Faulty Network Router

Thomas William Salmon

Module: CS39030

Supervisor: Dave Price

A dissertation submitted in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Software Engineering in the University of Wales, Aberystwyth.

April 21, 2004

Module code: CS39030

Declaration of Originality

This submission is my own work, except where clearly indicated.

I understand that there are severe penalties for plagiarism and other unfair practice,which can lead to loss of marks or even the withholding of a degree.

I have read the sections on unfair practice in the Students Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science. I understand and agree to abide by the University's regulations governing these issues.

Signature:..... (Thomas Salmon)
Date:.....

Acknowledgements

I would like to thank my supervisor, Dave Price, for suggesting the project and for providing much guidance on the project.

I have benefited from conversations held with Neil Rogers, who has undertaken a similar project that tackles different aspects of the same problem.

Abstract

The Internet is being used by an ever increasing number of multimedia applications, such as video conferencing and voice over IP. These applications are expected to deliver a good quality service, with a minimal delay.

However, the Internet is a large collection of networks, that do not always function perfectly. Applications and systems should attempt to operate as well as possible, even in less than ideal conditions.

To test how well these applications can work, under poor network conditions, there needs to be a mechanism for generating network faults, in a controllable manner.

There are many ways in which a large wide area network can malfunction and operate incorrectly. Only two aspects of improper operation are within the scope of this project:

1. Packet Drop, discarding of IP packets
2. Packet Reordering, IP packets arriving in a different order to that which they were sent

These operations are performed on a GNU/Linux based router. Using one router (one hop), these faults which normally occur when dozens of routers are traversed, can be reproduced in a controlled manner.

How the router should behave incorrectly, is specified by an arbitrary number of rules, supplied by the user. Packet size and transport layer protocol can be specified as parameters to these rules that govern the faulty behaviours.

Development of this project has been completed to schedule, and undergone testing. It satisfies the original specification and requirements. However, there is scope for future extensions, which will be released in the forthcoming months.

Contents

Chapter 1

Objectives

1.1 Context

Over recent years, large wide area networks such as the Internet, have been increasingly used for real time data transfer. Uses of the Internet for real time data transfer include video conferencing and voice over IP applications. As bandwidth availability is increasing, so is the use of such applications.

Previously, most heavy bandwidth usage has been with file transfers. If any network through which the file is being transfered, suffers congestion or other problems, the transfer is slowed down and takes longer. While the end user has to wait longer for the data to be sent or received, the end result is still as expected.

However, with applications such as video conferencing, the performance is likely to be degraded by problems on the network. The user of such applications expects to see and hear information in a meaningful form. This makes it important to get data sent across the Internet in a quick reliable manner.

The Internet uses Internet Protocol[5] (IP) as the Network Layer protocol. This is represented at the third layer, in the Open Systems Interconnection (OSI) Seven Layer Model[7].

IP is a connectionless network service[2]. This makes IP an unreliable protocol, as data transfer cannot be guaranteed. Reliability may however, be introduced by a higher level protocol.

Many of the applications that require fast data transfer, use UDP[15] for their Transport Layer protocol[6], which gets encapsulated within IP. UDP is also a connectionless protocol, which does not provide any reliability. If the actual application also provides no guarantee that a packet is received, then if a packet were to be lost in transit, it will not be resent and its data never received.

Lost packets, commonly referred to as dropped packets, may lead to distorted video or sound as information is not received. Unwanted packet drop is just one form of fault on a network.

When IP packets travel over wide area networks, they are likely to traverse many routers. The number of routers between two hosts, can number well over a dozen. How all these routers are connected is dynamically arranged, and may change rapidly[3].

The route that a packet takes is the ordered list of routers that it traverses. This list can be produced using the utility *traceroute* (ftp://ftp.ee.lbl.gov/traceroute.tar.gz). As

connections between routers may alter quickly, the route a packet takes may be different from those packets previously sent. With each route likely to take a different amount of time, it is therefore conceivable that a packet sent along a different route to those following it, may arrive after later packets.

The result of this, is that packets may arrive at their destination in a different order to that which they were sent.

Packets travelling across the Internet may also suffer irregular delay and latency. These forms of network errors are tackled in a similar project produced by Neil Rogers *(atr0@aber.ac.uk)*.

To help compensate for these conditions, quality of service (QoS) is often implemented on routers[8]. However, to test such implementations, IP packets need to be generated in a controlled manner, that exhibit these kinds of irregular behaviours.

Applications may also attempt to cope with many of these network problems. To test how well they perform this, they need to receive data that may have been affected by these faults.

1.2 Objectives

The aim of this project is to produce a network router that functions in a controllable poor manner. A normal router would aim to forward all appropriate packets onto the correct network. However there are occasions when this does not happen as well as it should.

While the faults detailed above, usually occur when traffic passes over many routers, they can be replicated using just one router. The packet forwarding mechanism of a single router may be adjusted so that it may reproduce the faults described.

The effect of a medley of networks, that cause packets to arrive in the wrong order, or not at all, should be recreated in just one hop, with the faults being controlled! One router can then be used to test many kinds of networking systems.

The extent and limitations of any faults reproduced, are specified by the administrator of the router, using a controlling interface.

Two types of faults may be specified, on the router:

1. Rate of Packet Drop
2. Rate and Delay of Packet Reordering

These faults may be specified with the transport layer protocol and packet size range for which they should be applied.

The original project outline is located in Appendix B.

Chapter 2

Relevant Literature and Similar Existing Implementation

2.1 Raddle Network Emulation

Raddle is mainly concerned with testing network management systems, by emulating SNMP agents. It does not appear to be capable of generating faults in the network, like those tackled in this project.

URL:
http://raddle.sourceforge.net/

2.2 Honeyd

Honeyd creates a whole virtual network, with many virtual IP addresses. TCP and UDP connections can be made to the virtual hosts.

"... it is possible to use Honeyd for simple network simulations: Physical hosts can be exposed to high latency or packet loss, arbitrary routing infrastructures, etc."

It does however not cover packet reordering, which constitutes a significant proportion of this project. A virtual network is not required, as the faulty router is required to be used between at least two real networks.

URL:
http://www.citi.umich.edu/u/provos/honeyd/

2.3 NIST Net

Nist Net is an extension to Linux, with a controllable graphical user interface, that allows network conditions to be emulated on a router. Emulated conditions include packet delay, congestion, bandwidth limitation and packet reordering.

This application provides its own framework, and operates outside of kernel space. It does not use existing kernel network routing modules, such as netfilter.

It provides a large range of functionality, that tries to encapsulate all the different types of faulty conditions that may exist on a router. It does not specialise in one or two specific areas, such as packet dropping or packet reordering, although these are featured.

Packet reordering is performed by varying the delay on different packets, and packet drop performed using 'uniform probability'.

For this project, a different approach is taken to implementing the faulty capabilities, discussed in section 4.3 and detailed in chapter 7. This includes the physical reordering of packets, using the queue provided by Netfilter modules of Linux.

URL:
http://dns.antd.nist.gov/nistnet/

Acknowledgement:
Neil Rogers (atr0@aber.ac.uk) for providing the link.

2.4 Relevant Literature

Further understanding of routing has been conducted. TCP/IP Protocol Suite (Behrouz Forouzan, 2003)[1] contains several chapters on routing and detailed information about IP packet structure.

Significant use is made of the Netfilter component, within the Linux kernel. Details of modifying the operation of the netfilter are made available in the Netfilter Hacking Howto[10].

As with most development on a Unix or GNU/Linux system, extensive use is made of the man pages (system tools and function manuals).

Chapter 3

Problem Analysis

3.1 Problem Description

As outlined in section 1.1, there is a need for a tool that can be used to reproduce the behaviour of faulty networks. A tool that can be used to test the performance of networking applications and configurations, in a controlled faulty environment.

Of the many ways that a large wide area network can work in a disorderly manner, only two possible faults fall into the scope of this project. These are issues of:

1. Packet Drop
2. Packet Reorder

These usually only occur when data is being sent across very large wide area networks, such as the Internet. If these faults can be replicated in a closed environment, it would allow networking applications to be tested as if they were running across the Internet.

3.2 Requirements

To reproduce such faulty behaviour, in a closed environment, a single router needs to be used that can create such an effect. No such routers are built to deliberately drop and reorder IP packets.

A standard router needs to be modified, to allow for these actions to take place. All modifications to the routing mechanism must cause a minimal system overhead, as such a router needs to forward a large amount of traffic.

The administrator of the router, must be able to specify faulty qualities. Faults may be specified to apply to a range of IP packet sizes and the transport layer protocol of a packet.

Faulty qualities need to be arranged in an ordered list, which allows faults to be acted upon in a certain order, for each passing packet. Therefore, as well as adding to and deleting from the list of faults, the user must also be able to insert a fault into a specific location in the list.

When a deliberate fault in the forwarding of packets occurs, it should happen at a random point. How frequent the random occurrence of a particular fault is, should be

dependent on the probability that the fault may occur. The chance of an event occurring is independent of any event that has previously occurred.

3.2.1 User Interface

At the very least, the administrator of the router must be able to control all faulty qualities using a command line program. A program similar to *iptables*, that uses command line arguments and runs non-interactively, would be suitable.

Additionally, a graphical user interface is required. The administrator should be able to add, insert, update and delete faults that are to be exhibited by the router when forwarding packets. This interface should display all faults that are currently being reproduced by the router.

3.2.2 Fault Specifying

Faults that are to be generated are to be specified by the type of fault (packet dropping or reordering), transport layer protocol and the packet size range. A fault may be applied for one or all transport layer protocols, and for any size range.

Faults should be able to cover the same protocol and similar packet size ranges as other faults specified.

3.2.3 Packet Dropping

A packet may be dropped for a specified fault, if the packet's transport layer protocol and length match that of a fault action being applied and that a randomly generated value is within the bounds of the probability.

Packets will be analysed individually, by each fault in turn until it is dropped. If the packet does not get dropped by any of the specified faults, it gets forwarded, as normal.

3.2.4 Packet Reordering

Packet reordering effects one or more packets, and offsets them by another amount of packets. The administrator of the router needs to specify the maximum number of packets to take out of order (max burst size) and the maximum number of packets to offset the burst by (max delay size).

The list of faults that are being performed by the router, get examined in turn for each packet that tries to pass. If the transport layer protocol, packet length and probability restrictions are met, the packet is to be stored. A random integer value between 1 and the maximum burst size is chosen, and this many following packets are also stored and not forwarded.

A random integer between 1 and the maximum delay size, should also have been generated. The group of stored packets are held on the router until further packets have been forwarded, the number of which is specified by the random delay size. Then the stored packets are forwarded on, behind those packets that should have proceeded them.

The administrator should be able to skew the choice of burst size and delay size, so that numbers closer to the maximum are more frequent, or smaller values more frequent.

3.2.5 Shutdown

The router administrator needs to be able to shutdown all operations that cause faulty operation. The router should be able to return to a state of correct functioning, without requiring a reboot.

When the graphical use interface is closed down, the router should return to normal operation, and any stored packets released.

Chapter 4

Background Study

The purpose of this study is to present possible solutions to the problem of how to construct a controllable faulty router.

4.1 Intended Users

This software is intended to be used for the testing of networks and networking applications. Therefore it should be expected that users should have a reasonable degree of proficiency with networking and routing issues.

4.2 Underlying Software

The faulty network router will not be constructed 'from scratch' as this would be far too great a task. Instead the project will build upon work that is already available. Much, if not all, of this software is open source.

4.2.1 Linux

The open source kernel Linux, will be used and may be modified.

The latest stable version of Linux at the start of the project, was version 2.4.22, released on 25^{th} August 2003. While version 2.6 was very close to becoming stable, it would be likely to undergo many updates and patches over the following months.

Configuration Requirements: To form an IP router, a kernel must be compiled with the following options (additional to the modules required for a working system):

- CONFIG_PACKET
- CONFIG_INET
- CONFIG_IP_ADVANCED_ROUTER
- CONFIG_NETFILTER
- CONFIG_IP_NF_QUEUE

- CONFIG_IP_NF_FILTER
- CONFIG_IP_MULTIPLE_TABLES
- CONFIG_IP_ROUTE_TOS
- CONFIG_IP_MROUTE (routing multicast traffic)
- CONFIG_IP_ROUTE_VERBOSE (debugging information)

To enable routing (as root):
echo 1 > /proc/sys/net/ipv4/ip_forward

4.2.2 GNU/Linux System

A GNU/Linux distribution is required, that will have the modified Linux kernel at its core. Almost any full distribution would be capable of running the faulty network router. Distributions include:
RedHat Linux, Debian GNU/Linux and Slackware Linux.

4.2.3 Additional Kernel Modules

Linux QoS layer 7 packet filter To be able to drop and reorder packets depending on the application layer protocol.

URL: *http://sourceforge.net/projects/l7-filter*
Description: "We provide modifications to the Linux kernel which allow it to classify packets based on patterns in application layer data. Therefore, we can classify P2P traffic that uses random ports as well as standard protocols running on non-standard ports."
Latest Release: October 2003
Development Status: Beta

While providing faulty abilities to be specified at the application layer, additionally to the transport layer, the extra complexity may place the inclusion of this beyond the time constraints of the project.

4.3 Kernel Space and User Space

There are two feasible mechanisms for implementing additional functionality into the core of a GNU/Linux system:

1. User Space Implementation
2. Kernel Space Implementation

Using a user space implementation, the new application communicates with the kernel using a pre-existing system interface provided by the kernel. However for the task required, after packets enter the kernel, they will be passed out of the kernel to user space for processing by the faulty router software, then passed back to the kernel, where the kernel would forward the packets. This whole procedure is likely to place a significant overhead on this type of implementation.

With a kernel based solution, a new system interface can be constructed, specifically for the required task. Being within the kernel, this implementation will run faster than that in user space.

4.3.1 User Space Implementation

The Netfilter (http://www.netfilter.org) library 'libipq' allows for packet handling to be conducted within user space, instead of within the kernel.

While this solution will most likely be easier to implement, it is unlikely to run as efficiently as a kernel level implementation, for the reasons already stated.

4.3.2 Kernel Space Implementation

The Netfilter modules within the Linux kernel provide a means to register a 'netfilter hook'. A 'hook' is passing a function as a parameter, to the netfilter module, that is to be called when a matching packet arrives.

A similar mechanism works for the module *CONFIG_IP_NF_QUEUE*, where additional modules can be written to handle the queue of packets.

Being able to manage the queue of packets should allow:

- the queue to be re-arranged (reordering of packets)
- removal of elements from the queue (dropping of packets)

(see [10])

4.4 User Space Application

Assuming that the faulty routing capabilities are implemented using kernel modules, there will need to be a user space system program to interact with the new modules.

To communicate with the new netfilter Linux kernel modules from user space, Unix sockets are used. See: *getsockopt*[14] and *setsockopt*[14].

4.5 User Interface

Standard netfilter operations can be controlled directly from the shell, usually using a shell script. The additional functionality should also be controllable directly from the shell, as this allows a flexible interface.

Shell scripts are not appropriate to every user, an interactive user interface should be provided. This interactive interface should itself interact with only the shell. The result should be a modular design, where a different interface can easily be applied to the same underlying system.

An interactive interface could be run from within a users terminal window, using the ncurses libraries. This option has the particular advantage that it does not require X Windows to be installed on the router, as it can be run directly from the console. The ncurses libraries also use less system resources than that of a graphical application.

If X Windows is installed on the router, and the router has enough computing power, then a graphical interface is a possibility. Such an interface could be written

using the Gimp Toolkit Libraries (GTK) (http://www.gtk.org). The major versions of GTK running on GNU/Linux distributions vary. This may require the use of an older library version, which is common to almost all software distributions.

4.5.1 Relevant Sources

Ncurses URL: http://www.gnu.org/software/ncurses/
Version: 5.2

Gimp Toolkit (GTK) URL: http://www.gtk.org/
Version: 2.2
NB: version 1.2 is still in common use

4.6 Implementation Language

The use of C as the predominant programming language seems most appropriate. This is due to all the modules and system interfaces that are to be used, are written in C.

Higher level languages, such as Java, are not feasible when writing software at the kernel level, or just above.

The options are wider for writing the user interface. All that is required is a language and libraries that can produce an interface and issue commands to the shell.

4.7 Testing Applications

The following are applications that can be used to test the implementation of a faulty network router:

Network Traffic Generator URL: *http://sourceforge.net/projects/traffic/*
Description: "This project generates TCP/UDP traffic from client(s) to server(s) to stress test routers/firewalls under heavy network load."
Latest Release: December 2002
Development Status: Stable

IP Packets Generator URL: *http://sourceforge.net/projects/ipgen/*
Description: "IP packet generator, use raw socket to generate various IP packets, including ICMP,TCP,UDP or user defined with fake source IP. Use for network testing. (I wrote this program to test the router)"
Latest Release: March 2001
Development Status: Alpha/Beta

IPCAD - IP accounting daemon URL: *http://sourceforge.net/projects/ipcad/*
Description: "This daemon counts traffic (packets and bytes) on the specified interfaces. It has the built-in rsh engine to allow collecting the accounting data as from the Cisco router. Supports many other rsh-driven functions." Latest Release: September 2003
Development Status: Mature

Chapter 5

Project Methodology

The project methodology followed, includes:

1. Background Study
2. Design
3. Prototyping
4. Implementation
5. Debugging & Testing

Documentation is produced throughout the course of the project. Design and prototyping are tasks that overlap, with results from the prototyping being used in the design.

Additionally, time has been allocated to the production of two posters, as required by the project guidelines.

5.1 Time Plan

The time allocated to all the tasks of this project is 300 hours. Figure 5.1 displays the hours allocated to each task, on a weekly basis. Totals are displayed for the hours scheduled for each week and each task.

5.2 Background Study

The study of the specific subject area, which involves researching possible methods and solutions. Any existing implementations should also be mentioned.

This project is building upon a larger project, namely the Linux Kernel[9]. The study should involve a good analysis of the kernel and how to write additional modules.

The study should also include details of configuring a GNU/Linux system as a network router. If necessary, this will include kernel configuration information.

Hours spent

	Week																						
Month	*10*		*11*				*12*				*2*				*3*					*4*				
Day	*20*	*27*	*3*	*10*	*17*	*24*	*1*	*8*	*15*		*2*	*9*	*16*	*23*	*1*	*8*	*15*	*22*	*29*	*5*	*12*	*19*	*26*	
TASK	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**Total**
Background Study	15																							*15*
Design		10	5	2	0	5	5																	*27*
Prototyping			10	3	0	8	8																	*29*
Coding								15	15		15													*45*
Testing												15	8	8	2	0	2							*35*
Debugging													10	10	2	0	2							*24*
Documentation	4	8					5	5	5									15	15					*55*
Poster 1				10	10	5																		*25*
Poster 2															10	10	5							*25*
Meetings	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1		*20*
Week Total	***20***	***17***	***16***	***16***	***11***	***19***	***19***	***21***	***20***		***16***	***16***	***19***	***19***	***15***	***11***	***10***	***16***	***16***	***1***	***1***	***1***	***0***	***300***

Figure 5.1: Time Plan

5.3 Prototyping

Prototyping is undertaken to learn the viability of possible solutions. It is conducted in parallel with the design. Feedback from prototyping should influence the design.

Prototyping should lessen the risks involved, as the chosen method has been proven to be suitable during this phase. For this reason, prototyping should focus on areas of the project which are considered high risk. This is likely to be the kernel module, where the 'active' code that performs the faulty actions, is located.

5.4 Design

The design, to be completed after the background study, and in parallel with the prototyping, should outline the implementation of the project. The results of prototyping will influence design decisions.

The manner in which the router should drop and reorder packets is yet to be defined. The design should make this explicit, hence implying the parameters that should be passed from user space to the kernel.

5.4.1 Test Design

Once the project has been designed, a test plan and any test scripts or applications should be designed. This should occur before any implementation takes place, as this should help to avoid a test that is based on the implementation. The test should be based on the project design rather than the implementation, so as to test the implementation against the design.

The nature of the project is likely to require 'sniffing' network traffic, to test the correct working of the project. The volume of output produced by using a packet sniffer is best analysed using a scripting language, such as Perl. Such scripts should be written before the main implementation starts.

5.5 Implementation

Implementation should not start until the design is complete. This stage is heavily involved with testing, as each unit should be tested individually.

All source code should be placed under the version control system CVS[12]. This will aid the recovery of old versions of code, as and when it is required.

5.6 Debugging & Testing

While unit testing should happen during implementation, the main testing should occur during the debugging and testing phase. This includes validation tests and verification tests of the faulty nature.

The use of scripts to verify the desired functioning of the faulty router is likely. These should be specified as part of the design.

5.7 Tools & Technologies Used

5.7.1 Compilation and Programming Tools

All code is written in ANSI C, for a GNU/Linux system. Code is written using the *vi* text editor, although any text editor supporting UNIX files is acceptable.

Programs that are written in C, for this project, are built using a *makefile*, which uses *gcc* for compiling from source. When extending other projects, such as the Linux Kernel[9], *autoconf* is used.

For debugging programs, *gdb* is used. Programs must be compiled with the necessary options to allow a debugger to be used.

5.7.2 Documentation

All documentation is written using LaTeX[13], which allows for documents to be split over several files then combined together using one format. As the source of the documentation is stored in plain text, it may be edited in any text editor, eg. *vi*, and changes tracked using CVS.

5.7.3 Scripting

Perl is used to parse large output files of test data, and provide summary information (see section 7.5). One of the main strengths of Perl is its text processing and pattern matching abilities, that are useful for this task.

Bash scripts are used to manage the development environment; to dynamically create appendices and provide summary information of documentation, for example, the word count.

5.7.4 Version Control

CVS[12] is used to track changes in all source code, scripts and documentation. Provided are facilities to fall back to previous versions, if required, and store a history of every update.

The entire working environment is stored under CVS. A copy of the repository is available on the include CD.

Chapter 6

Prototyping

6.1 Prototyping

The purpose of this prototype is to establish the plausibility of a possible solution to the problem presented by the project. Only the high risk components of the suggested solution are prototyped, due to the limited time available. Once successfully prototyped, a solution can then be further expanded in the design.

It would be unwise to fully design a system when not knowing the success of an approach. Thus the use of prototyping for this project, as selected approaches require further exploration.

6.1.1 Prototype Implementation

The prototype is a small kernel module, and communicating user space program. Written to work in a very predictable manner, by dropping every x packets, and performing reordering every y packets.

The prototype grabs hold of the netfilter queue and a netfilter hook, for every packet being forwarded. Fault actions are triggered by the prototype communicating user space program.

6.1.2 Aspects Prototyped

The following aspects of the faulty router, have been prototyped:

- Packet drop, using netfilter 'hook'
- Packet reordering, using the netfilter queue:
  - Temporary storing of packets
  - Reinjection of stored packets
- Passing controlling data structure from user space to kernel module, using Unix sockets

6.2 Functionality Testing

The functionality test was performed using two systems (*kaled.network.home* and *test.cs390.network.home*) that were located either side of the faulty router. The network configuration used, is shown in appendix A.

NB: The hosts *'kaled.network.home'* and *'test.cs390.network.home'* have the respective IP addresses: 192.168.1.3 and 192.168.2.2, each on a network with a 24 bit subnet mask. These are private IP addresses, and use a private domain name that could not possibly exist on the Internet (network.home).

6.2.1 Functionality Test: Dropping Packets

For this test, the faulty router was set to drop every $5th$ ICMP packet that it forwards. The settings for packet reordering were made so that reordering does not occur during this test.

An ICMP Echo Request (ping) was sent from host *'kaled.network.home' (192.168.1.3)* to *'test.cs390.network.home' (192.168.2.2)* which is located beyond the router.

Prototype Results

Observer the **'icmp_seq=X'** in the output.

```
PING test.cs390.network.home (192.168.2.2): 56 data bytes
64 bytes from 192.168.2.2: icmp_seq=0 ttl=63 time=1.0 ms
64 bytes from 192.168.2.2: icmp_seq=1 ttl=63 time=0.9 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=63 time=1.0 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=63 time=1.1 ms
64 bytes from 192.168.2.2: icmp_seq=6 ttl=63 time=7.3 ms
64 bytes from 192.168.2.2: icmp_seq=7 ttl=63 time=6.7 ms
64 bytes from 192.168.2.2: icmp_seq=9 ttl=63 time=1.1 ms
64 bytes from 192.168.2.2: icmp_seq=10 ttl=63 time=1.0 ms
64 bytes from 192.168.2.2: icmp_seq=12 ttl=63 time=104.7 ms
64 bytes from 192.168.2.2: icmp_seq=13 ttl=63 time=1.1 ms
64 bytes from 192.168.2.2: icmp_seq=15 ttl=63 time=1.4 ms
```

Each line in the above results represents the successful transmission of two packets:

1. ICMP Echo Request
2. ICMP Echo Reply

As it is every 5^{th} packet that is being dropped, the first two pairs transmit successfully but the third request 'icmp_seq=2' is discarded. There is no reply sent for 'icmp_seq=2' as the request is never received by the other system.

Again, the next 2 pairs of request-reply appear correctly, but the third request is lost. This pattern will keep continuing until the parameters are changed, a host is shutdown, or a genuine network fault occurs.

6.2.2 Functionality Test: Reordering Groups of Packets

This test of the prototype was done by sending an ICMP Echo Request from a host to a computer on the other side of the faulty router. The other computer, from a network located beyond the faulty router, responds with an ICMP Echo Reply.

Test Parameters

The kernel module 'ip_faulty_queue.c' was set to only drop and reorder ICMP packets, for this test. The faulty qualities were set as follows:

Dropping Frequency:	every 12^{th} packet
Reordering Frequency:	start reordering at every 15^{th} packet
Delay block size:	delay 4 packets at a time
Reordering Delay:	delay block by 5 packets

Note: this test is concerned only with the reordering of a group of packets. It is not concerned with packet drop.

Running the Test

To assist with the identification of each Echo Request and Echo Reply pair, ICMP packets of a incrementing size are used for each request/reply pair. This is useful in identifying each pair, as all ICMP Echo Request packets have the '$id = 0$', as specified by the ICMP Protocol[16].

The ICMP Echo Requests (ping's) are generated using the following bash script, so that each is of a different size:

```
#!/bin/bash
a=100
while(test $a -le 120)
do
    ping -s $a -c 1 kaled
let a=$a+1
done
```

Test Output

The following output is an extract of what was generated using tcpdump running on both server and client systems:
'tcpdump -nvv icmp'

Both extracts show the same packets, no extract shows any packets that is not present in the other extract, unless it was dropped. Delayed packets are shown in a slighlty larger font size.

Packet Trace from host originating ICMP Echo Requests

19:23:35.546350 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 134)
19:23:35.547871 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55075, len 134)
19:23:35.557504 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 135)
19:23:45.566374 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 136)
19:23:55.576442 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 137)
19:24:05.586842 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 138)
19:24:15.596827 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 139)
19:24:15.598382 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55077, len 139)
19:24:15.608549 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 140)
19:24:15.609935 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55078, len 140)
19:24:15.619847 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 141)
19:24:25.762388 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 64, id 0, len 142)
19:24:25.763222 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55076, len 135)
19:24:25.765848 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55079, len 139)
19:24:25.766039 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55080, len 140)
19:24:25.766239 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55081, len 141)
19:24:25.766418 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55082, len 136)
19:24:25.766607 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55083, len 137)
19:24:25.766781 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55084, len 138)
19:24:25.766960 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 62, id 55085, len 142)

Four Echo Requests packets were sent, that were not immediatly responded to with a reply. These packets are identified by their lengths: $135, 136, 137, 138$ bytes.
The replies to these requests were received after several more ICMP packets had been successfully sent through the router. This suggests that those four packets, or the reply to those packets, were delayed during transmissions between the two hosts.

Packet Trace from host responding to ICMP Echo Requests with Echo Reply

19:23:31.678755 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 134)
19:23:31.678815 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55075, len 134)
19:23:31.688562 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 135)
19:23:31.688609 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55076, len 135)
19:24:11.728854 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 139)
19:24:11.729019 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55077, len 139)
19:24:11.738596 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 140)
19:24:11.738645 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55078, len 140)
19:24:21.888543 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 139)
19:24:21.888576 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 140)
19:24:21.888607 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 141)
19:24:21.888801 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 136)
19:24:21.888830 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 137)
19:24:21.888860 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 138)
19:24:21.888889 192.168.2.2 > 192.168.1.3: icmp: echo request (DF) (ttl 62, id 0, len 142)
19:24:21.889173 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55079, len 139)
19:24:21.889313 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55080, len 140)
19:24:21.889809 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55081, len 141)
19:24:21.889945 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55082, len 136)
19:24:21.890352 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55083, len 137)
19:24:21.890393 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55084, len 138)
19:24:21.890431 192.168.1.3 > 192.168.2.2: icmp: echo reply (ttl 64, id 55085, len 142)

Identified are the ICMP packets affected by the delay. The Echo Request packet of size 135 bytes is received without any unexpected delay. The reply is sent, also of 135 bytes, but this is not received by the originating host until later down the sequence, it has been delayed through the process of reordering.

The three echo request packets identified by their lengths of: $136, 137, 138$, are received several packets after they were expected. The replies to these are sent in the normal manner.

6.3 Prototype Reuse

No part of the prototype is reused in the final product. The prototype is experiemental code, written to increase the understanding of available mechanisms.

For this reason, the code is likely to require much reworking to be usable in the final product. Such reworking is likely to introduce bugs, especially as it has been originally written with no prior knownledge of the netfilter system.

Chapter 7

Design

7.1 Design

The implementation of the Faulty Network Router will be made up of three distinct components:

- Linux kernel module to drop and reorder packets
- User space program that communications with the Kernel module
- Graphical User Interface that calls the user space program

The separate user space program and graphical user interface allow more flexible usage of the final system. The user space program can be invoked directly from the shell. The graphical user interface is not a mandatory component, as full functionality is provided at the command line, as this is Unix convention.

This not only allows for a new interface to be written, but also enables the faulty router implementation to be used on systems that do not have a graphical facility. This is important, as there is no requirement that a GNU/Linux system set up as a router, should have a graphical environment installed.

7.1.1 Target System

The Faulty Network Router involves modifying the Linux source code, the kernel at the heart of a GNU/Linux system. Consequently the final product is limited to computers running GNU/Linux, with a patch applied to their kernel.

To be able to route packets between networks, the target system must be multi-homed (multiple network interfaces). The Faulty Network Router assumes that all routing tables are correctly configured, IP forwarding has been enabled, and that the router would normally operate in a fully functional manner.

Target Requirements

The target should be running:

- GNU/Linux Distribution

- Linux 2.4.22
- XFree86 (Required to use the GUI)
- GTK 1.2.10 (Required to use the GUI)

Linux 2.4.22 was released on August 25^{th} 2003. This was the latest stable version of Linux at the start of the project.

The Graphical User Interface (GUI) should be written using the Gimp Toolkit libraries. This implies that XFree86 should be installed and fully configured, as this is a requirement for running GTK applications.

7.1.2 Implementation Language

The nature of kernel development places restrictions on the choice of programming language. The language used must compile to machine object code, and not interpreted byte code.

Languages like Java compile to byte code, that is interpreted by the Java Virtual Machine. The Java Virtual Machine runs at a level above the kernel, and makes calls to kernel functions. The kernel cannot be written using Java, as this would require a Java Virtual Machine running beneath the kernel, when there is usually nothing below the kernel except for hardware.

The language used for writing Linux is C. This compiles to object code using *gcc* and built by *make*.

C will be used to write all components of the Faulty Network Router. The User Space program is required to use system function calls, to communicate with the new kernel module. The Graphical User Interface makes calls to the underlying shell.

While the Graphical User Interface could be implemented using a language such as Java, it remains consistent with the rest of the system if this is done using C. Making calls to the shell in C programs is less problematic than with Java, which isn't specifically designed for the purpose.

7.2 Kernel Module Design

The Netfilter component of Linux is a package of kernel modules that handle much of the advanced networking and routing tasks. The new module will extend the functionality already provided by the Netfilter package, to enable Linux to route packets to the required level of correctness.

The new module will be called: 'ip_faulty_router.c' and placed in the directory 'net/ipv4/netfilter' within the Linux source code.

7.2.1 Netfilter Functions:

Netfilter provides the following functions:

- nf_register_hook
- nf_register_queue_handler
- nf_register_sockopt

nf_register_hook

When registering a *hook*, a function reference is passed in as a parameter. This function gets called and passed the packet, for any matching packets.

Packets are matched by the Network Layer protocol and the class of packet. The class of packet refers to whether the origin or destination is the host machine, or if the packet is being forward. Packets being forwarded are packets that are being routed between networks. This parameter is specified as *'NF_IP_FORWARD'*.

When a packet is to be discarded, the function that was passed to nf_register_hook, to be called for every matching packet, should return *NF_DROP*. Otherwise the packet should be queued up by returning *NF_QUEUE*.

A hook should be registered for handling the packet drop requirement of the Faulty Network Router.

nf_register_queue_handler

A function reference is passed to this function call, which is invoked every time a packet approaches the queue. This function is passed every packet that is submitted to the queue.

They may have been submitted to the queue by a function that has been registered as a hook (see section 7.2.1) and returned *NF_QUEUE*.

This function does not return a value that is a verdict for the action to be taken upon the specific packet. Instead it must call *nf_reinject* for the processing of the packet to continue.

Alternatively, the packet may be held back and stored in a data structure within the module, while other packets pass by on the queue, then reinjected at a later stage. This is the mechanism that will be used for reordering packets.

nf_register_sockopt

This registers a function that will be called when the associated user space program (see section 7.3) makes a call to *setsockopt*. This mechanism is an implementation of the BSD Socket Interface.

7.2.2 Packet Dropping

The rate of packet drop should be expressed in terms of the probability that a packet gets dropped. This probability may be applied to a packet of a specified Transport Layer Protocol (such as ICMP, TCP, UDP), or to a packet of size (in bytes) within a given range.

The probability should be handled as an integer value between 0 and 1000. This should allow a good degree of accuracy when specifying the chance of a packet being dropped. For example, if the chance of a packet being dropped was 25.5%, then the value would be 255.

7.2.3 Packet Reordering

Similar to how Packet Dropping is specified, the probability that a packet or group of packets, gets reordered should be expressed as the chance in one thousand. This may be specified for packets of a certain Transport Layer Protocol or packet size.

Also specified with the probability should be the maximum burst rate. This is specified as the maximum number of consecutive packets to take out of the sequence order. When a packet is selected to be reordered, a value is chosen at random between 1 and the maximum burst rate, which will be the number of packets that will be taken out of sequence.

Once taken out of the normal sequence of packets, the reordered group of packets are inserted back into the sequence of packets, at a later point. The maximum period of delay is specified with the probability and maximum burst rate. The delay of the group is a randomly chosen value between 1 and the maximum period of delay.

Skewed Random Data

When a random number is chosen from an even distribution, each value in the range has an equal chance of being selected. This may not be appropriate to the intended behaviour of the faulty router.

For example, when specifying the maximum burst rate, the intention may be that most of the bursts of packets are close to the this maximum rate, and are not evenly spread.

To allow for this requirement, when specifying the maximum burst rate and the maximum period of delay, a skew may also be supplied as a percentage for each one. When a value is chosen between 1 and the appropriate maximum limit, the skew will be added or subtracted to the randomly chosen number, as a percentage of that maximum. The result must be positive and no more than the maximum.
$n_1 = n + S/100 * M$
$(0 < n_1 <= M)$

n is the randomly generated value, S is the skew, M is the maximum value.

7.2.4 Internal Structures

The rules governing the faulty routing capabilities may number several. Rules should be stored as a linked list of structs, that are traversed when deciding what action should be taken on a packet.

Such a linked list allows for an arbitrary number of rules to be specified. Each rule will have to store its current progress, as a rule may be executed over several packets. This is particularly appropriate to rules governing packet reordering.

Rules Structure

struct rules:

```
int               rule_type; /* packet drop or packet reorder */
int               proto; /* transport layer protocol */
int               min_size; /* minimum packet size for rule, default: 0 */
int               max_size; /* maximum packet size, default: MTU */
int               probability; /* chance in one thousand */
int               max_burst_size; /* packet reordering only */
int               max_delay_size; /* packet reordering only */
int               burst_skew; /* percentage skew */
int               delay_skew;
struct reorder    *reorder; /* any reordering currently going on */
struct rules      *next; /* next rule */
```

Pointer **reorder* will be NULL when there is no reordering in progress, associated with the rule.

struct reorder:

```
int                    chosen_burst_size; /* no of packets to delay */
int                    delayed_packets; /* no packets so far delayed, in this burst */
int                    chosen_delay_size; /* no of packets to delay by */
int                    current_delay; /* how many delayed by, so far */
struct stored_packet   *packets; /* delayed packets */
struct reorder         *next; /* allows for multiple reordering per rule */
```

This struct is used in two stages. Firstly the *chosen_burst_size* and *chosen_delay_size* are chosen randomly as defined in section 7.2.3, and any skew is applied according to section 7.2.3. The *delayed_packets* is initially set to 0, but is incremented every time a packet matched by the rule is added to the end of **packets*, until the *chosen_burst_size* and the *delayed_packets* are equal.

All the packets in this burst are now stored in the linked list of stored packets: **packets*. From now on, every packet that is being forwarded causes the *current_delay* to be incremented. Once the *chosen_delay_size* and the *current_delay* are equal, all stored packets are reinjected.

NB: Packets are only stored as part of the delayed burst, when they match the rules from which this struct is descended. Packets do not need to meet the rules when being counted to measure the delay (otherwise this could cause very long delay, if small size ranges are used).

struct stored_packet:

```
struct sk_buff         *delayed_skb;
struct nf_info         *delayed_info;
struct stored_packet   *next;
```

When a group of packets is taken out of sequence, as demanded by packet reordering, they should be stored in a linked list with their original order preserved. The linked list allows for an arbitrary number of packets to be taken out of order.

7.2.5 Kernel and User Space Communication

Communication between user space and kernel space is one way, with data being sent from user space to the new Kernel module. The mechanism for doing this is provided by an implementation of BSD sockets, using *setsockopt*.

Through the socket connection can be passed a pointer. This will be a pointer to a struct, containing all the appropriate parameters for faulty operation. This is the convention for communication between user space and the Netfilter package.

Inside the kernel module, a call to the function *nf_register_sockopt* should be used to setup the receiving end of the socket. The function is passed a struct containing socket configuration settings and a pointer to a function to be called when a client (the user space program) instantiates a connection.

More information can be seen on setsockopt by running: *man setsockopt*

7.3 User Space Program Design

The user space program should be a non-interactive program this is used by running with command line arguments. This may be run several times, with different values for the arguments.

Using the command line arguments, a data structure is built (see section 7.3.2). A pointer to this data structure is passed to the kernel module.

Each time the program is correctly run, a new rule is added in the kernel. Unless the program is being run to list all rules, or to remove one or all rules.

7.3.1 Synopsis

faultctrl -A drop|reorder [options] probability
faultctrl -U num drop|reorder [options] probability
faultctrl -I [num] drop|reorder [options] probability
faultctrl -D num
faultctrl -F
faultctrl -L

Commands

-A Append a rule to the current rules
-U Update a rule specified by the rule number
-D Delete a rule that matches the options
-I Insert a rule at given position in chain of rules, or by default at end
-F Flush all rules, clear all faulty routing
-L List all rules in operation

Options

-p protocol	Transport Layer Protocol Values: igmp, icmp, tcp, udp
-min size	Minimum size (in bytes) of packet that is to be delayed
-max size	Maximum size (in bytes) of packet that is to be delayed
-maxburst size	Maximum number of packet in a reordered burst
-maxdelay size	Maximum number of packets that burst is delayed by
-burstskew skew	Percentage of skew that should be applied to the burst
-delayskew skew	Percentage of skew that should be applied to the delay

7.3.2 Data Structures

The most important data structure for the communicating user space program, is the struct that is passed to the kernel module.

struct params:

```
int   rule_type; /* packet drop or packet reorder */
int   operation; /* append, insert, update, delete, flush, list */
int   proto; /* transport layer protocol */
int   rule_num; /* only used to insert, update and delete, else −1 */
int   min_size; /* min size of packet to match */
int   max_size; /* max size of packet to match */
int   max_burst_size; /* only required for packet reordering */
int   max_delay_size; /* only required for packet reordering */
int   burst_skew; /* the percentage skew */
int   delay_skew; /* the percentage skew */
int   probability; /* chance in 1000 */
```

7.4 Common Definitions

Definitions that are common to both the kernel module and the user space program.

7.4.1 Rule Types

DROP	1
REORDER	2

7.4.2 Operation Types

APPEND	0
INSERT	1
UPDATE	2
DELETE	3
FLUSH	4
LIST	5

7.4.3 Transport Layer Protocol

PROTO_ALL	-1
PROTO_ICMP	1
PROTO_IGMP	2
PROTO_TCP	6
PROTO_UDP	17

These values, except for 'PROTO_ALL', correspond to the protocol numbers used by IP to define the Transport Layer Protocol.[4]

7.5 Test Design

To test the ability of the router, raw IP packets should be analysed on two test systems, either side of the Faulty Network Router. The raw packets will be retrieved using *tcpdump* (http://www.tcpdump.org).

Tcpdump is a Unix utility that reads the network traffic that uses a specified network interface. It should be run as follows, to produce numerical output:
tcpdump -nvvv > output.txt

The output file containing packet headers, can then be analysed. A Perl script will undertake this.

Volumes of network traffic will be generated, and sent between the two tests systems, through the faulty router.

NB: All packet sniffing takes place on a private network, that is separated from the rest of the world by an NAT gateway. There is no chance that any traffic that has been read using tcpdump, belongs to anybody else.

7.5.1 Generating Test Traffic

Using *ipgen*, packets can be generated of the following transport layer protocols:

- TCP
- UDP
- ICMP
- user defined

The size and the number of packets may also be specified.

The program must run with root privileges, as it requires use of raw sockets. The utility was originally written for testing routers.

Enough data must be generated to smooth out the results, as the dropping and reordering of packets is random, within given parameters.

A wrapper shell script may be required to call ipgen to generated large quantities of packets of many different sizes and transport layer protocols.

7.5.2 Analysis of Test Traffic - Results of Faulty Routing

Test scripts will be used to parse the output that tcpdump produces. Perl will be used to write the scripts, as this is a language that has many text processing and pattern matching facilities.

The test scripts must first discard any headers that do not contain the IP addresses of both test machines. This is needed as tcpdump will read all packets on the network, including those between other hosts on the network.

The scripts are described here, by specifying the arguments that are supplied, and the results that they should produce.

Packet Drop Analysis

One script will compare the two output files, and record when packets appear in one file but not the other.

Synopsis

chkdrop.pl [-range min,max,min,max.....]
[-icmp [min,max,min,max.....]]
[-igmp [min,max,min,max.....]]
[-tcp [min,max,min,max.....]]
[-udp [min,max,min,max.....]]
-s source_ip_address
-d destination_ip_address
sender_packet_headers.txt
receive_packet_headers.txt

The *min* and *max* may be specified in multiple pairs, and are used to specify size ranges for which statistic should be gathered.

The following results will be displayed (depending on the supplied arguments):

Total Packets		
	Sent	x
	Received	y
	Success	$r\%$
< for each size range >		
	Sent	x
	Received	y
	Success	$r\%$
Total IGMP Packets		
	Sent	x
	Received	y
	Success	$r\%$
< for each size range >		
	Sent	x
	Received	y
	Success	$r\%$

Total ICMP Packets		
	Sent	x
	Received	y
	Success	$r\%$
< for each size range >		
	Sent	x
	Received	y
	Success	$r\%$
Total TCP Packets		
	Sent	x
	Received	y
	Success	$r\%$
< for each size range >		
	Sent	x
	Received	y
	Success	$r\%$
Total UDP Packets		
	Sent	x
	Received	y
	Success	$r\%$
< for each size range >		
	Sent	x
	Received	y
	Success	$r\%$

Packet Reordering Analysis

This script will compare the sequences of the two output files. It should then form averages to quantify how the reordering has been done.

Synopsis

chkorder.pl [-range min,max,min,max.....]
[-icmp [min,max,min,max.....]]
[-igmp [min,max,min,max.....]]
[-tcp [min,max,min,max.....]]
[-udp [min,max,min,max.....]]
-s source_ip_address
-d destination_ip_address
sender_packet_headers.txt
receive_packet_headers.txt

Results, depending on the selected arguments used:

Total Packets		
	Average reorder interval	%
	Average burst	%
	Average delay	%
< for each size range >		
	Average reorder interval	%
	Average burst	%
	Average delay	%
Total IGMP Packets		
	Average reorder interval	%
	Average burst	%
	Average delay	%
< for each size range >		
	Average reorder interval	%
	Average burst	%
	Average delay	%
Total ICMP Packets		
	Average reorder interval	%
	Average burst	%
	Average delay	%
< for each size range >		
	Average reorder interval	%
	Average burst	%
	Average delay	%

Total TCP Packets		
	Average reorder interval	%
	Average burst	%
	Average delay	%
< for each size range >		
	Average reorder interval	%
	Average burst	%
	Average delay	%
Total UDP Packets		
	Average reorder interval	%
	Average burst	%
	Average delay	%
< for each size range >		
	Average reorder interval	%
	Average burst	%
	Average delay	%

Chapter 8

Implementation

Implementation was carried out according to the scheduled detailed in Figure 5.1.

8.1 Kernel Module Implementation

One kernel module was written, called 'ip_faulty_router.c' (see appendix D). Placed into the directory 'net/ipv4/netfilter' within the kernel source. In this directory the files 'Config.in' and 'Makefile' were modified.

The header file 'ip_faulty_router.h' (see appendix C) was placed into the directory 'include/linux' within the kernel source.

All modifications are specified in the kernel diff, which can be generated with the GNU 'diff' utility. This can then be applied as a patch to the source for Linux version 2.4.22.

The kernel module implementation is centred on the three structs specified in section 7.2.4. Almost all the functions in the module operate on the linked lists formed using these structures.

8.1.1 Implementation Process

The module was written in one continuous eight hour period. Subsequent bug fixes were made to the code over the following months.

The design was not modified during or after, the implementation.

8.1.2 Debugging

When a normal program run on a Unix or GNU/Linux system, encounters a fatal error, such as attempting to access memory not allocated to it, or accessing a variable that is a NULL pointer, the program crashes and depending on the system environment settings (ulimit), information about the error can be recovered, from the dumped core.

However when such a bug occurs in a kernel module, it also crashes but because the entire system is dependent on the kernel, the computer also crashes. This complicates the issue of debugging a kernel module.

There were few fatal bugs found in the new kernel module. These were found by commenting and uncommenting certain sections of code.

To aid the debugging process, after the module had been written, the code was manually checked by hand. This uncovered a few errors and a some memory management issues.

8.2 User Space Program Implementation

The user space program, called 'faultctrl' (see appendix E) specified in section 7.3, is a simple program that forms a data structure (see section 7.3.2) from supplied command line arguments.

The program is run from the command line, with given arguments. There is no other interaction with the program.

Validation checking is performed by the user space program, this is in addition to the validation of the supplied rules by the kernel module.

8.2.1 Implementation Process

The program was written in a continuous four hour period. Limited testing was performed, and only simple debugging required. No changes were made to the design during or after implementation.

8.3 Graphical User Interface Implementation

The graphical user interface 'gfaultctrl' (see figure 8.1), allows a root user to specify the properties of rules through a graphical interface. They are able to specify everything in the GUI, that they would have otherwise done through the command line user space program.

When the user performs an action, the parameters that have been specified in the GUI are mapped onto arguments for the program 'faultctrl'. 'faultctrl' is then called from the GUI.

Tool-tips are provided over the buttons. This is especially useful for the buttons marked: 'I', 'U', 'X' which represent the actions: Insert, Update, Delete that should be performed on the rule that they correspond to.

Upon exit, the application calls 'faultctrl -F' to remove all rules of faulty routing from the kernel.

8.3.1 Implementation Process

The GTK API documentation[18] was used to assist in the development of the interface. Due to unfamiliarity with GTK, this was written in around thirty hours, spread over six days. Extensive use was made of gdb for debugging.

8.4 Test Script Implementation

The test scripts are written in Perl (http://www.perl.org) and parse large text files that have been generated by tcpdump. This tcpdump output is in two files, one for each host either side of the faulty router.

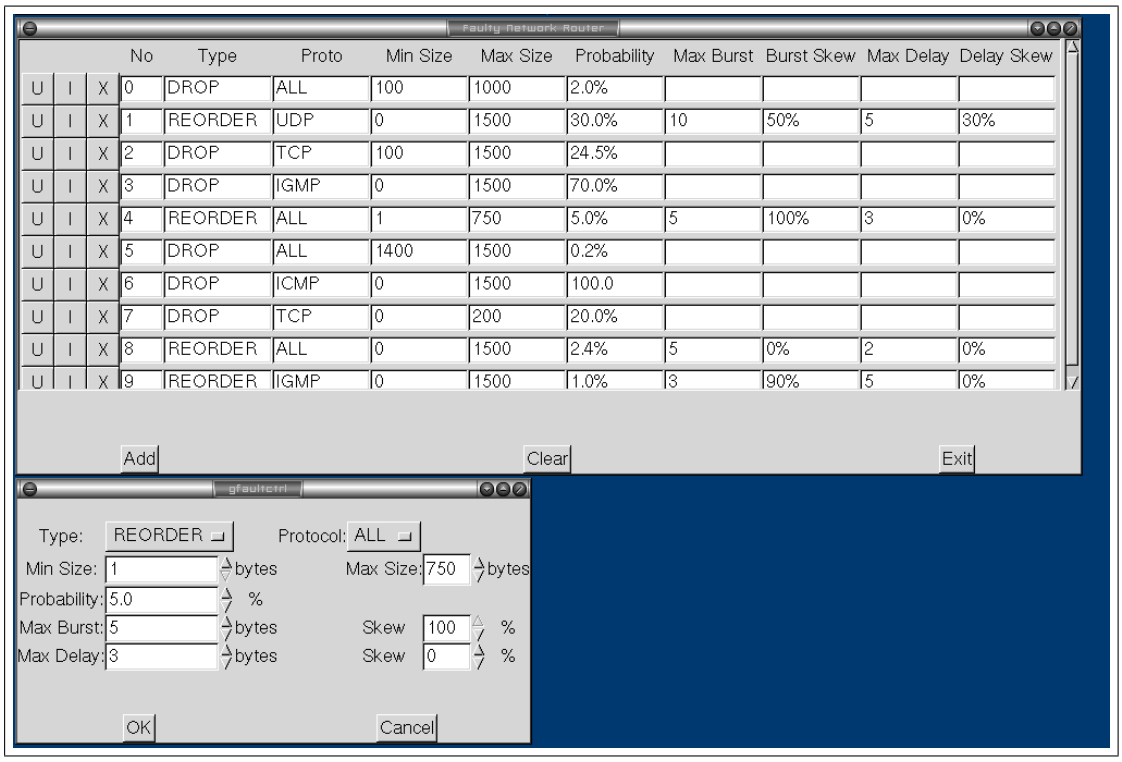


Figure 8.1: gfaultctrl

The perl scripts have been written to produce output summary data in the form specified by section 7.5.2.

The script chkdrop.pl is listed in Appendix F. This script chkorder.pl is listed in Appendix G.

8.4.1 Testing of Test Scripts

To test the test scripts, sample data files have been collected, using the prototype. The prototype works in a very predictable manner, there is no random manner to its operation, unlike the main product.

The data produced by the prototypes has been manually checked for correctness.

8.4.2 IP Packets

IP packets are generated using ipgen (http://sourceforge.net/projects/ipgen), being called by a bash script. Using arguments given to the bash script 'ip_blast.sh', ICMP, IGMP, TCP and UDP packets of incrementing sizes, between a given range, are sent across the faulty router.

TCP and UDP packets are directed at port 9 on the remote host, that is the other side of the router. Port 9 is the discard service, any packets sent to this port will be ignored by the receiving host.

8.4.3 chkdrop.pl

This script compares the two tcpdump output files. After removing packets that are not travelling between the two concerned hosts, it checks for packets that appear in one file but not the other. This is evidence that a packet has been discarded by the faulty router.

8.4.4 chkorder.pl

This script compares the two tcpdump output files. After removing packets that are not travelling between the two concerned hosts, it matches up the order of the packet ID numbers.

Each time that it finds packets that do not follow correctly, by checking the packet ID number, it records how late down the sequence they arrived and how many other packets have been taken out of order with it. It also records the gap between these reordering events. Once processed, the script produces summary data.

This script also allows for packets being dropped, when checking the packet sequence, and attempts to prevent dropped packets affecting the results of the reordering analysis.

Modification

The original script appeared to work correctly, except that it was considerably inefficient. The Test Plan (see section 9.2) involves the analysis of a few hundred thousand IP packet headers.

The core processing part of the script was rewritten, to be more efficient.

To test the new script, the old script was first run using the sample packet header data, and its output redirected to file. The new script was run using the same sample data and redirected to a different file. A diff was then taken of the two files that the output of the scripts were redirected to. Both implementations produced the same results.

The only difference being the time taken. The old script took 7.406 seconds to run. On the same small data sample, the new implementation took just 3.394 seconds.

High Packet Load Effect

Under high packet volume load, the router seems to unintentionally drop a small percentage of packets. This appears to be unavoidable, due to the necessary operations required for reordering packets, placing an increased overhead on packet handling.

The effect upon the reordering test script, is one of confusion. Where bursts and delays may be split into smaller portions, due to a packet being dropped during the burst.

This will lead to smaller bursts and delays being displayed than are actually being performed. The latest version of the script to check packet reordering, does not support gaps in bursts, but instead reports two or more bursts, when such an incident occurs.

Intentional Faulty Actions within Current Reordering

A current reordering process is defined as one that involves a group of packets being taken out of order, and then forwarded by the router after some more packets have been forwarded. During the execution of this reordering action, the module supports more reordering or packet drop, to occur within the set parameters.

It is therefore possible that while forwarding a delayed burst of packets, that some of these are dropped. Making a burst appear as two or more smaller bursts.

Alternatively, while waiting for packets to pass, before forwarding on a delayed burst, another burst maybe taken. In the tcpdump output, this would make the number of packets that the first burst waited for, before being forwarded (delay), appear less than it actually is.

Effects such as these are likely to make the results of this script appear incorrect for the supplied parameters to the faulty router kernel module. It seems very difficult and time consuming to write scripts that can allow for such difficulty in the analysis of the resultant tcpdump output.

Chapter 9

Testing

9.1 Testing Strategy

To test the faulty router, several rules are specified, while thousands of packets are sent accross the router using ipgen (see section 4.7). The rules that are to be set are defined in the test plans (see section 9.2).

The packet sniffer *tcpdump* (http://www.tcpdump.org/), will record the IP headers of all packets, on the networks either side of the router. The output is processed using the Perl test scripts defined in section 7.5, and summary data produced.

Expected summary data is calulated, from the set rules. For example, what percentage of TCP packets are expected to be dropped. The expected output can be compared to the actual output, after the test scripts have been executed.

The expected and actual outputs are unlikely to match exactly, due to the random nature of the faulty router. They should however, be of similar values.

9.2 Test Plan

The following rules will be entered using the Graphical User Interface (see figure 8.1):

Action	Protocol	Probability	Size	Burst (skew)	Delay (skew)
DROP	IGMP	1.5%	800-1000		
DROP	ICMP	0.3%	0-1500		
REORDER	TCP	0.2%	500-1000	10(75)	20(-10)
REORDER	UDP	5.0%	500-750	3(100)	5(0)
DROP	IGMP	100%	500-799		

Table 9.1: Test Plan Rules

Kernel Rules Listing

These are produced by running 'faultctrl -L', after setting up the rules in table 9.1. Output is displayed to the console, and should resemble that shown in the table 9.1.

Output from Kernel logs (/var/log/kern.log):

```
Mar 17 08:44:28 ferengi kernel: DROP -p IGMP  min: 800  max: 1000  Prob: 15
Mar 17 08:44:28 ferengi kernel: DROP -p ICMP  min: 0  max: 1500  Prob: 3
Mar 17 08:44:28 ferengi kernel: REORDER -p TCP  min: 500  max: 1000
Burst: 10 (75)  Delay: 20 (-10)  Prob: 2
Mar 17 08:44:28 ferengi kernel: REORDER -p UDP  min: 500  max: 750
Burst: 3 (100)  Delay: 5 (0)  Prob: 50
Mar 17 08:44:28 ferengi kernel: DROP -p IGMP  min: 500  max: 799  Prob: 1000
```

NB: Probability is expressed as the chance in 1000

9.2.1 Test Script Arguments

chkdrop.pl *-igmp 500,799,800,1000,1001,1500 -icmp*

chkorder.pl *-tcp 500,1000,1001,1500 -udp 0,499,500,750*

Packet Generation

IP packets of all 4 protcols, between the sizes of 400 and 1199 bytes, are produced. 100 packets for each size value.

$(1199 - 400) \times 4 \times 100 = 319,600$ packets, will be sent across the router.

This large volume of traffic, should lessen the effects of the random nature of faults happening.

Effect of High Packet Load

Preliminary tests of the test analysis scripts (see section 8.4), displayed unintended packet loss. This is possibly due to the increased load placed upon the router, while reordering packets.

The effect of this, is that unitended packet drop may effect the results. A burst may be recorded as two shorter bursts, due to one of its members being lost under the heavy load placed upon the router during these tests. From viewing the raw tcpdump output, this is clearly the case.

9.2.2 Expected Output

Packet drop and reordering is set to occur at random, within defined limits. Problems specified in section 8.4.4 are also likely to appear in the outputed data. Therefore the actual results may not mirror the expected results, even if the system is working correctly.

chkdrop

Protocol	Size range	Dropped
IGMP	500,799	100%
IGMP	800,1000	1.5%
IGMP	1001,1500	0%
ICMP		0.3%

chkorder

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,1000	10	8	500
TCP	1001,1500	0	0	0
UDP	0,499	0	0	0
UDP	500,750	3	2.5	20

9.2.3 Actual Output

Packet Drop Analysis

chkdrop.pl -igmp 500,799,800,1000,1001,1500 -icmp -s 192.168.2.2 -d 192.168.1.3

```
Total Packets:
        Packets Sent:   320002
        Packets Received:       289461
                Success:        90.4559971500178%
icmp:
        Packets Sent:   80002
        Packets Received:       79863
                Success:        99.8262543436414%
igmp:
        Packets Sent:   80000
        Packets Received:       49692
                Success:        62.115%
                Packet Range: 500 to 799
                        Sent:   29800
                        Received:       0
                        Success:        0%
                Packet Range: 800 to 1000
                        Sent:   19900
                        Received:       19593
                        Success:        98.4572864321608%
                Packet Range: 1001 to 1500
                        Sent:   19800
                        Received:       19800
                        Success:        100%
```

Results Analysis:

Expected:

Protocol	Size range	Dropped
IGMP	500,799	100%
IGMP	800,1000	1.5%
IGMP	1001,1500	0%
ICMP		0.3%

Actual:

Protocol	Size range	Dropped
IGMP	500,799	100%
IGMP	800,1000	1.543%
IGMP	1001,1500	0%
ICMP		0.174%

Packet Reorder Analysis

chkorder.pl -tcp 500,1000,1001,1500 -udp 0,499,500,750 -s 192.168.2.2 -d 192.168.1.3

```
Total Packets:
        Average reorder interval:       173.545212765957
        Average burst:  3.03102836879433
        Average delay:  2.39627659574468
udp:
        Average reorder interval:       101.09328358209
        Average burst:  2.69309701492537
        Average delay:  2.09608208955224
                Packet Range: 0 to 499
                        Average reorder interval:       0
                        Average burst:  0
                        Average delay:  0
                Packet Range: 500 to 750
                        Average reorder interval:       64.138418079096
                        Average burst:  2.69491525423729
                        Average delay:  2.09792843691149
tcp:
        Average reorder interval:       1560.48214285714
        Average burst:  9.5
        Average delay:  8.14285714285714
                Packet Range: 500 to 1000
                        Average reorder interval:       1560.48214285714
                        Average burst:  9.5
                        Average delay:  8.14285714285714
                Packet Range: 1001 to 1500
                        Average reorder interval:       0
                        Average burst:  0
                        Average delay:  0
```

Results Analysis:

Expected:

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,1000	10	8	500
TCP	1001,1500	0	0	0
UDP	0,499	0	0	0
UDP	500,750	3	2.5	20

Actual:

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,1000	9.50	8.14	1560.48
TCP	1001,1500	0	0	0
UDP	0,499	0	0	0
UDP	500,750	2.70	2.10	64.14

The actual results for average burst and delay are within the bounds of what may be considered acceptable, considering the random nature of events.

However, the average interval is significanlty larger than expected. This is suspected to be due to the IP packet generation script. It has been written to produce x packets of type TCP, then x of UDP, x of IGMP and x of ICMP, before incrementing the length of the packets that it generates.

If packet reordering is only occuring for TCP, then all the UDP, IGMP and ICMP packets are included in the test script analysis for calculating the average interval (chkorder.pl is included in appendix G).

For these tests, x is 100. Therefore after the last reordering in a block of TCP packets, there cannot be any reordering for at least another 300 packets, until more TCP packets are sent.

To test this hypothesis, a further test was carried out using only TCP data, in section 9.3.

This effect is one of the failings of the test script chkorder.pl (see appendix G). The other significant problem is the processing overhead. With the data produced from the test, the script took three and a half days to process on a Solaris server.

(Note: As Perl is a platform independent, portable scripting language, the scripts may be run on any computer, with Perl installed.)

Test Repeat

This test is not repeated due to the time taken in generating and processing the packet headers. The packets were forwarded across the router over a twelve hour period. The script to analyse the reordering took over three and a half days to run.

It is for this reason that this test has only been performed once.

9.3 Single Protocol Test

Action	Protocol	Probability	Size	Burst (skew)	Delay (skew)
REORDER	TCP	0.2%	500-700	10(75)	20(-10)
REORDER	TCP	2.5%	800-1000	5(0)	10(0)

Kernel Rules Listing

Output generated to kernel logs, after running 'faultctrl -L' to list rules:

```
Mar 22 09:54:22 ferengi kernel: REORDER -p TCP  min: 500  max: 700
Burst: 10 (75)  Delay: 20 (-10)  Prob: 2
Mar 22 09:54:22 ferengi kernel: REORDER -p TCP  min: 800  max: 1000
Burst: 5 (0)  Delay: 10 (0)  Prob: 25
```

NB: Probability is expressed as the chance in 1000

9.3.1 Test Script Arguments

chkorder.pl *-tcp 500,700,701,799,800,1000*

Packet Generation

IP packets of protcol TCP, between the sizes of 500 and 1000 bytes, will be produced. 100 packets for each size value.

$(1000 - 500) \times 100 = 50,000$ packets, will be sent across the router.

9.3.2 Expected Output

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,700	10	8	500
TCP	701,799	0	0	0
TCP	800,1000	2.5	5	40

9.3.3 Test 1: Actual Output

chkorder.pl -tcp 500,700,701,799,800,1000 -s 192.168.2.2 -d 192.168.1.3

```
Total Packets:
        Average reorder interval:       97.6107784431138
        Average burst:  2.35528942115768
        Average delay:  4.95209580838323
tcp:
        Average reorder interval:       97.6107784431138
        Average burst:  2.35528942115768
        Average delay:  4.95209580838323
                Packet Range: 500 to 700
                        Average reorder interval:       861.272727272727
                        Average burst:  9.63636363636364
```

```
                Average delay:  7.22727272727273
        Packet Range: 701 to 799
                Average reorder interval:       0
                Average burst:  0
                Average delay:  0
        Packet Range: 800 to 1000
                Average reorder interval:       40.8670886075949
                Average burst:  2.01898734177215
                Average delay:  4.84810126582278
```

9.3.4 Test 2: Actual Output

This is a repeat of Test 1, with the exact same parameters.
chkorder.pl -tcp 500,700,701,799,800,1000 -s 192.168.2.2 -d 192.168.1.3

```
Total Packets:
        Average reorder interval:       92.2846299810247
        Average burst:  2.35104364326376
        Average delay:  4.90891840607211
tcp:
        Average reorder interval:       92.2846299810247
        Average burst:  2.35104364326376
        Average delay:  4.90891840607211
                Packet Range: 500 to 700
                        Average reorder interval:       824.478260869565
                        Average burst:  9.69565217391304
                        Average delay:  6.73913043478261
                Packet Range: 701 to 799
                        Average reorder interval:       0
                        Average burst:  0
                        Average delay:  0
                Packet Range: 800 to 1000
                        Average reorder interval:       38.7364185110664
                        Average burst:  1.99798792756539
                        Average delay:  4.83299798792757
```

9.3.5 Results Analysis:

Expected:

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,700	10	8	500
TCP	701,799	0	0	0
TCP	800,1000	2.5	5	40

Test 1:

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,700	9.64	7.23	861
TCP	701,799	0	0	0
TCP	800,1000	2.02	4.85	40.87

Test 2:

Protocol	Size range	Average Burst	Average Delay	Average Interval
TCP	500,700	9.70	6.74	824
TCP	701,799	0	0	0
TCP	800,1000	2.00	4.83	38.74

All the results, except for the average interval for the first reordering rule, are similar to the expected outcomes.

An average delay of 824 packets, represents a reordering chance of 0.12%:
$1 \div 824 \times 100 = 0.12\%$

While this is a lower probability that was expected, it may be partly caused by the small size of the sample, or the random events that have occured.

9.4 Testing Summary

For these tests, over $400,000$ packets were sent across the faulty router, with different rules set. The rules have been tested, that they have been set correctly within the system, using the listing output from the kernel module.

As all reordering and packet drop is done on a random basis, within set parameters to reflect the probability of an event occuring, the results are always likely to be different for each run.

The time constraints of this project prevent conclusive, exhaustive testing being undertaken. However, it can be said that the system does appear to function as a faulty network router, that works based upon its supplied arguments.

Chapter 10

Distribution

10.1 Licensing

The Linux Kernel is licensed under the GNU General Public licence[19]. This requires that modifications to it are made freely available in source code form, under the same licence.

Therefore the faulty router kernel module must itself be licensed under the GNU General Public Licence and made freely available. The source will be made available on the web in the near future, using the resources provided on http://freshmeat.net.

10.2 Kernel Source Patch

The traditional way of releasing Linux kernel modifications, is to produce a 'diff' of the modified source, compared to the original source. This can then be used to patch a certain version of the Linux source code.

Releasing binary updates of Linux is not appropriate, as Linux is usually customised for its host machine. Even if binaries were appropriate, the source is still required to be released, due to licensing restrictions.

Location on CD:
The kernel source diff can be found in the directory 'src/kernel/'.

10.2.1 Production of Kernel Source Patch

To Produce a kernel source diff, using the version of Linux used for this project, Linux 2.4.22:
cd /usr/src
diff -rNu linux-2.4.22/ linux/ > faulty_router.diff

(where 'linux-2.4.22/' is an unmodified version of the source code, and 'linux/' is the modified source)

The file 'faulty_router.diff' contains all the differences between the modified and unmodified versions of Linux version 2.4.22. This is how kernel patches are released on ftp://ftp.kernel.org/, the home of the Linux kernel.

10.2.2 Application of Kernel Source Patch

A diff may be applied to its corresponding version of the Linux source code.
cd /usr/src
cat faulty_router.diff | patch -p0

Will patch the source code located in the directory 'linux/', which should contain the source code for Linux 2.4.22.

The kernel must be configured, and the faulty router module selected. If not already done so on a previous compilation, select the compilation options for the target system. Run any of the following, from the directory '/usr/src/linux':
make config
make menuconfig
make xconfig (requires X Windows)

The kernel module to provide faulty routing abilities, is located within the 'Netfilter Configuration' menu, that is found under 'Networking Options'. Select the module: 'Faulty Router Support' to be compiled in.

Compile the Kernel as usual; from the location '/usr/src/linux' run:
make dep
make bzImage

If using an i386 based system, the Kernel image is located in arch/i386/boot.

Copy the kernel image *(bzImage)* to the '/boot' directory of the target system, then reconfigure (and run if necessary) the boot-loader. If the compilation system and target system are different, the root device may need to be changed, refer to the man page for 'rdev' if this is necessary.

Reboot the computer!

10.3 Controlling Programs

There are two controlling programs available. The one that is required to be present, to be able to use the faulty router, is 'faultctrl' (see section 8.2).

For use of a graphical user interface, 'gfaultctrl' (see section 8.3) must be compiled. 'faultctrl' must still be available.

Both these programs include a makefile. Run 'make' to compile the applications.

When compiled, copy the files 'src/userspace/faultctrl' and 'src/gui/gfaultctrl' to a location specified in the PATH variable. A directory such as /usr/sbin is most appropriate, as only the root user may run these programs.

The root user may now run one of these programs, and control the faulty router. However, for the faulty router to be of use, the system must already be configured as a router (see section 4.2.1).

Location on CD:
src/userspace & src/gui

10.4 Portability

The faulty router kernel module and controlling programs will probably work on GNU/Linux systems running on other hardware architectures, such as Sparc, Alpha or PowerPC. No hardware depended code has been produced, only the implied hardware requirement of a multi-homed system, for the faulty router to be of value.

The faulty router kernel module builds upon the existing Netfilter framework. This framework is architecture independent.

However, the faulty router has only been tested on i386 (Intel) based architecture. Testing on other architectures has not been possible, due to hardware availability and time constraints.

10.5 Security Vulnerability

When running the router with faulty routing rules applied, the system may be vulnerable to denial of service attacks (DoS). If a huge volume of traffic is passed across the router, the processing of faulty actions and stored packets becomes great. Much of the available memory may be used as more and more packets become stored. This is likely to have an impact of the performance of the computer running the faulty router.

For these reasons, the faulty router should never be used on a public network. It is a tool designed to be used on closed networks, for testing purposes. It should be restricted to such uses.

10.6 Future Distribution Practice

A future extension to how this system is distributed, could be made using a 'Linux Live' compact disc. Such CD's contain a version of GNU/Linux that can be booted from, and run directly off the CD. No interaction or installation is required with the locally installed operating system.

Using a system such as Knoppix (http://knoppix.net/) or Damn Small Linux (http://damnsmalllinux.org/) which is a smaller version of Knoppix, the included kernel can be altered to include the faulty router module, and the controlling programs added to the CD.

This would allow the faulty router to be run on almost any PC, that has at least two network cards and a CD-ROM, by inserting the CD and rebooting the computer. The computer does not need to have GNU/Linux installed locally.

Creating a live CD places too great a demand on the time constraints of this project. Linux Live CD's, are a relatively new innovation and all tools to build the CD's are in the very early stages of development.

Chapter 11

Project Evaluation

11.1 Requirements

The requirements that are specified in section 3.2, have all been satisfied. While these requirements may still be considered to be correct, they fail to detail additional functionality which would have added extra value to the finished product.

The oversimplification of requirements, which specify just two vital functions of the software, has resulted in a project that has not shown its full potential. Possible extensions to the requirements are further discussed in section 11.7.

At the time of the production, of the requirements, it was judged to be beyond the time constraints of the project, to require additional functionality. The original requirements demanded much research and experimentation into the operations and modification of the netfilter framework within the Linux kernel.

While the above statement remains correct, it has become apparent that additional functionality could have been added without significant burden placed upon the time restrictions.

11.2 Design

The most important decision about the design of the product, was the choice made between a user space implementation and a kernel space implementation. This was discussed in section 4.3.

While a user space implementation would have likely to have been a simpler task to design and implement, a kernel space implementation was chosen. The main reasons for this, was the increased freedom gained from working within the kernel, and the lower processing overhead.

The overhead imposed by using a user space implementation, would have undoubtedly been greater. IP packets that are received by the kernel would be passed out of the kernel and into user space. They would then be processed in user space, by the faulty router program, and passed back into the kernel.

Whereas with the faulty router implemented as a kernel module, all processing of IP packets is undertaken within the kernel, with no need to pass packets into user space for processing. This in itself caused a processing overload, that appeared when testing the router under heavy load (see section 8.4.4). It was found that a very small proportion of

IP packets were lost, possibly due to the kernel executing its faulty abilities or processing stored packets.

If a user space implementation was chosen, the processing overhead would be more significant, and its effect on accomplishing the required task would have been harder to meet.

Therefore in hindsight, the decision to implement the faulty network router as a kernel module, was correct!

The design focused primarily on the data structures within the kernel module (see section 7.2.4). It was not immediately concerned with specifying the individual functions, their parameters and return types.

The data structures are 3 linked lists, specified in the kernel module header file (see appendix C). Linked lists allow for an undefined number of elements, which places no restriction on the number of rules, reordering actions or the number of packets that can be stored.

This design has caused no problems, at any stage of the development.

Specified in section 7.2.5 is the mechanism for allowing control of the kernel module, from a user space program. The decision was taken to pass a data structure into the kernel, through a Unix socket.

The alternative option, that taken by a similar project by Neil Rogers, is to use the 'proc' filesystem. This allows for a controlling program to write to files within the proc filesystem, that can then be read by a kernel module. Normally integer or boolean values are written to these files.

It was considered that being able to pass a data structure, containing many faulty parameters, was a more extensible method of communication. For this reason, the operation of passing a pointer to a data structure, through a Unix socket, was chosen.

Passing structures directly to the kernel module, using Unix sockets, has been a very effective way of performing the communication between a user space controlling program and the faulty router kernel module.

11.2.1 Effect of Prototyping

Heavy use of prototyping was made, before the design was finalised. This has proved crucial in getting the design correct.

Without prototyping, a viable method for performing the reordering and dropping of packets would have been unknown, at time of design. Also prototyped, was the communication mechanism between the faulty router module and the controlling user space program.

This prototyping proved that the chosen mechanisms operated to produce the intended result. It also served to demonstrate how extensible these mechanisms were, to allow for a full scale product to be built.

Prototyping has proved essential to the success of this project, without which a correct design would have been unlikely. The major high risk design issues were investigated and tested, before being finalised.

11.3 Implementation

The tools used for developing and documenting this project (see section 5.7), are typical of a standard Unix or GNU/Linux development environment. A Unix or GNU/Linux system provides all the development tools that are needed, for this project.

Many of the tools used for development, are already used for developing the Linux Kernel. Linux is written in C, compiled using GCC and built using autoconf and make. Changing these would have served little or no purpose, as they are all freely available.

Once the design had been finalised, the implementation process could proceed. This was done rapidly, but not rushed, so as to allow reasonable time for testing.

The kernel module and communicating user space program, were written without great difficulty. This is considered to be due to much time and effort being spent in the design and prototyping stages of development.

However, the graphical interface was not written with such ease. Much time was spent getting familiar with the GTK libraries, and GUI creation in a non-object oriented language.

Prior experience of GTK would have been invaluable at this stage, but even in hindsight, such a component should not have been prototyped. The GUI is not a core component, as the faulty router is fully functional without the GUI.

11.4 Usability and Intended Operation

As already stated in section 4.1, all users are expected to be competent and familiar with network and routing issues. The users of the product are likely to be either testing networks or networking applications. It is reasonable to consider that people who configure the tests, are skilled for such a task.

No testing of usability on the intended audience, has taken place. Such testing would require access to network development and testing engineers, with time available. This has not been possible.

The product does complete the intended operations, as originally required. However, value adding extensions could greatly improve the controllable nature of the faulty router (see section 11.7).

11.5 Project Achievement and Time Management

The completed product allows the user to specify IP packet drop and packet reordering, by setting faulty parameters. The parameters that may be supplied are:

- Transport Layer Protocol
- Packet Length Range
- Probability of event occurring (0.1% intervals)

The completed work satisfies the requirements (see section 3.2) and is in accordance with the original project outline that was submitted at the start of the project (see appendix B).

The source code for the project is extendible, to be able to include various extra options. This is planned to happen over the forthcoming months, with all source code being made freely available.

The project was undertaken in accordance, and to the time scales specified in the project methodology (see chapter 5). Most stages of development were completed to meet their appropriate milestone. However some stages of development fell behind their deadline, this was usually by being no more than a couple of days behind schedule.

The most significant missed milestone, was for the implementation phase. While the kernel module and communicating user space program, were written in a shorter than expected time, much time was spent working with GTK. It had not been taken into consideration, that using GTK would require much time to be spent learning how to use the libraries.

The slight slippage of certain stages of development, did not appear to hinder the project significantly, as a certain degree of flexibility was taken into consideration when planning the project. The project plan meant that completion could be reached in an orderly manner, even with the occasional slight delay. Time was left unallocated, towards the end of the project, to allow for task overrun.

11.6 Project Limitations and Deficiencies

Required was the ability to specify faulty abilities depending on the transport layer protocol. There is however, no feature to express faulty rules depending on the application layer protocol, eg. telnet, ftp or http.

Packet reordering is conducted using bursts of packets, so that whole groups of packets are affected. The same is not true for packet dropping, unless several successive packets are randomly dropped, within the set parameters. Whether a packet is dropped or not, is independent from the action taken on any previous packets. This may not be considered typical faulty behaviour, and may be considered to be a requirements or design oversight.

The probability that a packet may be dropped or reordered, can only be expressed as the chance in one thousand. This may not be high enough precision. However, there should be little work required in changing this to a higher value, even one too large to be an integer.

11.7 Possible Modifications and Extensions

Application layer protocols could be easily specified, at a basic level, for UDP and TCP transport layer protocols. This may be accomplished by analysing the port number to which a packet is being sent, or received from. This would incur a further limitation, for example if a faulty rule were to be set to affect http traffic, by specifying port 80 TCP (the default for http), any http network traffic not running on port 80 would be unaffected.

IP packet dropping could be specified in a similar fashion to packet reordering. It need not be as complicated as packet reordering as no packets are being stored, they are not expected to appear later, when they are set for drop.

The probability that a rule should be applied to a matching packet, could be given with far greater precision. It is not necessary to specify here what precision should be used, just that it can be easily changed to suit new requirements.

The above modifications can be applied to the existing software, with relative ease.

Extensions to the system, may include:

- Packet Duplication
- Bandwidth Limiting
- Variable Packet Latency

The last point, is dealt with by a similar project by Neil Rogers. His kernel module may be included within the faulty router produced by this project, at a later point.

These three possible extensions would all be specified according to transport layer protocol, application layer protocol (if applicable) and length range of the packets that they should affect.

Packet duplication and bandwidth limiting are likely to require much additional work to the software. Bandwidth limiting will need significant research and experimentation, to fully understand how this may be achieved. Packet duplication should be able to be performed using the netfilter functions: *skb_copy* and *nf_reinject*.

These extensions would have almost certainly taken the original requirements beyond the time limits placed upon this project. However they would result in a product of far greater value.

11.8 Proposed System Redevelopment

If the system were to be redeveloped, from the very start. Much of the methodology and design would be completed in the same manner. A kernel implementation would again be used, with a command line user space program to communicate with the kernel, and a graphical interface.

As outlined above, the user would be given more control of packet dropping, with the ability to drop blocks of sequential IP packets. Faulty rules would be able to be specified according to application layer protocol, in addition to the current parameters. Probability should be specified at least, as the chance in ten thousand (0.01% intervals).

This reflects that more time should have been spent analysing the requirements, for the project.

The development methodology (see chapter 5) used seems appropriate for this project, with perhaps slight modifications to allow more time for requirements analysis and implementation stages of development.

Bibliography

[1] Behrouz Forouzan (2003), *TCP/IP Protocol Suite,* McGraw-Hill 0-07-119962-4

[2] Behrouz Forouzan (2003), Delivery and Routing of IP Packets, *TCP/IP Protocol Suite,* McGraw-Hill 0-07-119962-4

[3] Behrouz Forouzan (2003), Routing Protocols (RIP, OSPF, BGP), *TCP/IP Protocol Suite,* McGraw-Hill 0-07-119962-4

[4] Behrouz Forouzan (2003), Internet Protocol (IP), Page 197, Table 8.4, *TCP/IP Protocol Suite,* McGraw-Hill 0-07-119962-4

[5] William Stallings (2004), Internet Protocol (18.4), *Data and Computer Communications,* Pearson Prentice Hall 0-13-183311-1

[6] William Stallings (2004), Transport Protocols (20), *Data and Computer Communications,* Pearson Prentice Hall 0-13-183311-1

[7] William Stallings (2004), Protocol Architecture: OSI (2.3), *Data and Computer Communications,* Pearson Prentice Hall 0-13-183311-1

[8] Alcatel Executive Brief, Quality of Service (QoS), February 2002. *www.ind.alcatel.com/library/e-briefing/eBrief_QoS.pdf*

[9] The Linux Kernel. http://www.kernel.org/

[10] Netfilter Hacking Howto, July 2002. http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-4.html

[11] Linux Kernel Programming. http://www.kernelhacking.org/

[12] CVSHome, CVS Online Manual. http://www.cvshome.org/

[13] LaTeX, A Document Preparation System. http://www.latex-project.org/

[14] Setsockopt, Linux Command, Unix Command, http://linux.about.com/library/cmd/blcmdl2_setsockopt.htm

[15] RFC 768 (28 August 1980), User Datagram Protocol. http://www.faqs.org/rfcs/rfc768.html

[16] RFC 792 (September 1981), Internet Control Message Protocol. http://www.faqs.org/rfcs/rfc792.html

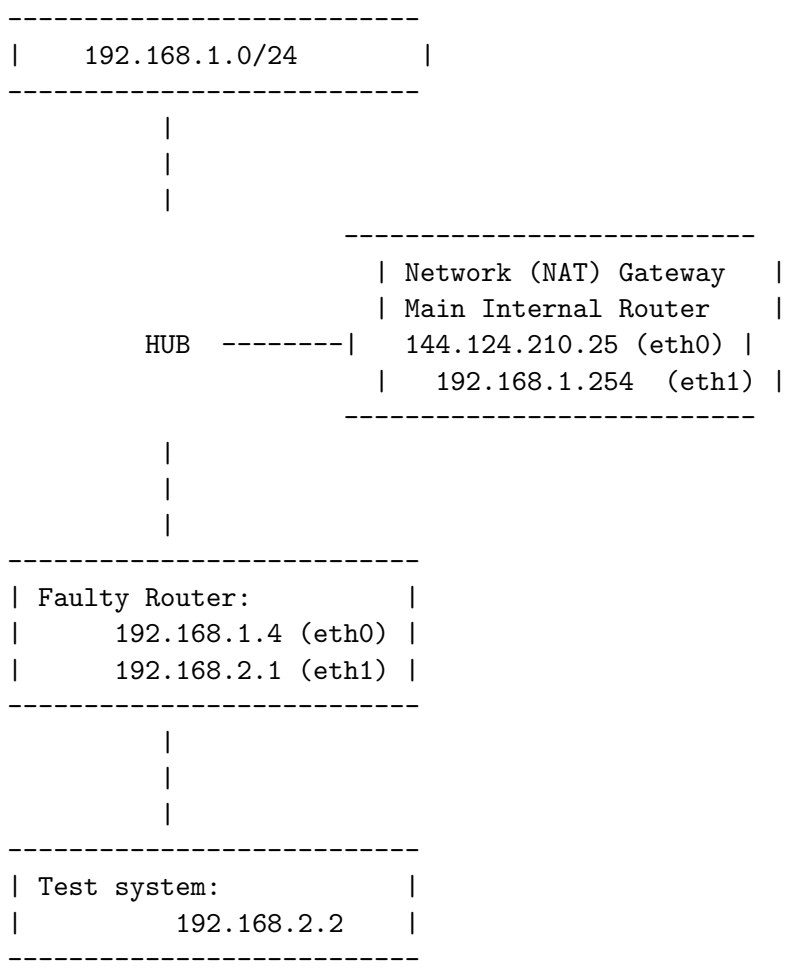
[17] RFC 793 (September 1981), Transmission Control Protocol. http://www.faqs.org/rfcs/rfc793.html

[18] GTK+ Reference Manual, http://developer.gnome.org/doc/API/gtk/index.html

[19] GNU GENERAL PUBLIC LICENSE, Version 2, June 1991. http://www.gnu.org/licenses/gpl.txt

Appendix A

Network Configuration



A.0.1 Server Configurations

Faulty Router
IP Addresses:
192.168.1.4 (eth0)
192.168.2.1 (eth1)

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.1.254   0.0.0.0         UG    0      0        0 eth0
```

Network Gateway (NAT) and Router
IP Addresses:
144.124.210.25 (eth0)
192.168.1.254 (eth1)

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.2.0     192.168.1.4     255.255.255.0   UG    0      0        0 eth1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
144.124.208.0   0.0.0.0         255.255.248.0   U     0      0        0 eth0
0.0.0.0         144.124.215.254 0.0.0.0         UG    0      0        0 eth0
```

A.0.2 Test Client Configuration

Test Client In Front of Faulty Router IP Addresses:
192.168.1.3 (eth0)

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.2.0     192.168.1.4     255.255.255.0   UG    0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.1.254   0.0.0.0         UG    0      0        0 eth0
```

Test Client Behind Faulty Router IP Addresses:
192.168.2.2 (eth0)

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0 eth0
```

Appendix B

Original Project Outline

Problem Description

Network applications working over the Internet, are often required to work under less than ideal networking conditions. These poor network conditions can affect the performance of applications such as video conferencing, where there is a need for continual data transfer.
When communicating applications are located far apart, often in separate countries, more than a dozen routers may be used to transfer data between the two hosts. It is easily feasible that not all this data is passed through the same routers, with some packets travelling by a different route. This may lead to packets arriving at their destination in a different order to how they were sent.
It is also conceivable that while travelling over the Internet, data packets become corrupted, and are then discarded by routers instead of being forwarded.
When testing networking applications, it is important to be able to test them over a network that behaves as badly as the Internet can. This behaviour cannot be reproduced in a closed environment using a standard network configuration.

Project Description

The aim of this project is to produce a network router that functions in a controllable poor manner. A normal router would aim to forward all appropriate packets onto the correct network. The faulty implementation will not aim to do routing as well, but instead will deliberately drop and reorder packets, in a specified manner.
Such a router would be used for the testing of real-time network applications, where poor network performance has the potential to drastically effect the quality of the communication.
There are several ways that networks can work in a faulty manner, these include: dropping packets, reordering packets, increased latency and delay variation (jitter). For the purpose of this project, just two of these possible faults will be involved:

1. Dropping Packets
2. Reordering Packets

How packets are dropped and reordered should be controllable. The manner in which this will occur may be set to correlate to the size of the packet or the rate at which packets arrive for routing.
Reordering may occur by having individual packets placed in the incorrect order, or by moving a whole series of packets to another location in the packet order. A packet may be swapped with the proceeding packet, so that the proceeding packet arrives first, or that a single or group of packets arrive several packets behind their correct position in the order.
Packet dropping will be the discarding of individual or groups of packets. The dropping of packets may also depend upon the size, type and upper layer protocol of the packet.

Sister project: Simulating Problematic Issues in IP Networks

It should be noted that another similar project is also being undertaken by Anthony Neil Rogers (atr0@aber.ac.uk). That project focuses on constructing a router that is faulty by increased latency, and delay variation.

Work to be Tackled

Modifying the way in which a system performs routing, means that the operating system of the router will most likely require modification at a low level. It would appear that a Linux based router would be a good choice, as the source code is freely available, with modification permitted.
It is likely that implementing the Faulty Network Router will require Linux kernel modules to be written. I have never written kernel modules before, or interacted directly with the kernel at such a low level. I foresee that this is an area that will require substantial amounts of research and study.
Additionally to any new kernel modules, a system program will be required that will facilitate communication with any new kernel modules. I am currently unsure how to do this.
I may also need to refresh and further my C and UNIX programming skills. While I have made no firm decision on a programming language, the use of C is likely, as that is the language that the Linux kernel is written in.
The user interface for controlling the faulty behaviour, may require the use of certain system libraries depending on how this is implemented. Libraries such as ncurses or the Gimp Toolkit (GTK) may be used. These are usually available as part of a standard GNU/Linux system.

Project Deliverables

Additionally to the required two posters and final report, my deliverables will be:

- Modules written for the Linux Kernel
- System program for communicating with the new kernel module
- Controlling interface

A demonstration will be given of the completed work.

Initial Bibliography

Books
Behrouz Forouzan. (2003) TCP/Protocol Suite. McGraw-Hill 0-07-119962-4
Journals
ACM transactions on internet technology
http://www.acm.org/pubs/periodicals/toit/
(2003)
Websites
The Linux Kernel Documentation: http://www.kernel.org/ (June 2003)
The Linux Documentation Project: http://en.tldp.org/ (October 2003)
Linux Kernel Programming: http://www.kernelhacking.org/ (April 2002)

Appendix C

Kernel Module Header

```
/*
 * Module that forms part of a controllable faulty network router
 *
 * Completed as part of a final year project in the
 *    Department of Computer Science,
 *    University of Wales, Aberstwyth
 *
 * (C) 2003 Thomas Salmon, this code is GNU GPL
 */

#include <linux/skbuff.h>
#include <linux/in.h>


/* types of rule that may be used */
#define DROP    1
#define REORDER    2

/* operation types */
#define APPEND    0
#define INSERT    1
#define UPDATE    2
#define    DELETE    3
#define FLUSH     4
#define LIST     5

/* define the protocol numbers */
#define PROTO_ALL        -1
#define PROTO_ICMP      1
#define PROTO_IGMP  2
#define PROTO_TCP       6
#define PROTO_UDP       17


/* define the structs that are specified in the Design */

/* holds a part that is being held (taken out of order) */
struct stored_packet{
```

```
    struct sk_buff *delayed_skb;
    struct nf_info *delayed_info;
    struct stored_packet *next;
};

/* controls reordering going on (does not define reordering rules) */
struct reorder {
    int chosen_burst_size; /* no of packets to delay */
    int delayed_packets; /* no of packets delayed so far, in this burst */
    int chosen_delay_size; /* no of packets to delay by */
    int current_delay; /* how many delayed already */
    struct stored_packet *reorder; /* packets taken out of order */
    struct reorder *next; /* any other reordering going on */
};

struct rules {
    int rule_type;
    int proto; /* transport layer protocol */
    int min_size; /* min packet size for this rule */
    int max_size; /* max size for this rule */
    int probability; /* chance in one thousand */
    int max_burst_size; /* reorder only! max no of packets in burst */
    int max_delay_size;; /* max no of packets to delay by */
    int burst_skew;
    int delay_skew;
    struct reorder *reorder; /* any reordering going on */
    struct rules *next; /* next rule */
};
```

Appendix D

Kernel Module Source Code

```
/*
 * Module that forms part of a controllable faulty network router
 *
 * Completed as part of a final year project in the
 *  Department of Computer Science,
 *  University of Wales, Aberstwyth
 *
 * (C) 2003 Thomas Salmon, this code is GNU GPL
 */

#include <linux/ip_faulty_router.h>

#include <linux/module.h>

#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/netfilter_ipv4/ip_tables.h>

#include <linux/random.h>

/* struct sent from user space */
struct params {
    int rule_type; /* packet order of packet drop */
    int operation; /* append, insert, update, delete, flush */
    int proto; /* transport layer protocol */
    int rule_num; /* only used for insert, update and delete operations */
    int min_size; /* min size of packet to match */
    int max_size; /* max size of packet to match */
    int max_burst_size; /* packet reordering only!, max no of packets in burst*/
    int max_delay_size; /* packet reordering only!, max packets to delay by */
    int burst_skew; /* percentage skew */
    int delay_skew; /* percentage skew */
    int probability; /* chance in 1000 */
};

struct rules *root_rule = NULL;

/* copy values contained in structs */
```

```
void copy_rule(struct rules *from, struct rules *to){
        to->rule_type = from->rule_type;
        to->proto = from->proto;
        to->min_size = from->min_size;
        to->max_size = from->max_size;
        to->probability = from->probability;
        to->max_burst_size = from->max_burst_size;
        to->max_delay_size = from->max_delay_size;
        to->burst_skew = from->burst_skew;
        to->delay_skew = from->delay_skew;
}

/* desends down the list of rules, printing each one out */
void list_rules(struct rules *root){
    if (root == NULL){
        return;
    }

    /* display this rule */
    if (root->rule_type == DROP){
        printk("DROP    ");
    }
    else{
        printk("REORDER    ");
    }

    if (root->proto == PROTO_ALL){
        printk("-p ALL    ");
    }
    else if (root->proto == PROTO_TCP){
        printk("-p TCP    ");
    }
    else if (root->proto == PROTO_UDP){
        printk("-p UDP    ");
    }
    else if (root->proto == PROTO_ICMP){
        printk("-p ICMP    ");
    }
    else if (root->proto == PROTO_IGMP){
        printk("-p IGMP    ");
    }

    printk("min: %d  max: %d    ", root->min_size, root->max_size);

    if (root->rule_type == REORDER){
        printk("Burst: %d (%d)  ", root->max_burst_size, root->burst_skew);
        printk("Delay: %d (%d)    ", root->max_delay_size, root->delay_skew);
    }

    printk("Prob: %d\n", root->probability);

    /* go to display next rule */
```

```
    if (root->next != NULL){
        list_rules(root->next);
    }
    return;
}

/* adds a rule to the list of rules */
/* if position specified, rule inserted before that position */
void add_rule(struct rules *root, struct rules *rule, int position){
    if (root == NULL){
        return;
    }
    else if (root->next == NULL){
        root->next = rule;
    }
    else if (position == 0){ /* insert rule here */
        struct rules *r = kmalloc(sizeof(struct rules), GFP_ATOMIC);
        copy_rule(root, r);
        r->next = root->next;
        r->reorder = root->reorder;
        copy_rule(rule, root);
        root->next = r;
        root->reorder = NULL;

        kfree(rule); /* no longer needed */
    }
    else{
        position--;
        add_rule(root->next, rule, position); /* recurrsive call */
    }
}

struct rules *get_rule(struct rules *root, int position){
    if (root == NULL || position < 0) return NULL;
    if (position == 0) return root;
    if (root->next != NULL) {
        position--;
        return get_rule(root->next, position);
    }
    return NULL; /* position beyond length of list */
}

void delete_rule(struct rules *root, int position){
    /* cannot work in following conditions */
    if (root == NULL) return;
    if (root->next == NULL) return;
    if (position < 1) return;

    /* we delete the following rule, not this one! */
    if (position == 1){
        struct rules *r = root->next->next;
        kfree(root->next);
```

```
        root->next = r;
    }
    else{
        position--;
        delete_rule(root->next, position);
    }
}

void flush_packets(struct stored_packet *root){
    if (root == NULL) return; /* shouldnt have been called */

    /* sanity check */
    if (root->delayed_skb != NULL && root->delayed_info != NULL){
        /* reinject the stored packets, dont just waste them*/
        nf_reinject(root->delayed_skb, root->delayed_info, NF_ACCEPT);
    }

    if (root->next != NULL){
        flush_packets(root->next);
        kfree(root->next);
        root->next = NULL;
    }
    return;
}

void flush_reorder(struct reorder *root){
    if (root == NULL) return; /* shouldnt have been called */
    if (root->next != NULL){
        flush_reorder(root->next);
        kfree(root->next);
        root->next = NULL;
    }
    if (root->reorder != NULL){
        flush_packets(root->reorder);
        kfree(root->reorder);
        root->reorder = NULL;
    }
    return;
}

void flush_rules(struct rules *root){
    if (root == NULL) return; /* shouldnt have been called */
    if (root->next != NULL){
        flush_rules(root->next);
        kfree(root->next);
        root->next = NULL;
    }
    if (root->reorder != NULL){
        flush_reorder(root->reorder);
        kfree(root->reorder);
        root->reorder = NULL;
    }
```

```
    return;
}

/* obeys instructions received from user space */
int rules_operation(struct params *rule){
    struct rules *add = NULL;

    if (rule->operation == LIST){
        /* list all stored rules */
        if (root_rule != NULL){
            list_rules(root_rule);
        }
        return 0;
    }
    else if (rule->operation == FLUSH){
        if (root_rule != NULL){
            flush_rules(root_rule);
            kfree(root_rule);
            root_rule = NULL;
        }
        return 0;
    }
    else if (rule->operation == DELETE){
        if (root_rule == NULL) return -1;
        if (rule->rule_num == 0){
            /* special case to delete the root node */

            if (root_rule->next == NULL){
                kfree(root_rule);
                root_rule = NULL;
            }
            else {
                struct rules *r = root_rule->next;
                kfree(root_rule);
                root_rule = r;
            }
        }
        else{
            delete_rule(root_rule, rule->rule_num);
        }
        return 0;
    }

    /* simple sanity checks */
    if (rule->min_size < 0 || rule->min_size > rule->max_size) return -1;
    if (rule->probability <= 0 || rule->probability > 1000) return -1;
    if (rule->rule_type == REORDER){
        if (rule->max_burst_size <= 0 || rule->max_delay_size <= 0) return -1;
    }

    /* if rule is append, insert or update, form the rule here */
    if (rule->operation == APPEND
```

```
      || rule->operation == INSERT
     || rule->operation == UPDATE){
        add = kmalloc(sizeof(struct rules), GFP_ATOMIC);

        add->rule_type = rule->rule_type;
        add->proto = rule->proto;
        add->min_size = rule->min_size;
        add->max_size = rule->max_size;
        if (rule->rule_type == REORDER){
            add->max_burst_size = rule->max_burst_size;
            add->max_delay_size = rule->max_delay_size;
            add->burst_skew = rule->burst_skew;
            add->delay_skew = rule->delay_skew;
        }
        else{
            add->max_burst_size = -1;
            add->max_delay_size = -1;
            add->burst_skew = 0;
            add->delay_skew = 0;
        }
        add->probability = rule->probability;
        add->reorder = NULL;
        add->next = NULL;
    }

    if (rule->operation == APPEND){
        if (root_rule == NULL) root_rule = add;
        else add_rule(root_rule, add, -1);
    }
    else if (rule->operation == INSERT){
        if (root_rule == NULL) root_rule = add;
        else add_rule(root_rule, add, rule->rule_num);
    }
    else if (rule->operation == UPDATE){
        struct rules *u = get_rule(root_rule, rule->rule_num);

        if (u != NULL){
            /* copy contents over */
            copy_rule(add , u);
        }
        kfree(add); /* free memory */

        if (u == NULL){
            return -1;
        }
    }
    else return -1; /* no such operation */
    return 0;
}

/* return a random int between 1 and max, with the skew applied as percentage */
int random_number(int max, int skew){
```

```
    unsigned int rand;
    int res;
    int tmp;
    get_random_bytes(&rand, sizeof(unsigned int));
    tmp = (((int)rand%max)/2) + (max/2); /* number produced is -max < x < +max*/
    res = tmp + ((skew*max)/100);
    if (res <= 0) return 1;
    if (res > max) return max;
    return res;
}

/* checks the probability of event occuring: 0-false, 1-true */
int chkprob(int prob){
    return (random_number(1000,0) < prob);
}



/****
 * function for dropping
 *****************/

/*
 * Traverses the rules, to see if this packet should be dropped
 * returns true (1) to drop packet
 */
int drop_traverser(struct rules *root, int proto, int len){
    if (root == NULL) return 0;
    if (root->rule_type != DROP) return drop_traverser(root->next, proto, len);
    if (root->proto > PROTO_ALL && root->proto != proto) /* wrong protocol */
        return drop_traverser(root->next, proto, len);
    if (root->min_size > len || root->max_size < len) /* wrong size */
        return drop_traverser(root->next, proto, len);

    /* the packet is the same protocol, and fits the length constraints of
       this rule */
    if (chkprob(root->probability)) return 1;
    else return drop_traverser(root->next, proto, len);
}







/*************************************************************************
 *
 * Function that packets out of the normal order
 * stores them in link lists
 * creates reordering chains, when a rule is being applied (perhaps being
 *                                              applied several times at once
 *
```

```
 ********************************************************************/

/* creates a new struct reorder, that forms part of the linked list */
struct reorder *add_reorder_chain(struct rules *root,
                                  struct reorder *r,
                                  struct stored_packet *p){
    /* sanity check */
    if (root == NULL || p == NULL){
printk("ERROR: root or packet is null\n");
        return NULL;
    }

    if (r != NULL){
        if (r->next == NULL) r->next = add_reorder_chain(root, NULL, p);
    }
    else if (r != NULL && r->next != NULL) {
        add_reorder_chain(root, r->next, p);
    }
    else{
        struct reorder *ro = kmalloc(sizeof(struct reorder), GFP_ATOMIC);
        ro->chosen_burst_size = random_number(root->max_burst_size,
                                              root->burst_skew);
        ro->delayed_packets = 1;
        ro->chosen_delay_size = random_number(root->max_delay_size,
                                              root->delay_skew);
        ro->current_delay = 0;
        ro->reorder = p;
        if (p->delayed_skb == NULL || p->delayed_info == NULL){
printk("Packet invalid\n");
            return NULL; /* failure */
        }
        p->next = NULL; /* initialise the following packet ptr to NULL */
        ro->next = NULL;
        return ro;
    }
    return r;
}

/* adds a packet to link list, returning its address */
struct stored_packet *add_packet(struct stored_packet *sp,
                                 struct sk_buff *skb,
                                 struct nf_info *nf){
    if (sp == NULL){
        struct stored_packet *p = kmalloc(sizeof(struct stored_packet),
                                          GFP_ATOMIC);
        /*p->delayed_skb = pskb_copy(skb, GFP_ATOMIC);
        p->delayed_skb = skb_copy(skb, GFP_ATOMIC);
        */
        p->delayed_skb = skb;
        p->delayed_info = nf;
        p->next = NULL;
        return p;
```

```
    }
    else{
        sp->next = add_packet(sp->next, skb, nf);
        return sp;
    }
}

/*
 * Traverses a chain of reordering operations currently being performed, that
 *are all descended from the same rule
 *Returns 1 if packet is not being delayed
 */
int traverse_reorder_chains(struct reorder *ro,
                            struct sk_buff *skb,
                            struct nf_info *nf){
    if (ro == NULL) return 1;
    if (ro->delayed_packets < ro->chosen_burst_size){
        ro->delayed_packets++;
        /* add packet */
        ro->reorder = add_packet(ro->reorder, skb, nf);
        return 0;
    }
    if (ro->next == NULL) return 1;
    return traverse_reorder_chains(ro->next, skb, nf);
}

/*
 * Traverses the rules regarding reordering
 * Returns true (1) if the packet should be reinjected (not being reordered)
 */
int reorder_traverser(struct rules *root, int proto, int len,
                      struct sk_buff *skb, struct nf_info *nf){
    if (root == NULL) return 1;
    if (root->rule_type != REORDER)
        return reorder_traverser(root->next, proto, len, skb, nf);
    if (root->proto > PROTO_ALL && root->proto != proto) /* wrong protocol */
        return reorder_traverser(root->next, proto, len, skb, nf);
    if (root->min_size > len || root->max_size < len) /* wrong size */
        return reorder_traverser(root->next, proto, len, skb, nf);

    /* this rule can be applied to the packet */
    if (chkprob(root->probability)){ /* forming new reorder chain */
        struct stored_packet *p = kmalloc(sizeof(struct stored_packet),
                                          GFP_ATOMIC);
        /*p->delayed_skb = pskb_copy(skb, GFP_ATOMIC);*/
        /*p->delayed_skb = skb_copy(skb, GFP_ATOMIC);*/
        p->delayed_skb = skb;
        p->delayed_info = nf;
        p->next = NULL;
        root->reorder = add_reorder_chain(root, root->reorder, p);
    }
    else if (traverse_reorder_chains(root->reorder,skb,nf))
```

```
        return reorder_traverser(root->next, proto, len, skb, nf);
        /* try and fill any correct reordering going on */
        /* else check next rule */

        /* packet was not delayed on any reorder chains */
    else return 0; /* packet has been taken out of order */
    return 0;
}




/**************************************************************************
 *
 * Function that take reordered packets and reinject them, as appropriate
 *
 **************************************************************************/

/* called to reinject all packets that have been taken out of order */
void reinject_packets(struct stored_packet *p){
    if (p == NULL) return;
    if (p->delayed_skb != NULL && p->delayed_info != NULL){
        nf_reinject(p->delayed_skb, p->delayed_info, NF_ACCEPT);
    }
    if (p->next == NULL) return; /* stop here */
    reinject_packets(p->next);
    kfree(p->next);
    p->next = NULL;
}

/* assess and updates all the reorder which is under way
 * called by rule_update every time any packet arrives
 * Returns: 1 - if reorder job complete (delete it) */
int reorder_update(struct reorder *r){
    int completed = 0;
    if (r == NULL) return 0;
    if (r->delayed_packets >= r->chosen_burst_size){
        if (r->current_delay >= r->chosen_delay_size){
            if (r->reorder != NULL){
                reinject_packets(r->reorder);
                kfree(r->reorder);
                r->reorder = NULL;
            }
            completed = 1; /* this reorder process is finished */
        }
        else{
            r->current_delay++; /* increment how long they've been delayed */
        }
    }
    if (r->next != NULL){
        if (reorder_update(r->next)){
            /* the next reordering terminated, erase it from linked list */
```

```
            struct reorder *tmp = r->next->next;
            kfree(r->next);
            r->next = tmp;
        }
    }
    return completed;
}

/*********************************************************
 * SHOULD BE CALLED FOR EVERY PACKET THAT ARRIVES
 *********************************************************/
/* scans through the rules, checking any reordering that is going on */
void rule_update(struct rules *root){
    if (root == NULL) return;
    if (root->reorder != NULL){
        if (reorder_update(root->reorder)){
            /* the root reorder process is completed, delete it! */

            if (root->reorder != NULL){
                if (root->reorder->next != NULL){
                    struct reorder *tmp;
                    tmp = root->reorder->next;
                    kfree(root->reorder);
                    root->reorder = tmp;
                }
                else{
                    kfree(root->reorder);
                    root->reorder = NULL;
                }
            }
        }
    }
    if (root->next != NULL) rule_update(root->next);
}




/*
 * First function called when packets arrive
 * Returns:
 *        NF_DROP - to discard
 *        NF_QUEUE - to pass on the packets
 */
static unsigned int dropper(unsigned int hook,
                            struct sk_buff **skb,
                            const struct net_device *in,
                            const struct net_device *out){
    if (root_rule == NULL) return NF_QUEUE; /* there are no rules */
    if (drop_traverser(root_rule, skb[0]->h.ipiph->protocol,
       skb[0]->len)) return NF_DROP;
```

```
    return NF_QUEUE; /* packet should be passed on */
}


/*
 * Function called for packets that have 'NF_QUEUE' returned for them
 * To allow packets to pass, nf_reinject must be called for them
 */
void reorder(struct sk_buff *skb, struct nf_info *nf, void *data){
    if (reorder_traverser(root_rule, skb->h.ipiph->protocol, skb->len, skb, nf))
        nf_reinject(skb, nf, NF_ACCEPT);
    rule_update(root_rule);
}


/*
 * User space communication function
 */
int user_control(struct sock *sk, int cmd, void *user, unsigned int len){
    if (!capable(CAP_NET_ADMIN)) return -EPERM;
    rules_operation( ((struct params *)user) );
    return 0;
}
int user_control_signed(struct sock *sk, int cmd, void *user, int *len){
    return user_control(sk, cmd, user, (unsigned int)*len);
}

/* structs for registring components */
static struct nf_hook_ops drop_ops = {
    { NULL, NULL },
    (nf_hookfn *)dropper,
    PF_INET,
    NF_IP_FORWARD,
    NF_IP_PRI_FILTER
};

static struct nf_sockopt_ops user_comms = {
    { NULL, NULL },
    PF_INET,
    IPT_BASE_CTL+2, /* 66 */
    IPT_BASE_CTL+3, /* 67 */
    user_control, /* function to be called */
    IPT_BASE_CTL+2, /* 66 */
    IPT_BASE_CTL+3, /* 67 */
    user_control_signed, /* function to be called */
    0,
    NULL
};

static int __init init(void){
    printk("Faulty Network Router - Thomas Salmon (tws9@aber.ac.uk)\n");
```

```
    if (nf_register_hook(&drop_ops) < 0){
        printk(KERN_ERR "Failed to register hook for dropping\n");
        return -1;
    }
    if (nf_register_queue_handler(PF_INET, (nf_queue_outfn_t)reorder, NULL) < 0){
        printk(KERN_ERR "Failed to register queue for reordering\n");
        return -1;
    }
    if (nf_register_sockopt(&user_comms) < 0){
        printk(KERN_ERR "Failed to register communication socket with user\n");
        return -1;
    }
    return 0;
}

static void __exit fini(void){
    /* dispose of memory */
    if (root_rule != NULL){
        flush_rules(root_rule);
        kfree(root_rule);
    }

    /* unregister any hooks */
    nf_unregister_queue_handler(PF_INET);
    nf_unregister_hook(&drop_ops);
}

module_init(init);
module_exit(fini);
```

Appendix E

User Space program ‘faultctrl’ Source Code

```
/*
 * User space program to communicate with kernel module: ip_faulty_router.o
 *enabling the faulty parameters inside the module
 *
 * Author: Thomas Salmon tws9@aber.ac.uk
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include "kernel_comms.h"
#include <linux/netfilter_ipv4/ip_tables.h>
#include "faultctrl.h"

void usage(){
    printf("FaultCTRL: Tool to control Faulty Network Router\n\nUSAGE:\n");
    printf("faultctrl \t -A drop|reorder [options] probability \n");
    printf("faultctrl \t -U num drop|reorder [options] probability \n");
    printf("faultctrl \t -I [num] drop|reorder [options] probability \n");
    printf("faultctrl \t -D num\n");
    printf("faultctrl \t -F\n");
    printf("faultctrl \t -L\n\n");
    printf("Rules:\n");
    printf("\t -A \t Append a rule to the current rules\n");
    printf("\t -U \t Update a rule specified by the rule number\n");
    printf("\t -D \t Delete a rule that matches the options\n");
    printf("\t -I \t Insert a rule at given position in chain of rules\n");
    printf("\t -F \t Flush all rules, clear all faulty routing\n");
    printf("\t -L \t List all rules in operation\n\n");
    printf("Options:\n");
    printf("\t -p protocol \t Transport Layer Protocol \n");
    printf("\t  \t Values: igmp, icmp, tcp, udp \n");
```

```
    printf("\t -min size \t Minimum size (bytes) of packet to be delayed\n");
    printf("\t -max size \t Maximum size (bytes) of packet to be delayed\n");
    printf("\t -maxburst size \t Maximum no. packets to be reordered\n");
    printf("\t -maxdelay size \t Maximum no. packets to delay by\n");
    printf("\t -burstskew skew\t Percentage skew to be applied to the burst\n");
    printf("\t -delayskew skew\t Percentage skew to be applied to the delay\n");
}


/* fault will contain all parameters for this rule */
struct params *fault;

/* performs sanity checks on *fault, return 1 if they pass */
int check_params(){
    if (fault->operation == FLUSH) return 1; /* special case */
    if (fault->operation == LIST) return 1; /* special case */

    if ((fault->operation == UPDATE || fault->operation == DELETE)
        && fault->rule_num < 0) return -1;
    if (fault->operation == DELETE) return 1; /* special case */

    if (!(fault->rule_type == DROP || fault->rule_type == REORDER)) return -1;
    if (fault->operation < APPEND || fault->operation > FLUSH) return -1;
    if ((fault->operation == UPDATE || fault->operation == DELETE)
        && fault->rule_num < 0) return -1;
    if (fault->min_size < 0 || fault->max_size < 0) return -1;
    if (fault->min_size >= fault->max_size) return -1;
    if (fault->max_burst_size < 0) return -1;
    if (fault->max_delay_size < 0) return -1;
    if (fault->probability < 1 || fault->probability > 1000) return -1;
    return 1; /* sanity checked */
}

/* outputs values stored in params struct */
void output_debugging(){
    printf("Rule Type: \t%d\n", fault->rule_type);
    printf("Operation: \t%d\n", fault->operation);
    printf("Proto: \t%d\n", fault->proto);
    printf("Rule num: \t%d\n", fault->rule_num);
    printf("Min size: \t%d\n", fault->min_size);
    printf("Max size: \t%d\n", fault->max_size);
    printf("Max burst: \t%d\n", fault->max_burst_size);
    printf("Max delay: \t%d\n", fault->max_delay_size);
    printf("Burst skew: \t%d\n", fault->burst_skew);
    printf("Delay skew: \t%d\n", fault->delay_skew);
    printf("Probability: \t%d\n", fault->probability);
}


/* passes params to kernel */
int pass_to_kernel(){
    int fd;
```

```
    if (check_params() == -1) return -1;

    /* connect to kernel module, ip_faulty_router */
    fd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    return setsockopt(fd, 0, IPT_BASE_CTL+2, fault, sizeof(struct params));

    output_debugging();

    printf("\nMODULE IS NOT COMMUNICATING WITH KERNEL\n");
    return -1;
}

/* load the default values into the params */
void load_defaults(){
    fault->rule_type = DROP;
    fault->operation = APPEND;
    fault->proto = PROTO_ALL;
    fault->rule_num = -1;
    fault->min_size = 0;
    fault->max_size = 1500;
    fault->max_burst_size = 0;
    fault->max_delay_size = 0;
    fault->burst_skew = 0;
    fault->delay_skew = 0;
    fault->probability = 0;
}

int main(int argc, char **argv){
    int current_arg = 2;
    if (argc < 2){
        usage();
        return -1;
    }

    fault = malloc(sizeof(struct params));
    load_defaults();

    if (strcmp(argv[1], "-A") == 0 && argc >= 4){ /* append */
        fault->operation = APPEND;
    }
    else if (strcmp(argv[1], "-U") == 0 && argc >= 5){ /* update */
        fault->operation = UPDATE;
        fault->rule_num = atoi(argv[2]);
        current_arg++;
    }
    else if (strcmp(argv[1], "-I") == 0 && argc >= 4){ /* insert */
        fault->operation = INSERT;
        if (strcmp(argv[2], "drop") != 0 && strcmp(argv[2], "reorder") != 0){
            /* next arg is the rule_num */
            fault->rule_num = atoi(argv[2]);
            current_arg++;
        }
```

```
}
else if (strcmp(argv[1], "-D") == 0 && argc >= 3){ /* delete */
    fault->operation = DELETE;
    fault->rule_num = atoi(argv[2]);
    if (pass_to_kernel() != 0){
        printf("Error when deleting rule\n");
        return -1;
    }
    return 0;
}
else if (strcmp(argv[1], "-F") == 0){
    fault->operation = FLUSH;
    if (pass_to_kernel() != 0){
        printf("Error when flushing rules\n");
        return -1;
    }
    return 0;
}
else if (strcmp(argv[1], "-L") == 0){
    fault->operation = LIST;
    if (pass_to_kernel() != 0){
        printf("Error when flushing rules\n");
        return -1;
    }
    return 0;
}
else{
    usage();
    return -1;
}

/* read in the drop or reorder instruction */
if (strcmp(argv[current_arg],"drop") == 0){
    fault->rule_type = DROP;
}
else if (strcmp(argv[current_arg],"reorder") == 0){
    fault->rule_type = REORDER;
}
else{
    usage();
    return -1;
}
current_arg++;

/* read in the options */
while (current_arg < argc-2){ /* -2 as all options are pairs */
    if (strcmp(argv[current_arg], "-p") == 0){
        if (strcmp(argv[current_arg+1], "icmp") == 0){
            fault->proto = PROTO_ICMP;
        }
        else if (strcmp(argv[current_arg+1], "igmp") == 0){
            fault->proto = PROTO_IGMP;
```

```
            }
            else if (strcmp(argv[current_arg+1], "tcp") == 0){
                fault->proto = PROTO_TCP;
            }
            else if (strcmp(argv[current_arg+1], "udp") == 0){
                fault->proto = PROTO_UDP;
            }
            else {
                usage();
                return -1;
            }
        }
        else if (strcmp(argv[current_arg], "-min") == 0){
            fault->min_size = atoi(argv[current_arg+1]);
        }
        else if (strcmp(argv[current_arg], "-max") == 0){
            fault->max_size = atoi(argv[current_arg+1]);
        }
        else if (strcmp(argv[current_arg], "-maxburst") == 0){
            fault->max_burst_size = atoi(argv[current_arg+1]);
        }
        else if (strcmp(argv[current_arg], "-maxdelay") == 0){
            fault->max_delay_size = atoi(argv[current_arg+1]);
        }
        else if (strcmp(argv[current_arg], "-burstskew") == 0){
            fault->burst_skew = atoi(argv[current_arg+1]);
        }
        else if (strcmp(argv[current_arg], "-delayskew") == 0){
            fault->delay_skew = atoi(argv[current_arg+1]);
        }
        else{
            usage();
            return -1;
        }
        current_arg = current_arg + 2; /* increment by 2, as options in pairs */
    }

    if (current_arg != argc-1){
        /* current_arg should be pointing to the last arg */
        usage();
        return -1;
    }

    /* read the probability, last arg */
    fault->probability = atoi(argv[argc-1]);

    if (fault->rule_type == REORDER && (fault->max_burst_size == 0 || fault->max_delay_size == 0)
        usage();
        printf("\nNB: Please specify a max burst rate and delay size for reordring\n");
        return -1;
    }
```

```
    if (check_params() < 0){
        usage();
    }
    else if (pass_to_kernel() != 0)
        printf("Error communicating with kernel module\n");

    free(fault);
    return 0;
}
```

Appendix F

Packet Drop Test Analysis Script

```
#!/usr/bin/perl -w

# Test script to determine the extent of pack dropping, shown between two
#  tcpdump output (tcpdump -nvvv)
#       numerical output should be used for tcpdump, so that host and ports
#              numbers appear in number format

# Author:
#    Thomas William Salmon - tws9@aber.ac.uk tom@slashtom.org

# USAGE:
#  chkdrop.pl
#       [-range min,max,min,max]
#        [-icmp [min,max,min,max,....]]
#        [-igmp [min,max,min,max,....]]
#        [-tcp [min,max,min,max,....]]
#        [-udp [min,max,min,max,....]]
#        -s source_ip_address
#        -d destination_ip_address
#        sender_packet_headers.txt
#        receiver_packet_headers.txt

use strict;

sub usage{
    print " USAGE:\n";
    print "  chkdrop.pl\n";
    print "     [-range min,max,min,max]\n";
    print "        [-icmp [min,max,min,max,....]]\n";
    print "        [-igmp [min,max,min,max,....]]\n";
    print "        [-tcp [min,max,min,max,....]]\n";
    print "        [-udp [min,max,min,max,....]]\n";
    print "        -s source_ip_address\n";
    print "        -d destination_ip_address\n";
    print "        sender_packet_headers.txt\n";
    print "        receiver_packet_headers.txt\n";
    exit;
}
```

```
if ($#ARGV < 5){
    usage();
}

my $srcIP;
my $dstIP;
my $srcHeaders;
my $dstHeaders;

my $totalPacketsSent = 0;
my $totalPacketsReceived = 0;
my $minmax;

my %icmp = (
    info    =>    0,
    minmax    =>    0,
);
my %igmp = (
    info    =>    0,
    minmax    =>    0,
);
my %tcp = (
    info    =>    0,
    minmax    =>    0,
);
my %udp = (
    info    =>    0,
    minmax    =>    0,
);

# read in arguments
for (my $i=0; $i <= $#ARGV-2; $i++){
    if ($ARGV[$i] =~ m/-s/){ # tis the source ip
        $srcIP = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-d/){ # tis the destination ip
        $dstIP = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-range/){
        $minmax = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-icmp/){
        $icmp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            #$icmp{minmax} = [ split(/,/, $ARGV[$i+1]) ];
            $icmp{minmax} = $ARGV[$i+1];
            $i++;
```

```
        }
    }
    elsif ($ARGV[$i] =~ m/-igmp/){
        $igmp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $igmp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    elsif ($ARGV[$i] =~ m/-tcp/){
        $tcp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $tcp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    elsif ($ARGV[$i] =~ m/-udp/){
        $udp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $udp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    else{
        usage();
    }
}
#last 2 args are the headers
$srcHeaders = $ARGV[$#ARGV-1];
$dstHeaders = $ARGV[$#ARGV];


#open files containing headers
open(SRC, $srcHeaders);
open(DST, $dstHeaders);
my @src_contents = <SRC>;
my @dst_contents = <DST>;
close(SRC);
close(DST);

# read in the data from the files - only record records containing src and
#dst IP addresses (remove the noise from the rest of the network)
my @src_data = grep /$srcIP.* > $dstIP/, @src_contents;
my @dst_data = grep /$srcIP.* > $dstIP/, @dst_contents;
push @src_data, grep /$dstIP.* > $srcIP/, @src_contents;
push @dst_data, grep /$dstIP.* > $srcIP/, @dst_contents;

# strip away the time stamp in each, as that differs on the two machines
s/^[0-9]+:[0-9]+:[0-9]+\.[0-9]+ //g for @src_data;
```

```
s/^[0-9]+:[0-9]+:[0-9]+\.[0-9]+ //g for @dst_data;
# remove the ttl, as this WILL be different
s/ttl [0-9]+,//g for @src_data;
s/ttl [0-9]+,//g for @dst_data;

# strip away the MAC addresses, as these will be difference for each,
#as router will change MAC address
s/^.*:.*:.*:.*:.*:.* .*:.*:.*:.*:.*:.. //g for @src_data;
s/^.*:.*:.*:.*:.*:.* .*:.*:.*:.*:.*:.. //g for @dst_data;



# displays information by the protocol, and range
#PARAMS:
#    string - protocol (blank for all)
#    string - range of lengths: min,max,min,max....
sub proto{
    my $proto = $_[0];
    my $range = $_[1];
    my @sent = grep /$proto/, @src_data;
    my @received = grep /$proto/, @dst_data;
    #$udp{totalSent} = scalar(@sent);
    #$udp{totalReceived} = scalar(@received);
    if ($proto =~ m/win/){
        # change name to TCP
        print("tcp:\n");
    }
    else{
        print("$proto:\n");
    }
    print("\tPackets Sent:\t".scalar(@sent)."\n");
    print("\tPackets Received:\t".scalar(@received)."\n");
    if (scalar(@sent) != 0){
        print("\t\tSuccess:\t" . (scalar(@received)/scalar(@sent))*100 ."%\n");
    }
    else{
        print("\t\tSuccess: 0%\n");
    }

    if (defined $range){
        my @minmax = split(/,/, $range);
        for (my $i=0; $i+1 < scalar(@minmax); $i=$i+2){
            print("\t\tPacket Range: $minmax[$i] to $minmax[$i+1]\n");
            my $scount=0;
            for (@sent){
                /(.*)len ([0-9]+)(.*)/;
                if ($2 > $minmax[$i] && $2 < $minmax[$i+1]){
                    $scount++;
                }
            }
            print("\t\t\tSent:\t$scount\n");
```

```
            my $dcount=0;
            for (@received){
                /(.*)len ([0-9]+)(.*)/;
                if ($2 > $minmax[$i] && $2 < $minmax[$i+1]){
                    $dcount++;
                }
            }
            print("\t\t\tReceived:\t$dcount\n");
            if ($scount == 0){
                print("\t\t\tSuccess: 0%\n");
            }
            else{
                print("\t\t\tSuccess:\t".($dcount/$scount)*100 ."%\n");
            }
        }
    }
}


# output section

print("Total Packets");
proto("",$minmax);

if ($icmp{info} == 1) { proto("icmp", $icmp{minmax}); }
if ($igmp{info} == 1) { proto("igmp", $igmp{minmax}); }
if ($udp{info} == 1) { proto("udp", $udp{minmax}); }
if ($tcp{info} == 1) { proto("win", $tcp{minmax}); } # tcp
```

Appendix G

Packet Reorder Test Analysis Script

```
#!/usr/bin/perl -w

# Test script to determine the extent of pack dropping, shown between two
#  tcpdump output (tcpdump -nvvv)
#       numerical output should be used for tcpdump, so that host and ports
#              numbers appear in number format

# Author:
#    Thomas William Salmon - tws9@aber.ac.uk tom@slashtom.org

# USAGE:
#  chkdrop.pl
#       [-range min,max,min,max]
#        [-icmp [min,max,min,max,....]]
#        [-igmp [min,max,min,max,....]]
#        [-tcp [min,max,min,max,....]]
#        [-udp [min,max,min,max,....]]
#        -s source_ip_address
#        -d destination_ip_address
#        sender_packet_headers.txt
#        receiver_packet_headers.txt

use strict;

sub usage{
    print " USAGE:\n";
    print "  chkdrop.pl\n";
    print "     [-range min,max,min,max]\n";
    print "        [-icmp [min,max,min,max,....]]\n";
    print "        [-igmp [min,max,min,max,....]]\n";
    print "        [-tcp [min,max,min,max,....]]\n";
    print "        [-udp [min,max,min,max,....]]\n";
    print "        -s source_ip_address\n";
    print "        -d destination_ip_address\n";
    print "        sender_packet_headers.txt\n";
```

```
    print "        receiver_packet_headers.txt\n";
    exit;
}

if ($#ARGV < 5){
    usage();
}

my $srcIP;
my $dstIP;
my $srcHeaders;
my $dstHeaders;

my $minmax;

my %icmp = (
    info    =>    0,
    minmax    =>    0,
);
my %igmp = (
    info    =>    0,
    minmax    =>    0,
);
my %tcp = (
    info    =>    0,
    minmax    =>    0,
);
my %udp = (
    info    =>    0,
    minmax    =>    0,
);

# read in arguments
for (my $i=0; $i <= $#ARGV-2; $i++){
    if ($ARGV[$i] =~ m/-s/){ # tis the source ip
        $srcIP = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-d/){ # tis the destination ip
        $dstIP = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-range/){
        $minmax = $ARGV[$i+1];
        $i++;
    }
    elsif ($ARGV[$i] =~ m/-icmp/){
        $icmp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $icmp{minmax} = $ARGV[$i+1];
            $i++;
```

```
        }
    }
    elsif ($ARGV[$i] =~ m/-igmp/){
        $igmp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $igmp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    elsif ($ARGV[$i] =~ m/-tcp/){
        $tcp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $tcp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    elsif ($ARGV[$i] =~ m/-udp/){
        $udp{'info'} = 1;
        if (!($ARGV[$i+1] =~ m/^-.*/)){
            # we have min max ranges
            $udp{minmax} = $ARGV[$i+1];
            $i++;
        }
    }
    else{
        usage();
    }
}
#last 2 args are the headers
$srcHeaders = $ARGV[$#ARGV-1];
$dstHeaders = $ARGV[$#ARGV];


#open files containing headers
open(SRC, $srcHeaders);
open(DST, $dstHeaders);
my @src_contents = <SRC>;
my @dst_contents = <DST>;
close(SRC);
close(DST);

# read in the data from the files - only record records containing src and
#dst IP addresses (remove the noise from the rest of the network)
my @src_data1 = grep /$dstIP/, @src_contents;
my @dst_data1 = grep /$srcIP/, @dst_contents;
my @src_data = grep /$dstIP/, @src_data1;
my @dst_data = grep /$dstIP/, @dst_data1;

# strip away the time stamp in each, as that differs on the two machines
s/^[0-9]+:[0-9]+:[0-9]+\.[0-9]+ //g for @src_data;
```

```
s/^[0-9]+:[0-9]+:[0-9]+\.[0-9]+ //g for @dst_data;
# remove the ttl, as this WILL be different
s/ttl [0-9]+,//g for @src_data;
s/ttl [0-9]+,//g for @dst_data;

# strip away the MAC addresses, as these will be difference for each,
#as router will change MAC address
s/^.*:.*:.*:.*:.*:.* .*:.*:.*:.*:.*:.. //g for @src_data;
s/^.*:.*:.*:.*:.*:.* .*:.*:.*:.*:.*:.. //g for @dst_data;

my @dst_orig = @dst_data; # take a backup of dst data


my @reordered_packets;

# records the delay experienced
#    PARAMS:    String line, int delay, int burst, int interval
sub record{
    # print("RECORD:\t$_[1] : $_[2] : $_[3] : $_[0]\n");
    push @reordered_packets, "$_[1] : $_[2] : $_[3] : $_[0]";
}


my $s = 0;;
my $d = 0;
my $orig_d = 0;
my $last = 0; # index of last packet taken out of order
my $b = 0;;
my $id;
my $delay;
my $tmp_d;
my $tmp_s;

my $max_d = scalar(@dst_data);
my $max_s = scalar(@src_data);
my $max_delay = 200; #should not exceed this

my $last_burst = 0;

# synchronise data first
#while($d < $max_d-1 && $src_data[$s] ne $dst_data[$d]){
#    $d++;
#}

for (; $s<$max_s && $d<$max_d; $s++){
    # check for packet drop
    if (scalar (grep /\Q$src_data[$s]/, @dst_data) == 0){
        next;
    }
    $delay = 0;
    $orig_d = $d;
#print("$s\t$d\n");
```

```
#print(" $src_data[$s] $dst_data[$d] \n");
    while($delay < $max_delay && $d < $max_d && $src_data[$s] ne $dst_data[$d]){
        $d++;
        $delay++;
    }
    if ($delay > 0 && $delay < $max_delay && $delay < $max_d){
#print(" $src_data[$s] $dst_data[$d] Delay:\t$delay\n\n");
        if ($d <= $last + $b){
            #$last++;
            $d = $orig_d + 1;
            next;
        }

        $b = 1; # set burst to one
        $tmp_d = $d+1;
        $tmp_s = $s+1;
        # skip over packet dropped entries
        while (scalar (grep /\Q$src_data[$tmp_s]/, @dst_data) == 0){
            $tmp_s++;
        }
        while ($tmp_s < $max_s && $tmp_d < $max_d &&
                $src_data[$tmp_s] eq $dst_data[$tmp_d]){
            $b++; # increment burst
            $tmp_d++;
            $tmp_s++;

            # skip of packet dropped stuff
            while (scalar (grep /\Q$src_data[$tmp_s]/, @dst_data) == 0){
                $tmp_s++;
            }
        }

        # record the burst
        if ($d == $max_d){ $d = $orig_d + 1; next; }
#        print("$dst_data[$d], $delay, $b, ($d - $last_burst)\n");
        if ($last_burst == 0){ # special case
            record($dst_data[$d], $delay, $b, ($d - $last_burst - $delay + 1));
        }
        else{
            record($dst_data[$d], $delay, $b, ($d - $last_burst));
        }
        $last_burst = $d + $b - 1;
        $last = $d;

        $d = $orig_d + 1;
        next;
    }
    $d = $orig_d + 1;
}

#print("$s\t$d\n");
#print(" $src_data[$s] $dst_data[$d] \n");
```

```
# gets stats of @_
# returns average_interval, average_burst, average_delay
sub get_stats{
    my $total_entries = scalar(@_);
    my $total_delay = 0;
    my $total_burst = 0;
    my $total_interval = 0;
    my $i;
    my $j;
    my $k;
    my $l;
    my $loop_count = 0;
    for (@_){
        ($i, $j, $k, $l) = split(/:/, $_, 4);
        $loop_count++;
        $total_delay = $total_delay + $i;
        $total_burst = $total_burst + $j;
        if ($loop_count == 1){
            next; # discount the first interval reading
        }
        $total_interval = $total_interval + $k;
    }

    if ($total_entries > 0){
        return ($total_interval/$total_entries,
                $total_burst/$total_entries,
                $total_delay/$total_entries);
    }
    else{
        return (0, 0, 0);
    }
}



sub proto{
    my $proto = $_[0];
    my $range = $_[1];
    my @reorder = grep /$proto/, @reordered_packets;
    if ($proto =~ m/win/){
        # change name to TCP
        print("tcp:\n");
    }
    else{
        print("$proto:\n");
    }

    #form some statistics as regards reorder interval, burst and delay averages

    my $aInt;
    my $aBurst;
```

```
    my $aDelay;
    ($aInt, $aBurst, $aDelay) = get_stats(@reorder);
    print("\tAverage reorder interval:\t".$aInt."\n");
    print("\tAverage burst:\t".$aBurst."\n");
    print("\tAverage delay:\t".$aDelay."\n");

    if (defined $range){
        my @minmax = split(/,/, $range);
        for (my $i=0; $i+1 < scalar(@minmax); $i=$i+2){
            my @reorder_size;
            print("\t\tPacket Range: $minmax[$i] to $minmax[$i+1]\n");
            my $scount=0;

            for (@reorder){
                /(.*)len ([0-9]+)(.*)/;
                if ($2 > $minmax[$i] && $2 < $minmax[$i+1]){
                    push @reorder_size, $_; # add this entry to array
                }
            }
            ($aInt, $aBurst, $aDelay) = get_stats(@reorder_size);
            print("\t\t\tAverage reorder interval:\t".$aInt."\n");
            print("\t\t\tAverage burst:\t".$aBurst."\n");
            print("\t\t\tAverage delay:\t".$aDelay."\n");

            undef &reorder_size; # clear old array
        }
    }
}
# output section

print("Total Packets");
proto("",$minmax);

if ($icmp{info} == 1) { proto("icmp", $icmp{minmax}); }
if ($igmp{info} == 1) { proto("igmp", $igmp{minmax}); }
if ($udp{info} == 1) { proto("udp", $udp{minmax}); }
if ($tcp{info} == 1) { proto("win", $tcp{minmax}); } # tcp
```