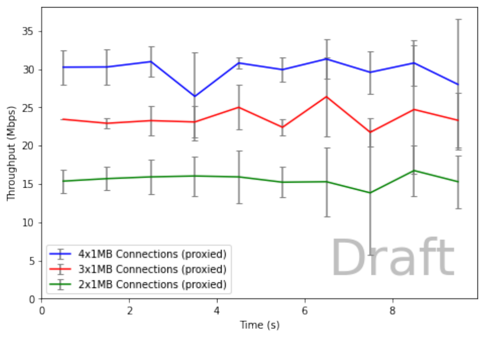
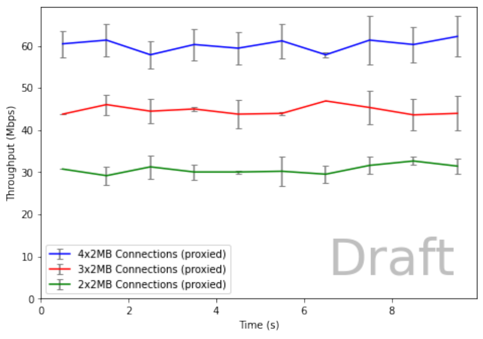


Fig. 4.5 Unequal connections compared against equal of both sides



(a) 1MB connections



(b) 2MB connections

Fig. 4.6 Scaling of equal connections

```
# IPv4 Forwarding
sysctl -w net.ipv4.ip_forward=1
sysctl -w net.ipv4.conf.eth0.proxy_arp=1

# Deliberately break local routing
ip rule add from all table local priority 20
ip rule del 0 || true

# Route packets to the interface but for nc to this host
ip rule add to 1.1.1.3 dport 1234 table local priority 9

# Route packets to the interface but not for nc via the tunnel
ip route flush 10
ip route add table 10 to 1.1.1.3 via 172.19.152.3 dev nc0
ip rule add to 1.1.1.3 table 10 priority 10
```

Fig. 4.7 The scripts necessary to allow the Remote Portal to only use a single interface.

4.4.5 Single Interface Remote Portal

The single interface Remote Portal is achieved using a similar set of commands to IP Spoofing. The majority of the work is again done by policy based routing, with some kernel parameters needing to be set too. A sample script is shown in figure 4.7.

4.4.6 Connection Metric Values

Not implemented yet.

4.5 Stretch Goals

4.5.1 IPv4/IPv6 Support

The project is only tested with IPv4.

4.5.2 UDP Proxy Datagrams

TODO

4.5.3 IP Proxy Packets

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

4.6 Performance Evaluation

The discussion of success criteria above used 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.

4.6.1 Faster Connections Scaling

TODO

4.6.2 Number of Connections Scaling

TODO

4.6.3 Real World Testing

TODO

Chapter 5

Conclusions

5.1 Future Work

The most interesting future work on multi-homed devices would focus on adding additional features to gateways.

Work on the most effective method of allowing a gateway to inform a device behind it that it is worth adding additional MPTCP subflows.

Work on gateways understanding the Layer 4 concepts of MPTCP and adapting their load balancing algorithms to ensure that multiple subflows of the same MPTCP flow are split appropriately between the available links.

Work on gateways that understand MPTCP to take a non-MPTCP flow and transparently convert it into a MPTCP flow at the gateway, and back again as it reaches the device behind.

Work on IPv6 multi-homing to more effectively inform devices behind it of when they have multiple homes.

TODO: Check, for all of these, whether they should actually be in past work. Particularly the IPv6 multi-homing one.

Bibliography

[1] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. (2013). BLAKE2: Simpler, Smaller, Fast as MD5. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Jacobson, M., Locasto, M., Mohassel, P., and Safavi-Naini, R., editors, *Applied Cryptography and Network Security*, volume 7954, pages 119–135. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

[2] Ofcom (2020). The performance of fixed-line broadband delivered to UK residential customers.

[3] Wischik, D., Raiciu, C., Greenhalgh, A., and Handley, M. (2011). Design, implementation and evaluation of congestion control for multipath TCP. page 14.

Appendix A

Graph Generation Script

```python
from itertools import cycle
import matplotlib.pyplot as plt

def plot_iperf_results(
        series: Dict[str, StandardTest],
        title: str = None,
        direction = 'inbound',
        error_bars_x=False,
        error_bars_y=False,
        filename=None,
        start_at_zero=True,
):
    if filename in ['png', 'eps']:
        filename = 'graphs/{}{}{}{}.{}'.format(
            'I' if direction == 'inbound' else 'O',
            'Ex' if error_bars_x else '',
            'Ey' if error_bars_y else '',
            ''.join(['S{}-{}'.format(i,x.name()) for (i, x) in
             ↪  enumerate(series.values())]),
            filename,
        )

    series = {
        k: (directionInbound if direction == 'inbound' else
         ↪  directionOutbound)[v.name()] for (k, v) in series.items()
    }

    cycol = cycle('brgy')

    fig = plt.figure()
    axes = fig.add_axes([0,0,1,1])

    if title is not None:
        axes.set_title(title, pad=20.0 if True in [len(x.test.events) > 0 for x in
        ↪  series.values()] else None)

    axes.set_xlabel('Time (s)')
    axes.set_ylabel('Throughput (Mbps)')

    for k, v in series.items():
        data = v.summarise()

        axes.errorbar(
            data.keys(),
            [x/1e6 for x in data.values()],
            xerr=([x[0] for x in v.time_range().values()], [x[1] for x in
            ↪  v.time_range().values()]) if error_bars_x else None,
            yerr=[x*1.5/1e6 for x in v.standard_deviation().values()] if
            ↪  error_bars_y else None,
            capsize=3,
            ecolor='grey',
            color=next(cycol),
            label=k,
        )

    legend = axes.legend()

    if start_at_zero:
        axes.set_ylim(bottom=0)
        axes.set_xlim(left=0)

    if False:
        for k, v in events.items():
            axes.axvline(k, linestyle='--', color='grey')
            axes.annotate(v, (k, 1.02), xycoords=axes.get_xaxis_transform(),
            ↪  ha='center')

    if filename is not None:
        fig.savefig(filename, bbox_extra_artists=(legend,), bbox_inches='tight',
        ↪  pad_inches=0.3)
```

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.

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.

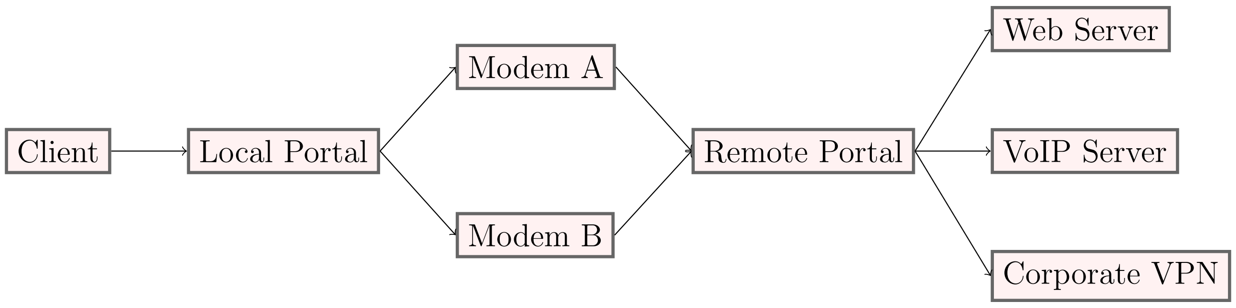


Figure 1: A network applying this proxy

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

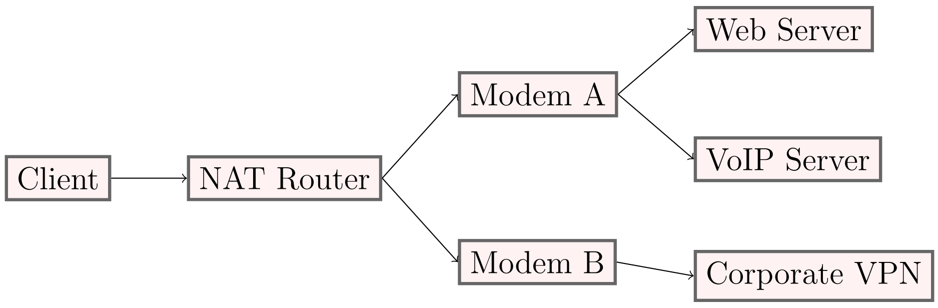


Figure 2: A network with a NAT Router and two modems

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.