Implementation of AQMs on Linux made Easy

Stephen Braithwaite
University of Southern Queensland

Abstract

In this project we implemented a Mice and Elephants IP packet queueing discipline and policer on Linux. The logical algorithm or the implemented code that is responsible for managing packets destined for a network interface is termed a queueing discipline. The logical algorithm or the implemented code that chooses which packets to accept from a network interface is termed a policer. A Mice and Elephants queueing discipline or policer favours short IP flows over long IP flows. Packets which collectively over time accomplishes a single task or set of tasks may be termed a flow. Packets from or to a single location or set of locations may also be termed a flow.

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.

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* discipline (AQM) [25].

In this project I have investigated and implemented a particular type of AQM, called a *mice and elephants* AQM [21]. 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* [26]. 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. In Linux, the code module that implements a queueing discipline algorithm is also called a *queueing discipline* [12].

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.

Mice and Elephants Strategies

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 [26]. 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 [33]. 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 [31]. We can implement fair queueing on Linux by deploying a fair queueing discipline, such as SFQ [29] in place of the default drop tail queueing discipline [18]. 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.

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) [26]. The mice are favoured in two ways:-

- Packets from elephant flows (elephant packets) are dropped according to the Random Early Detection (RED) algorithm [25] 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 [10,26,30].

Linux Networking Concepts

Classifiers, Policers, Queueing Disciplines and Filters

Illustration 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. Policers are often used to limit the rate of a traffic flow entering a network interface. In this example, packets that are not dropped are delivered to the queueing discipline on the right.

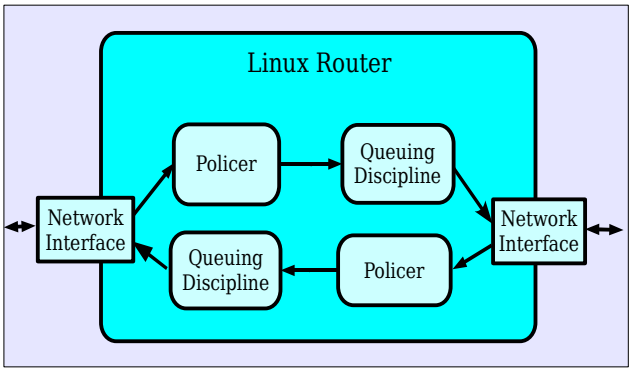

Illustration 2: A Linux router with two network interfaces, each with a policer and a queueing discipline.

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. Illustration 2 shows the flow of packets through a queueing discipline.

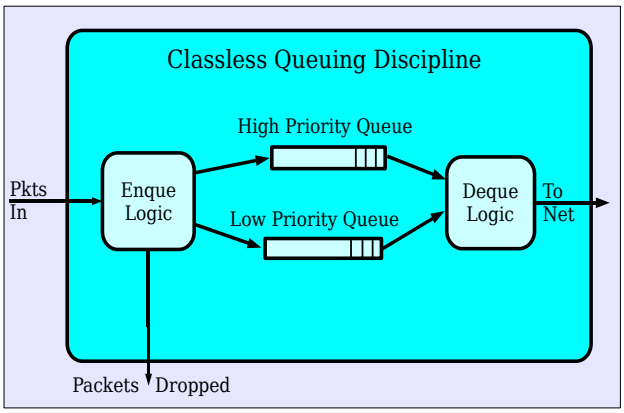

Illustration 3: An example of a classless queueing discipline with two queues.

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

Illustration 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 [12]. The inner queueing disciplines could be either classless or classfull.

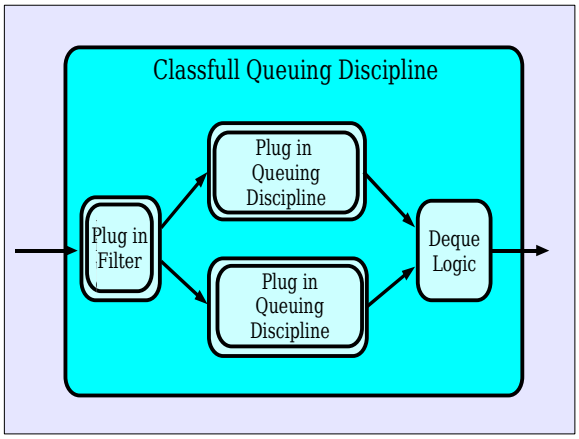

Illustration 4: An example of a classfull queueing discipline with a classifier that directs traffic between two inner queueing disciplines.

On Linux, the policer is formed by combining a classifier with 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 does no queueing, but only polices packet arrivals. Illustration 4 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.

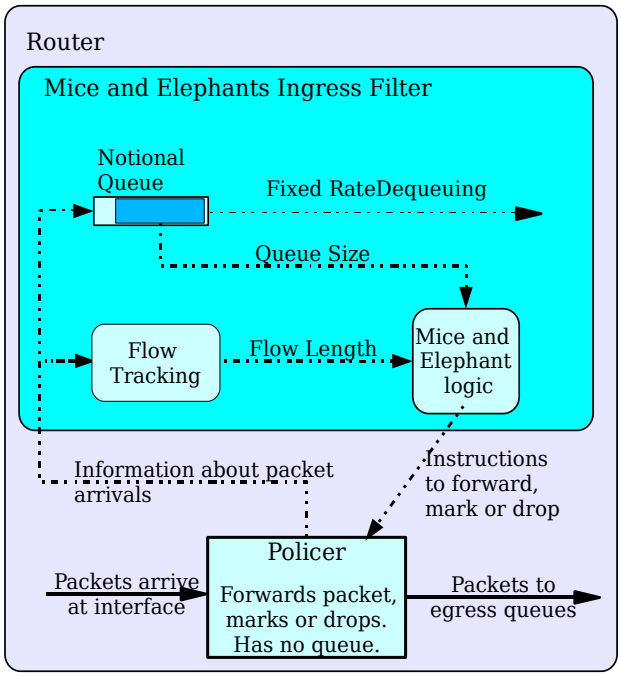

Illustration 5: A Mice and Elephants Ingress Filter on Linux

Example Classifier - 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 [8].

Example Classfull Queuing Discipline - HTB

An example of a classfull queueing discipline is HTB (Hierarchical Token Bucket) [11]. 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 [20]. 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.

The Linux Queueing Discipline Interface

The Linux kernel has a well defined, but poorly documented interface for queueing disciplines. A Linux man page has been written for this interface as part of this project [17].

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 [5]. *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" [19].

It is generally agreed that the *tc* command is difficult to use [20]. The TCNG package provides a higher level language with which to control Linux queueing disciplines [13]. 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.

Linux queueing disciplines and ingress filters are to be found *inside* the Linux kernel, so they are not normal user space programs [27]. 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.

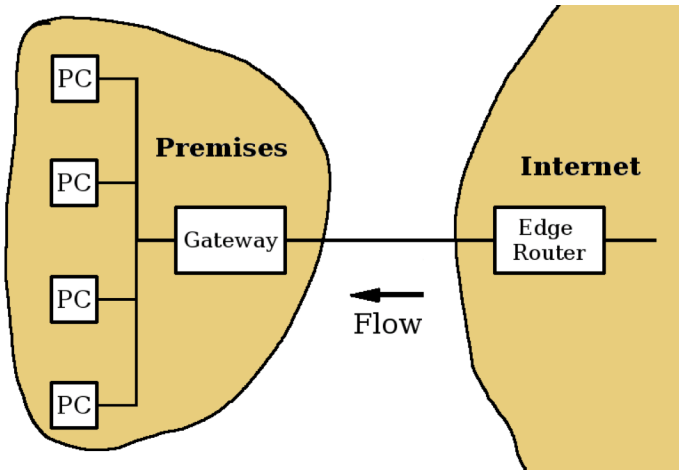

Illustration 6: Download traffic is typically heavier than upload traffic

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:-

TCSIM

The traffic control simulator (TCSIM) [13] 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 [9].

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 [14] 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.

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) [34]. 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 [23] :-

- 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 rebooting.
- 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. While it is relatively easy to run UML on other Unix systems, it has not yet 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 :- [6, 2, 1, 4]
- 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.

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 [7, 14] 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" \cite{umlsim1}. 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.

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 [16].

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.

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.

Testing

Scenario Testing

As part of the project, two scenarios were explored by means of tests. These tests are contrived, and I acknowledge that running scenarios in a test harness is not a substitute for testing the queuing disciplines in real life.

Comparison of Tests

Potential benefits of the Mice and Elephants queueing discipline or the Mice and Elephants classifier were demonstrated for a set of conditions by comparing network behaviour in a set of three separate tests. These were:-

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

This is interesting because it shows what performance could be obtained if there were no elephant flows. Thus, this performance might be seen as an ultimate goal of Mice and Elephant queueing disciplines.

The *Congested* test In this test the drop tail queueing discipline was retained, and an elephant flow was added.

It is expected that the connection time and response time of the mice flows will suffer. One elephant flow should be enough, because elephant TCP flows normally use the entire available bandwidth [28].

The *Mice and Elephants* test In this test the elephant flow was retained and the mice and elephant queuing discipline or classifier was used instead of the default.

Hopefully the response time of the mouse flows

should improve. In the (unattainable) ideal case, the mice flows should have response times as if there was no elephant flows.

Queuing Discipline Scenario Experiment

The link speed used in the experiment was the maximum dialup modem speed of 56Kbit/second. Each experiment involved mouse flows which took the form of TCP uploads. The average flow length used was 5Kb, and were initiated at an average rate of of one mouse flow per second.

The results of the experiment is shown in Illustration 6. 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* test and the *congested* test. Despite being unable to start all of the mouse flows in the congested test, 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.

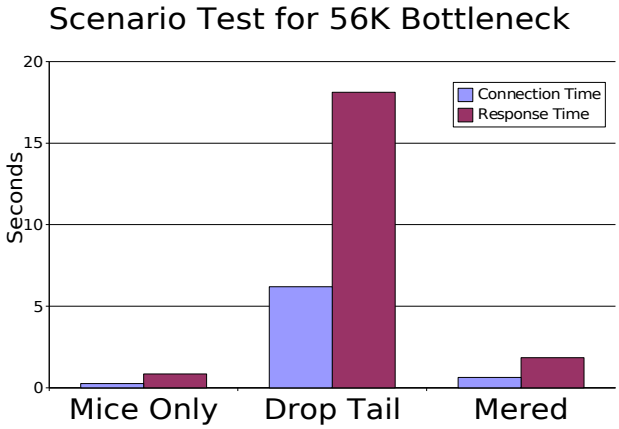

Illustration 7: Average response and connection times for mouse flows on 56 Kbit/sec link using Mice and Elephants queueing discipline.

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 between the *congested* test and the *Mice and Elephants* test.

Classifier Scenario Experiment

The link speed used in the experiment was the 100Mbit/second. Each experiment involved mouse flows which took the form of TCP Downloads. The average flow length used was 100Kb, and were initiated at an average rate of of one mouse flow per second.

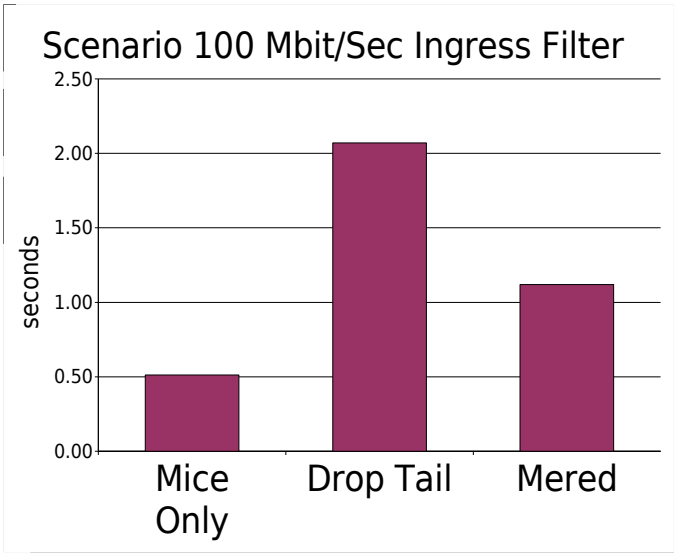

Illustration 8: Average response times for mouse flows on 100 Mbit/sec link using Ingress filter with Mice and Elephants classifier

The results of the experiment is shown in Illustration7. With the classifier, mouse flows could not benefit from priority queuing, of course, and so the potential improvement in response times that could be obtained is less than that obtained with the queuing discipline, but is still impressive.

Load Testing

One of the aims of this project is to make a contribution to Linux by making an implementation 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 [3] to generate, receive and measure traffic. Illustration 8 shows the CPU loads for the queuing discipline load tests. CPU loads are averages.

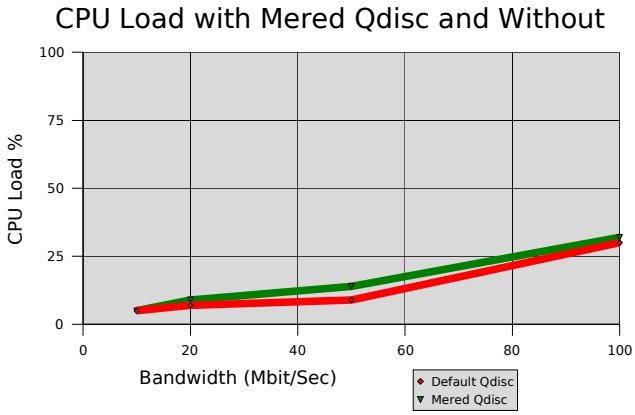

Illustration 9: Percentage of CPU load for Mered queuing discipline on a Pentium 450

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 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. Illustration 9 shows the CPU loads for the ingress filter load tests. CPU loads are averages.

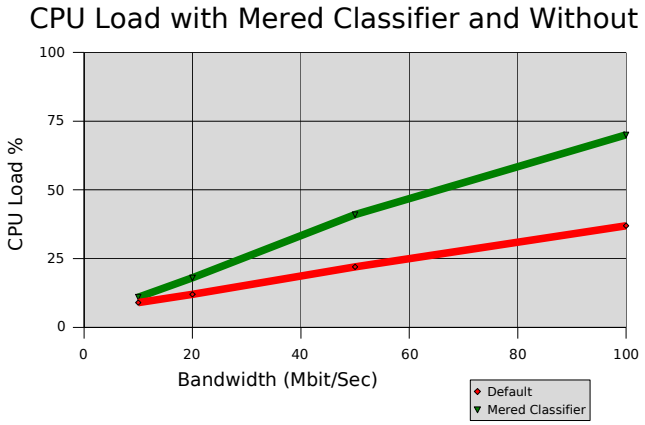

Illustration 10: Percentage of CPU load for Mered classifier on a Pentium 450

The load testing shows that most of the CPU power of a Pentium 450 based system would be used for a Mice and Elephants classifier to cope with 100Mbit/second IP traffic. A small business or domestic residence could potentially use this queueing discipline and classifier on a gateway machine. A Pentium 450 would suffice for this purpose and the benefit would be the improved response time.

Conclusion

Floyd and Van Jacobsen founded the field of Active Queue Management(AQM) by publishing the paper on Random Early Detection (RED) in 1993. 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.

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 an approximation to SRPT and a Mice and Elephants Router is an 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. 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.

The first aim of my project was 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.

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. A Mice and Elephants queueing discipline and ingress filter have been created. 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. The code and documentation is available from [15]. It is left for Linux distributions to include them and for system administrators to deploy them.

Also, the developed queuing disciplines were based on ARED [24], and so ARED had to be developed too. This has been made available. This is significant, as the ARED queueing discipline is a significant improvement on the existing RED [25] queueing discipline [24, 22, 32].

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. It is documented with a HOWTO [16] and includes a generously documented queueing discipline template to base future queueing disciplines on. I have also documented the Linux Kernel interface for queueing disciplines by means of a man page [17], which previously did not exist.

References

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

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

[15] Stephen Braithwaite. Source code and documentation of linquede and queueing disciplines. URL: http://eprints.usq.edu.au/archive/00001280.

[16] Stephen Braithwaite. Creating new linux queuing disciplines howto. *2005*. URL: http://eprints.usq.edu.au/archive/00001280/02/BraithwaiteAppendixA_howto.html.

[17] Stephen Braithwaite. Linux queuing discipline module interface. *2005*. URL: http://eprints.usq.edu.au/archive/00001280/03/Braithwaite_AppendixB_qdisc.pdf.

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

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

[20] 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.

[21] 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.

[22] 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://citeseer.ist.psu.edu/feng99selfconfiguring.html.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

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

[32] R.J.La, P.Ranjan, and E.H.Abed. Analysis of adaptive random early detection (ared). *Proceedings of the International Teletraffic Conference, September 2003.* URL: http://www.isr.umd.edu/~priya/paper272.pdf.

[33] 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/~vastola/pubs/sapm00.ps.gz.

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