

*Computing Science and Mathematics*  
*University of Stirling*

**Binder**

**Duncan Eastoe**

*Submitted in partial fulfilment of the requirements for the degree of*  
**B.Sc. in Computing Science**

*April 2014*

# Abstract

Community wireless mesh networks are being used to share access to the Internet. Although such networks may contain multiple Internet gateways they cannot be utilised simultaneously to achieve bandwidth aggregation and seamless fault tolerance. My project is part of the Binder project, which aims to solve these issues by logically aggregating all Internet gateways in a network, wherever they are located. This will improve Internet access to areas provided by such networks.

I worked on updating software that makes up the Binder system, implementing new solutions to solve problems affecting Binder and performing measurements of the Binder system. These contributions to Binder will help increase its uptake and acceptance, hopefully resulting in its deployment on networks other than that which spawned the project.

I have updated Binder's modifications to the Linux implementation of MPTCP and begun the process of providing these alterations upstream. I have also developed a Linux kernel module that performs stripping of IP Options, which are used within a Binder network, to prevent Binder traffic experiencing issues on the Internet. Furthermore, I have written tools that help developing utilities for and running tests using the D-ITG traffic generator. Finally, I have designed and performed tests on a series of wireless mesh networks.

# Attestation

I understand the nature of plagiarism, and am aware of the University's policy on this. I certify that this dissertation reports original work by me during my University project except for the following:

- The IP Option stripping solution supporting code, including build files, are largely based on the contents of the Writing Netfilter Modules book [33].
- The original Binder MPTCP patch was written by Luca Boccassi as part of his Masters Thesis.
- Although an original Java implementation, the operations of JITGApi are based upon those in the official C++ ITGApi.
- Figures 1.1 and 1.2 were obtained from the Binder paper [21].

**Signature:**

**Date:**

# Acknowledgements

I would like to give thanks to Marwan Fayed, Luca Boccassi, Arsham Farshad and Christoph Paasch for their help in completing my project. Specifically:

- Marwan for his invaluable guidance and vision on the project as a whole, and beyond
- Luca for his help with working on the MPTCP patch and setting up the testbed
- Arsham for his assistance in configuring the OpenWRT wireless nodes at the Informatics Forum, University of Edinburgh
- Christoph for his help in the MPTCP upstreaming process

Finally, I would like to thank Gay, Richard, Em, Pete, Phil, Fi and Chris for all their support.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Attestation</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Context . . . . .	1
1.1.1 Mesh Networks . . . . .	1
1.1.2 Advantages . . . . .	1
1.1.3 Design . . . . .	2
1.2 Scope and Objectives . . . . .	2
1.2.1 Add Binder support to MPTCP . . . . .	2
1.2.2 Implement IP Option Stripping . . . . .	4
1.2.3 Testing Binder . . . . .	4
1.3 Achievements . . . . .	5
1.4 Overview of Dissertation . . . . .	5
1.4.1 State-of-The-Art . . . . .	5
1.4.2 Implementation of LSRR in MPTCP . . . . .	5
1.4.3 D-ITG Applications . . . . .	5
1.4.4 IP Option Stripping . . . . .	5
1.4.5 Measurements . . . . .	5
<b>2 State-of-The-Art</b>	<b>6</b>
2.1 Link Layer Aggregation . . . . .	6
2.2 Network Layer Aggregation . . . . .	7
2.3 Transport Layer Aggregation . . . . .	7
2.4 Foundation . . . . .	8
2.4.1 Loose Source Record Routing . . . . .	8
2.4.2 OpenVPN . . . . .	8
2.5 Supporting . . . . .	8
2.5.1 D-ITG . . . . .	8
2.5.2 tcp-intercept . . . . .	9

<b>3</b>	<b>Scripts and Script Management</b>	<b>10</b>
3.1	Rationale . . . . .	10
3.1.1	Example: binder-scan . . . . .	10
3.2	sys-vars . . . . .	11
<b>4</b>	<b>Implementation of LSRR in MPTCP</b>	<b>13</b>
4.1	Introduction . . . . .	13
4.2	Development Testbed . . . . .	13
4.3	Scripts . . . . .	15
4.3.1	host-setup . . . . .	15
4.3.2	mptcp-setup . . . . .	15
4.3.3	start-virtual-instances . . . . .	15
4.4	Binder MPTCP Patch . . . . .	16
4.4.1	Description as of September 2013 . . . . .	16
4.4.2	Rebasing . . . . .	16
4.4.3	IPv6 . . . . .	17
4.5	Upstreaming . . . . .	19
4.5.1	Path Manager . . . . .	19
4.5.2	Re-Implementation . . . . .	20
<b>5</b>	<b>D-ITG Applications</b>	<b>23</b>
5.1	Introduction . . . . .	23
5.2	JITGApi . . . . .	23
5.2.1	Rationale . . . . .	24
5.2.2	Implementation . . . . .	24
5.2.3	Documentation . . . . .	27
5.2.4	Testing . . . . .	27
5.3	ITGController . . . . .	28
5.3.1	Implementation . . . . .	28
5.3.2	Example Config File . . . . .	30
<b>6</b>	<b>IP Option Stripping</b>	<b>32</b>
6.1	Introduction . . . . .	32
6.2	Development Testbed . . . . .	33
6.3	Implementation . . . . .	34
6.3.1	iptables vs nftables . . . . .	34
6.3.2	Additional Kernel Module - xt_IPOPTSTRIP . . . . .	34
6.3.3	Additional iptables Extension - libxt_IPOPTSTRIP . . . . .	36
6.4	Scripts . . . . .	36
6.4.1	reload . . . . .	36
6.4.2	router-setup . . . . .	36
6.4.3	wireshark . . . . .	37
6.4.4	remote-wireshark . . . . .	37

<b>7</b>	<b>Measurements</b>	<b>38</b>
7.1	Testbed . . . . .	38
7.1.1	Introduction . . . . .	38
7.1.2	Setup . . . . .	38
7.1.3	Hardware Upgrades . . . . .	39
7.1.4	OpenWRT Node Configuration . . . . .	39
7.1.5	Wireless Frequencies and Channel Allocations . . . . .	39
7.1.6	Routing . . . . .	39
7.1.7	Iteration One . . . . .	40
7.1.8	Iteration Two . . . . .	42
7.1.9	Iperf Test Procedure . . . . .	42
7.1.10	Iteration Three . . . . .	42
7.1.11	Iteration Four . . . . .	45
7.1.12	Binder . . . . .	51
7.1.13	Concurrent Tests . . . . .	52
7.2	TCP Splitting . . . . .	53
7.3	Scripts . . . . .	53
7.3.1	Wireless Nodes . . . . .	53
7.3.2	Iperf . . . . .	55
7.3.3	D-ITG . . . . .	55
7.3.4	TCP Splitting . . . . .	57
<b>8</b>	<b>Conclusion</b>	<b>58</b>
8.1	Evaluation . . . . .	58
8.1.1	Implementation of LSRR in MPTCP . . . . .	58
8.1.2	D-ITG Applications . . . . .	58
8.1.3	IP Option Stripping . . . . .	58
8.1.4	Measurements . . . . .	59
8.2	Future Work . . . . .	59
8.2.1	Re-inserting Stripped IP Options . . . . .	59
8.2.2	Routing Protocol for Low-Power and Lossy Networks . . . . .	60
8.2.3	OLSR . . . . .	60
8.2.4	ITGController Unit Tests . . . . .	60
8.2.5	Measurements . . . . .	60

# List of Figures

1.1	Software architecture of Binder . . . . .	3
1.2	Example network topology with Binder . . . . .	3
4.1	IPv4 QEMU Testbed Topology . . . . .	14
4.2	IPv6 QEMU Testbed Topology . . . . .	14
4.3	Example routing topology . . . . .	18
5.1	Payload structure of an ITGSend Response Packet . . . . .	25
5.2	JITGApi Class Diagram . . . . .	26
5.3	ITGController Class Diagram . . . . .	29
5.4	Writing an ITGController config file in Emacs . . . . .	30
6.1	IPOPTSTRIP VirtualBox Testbed Topology . . . . .	33
7.1	Testbed Diagram Key . . . . .	40
7.2	Testbed Configuration 1 . . . . .	41
7.3	Testbed Configuration 2 . . . . .	43
7.4	Testbed Configuration 3 . . . . .	44
7.5	Testbed Configuration 3 - All Interfaces TCP Results (Mbps) . . . . .	44
7.6	Testbed Configuration 3 - Req. Interfaces TCP Results (Mbps) . . . . .	45
7.7	Testbed Configuration 4.1 . . . . .	46
7.8	Testbed Configuration 4.1 - TCP throughput (Mbps) . . . . .	46
7.9	Testbed Configuration 4.1 alteration - TCP throughput (Mbps) . . . . .	46
7.10	Testbed Configuration 4.2 . . . . .	47
7.11	Testbed Configuration 4.2 - TCP throughput (Mbps) . . . . .	47
7.12	Testbed Configuration 4.2 - UDP throughput (Mbps) . . . . .	47
7.13	Testbed Configuration 4.2 alteration - TCP throughput (Mbps) . . . . .	48
7.14	Testbed Configuration 4.2 alteration - UDP throughput (Mbps) . . . . .	48
7.15	Testbed Configuration 4.3 . . . . .	48
7.16	Gateway-Server TCP throughput (Mbps) . . . . .	49
7.17	Testbed Configuration 4.3 - TCP throughput (Mbps) . . . . .	49
7.18	Testbed Configuration 4.3 alteration - UDP throughput (Mbps) . . . . .	49
7.19	Testbed Configuration 4.4 . . . . .	50
7.20	Testbed Configuration 4.4 - Branch TCP throughput without Binder (Mbps) . . . . .	51
7.21	Testbed Configuration 4.4 - Branch TCP throughput with Binder (Mbps) . . . . .	51
7.22	Testbed Configuration 4.4 - Branch UDP throughput with Binder (Mbps) . . . . .	52
7.23	Testbed Configuration 4.4 - Concurrent branch TCP throughput (Mbps) . . . . .	52



# Chapter 1

## Introduction

My project concerns continuing work on the Binder project. The original authors of Binder are Luca Boccassi, Dr. Marwan Fayed and Prof. Mahesh Marina. The paper may be found at [21].

### 1.1 Background and Context

#### 1.1.1 Mesh Networks

Binder is a system that performs aggregation of Internet gateways within community mesh networks. These networks are expanding world wide [53] as a grassroots effort to bring connectivity to areas traditionally left cut off.

A mesh network consists of nodes which communicate directly with the other nodes in the network. Typically this is achieved by the nodes using directional antennae positioned strategically to build out the network. At one edge of the network will be a gateway providing a connection to the Internet. Access points co-located with communities allow the connection of commodity client equipment to the network. Multiple Internet gateways may be present although they cannot be utilised effectively from anywhere on the network.

A network closely connected with the project is HUBS (High-Speed Universal Broadband for Scotland) [36] which maintains a series of networks in rural Scotland. HUBS grew out of the Tegola research project which involved a test mesh network built around the communities of Loch Hourn.

Binder performs aggregation of gateways wherever they are located on the network, allowing clients to make full use of the network rather than being restricted to a statically configured gateway.

#### 1.1.2 Advantages

Binder provides three main advantages to users of these networks:

**Bandwidth aggregation** allows users to make connections using all the combined bandwidth of the available gateways on the network. This happens even for single users and single flows. In other words the bandwidth increase drastically improves all activity on the network, not just heavy use.

**Fault tolerance** is provided in a seamless way. Since outbound traffic from the network travels through multiple gateways there is protection should any of the gateways or routes to them become unavailable.

**Load balancing** ensures that the network traffic is spread evenly across all of the available gateways. This ensures that no single gateway is swamped with traffic from the entire network which might result in loss of connection.

### 1.1.3 Design

Binder consists of several components to achieve the overall effect of bandwidth aggregation. Co-located with each access point, at the end-user edge of the mesh network, is a relay machine. Traffic from the access point is routed to the relay. At the relay, the traffic is routed through an OpenVPN [46] tunnel.

The OpenVPN tunnel is split into multiple sub-flows using MPTCP (MultiPath TCP) [48], Binder modifies MPTCP which allows us to influence the routing decisions that are performed on each sub-flow. By ensuring that each sub-flow will traverse a particular hop on the network we can transfer traffic across all the gateways available in the network. This is done by adding a Loose Source Record Routing (LSRR) IP Option to each packet in a sub-flow. The option specifies a path of gateways that the packet must traverse to reach the destination. The routing algorithm remains unchanged and gateways anywhere on the network may be used, this is a key design feature in Binder.

The OpenVPN tunnel transfers the traffic to a proxy server which exists on the wider Internet. Here, the traffic is forwarded onto the Internet after performing Network Address and Port Translation. This ensures that all connections from the mesh network, which are not directly addressable from the public Internet, appear to come from the proxy server. Return packets from the initial request will be returned to the proxy server where NAT will be reversed and the traffic will be sent back to the relay over the tunnel. Again making use of the gateways used in the outbound connection.

Binder is designed to be a practical and easily deployable solution. There is no modification of end client devices required and no specific knowledge of the system is needed to use a network implementing Binder. Clients make use of the benefits provided to the network by Binder by connecting to their local access point.

The software and network structure of Binder described above is shown in figures 1.1 and 1.2.

## 1.2 Scope and Objectives

My project has a clearly defined set of objectives that will help advance the Binder project. These objectives fall into three high-level categories:

### 1.2.1 Add Binder support to MPTCP

This will involve updating the current support for Binder in the IETF standardised codebase for the Linux kernel implementation of MPTCP as well as introducing compatibility of Binder with IPv6 networks.

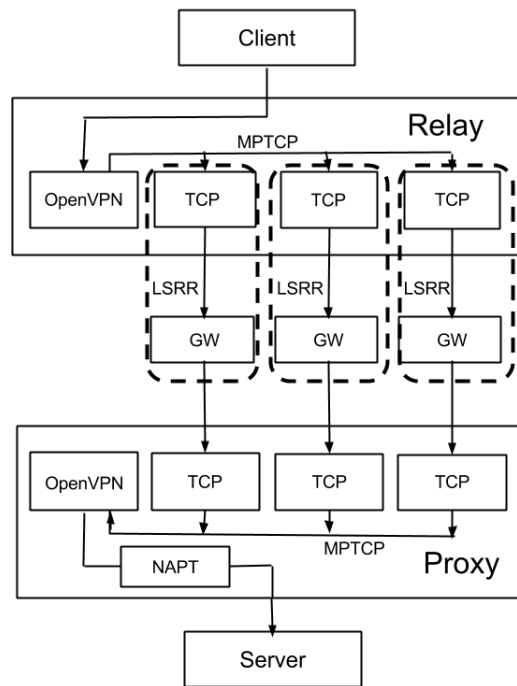


Figure 1.1: Software architecture of Binder

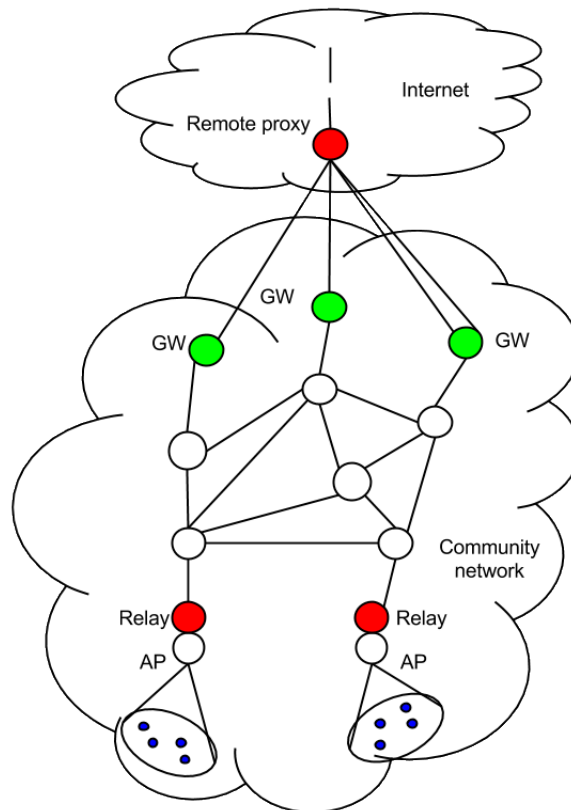


Figure 1.2: Example network topology with Binder

## **MPTCP patch rebase**

Binder's use of MPTCP is subject to a patch against the upstream code which allows Loose Source Record Routing IP Options to be set. This patch was most recently updated in 2012 against an earlier kernel version. I will update the patch to match the latest upstream MPTCP code, on the latest 3.x Linux kernel. Throughout the project I will also regularly pull in the latest changes from upstream and test that the patch still functions as expected. Keeping in sync with upstream will make the submission of code for inclusion upstream much easier in the future.

## **IPv6 MPTCP patch**

The current patch against MPTCP allows Binder to influence the routing decisions made for IPv4 traffic. With increasing demand for IPv6 capable solutions it is important that Binder is also compatible with IPv6 networks. This objective will involve investigating the possible ways of influencing the routing decisions made on IPv6 networks. The second phase will involve the implementation of the solution into the MPTCP code. This is where understanding of the previous work and patch will be required.

### **1.2.2 Implement IP Option Stripping**

The Binder system uses the Loose Source Record Routing (LSRR) IP Option in order to perform source routing on network traffic. Traffic with such properties may be dropped when experienced on the Internet, posing a problem when deploying Binder. By stripping the options when traffic leaves the Binder network this can be avoided.

### **1.2.3 Testing Binder**

The second objective will be to perform extensive testing of Binder under different conditions. This will take place in a more realistic environment, utilising a real world mesh network. The results will allow evaluation and characterisation of Binder's performance.

#### **Testing on a mesh network**

To date, the testing of Binder has been limited to a laboratory setup of a single client and single server machine. These were separated by routers and wired hops. Simulated loss and latency was introduced to attempt to re-create more realistic conditions. However, this can only be taken so far and testing of Binder on a real mesh network will be important to ensure that the solution is effective. Realistic traffic generators will also be used to try and discover real world application performance.

#### **Investigate TCP splitting**

The OpenVPN tunnel runs over TCP and then TCP connections will run through the tunnel. Sending TCP traffic across TCP tunnels in this manner can cause undesirable effects when packet loss is experienced [35]. TCP splitting involves terminating a TCP connection and re-establishing it at a middlebox which may remove the need for a TCP tunnel.

## **1.3 Achievements**

In this project I have achieved several objectives:

- Update the support for Binder in MPTCP and begin the upstreaming process
- Start two new projects related to making experiments with D-ITG easier to perform
- Implement a solution to perform IP Option stripping from IPv4 packets on Linux
- Designed and experimented with a number of wireless mesh networks

## **1.4 Overview of Dissertation**

This dissertation will discuss the following:

### **1.4.1 State-of-The-Art**

This section details alternative solutions to Binder as well as explaining in further detail some of the existing software solutions that are used to make up the Binder system.

### **1.4.2 Implementation of LSRR in MPTCP**

This chapter discusses how per sub-flow Loose Source Record Routing has been added to MPTCP in order to support Binder. This is followed by details of the process of upstreaming this work to the MPTCP codebase.

### **1.4.3 D-ITG Applications**

Here I explain two software projects that I have written that extend upon the capabilities of D-ITG, a real traffic generator. This includes the rationale, research that lead to their development and their implementation.

### **1.4.4 IP Option Stripping**

This section discusses how I researched and implemented a way of implementing IP Option stripping on Linux.

### **1.4.5 Measurements**

A description of the testbeds I have designed as well as the experiments and corresponding results are contained within this chapter.

## Chapter 2

# State-of-The-Art

There have been many projects which have tackled the problem of bandwidth aggregation. This has typically been approached from both the link layer, bonding links from node to node, and at the network layer, from host to host. Binder is an example of bandwidth aggregation performed at the transport layer by influencing the network layer.

### 2.1 Link Layer Aggregation

Aggregation of bandwidth at the link layer will usually involve the creation of a single channel by joining together the channels of several different ports on a network device, such as a switch. This is the case in the Link Aggregation Control Protocol (LACP) which is the IEEE 802.1ax standard [37]. This protocol is widely implemented on network devices and provides good fault tolerance should any of the channels fail. Bandwidth aggregation ensures bandwidth is higher than that of individual ports. To use this solution, the two communicating nodes must bond the same type of links. For example, gigabit links must be bonded to other gigabit links.

Commodity software implementations of link layer aggregation are also available. The Linux kernel includes a bonding driver [22] which allows the creation of a new network interface that consists of several network interfaces bonded together. This can be configured in a number of ways to provide a number of different qualities. The driver can be set to use a round-robin scheduler where packets are sent in turn through each interface making up the bonded interface in order. This mode provides fault tolerance, load balancing and aggregation of bandwidth. Alternatively the driver can be set such that it will use the extra interfaces only as a backup, traffic is sent over one of the bonded interfaces until such time that it fails when traffic is sent over the next interface. Another available mode is broadcast, where all traffic is sent over all the bonded links providing fault tolerance but not bandwidth aggregation. Instead, rather inefficient use of bandwidth is made in this mode. The bonding driver is also able to perform aggregation using LACP, ensuring that bonded links all share the same speed and duplex settings. Duplex concerns the ability to communicate in either direction at one time, half-duplex allowing only single-way communications, unlike full-duplex.

Examples of link layer bonding are easily available and provide good fault tolerance and aggregation characteristics. The limitations of these systems in the context of Binder is apparent, however. Link layer bonding assumes that all bonded links are located locally with direct physical communications, this is not realistic in the context of mesh networks. The bonding at the link layer will also only

affect the communications between the two configured nodes. This is not feasible in the use case laid out for Binder since the network beyond the gateways is out of our control.

## 2.2 Network Layer Aggregation

There are many examples of bandwidth aggregation at the network layer. Multi-Gateway Association [41] introduces the concept of a super-gateway into the network which serves as a single point to collect flows from the multiple gateways. This approach makes a number of assumptions, one being that the super-gateway and the separate Internet gateways are connected with wired connections. For Binder this cannot be assumed, the Tegola network grew as the combination of several networks which each had their own gateways. The distances and topology between these gateways mean a wired connection is not possible. Multi-Gateway Association also makes use of algorithms designed specifically for this solution requiring that the end user client devices be modified. Again, this is an unacceptable constraint for Binder, where commodity hardware and software will be used.

Aggregation techniques such as that defined in [24] are more similar to Binder in their use of intermediate nodes to martial traffic across gateways. Here a proxy is used to split and assemble traffic and a scheduling algorithm is defined that takes into account the delay on the path from the proxy to the client and the available bandwidth. This solution has only been tested on UDP traffic. Although a widely used transport layer alternative to TCP with applications in online gaming, video conferencing and streaming, a large amount of traffic still relies on TCP connections, eg. HTTP. There are vast differences in the effects exhibited on a network by UDP and TCP connections. UDP is a best effort and greedy service, making no guarantees whatsoever. UDP traffic does not guarantee to be delivered, nor does it ensure that it will be delivered in order or that it will make fair use of the available bandwidth. A UDP connection may saturate the available bandwidth. TCP connections on the other hand do make guarantees that packets will arrive and that they will be in the order they were sent. The saw tooth bandwidth profile of a TCP connection shows the ramp up algorithms that slowly increase the bandwidth utilisation of a connection and immediately fall back once loss is detected. This ensures that fair use of the bandwidth is maintained. It is unclear how this system would respond to cases of packet loss and re-transmission for TCP connections, particularly the complex scheduling algorithm. Likewise with Multi-Gateway Association, this approach also requires modifications to end user devices.

## 2.3 Transport Layer Aggregation

Binder makes use of MultiPath TCP, a transport layer protocol that opens parallel connections to achieve increased performance. Transport layer protocols need not worry about the network they run on, instead focusing on the delivery of packets between services. Although parallel connections may be opened, either on single or multiple interfaces, this does not guarantee that either bandwidth aggregation or load balancing will be performed unless it is configured on a machine supporting multiple network interfaces. The Binder patch to MPTCP allows us to ensure that all available gateway paths on a network are used, rather than simply relying on the network's routing algorithm to choose gateways for us. This is a key feature that Binder builds on top of MPTCP.

Another transport layer protocol that supports parallel transmission of data in the same connection is Stream Control Transmission Protocol (SCTP) [44]. SCTP provides some features of TCP while

relaxing others, such as the strict in order delivery of packets. Like MPTCP, SCTP also supports the use of multiple interfaces to provide bandwidth aggregation and fault tolerance. MPTCP has the advantage of simply being an extension of TCP. MPTCP enabled clients can communicate with non-MPTCP enabled services and vice-versa, only if both machines support the protocol will an attempt be made to negotiate sub-flows. This is a disadvantage of transport layer techniques for performing bandwidth aggregation, the agnostic attitude to the network infrastructure requires support at the network edge for end to end aggregation.

The Binder modifications to MPTCP ensures that bandwidth aggregation, fault tolerance and load balancing can be achieved without the need for multiple interfaces per gateway and the subsequent restrictions this would impose upon the configuration and routes of the mesh network.

## **2.4 Foundation**

As mentioned above Binder uses a patched version of MPTCP to help perform bandwidth aggregation. Binder also uses a couple of other foundation technologies:

### **2.4.1 Loose Source Record Routing**

Loose source routing is a technique that allows the sender of a packet to influence the network layer by requesting which route it will take through the network. This is performed by providing a list of IP addresses that the packet must visit. On arrival at each router node the packet will be delivered to the next defined address. This is implemented in IPv4 through an IP option which the sender can add to the IP header of a packet.

### **2.4.2 OpenVPN**

OpenVPN is a secure open-source VPN tunnelling solution. This allows the creation of a route between two hosts, as if they were in the same network. In the context of Binder this allows for a logical connection between the relay and proxy components. OpenVPN can be configured to run on top of TCP or UDP packets. This allows it to traverse middle nodes, which may block other types of traffic, without issue. Since the OpenVPN tunnel runs on TCP between the relay and the proxy, which both support MPTCP, the tunnel traffic is transferred with parallel TCP connections.

## **2.5 Supporting**

### **2.5.1 D-ITG**

The Distributed Internet Traffic Generator [23] is a suite of tools, including an API, that allows complex network measurements to be performed across a network. The suite allows for multiple senders and receivers to exist on a network and for them to be controlled remotely. Results are recorded for each flow of traffic sent by the tools. As well as allowing for manual control of the traffic and transmission properties such as send rate and payload size, D-ITG also contains a number of presets such as VoIP, DNS and even Quake3 traffic. D-ITG will be used for performing real world traffic tests on a network with and without Binder support.

The flowgrind traffic generation tool [34] was also considered for use in testing Binder. However,



the main problem was that it only supports generation of TCP traffic, while D-ITG also supports UDP traffic. Also, the available documentation for D-ITG seemed more thorough and easier to pick up. I also decided that the API provided by D-ITG for remote control was an interesting proposition. Both projects, do however already include some form of remote control tool.

### **2.5.2 tcp-intercept**

tcp-intercept [42] is a project that performs TCP splitting, this involves terminating a TCP connection between the source and destination and starting a new connection between the terminator and the destination. This program was specifically written to convert standard TCP connections to MPTCP connections and this is how it will be used as part of an experiment in changing how Binder functions.

## Chapter 3

# Scripts and Script Management

### 3.1 Rationale

Interspersed throughout this document is discussion of a number of scripts I have written throughout the project which enabled me to work more quickly and effectively. Several of the scripts involve logic which, if performed manually, would require the user to maintain many states in their mind and/or keep detailed notes. The importance of ensuring that scripts are correctly written is high; although once satisfied that a script has been correctly written it can be used to reduce the likelihood of errors since the effects of the script are easily reproducible by myself or others. All of the scripts written for this project are maintained in a Git repository [28].

Most of the scripts use the Bash shell interpreter. Bash is not available on the wireless testbed nodes so some scripts instead make use of whichever interpreter is linked to `/bin/sh`. This is not a problem in these cases as they do not make use of any shell specific features. One of the scripts contained in the repository is really a Java program. The complexity of its requirements, along with my experience with Java, made this the best choice in that case.

The scripts will be mentioned as appropriate in their corresponding chapters. An example of the need for scripts is shown below, followed by a description of one script which is used by several others.

#### 3.1.1 Example: binder-scan

The binder-scan script performs scans for wireless access points that are in range, using the `iwlist` tool. The script should be customised for the specific machine that it is running on. This is required because the output of `iwlist` may vary between nodes. Customisation involves altering three variables in the script:

```
NODE="11"  
CONTEXT="B"  
INTERFACES="wlan0 wlan1 wlan2"
```

The `NODE` variable contains the identifier of the machine that is running the script. This is used to filter out any access points provided by the interfaces of the node other than the one on which the scan is being performed. This is possible since all access point SSIDs are of the form `Binder-$NODE-$INTERFACE_NUM`. The `CONTEXT` variable defines a `grep` argument that should be used

to provide the lines before (B), after (A) or around (C) a line that matches the grep expression, allowing the signal strength to be retrieved. This must be altered as the ordering of iwlist output varies. The final variable, INTERFACES, is a whitespace separated list of interface identifiers. These represent the interfaces that should perform a scan.

Rather than customising the script, the values for these variables could have been passed in as arguments. For the sake of simplicity and given that the same arguments would be used each time it was run on a node, they remained as variables.

Once customisation has been performed, a scan for Binder networks may be performed by issuing the following command (output is redirected into the node-11-scan file):

```
./binder-scan > node-11-scan
```

Without the script, the following commands would be run in order to generate the same output:

```
echo wlan0 > node-11-scan
iwlist wlan0 scan | grep -B 3 -e Binder\[^[11] |
                    grep -v Mode | grep -v Encryption >> node-11-scan
echo >> node-11-scan
```

```
echo wlan1 >> node-11-scan
iwlist wlan1 scan | grep -B 3 -e Binder\[^[11] |
                    grep -v Mode | grep -v Encryption >> node-11-scan
echo >> node-11-scan
```

```
echo wlan2 >> node-11-scan
iwlist wlan2 scan | grep -B 3 -e Binder\[^[11] |
                    grep -v Mode | grep -v Encryption >> node-11-scan
```

## 3.2 sys-vars

Several of the scripts set many options through the Linux sysctl tool. These options are used to configure kernel options from userspace. The sysctl options are maintained in a hierarchy and in some cases a script may need to set several identical options which are located under different parents. An example of this is the accept\_source\_route option which is used to allow traffic with the LSRR IP Option set. This option exists under several parents, including but not limited to, each network interface present on the host. Forgetting to set this option under a single parent can mean extra time wasted debugging!

The sysctl tool does not allow for this usecase of setting options under multiple parents. However, sysctl does not store the options itself. All options available to set are maintained as flat files in a filesystem hierarchy in /proc/sys. Echoing values into these files sets them. This script implements a function which may be used by other scripts. The setSysVars function loops through the each parent directory and sets the desired option in each.

Several options must be passed to the function. The path under which all of the parent directories

reside must be provided. This path is given in sysctl dot notation and is converted to / filesystem notation automatically. The variable name and desired value are then given.

## Chapter 4

# Implementation of LSRR in MPTCP

### 4.1 Introduction

Part of the Binder system includes a patch on top of the Linux implementation of MPTCP. This patch was previously last updated in October 2012 and there has since been a couple of stable releases of MPTCP. My work on the Binder project involved updating the patch to work with the latest MPTCP release and also to investigate the implementation of IPv6 support within the patch. The final objective would be to work with guidance from the upstream MPTCP developers in order to have the patch carried upstream in a future MPTCP release. The modifications to MPTCP are maintained in a Git repository [20].

### 4.2 Development Testbed

To test the changes made while altering the MPTCP code I used a testbed which consisted of QEMU [47] virtual machines running Debian. QEMU has an argument that allows a kernel image to be booted in a virtual machine; after making changes to the kernel I was able to boot it in each machine and perform tests. To ensure that the testing environment stays the same between reboots the mptcp-setup script is run upon boot of each virtual machine to set the required options.

Each virtual machine is connected to my host machine via two bridge interfaces and associated tap interfaces, two per machine, one each for IPv4 and IPv6. The routing tables on the host machine redirect traffic to the appropriate tap interface. The bridge interfaces are assigned a gateway address and then each virtual machine configures an address for the subnet of the bridge its associated tap interface is connected to. Figures 4.1 and 4.2 show the network topology for IPv4 and IPv6.

This testbed allows the two virtual machines to communicate only through an intermediate node (my host machine) rather than directly. Packet capture points can be configured on the host machine at each tap and bridge interface. Live packet capture is performed on the host using Wireshark [56] which allows all the packets to be inspected between the two virtual machines. Opening an SSH session from one virtual machine to the other allows me to check the establishment of TCP connections between the two machines and check that the routing options specified are correctly set.

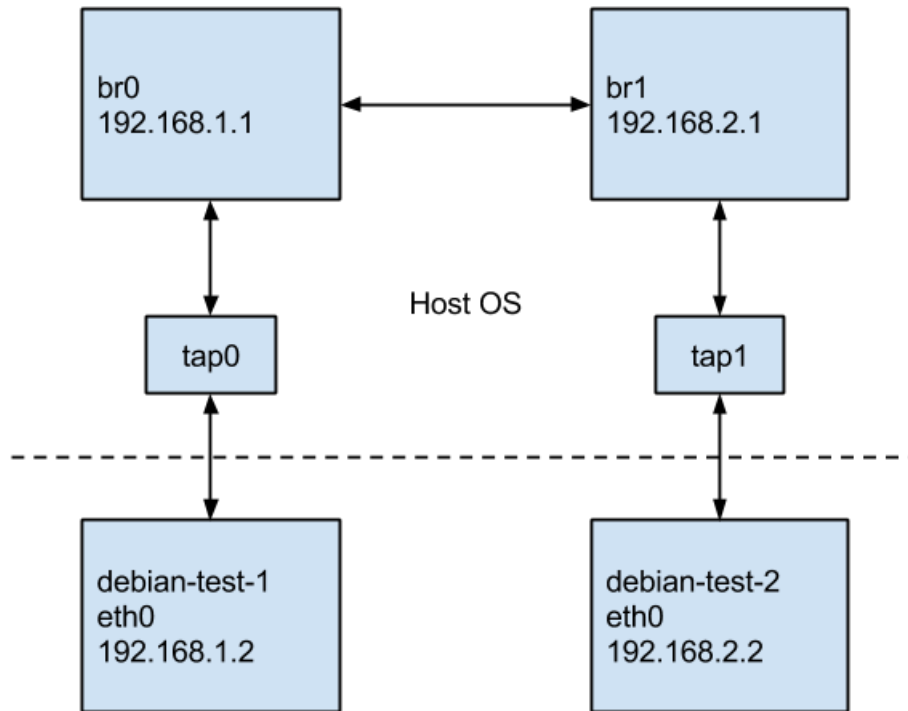


Figure 4.1: IPv4 QEMU Testbed Topology

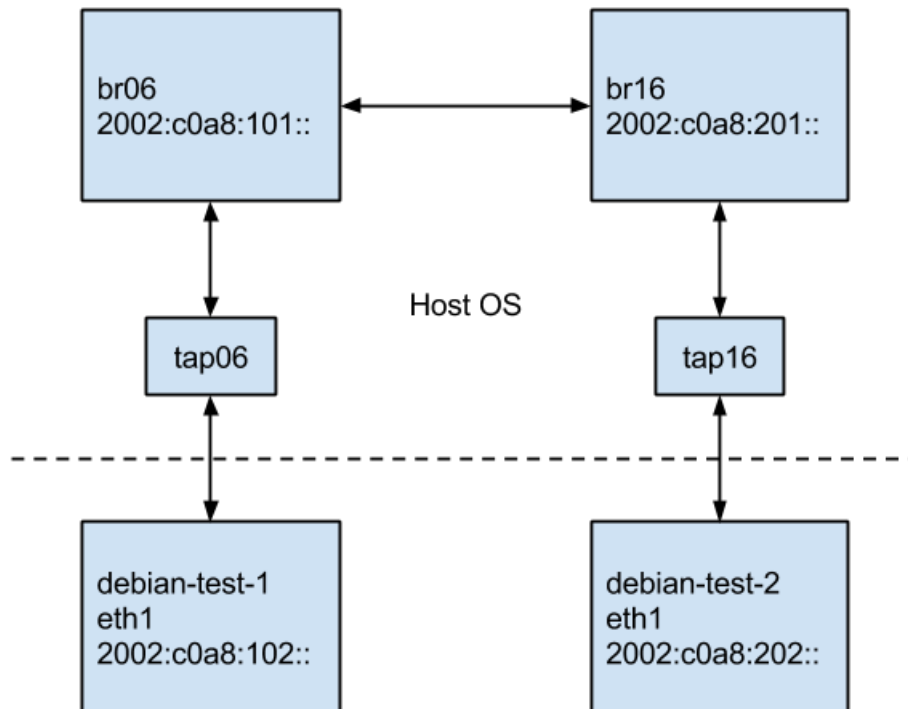


Figure 4.2: IPv6 QEMU Testbed Topology

## 4.3 Scripts

### 4.3.1 host-setup

This script sets a number of options on the host machine which are necessary for the testbed to function correctly. The source routing and forward options must be set to ensure that the traffic with LSRR options set is not dropped between the source and destination. Several NAT mappings are set up to ensure that traffic between the virtual machines must travel through the host machine's bridge interfaces. It also ensures that the traffic can be easily captured and inspected. The script also creates an additional NAT mapping for the virtual machine subnets on my WAN interface. This provides Internet access to the virtual machines.

The script may be provided with the `disable` argument which will reverse all effects. If provided with no argument, or optionally with the `enable` argument, then the script will implement the desired effects.

### 4.3.2 mptcp-setup

This script is run on both the client and server virtual machines in the testbed. It ensures that the required MPTCP options are enabled for the effects of the Binder MPTCP patch/module to be shown. The required source routing and forwarding options are also set by the script. Two arguments may be provided to the script which affect the number and length of the source route lists that are passed to MPTCP. This was used during testing of the parsing of these lists by the Binder patch to ensure that different types of lists were accepted or rejected as appropriate and also that they were parsed correctly. A mistake made when writing this script helped uncover a bug in the Binder MPTCP patch where parsing the gateway user input would fail under some circumstances.

### 4.3.3 start-virtual-instances

The QEMU command to start both the client and server virtual machines is quite long and complex. This script allows both machines to be quickly started with differing options as are required. As well as starting the machines it also ensures that the required virtual bridge interfaces are up which enable the networking aspect of the testbed. Each machine is configured with two interfaces, one for IPv4 and one for IPv6 communications. Although both addresses could be assigned to the same interface I find it easier to configure and maintain when there are separate interfaces.

By default each machine boots the kernel image which is located in a directory known to the script through a variable. Alternatively a path to a kernel image may be provided as an argument to the script, both machines will then boot this image. This can be helpful when testing multiple images since the machines can be quickly and easily provisioned with different images. Another argument, `-d`, may be provided which will enable a GDB (GNU Debugger) server on the second (server) virtual machine. A GDB client may connect to the GDB server which enables kernel level debugging. This was used to some extent during the Mobility IPv6 Routing Header 2 debugging. Use of the debugger on network code was limited as it is tricky to ensure that you were locked onto the correct packet/thread.

## **4.4 Binder MPTCP Patch**

### **4.4.1 Description as of September 2013**

The patch allows Loose Source Routing IP Options to be inserted into each of the sub-flows that are generated by MPTCP for any master TCP flow. The patch adds sysctl options which allow the user to define a list of source routes. Each list acts as the source routing path for an individual sub-flow which allows sub-flows to be routed separately and thereby provides the characteristics of Binder. As the main flow socket has already been connected it is not possible to alter the IP Options for it from MPTCP code, instead this would require modifications of the applications requesting the TCP connection.

After entering the desired list of routes, the patch parses the lists and stores the list of lists in a 2D array along with a timestamp so that changes to the list can be easily recognised. The MD5 hash of each list is also calculated and stored in a separate 2D array. By doing this, any change to a list can be detected in constant time, rather than having to iterate through each item within a list to check it against the currently stored list.

When a new TCP connection is created MPTCP allocates its own data structures. The patch adds a new data structure to this per flow allocation which contains a copy of the fingerprint array, timestamp and an array indicating the usage status of each address list. This information can be updated each time a new sub-flow is opened, by first checking the two timestamps to determine whether an update to the per flow data is required.

The patch also adds a small data structure to a per sub-flow MPTCP data structure. This stores the fingerprint of a single source route list and a flag which indicates whether that list has been added to the sub-flow socket as an IP Option. By storing this information we can work out which sub-flows are using which address lists and can update the per flow usage status of each list accordingly.

The patch performs a new function call after a sub-flow socket has been bound, but before it is connected. Here, the next free source route list is retrieved by looking up the hash from the per flow list and then retrieving it from the list of route lists. The IP Options are set and the list hash is copied into the per sub-flow data structure. When a sub-flow finishes and its socket is deleted, the patch retrieves the fingerprint from the sub-flow data structure and sets that list fingerprint as available once again in the per flow list of fingerprints.

### **4.4.2 Rebasing**

My work on the patch involved implementing it on top of the latest stable release of MPTCP, first 0.87 and then later 0.88, in a process known as rebasing. I also performed further testing on the sysctl option which allows the lists of source routes to be set. Updating the patch was mostly straightforward and I performed a manual merge of the patch diff with a new branch. The most major change was a differing method header with a changed return type, requiring a change in how failure is handled while allocating our new data structures.

There were several issues involving the parsing of address lists which would cause a silent failure but which would then mean that the expected IP Options were not correctly set on the sub-flows. Di-



agnosing and fixing these issues were somewhat non-trivial since the parsing code is not particularly easy to jump straight into.

#### 4.4.3 IPv6

The patch originally only supported IPv4 connections and it is desirable for Binder to also support IPv6 connections. When deployed on Tegola, Binder will be contained within a private network that can more than sufficiently run on IPv4 addressing. To enable more widespread use of Binder, however, others may wish to use it on IPv6 networks. An initial framework for providing IPv6 support was present in the patch. This allowed a list of IPv6 source routes to be added to the system although they would not be used. I updated this to ensure that setting both IPv4 and IPv6 source routes on a machine would not result in adverse effects. A bug existed in the patch where setting the the list for one IP version would overwrite the list of the other. After establishing that the routes weren't being made available to the sockets when they opened, I discovered that memory holding both source route lists was being freed when either of the lists were set. Adding separate structures to hold the route lists for each IP version solved the issue.

After testing to ensure the infrastructure to support inserting the routes into sub-flows was working, it was time to investigate the implementation of routing options for IPv6. In the original IPv6 specification a type 0 routing header was defined which acts similarly to the Loose Source Record Routing IP option for IPv4. Multiple addresses could be specified and the packet would be routed via each address in turn. This routing header was later deprecated from the IPv6 standard and any host receiving a packet including the header will silently drop it [45]. RFC 5095 details the decision to deprecate the header from the standard which was made due to the potential for it to be used for denial of service attacks. By allowing a packet to circulate between two routers, the header allows for an amplification attack where the path between the routers is overwhelmed with these circulating packets. However, a type 2 routing header does exist in Mobile IPv6. This was designed to route traffic for a mobile host when it is away from its home network [38]. This header allows for a single address to be specified and it is ensured that the packet will be delivered via that address.

To implement adding the type 2 routing header to the sub-flow, I examined the setsockopt API. The user-space setsockopt API allows setting the type 2 routing header for all packets on a socket by creating an IPV6\_RTHDR structure containing the address. The kernel-space setsockopt API calls do not accept structure parameters and instead require the header details to be passed as a char array. The RFC for Mobile IPv6 details how a type 2 routing header is constructed, including the correct paddings for fields. To add the header to the socket in kernel-space I constructed a char array containing the required fields at the correct bit positions. Passing this to the setsockopt function call correctly sets the type 2 routing header on sub-flows opened in an MPTCP connection.

Unfortunately this implementation resulted in the SYN TCP packet being dropped at the host bridge interface. This was caused by inserting the gateway into the extension header, rather than setting it as the destination and inserting the final destination into the header. By swapping the addresses the packet would now correctly reach the destination although it would be silently dropped there.

After performing debugging using GDB and inserting printk statements into the kernel code that handles IPv6 extension headers, the `ipv6_rthdr_rcv` function within `/net/ipv6/exthdrs.c`, I discovered that the `xfrm6_input_addr` function is called while processing a type 2 routing header. This function

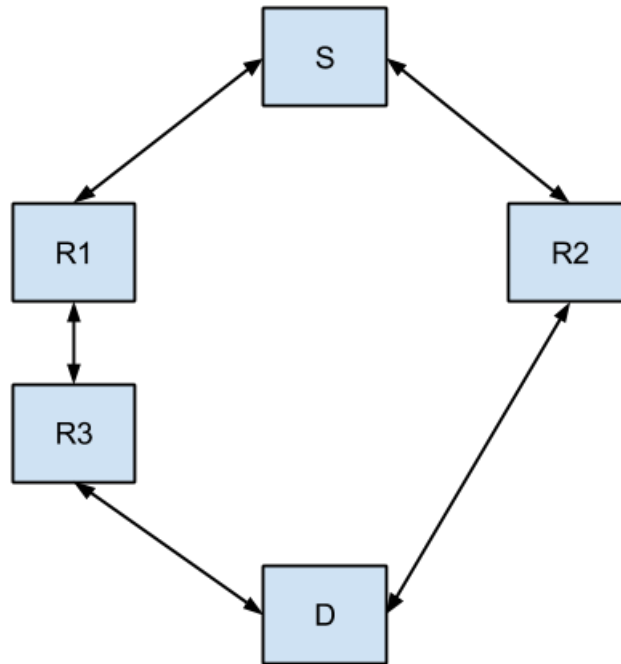


Figure 4.3: Example routing topology

performs a lookup for an XFRM state; XFRM performs transformations on packets. The lookup for an XFRM state fails because there is no state matching the routing header received, it hasn't been configured, and the packet is dropped. Indeed, after further research I discovered that a packet with a type 2 routing header will not be accepted unless there is a corresponding XFRM state and XFRM policy [40]. Since this problem occurs at the destination of the packet it is not possible for us to configure the machine correctly to accept the header, the destination may be any server on the Internet.

It is possible that a version of IP Option stripping could be implemented for the type 2 routing header. This solution could work since we can control any machine within the Binder network; the header would then be removed before leaving the network such that any external machine would not have knowledge of the header. Even if this was to be implemented there is another issue with the type 2 routing header for the source routing required by Binder. An example routing topology is shown in figure 4.3; suppose that a packet from machine S is sent to machine D and contains a type 2 routing header indicating that it should be source routed via R1. Since the type 2 extension header can only contain one address to route via, and the header cannot be chained like the type 0 header could be, a packet can only be source routed one hop. Therefore, when the packet arrives at R1 it will then be routed using the routing table there, populated by whatever routing protocol is in use. The routing table may define the lowest cost path to D to be via S. In this case the packet will then be routed back to S to then be routed via R2 to D. This not only negates the effects of Binder but actually increases transmission time since there are two worthless hops from S to R1 and from R1 to S.

## Routing Protocol for Low-Power and Lossy Networks

There is a proposed standard, IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [55]. In conjunction, there is another proposed standard to add a source routing header to RPL [27]. In RFC 6554 for this routing header it explicitly mentions that it draws from the deprecated type 0 routing header while adding constraints to prevent the issues which lead to the deprecation of the type 0 header. The RPL source routing header enforces that the source route path may not have any gaps, it must be specified hop to hop. It also only allows source routing between RPL enabled routers on the same domain, hopefully preventing attacks from external networks. Finally, source routes may not specify multiple addresses which are assigned to the same router (ie. different interfaces) to prevent routing loops from occurring.

With the requirement to provide Binder in a timely fashion, it is not possible to implement an RPL IPv6 solution in Binder for a couple of reasons. The first being that it is still a proposed standard and has been since early 2012. The specification may change in the future, perhaps removing features required for its potential use in Binder. The second issue is that there are only a few implementations of RPL, for Linux at least [14] [4]. Without mainline support for RPL and its source routing header this solution is not currently practical. This is certainly something to keep in mind for future work on Binder, however.

## 4.5 Upstreaming

A significant goal for my project is to contribute the changes made to the MPTCP codebase to support Binder back to the upstream project. This will widen the availability of the Binder system and hopefully help increase its quality. We approached Christoph Paasch, maintainer of the Linux MPTCP implementation, in order to begin the process of upstreaming the Binder MPTCP patch.

### 4.5.1 Path Manager

Before submitting the patch, Christoph informed us that our solution must be implemented as an MPTCP path manager. This is opposed to patching the core MPTCP code as we had been doing up until this point. This threw up a few contentions and has resulted in switching the implementation of Binder in MPTCP somewhat.

The path manager framework was introduced in the MPTCP v0.88, the latest stable release. A path manager is a kernel module that may be loaded and chosen as the desired one for use. The purpose of a path manager is to establish the MPTCP sub-flows according to some designated criteria. There are two path managers included with the standard MPTCP distribution:

**Full-Mesh** creates a sub-flow for each available address (ie. interface) on the machine, thus allowing the aggregation of multiple interfaces' bandwidth.

**ndiffports** allows the user to specify the number of flows to create. The flows are then initiated between the same address pair but using a different port, this is therefore used on machines where there is only one interface available.

The first issue with implementing Binder as a path manager is that to configure a Binder system

we make use of `ndiffports` as the path manager. The number of sub-flows is usually set to the number of Binder source routes that have been set plus one (to account for the non source routed main flow), meaning that a number of sub-flows are created with each being separately source routed which provides the effects of Binder. Conceptually Binder and a path manager are different; a path manager is concerned with how and why MPTCP sub-flows are created while Binder is only concerned that there are some sub-flows in existence and that they are available to be source routed.

Another issue stems from this which is that there is no method for writing a module that extends upon an existing path manager. This means that implementing Binder support as a path manager requires duplicating the `ndiffports` path manager and adding in the appropriate LSRR support for Binder. The result of this is a large duplication of code. For the moment Christoph is happy with this however, since the code duplication is contained within a module and the core MPTCP code does not require significant modifications.

The current path manager framework provides callbacks that are invoked, for example when a master socket is opened, which allows the desired path manager to then perform the opening of sub-flows. To implement Binder as a path manager module, callbacks are required at certain stages that do not currently exist in the framework. These are:

**Sub-socket bind** to allow source routing options to be set

**Socket delete** to enable resetting availability of source routes and/or memory cleanup

In addition to the new callbacks a generic memory allocation is required in the MPTCP sub-socket data structures to allow the storing of the source route in use for that socket.

#### 4.5.2 Re-Implementation

Both Luca and myself worked independently on re-implementing the Binder support as a path manager module by creating a fork of the `ndiffports` path manager. This was a team decision as we felt that parallel development would allow for the best quality solution. High quality code is especially important given that it will be included upstream. Luca implemented the Binder path manager module as an in-tree module, meaning that the module is contained within the MPTCP (Linux) source tree and is built along with the kernel which also allows it to be embedded into the kernel. Meanwhile, I approached the problem by creating an out-of-tree module meaning that it can be distributed and built separately from a kernel source tree. My decision to approach the problem this way was partially as a proof of concept to show that Binder could be implemented with minimal in-tree modifications (only the callbacks and memory allocation) and also due to a worry that the path manager module would not be carried upstream with MPTCP. If this was the case, an in-tree module creates maintenance work for us since we would still be required to patch a kernel to add in the module and build support.

My implementation of Binder as an out-of-tree module worked as expected and after comparing with Luca's solution revealed that the differences were minimal between our implementations. I had excluded IPv6 support entirely from my module, for the reasons mentioned earlier, while Luca had altered my type 2 routing header implementation to support the type 0 header instead. Due to its deprecation, we have not been able to test this solution (as expected, Linux no longer contains support for the header) and it is likewise extremely unlikely to be accepted upstream. The support for type 0

may remain as a very experimental feature that some may wish to use for research.

The final implementation consisted of Luca's module with some cleanup and minor changes from my module, along with a generic memory allocation (inspired by another already in use by MPTCP path managers) in the sub-socket data structures and modified to use the callback parameters used by my module. Luca is happy that they will accept the module for inclusion upstream which is why we chose his in-tree solution.

The patch was then generated using the 'git format-patch' command which generates a diff of our changes against the upstream source, in a suitable format for submission by e-mail. Christoph responded to our patch submission requesting that some changes were made. In our patch we had increased the memory overhead for each master socket by 320 bytes (to store the fingerprint lists) and the overhead per sub-socket by 24 bytes. Christoph was unhappy about this as it increases memory load on the system, regardless of whether the Binder path manager is in use. Another point he raised was to ensure that patches pass the scripts/checkpatch.pl script before they are submitted. This script is located in the kernel source tree and checks the formatted patches for bad practice and ensures that the additions adhere to the code style of the kernel. Christoph also raised concerns with the amount of memory operations used to implement the Binder solution, such operations can be slow and can cause issues if performed during a network routine. He suggested that we look at implementing the Binder modifications without increasing the memory overhead for the core MPTCP code. Rather than store the source route list fingerprints for each connection and sub-socket (which is done to find a free source route list when a new sub-socket is opened), Christoph suggested that we could iterate over each socket in the connection in order to discover the lists already in use by the connection.

As a proof of concept, to try and avoid the allocations in the socket data structures, I allocated the required structures in the module and then only stored a pointer to these structures in the socket data structures. This allows for no increase in overhead for the master socket allocation and only an 8 byte increase for the sub-socket allocation (since there is no existing generic allocation for each sub-socket). The issue with this solution is that although the memory overhead is reduced when the Binder module is not in use, it also greatly increases the number of expensive memory allocations when it is in use.

The current implementation of Binder support in MPTCP has been completed according to the recommendations made by Christoph and at the time of writing is being prepared for re-submission to MPTCP. The Binder path manager module no longer requires any memory allocations in the MPTCP core and the number of new callbacks has been reduced to two through the removal of the socket delete callback. Our path manager no longer needs to be notified when a sub-socket is closed since the source lists in use by a connection are defined by those set in the open sockets of that connection. When a socket is closed it no longer exists as part of that connection, thus its list is set as usable implicitly. When the socket bind callback is called, the Binder path manager iterates over each source route that has been set by the user keeping a count of the number of sub-sockets and how many sub-sockets have marked the list as not used by them. For each list, each sub-socket in the current connection is also iterated. If the current socket does not have IP Options set then the list is marked as unused by that socket. If there are LSRR IP Options and the current list and the list set in the sub-socket are different lengths then the list is also marked as unused. Otherwise, each address in the current list and the list set in the current sub-socket are compared, if a different address is found then the lists are different. If any of these conditions do not hold then the lists are identical and the current list being

iterated is not marked as being free for that sub-socket. Once each sub-socket has been checked, the current list is free if the number of sub-sockets matches the number of sub-sockets that marked the list as unused. Iterating over the lists continues until a free list is found or returns an error status if no list could be found. A new lock mechanism (spinlock) is also introduced for each flow, this prevents two sub-sockets which open at the same time from both taking the same list.

## Chapter 5

# D-ITG Applications

### 5.1 Introduction

After deciding to use D-ITG for the network measurement portion of my project I investigated how the tool can be controlled. The ITGSend process which generates and sends the traffic can be used in two modes, regular and daemon. In regular mode the traffic generation arguments are provided as arguments to the ITGSend process at runtime. Conversely, in daemon mode the ITGSend process starts but no traffic arguments are provided. It waits to receive these arguments which are sent within UDP packets via an API.

The D-ITG system includes a C++ API which may be used within an application to send commands to an ITGSend process. Included with the API is a sample application which allows the user to send a single traffic generation command to a single ITGSend process. After studying the features of the API, I decided that I would be able to write a program to control multiple ITGSend processes, easily and from a single 'control' host.

In the following sections I discuss a new library and application that I have written: JITGApi and ITGController. The desire to start these projects arose from my experiments in D-ITG and its API. I have notified members of the ditg-users mailing list of these tools so that they can gain exposure and hopefully use by others.

### 5.2 JITGApi

JITGApi (Java ITGApi) is a complete implementation of all operations contained within the official C++ API, written natively in Java. The implementation of this library started during the development of the ITGController application which will be described in a later section. After reaching a working state, the code was split from ITGController and it is now available in a Git repository [30] under the GPL v3 license.

There is no existing implementation of the D-ITG API in Java. However, there is a freeware GUI written in Java [49] although this does not allow configuration of remote ITGSend processes. Instead, it is a wrapper around D-ITG processes which run on the same machine running the GUI. For my testing this would be a hindrance anyway, since the test machines are both headless and do not have a graphical environment configured.

### 5.2.1 Rationale

#### D-ITG C++ API Sample Application

During my first experiment with the D-ITG API, I studied and modified the sample application provided with the API in order to accept a comma separated list of traffic generation commands, rather than a single command. Although this was successful, other problems were apparent with this solution. It was still not possible to configure multiple ITGSend processes, perhaps residing on separate hosts. Also, due to the way the sample application was written it looped, waiting for a response on the sending socket. This meant that in order to receive responses for multiple commands it would have required much greater changes. Due to the magnitude of changes required I decided to write a tool in Java due to my familiarity with the language, I have no specific experience with C++.

#### Java Native Access

The next solution was to use the Java Native Access library (JNA) [3] in conjunction with the C++ API. JNA allows Java programs to load shared objects written in native languages such as C and C++ and call functions defined within. This solution appeared to work reasonably well once it had been correctly configured. There is good documentation available in the JNA Git repository which aided with this. I had most trouble with ensuring that JNA was aware of the path of the libITG.so file so that it could be correctly loaded.

Although this solution worked quite well and allowed me to write the initial version of ITGController it also presented a couple of issues. The first of these relates to the build process required for ITGController. Since libITG.so is a native library it must be compiled for the correct operating system and architecture prior to building the Java application. This is something that users of Java programs may not be accustomed to. Compilation by the user is also required in most cases since D-ITG is only officially distributed as source code for Linux. Packages are available in Debian but they do not include the API library. Another issue is that different compilers or even compiler versions may result in the API functions being exported differently in the shared object. The exported function identifiers for my compiled version of the library were discovered using the nm tool and are hard-coded into the Java application. As a result, the Java application may not work unless modified if a compiler exports the functions differently. Finally, the API operation to receive a response from an ITGSend process is non-blocking and response UDP packets are not buffered; instead the operation is invoked from the Java application continuously with a specified delay. If there is a response packet waiting it is returned. Although this solution works sufficiently when only sending one or two commands; if any more are sent then their responses are typically lost. This is not helped through the use of the best effort UDP as the transport protocol for both sending commands and receiving responses.

Through my experience with JNA it is a very thorough solution if your only option is to make use of a native library in Java. Since the API deals with UDP packets it is possible to re-implement it purely in Java. This negates all of the disadvantages previously mentioned.

### 5.2.2 Implementation

The pure Java implementation of JITGApi means that integrating into other Java applications is easy. Compilation is also simple as it does not require the C++ API. The library is built using an Ant [1]



0	1	2	3	4	5	6	7	8	9	10	.....	300
Type				Message Length				Message				

Figure 5.1: Payload structure of an ITGSend Response Packet

build script which generates a jar file containing the API classes. The API can then be imported into other Java applications as any other Java library would be.

### Packet structure

The structure of both the command and response packets were discovered by studying the C++ API implementation and also by intercepting and studying the associated API packets in Wireshark. The payload structures for both packet types are described below.

**Command packet** The payload of this packet consists only of the ASCII encoded traffic generation command that should be run by the ITGSend process running at the packet destination.

**Response packet** The payload of this packet is shown in figure 5.1. The C++ API defines a total limit of 300 bytes for the length of the payload on a response packet and it consists of three fields:

**Type** This field is 4 bytes in length, starting at offset zero, and contains an integer which represents the type of response message being carried. This may have the value of either 1 or 2 for a traffic generation start or end respectively.

**Length** This field is also 4 bytes in length, starting at offset 4, and contains the length of the actual message carried within the payload.

**Message** Starting from offset 8 this field contains the response message from the ITGSend process at the source. The message will be the command that was originally sent to the ITGSend process. Along with the value contained in the type field, this indicates when a particular traffic generation command starts or finishes.

### Class Descriptions

JITGApi consists of just two classes. The class diagram is shown in figure 5.2.

**ITGMessage** This class represents a response message that may be received from an ITGSend process. It contains the offset values and field lengths as described above and uses these to parse the payload of response packets. To abstract away these details for applications using JITGApi, the Type enum is defined which indicates what type of response an ITGMessage object represents. Once the payload is parsed, the type and message are available through method calls, as well as the InetAddress object representing the sender. An equals(...) method is provided which allows ITGMessage objects to be compared, the sender, type and message are checked for equality. The toString() method allows easy console logging of ITGMessage objects.

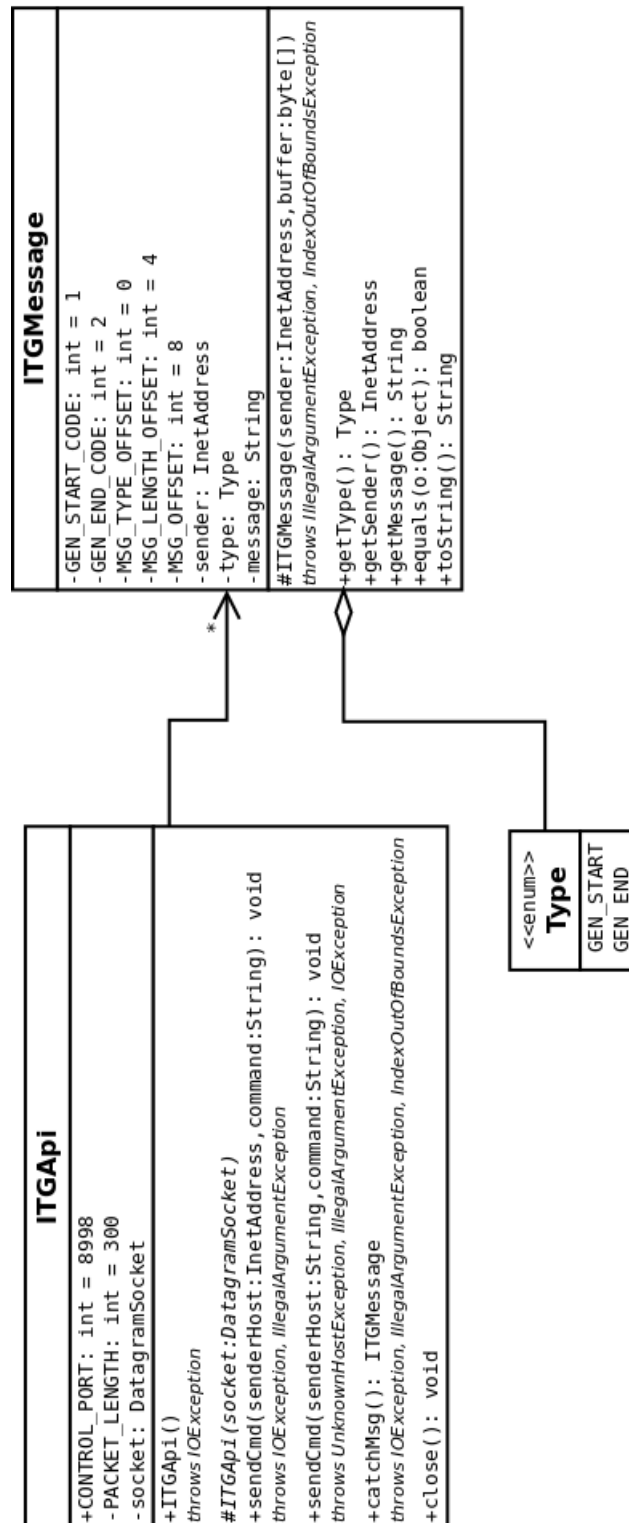


Figure 5.2: JITGApi Class Diagram

**ITGApi** This is where the main operations of the API are implemented. An application wishing to use the API must construct an instance of this class, no parameters are required. A protected constructor is also defined which allows an existing DatagramSocket (Java UDP socket) to be passed and used by the API operations. This is used for unit testing the API.

**sendCmd** The sendCmd API operation is provided by two overloaded methods which require passing the address of the host running ITGSend (either as a String or InetAddress object) and also the command to run. The operation then constructs a new packet and sends it through the DatagramSocket object.

**catchMsg** The catchMsg operation is a blocking call which waits until a response packet is received at the socket. Packets to sockets represented by DatagramSocket are buffered so unless the UDP packet is lost during transmission there will be no lost response messages, as I experienced with C++ API. When a packet is received, a new ITGMessage object is constructed with the packet payload and then returned to the caller.

### 5.2.3 Documentation

Full JavaDoc documentation of the API is provided and generation of the HTML documentation pages is implemented in the Ant build file, using 'ant doc'. For stable releases of the library, hosted documentation is also maintained [31].

### 5.2.4 Testing

The JITGApi source tree also includes a set of JUnit tests which are designed to provide maximum code coverage of the API. The tests are also hooked into the Ant build file, allowing them to be run with the 'ant test' command. Descriptions of the test classes are provided below.

**MockDatagramSocket** This is used to test the ITGApi class. MockDatagramSocket is a test harness that extends DatagramSocket and overrides the send, receive and close method calls. This allows the tests to inspect the packets that the API sends to or receives from the socket.

**send** This method updates a stored reference to the latest sent packet. Tests then retrieve this latest packet by calling the getLastSentPacket() method. The packet can then be checked to ensure that it has been properly formed according to the parameters passed to the sendCmd API operation call.

**receive** Before a packet is received by the socket the tests call setPacketForRetrieval(DatagramPacket), passing in a packet with known values. The tests can then call the catchMsg API operation which will return an ITGMessage object constructed from the payload contained in the packet passed to the mock socket. The values contained within the ITGMessage object should then be compared to the values in the packet to ensure that they were correctly received and parsed.

**close** This method simulates the close of the socket and is called by the ITGApi.close() method. Once this method has been called any subsequent calls to send(...) or receive(...) will result in an IOException. This can be used to test the effect of ITGApi.close() and also to ensure that the API operations respond appropriately to a closed socket.

**ITGTestUtils** This class contains a number of constants and static methods that will be utilised by several tests. The constants define values such as the offsets of fields in the response packet payload. They are duplicated here, instead of using the values already available in the API, to ensure that any mistakes in the values in the API are caught. Of course, this assumes the same mistakes were not made for these constants! A method to construct a payload (byte array) buffer from the field values is implemented, along with a method to insert a String into such a buffer at a particular offset.

**ITGMessageTest** This class contains tests that are concerned with the ITGMessage class. These test the parsing of the payload buffer, done during object construction, and all other methods in the class which are concerned with retrieving the parsed values. Parsing is thoroughly tested with different types of payload buffers, valid and invalid, being passed to the constructor. The function of the equals(...) method is also thoroughly tested with multiple different types of object.

**ITGApiTest** This contains tests that check the API operations from call to response. Since the catchMsg operation returns an ITGMessage object there is some coverage of that class in these tests also. The API operations are called with different parameters and the packets or responses generated by the API are then checked to ensure they correspond to the arguments passed. As such, this is an end to end test of the operation of the API.

## 5.3 ITGController

As mentioned above, ITGController is a Java application that originally spawned, and now makes use of, the JITGApi library. ITGController can be used to control multiple instances of ITGSend running on disparate hosts. The commands for each host are defined in a single config file which allows different tests for an entire network of senders and receivers to be easily defined. The config file contains commands exactly as they would be provided to the ITGSend process or the sample C++ application. In order to support multiple hosts the commands must be contained within special Host blocks which denote which ITGSend processes should receive what commands. Console output is provided showing sender responses and confirmation is given when all commands have started/stopped. This is particularly helpful when sending large numbers of commands. ITGController is available in a Git repository [29] under the GNU GPL v3.

### 5.3.1 Implementation

#### Class Descriptions

The class diagram for ITGController is shown in figure 5.3.

**Config** is responsible for parsing the config file provided by the user. A mapping between a host and a set of commands which should be run on that host is maintained as the file is parsed. Parsing is robust and malformed files should be detected and errors shown as appropriate. Once parsing is complete the mapping of hosts to commands may be retrieved.

**ParserException** extends Exception and provides a descriptive error (giving the line number and showing the line in question) when a parsing error occurs. An example of when such an error would appear is in the case of a command in the config file appearing outside of a Host block.

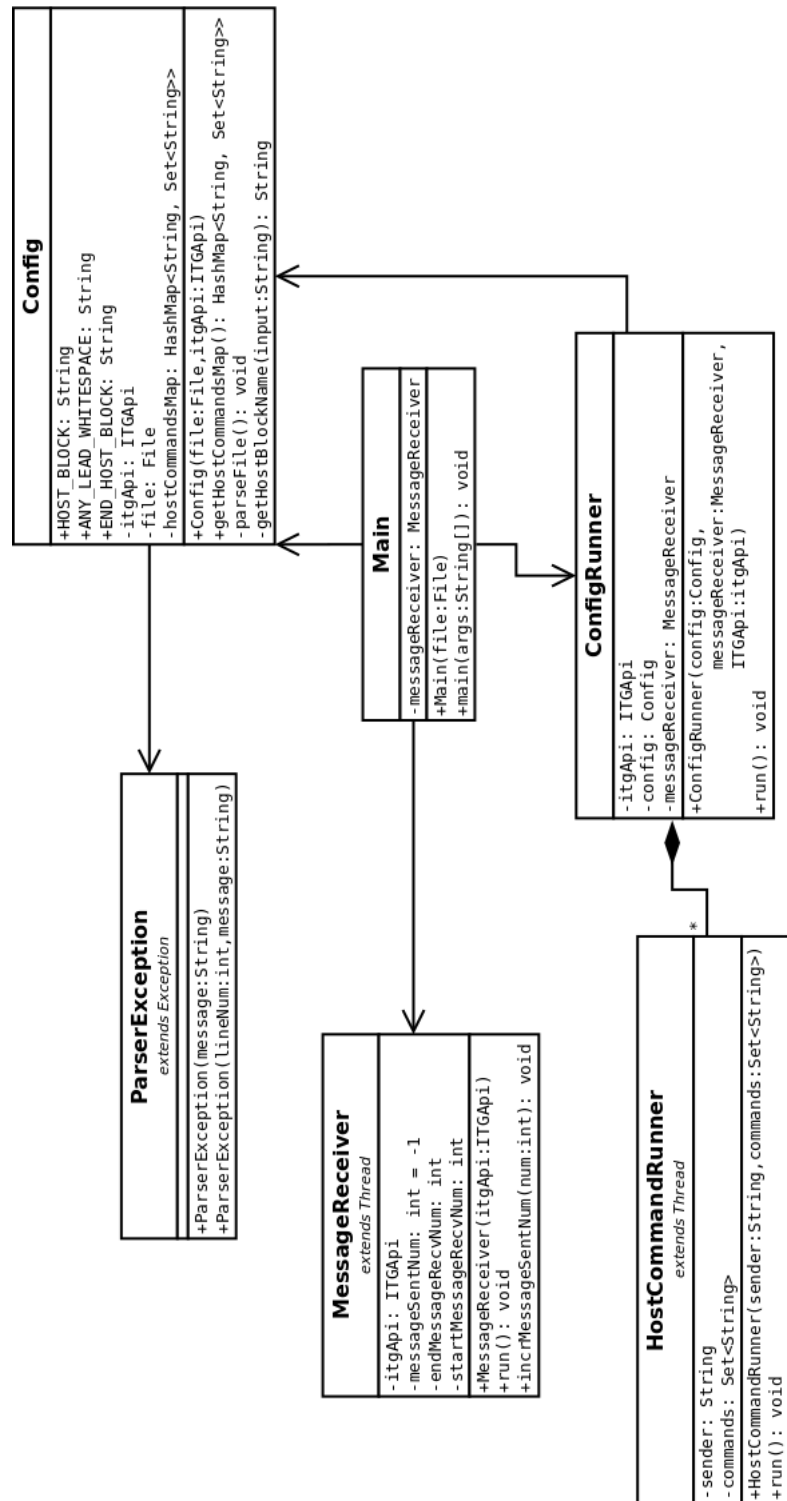
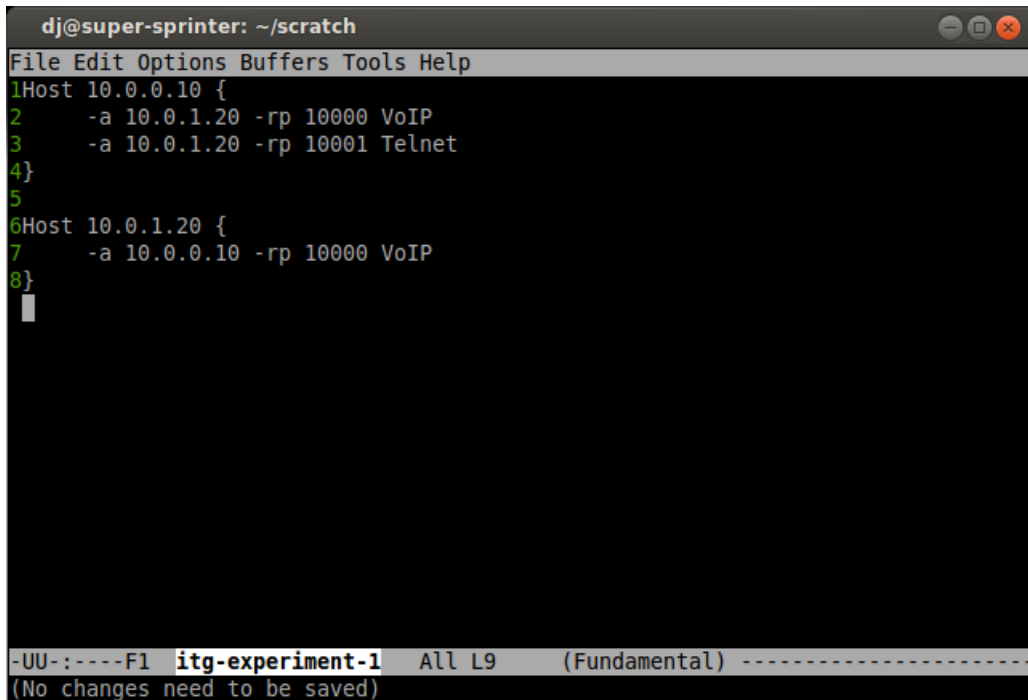


Figure 5.3: ITGController Class Diagram



```
dj@super-sprinter: ~/scratch
File Edit Options Buffers Tools Help
1Host 10.0.0.10 {
2  -a 10.0.1.20 -rp 10000 VoIP
3  -a 10.0.1.20 -rp 10001 Telnet
4}
5
6Host 10.0.1.20 {
7  -a 10.0.0.10 -rp 10000 VoIP
8}
|

-UU-:---F1 itg-experiment-1 All L9 (Fundamental) -----
(No changes need to be saved)
```

Figure 5.4: Writing an ITGController config file in Emacs

**ConfigRunner** receives the mapping of hosts to commands and iterates through each host. For every host it creates and starts a new **HostCommandRunner** thread.

**HostCommandRunner** is an embedded private class that runs commands for a particular sender. Each instance runs on a separate thread in an effort to reduce delay between multiple senders being configured, particularly when one sender may be configured for many commands. The `sendCmd` API call is used for each command and once all commands have been sent it informs the **MessageReceiver** instance of the number of commands that it sent.

**MessageReceiver** runs in a separate thread and is responsible for receiving response messages from the **ITGSend** processes. Since it is informed of the number of commands sent, a count of the traffic generation start and stop response messages can be maintained. When these counts equal the number of sent commands a confirmation can be printed out, informing the user that all commands have been started or have finished. When all finish responses have been received the program will exit.

**Main** contains the main method and handles the single argument to the program (the path to the config file). It is then responsible for starting a **MessageReceiver** instance and finally invoking the parsing of the config file and the running of all commands.

### 5.3.2 Example Config File

An example of an ITGController configuration file being written in a text editor is shown in figure 5.4. When provided to ITGController this config would result in the configuration of two **ITGSend** instances, this is denoted by the two **Host** blocks. Each sender is configured to send a single flow of

VoIP (Voice Over IP) traffic to the other machine, this config therefore assumes that there is also an instance of ITGRecv running on each machine. The sender at 10.0.0.10 will also send a flow of Telnet traffic to 10.0.1.20. Each Host block could be expanded to contain any number of flow command as are required.

## Chapter 6

# IP Option Stripping

### 6.1 Introduction

As previously mentioned, the Binder system makes use of the Loose Source Routing IP Option to distribute traffic across multiple gateways within the Binder network. The use of this option on the Internet is generally not welcomed, indeed when testing Binder you must remember to explicitly enable source routed traffic. LSRR enabled traffic is generally not accepted by routers [19].

The use of LSRR in the Binder system is sufficient since we are able to control the routers within the Binder network. The issue arises when the traffic leaves our network and is routed across the Internet to the Binder proxy and then finally to the destination. If any router on the path from the Binder network gateways to the destination do not accept LSRR enabled traffic then all WAN access provided by Binder will be lost.

For this reason, an important goal of my project was to implement a solution which would strip the IP Options from outbound traffic at each Binder network gateway. Additionally it would be most useful if there was a method of storing the stripped IP Options so that they can be re-inserted into any corresponding return traffic when it enters through a gateway.

The solution required writing a Linux kernel module which acts as a Netfilter extension and also an iptables extension which allows the module to be configured from userspace. The Writing Netfilter Extensions [33] book was invaluable in writing both the module and userspace extension.

There are actually two types of Netfilter extension, matches and targets. A match will define some properties that a packet must match. A target extension performs some kind of mangling on the packet. A rule will typically contain both a match and a target. Packets that are matched upon by the properties of the match defined will be mangled by the provided target.

A match extension that matches on all, none or some IP Options is already available from the xtables-addons project [18]. This is used in rules along with my target extension, which performs the stripping. At some point there was an extension to perform IP Option stripping, it was available via the Patch-O-Matic system [5] [12] which is no longer maintained. After an investigation, including a request to the netfilter-devel mailing list, I concluded that the source code for this extension is no longer available.



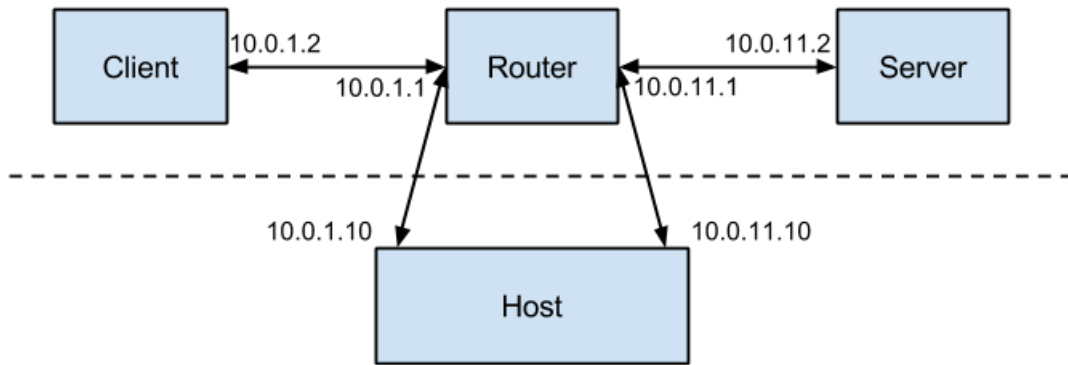


Figure 6.1: IPOPTSTRIP VirtualBox Testbed Topology

Both the module and userspace extension are available in a Git repository [32] under the GNU GPL v3.

## 6.2 Development Testbed

Since the solution involves the development of a kernel module and several machines to test its effects I needed to setup another testbed. The topology involves three virtual machines representing a client, router and server and is shown in figure 6.1. The client and server are on different subnets and each is connected to the router such that all traffic between the client and server must pass through the router. The IP Option stripping would be performed on the router with a wireshark instance running for both the ingress and egress interfaces to check the options were removed after traversing the router.

I had originally intended to create QEMU virtual machines for the testbed, as I had done for the testbed used while working on the Binder MPTCP LSRR support. While attempting to setup a QEMU based testbed I experienced issues trying to enable direct networking (socket mode) between the virtual machines. A permission error prevented me from completing this setup, despite running as root. Following this I decided to use VirtualBox [15] for this testbed since I have had good experiences with its networking options in the past. A key feature of QEMU is its ability to directly boot a kernel image, rather than installing it on a virtual disk image first. This was useful during the work on the Binder MPTCP patch since I was modifying and building new kernels which needed to be tested. Since I'm developing a module for the IP Option stripping solution it can be copied to and inserted into an already running system with no need to change kernels; rendering the kernel image boot feature of QEMU unneeded.

However, VirtualBox also presented its own issues. I had installed a point release update to VirtualBox about a week before setting the testbed up which contained a bug that caused virtual machines running Linux to kernel panic during boot. After taking a little time to track down the bug report [16] and then roll back to the previous version I was finally able to get the testbed up and running!

## 6.3 Implementation

### 6.3.1 iptables vs nftables

With the release of the 3.13 version of the Linux kernel there is now a new firewall available, nftables, which will eventually replace the incumbent Netfilter and iptables [26]. In order to reduce duplication in code between, for example, IPv4 and IPv6 support, nftables embeds a virtual machine in the kernel. Firewall rules for nftables generate byte code which can be loaded into the virtual machine, creating the desired functionality. The introduction of a new firewall is very relevant to my project since it affects how the IP Option stripping will be implemented.

For my project I decided to implement a solution for systems with iptables, there were a couple of reasons for this:

**Maturity** At the time of writing version 3.14 of the kernel has just been released meaning that nftables has only been available in the mainline kernel for two versions. iptables was introduced in Linux 2.4.x [2] and still exists alongside nftables, it will likely do so for the foreseeable future releases.

**Availability** The testbed used for performing measurements of Binder and also for developing the IP Option stripping solution runs on the current version of Debian Stable (Wheezy) which includes kernel 3.2. The current stable release of MPTCP runs on kernel 3.11 so testbed machines requiring support for MPTCP run kernel 3.11. The testbed machines where the solution will be implemented are running the 3.2 kernel since MPTCP support is not required here.

**Documentation** Since nftables has only recently been included in the mainline kernel and is still under very active development there is very little documentation regarding implementing extensions. A query to the netfilter-devel mailing list went mostly unanswered, unfortunately. The excellent book Writing Netfilter Extensions [33] was the main reference used while implementing IP Option stripping and covers the creation of Netfilter and iptables extensions.

Although an iptables dependent solution is not the most future proof, I believe it is the best solution available at this time. There is some support for using existing Netfilter extensions with nftables [26] and this may be a solution at a time when nftables must be used. Future work will include monitoring the development of nftables and migrating the IP Option stripping, once it becomes a more appropriate solution.

### 6.3.2 Additional Kernel Module - xt\_IPOPTSTRIP

#### Introduction

Specifically the module is a Netfilter target extension which, when teamed with a Netfilter match extension, is invoked for packets matching criteria defined in the iptables rules. The target extension is able to mangle any and all aspects of the packet, including the IP header; it is also possible to cause a packet to be dropped.

## IP Header

To remove IP Options several fields of the IP header [39] (summarised [51]) must be checked and/or modified. A description of these fields and how they are modified follows:

**IHL** or Internet Header Length is a 4 bit field that contains the number of 32-bit words within the header. For a packet without IP Options this is always 5, so this must be set.

**Total Length** is a 16 bit field which represents the total length (in bytes) of the IP header and payload. Since we are removing options from the header the value held in this field must be reduced by the number of bytes removed from the header.

**Header Checksum** contains a checksum calculated over all the values in the header (excluding this field, which is zeroed) [52]. This is calculated with the values present in the header at the time of transmission so it must be re-calculated as several field values are changed.

**Destination IP** contains the IP address of the packet destination. When source routing is used the value contained is either the final destination, if source routing is complete, or the current router hop that has been set in the options. The field must be modified to contain the final destination to ensure that the packet safely reaches it.

**Options** This field contains any IP Options that have been set. Since we want to remove all of the options this portion of the header should simply be removed.

## Packet Mangling

A packet matching an iptables rule with IPOPTSTRIP as the target will trigger the calling of the `ipopt-strip_tg` function in `xt_IPOPTSTRIP.c`. This function is passed a pointer to an `skb_buff` struct which is essentially a buffer containing all of a packet's data. The function performs an additional check to ensure that there are some options to process, hopefully avoiding any nasty errors that could arise from accessing unallocated memory. The module then performs a lookup to check whether the destination no-re-write flag has been set. If the flag has not been set then the destination field is updated to contain the value of the last address in the LSRR list. Next, the header and total length fields are adjusted to reflect the removal of the options. By the time the module receives a packet for processing the transport header and payload has already been added to the buffer, it is therefore necessary to move everything in the buffer up by the length of the removed options to overwrite them in the buffer. The buffer is then trimmed by the amount removed and finally the checksum is recalculated and inserted into the header. The function then returns `XT_CONTINUE` and the packet continues to be processed.

The first version of the extension used the `nexthop` member of the `ip_options` struct in order to repopulate the destination field. This worked correctly on the testbed since I only specified one hop, the router. This would have to be changed in future, however, in case the stripping was not done at the last hop before the destination. In such a case the packet would never reach the final destination since the destination would become that of the next hop from the hop doing the stripping. The final version correctly parses the LSRR IP Option, if present, and retrieves the correct destination address from the end of the list of addresses. This address is then inserted into the destination field of the IP header.

### 6.3.3 Additional iptables Extension - libxt\_IPOPTSTRIP

#### Introduction

The iptables extension allows the IPOPTSTRIP Netfilter target extension to be used within firewall rules. It is responsible for displaying usage information to the user, defining any available arguments that may be provided, parsing options and setting flags that will be passed to the kernel module.

#### Arguments

libxt\_IPOPTSTRIP defines one argument, `-keep-dest`. This argument may be used when, for some reason, the destination field in the IP header should not be re-written to contain the final destination retrieved from the source route list. By default, without this argument, the destination will be re-written.

#### Example iptables Rule

This rule will cause all packets on the FORWARD chain (packets traversing, not originating from the host) in the default filter table and containing some IP Options to invoke the IPOPTSTRIP extension.

```
iptables -A FORWARD -m ipv4options -j IPOPTSTRIP
```

## 6.4 Scripts

### 6.4.1 reload

After re-building the xt\_IPOPTSTRIP kernel module it is necessary to reload it into the running kernel, or perhaps to load it initially after boot. There are a number of pre-requisites for this to take place, thus it makes sense to delegate this to a script.

When wanting to load the module we must first ensure that there are no iptables rules, using the module, in existence. If there are any such rules it will not be possible to unload any previously loaded version of the module. Once any rules are removed, the next step is to remove any existing modules from the kernel. Following this, the script ensures that the x\_tables module dependency is loaded. It is now safe to load the xt\_IPOPTSTRIP module into the kernel so the script does so and then creates an iptables rule that allows the module to be tested.

The iptables rules created by the script ensures that all traffic in the FORWARD chain is passed to the module. The FORWARD chain contains traffic which is crossing the host, but for which the host is neither the source or destination. Any and all arguments passed to the script are also passed as module options when creating the iptables rules. This allows for different options to be tested each time the module is reloaded.

### 6.4.2 router-setup

This script is run after boot of the router machine in the testbed. Since this is the machine that is responsible for forwarding traffic from the client to the server several options must be set. Options are set for both IPv4 and IPv6. Although the module implementation currently only works with IPv4 there is no harm in enabling the appropriate options.

Since I am testing the module by creating traffic with the Loose Source Routing IP Option the script enables the `accept_source_route` flag. Without this flag, traffic with this option set will be dropped. The forwarding options are also set which allows the host to route traffic.

### **6.4.3 wireshark**

This is a simple script which opens two Wireshark instances configured for remote capture on the ingress and egress interfaces of the router. It makes use of the `remote-wireshark` script.

### **6.4.4 remote-wireshark**

This script is based on logic from the Wireshark wiki [17]. It may not be run alone but instead implements a function and it may be included by other scripts for use there. It opens a Wireshark instance configured for remote capture according to the arguments provided.

A pipe is created and a Wireshark instance is opened, given the pipe to read packets from. A `dumpcap` process is started over SSH on the remote host which captures traffic and the output from this is redirected to the pipe. A filter is provided to the `dumpcap` process to ensure that the SSH traffic is not included in the capture. This is essential as it prevents the captures being filled with irrelevant noise.

Several arguments must be provided to the function. The name of the pipe (which will be opened in `/tmp`) is provided, Wireshark displays this in the window title which makes it easier to distinguish between Wireshark instances if there are many open. The next arguments are the user and address for the SSH login to the remote host. Next, the identifier for the capture interface on the remote host is provided, only traffic passing through this interface will be captured. Finally, the address of the local machine is provided. This is used in the filter to remove SSH traffic from the capture.

## Chapter 7

# Measurements

### 7.1 Testbed

#### 7.1.1 Introduction

Previous measurements of the Binder system have been performed on a solely wired Ethernet network which provided good basis to show that the effects of Binder were real. It is imperative, however, that measurements are performed using a wireless mesh network since this is much more closely aligned with the networks that Binder will eventually be deployed upon.

The new testbed will be located at the Informatics Forum, University of Edinburgh and will consist of a number of wireless nodes, running OpenWRT [8], that make up the mesh network. After an initial test iteration the basic structure of the testbed was defined as follows. Connected over an Ethernet backbone to two nodes, designated as access points (just like eg. a home router), will be the Binder relays. A client machine will connect wirelessly to each of the access points. A dedicated beige box will also be connected to two other wireless nodes and form part of the mesh network. This box will take the role of the Binder gateway. The proxy and server machines will be connected directly to the gateway machine using the Ethernet backbone.

With such a hardware topology and correct configuration of the network this testbed will be able to function as follows. The client machine can send traffic to either access point, the access point will then redirect all traffic from the client to its associated relay. Each relay is able to access the proxy machine via the gateway and the gateway is accessible via the mesh network, through the wireless nodes. Each relay maintains an OpenVPN tunnel with which to route traffic via the proxy. Traffic from the proxy can then be routed from the proxy to the server via the gateway.

#### 7.1.2 Setup

I have had a mixed experience with attempting to implement a reliable testbed that I can use to perform experiments to explore the capabilities of Binder. The following sections describe the various iterations that I have created in search of a stable testbed. For each I will discuss the problems I encountered and the steps I took to attempt mitigation of the issues.

### **7.1.3 Hardware Upgrades**

Over the course of designing and implementing the topologies mentioned later I performed a number of hardware upgrades to the testbed machines. This allowed the client and gateway to support multiple wireless connections. This went mostly without incident, with only a few issues ensuring that the correct drivers were available for the new hardware on systems which used our custom Linux kernel.

### **7.1.4 OpenWRT Node Configuration**

There are two separate configuration files that are of interest when setting up the wireless nodes, these are `/etc/config/network` and `/etc/config/wireless`. The network configuration file is responsible for setting up interface properties such as IP address, gateway, netmask etc. Meanwhile, the wireless file is used to configure the wireless specific attributes of each wireless interface on the node, such as SSID, channel, mode, encryption etc. Extensive documentation is available for both network [10] and wireless [11] configurations.

### **7.1.5 Wireless Frequencies and Channel Allocations**

Since the networks that Binder will eventually be deployed upon use directed antennas at each node to communicate, they do not interfere with each other. In order to attempt to emulate this on a testbed network using multi-directional antennas I used isolated channels for each link.

The testbed network will be 802.11 based, which supports a number of frequencies and channels [54]. Such networks commonly use two frequency bands, either 2.4Ghz (the most widely used) or 5Ghz. Within each frequency range is a number of channels. Better performance should theoretically be attained over 5Ghz as it is less crowded and the channels do not overlap each other. On 2.4Ghz there are only 3 distinct channels, all others overlap at least slightly which can cause interference and packet loss. The 5Ghz frequencies have trouble with penetrating objects however, so they are best used when line of sight is available. This issue caused me problems when testing 5Ghz links for the testbed. Although the wireless nodes are mostly able to maintain 5Ghz connections with each other, they are ideally situated high up on the wall and use more powerful antennas. The client and gateway machines, however, are located on the floor in a lab, in close proximity to each other. These machines can struggle to find a 5Ghz connection to the required number of nodes (four) on 5Ghz.

### **7.1.6 Routing**

In order to be able to test the effects of Binder on a network in which nodes can go down and up, it is required to support dynamic routing such that each node will create and adjust its routing table according to the routes currently available on the network.

#### **OLSR**

The Optimised Link State Routing protocol [25] is specifically designed for use on wireless networks. It is a link state routing protocol meaning that it chooses the best routes according to some assessment of each possible route. I planned to use the `olsrd` implementation of the protocol [6]. It runs on mesh networks where the nodes are in ad-hoc mode, meaning that there are no nodes acting as access point or station, any node can connect to any other which is in range. The advantage of this is that the topology of the network does not need to be designed by hand, it is handled automatically as the network

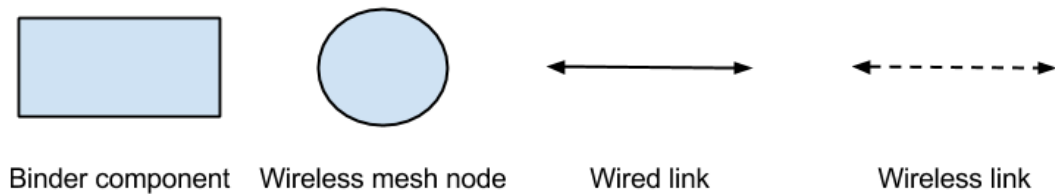


Figure 7.1: Testbed Diagram Key

environment changes.

The testbed topologies discussed later would not have been required if OLSR was in use, all nodes would simply be placed in ad-hoc mode and left to sort the topology between themselves. Unfortunately, due to my requirement to be able to specify the channel that each interface on each node uses I was unable to use OLSR. On some of the nodes, those with Atheros based chipsets, there is a bug [9] which prevented me from manually specifying the channel. The channel that I chose was ignored and an arbitrary channel chosen instead. The bug is marked as fixed so it may be possible that this could be alleviated by re-flashing a later version of OpenWRT on the affected nodes, indeed this could be done in future versions of the testbed.

## OSPF

The Open Shortest Path First protocol [7] is another link state routing protocol, it is well proven and typically used on large networks. The algorithm used for choosing routes is based loosely around the known bandwidth of a route, with higher bandwidth routes being favoured. OSPF is implemented on Linux in the Quagga routing suite [13] and I was able to successfully set this up to perform dynamic routing on the mesh network. In order to get routes to distribute correctly, each link in the mesh network must be on a different subnet. Perhaps as expected, Quagga will not distribute routes for the same subnet.

### 7.1.7 Iteration One

This initial testbed topology is shown in figure 7.2 and consists of four wireless nodes with the relays and server connecting to two each, with no intermediary nodes. The issue encountered with this topology was with node 120 which was the among the only ones available to me at the time of creating this topology. For some reason which I could not discover, the board would not route traffic although it would send traffic of its own. Ensuring that all the necessary forwarding options were set and flushing all iptables rules made no effect on the board's reluctance to route. Due to this issue and the lack of access to a fourth node (which the second relay would connect to) to complete it, this topology was abandoned.

Another issue with this topology is that the client machine connects directly to the relays over Ethernet, rather than connecting wirelessly to an access point. This is typically how it would be setup in a real world Binder network. This became the desired setup and would result in the relay being invisible to the client.



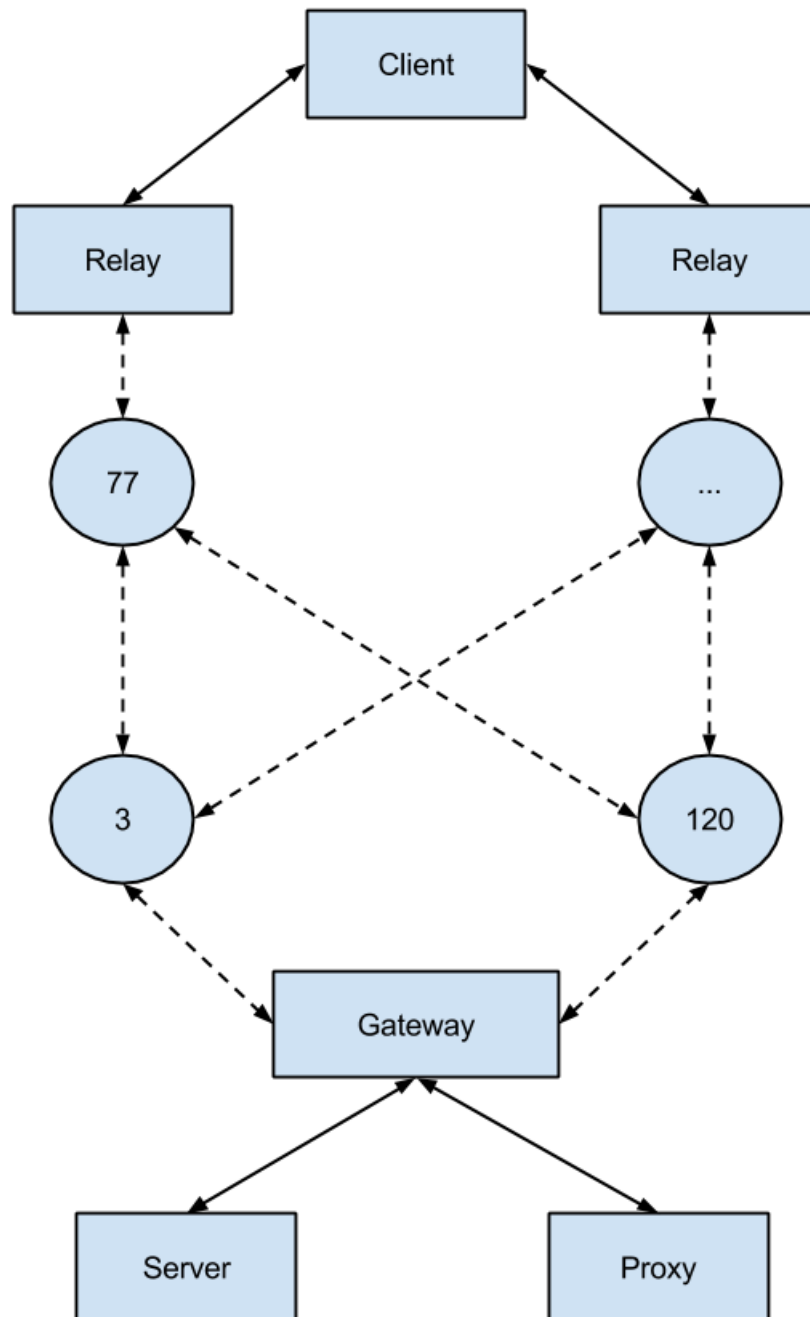


Figure 7.2: Testbed Configuration 1

### 7.1.8 Iteration Two

The next topology is shown in figure 7.3 and features an increase in the number of wireless nodes to six. This was done in order to introduce loops in the topology and increase the number of alternative routes if a node goes down. I now had access to all the available wireless nodes and so I had ten nodes of which I could choose, the six best connected, from. Using the binder-scan script enabled me to work out which other nodes were visible from each node. Since the client and gateway machines are the hardest to find good quality connections for, those edge nodes were first selected, followed by the two intermediary nodes. The intermediary nodes were chosen knowing the constraints of choosing the best possible connections for the client and gateway.

The mistake I made with this topology was not giving close enough attention to the strength and quality values of each available node. This resulted in a network that did not feature the best signal qualities possible. The configured network also did not use entirely non-overlapping channels which likely also affected performance. As a result, node to node throughput tests with Iperf resulted in less than 1Mbps in some cases which is too low to perform meaningful tests.

### 7.1.9 Iperf Test Procedure

In the following discussion of testbed configurations I mention Iperf tests which were used to discover the performance of the configurations. Each test was performed using the same script (iperf-test) and procedure to ensure comparable and reproducible results between testbeds. Each test procedure consisted of: a series of three sets of Iperf TCP and UDP tests, each test lasted a minute and the throughput was reported every second. Each test set was separated by five minutes to reduce the impact of potential environmental effects, such as interference from unrelated networks. To produce the final results for each test set the minimum, maximum, difference between min. and max., mean and standard deviation were calculated over all reported values from the series of three Iperf tests.

Since Binder currently generates solely TCP traffic, via the OpenVPN tunnel, the results from the TCP Iperf tests are of most interest. The UDP tests are useful when it is not possible to complete a TCP test, as happened in a few cases. Also, if there was a vast difference between the TCP and UDP results for the same link this could suggest an issue with TCP window sizes, rather than link layer instabilities in the network.

### 7.1.10 Iteration Three

This topology, as shown in figure 7.4 was designed to ensure that the average signal quality across the entire network was as high as possible which should hopefully result in the highest possible end to end throughput. It was when starting this topology design that the binder-scan-parser program was written, in order to sort the network scans by signal quality. Having a sorted wireless scan made it much easier to determine the best possible node to use in each case. The same strategy was used as before where I first chose the edge nodes and then chose the best intermediary nodes from those that remained. The best topology resulting from this only involved five nodes, removing the intermediary between the left-most nodes. This configuration also moved each relay, the proxy and the server onto distinct subnets to make routing easier to manage. Channel allocation for this configuration was performed such that each node would use non-overlapping channels where possible.

Per link throughput on this topology was typically good although the connection between node 3

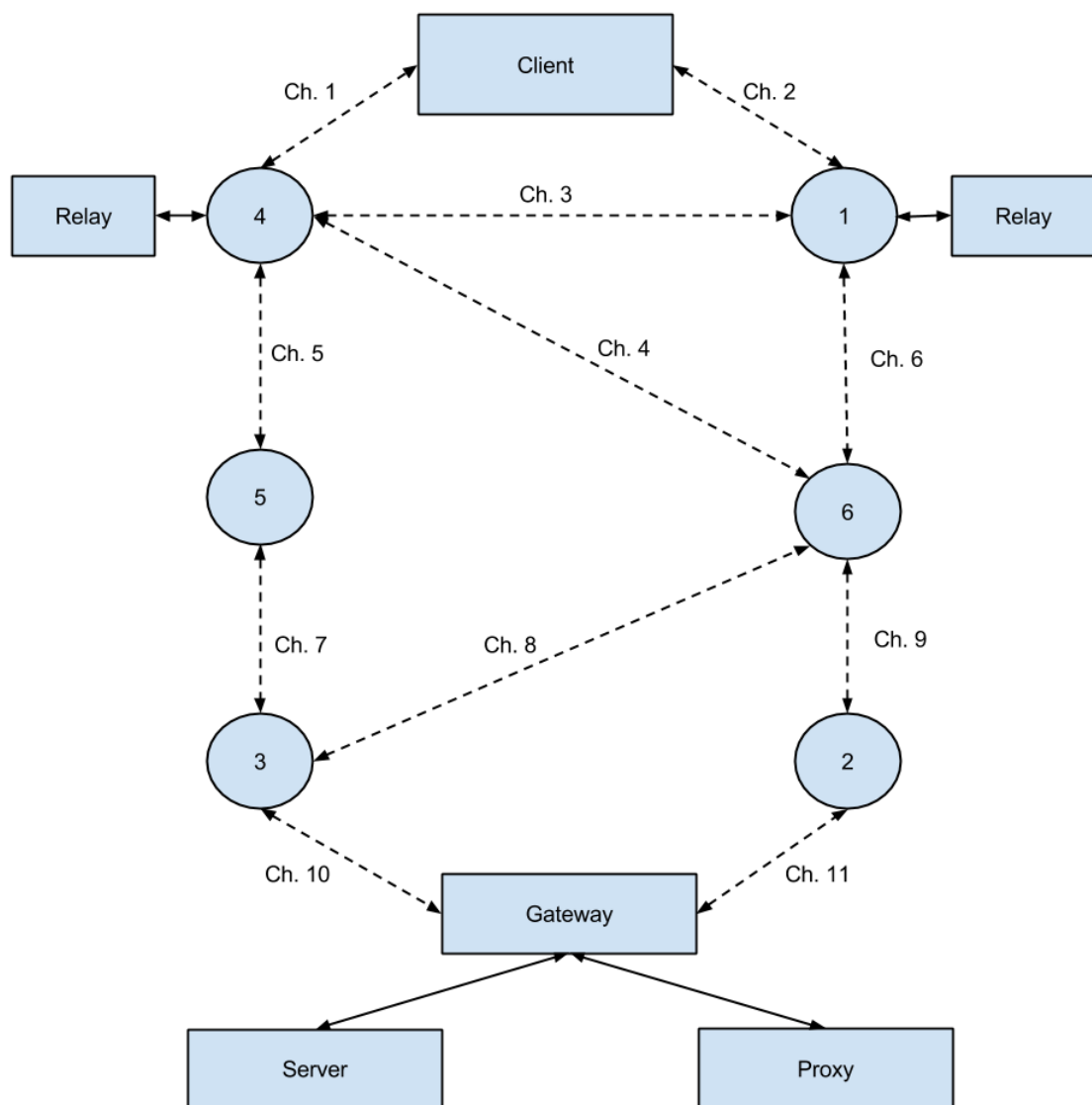


Figure 7.3: Testbed Configuration 2

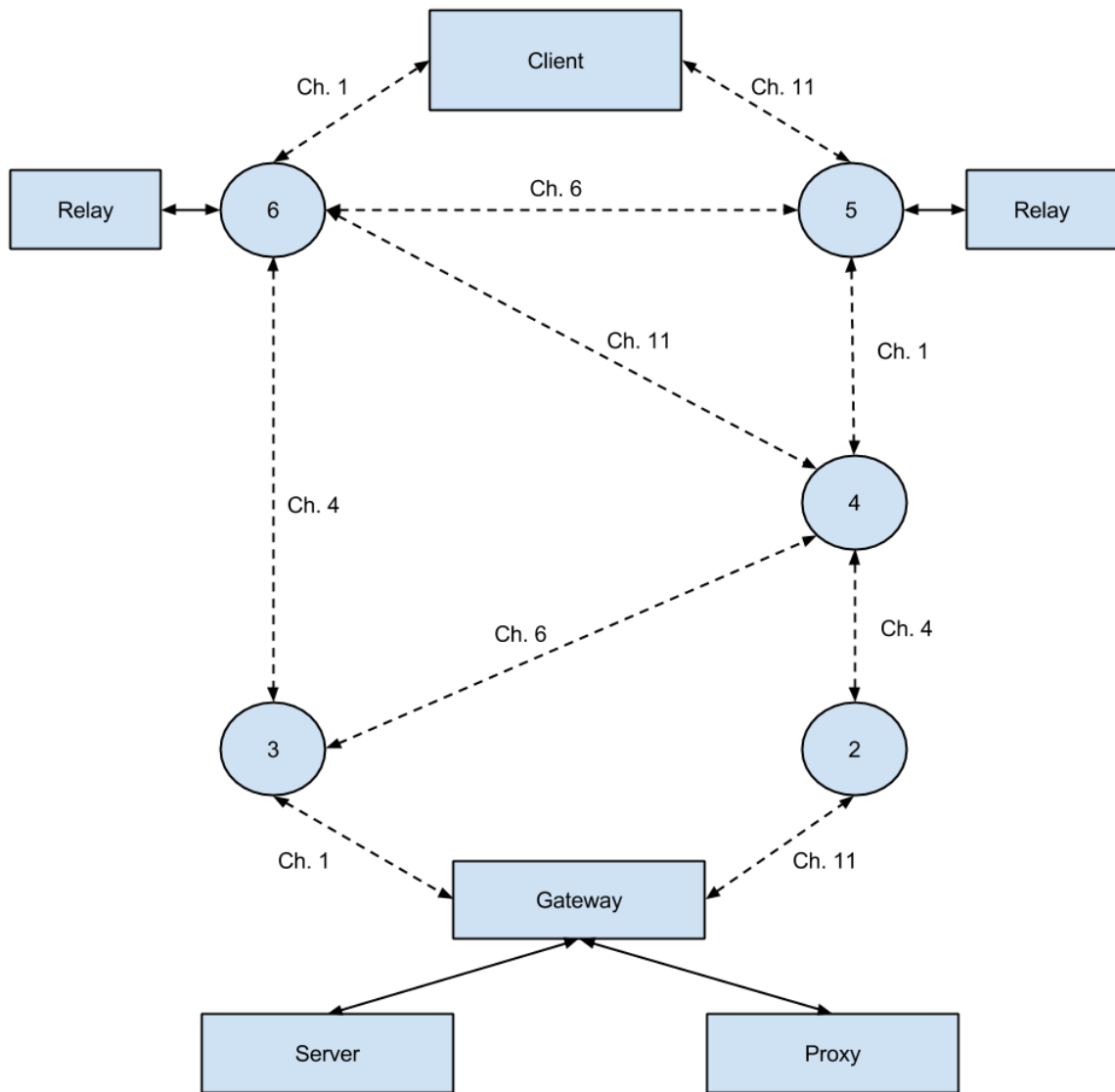


Figure 7.4: Testbed Configuration 3

	Client-6	Client-5	6-5	6-3	5-4	6-4	4-2	4-3	3-Gateway	2-Gateway
<b>Min</b>	4.19	1.05	22.00	3.15	10.20	8.85	1.05	6.29	8.39	3.15
<b>Max</b>	6.29	5.24	28.00	9.44	14.40	14.20	8.39	13.60	11.50	14.70
<b>Diff</b>	2.10	4.19	6.00	6.29	4.20	5.35	7.34	7.31	3.11	11.55
<b>Mean</b>	4.73	4.36	25.44	6.56	12.54	12.52	7.31	11.08	10.14	13.76
<b>Std. Dev.</b>	0.53	0.69	1.08	0.80	0.60	1.47	0.99	1.60	0.58	1.08

Figure 7.5: Testbed Configuration 3 - All Interfaces TCP Results (Mbps)

	Client-6	Client-5	6-5	6-3	5-4	6-4	4-2	4-3	3-Gateway	2-Gateway
<b>Min</b>	1.05	2.10	22.50	7.34	12.10	6.55	1.05	7.34	4.19	15.70
<b>Max</b>	4.19	5.24	28.90	11.50	14.90	15.30	9.44	14.70	11.50	19.90
<b>Diff</b>	3.14	3.14	6.40	4.16	2.80	8.75	8.39	7.36	7.31	4.20
<b>Mean</b>	2.66	4.08	26.11	9.93	13.53	13.83	5.39	12.39	10.34	18.80
<b>Std. Dev.</b>	0.63	0.57	0.74	0.79	0.44	1.14	2.47	1.21	0.70	0.78

Figure 7.6: Testbed Configuration 3 - Req. Interfaces TCP Results (Mbps)

and the gateway was unusable. This was strange since they are at a reasonably close physical distance (less than 10m). I checked that all interfaces in the testbed were transmitting on full power to ensure that the strongest possible signals were available. Following this, I performed a number of Iperf throughput tests. The test procedure was performed for each link twice, with all a node's interfaces up and also with only the required interfaces for the link under test up. By testing this way it should show the effect of interference of other interfaces' broadcasts on the link under test.

The issue with the node 3 to gateway connection was discovered to be an issue of channel 1, for some reason. Perhaps it was too crowded in the location of the two machines. As such the test results shown below for this link were obtained after changing the channel from 1 to 11. This also required changing the channel for the link between the gateway and node 2 from 11 to 6. The results for the tests can be found in figures 7.5 and 7.6.

The difference between the results for all interfaces and just the required interface are inconclusive. While most nodes have an increased average throughput when only the required interfaces are up, others do not. For example, the link between the client and node 6 has a better average and lower deviation from the mean when all interfaces are up than when only the required interfaces are up. Three links have lower deviation when all interfaces are up and three nodes have greater average throughput when all interfaces are up. The reason for these irregularities are possibly due to environment conditions that are beyond my control, for example there could have been a sudden influx of new smartphone users that were in range of my testbed during some of the tests but not others. Due to time taken to perform the tests, those with only the required interfaces up were done the day after those with all interfaces up. Since it is not possible to create a topology using only one node per interface, it is not possible to take advantage of any potential improvement in throughput gained through the use of only one interface per node.

Preliminary end to end throughput tests resulted in less than 500Kbps and showed that although each link alone had a decent throughput there was still an issue when nodes were simultaneously broadcasting heavily. Since nodes 6 and 4 both have 4 connections and thus require 4 interfaces it isn't possible to have entirely distinct channels; channel 4, which was arbitrarily chosen, overlaps with both channel 1 and 6. This overlap of channels could be causing interference that results in the poor end to end throughput when many interfaces are broadcasting.

#### 7.1.11 Iteration Four

In an attempt to create the fourth testbed topology I took a different approach. Instead of designing the entire topology from scratch before implementing it, I would set a small part of the topology up, test it to check performance and then continue expanding it by adding one node/link at a time.

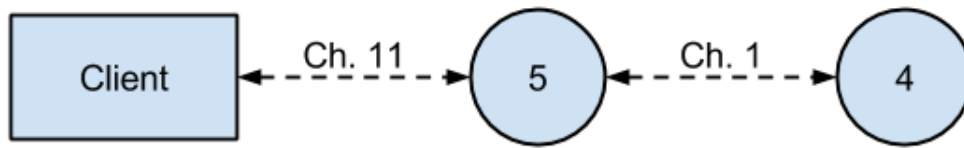


Figure 7.7: Testbed Configuration 4.1

Link	Client-5	5-4	Client-5-4
Min	3.19	11.50	0.00
Max	4.89	14.50	1.51
Diff	1.70	3.00	1.51
Mean	3.91	13.47	0.28
Std. Dev	0.88	0.54	0.15

Figure 7.8: Testbed Configuration 4.1 - TCP throughput (Mbps)

### Test Topology One

**Description** The topology for this test is shown in figure 7.7. It consists of the client connected to node 5 with node 4 linked to node 5. Both of the links are running on entirely non-overlapping 2.4Ghz channels (1 and 11). Only the required interfaces on each node were up and static routing was used.

**Results** The results obtained from Iperf testing are shown in 7.8. The results for the client to node 5 link were calculated only from the summary for each test, rather than each (per second) reported value during each test. This is due to the Iperf server on node 5 not reporting this value for some reason. The version of Iperf on node 5 differs from that on other nodes and Binder machines (2.0.4 single threaded vs 2.0.5 pthread) which could be the reason for this issue. The Iperf test results for this topology show some interesting details. The link between the client and node 5 is sufficient, achieving a mean 3.91 Mbps on the TCP test. The throughput from node 5 to node 4 is good, achieving an average of 13.47 Mbps on TCP. Unfortunately, running an end to end test from the client to node 4 shows extremely poor performance with a mean TCP throughput of 0.28 Mbps and low deviation from that value.

Link	5-4	Client-5-4
Min	4.92	2.41
Max	12.40	4.67
Diff	7.48	2.26
Mean	10.55	3.58
Std. Dev.	1.33	0.54

Figure 7.9: Testbed Configuration 4.1 alteration - TCP throughput (Mbps)

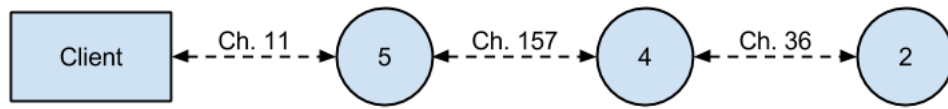


Figure 7.10: Testbed Configuration 4.2

Link	4-2
Min	1.67
Max	6.48
Diff	4.81
Mean	5.86
Std. Dev.	0.70

Figure 7.11: Testbed Configuration 4.2 - TCP throughput (Mbps)

**Alteration** I decided to change the link between node 5 and 4 from channel 11 to channel 157 which is a 5Ghz frequency. The tests were repeated from node 5 to 4 and also from the client to node 4. The results from these tests are shown in figure 7.9. The link throughput for TCP has dropped about 3 Mbps. However, performing an end to end test now results in much improved performance with an average 3.58 Mbps on TCP.

### Test Topology Two

**Description** I took the alteration made to the initial topology and added a link to node 2. Since I had a good experience with changing the link to 5Ghz in the previous topology I started by using channel 36 for the connection from node 4 to node 2. The topology is shown in figure 7.10.

**Results** The throughput from node 2 to node 4 was typically quite good, as I expected, with a mean 5.86 Mbps on TCP. It appears to be a very solid link so I performed an end to end test to node 2 from the client. The end to end test did not run as expected, I was unable to successfully complete a TCP throughput test and a UDP test gave a mean 2.44 Mbps throughput although with a high 58% packet loss. The results are shown in figures 7.11 and 7.12.

Link	4-2	Client-5-4-2
Min	4.63	1.07
Max	7.84	3.02
Diff	3.21	1.95
Mean	7.36	2.44
Std. Dev.	0.57	0.26

Figure 7.12: Testbed Configuration 4.2 - UDP throughput (Mbps)

Link	4-2	Client-5-4-2
Min	3.50	0.00
Max	4.79	3.43
Diff	1.29	3.43
Mean	4.00	2.31
Std. Dev.	0.20	0.52

Figure 7.13: Testbed Configuration 4.2 alteration - TCP throughput (Mbps)

Link	4-2	Client-5-4-2
Min	3.89	2.81
Max	5.29	5.22
Diff	1.40	2.41
Mean	4.63	4.17
Std. Dev.	0.28	0.44

Figure 7.14: Testbed Configuration 4.2 alteration - UDP throughput (Mbps)

**Alteration** The link between node 4 and node 2 was changed to run on channel 1 (2.4Ghz). This results in a lower link throughput, most likely as a result of the interference caused by greater use of the 2.4Ghz band. However, I could now complete an end to end TCP test which results in a low 2.31 Mbps while the UDP throughput increases to 4.17 Mbps. The reason for poor performance when using a 5Ghz channel for this link is most likely due to the decreased signal strength when compared to a 2.4Ghz channel. However, this should have also been the case for the node 4 to 5 link yet the result is the opposite. The throughput results are shown in figures 7.13 and 7.14.

### Test Topology Three

**Description** Moving forward, I took the latest iteration and added the binder gateway machine to the topology, linked to node 2. By extension, this also results in the server being added to the topology. A pattern has emerged in the previous iterations showing that subsequent links on the same band do not perform well in end to end tests, even when the channels in use do not overlap. For this reason I used

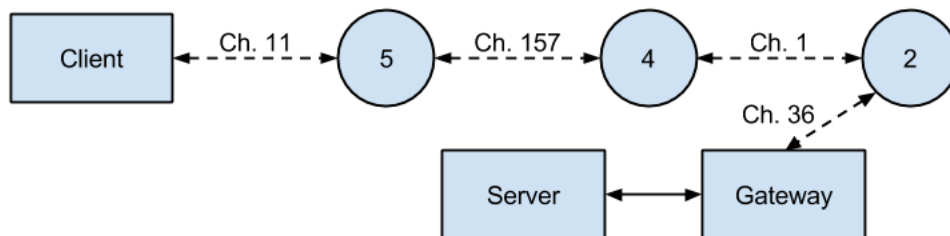


Figure 7.15: Testbed Configuration 4.3



Link	Gateway-Server
Min	935.00
Max	937.00
Diff	2.00
Mean	936.76
Std. Dev.	0.44

Figure 7.16: Gateway-Server TCP throughput (Mbps)

Link	2-Gateway	Client-Server	Client-Server (OSPF)
Min	2.46	0.00	0.00
Max	7.26	2.93	2.98
Diff	4.80	2.93	2.98
Mean	6.26	2.43	2.53
Std. Dev.	0.49	0.42	0.46

Figure 7.17: Testbed Configuration 4.3 - TCP throughput (Mbps)

a 5Ghz link between the gateway and node 2, with channel 36. The server is linked to the gateway by Gigabit Ethernet, as expected this is a stable link with extremely high throughput. Results from tests between the gateway and server are provided in figure 7.16 to show that this link is not a bottleneck when used in end to end tests from the client to the server. The topology is shown in figure 7.15.

**Results** The throughput for the link between the gateway and node 2 is encouragingly good, with a mean 6.26 Mbps on the TCP test. Performing an end to end test from the client to the server results in a vastly lowered throughput, as expected due to the bottlenecks of other links in the network. I also performed a test with dynamic routing, via OSPF, to determine the performance effects, either due to processor or bandwidth use, over static routing. The results show that there is no appreciable effect on performance with OSPF enabled, although TCP throughput is slightly increased. Realistically, however, the results do not show an effect on performance by enabling OSPF. The variations are just typical from what I have seen while performing tests on these testbeds. These results are shown in figure 7.17.

Link	Client-Server
Min	0.00
Max	3.97
Diff	3.97
Mean	0.57
Std. Dev.	0.58

Figure 7.18: Testbed Configuration 4.3 alteration - UDP throughput (Mbps)

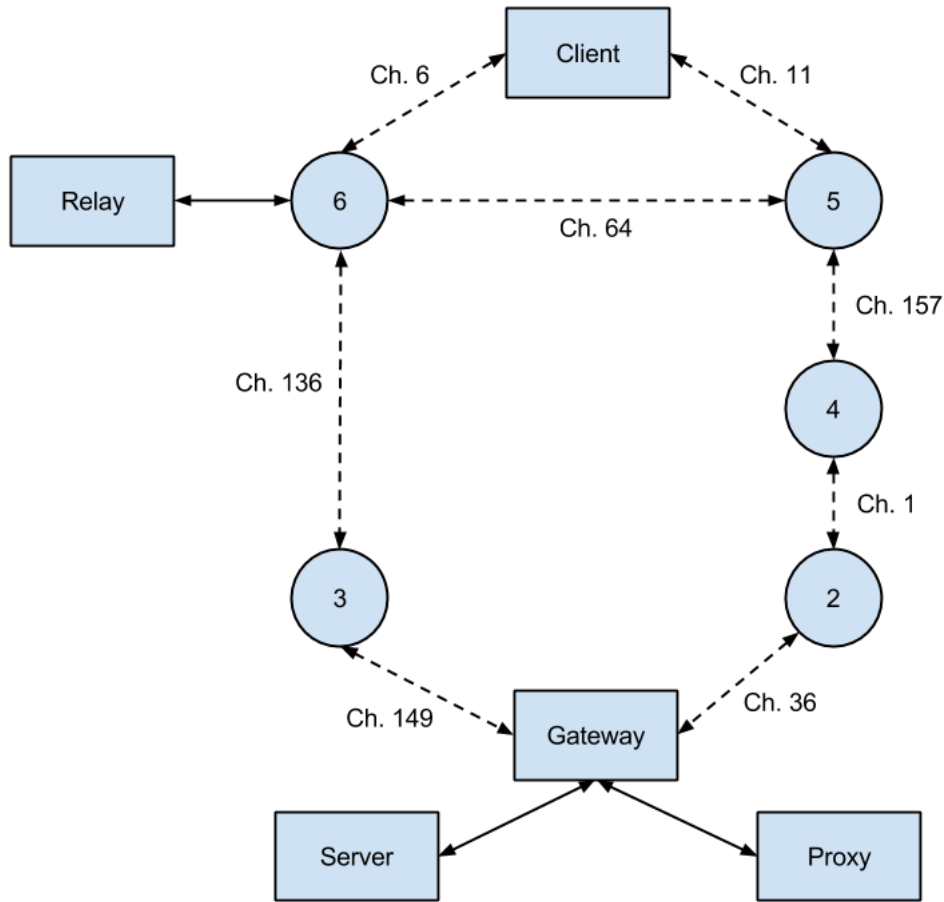


Figure 7.19: Testbed Configuration 4.4

**Alteration** I switched the link from the gateway to node 2 from channel 36 to the 2.4Ghz channel 6 in order to test the performance of using 2.4Ghz for this link. The results were remarkable in their difference. I could not complete an end to end TCP test successfully and average UDP throughput was a low 0.57 Mbps, as shown in figure 7.18.

#### Test Topology Four

**Description** With one branch of the network complete it is now possible to add in another branch. Adding another branch will allow the Binder system to be tested as it provides two separate routes for traffic across the network. Adding nodes 3 and 6, I am forced to use a 2.4Ghz channel for the link from the client to node 6 in order for them to be in range. I will start by using distinct 5Ghz channels for the remaining links. The topology is shown in figure 7.19.

**Results** From my previous tests I have learnt that the link between the client and node 6 is limited to about 4.7 Mbps for TCP traffic. To ensure that this link is not a bottleneck for aggregated bandwidth, I started by performing end to end tests for each branch of the network from the relay to the server. Since this machine is connected to node 6 over an Ethernet connection there is no bottleneck here. Both branches reported sufficient mean TCP throughput, 2.38 and 2.39 Mbps, to perform a test of

Link	R-6-3-G-S	R-6-5-4-2-G-S
Min	0.00	1.76
Max	3.39	2.84
Diff	3.39	1.08
Mean	2.38	2.39
Std. Dev.	0.55	0.18

Figure 7.20: Testbed Configuration 4.4 - Branch TCP throughput without Binder (Mbps)

Link	Relay-Server
Min	0.00
Max	2.35
Diff	2.35
Mean	1.12
Std. Dev.	0.90

Figure 7.21: Testbed Configuration 4.4 - Branch TCP throughput with Binder (Mbps)

Binder and show the aggregated bandwidth of both branches. The 6-3 branch is potentially less stable than the 6-5-4-2 branch which is shown by the minimum throughput figure of 0 Mbps and greater deviation from the mean throughput. Due to instabilities in the OSPF routing daemon on some nodes, I used static routing on this testbed configuration. The full results from the TCP test are shown in figure 7.20.

### 7.1.12 Binder

Performing a test of Binder requires it to be configured on the relay and proxy machines. On both the relay and proxy MPTCP must be enabled and the number of flows set to three (through the `ndiffports` option). This allows for the main flow, along with two sub-flows which can be source routed separately along each branch of the network. At the relay, two source route lists are configured which represent the two different branches in the network: 6-3-Gateway and 6-5-4-2-Gateway. Now that MPTCP is correctly configured the OpenVPN tunnel must be setup, the server is started at the proxy followed by the client at the relay. Since MPTCP is already configured, the tunnel will now be split over three flows, two of which are source routed along both branches of the network. A default static route at the relay forwards all traffic without another route through the tunnel.

### Tests

With the Binder system configured, I performed the throughput tests. The tests were run end to end from the relay to the server machine, via the proxy. Since the relay, proxy and server machines are connected into the testbed via Ethernet there are no bottlenecks here and the aggregated bandwidth should be very roughly that of each branch of the network combined.

Link	Relay-Server
Min	0.72
Max	2.48
Diff	1.76
Mean	1.66
Std. Dev.	0.34

Figure 7.22: Testbed Configuration 4.4 - Branch UDP throughput with Binder (Mbps)

Link	6-3-G-S	5-4-2-G-S
Min	0.00	0.26
Max	3.47	3.01
Diff	3.47	2.75
Mean	0.18	2.42
Std. Dev.	0.67	0.28

Figure 7.23: Testbed Configuration 4.4 - Concurrent branch TCP throughput (Mbps)

## Results

The results obtained are shown in figures 7.21 and 7.22 and do not show that Binder has increased throughput on the network. In fact a negative effect is shown, the average TCP throughput is 1.12 Mbps, a considerable drop from what was achieved on both branches without the use of Binder. The mean UDP throughput is not much better at 1.66 Mbps showing that the TCP throughput result was not greatly affected by the TCP on TCP connection. Since Binder has previously been shown to be effective on a wired network, I was surprised that it caused a serious detrimental effect to throughput.

### 7.1.13 Concurrent Tests

Due to my previous issues with maintaining a useful end to end throughput on the testbed I decided that there could be an issue with both branches of the network transmitting concurrently, rather than there being an issue with Binder. In order to test this hypothesis I ran end to end tests as follows: 6-3-Gateway-Server and 5-4-2-Gateway-Server. Testing in this way also ensures that the bottleneck is not at node 6, as all traffic from the relay was routed via node 6 in the previous Binder tests.

## Results

The results, shown in figure 7.23 from these latest tests confirm my suspicions that the effects of Binder were not shown due to unreliability in the wireless testbed, rather than an issue in Binder. The TCP throughput achieved on the route through nodes 6 and 3 was an average of 0.18 Mbps, while the other branch measured a mean throughput of 2.42 Mbps. A vast difference is apparent between the result for the 6-3 route compared to the result previously achieved when testing the branch alone. It was hoped that by using distinct channels that any issues with interference would be avoided. The issue is most likely introduced at the gateway since this is where both branches converge. The gateway

is configured with two wireless interfaces which, due to availability of PCI slots, are located closely together. It could be that, despite the use of isolated channels, the close proximity of the antennas is causing the issues.

## 7.2 TCP Splitting

In order to be able to source route traffic on the network, Binder uses an OpenVPN tunnel between the relays and the proxy which runs over TCP. The tunnel is then split into MPTCP sub-flows. Performance issues can be encountered when TCP is running on top of a TCP tunnel [35]. When loss is experienced, the TCP segments may be re-sent by either the tunnel sender or the original source causing an increase in traffic being sent across the network and the destination likely receiving duplicate TCP segments which have been sent multiple times by the source and the tunnel.

In order to prevent this use of TCP connections on top of another TCP connection it is possible to use a technique known as TCP splitting. A middlebox or boxes within the network, between the source and destination, terminate the TCP connection from the source, maintaining its state, and then open a new connection to the destination. In the context of Binder it is possible to use two conceptual middleboxes, at the relays and the proxy, such that one TCP connection actually becomes three: source to relay, relay to proxy and proxy to destination. An OpenVPN tunnel running over UDP would then be used solely for routing the traffic between the relays and proxy, rather than also creating the MPTCP sub-flows. When a split TCP connection is established between either of the relays and the proxy it will become an MPTCP connection since they are both MPTCP capable.

I have installed and tested the effects of tcp-intercept, mentioned earlier, on the development testbed used for testing the IP Option stripping solution. Following this I also configured it on the relays and proxy machines on the testbed so that the TCP over UDP tunnel can be compared to a TCP over TCP tunnel in future experiments.

## 7.3 Scripts

### 7.3.1 Wireless Nodes

#### **binder-scan**

While implementing the wireless mesh network portion of the testbed it was necessary to discover which of the boards were in range of each other. This script may be run on each wireless node and performs a scan for other nodes on each wireless interface. The output of each scan is filtered to only contain pertinent information (SSID and signal strength/quality). In my usage the output from this script was redirected to a file so that I could later sort and print the results in order to make it easy to design network topologies.

#### **node-backup**

There are ten nodes available to be used in the mesh testbed. These nodes may be used by many people for differing experiments. It is critical that backups are made to reduce inconvenience to other users. This script copies the /root and /etc directories on each node into a local directory on the calling machine. These directories contain results and scripts, and configuration respectively.

## **switch-mode**

There are several operations which are performed frequently on the wireless nodes. These operations usually require editing a configuration file and then restarting the network service. This script automates performing some of these tasks. It takes one argument which is the operation to be performed. The operations available are to enable or disable all of the wireless interfaces on a node and to switch any enabled interfaces between station and access point mode.

## **binder-scan-parser**

This Java program is used to sort the output from the binder-scan script. This is necessary because the output is not very readable for the purposes of finding the strongest signals on each node. The wireless scan results are not provided in a sorted order, I particularly wished to browse the scans in order of strongest signal to weakest.

The program takes input in the form of a single scan file from a node and a directory for output. A sorted list of scan results is generated as a separate file for each interface in the output directory. Two classes make up the program: BinderScanParser and ScanResult.

**ScanResult** This class represents an individual scan result. It stores the information about the result which is gained from the scan file. The `getSignalStrength()` operation is provided which parses the result description to obtain the strength of the signal. When adding to a result description, as the scan file is read, the text is cleaned up so that it will be easier to read once the final output is written.

**BinderScanParser** The main logic is contained in this class. It is responsible for reading the input, sorting the contents and then writing the output files. A mapping of interface to a list of scan results from the interface is maintained which allows writing separate output for each interface.

The class reads in the scan file and processes each line in turn. When it encounters a new interface, it creates a new mapping for the interface to a new list of scan results. When a new scan result is encountered it inserts the current result into the current interfaces list and then creates a new `ScanResult` object to continue with. Other lines in the scan file can be assumed to be part of a particular scan result so when encountered they are appended to the description for the current `ScanResult`.

The sorting is done each time a new `ScanResult` is inserted into an interface's list. The list is iterated and the result inserted at a position such that its signal is greater than that of the result at the current position in the list. Although this method of sorting is not particularly efficient, since the list is iterated many times, it is both reliable and sufficient for the purposes of this program. The number of elements in each list is never more than 31 since this is the total number of broadcast capable interfaces across all of the wireless nodes. In fact, it is always up to four less elements since the binder-scan script filters out results from the scanning node's other interfaces. With so few elements the sorting algorithm employed is more than acceptable.

Once a complete sorted list of `ScanResult` objects has been obtained for each interface, the program then writes the output files. For each interface a new output file is opened and the description of each `ScanResult` for that interface is written to the file in sorted order of signal strength.

### 7.3.2 Iperf

#### **iperf-test**

This script was written to ensure that the same Iperf tests were run on many different nodes which made up the wireless mesh testbed. The Iperf tests were used to check the quality of each of the links in the network, under different conditions.

The address of the Iperf server and an output directory is provided to the script and a series of both TCP and UDP tests are run. For each test that is run a file is generated in the output directory which contains the results of that test. The number of tests to run is configurable, along with the length of each test, interval between tests and the interval between reporting running results during each test.

#### **iperf-test-server**

This is a very simple script that allows starting an TCP Iperf server instance, as well as a UDP instance through the use of one command. This saves time and frustration in cases where you may forget to start one instance or the other!

A single argument must be provided, which is the output directory. A single output file will be created in this directory for each of the TCP and UDP servers. Each file will detail the current results, per second, of each test aimed at the corresponding Iperf server.

### 7.3.3 D-ITG

#### **testbed**

This script implements a function that allows a command to be run on a machine that is part of the Informatics Forum testbed. Since remote access to the testbed network is only possible through a single gateway machine it means that a minimum of two successive logins are required in order to access a machine. In some cases three logins may be required. With more than ten machines within the testbed this can be tedious if commands must be run on many machines.

The function can take any number of arguments. The first argument must always be the desired command to be run on a machine. Each following argument is the address of a machine that must be logged in to in order to access another machine. The last argument is therefore the address of the target machine. The function sets a variable that can then be accessed by the calling script. This variable contains a chain of embedded ssh calls which will result in the command being run on the target machine.

#### **itg\_send-recv\_setup**

The setup of the ITGSend and ITGRecv daemons are very similar. To reduce duplicating code in the scripts to configure both of these daemons, it is maintained in this script which is sourced by the dedicated scripts for each daemon. Reducing the duplication removes the likelihood of error and eases maintenance. It sets out the required arguments and usage text if these are not provided. It also retrieves the address of the ITGLog server and the suffix that should be used on log files and places them in variables accessible to the dedicated scripts.

### **start-sender**

This script is responsible for configuring an ITGSend daemon on a testbed machine. It makes use of the previously mentioned testbed and itg\_send-recv\_setup scripts in order to receive options and launch the daemon on the target machine.

Arguments to this script are the log server address and suffix, followed by the path of machines to reach the target.

### **start-receiver**

Almost identical to start-sender, this script is responsible for configuring an ITGRecv daemon on a testbed machine. The same arguments must be passed.

### **start-log**

This script also makes use of the testbed script to launch the ITGLog daemon on the target machine. The arguments to the script must be the chain of machines used to reach the target.

### **retrieve-logs**

Following the running of a test it is necessary to retrieve the log files generated by the ITGLog daemon. Doing this manually is frustrating and time consuming. This is because the logs must be copied from the log machine to the SSH gateway machine so that they can finally be copied out of the testbed network for analysis. This script also makes use of the testbed script to run a command on a target machine, scp in this case.

The local destination directory for the log files must be provided as an argument. All log files on the machine running ITGLog will then be copied locally and then removed from the remote host.

### **launch**

It can be frustrating and time consuming to manage multiple instances of each script mentioned above. This script launches a script in a new xterm window with a descriptive title. Another advantage is that all xterm windows can be easily killed at once (for example using the killall command) which makes it quicker to reset and start again if there is an error on the testbed.

The arguments to this script are the script to run and the title to display in the xterm window.

### **setup**

This script is a compilation of the previous scripts which allows me to configure the testbed to my needs through the use of one command. No arguments may be passed.



### **7.3.4 TCP Splitting**

#### **setup-tcp-intercept**

This script is based on the steps provided in the tcp-intercept README [42]. Setting up tcp-intercept requires configuring several iptables rules and a new chain, among others. setup-tcp-intercept automates the process of performing these changes. If the disable argument is provided then the changes are reversed and, once the tcp-intercept process is killed, the machine can return to operating normally by only forwarding TCP traffic.

## Chapter 8

# Conclusion

### 8.1 Evaluation

This project was considerably more involved than I had originally anticipated. The issues I encountered with attempting to implement a stable wireless mesh testbed, along with the time and work required to upstream the changes Binder has added to MPTCP were the chief factors in this. Moreover, the large number of scripts written during the project were required in order to work efficiently during both development and while deploying testbed configurations. Despite the issues I experienced, I believe that these will be useful for future work on Binder and that overall I have made worthwhile contributions to the project.

#### 8.1.1 Implementation of LSRR in MPTCP

As mentioned previously the work in this area is ongoing. The changes to MPTCP, required to support Binder, have gone through a number of significant revisions in order to satisfy the MPTCP developers. This has resulted in a smaller, less complicated solution and I am confident that we are approaching a stage where the code will be accepted. Further testing along with final clean up and code checking will take place before this happens, however.

#### 8.1.2 D-ITG Applications

My work on supporting applications for D-ITG has resulted in two new projects: JITGApi will enable easier development of Java applications for controlling D-ITG and ITGController makes orchestrating a D-ITG testbed significantly easier. I am happy with the results of these projects and, now that they have been submitted to the D-ITG community, they may prove useful to other users of D-ITG. Once a stable testbed for Binder has been created, they will also help in the running of extensive real-world tests.

#### 8.1.3 IP Option Stripping

The IP Option Stripping solution I have implemented provides a solid foundation that can be further improved upon. At the current stage it performs removal of IP Options which acts as a proof-of-concept showing the viability of the complete solution, which will involve re-inserting the removed options for returning traffic. A description of what may be involved to complete the final solution is detailed in section 8.2.1.

#### 8.1.4 Measurements

Attempting to perform measurements of Binder on a wireless testbed has been the most frustrating part of this project. At the start of the project, this objective was one that I believed would present the least issues. In retrospect, it became the most difficult and time consuming. My experiences in designing and implementing the testbeds shows how difficult this can be, especially when environmental factors such as interference from other networks is involved. Although I have been unable to show the effectiveness of Binder on a wireless network, this is not due to the Binder system but the issues of the testbed. My results will aid future work in the measurement of Binder and a number of possible avenues for future work in this area are detailed in section 8.2.5.

## 8.2 Future Work

### 8.2.1 Re-inserting Stripped IP Options

As mentioned previously, in an ideal case the IP Option stripping solution would be able to re-insert the appropriate options for return packets that enter the Binder network. This is because the options are used to route the return packets by the original route used for the outgoing packets. The source of the return packets cannot explicitly route them, both due to lack of knowledge of the topology of the Binder network and the issues of packets with IP Options on the Internet.

In order to re-insert the correct options for the corresponding packets, the `xt.IPOPTSTRIP` kernel module would need to store the options when they are removed from the packet buffer as an outgoing packet is being mangled. The best solution to this would be some form of hash table. The Linux kernel does not currently provide a generic hash table implementation so this throws up some issues of itself. There would be a couple of approaches to solving this: writing a hash table implementation based on another already present in the kernel or using `uthash` [50]. The first solution would be best used if the IP Option stripping solution was to be provided upstream, given it is a Netfilter module and not natively compatible with `nftables`, this is unlikely. `uthash` provides a generic hash table type which is implemented entirely as pre-processor macros in a single header file which would make it easy to integrate into the `xt.IPOPTSTRIP` module build process. This would likely be the best option for getting a working solution running quickly.

Once a hash table is available the implementation is more straightforward. The hash key used for each entry would be the source and destination IP addresses, along with the source and destination ports of an outgoing packet. If an entry with such a key is not in the table, then the data removed from the packet buffer would be inserted using that key. Incoming packets would provide source IP/port as destination IP/port and vice-versa in order to perform a look up in the hash table. If an entry exists, the options data can be retrieved and appropriately re-inserted into the buffer. The Netfilter `conntrack` system can be used to determine the direction of a packet when received by `xt.IPOPTSTRIP` [33].

Since the `conntrack` system does not detect the close of a connection, at the network layer, then there would need to be some other procedure within `xt.IPOPTSTRIP` that would age and ultimately remove entries from the hash table as they are no longer required. This could be achieved through the use of a timestamp or time-to-live value that would be stored in the hash table along with options data. Each time a packet is processed, it would update this value which indicates that the entry in the hash table is still in use. A separate thread in the module, perhaps using a work queue [43], would iterate over

the entries in the hash table, checking when each entry was last used. Entries which had passed some threshold would be removed from the hash table.

### **8.2.2 Routing Protocol for Low-Power and Lossy Networks**

A solution based on the routing header proposed for use by RPL [27] was not feasible based on the maturity of these standards at the time of this project. Future work on Binder should watch the development of the standards as it may be possible to implement an IPv6 source routing solution based upon them in the future.

### **8.2.3 OLSR**

I was unable to utilise OLSR dynamic routing on any of the testbeds which was unfortunate. The OLSR implementation intended for use on the testbed, olsrd [6], makes use of wireless interfaces in ad-hoc mode. This means that a defined topology is not required, each node will communicate with any other node that it is in range of. This would alleviate some of the issues that I had while attempting to implement the testbed; that is trying to discover the best possible topology based on the signal qualities of each node.

### **8.2.4 ITGController Unit Tests**

I would like for any future work on ITGController to include completion of a set of comprehensive JUnit tests which test the full range of functions from config file parsing and loading to receiving and processing responses from the API. Although I have performed significant runtime testing with different configs, both correct and malformed, the program would be much better served by automated testing. This would ease further development or maintenance in the future, particularly by 3rd parties.

### **8.2.5 Measurements**

Since I was unable to correctly implement a testbed in which to measure the effect of Binder, future work will require designing a stable testbed and then performing tests to categorise Binder. Given my experiences with attempting to create a wireless testbed that is stable enough to test Binder there are a couple of avenues which I believe should be explored:

#### **Gateway Alterations**

To reduce the possibility of interference being introduced at the gateway machine it could be possible to use antennas which connect to the interface via a cable rather than directly. This would allow the antennas for each interface to be placed so that they can both obtain a better signal to the appropriate wireless node and be further from each other in order to reduce the possibility of interference. Alternatively, there could be multiple gateway machines with only a single wireless interface each. This could potentially provide the protection from interference as well as more closely emulate a deployed mesh network, where there are multiple Internet gateways.

#### **Machine/Node Placement**

While attempting to deploy a testbed I had to make do with where the Binder machines, as well as the wireless nodes, were physically located. Due to the nature of the building layout and machine loca-

tions I think that this contributed significantly to the difficulties I experienced. By moving machines to more strategic locations it would be possible to obtain a better signal for several links which would help increase throughput.

# References

- [1] Apache Ant. <http://ant.apache.org/>.
- [2] iptables. <http://www.netfilter.org/projects/iptables/index.html>.
- [3] Java Native Access (JNA). <https://github.com/twall/jna>.
- [4] linux-rpl. <https://github.com/joaopedrotaveira/linux-rpl>.
- [5] The netfilter/iptables patch-o-matic system. <http://www.netfilter.org/projects/patch-o-matic/>.
- [6] olsrd. <http://olsr.org/>.
- [7] Open shortest path first. [http://docwiki.cisco.com/wiki/Open\\_Shortest\\_Path\\_First](http://docwiki.cisco.com/wiki/Open_Shortest_Path_First).
- [8] OpenWRT. <https://openwrt.org/>.
- [9] OpenWRT Bug #5330 - Atheros adhoc manual channel selection. <https://dev.openwrt.org/ticket/5330>.
- [10] Openwrt wiki - network configuration. <http://wiki.openwrt.org/doc/uci/network>.
- [11] Openwrt wiki - wireless configuration. <http://wiki.openwrt.org/doc/uci/wireless>.
- [12] Patch-o-matic ipv4optstrip extension. <http://www.netfilter.org/documentation/HOWTO/netfilter-extensions-HOWTO-4.html#ss4.2>.
- [13] Quagga routing suite. <http://www.nongnu.org/quagga/>.
- [14] SimpleRPL. <https://github.com/tcheneau/simpleRPL>.
- [15] VirtualBox. <http://virtualbox.org>.
- [16] VirtualBox Ticket 12748. <https://www.virtualbox.org/ticket/12748>.
- [17] Wireshark Wiki - Pipes. <http://wiki.wireshark.org/CaptureSetup/Pipes>.
- [18] Xtables-addons. <http://xtables-addons.sourceforge.net/>.
- [19] Arnaud Ebalard. IPv6 Type 0 Routing Header. *IETF Journal*, 3, 2007.
- [20] L. Bocassi and D. Eastoe. Binder: Mptcp with source routing git repository. <https://www.github.com/bluca/mptcp/>.

- [21] L. Bocassi, M. Fayed, and M. Marina. Binder: A system to aggregate multiple internet gateways in community networks, 2013.
- [22] Linux kernel ethernet bonding driver howto. <https://www.kernel.org/doc/Documentation/networking/bonding.txt>.
- [23] A. Botta, A. Dainotti, and A. Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.
- [24] K. Chebrolu and R. R. Rao. Bandwidth aggregation for real-time applications in heterogeneous wireless networks. *Mobile Computing, IEEE Transactions on*, 5(4):388–403, 2006.
- [25] T. Clausen and P. Jacquet. RFC 3626 - Optimized Link State Routing Protocol (OLSR), 2003.
- [26] J. Corbet. The return of nftables. <https://lwn.net/Articles/564095/>, Aug 2013.
- [27] D. E. Culler, V. Manral, and J. W. Hui. RFC 6554 - An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL). 2012.
- [28] D. Eastoe. Dissertation scripts git repository. <https://www.github.com/duncanje/dissertation/>.
- [29] D. Eastoe. ITGController Git Repository. <https://www.github.com/duncanje/ITGController>.
- [30] D. Eastoe. JITGApi Git Repository. <https://www.github.com/duncanje/jitgapi>.
- [31] D. Eastoe. JITGApi JavaDoc. <https://duncanje.github.io/jitgapi/doc/0.2>.
- [32] D. Eastoe. xt.IPOPTSTRIP Git Repository. <https://www.github.com/duncanje/xt-IPOPTSTRIP/>.
- [33] J. Engelhardt and N. Bouliane. Writing netfilter modules. Jul 2012.
- [34] Flowgrind. <https://github.com/flowgrind/flowgrind>.
- [35] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama. Understanding tcp over tcp: effects of tcp tunneling on end-to-end throughput and latency. In *Optics East 2005*. International Society for Optics and Photonics, 2005.
- [36] HUBS. <http://www.tegola.org.uk/>.
- [37] IEEE. IEEE Standard for Local and metropolitan area networks - Link Aggregation. Technical report, IEEE, 2008.
- [38] IETF. RFC 6275 - Mobility Support in in IPv6, 2011.
- [39] Information Sciences Institute, University of Southern California. RFC 791 - Internet Protocol, 1981.
- [40] J. Korhonen. Mobile ipv6 in linux kernel and user space. In *Proceedings of Seminar on Network Protocols in Operating Systems*, page 40.
- [41] S. Lakshmanan, R. Sivakumar, and K. Sundaresan. Multi-gateway association in wireless mesh networks. *Ad Hoc Networks*, 7(3):622–637, 2009.

- [42] N. Laukens. tcp-intercept. <https://github.com/VRT-onderzoek-en-innovatie/tcp-intercept>.
- [43] M. Tim Jones. Kernel APIs, Part 2: Deferrable functions, kernel tasklets, and work queues. <http://www.ibm.com/developerworks/library/l-tasklets/l-tasklets-pdf.pdf>, Mar 2010.
- [44] Network Working Group. RFC 2960 - Stream Control Transmission Protocol, 2000.
- [45] Network Working Group. RFC 5095 - Deprecation of Type 0 Routing Headers in IPv6, 2007.
- [46] OpenVPN. <http://www.openvpn.net/>.
- [47] QEMU. <http://www.qemu.org/>.
- [48] C. Raiciu, C. Paasch, S. Barr, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12)*, San Jose (CA), 2012.
- [49] V. Semken. Graphical User Interface for D-ITG 2.7. <http://www.semken.com/projekte/index.html>.
- [50] Troy D. Hanson. uthash: a hash table for C structures. <http://troydhanson.github.io/uthash/>.
- [51] Wikipedia. IPv4. <http://en.wikipedia.org/wiki/IPv4>.
- [52] Wikipedia. IPv4 header checksum. [http://en.wikipedia.org/wiki/IPv4\\_header\\_checksum](http://en.wikipedia.org/wiki/IPv4_header_checksum).
- [53] Wikipedia. List of wireless community networks by region. [http://en.wikipedia.org/wiki/List\\_of\\_wireless\\_community\\_networks\\_by\\_region/](http://en.wikipedia.org/wiki/List_of_wireless_community_networks_by_region/).
- [54] Wikipedia. List of WLAN channels. [http://en.wikipedia.org/wiki/List\\_of\\_WLAN\\_channels](http://en.wikipedia.org/wiki/List_of_WLAN_channels).
- [55] T. Winter, P. Thubert, and R. Team. RFC 6550 - RPL: IPv6 routing protocol for low power and lossy networks. *IETF ROLL WG, Tech. Rep*, 2012.
- [56] Wireshark. <https://www.wireshark.org/>.