Implementation of AQMs on Linux made Easy

Stephen Braithwaite
Department of Mathematics and Computing
The University of Southern Queensland, Toowoomba, QLD

July 29, 2006

Abstract

This is a project to implement a Mice and Elephants queueing discipline, which favours short flows over long flows, on Linux. The project has three aims. The first aim is to produce a prototype Mice and Elephants router for the purpose of further evaluation of the Mice and Elephants strategy and the Shortest Job First strategy. The second aim is to make a contribution to Linux by making my implementation as code that is both fit for distribution with Linux and useful in a small business or domestic setting. The third aim is to explore and document a method of creating Linux queueing disciplines in general.

Contents

1 Introduction **9**
- 1.1 Rationale . . . 10
- 1.2 Mice and Elephants strategies . . . 10
- 1.3 Linux Queueing Disciplines . . . 11
- 1.4 Ingress Filter . . . 11
- 1.5 Documentation Created . . . 12
- 1.6 User Space Development Environment . . . 12
- 1.7 Overview . . . 12

2 Active Queue Management **15**
- 2.1 Random Early Detection . . . 16
- 2.2 Fairness algorithms . . . 19
- 2.3 Mice and Elephants . . . 20
- 2.4 A Mice and Elephants Queueing Discipline . . . 21
- 2.5 A Mice and Elephants Ingress Filter . . . 22

3 Linux Queueing Disciplines **27**
- 3.1 History of Net-4 . . . 27
- 3.2 Linux Networking Concepts - Classifiers, Policers, Queueing Disciplines and Filters . . . 27
  - 3.2.1 U32 . . . 30
  - 3.2.2 HTB . . . 30
- 3.3 Kernel Modules . . . 31
- 3.4 The Linux Queueing Discipline Interface . . . 31
- 3.5 Setting up a Linux Queueing Discipline Interface . . . 31

4 Implementation **33**
- 4.1 Summary of Queueing Disciplines Developed . . . 33
  - 4.1.1 Drop Tail . . . 33
  - 4.1.2 RED . . . 33
  - 4.1.3 ARED . . . 34

4.1.4 Mice and Elephants . . . 35
4.1.5 Mice and Elephants Classifier . . . 36
4.2 Design Details . . . 37
4.2.1 RED . . . 37
4.2.2 ARED . . . 39
4.2.3 Mered - The First Mice and Elephants Queueing Discipline . . . 39
4.2.4 Meredt - The Second Mice and Elephants Queueing Discipline . . . 44
4.2.5 Meredtf - The Mice and Elephants Ingress Filter . . . 48
4.3 Queueing Discipline Development Environments . . . 49
4.3.1 Tcsim . . . 49
4.3.2 User Mode Linux . . . 50
4.3.3 Umlsim . . . 51
4.3.4 Linquede . . . 52
4.3.5 Comparison of AQM testing environments . . . 52
4.4 Platform . . . 53

5 Testing **55**
5.1 Load Testing . . . 55
5.2 Test Harness . . . 56
5.3 Three Experimental Scenarios . . . 56
5.3.1 The *Mice Only* Scenario . . . 57
5.3.2 The *Congested* Scenario . . . 57
5.3.3 The *Mice and Elephants* Scenario . . . 57
5.3.4 Results . . . 57

6 Deployment and Configuration **59**
6.1 Congestion on the Edge Router . . . 59
6.2 Dropping Packets at the Gateway Router . . . 59
6.3 Edge Router Buffer Level Estimation . . . 61
6.4 Testing the Idea . . . 63

7 Conclusion **65**

A Linquede HOWTO **75**

B Qdisc Man Page **93**

C ARED Man Page **99**

D MERED Man Page **105**

E MEREDT Man Page **111**

F MEREDTF Man Page **117**

G Example TC script **125**

Chapter 1

Introduction

Within IP networks, packets may arrive at a router faster than they may be placed onto the network. When this happens, they are stored in a queue. This queue may not be a pure queue or FIFO, however, in the sense that the first packet that arrived will not necessarily be placed onto the network first. In order to improve performance, the algorithm managing this queue may choose to deliver packets in a different order to that in which they arrived. It may also choose to drop packets or slightly modify (*mark*) them. Such strategies or algorithms for choosing what packets to drop or mark and what order to deliver them in are called *queuing disciplines*. A non trivial queuing discipline, one that implements more than pure queueing is called an *active queue management* (AQM) [44].

In this project I have investigated and implemented a particular type of AQM, called a *mice and elephants* AQM [31]. A mice and elephants AQM can improve the responsiveness of communication. In the context of a mice and elephants queueing discipline, the shorter flows are called *mice*, and the longer flows are called *elephants* [45]. A mice and elephants queuing discipline works by giving priority to the mice.

The queuing disciplines that I have implemented are suitable for use in the access network for a business or domestic residence. I have implemented them in the Linux operating system, due to the ready availability of the source code and the software development tools. On Linux, the code module that implements a queueing discipline algorithm is also called a *queueing discipline* [21].

As part of this project, the mice and elephant AQM has been implemented in the form of Linux queuing disciplines. These have documented in the form of Linux *man* pages. An environment for the development of queuing disciplines on Linux has also been created. This has been documented in the form of a HOWTO.

1.1 Rationale

Responsiveness is one of the key qualities of an internet service. People will judge an internet service by how long it takes to have keystroke echoed, or to see a web page.

During periods of high congestion, the long flows although few in number, account for a disproportionate amount of the traffic flow [45]. The remaining traffic is made up of short flows. Thus, the long flows are termed *elephants* and the short flows are termed *mice*.

Long flows, such as a large download of software, are typically not sensitive to the delays caused by this congestion, but the short flows are [75]. You can imagine that if packets generated by keystrokes in a telnet session had to wait in a queue congested by packets from a large download, the telnet session will be adversely affected by the delays.

Fair queueing is one way to improve the responsiveness of the internet. By ensuring "fair" use amongst traffic flows, we can prevent heavy flows from making the internet unusable for interactive flows [64]. We can implement fair queueing on Linux by deploying a fair queueing discipline, such as SFQ [60] in place of the default drop tail queueing discipline [28]. The rationale behind fair queueing assumes that allocating the same bandwidth to each flow is fair. This project will express the idea that allocating the same bandwidth to each flow is actually disadvantaging the short flows. What is needed is a queueing discipline that actively favours the short flows.

1.2 Mice and Elephants strategies

A Mice and Elephants strategy is one that favours short traffic flows over long traffic flows. Packets arriving at a router are classified according to the count of bytes or packets in the flow that the packet belongs to. Packets belonging to short flows (mice) are given priority over packets that belong to long flows (elephants) [45]. The mice are favoured in two ways:-

- Packets from elephant flows (elephant packets) are dropped according to the Random Early Detection (RED) algorithm [44] whereas packets from mouse flows (mouse packets) are only dropped when the queueing discipline is full, which should be a rare occurrence.

- Mouse packets have priority over elephant packets for dequeuing.

It has been shown in mathematical models and in simulations that a "Mice and Elephants" strategy can produce substantial benefits in situations where there is congestion over a link at the edge of the internet [17, 45, 61].

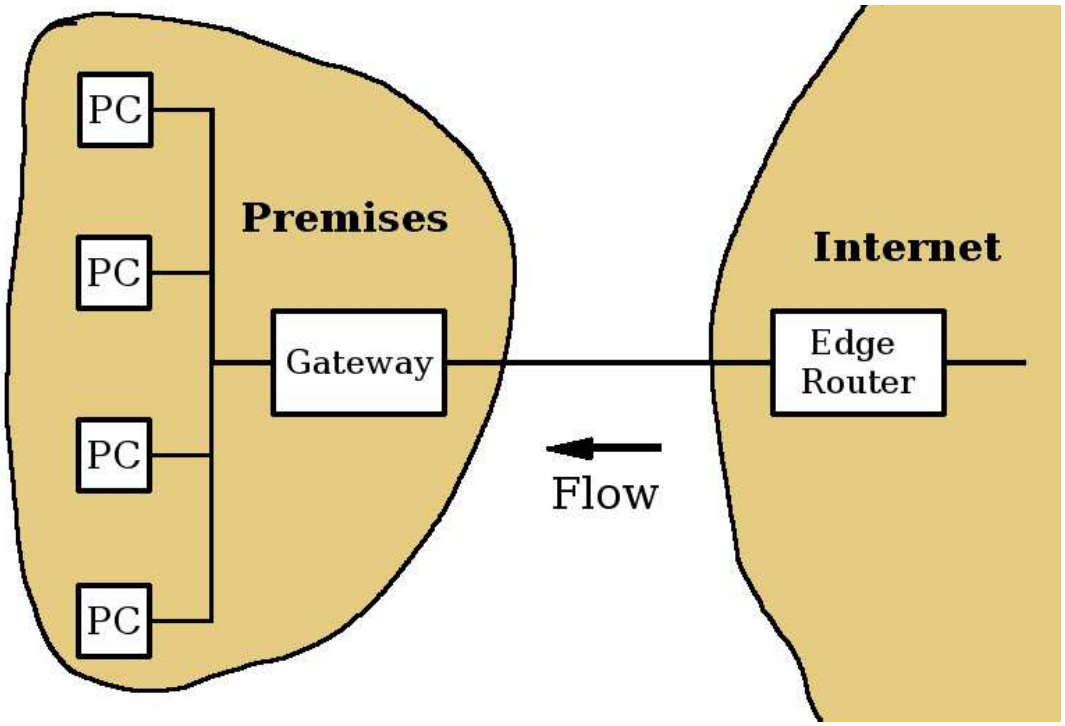

Figure 1.1: Download traffic is typically heavier than upload traffic.

1.3 Linux Queueing Disciplines

On a Linux router the two mechanisms that implement a queueing strategy involving the scheduling, dropping and marking of packets destined for a networking interface are the *queueing discipline* and the *ingress filter* [21]. A queueing discipline controls queueing of outward packets to a network interface. It can mark or drop packets, and also control the order of dequeuing. The ingress filter controls packets after they have entered a network interface. The only action that an ingress filter can take is the dropping or marking of packets. Linux queueing disciplines and ingress filters are to be found *inside* the Linux kernel, so they are not normal user space programs [48]. Even so, Linux has interfaces within the kernel for queueing disciplines and ingress filters and provisions have been made so these can be implemented as kernel modules. Thus it is possible to create a new queueing discipline and attach it into the kernel without rebooting the computer.

1.4 Ingress Filter

The intention is that queueing disciplines developed as part of this project might be useful in a domestic residence or small business. The volume of traffic there will be much less

than in the core of the network, and it is possible to make a more complicated program work here without consuming an excessive amount of computing resources. In this situation, most of the traffic would typically be download traffic and the bottleneck would be the ADSL modem. (See Figure 1.1.) As such, the ideal deployment for our queueing discipline would be on the *edge router*, the router on the internet side of the link where the download traffic is queued. But since such customers have little control over the edge router this would have to be deployed on the gateway router. For this reason, a Mice and Elephants ingress filter was created as well as a queueing discipline.

1.5 Documentation Created

A search of the internet reveals that there is little documentation publicly available on the creation of queueing disciplines. Within the Linux Kernel there is an API for queueing disciplines, but no formal documentation such as Linux man pages, Linux info pages or HOWTO exist. This project has provided documentation of the API in the form of a man page and also created a set of instructions on how to create a queueing discipline in the form of a Linux HOWTO. The HOWTO is complemented with a simple environment for queueing discipline creation and a well documented template for a new queueing discipline.

1.6 User Space Development Environment

In systems running under the Linux operating system, queueing disciplines are implemented in the kernel. The normal safeguards that user space programs enjoy are not present for code within the Linux Kernel. A loose pointer could result in changed memory in any user process in the system. Resources not released cannot be released until the next reboot. Also, debugging is difficult. It is therefore almost essential to use some sort of user space simulation of the code being debugged.

1.7 Overview

The second chapter of this document entitled *Active Queue Management* introduces Random Early Detection, Fair Queuing and the Mice and Elephants strategy.

The third chapter entitled *Linux Queueing Disciplines* will explore Linux's queueing discipline architecture. It introduces Linux's AQM terminology and some important Linux AQM modules. It also describes how queuing disciplines are implemented on Linux.

The fourth chapter entitled *Implementation* discusses the Linux queueing disciplines implemented as a part of this project. It explains the design decisions made, the variant of

Linux used and the environment required to develop queuing disciplines, and testing that was performed.

The fifth chapter entitled *Testing* describes the methods used for load testing and provides information about the efficiency of the resulting queuing disciplines. It describes some experiments that demonstrate the benefits of the created mice and elephants queuing disciplines. It also introduces the test harness that was used for live debugging and scenario testing.

The sixth chapter entitled *Deployment and Configuration* describes an alternative method that was considered for the control of download traffic from a gateway router.

The last chapter entitled *Conclusion* gives the concluding remarks.

Chapter 2

Active Queue Management

In 1986 the internet had what was described as a series of congestion collapses [49]. New algorithms put into the Berkeley Unix BSD TCP that were developed to cope with network congestion [63]. These include:-

- round trip variance estimation
- exponential retransmit timer backoff
- slow start
- more aggressive receiver ack policy
- dynamic window sizing on congestion
- clamped retransmit backoff

Testing showed that the resulting product is fairly good at dealing with congested conditions on the internet [49]. These enhancements to TCP became known as *Tahoe* TCP, after BSD 4.3 Tahoe [37, 51].

The outgoing packet queues on routers have to be kept small. If they were large, then the queueing delay also would be large. So if these queues become full, they drop arriving packets. TCP sources use packet acknowledgments (*ack*) to confirm that sent packets actually arrived. If an acknowledgment fails to arrive, then it is assumed to be because the packet was lost (or the acknowledgment was lost) due to overflowing buffers on packet routers. Thus TCP sources detects network congestion from the failure to receive an acknowledgement for a sent packet, and in response reduces its congestion window by half, effectively reducing its output. Thus, as the TCP sources slow down their traffic generation, the congestion is controlled.

2.1 Random Early Detection

On a router, each network interface has a queue associated with it. Each IP packet to be sent out on an interface will wait in the queue, until it can be transmitted onto the link. The sizes of these queues are not infinite, however, and if they were, the delays would become intolerable. Thus, packets arriving at a queue that is already full, are normally discarded. This queueing behaviour is known as *drop tail*.

In 1993, Random Early Detection (RED) [44] was proposed. Traffic sources normally detect congestion when queues become so full that their packets are lost. RED introduced the idea of notifying sources about congestion *before* the the queue became full by randomly dropping packets. The probability of dropping packets is set to zero if a queue is empty, and increased as a queue became full over a period of time. The main points of the proposal included the following:-

- Congestion can be detected at the packet sources, for instance, but it is easier and more effective to detect and act on this at the congested node.

- IP routers capable of routing large volumes of packets will typically have large queue buffers. If queues at the slowest link are full, this will result high queueing delays and have a negative impact on certain types of traffic, e.g. interactive traffic. It is therefore in our interest to ensure that these queue sizes remain low.

- The only effective feedback mechanism for retarding packets sources in TCP/IP is by utilising the TCP response to dropped packets.

- Dropping packets can help manage a queue even without a source quench feedback mechanism. It even works with non-cooperative packet sources. The following quote is from [44]:

  > If RED gateways mark packets by dropping them, rather than by setting a bit in the packet header, when the average queue size exceeds the maximum threshold, then the RED gateway controls the average queue size even in the absence of a cooperating transport protocol.

- Randomly dropping packets early rather than being forced to drop packets when the queue is full avoids the phenomenon known as *source synchronisation*, which is when all packet sources performing a slow start simultaneously. The following quote is from [44]:

  > ... an additional motivation for using randomisation in the method for marking packets is to avoid the global synchronisation that results from many TCP connections reducing their window at the same time.

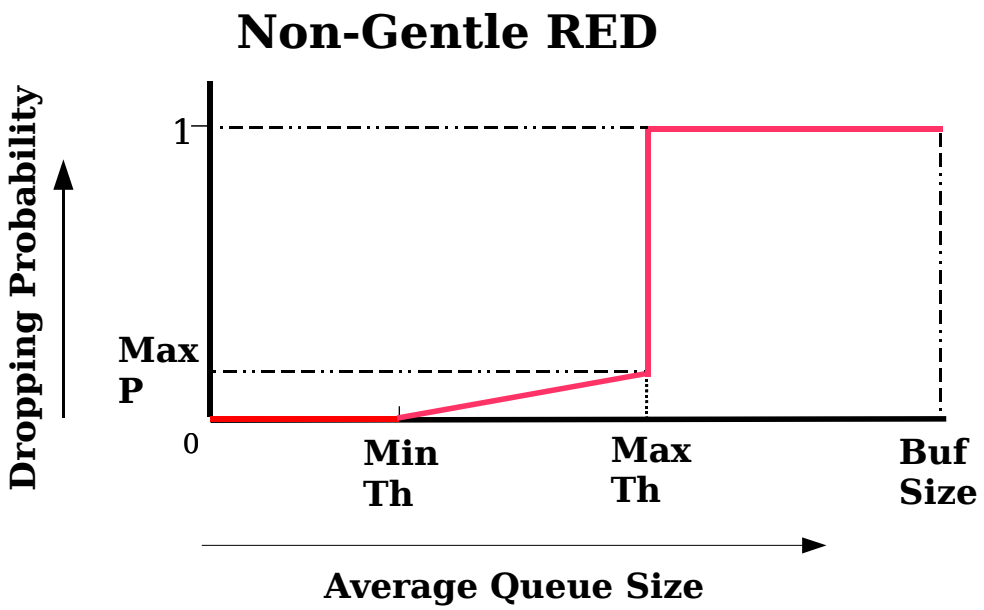

Figure 2.1: The original RED. When the average queue size exceeds the threshold thmax, then the dropping probability is set to 1.

- Design goals are:- minimise packet loss and queueing delay, avoid synchronisation of sources, maintain high link utilisation and remove biases against bursty sources.
- The average queue size is calculated using a time based exponential decay. This is so that sudden bursts of packets from bursty applications do not produce an immediate dropping of packets.
- Normally all packets in a given time slice will have an equal probability of being dropped. Thus the probability that a source will receive a source quench notification (dropped packet) is proportional to the number of packets the source sends.

The algorithm for RED [44] maintains an average queue size which is averaged over time using a time based exponential decay. The dropping probability used by RED depends on this average queue size. There are two average queue size thresholds. One is min_{th}. Below a queue size of min_{th}, the dropping probability is set to zero. The other threshold is max_{th}. For average queue sizes above maxth the dropping probability is set to 1. For queue sizes between min_{th} and max_{th} the dropping probability varies linearly between 0 and one of RED's parameters max_p as shown in Figure 2.1.

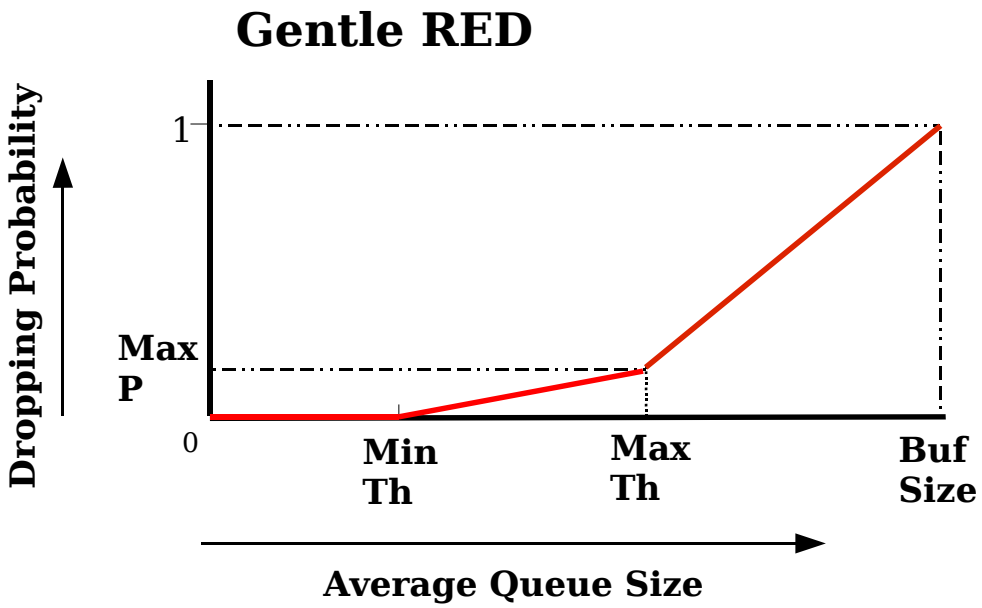

Figure 2.2: Gentle RED. When the average queue size exceeds the threshold, the dropping probability is linearly increased until it is 1 at the point where the queue is absolutely full.

A variation on the RED AQM known as *Gentle* RED [40, 43] has been defined. Gentle RED is the same as RED except that the probability of dropping a packet follows Figure 2.2 instead of Figure 2.1. The difference between these graphs is that in Figure 2.2 the probability of dropping a packet increases linearly from max_p at max_{th} to 1 at the point where the buffer is full.

Since the first paper on RED, many papers have been published on its problems, and many suggest improvements and modifications to RED. One of the problems with RED includes that of proper configuration. Improperly adjusted, the queue size of RED will typically be either empty or just under the point where the drop probability is 1. Traffic can be bursty, causing different congestion scenarios with regularity. The efficiency of RED depends on correct configuration [34, 42], and yet there is no single set of RED parameters that work well under different congestion scenarios [33]. This leads to oscillatory behaviour that leads to packet losses when the queue is full, and network underutilisation when the queue is empty [62, 39, 41].

This problem has been solved in different ways with different mechanisms. Some of these include Stabilised RED [79], BLUE [39] Loss Ratio based RED [81], and Adaptive RED (ARED) [41]. ARED involves using the original RED with the addition of a mechanism which constantly adjusts the most critical RED parameter max_p to achieve a target average queue length. It uses an additional measure of average queue length to that of RED, one which is more long term. If the long term average queue size is larger than the target queue size it makes the RED algorithm more aggressive by increasing max_p. Otherwise it makes the RED algorithm less aggressive by decreasing max_p [41]. ARED solves the problem of the oscillatory behaviour and the problem of the sensitivity of the algorithm to load [33, 57]. It is also much easier to deploy due to the fact that it requires less parameters to be set.

2.2 Fairness algorithms

In 1985 John Nagle produced an RFC entitled "On Packet Switches With Infinite Storage" which introduced the notion of fairness amongst flows [64]. He proposed a system of multiple queues serviced in a round robin fashion. He wrote "each source host should be able to obtain an equal fraction of the resources of each packet switch". He did not differentiate between large multi-user hosts and small hosts or IP masquerading. This idea has come to be known as *Fair Queueing* or *Processor Sharing*.

Most of the various mechanism proposed aim not to produce fairness amongst source hosts, but fairness amongst flows defined by the source and destination IP numbers and port numbers. These include Packet by Packet Generalised Processor Sharing [67], Stochastic Fairness Queueing [60], Approximate Fairness through Differential Dropping [66], Flow RED [58], Approximate Fairness Dropping [66], Stochastic Fair Blue [38], Choose and Keep [72] and Network Fair Bandwidth Share using Hash Rate Estimation

[54].

2.3 Mice and Elephants

"Mice and Elephants" and "Shortest Job First" strategies are ones which favour the short flows over long flows. In a mice and elephants strategy the short flows or the packets from them are called *mice*, and the long flows or the packets from them are called *elephants*. The strategy involves identifying flows and associating packets with their flows in order to be able to treat long flows differently to short flows. One way to favour the mice is to give the mice priority when dequeuing. Another is to avoid dropping mouse packets by dropping elephant packets instead. This can be achieved by dropping packets early, before the queue is full.

Proponents of "Mice and Elephants" queueing strategies argue that equal throughput for each flow or host (sometimes called "Processor Sharing" or "Fair Queueing") is the wrong goal. Mice and Elephants strategy relative response times are significantly better than those obtained using Fair Queueing.

The Shortest Remaining Processing Time (SRPT) strategy is one of giving preference to the flows that have the least bytes remaining. This requires prior knowledge about the length of flows. SRPT been shown to give better results than Processor Sharing for a range of measures including average task turnover time [36, 26, 76]. Harcho-Balter, Crovella and Park [46] uses mean task turnover time divided by job length as a measure of starvation, and show that starvation of the long jobs is not an issue in internet traffic. They show both analytically and by simulation that no class of jobs are worse off when the the job sizes are heavy tailed, as they are in internet traffic.

In reality, SRPT would be difficult in a queueing discipline, because we don't have prior knowledge of job lengths. We can only know the size of a job so far, and therefore can only implement Shortest Job First (SJF). But SJF has been shown to be a sufficiently good approximation to SRPT, to enjoy similar benefits over Processor Sharing that SRPT does. McNickle and Addie [61] show that shortest job first gives near optimal response time regardless of which group of flows we care to observe. For example, Shortest Job First gives as good a result to medium length jobs than if we were to give medium length jobs absolute priority. Simulation of an implementation of Shortest Job First is described in [17], with results that show significant gains over other strategies.

Two cases of congested queues fed by Poisson Pareto Burst Processes were mathematically modelled in [18]. One had a Pareto distribution shape parameter of 1.4 (heavy tails) and the other had a Pareto distribution shape parameter of 1.2 (very heavy tails). Both cases were modelled with a Mice and Elephants strategy and without. The benefit from the Mice and Elephants strategy was assessed by calculating the extra capacity needed when the Mice and Elephant strategy was not used in order that at most 5% of flows are delayed by more than 20%. In the heavy tails case, 16% more capacity was required. In

the very heavy tails case 40% more capacity was required. The modelling showed that the benefit of a mice and elephants strategy would be quite significant.

Long flows constitute a small minority of flows, but carry the vast majority of bytes. About 20% of the flows have more than 10 packets but these flows carry 85% of the total traffic [74, 32]. During periods of traffic congestion the long flows account for an even greater percentage of the traffic than they do if we take overall traffic measurements. During periods of high congestion, the proportion of bytes in long flows is even greater because the long flows play a greater role in causing congestion than the short flows. In [19] an example was given where the short flows accounted for 89% of the traffic flow and the long flows accounted for the other 11% of the traffic flow overall. accounted for a disproportionate amount of the traffic flow - perhaps 88%.

It stands to reason that interactive short flows are delay sensitive as far as the perceived quality of service is concerned, because a human being will have an active process happening and will be impatient to wait for a result from her mouse click or keystroke. For example, the keystrokes in a telnet session will have to wait in a queue congested by packets from long flows.

It is also worth mentioning that short flows are particularly sensitive to dropped packets [45]. Treating mice and elephants equally is not truly ”fair”, and it would be more fair to assist the mice in order to achieve a better perceived quality of service.

2.4 A Mice and Elephants Queueing Discipline

The Mice and Elephants Queueing Discipline constructed in this project benefits the Mice in two ways. The first way was to give absolute priority to the mice when dequeuing packets from the queueing discipline. This was most easily implemented by having two queues, one for the mice and one for the elephants.

The second way to benefit the mice is to drop elephant packets, but not mouse packets when it is necessary to drop packets. This becomes difficult if we are forced to drop packets because the queue is full, as normally happens in drop tail. We avoid this problem by using Random Early Dropping (RED). Hence we can drop a packet at any time, thus allowing us to drop elephant packets only.

It takes expertise and thought to configure RED and it is impossible to configure it to cope well for varying congestion scenarios. It is easy to provide an ARED with sensible defaults for most parameters so that it would be a relatively easy queueing discipline to configure and it automatically adjusts to changing load conditions. ARED was chosen over RED as a base for a Mice and Elephants queueing discipline for these reasons.

Also, because there was no currently existing freeware ARED queueing discipline for Linux this has given me the opportunity to provide one. As part of the evolution of the Mice and Elephants queueing discipline, an ARED queueing discipline was created first, as a useful queueing discipline in its own right.

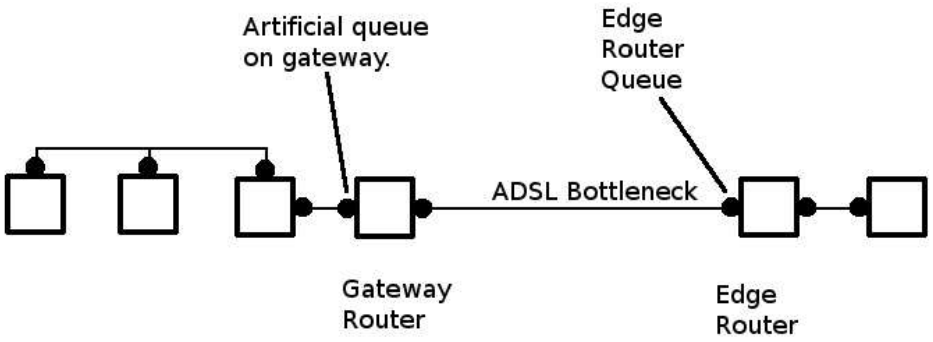

Figure 2.3: The Premises and the Gateway Router with an extra artificial queue.

2.5 A Mice and Elephants Ingress Filter

Routers in the core of the internet must operate fast in order to handle large volumes of traffic. Tracking flows requires CPU and memory resources, and a Mice and Elephants strategy will be more difficult to implement in the core, probably requiring special strategies to reduce complexity, at the very least.

The entrance to a residential or business premises, on the other hand, would be an ideal place to deploy a Mice and Elephants queueing discipline. The volume of traffic is lower and the number of flows will be lower. For this reason, my project focuses on the gateway router and its link to the edge router (See Figure 6.1.). This is where the bottleneck will typically be and therefore where improvements can be more easily made.

Typically in this situation, the flow of traffic is mostly in the *download* direction, coming from the internet into the premises. The queue for this downloading traffic across the link will be on the edge router. Sadly the owner of the premises will have no direct control over queue on the edge router. In [56] it has been demonstrated that we can deliberately create another bottleneck with (say) 90% of the bandwidth, and the effect will be that the queue on the real bottleneck will grow small enough not to be significant. It is then possible to use a mice and elephants queueing discipline on the new queue. Figure 2.3 shows the gateway router with our artificial queue. Figure 2.4 show results of NS2 network simulations. It shows the queue on the edge router for three different bandwidths of our artificial bottleneck. It shows that most of the queue has disappeared from the edge router when the bandwidth of the artificial bottleneck is 90% of the bandwidth of the link.

We can reason that the queue on our artificial bottleneck does not need to exist, and we should be able to get the same effect by merely slowing the traffic by means of a virtual queue. To this end, I have created an ingress filter that marks packets in an attempt to

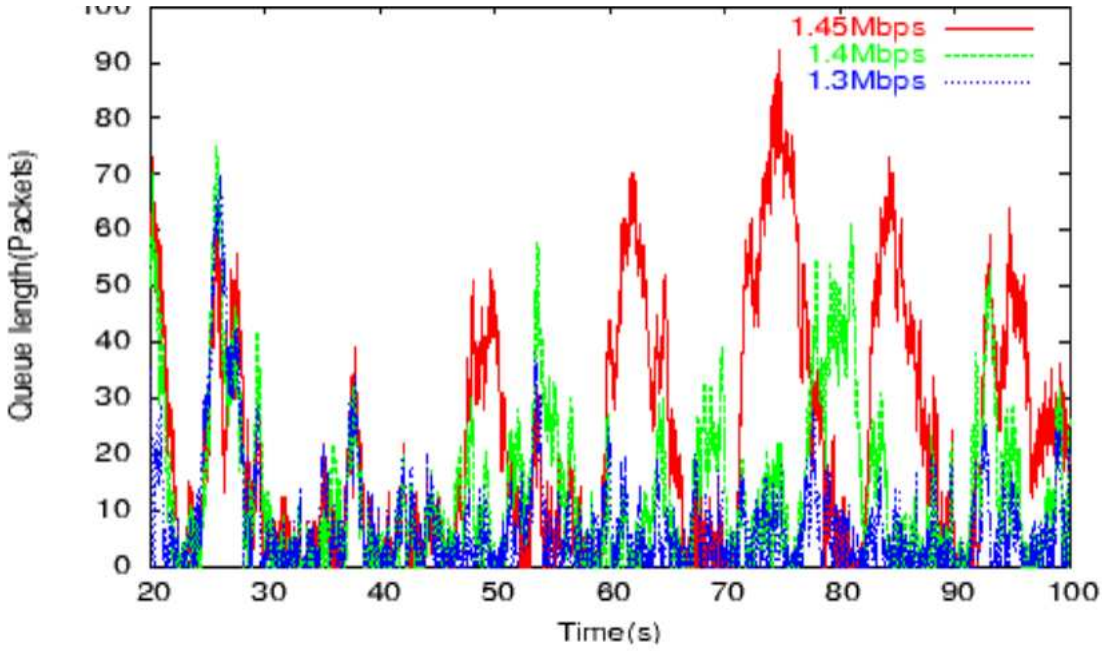

Figure 2.4: Graph, supplied by Zhi Li, of the instantaneous size of the queue on the edge router during simulations of 3 different artificial bottleneck speeds. Red shows results at at 95% of the speed of the real bottleneck. Green shows results at 90%. Blue shows results at at 85%.

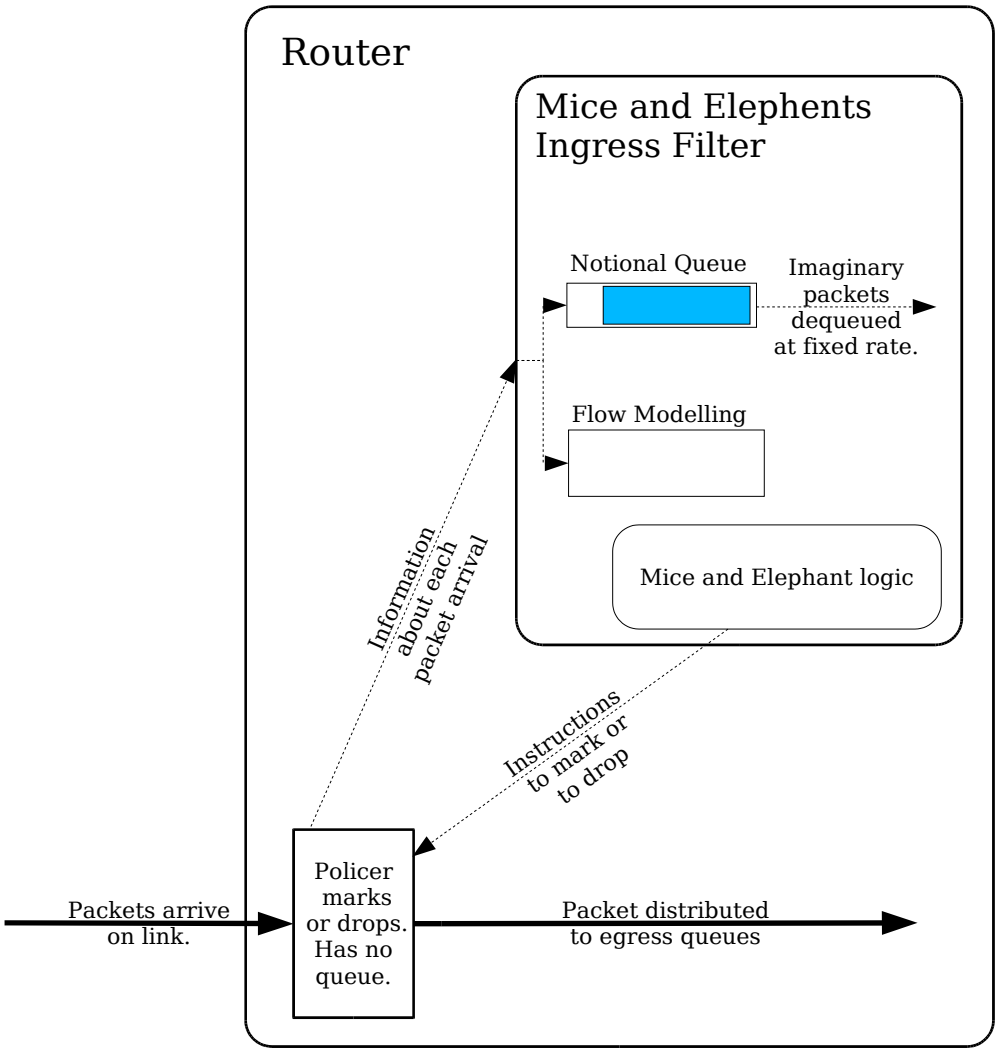

Figure 2.5: A Mice and Elephants Ingress Filter on Linux

limit traffic bandwidth to a fixed value. (See Figure 2.5.) It works in the same way as the simulations in [56], but has a notional queue instead of a real one. The notional queue holds no packets, but keeps the size of the notional queue in bytes. Bytes are dequeued from the notional queue at the target bandwidth given in the configuration of the ingress filter. ARED is used to vary the marking probability of the ingress filter, just as in the queueing disciplines to achieve a fixed notional queue size. Flows are monitored just as they are in the Mice and Elephants queueing discipline, and the same strategy as is used to mark the packets belonging to elephant flows, and not mouse flows.

One could argue that a 10% slowdown in bandwidth is too high a price to pay. In many circumstances that will be true, but most often the key performance indicator is response time, not bandwidth. If interactive flows are being crowded out by large downloads, then a 10% slowdown may well be not such high a price to pay, because the perceived quality of service will be increased, i.e. the quality of service would have improved.

Targeted packets can be marked using explicit congestion notification (ECN) [71] or the packets may be dropped. One reason that packets may be dropped instead of marked is that a flow may not support ECN. If the packets are dropped, one could make the argument that dropping packets after they have arrived, having successfully made it across the bottleneck is too high a price to pay. This issue is explored in ”Dropping Packets after they are Received: What are the Benefits” [55]. It shows the results of simulations and demonstrate that mouse flows enjoy a significant benefit from the use of an extra Mice and Elephants virtual queue added to the gateway side of the router. It also shows that having a Mice and Elephants queue deployed on the edge router is better, but not dramatically better than having a Mice and Elephants ingress filter deployed on the gateway router.

Chapter 3

Linux Queueing Disciplines

3.1 History of Net-4

Linus Torvolds announced that he had a working operating system in August 1991 [7, 1]. Implementation of IP handling was started 6 months later by Ross Biro [53]. Fred van Kempen started rewriting TCP/IP one year later. The project was called ”Net-2”. He was later assisted by Alan Cox, and the result was called ”Net-3”, and was available in Linux 1.0 [53]. Alex Kuznetsov wrote ”Net-4”. It was released with Linux 2.2 and is essentially what we have now in Linux 2.6 [22]. Net-4 has a flexible modular framework into which the system administrator may configure classifiers, policers and queueing disciplines for incoming or outgoing packets. Outgoing packets queue. Incoming packets do not.

3.2 Linux Networking Concepts - Classifiers, Policers, Queueing Disciplines and Filters

Figure 3.1 shows a simple router with two network interfaces. As packets enter the interface on the left they are subject to a *policer*. A policer in this context, has the power to drop or mark any packet that it chooses. It cannot, however, queue any packets. In this example, packets that are not dropped are delivered to the queueing discipline on the right.

Unlike policers, *queueing disciplines*, have one or more queues in which to queue packets. When a packet is delivered to a queueing discipline, it may drop, mark, or queue the packet. It may even choose to queue this packet, and drop another already queued. When a network interface is ready to send, the queueing discipline is asked for a packet. The queueing discipline will normally satisfy the request by dequeuing one packet. Figure 3.2 shows the flow of packets through a queueing discipline.

Queueing disciplines may be *classless* or *classfull*. A classfull queueing discipline is

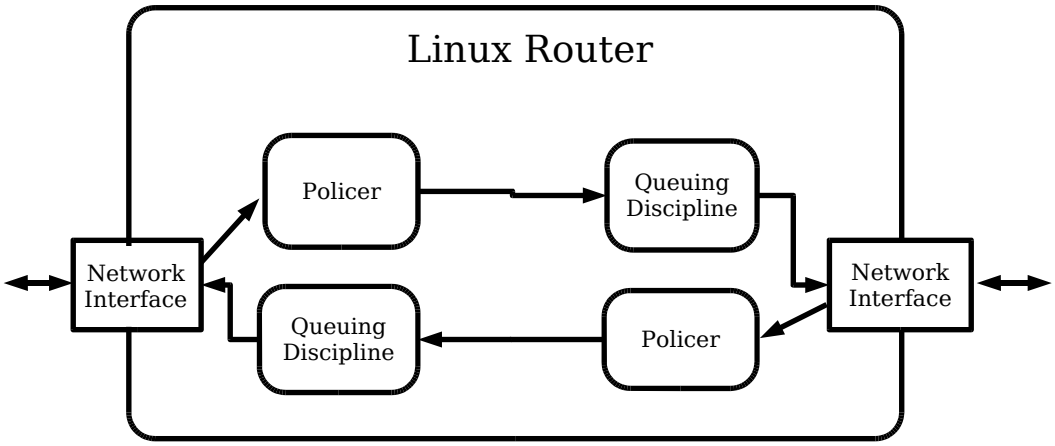

Figure 3.1: A Linux router with two network interfaces, each with a policer and a queueing discipline.

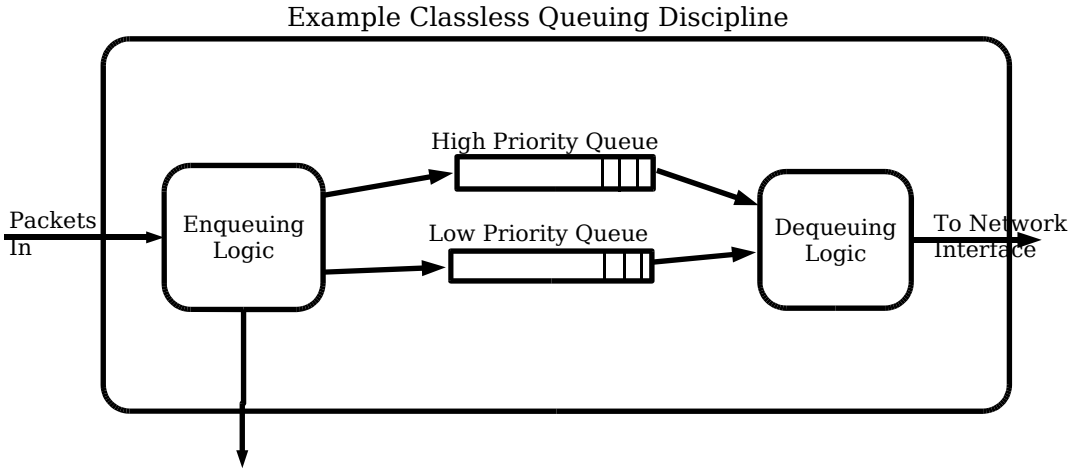

Figure 3.2: An example of a classless queueing discipline with two queues.

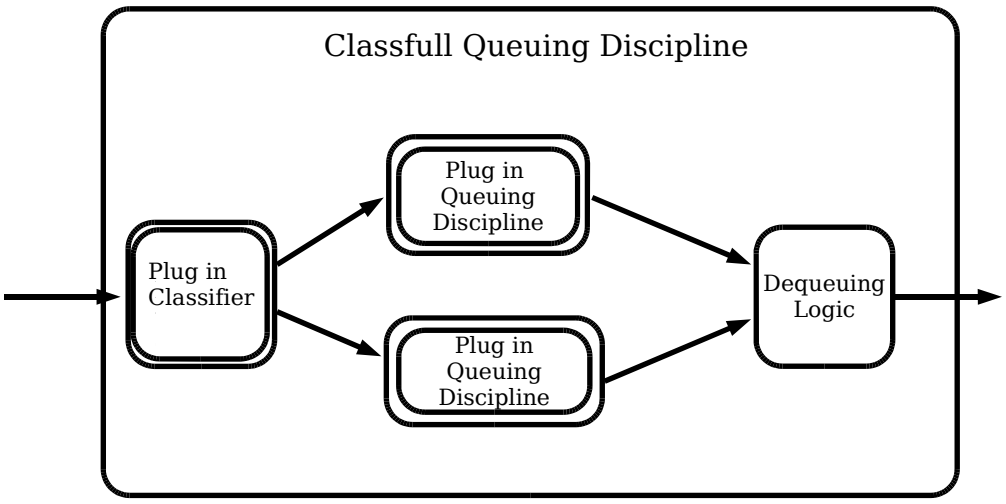

Figure 3.3: An example of a classfull queueing discipline with a classifier that directs traffic between two inner queueing disciplines. The inner queueing disciplines could be either classless or classfull.

one that is a container for child queueing disciplines and uses a *classifier* (alternatively called a *filter*) to decide which inner queueing discipline to send incoming packets to. Figure 3.3 shows the flow of packets through a classfull queueing discipline. The classifier and inner queueing disciplines of a classfull queueing discipline are specified as configuration [21].

In Linux, the policer is formed by combining a classifier with the the special *ingress* queueing discipline. Normal queueing disciplines are sometimes referred to as *egress* queueing disciplines because they deal with packet departures from a network interface. The so called ingress queueing discipline deals with packet arrivals. Figure 2.5 shows the flow of packets in a policer. Thus classifiers may be used to direct packets within classfull queueing disciplines, or they may be used decide when to drop packets. Policers are optional, and by default will not be used.

Queueing disciplines that manage packets without the use of child queueing discipline are called *classless* queueing disciplines. Eventually, an enqueued packet must reach a classless queueing discipline. Algorithms from the field of Active Queue Management, such as ”Drop Tail”, ”RED”, ”SFQ” are implemented in classless queueing disciplines.

To summarise, outgoing packets to an interface will go to a queueing discipline. This could be the default, which would be a simple drop tail queueing discipline, or the system administrator may have specified an advanced queueing discipline (for example SFQ), or it could be a classfull queueing discipline. Such a classfull queueing discipline would be a container for classes, filters (which will direct packets to classes), and ultimately queueing disciplines. These queueing disciplines may themselves be classless or classfull.

3.2.1 U32

An example of a classifier is ”U32”. It allows you to specify tests on packet headers. A packet that passes of all of those tests will be sent to a specified *class*, which may be an instance of a queueing discipline. The tests of U32 have two parts, - a mask which specifies what bits of the packet header get tested and a value that the tested bits must match in order to pass. This may seem to be rather limited, but in practice it is quite powerful, allowing the system administrator to send different types of service to different queues, or to treat ”syn” packets differently [15].

3.2.2 HTB

An example of a classfull queueing discipline is HTB (Hierarchical Token Bucket) [20]. It is a queueing discipline that allows you to define *classes*. Classifiers are used to determine which packets are enqueued to each class. A given class is called a *leaf* class if it is associated with a queueing discipline. Otherwise, the class may be associated with child classes.

Two properties associated with each class are the minimum and maximum bandwidth for dequeuing [30]. HTB attempts to satisfy the minimum dequeuing bandwidth first. It may not succeed if the interface (or the next level up class) is too slow. If it does not succeed, then it dequeues from each class in proportion to the minimum bandwidth of each class. If, on the other hand, it does satisfy the minimum bandwidths of each class, and there is yet more bandwidth available, it will allocate to each in proportion to the maximum bandwidths. It will not deliver more than the maximum bandwidth to any class.

3.3 Kernel Modules

Linux queueing disciplines and classifiers may be compiled directly into the kernel, but for development it is *much* more convenient to use a kernel module. One can compile a kernel module alone and insert it into the kernel for testing. One is then able to remove the module from the kernel for further modifications. If, on the other hand, you chose to compile into the kernel directly you would have to reboot the computer in order to try modifications. Kernel modules are also better for the system administrator because the modules that are not used will not be loaded, and therefore will not increase the size of the kernel. Loadable kernel modules make the framework for Linux queueing modules much more flexible. Loadable modules first appeared in Linux 1.2 in 1995 [47], but were greatly enhanced with in Linux 2.0 in 1996, with the introduction of automatic kernel loading and module dependency handling [8].

3.4 The Linux Queueing Discipline Interface

The Linux kernel has a well defined, but poorly documented interface for queueing disciplines. I have written a Linux man page in order to fully document the interface. This is included Appendix B.

3.5 Setting up a Linux Queueing Discipline Interface

The system administrator sets up, configures, and removes queueing disciplines from network interfaces using the linux *tc* command [10]. *tc* stands for Traffic Control. It is necessary to extend *tc* when providing a new queueing discipline. The use of *tc* is best explained in the ”Traffic Control HOWTO” [29]. A generously commented script using the *tc* command that I used for testing my queueing disciplines is provided in Appendix G.

It is generally agreed that the *tc* command is difficult to use [30]. The TCNG package provides a higher level language with which to control Linux queueing disciplines [23].

The *tc* command is the most important, however, and in the short term, TCNG is not likely to replace the *tc* command. The TCNG (TC Next Generation) package interprets the higher level language and issues *tc* commands to standard output. It is the *tc* command which still provides the low level interface to Linux queueing disciplines. The TCNG language also features an escape in order to directly issue *tc* commands, in case the system administrator wishes to perform some task which it is not possible to do using the higher level language, such as control a new queueing discipline.

Chapter 4

Implementation

4.1 Summary of Queueing Disciplines Developed

4.1.1 Drop Tail

As a first step in the implementation of Mice and Elephant queueing disciplines, I implemented the simplest possible Linux queueing discipline *drop tail*. The drop tail queueing discipline was created as learning step rather than an end goal in its own right. Creation of this most simple queueing discipline demonstrated running code that utilised Linux's queueing discipline framework. Later, the drop tail queueing discipline was modified and heavily commented to turn it into a template for the development of queueing disciplines, as a companion to the queueing discipline how-to, delivered as part of this project.

4.1.2 RED

This code was then re-used in the creation of a RED queueing discipline. As Linux already has a RED queueing discipline, this was not useful as an end deliverable, but it was useful in that it could be tested, and the code could then be re-used for the next step.

The RED algorithm is a mechanism which marks or drops packets before the queue is full. Its configurations defines two dropping thresholds min_{th} and max_{th}. When the average queue size is less than min_{th}, no packet dropping occurs. When the average queue size is between min_{th} and max_{th}, then a packet dropping probability applies as shown in the Figure 2.2. When the average queue size is greater than max_{th} then the dropping probability is applied as in Figure 2.2, depending on whether the gently variety of RED is being used.

The basic algorithm of RED, which follows, is directly quoted from [44]:-

for each packet arrival
    calculate the average queue size *avg*

if $min_{th} \leq avg < max_{th}$
    calculate probability p_a
    with probability p_a :
        mark the arriving packet
else if $max_{th} <= avg$
    mark the arriving packet

In order that the queue size that results from bursty traffic or from transient congestion does not result in a significant increase in the average queue size, the average queue size calculated by means of a weighted exponential running average of the queue length as follows :-

$$avg \leftarrow (1 - w_q)avg + w_q q$$

A time based weighted exponential should be updated at regular time intervals. For efficiency considerations, however, this is actually updated at each packet arrival. There would be a problem here if the queue has been idle. In this case our formula would not take into account the idle time at all. That is why the following formula is used instead if a packet arrives at an empty queue :-

$$m \leftarrow (time - qtime)/s$$

$$avg \leftarrow avg(1 - wq)^m$$

Where s is the average transmission time for a packet, $time$ is the current time, and $qtime$ is the start of the queue idle time.

The dropping probability p_a is calculated in two stages as follows:-

$$p_b \leftarrow max_p(avg - min_{th})/(max_{th} - min_{th})$$

Then the packet marking probability p_a is dependant on the count of packets since the last marked packet:-

$$p_a \leftarrow p_b/(1 - count\, p_b)$$

4.1.3 ARED

The next queueing discipline developed was an Adaptive RED (ARED) queueing discipline. Unlike RED, there is no ARED distributed with the Linux kernel. Therefore it is of significant value to provide an implementation of ARED for Linux.

A RED queueing discipline can be carefully configured for a given traffic condition, but if the traffic load then changes then RED will no longer correctly configured [34, 42, 33]. Incorrectly configured, RED can cause both network underutilisation and packet loss

[62, 39, 41]. Adaptive RED solves this by dynamically configuring RED's most critical configuration parameter.

The algorithm that achieves this is simple. The following definition of ARED is taken from Floyd, Gummadi and Shenker [41]:-

Every (interval) seconds:
    if (avg $>$ target $max_p \leq 0.5$)
        $max_p \leftarrow max_p + \alpha$ { increase max_p }
    elseif (avg $<$ target $max_p \geq 0.01$)
        $max_p \leftarrow max_p \times \beta$ { decrease max_p }

where

avg is average queue size

interval is the time between adjustments. 0.5 seconds is recommended.

target is the desired average queue length. Half way between min_{th} and max_{th} is recommended.

α is the amount to increase max_p by, and $min(0.01, max_p/4)$ is recommended.

β is the amount to multiply max_p by in order to decrease it. 0.9 is recommended.

A manual page for the ARED Linux queueing discipline is included in Appendix C.

4.1.4 Mice and Elephants

A Mice and Elephants queueing discipline associates packets with a flow, and these flows are classified as being small (mouse flows) or large (elephant flows) depending on whether the length in bytes (or possibly the length in packets) is smaller or larger than a given constant threshold.

The Mice and Elephant queueing discipline has two queues, an elephants queue for packets associated with elephant flows and a mouse queue for packets associated with mouse flows. Packets in the mouse queue are given absolute priority over packets in the elephants queue.

Early packet dropping (ARED) is used to indicate when packets may be dropped, but mice packets are given immunity to packet marking. If a mice packet is the target of an ARED marking decision, then the next elephant packet will be dropped instead. It should be clear that if we used drop tail instead of ARED was used as the basis of a Mice and Elephants queueing discipline then we would be forced to drop when the queue was full, and hence we would not be free to mark packets in a discriminatory fashion.

Two different Mice and Elephant queueing disciplines were produced. *mered* is simply as described above. *meredt* is configurable so that it behaves as mered does, but features a more flexible configuration so that it may be made to perform fair queueing, for example. Manual pages for these queueing disciplines are provided Appendix D and Appendix E.

4.1.5 Mice and Elephants Classifier

This contribution to Linux is aimed at incoming packets from a network interface rather than a queue for outgoing packets.

Taking the case of a typical domestic residence or small business, the bottleneck is the link entering the premises, as shown in Figure 1.1. Typically, most of the network traffic across the link is in the *download* direction, coming from the internet into the premises. Therefore, in order to control this downward traffic, it would be best to place a queueing discipline on the *edge router*, i.e. the router on the internet side of the link. Sadly the owner of the premises has little control over the edge router.

One way of controlling the traffic over this link is by means of a policer. There is no queue here, but a policer may selectively mark or drop packets entering an interface. Thus if deployed on the gateway, it can be used to control downward flows.

The Mice and Elephants classifier works with the Linux *ingress* queueing discipline to form a policer that will favour packets associated with mouse flows over packets associated with elephant flows. This policer works by simulating an artificial bottleneck, which is explained in section 2.5. If the overall bandwidth arriving from the interface exceeds a given threshold, then this policer attempts to reduce the flow to the threshold by marking or dropping packets. It is a Mice and Elephant policer, because it keeps flow records and prefers to drop packets from mouse flows rather than packets from elephant flows.

As explained in section 3.2, the policer can mark packets, drop packets or permit their passage, but it cannot retain or queue packets, so we cannot have a real queue here. Instead, we keep a virtual or imaginary queue, that holds no packets. It has only a number holding the queue size in bytes. For each real packet arrival, an imaginary packet of the same size is added to our virtual queue. At the same time, in order to simulate dequeueing, the size of the queue is decreased by the target bandwidth multiplied by the time since the last packet. Packets are marked or dropped as indicated by the ARED algorithm which will base its decisions on the size of the virtual queue, rather than the size of an actual queue.

A manual page for the Mice and Elephants classifier is provided in Appendix F.

4.2 Design Details

4.2.1 RED

The coding used in Linux's RED queueing discipline is, like much kernel code, highly optimised. There is not a direct one to one correspondence between the published RED algorithm [44] and the coding in Linux's RED queueing discipline. The programmers have found approximations to the published algorithm that are computationally faster. These same optimisations make Linux's RED difficult to understand and even more difficult to extend. Thus, it was considerably easier to write my own RED from scratch using the published algorithm.

The code in Linux's RED avoids floating point arithmetic within the Linux kernel, preferring to implement fixed point arithmetic using integers. I did not take that approach because I believe it to be outdated. Once, integer arithmetic was much faster than floating point arithmetic, but modern CPUs come equipped with fast floating point processors, and the Pentiums found in modern PCs are no exception [82, 80, 68]. On modern Pentiums an integer multiply takes 4/7 CPU cycles. That means it will take 4 cycles from the beginning of an integer multiply instruction till the next CPU instruction can begin, but the result of the operation wont be available until 7 CPU cycles. A floating point multiply takes 1/3 CPU cycles, which is at least twice as fast as the integer multiply [82, 80]. My approach was to not care about the CPU times for these operations, i.e. to use floating point or integer arithmetic wherever it was most convenient.

One of the creators of RED, Sally Floyd, has since recommended the gentle variety of RED [43]. I chose to implement the gentle variety of RED.

The published algorithm for RED suggests the use of a *uniform* random variable for intermarking times as opposed to *geometric* random variable for intermarking times [44]. By way of explanation, this means that it uses a random number generator to establish a count of packets until the next packet dropped, rather than apply a dropping probability to each individual packet. Floyd and Van Jacobsen claim that this is preferable as applying a dropping probability to each individual packet can result in too many dropped packets close together or alternatively, too long an interval between marked packets. Linux's RED queueing discipline uses this suggested uniform random variable, and my RED does also.

Smoothed Estimate of Queue Size

Ideally, the estimate of the queue size should be updated at regular time intervals. This would cause extra computational overhead, as this can be approximated in other ways. The RED algorithm [44] specifies a practical method of calculating the average queue size, and I have used this method with some improvements.

The average queue size is only updated when a packet arrives. If a packet is enqueued when the queue has at least one packet already in it, then the average queue size will be

updated by:-

$$avg \leftarrow (1-w)avg + w \times qsize$$

where *avg* is the average queue size, *qsize* is the current queue size and *w* is a constant which controls the weighting of past history verses current queue size.

If there are no packets in the queue on a packet arrival, then the average queue size will be calculated according to how many packets might normally have arrived in the period of time since the last packet arrival. Thus the following assignment is used :-

$$avg \leftarrow avg \times (1-w)^{(time-qtime)/s}$$

where *time* is the current time, *qtime* is the time of departure of the last packet, and *s* is the average transmission time. For convenience I will call the term used to multiply the average queue size the *average queue size multiplier*.

Floyd and Van Jacobson [44] recommend that the raising of power be achieved by means of a table. In Linux's RED, this table is implemented grudgingly. The following quote is from comments in the code of Linux's RED:-

> This is an apparently over-complicated solution (i.e. we have to precompute a table to make this calculation in reasonable time) I believe that a simpler model may be used here, but it is field for experiments.

I agree with the author's sentiments. I implemented that part by approximating the exponential curve with a quadratic equation over the relevant part of the curve. RED's algorithm for estimating the average queue size is only an approximation, and my quadratic fit is an approximation to that.

I use the further simplification that once sufficient time has passed and therefore the average queue size multiplier is close enough to zero, I do not do a calculation at all. I simply set the average queue size to zero. Let f be the size of the average queue multiplier at this point where I regard it as sufficiently close to zero.

Let x be the number of packets that might normally have arrived in the period of time since the last packet. Since we have to choose an arbitrary rate of packet arrival, x is synonymous with time. Let

$$v = 1 - w$$

Let x_0 be the number of packets such that

$$v^{x_0} = f$$

then

$$x_0 = \frac{ln(f)}{ln(v)}$$

My quadratic approximation has the form

$$y = a * x^2 + b * x + c$$

where y is the number to multiply the the new average by. In order to select a suitable quadratic approximation, I chose to select intuitive boundary conditions, which should ensure that it is accurate in the region of interest. These boundary conditions are sufficient to dictate the parameters of the quadratic approximation.

The first boundary condition is important: When no time has passed, then there is no reduction in the average queue size, i.e.

$$y(0) = 1$$

The next ensures that the curve is continuous. Recall that we use the simplification that we use 0 when x is greater than x_0 : The average queue size should be set to 0 at x_0, i.e.

$$y(x_0) = 0$$

The last prevents the curve from dipping below 0 while x is less than x_0, and give the most curvature: The curve should be flat at x_0 , i.e.

$$y'(x_0) = 0$$

From this I derived the coefficients of my quadratic equation. The quadratic equation became:-

$$y = \frac{1}{{x_0}^2} * x^2 - \frac{2}{x_0} * x + 1$$

I plotted the results for various values of f and by trial and error found that a value of f which keeps the approximation close to the exponential curve was $\frac{1}{15}$. Figure 4.2.1 shows my quadratic approximation together with the exponential curve as per the published RED algorithm for various values of v. It shows that the approximation is valid regardless of the value of v or w. In each of these graphs the two curves cross each other at $y = 0.2779$.

4.2.2 ARED

I directly implemented the algorithm as described by Floyd, Gummadi and Shenker [41]. I also implemented their suggestions for setting the parameters, so now most of my ARED parameters have good default values. There are only three parameter that a system administrator should set, and these do not require an understanding of ARED. These are the queue length, the Maximum Transmission Unit (MTU) and the speed of the interface. The last two parameters could potentially be discovered automatically, but that is work that I have not undertaken.

4.2.3 Mered - The First Mice and Elephants Queueing Discipline

The basic idea of a mice and elephants router is to preferentially treat mice packets in two ways. The first is to allow them absolute priority when dequeuing. This is achieved by

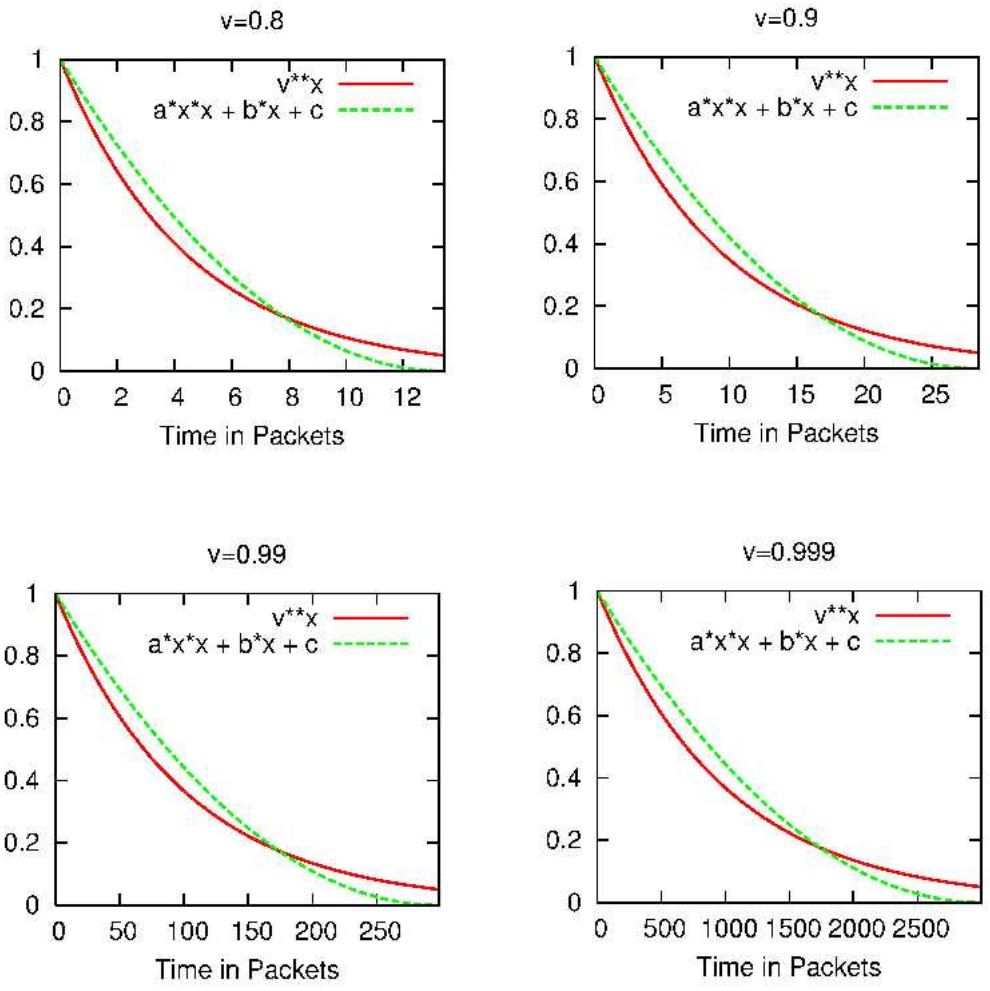

Figure 4.1: Here are graphs of the multiplier for the average size showing both raising to a power and my quadratic approximation. The graphs are for $v = 0.8$, $v = 0.9$, $v = 0.99$ and $v = 0.999$

having two queues. Packets in the queue that the mice packets go into will have absolute priority over packets in the Elephants queue.

The second way to preferentially treat mice packets is to attempt to give the mouse packets immunity from being dropped. As explained earlier I use a random number generator to establish a count of packets that will be accepted before dropping a packet. If the count is up, thus indicating that the current packet should be dropped, then I drop it only if it is an elephant packet. Dropping is suspended until either an elephant packet arrives or the queue gets too full to accept more packets. Hopefully the latter case is rare.

In order to preferentially treat mouse packets, I have to be able to tell between mouse packets and elephant packets. To do this I keep track of the flow for each packet and keep a count of the number of bytes of each flow. If a flow has had less than a given threshold of bytes then all packets belonging to that flow are classified as mouse packets. Otherwise packets belonging to that flow are classified as elephant packets. I keep not just one mouse/elephant threshold, but two. One is for classification as mice for the purpose of dropping and the other is for classification between mice and elephants for the purpose of queueing.

Hashing of Flow Records

In this Linux queueing discipline, flows are identified by some combination of source and destination IP numbers (as well as the source and destination port numbers in the case of TCP or UDP packets). These values are extracted from each packet, and then a mechanism is used to quickly find the records for the flow associated with the packet.

Any one of a range of mechanisms such as a tree or Bloom [27] filter or bucket mechanism similar to that used in SFQ [60]. I chose to use traditional hashing technique because it simple and also fast in execution. The IP and port numbers themselves are not kept in each flow record. Only the hashed value is kept. This reduces the storage space associated with each flow record and CPU time needed to compare the flow of the flow record with the flow of the packet.

For distinct flows to be distinct, it is desirable for the hashing function to be of good quality. A good quality hashing function will approximate uniform hashing [83, 65]. If we use uniform hashing or a hashing that approximates uniform hashing, the probability of packet misclassification will be tiny. The probability of any two flows having an equal 32 bit hash value will be will be the same as the probability of any two random 32 bit integers being equal, which is $\frac{1}{2^{32}}$. The probability that no two flows hash to the same value would be :-

$$\prod_{i=1}^{n-1}(\frac{N-i}{N})$$

where n is the number of flows and $N = 2^{32}$. This comes to 0.999999987 for 10 flows, 0.9999988 for 100 flows, 0.99988 for 1000 flows, and 0.988 for 10000 flows. Since the

unlikely event of packet misclassification would not be a disaster we need not worry.

I considered three separate hashing functions. The first hashing function, which I will call hash1 is based on one given in Kernighan and Ritchie [52]. In it, we set A to some initial starting value, and then we do the following for each byte of the key:-

$$A \leftarrow (A * m + ch)\ MOD\ M$$

where A is the hash value, m is some large number co-prime with M , ch is the next byte from our key and $M = 2^{32}$. The MOD operation happens automatically by integer overflow. My concern was that because this operation happened for each byte this might be slow. This has 4 add operations and 4 multiply operations when used on a 32 bit words. When used to distinguish flows, this has to be done at least 3 times, so that is 12 adds and 12 multiplies, 24 operations in total.

The second hashing function, which I will call hash2, is based on the one used in Linux's SFQ queueing discipline [60]. We set A to some arbitrary value, and then do the following for each four byte word of the key:-

$$A \leftarrow (A \gg 16 + A * m) \oplus b4\ MOD\ M$$

where A is the hash value, m is some large number co-prime with M , $b4$ is the next 4 byte word of our key and $M = 2^{32}$. The MOD operation happens automatically by integer overflow.

To put it in words, for each four bytes, the current hash value is recreated by adding a shifted right version to a multiplied version of the hash value. The result is then exclusive OR'd with the 4 byte value being incorporated into the hash. I use power of two modulus in my arithmetic for the sake of efficiency.

The third hash function considered was *lookup2* [50].

The data that I wanted to hash are the IP protocol, the source and destination IP and port numbers associated with a packet, i.e. several short pieces of data. In order to get some idea of the quality of hashing, I started with a data value of 0 and incremented by the value which I called the "stripe", in order to explore interesting data values. I used my hashing algorithms to obtain a hash value of each resulting data value. I then used the hash value to index an array by using the modulus operator and increment the array element whenever I got a hit. I plotted the hit counts verses the array index.

I would expect a good quality random number generator, and a good quality encryption algorithm would both give a very good approximations uniform hashing and I used them as a control to show me what these ought to look like under uniform hashing. I used blowfish encryption [73] for my encryption. For the random number generator I used the UNIX successor to the ANSI C random number generator *rand*, called *random* [14].

I experimented with different array sizes and different stripes, and found that the largest variations from uniform hashing occurred with an array size of a power of two.

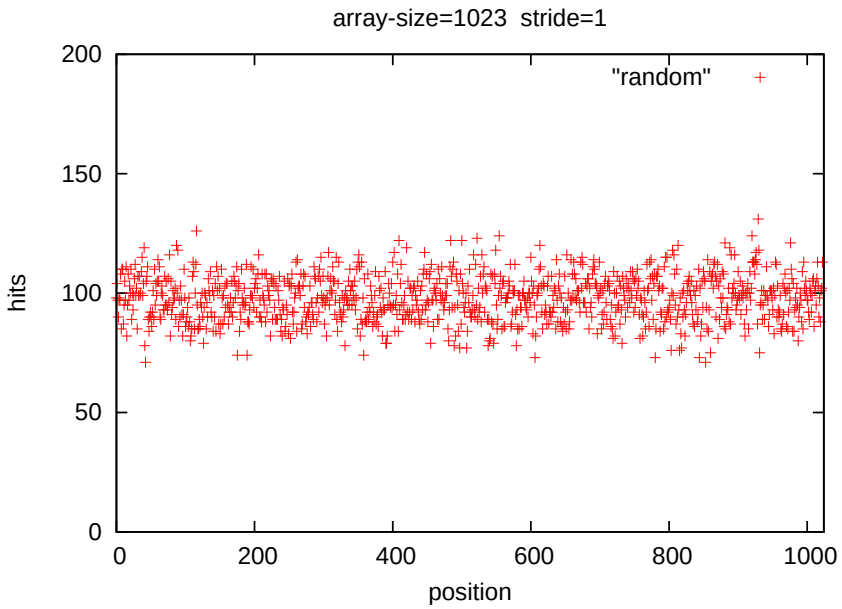

Figure 4.2: Hit count of hash table index. This plot was generated using the UNIX successor to the ANSI C random number generator *rand*, called *random*.

As having an array size of a power of two appears to be the worst case scenario, and the most interesting, I will present examples from this case.

Figure 4.3 and Figure 4.2 are provided as a reference to compare our hash functions to. Figure 4.4 shows the hit count of array indices for hash1. It shows a very smooth distribution, and a pattern is visible under closer inspection. It could potentially be a good hashing function, but it is clearly distinguishable from normal hashing. Figure 4.5 shows the hit count of array indices for hash1. Most hash indices have not been hit at all, and many that have been hit, have been hit 5 times more than the average hit rate. Figure 4.6 appears to be consistent with normal hashing.

These results seem to indicate that if random and blowfish are consistent with uniform hashing, then both hash1 and hash2 are inconsistent with normal hashing. Hash2, in particular would make a poor hashing function. Lookup2, on the other hand, appears to be consistent with normal hashing.

It is also important for the hashing to be efficient. My experimentation involved performing the hash as if on a flow key 100 million times on my 1200MHz AMD Duron Pentium PC. I used the hash function in-lined 10 times in the innermost loop to drown out looping overheads. Hash1 took 2.25 seconds. Hash2 took 0.723 seconds. Lookup2 took 1.35 seconds.

I used lookup2 because it was fast and consistent with normal hashing.

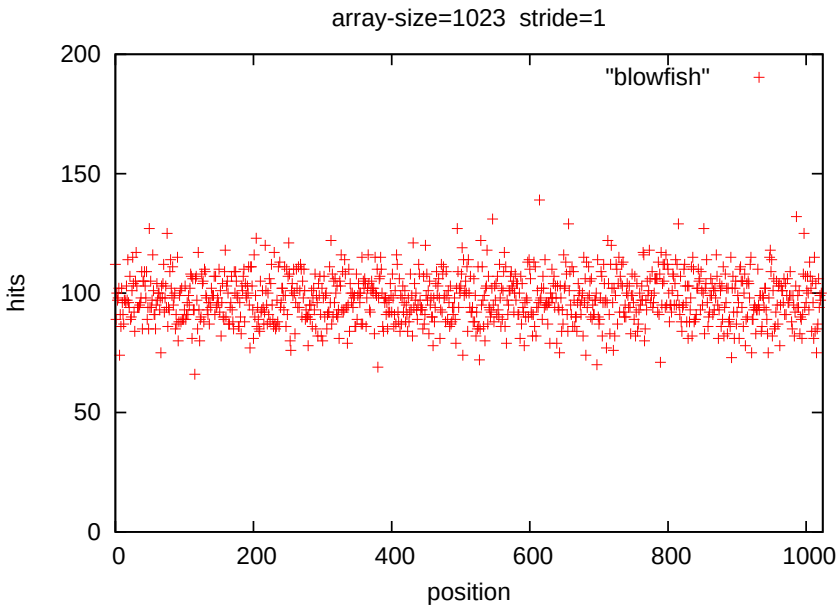

Figure 4.3: Hit count of hash table index. This plot was generated using the blowfish encryption algorithm as a hashing function.

4.2.4 Meredt - The Second Mice and Elephants Queueing Discipline

The creation of the first mice and elephants queueing discipline, *mered*, represented the achievement of a milestone in the evolution plan. A second mice and elephants queueing discipline *meredt* was created that would have the flexibility that is typically required of useful products. The additional functionality of *meredt* over and above that of *mered* is described below.

Flow Sizes and Queue Size Measured in Bytes

The published algorithm [44] allows RED to measure the size of the queue in packets or bytes. Measuring flows in packets places flows that use smaller packets such as interactive flows or voice over IP at a disadvantage because they prematurely register as elephants. Also, measuring the queue size in bytes allows the average queue size to more accurately reflect the average delay at the gateway. The first Mice and Elephants queueing discipline measured the queue size in packets. The second Mice and Elephants queueing discipline would permit the measurement of the queue size and the flows in bytes, but would allow a configurable packet overhead to be counted as well to account for the fact that there will always be delays between packets.

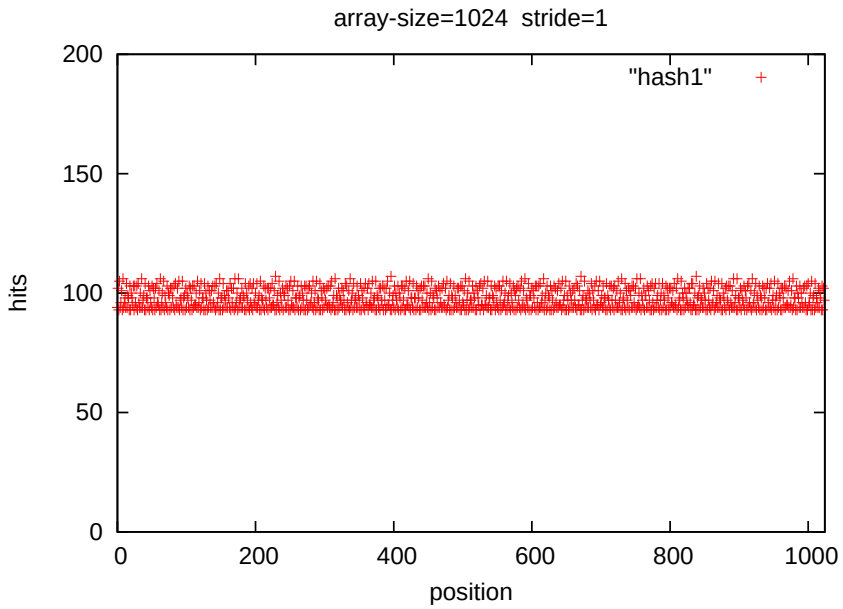

Figure 4.4: Hit count of hash table index. This plot was generated using hash1 as a hashing function.

Flexible Flow Weightings

In a domestic situation where there was an interactive telnet session competing for bandwidth with people using a graphical web browser, then the telnet session would constitute a mouse flow, and the web browser would constitute an elephant flow. Alternatively if the web browser was competing for bandwidth with a large download, then the web browser would constitute a mouse flow, and the download would constitute an elephant flow. In my mind at least, mice and elephants are relative descriptions, so by default the second Mice and Elephants queueing discipline allows flows to become elephants gradually, instead of having a single hard threshold.

The configuration optionally includes a piecewise linear function that maps actual flow size to flow length weighting. Thus, the system administrator need not accept the default behaviour.

An average flow length weighting is maintained, where the averaging is performed over all packets as they arrive using a similar method to that used to maintain an average queue size in RED, which might well be called exponential time decayed weighted average. The average dropping probability is provided by the ARED algorithm. This average dropping probability is multiplied by the flow length weighting and divided by the average flow length weighting to give the dropping probability actually applied. Thus

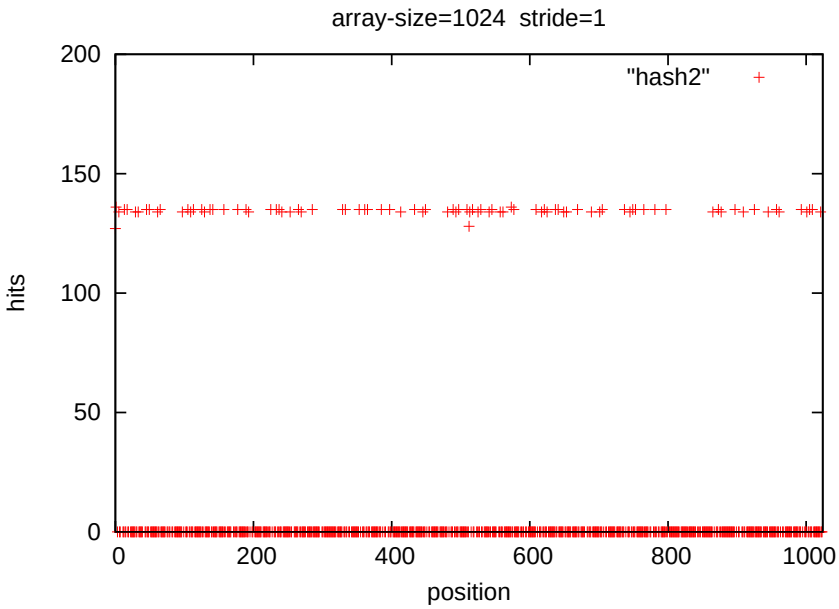

Figure 4.5: Hit count of hash table index. This plot was generated using hash2 as a hashing function.

the dropping probability of a packet will be proportional to the flow length weighting of its flow. This way, depending on the piecewise linear function given in the configuration, mouse flows are not subject to dropping but elephant flows are.

If the average dropping probability provided by ARED and the average flow length weighting are relatively stable, then it is easy to show that the average of the dropping probability actually applied is the average dropping probability provided by ARED. The applied dropping probability for the packet p_p is given by the following formula:-

$$p_p = p_r * \frac{w_p}{w_{av}}$$

where p_r is the dropping probability supplied by ARED, w_p is the flow length weighting for the packet, and w_{av} is the maintained average flow length weighting. Thus the average of the dropping probability applied is given by:-

$$E(p_p) = E(p_r * \frac{w_p}{w_{av}})$$

$$= \frac{p_r}{w_{av}} * E(w_p)$$

$$= p_r * \frac{w_{av}}{w_{av}}$$

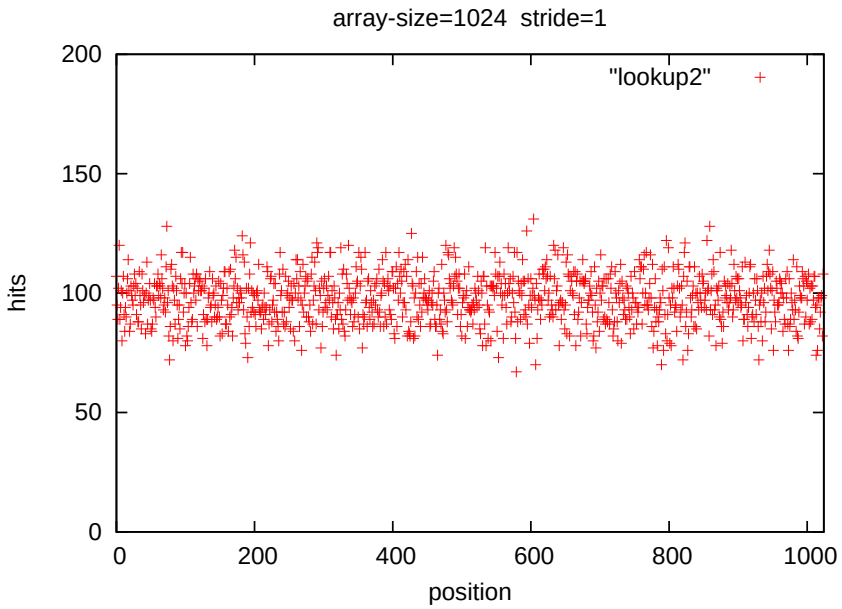

Figure 4.6: Hit count of hash table index. This plot was generated using lookup2 as a hashing function.

$$= p_r$$

There may well be occasions when the dropping probability for the packet p_p is calculated to be greater than 1. In this case, the packet will be marked or dropped, effectively making p_p equal to 1, thus breaking the relationship $E(p_p) = p_r$. Because this cannot happen in a sustained way, and so should be rare, the average dropping probability applied will still be approximately equal to the dropping probability provided by ARED,

It is worth noting that flexible flow length weightings allow the system administrator to implement a strategy that is more consistent with Shortest Job First than Mice and Elephants queueing.

Fair Queueing Hybrid

A mice and elephants queueing discipline classifies flows according to their length in bytes or packets. Once a flow has been classified as being large, it will always be regarded as a large flow. A fair queueing discipline, on the other hand classifies flows according to their current behaviour.

The system administrator may specify an optional fair queueing factor which will make the queueing discipline behave more like a fair queueing discipline. This is achieved by reducing the size of this queueing discipline's idea of individual flow lengths in bytes

by a value derived by dividing the link capacity by the total number of active flows and then multiplying by the fair queueing factor. In other words, a flows perceived size will slowly reduce if it is relatively inactive and thus the perceived size reflects the flow rate rather than its length. This factor is zero by default, so that by default the queueing discipline is a pure Mice and Elephants queueing discipline. If it is 1 or above, and the mice and elephants threshold is low, then this queueing discipline will provide fair queueing. If this factor is between 0 and 1, then this is a hybrid between fair queueing and mice and elephants.

Flexible Definition of Flows

The default definition of a flow is the TCP definition of a flow or TCP connection (i.e. source and destination IP numbers together with source and destination port numbers). Although there is no such thing as a UDP connection, the same attributes can be used to define a UDP flow. If the source port of a UDP packet is non zero then it indicates the port to which a reply should be addressed [69].

It is clear that the TCP definition of a flow is not always suitable. If we wished the definition of flows to be the computer to computer traffic, for example, we can do this by using only the source and destination IP numbers. Another example of a flow might be all the traffic associated with a human activity.

This queueing discipline optionally permits the system administrator to configure it to use the source IP address, the destination address, the source port number, the destination port number, the protocol or any combination of the above as the flow definition.

Flexibility in Marking or Dropping Packets

The default action is to drop targeted packets. This queueing discipline may optionally be configured to mark packets using explicit congestion notification or it may be configured to drop them. It may also be configured so that it will mark targeted packets if they are ECN capable and drop them if they are not.

4.2.5 Meredtf - The Mice and Elephants Ingress Filter

Recall from section 4.1.5 that the mice and elephants ingress filter can be used in the gateway router at the entrance to a premises to control the download traffic that normally dominates the link.

I initially created a very simple ”Hello World” type ingress filter. This was tested, and then made into a template for future ingress filters. Then I grafted most of the code of the second mice and elephants queueing discipline, *meredt* onto my template, and replaced code that is related to queueing of actual packets with that of a notional queue which dequeues virtual packets at a fixed rate given in the configuration.

4.3 Queueing Discipline Development Environments

If I run a program under Linux, I am running it under an environment which I will call ”user space”. I will call such a program a ”user space program” or just a ”process”.

There are various safeguards available to you when you run a program in user space. If your program accesses memory that it should not, the kernel stops your program at that point. You can even get the kernel to create an ”image” at that point for debugging. Your program is prevented from accidentally damaging the memory of other processes. Other exceptional conditions such as division by zero will cause a similar action. The allocated resources of a user space program such as computer memory will be automatically released by the kernel upon termination of the program. The user has the option of terminating a user space program. Also, it is easy to create and control diagnostic output from a user space program. In addition, debugging tools will let you open a user space program and allow you to inspect or to modify memory within it.

The Linux kernel is the program that provides this ”User Space”, but for the kernel itself there are no such safeguards and little in the way of debugging tools. If you run code directly in the Linux kernel, a pointer with a bad value is likely to crash your computer, if you are lucky. In this case you will have to perform a time consuming reboot of the computer. There is no saying what might happen if you are unlucky. Sometimes such errors have caused random writing to the hard drive. It is therefore almost essential to use a user mode environment in order to test AQM code as it is being developed.

There are a some publicly available schemes which may be employed to run queueing disciplines in user space and descriptions of them follow below:-

4.3.1 Tcsim

The traffic control simulator (TCSIM) [23] can be used to test a queueing discipline. TCSIM allows one to drive a queueing discipline under the control of a script. It may well be a very simple script that fires one packet per millisecond, or it may be more complicated and do things like have several packets arrive at once, for example. The script controls the enqueuing of packets as well as the dequeuing of packets [16].

The language for the script provides full control over the length of the packet and the contents of each packet. TCSIM has include files which provide offsets and widths for the fields of IP, TCP, UDP and ICMP. These make it easy to specify just what should go into a packet.

TCSIM provides control of the time that is perceived by the queueing discipline when it receives the packet. If the script says that packet A will arrive at time B, then the ”clock” will be set to B when the queueing discipline receives the packet A. It therefore possible for the script to effectively be the replay of a recording of the arrival of packets at a router.

TCSIM has a fairly comprehensive suite of output processing that includes filters and the generation of four different types of graphical plots.

A limitation of this scripting language is that it is not two way. TCSIM doesn't have the ability to respond to traffic. It has a "poll" command which dequeues packets, but the poll command does not return information to the script. Nor does TCSIM have the ability to simulate the receiving application's response to packets. TCSIM can only generate packets and monitor the effects in the queueing disciplines.

In reality there is a two way interaction, that is, the behaviour of a TCP source is dependent on the acknowledgements. The arrival of TCP/IP packets at a router is dependent on the history of each routers treatment of previous packets. Therefore, no pre-arrangement of packets will be a satisfactory simulation of TCP/IP traffic. Give that TCSIM can only generate pre-arranged sequence of packets, I would not recommend TCSIM for realistic traffic simulations.

TCSIM replicates a substantial quantity of the Linux network stack in order to call the queueing disciplines. In [24] the creator of TCSIM, Werner Almesberger offers the following justification:-

> Simulators usually implement an abstract model of the system they simulate. In the case of TCNG, this approach could lead to a simulator that contains the same mis-interpretations as the program being tested, so both would happily agree on incorrect results. To avoid this problem, TCSIM reduces the amount of abstraction needed by building the simulation environment from portions of the original traffic control code of the kernel, and the tc utility.

I used TCSIM and tried out some of the built in queueing disciplines. I found it relatively easy to accomplish. I also built and ran my mice and elephants queueing discipline in it. It was not so easy, however, to integrate my queueing discipline into it. There is a fairly large directory tree to compile, and the makefile did not properly clean away some libraries from an old compile. When I did understand enough about it to remove the old libraries, my queueing discipline integrated into TCSIM as it should according to the documentation. I regard TCSIM as a feasible way of testing queueing disciplines, once the developer has learned how to use this environment.

4.3.2 User Mode Linux

The Linux kernel has been ported so that it runs in user space, and the result has been called User Mode Linux (UML) [78]. Hardware calls for the hosted Linux operating system are implemented as system calls into the host operating system. UML allows one to debug any modifications to the Linux kernel in user space. The result is called User Mode Linux. Applications in general do not have to be ported or even recompiled in order to run under User Mode Linux.

UML initially started out as a kernel debugging tool. UML provides a safe platform to test queueing disciplines, or any other aspect of the Linux Kernel. UML soon found other uses [35] :-

- Hostile or insecure or potentially hostile or insecure applications can be monitored and run safely in a virtual operating system with its own file system.
- Users or applications can be allocated a set amount of system resources.
- You could use it to test Debian verses Redhat all on the one machine without re-booting.
- Although this uses Linux as the host machine, it would be relatively easy to port this virtual operating system to run natively under other operating systems. It is relatively easy to run UML on other Unix systems, so far this has not been run under Windows.
- One of the popular uses for UML is for web hosters. Web hosters can now lease virtual machines complete with the root password. Examples of this are :- [11] [4] [2] [6]
- Virtual local area networks can be set up and prototyped.

I have not installed UML, let alone attempted to use it to develop Linux queueing disciplines. I instinctively felt that this would probably be a project in itself, and my time constraints would not permit this.

4.3.3 Umlsim

With UML, it is possible to set up virtual networks and try them out. It is also possible to perform user space debugging on the kernel. UMLSim [13, 24] takes this further by providing a language that specialises in debugging processes and in specifying and setting up network experiments. It modifies UML in order to enable debugging and to control the Linux's idea of the time.

UMLSim has a very powerful language which features user defined routines, types and objects. It also has the capability to debug programs set breakpoints in programs, examine variables, force function calls or returns, all in an automated fashion as part of the language. UMLSim places the perceived time of the UML instances under control of the language. The language can disable the normal incrementation of time, and set the time explicitly itself as it needs. It can quickly create virtual Linux instances on a real machine, and carry out experiments.

Despite richness of this AQM simulation environment, it concerned me that traffic simulations are one way only. Return "ack" traffic is not simulated as a part of this. It would not be easy to monitor "real" traffic generated by applications in the way that one could using UML alone.

UMLSim is powerful, but raw. The documentation is insufficient to understand it easily. In the words of the author, "UMLSIM today is clearly a hacker's toy." ... "Also,

as befits a hacker's toy, documentation is incoherent and spotty" [24]. As such I did not think that my project was big enough that I could afford to spend enough time learning about this environment to use it profitably. In a large project, I suspect that the ability to set up live traffic in the way that one could with UML alone, would make the use of UML a better option.

4.3.4 Linquede

Ultimately, I used a fairly simple scheme in order to test my AQM code in user space. The AQM code is linked in two different ways by two different makefiles. One of them compiles and links the queueing disciplines into a Linux kernel module that can be run live in the Linux kernel. The other links into a user space test harness that drives the queueing disciplines according to data provided in the form of ASCII text files. This scheme grew into a small queueing discipline development environment, which I called Linquede.

In addition to the linking and running in user space, Linquede provides a well documented queueing discipline template that after installation and running the setup script will compile in both user space and as a Linux kernel module. One could describe it as a "Hello World" queueing discipline. It is a simple drop tail queueing discipline ready for a new queueing discipline developer to extend.

Instructions for setting it up are provided in a HOWTO together with instructions on developing Queueing disciplines for Linux in general. Documentation of the API (programming interface) for Linux queueing disciplines is scarce and spotty. I have documented the this interface in the form of a UNIX manual page.

Unlike TCSIM and UMLSim, Linquede does not reproduce Linux's network stack, or the whole kernel, but takes a much shallower approach. Linquede's data files are actually recordings of calls to the queueing discipline's API made from real networking. As the driver program runs, it replays the calls in its data file, thus running your queueing discipline in user space.

Linquede is also shallow in another sense. The only library routines and declarations reproduced in Linquede are those needed for the queueing disciplines that I actually developed. If you need more, then the easy way to make them is to copy them from the kernel code and modify them to suit.

4.3.5 Comparison of AQM testing environments

Linquede's weak point is that it has very limited functionality. It has no features for simulation, and no scripting language. It is only good for running and debugging a queueing discipline.

Linquede is simple, and easy to use. This is Linquede's only strong point, but this alone is enough. My opinion is that the effort expended in learning Linquede is minimal

and will easily pay off in the short term.

Of the other potential testing environments, only TCSIM seemed simple enough that the benefits obtained in developing queueing disciplines over the course of a relatively small project might outweigh the effort expended in learning it. From my point of view, though, TCSIM did not offer anything that I needed that I could not already do. TCSIM gives control over packet arrival times and packet contents under the control of a script. The input file of Linquede has packet times and packet sizes and attributes for each packet, which means that I may fire packets at will according to my desire.

There are other good reasons to use UML other than the development of queueing disciplines. If it so happened that other reasons for learning UML applied to you, or it happened that you were already using UML, then UML could be the best option. Also, if you wanted the ability to run live application traffic and monitor and debug it, then the use of UML would become a very attractive option.

4.4 Platform

Time in the Linux kernel is held and calculated in microseconds, but the kernels idea of time is in reality much courser than that. A timer interrupt every microsecond would cause a CPU overhead that would be too high.

In the Linux 2.4 kernel, the constant HZ is defined in the CPU specific header files [25]. This constant defines how often the timer hardware interrupts the CPU, and hence how fine the time granularity is. It defines how accurate a queueing disciplines knowledge of time passed since the previous packet can be. It is 100 by default for Pentium CPUs, meaning that timing can be accurate to $1/100$th of a second. Brave kernel programmers can alter this value, carefully altering other code as required to compensate. A queueing discipline written for any machine, on the other hand, must not assume or alter such a basic constant. In the Linux 2.6 kernel it is set to 1000 by default for Pentium CPUs [3].

I originally developed my queueing disciplines under Linux 2.4, until it became clear to me that the finer grained timing that was available under Linux 2.6 was of importance for queueing disciplines in general. In particular, it was important to RED [44] as the time between packet arrivals plays a vital role. I ported my queueing disciplines from Linux 2.4 to Linux 2.6 and completed the implementation and testing on Linux 2.6..

Chapter 5

Testing

5.1 Load Testing

One of the aims of this project is to make a contribution to Linux by making my implementation as code that is both fit for distribution with Linux and useful in a small business or domestic setting. The load tests were performed in order to determine if performance would be satisfactory for a gateway to a small business or domestic residence. I chose a machine with modest specifications, a Pentium 450 with 128Mb of RAM, to run the queueing disciplines for the purpose of load testing. I used IPERF [5] to generate, receive and measure traffic.

IPERF cannot control the bandwidth of TCP flows. If you need to control the bandwidth of flows with IPERF, then you need to use UDP flows. In doing so, however, IPERF uses the entire CPU bandwidth. For the testing of the queueing discipline, the traffic had to be generated on the test machine and received on an another machine. I therefore couldn't control the bandwidth using features of IPERF, so I used TCP flows, and used HTB (described in 3.2.2) to control the bandwidth. For the ingress filter, I used features of IPERF to control the bandwidth of UDP flows, which was satisfactory as this occurred on the other machine and did not contribute to the CPU load on the test machine.

Table 5.1 shows the CPU loads for the queueing discipline load tests. Mb is used to indicate million bits per second. CPU loads are averages and are shown as a percentage.

Bandwidth	CPU Load for system default qdisc	CPU load for meredt qdisc
10M	5	5
20M	7	9
50M	9	14
100M	30	32

Table 5.1: Percentage of CPU load for queuing disciplines on a Pentium 450

Bandwidth	CPU Load for no ingress filter	CPU load for Meredtf ingress filter
10M	9	11
20M	12	18
50M	22	41
100M	37	70

Table 5.2: Percentage of CPU load for ingress filter on a Pentium 450

Table 5.2 shows the CPU loads for the ingress filter load tests. Mb is used to indicate million bits per second. CPU loads are averages and are shown as a percentage.

5.2 Test Harness

I set up a test harness that allowed me perform debugging, and scenario testing with real TCP flows. This test harness consists of programs to generate traffic and to receive it, as well as a network arrangement that serves as an artificial bottleneck and traffic switch.

One part of this test harness is a program created for this project that generates TCP flows whose flow starting times are random, according to a Poisson distribution, and whose flow lengths are random, according to a Pareto distribution. Parameters to this program include the average time between flow starts, average flow lengths and the Pareto shape parameter. Connection times and response times are are included in the output of this program. I created a server to work with this program that accepts TCP connections and reads in the data from them.

In order to create a bottleneck for testing, and yet still be able to comfortably transfer software and data to and from my test machine, I used the Hierarchical Token Bucket (HTB) [9] [30] (described in 3.2.2) to dedicate just some of my available bandwidth to traffic testing. It allowed me to limit the traffic bandwidth to any given queueing discipline to a nominated value. I chose 56Kbit/second. It also allowed me to use the port numbers as a means of directing traffic flows to a chosen queuing discipline. The script used to perform this is provided in Appendix G.

5.3 Three Experimental Scenarios

As part of the project, three scenarios were explored by means of experiments. I do not make any claims or inferences about the real life performance benefits obtainable using and Elephants queuing disciplines on the basis of these experiments. These experiments

are contrived, and I acknowledge that running scenarios in a test harness is not a substitute for testing the queuing disciplines in real life. Real life testing is beyond the scope of this project and is a good candidate for further work. These experiments are intended to only demonstrate the benefit that might be *potentially* obtained by using mice and elephant queuing disciplines.

The experiments were performed using the test harness described in Section 5.2. The link speed used in the experiment was the maximum modem dialup speed of 56Kbit/second. The measurement of connection times for all started flows and response times for all completed flows was calculated in the traffic generating software as described in Section 5.2.

Each experiment involved mouse flows which took the form of TCP uploads. The average flow length used was 5000 bytes, and were initiated at an average rate of of one mouse flow per second.

Two of the experiments involved the use of an elephant flow, which took the form of a TCP upload that was large enough so that it did not end during the course of the experiment. Elephant TCP flows normally use the entire available bandwidth [59]. Hence, a single elephant flow causes just as much congestion as two or more elephant flows and therefore there is no need for experiments with two or more elephant flows.

5.3.1 The *Mice Only* Scenario

In this experiment the mouse flows were allowed to run with no elephant flow to cause congestion. The queueing discipline used was a drop tail queueing discipline.

5.3.2 The *Congested* Scenario

In this experiment the drop tail queueing discipline was retained, and an elephant flow was added.

TCP connections take time to connect on a congested line, and I was unable to get the mouse flows connections to begin on time. Thus I did not succeed in starting one mouse flow per second. In 5 minutes I was only able to complete only 48 mouse flows.

5.3.3 The *Mice and Elephants* Scenario

In this experiment the elephant flow was retained and the mice and elephant queuing discipline *mered* was used instead of drop tail. The threshold between mice flows and elephant flows was set to 20000 bytes.

5.3.4 Results

The results of the experiments are shown in Table 5.3.

Scenario	Mice Only	Congested	Mice and Elephants
Average connection time	0.68	6.23	0.87
Average response time in seconds	1.85	18.37	3.20

Table 5.3: Average response and connection times experienced by mouse flows for experimental scenarios

The performance degradation on the mouse flows caused by the addition of the elephant flow was demonstrated by the difference in results between the *mice only* scenario and the *congested* scenario. Despite being unable to start all of the mouse flows in the congested scenario, the average response time degraded from 1.85 seconds to 18.37 seconds. Also, the average connection time degraded from 0.68 seconds to 6.23 seconds. The fact that I was unable to start all of the mouse flows further demonstrates how sensitive mouse flows can be to the congestion caused by elephant flows.

The potential benefit that could be obtained by using a Mice and Elephant queuing discipline is demonstrated by the dramatic improvement of the response and connection times. The average response time improved from 18.37 seconds to 3.20 seconds, and the average connection time improved from 6.23 seconds to 0.87 seconds.

The response times obtained is only for flows that completed during the course of the experiment. The response times shown include the connection time, but due to technical difficulties involved in making the measurements, do not include the TCP connection closing time.

Chapter 6

Deployment and Configuration

This chapter explores an idea for estimating the size of the queue on the edge router by sending ping or timestamp packets to the edge router and using the timing estimation to estimate. I did not have success with this method using my service provider, and so I discontinued my endeavour. This does not necessarily mean that the ideas are wrong, however, and it may well be worth exploring them further.

6.1 Congestion on the Edge Router

As shown in Figure 6.1, the flow of traffic across the link into a residence is in the downward direction (from the internet) in the majority of cases. The bottleneck, in most cases, will be the link between the edge router and the gateway router, and therefore packets will accumulate on the egress queue on the edge router. If we wished to deploy a Mice and Elephants queueing discipline in order to control this traffic, then the best place to place the queueing discipline is therefore on the edge router. Traffic destined for the premises will accumulate on the egress queue of the edge router, where it will be controlled by a Mice and Elephants queueing discipline. Packets will be dropped, or marked according to the algorithm which uses the size of the egress queue to control overall dropping probability.

Sadly however, the owners of the premises is unlikely to have control over the edge router, and in most cases will have to accept whatever the service provider uses.

6.2 Dropping Packets at the Gateway Router

With further thought it is clear that for the purpose of dropping packets it doesn't really matter where packets are dropped or marked from, if only the dropping router could know the current size of the queue on the edge router. In the case of packet marking using Explicit Congestion Notification (ECN), there will be no difference in the bottleneck

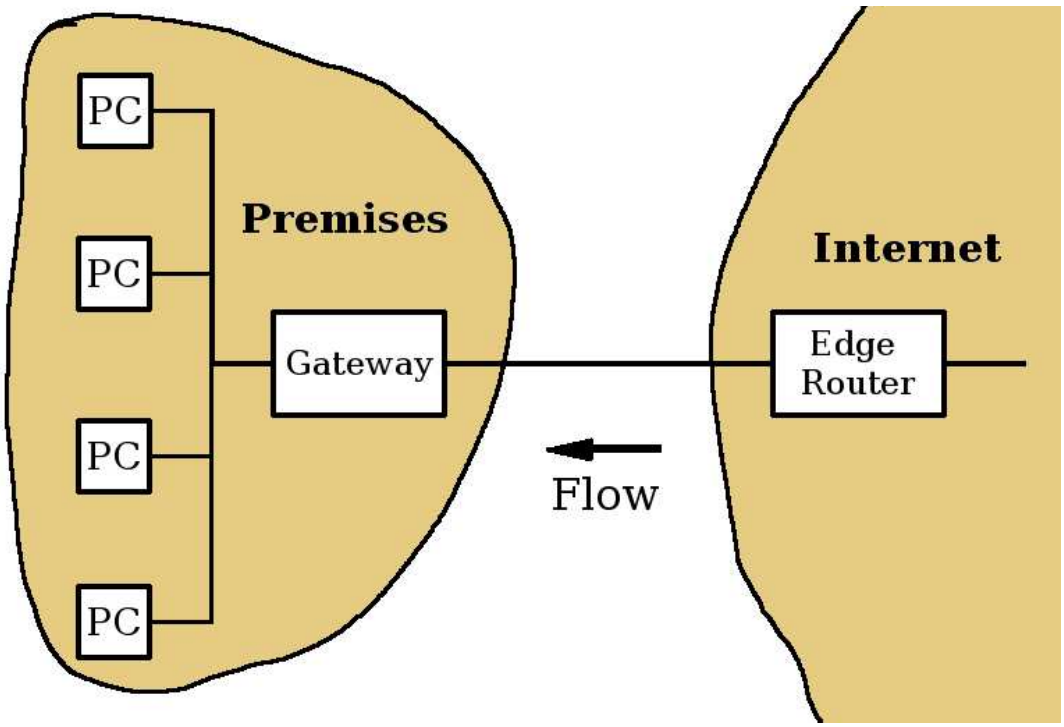

Figure 6.1: The Premises and the Gateway Router

usage regardless of where the packet was marked. In the case of packet dropping there will be a wasting of resource as packets are dropped after they have crossed the bottleneck. This wastage of resources, however, has been shown to be not so important as it might seem. As shown in [55], it is more important that responsive flows are notified of the congestion.

6.3 Edge Router Buffer Level Estimation

We have to ask whether it would it be possible to estimate the size of the queue on the edge router by sending ping packets to it. It will depend on just how the edge router handled these packets. If ping packets were made to queue then the round trip time would reflect the size of the queue, and thus it will be possible to estimate the queue size. If, on the other hand, ping packets were acted on at a low level of the TCP protocol layer they may not end out being queued at all.

Even better than using ping packets would be the use of the ICMP timestamp message [70, 12]. The ICMP timestamp message has space allocated for an originate timestamp, a receive timestamp and a transmit timestamp. It is intended that the source of the packet will supply the originate timestamp. The targeted router is supposed to supply the receive timestamp when it receives the packet and supply the transmit timestamp just before the packet is sent back to the source. If the ICMP timestamp message is implemented on the edge router, and the reply will be timestamped with the edge router's receive time and its transmit time. The difference between these two times will be the queueing time. The implementation of the ICMP timestamp message is optional, however [70].

The size of the queue in packets or bytes will not need to be calculated using this approach. Queueing time alone will be an adequate measure, once we found out how large that delay will get before packets were dropped.

Of course this estimate of the size of the queue on the edge router will be always a little out of date. The most recent information that we would be able to obtain is what the size of the queue was when the packet arrived. If we say that this was at T milliseconds ago then our information will be at best T milliseconds out of date. If we always had about N timestamp packets in the queue, then at the point just before receiving a timestamp packet, our information will be T(1+1/N) milliseconds out of date. Hence there is a clear case of diminishing returns as N is increased. It seems reasonable to always have somewhere in the range of 1 to 4 timestamp packets in the queue.

The timestamp packet traffic should not use up too much of the bandwidth. Up to 5% of the bandwidth may well be acceptable. The length of of the timestamp ICMP packet is 20 bytes [70]. The length of the IP header is also 20 bytes [70]. Assuming 20 bytes for ADSL framing overhead, we can estimate 60 bytes as the cost of a ping or timestamp packet. Typical TCP packets for a bulk transfer ought to be about the the size of the MTU, (1500 bytes for ADSL). Therefore we could put approximately 25 timestamp packets into

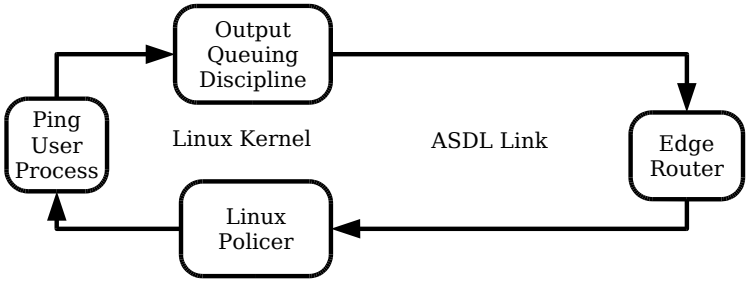

Figure 6.2: Journey of a Timestamp Packet

the transmission space of one bulk transfer TCP packet.

Therefore, if we had a queue just 4 * 1500 bytes long and 4 timestamp packets in the queue, then our timestamp traffic will consume 5% of the bandwidth. This is acceptable, because it meets the 5% criterion. In a much more likely case, if we always had 4 timestamp packets in the queue and the queue was 50 MTU's long, then we will consume 0.4% of the bandwidth, which is negligible. We can therefore comfortably aim to always have about 4 timestamp packets in the queue. It should be acceptable to send a packet every T/4 milliseconds. A smaller queue will allow faster updated estimates of the queue size, but the relative size of smaller queues will vary faster and hence require faster updated information.

I hoped to create a system as depicted in Figure 6.2.

The architecture of the networking system of the kernel has no way for queueing disciplines to talk to each other or to generate packets without modifications that I do not have the experience to undertake. Therefore a user process was to be responsible for generating ping or timestamp packets at intervals according to feedback it will receive from these same packets on their return. The output queueing discipline will give absolute priority to these packets in order to avoid queueing delays at the gateway and set the originate timestamp in them.

The information provided by the edge router will be noticed by the Linux policer on the gateway. It will deduce the size of the queue on the edge router and drop packets according to a Mice and Elephants queueing strategy. The timestamp packet itself will be immune from dropping, and it will continue on its way to our user space ping program. The user space ping program will use the information from the reply to form an estimate of the queue size in the same way as the policer did, and use that information to set its interval for packet generation.

6.4 Testing the Idea

I used my Optusnet broadband ADSL service as a means to test this out. I fashioned a crude program to send and receive ping or timestamp ICMP packets, by copying and modifying code from Stevens [77]. Of course a ping program already exists without me having to write it, but having my program work sending and receiving ping packets gives me confidence with sending and receiving the timestamp packets.

Sadly, when I sent my timestamp packets the actual timestamp data returned from the edge router did not seem to show a delay. The receive timestamp was equal to the transmission timestamp, even when there was a delay due to large downloads. This indicates that the timestamp packet was not handled.

I sent ICMP ping packets at the edge router while downloading 0, 1 and 3 simultaneous large downloads. I found that I obtained delays of approximately 70ms +- 8ms when I pinged the edge router in the absence of traffic. This is consistent with my expectations. My ADSL speed was limited to 20 to 28 Kbps. Frame sizes were expected to be 60 (IP packet size) + 20 (guessed framing overhead), i.e. 80 bytes. Delay one way was estimated to be 26 ms. Round trip was estimated to be 52 ms. This is not far out from the experimental 60 to 80 ms.

Then I measured my delays while performing a large download. I used a web browser to initialise a large download. The download manager reported 4 kilobytes per second during a single large download. This corresponds roughly to 32 kilobits per second. Optusnet say that I should get between 20 to 28 kilobits of download speed, so I will guess use the speed of 20 kilobits per second. The size of IP packets was determined to be 1480 bytes by means of tcpdump. Each packet should have taken 1480 * 8 / 28000, i.e. 422 milli seconds. With a single large download, I got pinging delays that ranged from roughly 70ms to 500ms. There were many delays of only 70ms, which seems to indicate that the queue was most often empty. The usual pattern was about 3 delays of 70ms followed by a single large delay. There was never a delay more than a single MTU sized packet.

I attempted my pings while performing 3 large downloads simultaneously. the result was 3 longer delays in the range 100 to 500 ms followed by 1 short delay of about 70ms. This indicates that the queue was empty 1/4 of the time, and when it wasn't it had only a single packet in the queue, or else it indicates the ping packets had priority.

I can think of three possible explanations for this. Perhaps there is only a very small queue indeed on the edge router that can have at most one MTU sized packet, or perhaps the ping packets were given priority and the delays were only due to not being able to transmit until an MTU sized packet finished downloading. The other possibility is that my ADSL broadband link is not the bottleneck.

I decided that I could not achieve anything useful with my current setup regardless of which possibility was the correct one. Given time and money it could well have been worth experimenting with different service providers, or even set up my own edge router

for test purposes, but I have not done so.

An alternative method of controlling download traffic from the gateway router would be to use the mice and elephants Ingress filter as described in the chapter on active queue management 2.5.

Chapter 7

Conclusion

Floyd and Van Jacobsen founded the field of Active Queue Management(AQM) by publishing the paper on Random Early Detection (RED) in 1993. It was possible to implement RED in high speed routers, because little state information was needed and thus not much computer memory required. Since then, there has been some debate about whether RED was truly useful. Few people, however, argued that fair queueing was not useful. Throughput remains unchanged, but fair queueing prevents a minority of flows from hogging all the bandwidth, and overall response times are improved as a result. Many schemes were devised which allow fair queueing while keeping a minimum of state information.

Fair queueing, however, is not optimal. It does not minimise overall response times. A Shortest Remaining Processing Time (SRPT) strategy, on the other hand, has been shown to be optimal with regard to response times. Sadly, a router would have to know the lengths of all flows to implement SRPT. Because of the heavy tailed nature of internet traffic, Shortest Job First (SJF) is a good approximation to SRPT in practice, and is potentially implementable in today's routers. A Mice and Elephants Router is a course approximation to SJF where flows are classified into only two classifications.

Linux has been a ready target for the implementation of queueing disciplines because it is ubiquitous and because of its open source nature, there is no need to apply for a licence to work with the kernel code. Now, a variety of queueing disciplines are shipped with Linux distributions. The existence of these queueing disciplines indicate a maturity of concept. Once implemented, these AQMs have become real. Now they exist.

The first aim of my project were to produce a prototype Mice and Elephants queueing discipline for the purpose of further evaluation of the Mice and elephants strategy. I have created two prototype Mice and Elephant queueing disciplines, as well as a Mice and Elephants ingress filter, thus satisfying the first aim of the project. I hope that researchers might use them to demonstrate Mice and Elephants queueing disciplines, or indeed Shortest Job First.

The second aim was to make a contribution to Linux by making queueing disciplines for Linux that would be useful in a small business or domestic setting. With my contri-

bution, Linux now has an ARED queueing discipline with accompanying documentation. This is significant, as RED cannot be made to be stable under all load conditions. The ARED queueing discipline is also easier to set up than the existing RED as some parameters have been set up to sensible defaults, and the most critical parameter is dynamically adjusted.

There is also now a Mice and Elephants queueing discipline and a Mice and Elephants ingress filter. These have been made sufficiently flexible to be useful in practice and could potentially improve the response time for interactive flows in a small business or domestic residence. They have been announced and are currently available for deployment. It is left for Linux distributions to include them and for system administrators to deploy them.

The third aim of my project was to explore and document a method of creating Linux queueing disciplines. I have achieved this aim in several ways. I have provided an environment for the easy creation and testing of Linux queueing disciplines. This includes a queueing discipline template to base future queueing disciplines on. This template is a generously commented drop tail queueing discipline with place holders for development. The environment includes code that forms a test harness so that queueing disciplines and ingress filters can be tried out in user space before being tested live in the Linux Kernel. It comes with an installation script. It also comes with a HOWTO, that documents how to develop queueing discipline with it. As far as I am aware, there is no other publicly available HOWTO for the creation of queueing disciplines, no other explicit documentation of the Linux kernel interface for queueing disciplines and no other queueing discipline template. I have also documented the Linux Kernel interface for queueing disciplines by means of a man page, which previously did not exist.

Bibliography

[1] URL: http://www.linux10.org/history/.

[2] Advanced virtual private server hosting. URL: http://www.vpsland.com.

[3] Clock in a linux guest runs more slowly or quickly than real time. URL: http://www.vmware.com/support/kb/enduser/std_adp.php?p_faqid=1420.

[4] Host hideout web hosting community. URL: http://www.hosthideout.com/showthread.php?t=16015.

[5] Iperf home page. URL: http://dast.nlanr.net/Projects/Iperf/.

[6] Linode web hosting. URL: http://www.linode.com.

[7] Linux history. URL: http://www.li.org/linuxhistory.php.

[8] The linux system. URL: http://www.soe.ucsc.edu/~sbrandt/courses/Spring00/111/slides/silbershatz/mod22.1.pdf.

[9] Man page of htb. URL: http://annys.eines.info/cgi-bin/man/man2html?tc-htb+8.

[10] Man page of tc. URL: http://www.manpage.org/cgi-bin/man/man2html?8+tc.

[11] Quantact hosting. URL: http://www.quantact.com/plans.shtml.

[12] Timestamp or timestamp reply message. URL: http://www.freesoft.org/CIE/RFC/792/9.htm.

[13] Umlsim home page. URL: http://umlsim.sourceforge.net.

[14] Unix man page for random. URL: http://www.scit.wlv.ac.uk/cgi-bin/mansec?3C+srandom.

[15] Managing traffic with altq. *Proceedings of the FREENIX Track 1999 USENIX Annual Technical Conference*, June 611, 1999 1999. URL: http://www.usenix.org/events/usenix99/full_papers/cho/cho.pdf.

[16] Manual page for tcsim. June 2003. URL: http://linux-ip.net/gl/tcng/node98.html.

[17] Ron Addie, Zhi Li, and Don McNickle. Implementing shortest job first order of service in the internet. *MODSIM International Congress on Modelling and Simulation*, December 2005. URL: http://mssanz.org.au/modsim05/papers/addie.pdf.

[18] Ron. Addie, Zhi Li, Zhongwei Zhang, and Moshe Zukerman. Controlling congestion by separating the mice from the elephants. *unpublished*, February 2004.

[19] Ronald G. Addie, Timothy D. Neame, and Moshe Zukerman. Performance evaluation of a queue fed by a poisson pareto burst process. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40, October 2002.

[20] Martin Devera aka devik and Don Cohen. Htb linux queuing discipline manual - user guide. June 2002. URL: http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm.

[21] Werner Almesberger. Linux network traffic control - implementation overview. 1999. URL: http://www.ing-steen.se/share/text/school/security/tc/tc.pdf.

[22] Werner Almesberger. Qos networking with linux. Feb 1999. URL: http://snafu.freedom.org/linux2.2/docs/slc-slides-Feb17.ps.

[23] Werner Almesberger. Linux traffic control - next generation. *Linux Kongress 2002*, September 2002. URL: http://www.linux-kongress.org/2002/papers/lk2002-almesberger.pdf.

[24] Werner Almesberger. Uml simulator. *Ottawa Linux Symposium*, July 2003. URL: http://umlsim.sourceforge.net/doc/umlsim-overview.ps.gz.

[25] Jeremy Andrews. Robert love explains variable hz. *KernelTrap.org*, 2002. URL: http://kerneltrap.org/node/464.

[26] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: investigating unfairness. *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*, June 2001. URL: http://www.cs.cmu.edu/~harchol/Papers/Sigmetrics01.ps.

[27] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol 13, Issue 7:422 to 466, July 1970. URL: http://portal.acm.org/citation.cfm?id=362692&dl=ACM&coll=portal.

[28] Martin A Brown. Traffic control howto. November 2003. URL: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/index.html.

[29] Martin A Brown. Traffic control howto. 2003. URL: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/.

[30] Martin A Brown. Traffic control using tcng and htb howto. April 2003. URL: http://www.linux.org/docs/ldp/howto/Traffic-Control-tcng-HTB-HOWTO/.

[31] Nevil Brownlee. Understanding internet traffic streams: Dragonflies and tortoises. *Communications Magazine, IEEE*, October 2002. URL: http://www.caida.org/outreach/papers/2002/Dragonflies/cnit.pdf.

[32] R. Caceres, P. Danzig, S. Jamin, and D. Mitzel. Characteristics of wide-area tcp/ip conversations. *Proceedings of SIGCOMM*, September 1991. URL: http://irl.eecs.umich.edu/jamin/papers/traffic/traffic.ps.Z.

[33] Wu chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. A self-configuring red gateway. *Proceedings of IEEE INFOCOM*, page 1320 to 1328, March 1999. URL: http://www.ieee-infocom.org/1999/papers/09e_01.pdf.

[34] Mikkel Christiansen, Kevin Jeffay, David Ott, and F. Donelson Smith. Tuning red for web traffic. *IEEE/ACM Transactions on Networking*, June 2001. URL: http://www.cs.unc.edu/~jeffay/papers/SIGCOMM-2000.pdf.

[35] Jeff Dike. A user-mode port of the linux kernel. *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta*, October 2000. URL: http://user-mode-linux.sourceforge.net/als2000/index.html.

[36] D.R.Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26:197–199, 1976. URL: http://links.jstor.org/sici?sici=0030-364X(197801%2F02)26%3A1%3C197%3AANPOTO%3E2.0.CO%3B2-7.

[37] Kevin Fall and Sally Floyd. Simulation-based comparisons of tahoe, reno, and sack tcp. *Computer Communication Review*, July 1996. URL: http://www.arl.wustl.edu/~gorinsky/cse573s/spring2006/TCP_versions.pdf.

[38] Feng, Kandlur, Saha, and Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. *Proceedings of INFOCOMM*, April 2001.

[39] Feng, Shin, Kandlur, and Saha. The blue active queue management algorithms. *IEEE/ACM Transactions on Networking*, 10, August 2002. URL: http://www.thefengs.com/wuchang/work/blue/ToN-02.pdf.

[40] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transaction on Networking*, August 1999. URL: http://www.icir.org/floyd/papers/collapse.may99.pdf.

[41] S. Floyd, R. Gummadi, , and S. Shenker. Adaptive red, an algorithm for increasing the robustness of red's active queue management. *ACM SIGMETRICS Performance Evaluation Review*, 3, June 2001. URL: http://www.icir.org/floyd/papers/adaptiveRed.pdf.

[42] Sally Floyd. Red: Discussions of setting parameters. 1997. URL: http://www.icir.org/floyd/REDparameters.txt.

[43] Sally Floyd. Recommendation on using the "gentle" variant of red. March 2000. URL: http://www.icir.org/floyd/red/gentle.html.

[44] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM transactions on networking*, 1993. URL: http://www.icir.org/floyd/papers/red/red.html.

[45] Liang Guo and Ibrahim Matta. The war between mice and elephants. *Technical Report 2001-005. citeseer.nj.nec.com/guo01war.html*, May 2001. URL: http://www.cs.bu.edu/techreports/pdf/2001-005-war-tcp-rio.pdf.

[46] Mor Harchol-Balter, Mark Crovella, and SungSim Park. The case for srpt scheduling in web servers. *Technical Report MIT-LCS-TR-767*, October 1998. URL: http://www.cs.cmu.edu/~harchol/Papers/papers.html.

[47] Bryan Henderson. Linux loadable kernel module howto. January 2005. URL: http://tldp.org/HOWTO/Module-HOWTO/index.html.

[48] Bert Hubert, Thomas Graf, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spanns, and Pedro Larroy. Linux advanced routing and traffic control howto. URL: http://lartc.org/howto/.

[49] Van Jacobson and Michael J Karels. Congestion avoidance and control. *Proceedings of SIGCOMM 88*, November 1988. URL: http://www.cs.unc.edu/~jasleen/Courses/Fall03/congestion-avoidance.pdf.

[50] B. Jenkins. A hash function for hash table lookup. *Dr. Dobb's Journal*, September 1997. URL: http://burtleburtle.net/bob/hash/doobs.html.

[51] Zheng Wang Jon. Eliminating periodic packet losses in the 4.3-tahoe bsd tcp congestion control algorithm. *ACM Computer Communications Review*, April 1992. URL: http://citeseer.ist.psu.edu/224009.html.

[52] Kernighan and Ritchie. The c programming language.

[53] Olaf Kirch and Terry Dawson. Linux network administrators guide. URL: http://www.faqs.org/docs/linux_network/.

[54] Jung-Shian Li and Ming Shiann Leu. Network fair bandwidth share using hash rate estimation. *Networks*, 40:125–141, May 2002. URL: http://140.116.163.3/4/02-09.pdf.

[55] Z. Li, Z. Zhang, C. Desem, and R. G. Addie. Dropping packets after they are received: What are the benefits? May 2004.

[56] Zhi Li. Presented work by zhi li. *unpublished*. URL: .

[57] Zhi Li, Zhongwei Zhang, Ron Addie, , and Fabrice Clerot. Improving the adaptability of aqm algorithms to traffic load using fuzzy logic. *Proceedings of Australian Telecommunication Networks and Application Conference*, 2004. URL: http://www.ieee-infocom.org/2004/Papers/01_1.PDF.

[58] Dong Lin and Robert Morris. Dynamics of random early detection. *SIGCOMM*, 1997. URL: http://www.pdos.lcs.mit.edu/~rtm/papers/fred-abstract.html.

[59] M Mathis and M Allman. A framework for defining empirical bulk transfer capacity metrics. *Request for Comments 3148*, 2001. URL: http://www.ietf.org/rfc/rfc3148.txt.

[60] Paul McKenney. Stochastic fairness queueing. *IEEE INFOCOM*, March 1990. URL: http://www.rdrop.com/users/paulmck/paper/sfq.2002.06.04.pdf.

[61] Don McNickle and Ron Addie. Comparing different approaches to the use of diffserv in the internet. *Proceedings of Tencon Conference*, November 2005. URL: http://eprints.usq.edu.au/archive/00000186/01/tenconcamera.pdf.

[62] Misra, Ott, and Baras. Effect of exponential averaging on the variability of a red queue. *Proceedings of IEEE International Communications Conference (ICC)*, June 2001.

[63] John Nagle. Rfc 896 - congestion control in ip/tcp internetworks. January 1984. URL: http://www.freesoft.org/CIE/RFC/896/index.htm.

[64] John Nagle. Rfc 970 - on packet switches with infinite storage. December 1985. URL: http://www.faqs.org/rfcs/rfc970.html.

[65] Anna Ostlin and Rasmus Pagh. Simulating uniform hashing in constant time and optimal space. *BRICS - Basic Research in Computer Science*, 2002. URL: http://www.brics.dk/RS/02/27/BRICS-RS-02-27.pdf.

[66] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *todo*, 2001. URL: http://ccr.csail.mit.edu/issues/folder.2004-06-17.7910819251/academicpaper.2004-06-17.8170925356/File/.

[67] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions On Networking,*, 1 No3:344, June 1993. URL: http://www.cs.cmu.edu/~bumba/filing_cabinet/papers/parekh-gps.pdf.

[68] Bob Pendleton. Doing it fast. URL: http://www.gameprogrammer.com/4-fixed.html.

[69] J Postel. Rfc 768 - user datagram protocal. *Requests for Comments*. URL: http://www.ietf.org/rfc/rfc768.txt.

[70] J. Postel. Rfc internet control message protocol. *1981*, September. URL: http://www.ietf.org/rfc/rfc0792.txt.

[71] K. Ramakrishnan, S. Floyd, and D. Black. Rfc 3168 - the addition of explicit congestion notification (ecn) to ip. *Request for Comments 3168*, September 2001. URL: http://www.faqs.org/rfcs/rfc3168.html.

[72] R.Pan, B. Prabhakir, and K. Psounis. A stateless active queue management for approximating fair bandwidth allocation. *Proceedings of IEEE INFOCOM 2000*, March 2000. URL: http://bat710.univ-lyon1.fr/~cpham/TCP/CHOKe-infocom00.pdf.

[73] Bruce Schneier. Blowfish encryption. 1993. URL: http://www.cebrasoft.co.uk/encryption/blowfish.htm.

[74] Anees Shaikh, Jennifer Rexford, , and Kang G. Shin. Load-sensitive routing of long-lived ip flows. URL: http://www.eecs.umich.edu/techreports/cse/1999/CSE-TR-392-99.pdf.

[75] B. Sikdar, S. Kalyanaraman, and K.S. Vastola. An integrated model for the latency and steady state throughput of tcp connections. *Proceedings of the IFIP Symposium*

on Advanced Performance Modeling, Florida, November 2000. URL: http://networks.ecse.rpi.edu/~bsikdar/papers/peva.ps.gz.

[76] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[77] W. Richard Stevens. Unix network programming volume 1. page 661.

[78] User Mode Linux Core Team. User mode linux howto. January 2005. URL: http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO.html.

[79] T.J.Ott, T.V.Lakshman, and L.H.Wong. Sred: Stabilized red. *Proceedings of INFOCOMM*, March 1999. URL: http://web.njit.edu/~ott/Papers/Lakshman/Infocom1999.pdf.

[80] Chris Townsend. Speed! float vs int. URL: http://aulos.calarts.edu/pipermail/music-dsp/1998-July/020377.html.

[81] Chonggang Wang, Bin Li, Y. Thomas Hou, Kazem Sohraby, and Yu Lin. Lred: A robust active queue management scheme based on packet loss ratio. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, page 1 to 12, March 2004. URL: http://www.ieee-infocom.org/2004/Papers/01_1.PDF.

[82] Bob Webster. More junkmail from bob. URL: http://xpda.com/junkmail/junk103/junk103.htm.

[83] Andrew Chi Chih Yao. Uniform hashing is optimal. *Journal of the ACM*, 32, 1985. URL: http://historical.ncstrl.org/litesite-data/stan/CS-TR-85-1038.pdf.

Appendix A

Linquede HOWTO

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Appendix B

Qdisc Man Page

.

.

.

.

.

Appendix C

ARED Man Page

.

.

.

.

.

Appendix D

MERED Man Page

.

.

.

.

.

Appendix E

MEREDT Man Page

.

.

.

.

Appendix F

MEREDTF Man Page

.

.

.

.

.

.

Appendix G

Example TC script