Implementing Real Time Packet Forwarding Policies using Streams

Ian Wakeman, Atanu Ghosh, Jon Crowcroft [*]
Computer Science Dept, University College London,
Gower Street, London WC1E 6BT.
Van Jacobson and Sally Floyd
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720

November 14, 1994

Abstract

This paper describes an implementation of the class based queueing (CBQ) mechanisms proposed by Sally Floyd and Van Jacobson [1] [2] to provide real time policies for packet forwarding. CBQ allows the traffic flows sharing a data link to be guaranteed a share of the bandwidth when the link is congested, yet allows flexible sharing of the unused bandwidth when the link is unloaded. In addition, CBQ provides mechanisms which give flows requiring low delay priority over other flows. In this way, links can be shared by multiple flows yet still meet the policy and Quality of Service (QoS) requirements of the flows.

We present a brief description of the implementation and some preliminary performance measurements. The problems of packet classification are addressed in a flexible and extensible, yet efficient manner, and whilst the Streams implementation cannot cope with very high speed interfaces, it can cope with the serial link speeds that are likely to be loaded.

1 Introduction

The Internet is fast approaching a period of revolutionary change in the services provided and how they are paid for. New architectures and protocols are being designed and implemented to extend the Internet to support Integrated Services, based on the work of the INT-SERV working group of the IETF [3]. It is envisioned that audio, video and other real-time services will be sent over the Internet. The ongoing commercialisation of the Internet necessitates a new model of service, where the users and providers exchange money for a guarantee of a basic level of service. Since the delivered service for both real time applications and contractual guarantees is dependent upon the mix of packets on the links and in the switches, a key component of the new Internet will be the packet forwarding scheduler within the switch. This must provide both a method for sharing bandwidth amongst the organisations who pay for the link and provide appropriate levels of Quality of Service for flows with real-time requirements.

One vision of how to design this building block has been offered by Sally Floyd and Van Jacobson [1] [2]. They start from the premise of link sharing, where links are leased by multiple organisations, or agencies, who then require a guarantee of a share of the bandwidth when they need it, but if the bandwidth is not used, then other users can send packets. These requirements can only be satisfied in any sensible manner through the scheduling of packets to be forwarded. Each of the agencies are guaranteed a minimum amount of the bandwidth, with the proviso that any instantaneously unused bandwidth is shared amongst the agencies in some previously agreed upon manner. This technique for allocating band-

[*] Supported by DARPA grant number AFOSR890514 and generous donations from Sun Microsystems Laboratories Inc

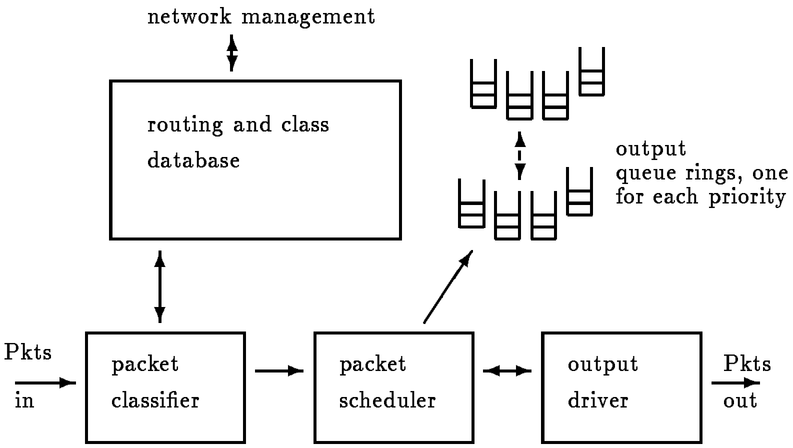

Figure 1: Conceptual Breakdown of the CBQ filter Code

width can be naturally extended to provide the allocation mechanisms for real-time traffic, providing the required Quality of Service (QoS) through guaranteeing bandwidth and low delay. The unifying abstraction for link-sharing and real-time traffic is the use of a class hierarchy; each class is provided with a share of the bandwidth and a delay characteristic, and the hierarchy enables mechanisms and policies for "borrowing" of bandwidth from other classes.

The conceptual breakdown of how CBQ should be implemented has been proposed by Van Jacobson and is illustrated in Figure 1. The classifier interprets the header information of an incoming packet to determine the class of service that the packet should receive from the scheduler. The classifier returns a pointer to the class structure that holds the queues and associated information. The packet is placed on the appropriate queue by the scheduler if the queue is not full. The output driver works asynchronously, invoking the scheduler to determine from which queue to send the next packet, according to the current utilisation of the queues and their priorities.

Classifying a packet is very similar to the problem of determining the route matching a destination address in a packet. In both cases, fields in the header are used to look up information in a table. However, the classification problem is more complex because the patterns upon which the packet may match a class must be more general and can match any part of the header. For instance packets may be from one of multiple agencies, requiring the examination of destination and *source* addresses, classified on transport or other protocols such as TCP, UDP or ICMP, or applications such as ftp or telnet, requiring the examination of the port numbers. For video streams we may look even further within the packet to determine the level and "droppability" of a packet within a hierarchically encoded video stream [4].

It is a point of some controversy as to how widely the above mechanisms need to be fielded within the Internet. It could be argued that they need only be used where the links are heavily utilised, and thus subject to congestion. Links which have low utilisation can supply the necessary Quality of Service for all types of stream with normal FIFO packet scheduling, since queues on the links are very small or non-existent. The motivation for the work described in this paper came from attempting to make the allocation of bandwidth more efficient on the Trans-Atlantic link (known as the FAT pipe) used to connect the data networks of the UK Ministry of Defence (MoD) and the Department of Defence (DoD), of the European Space Agency (ESA) and NASA, and of the UK academic IP network and the US academic network. Currently, the link is hard-multiplexed into three parts, one for each agency pair[1]. However, large cost sav-

[1] Currently split into an E1 and a T1 link, with the T1 link split into two equal channels. For details on

ings could be achieved on an expensive link if we guarantee each of the agencies a minimum share of the bandwidth for their mission, yet could allow any unused bandwidth to be used by agencies with excess traffic. The current version of the CBQ filter is fielded on the FAT pipe.

In this paper we describe the implementation of a programmable CBQ mechanism within the Streams implementation of IP forwarding. The classes and the patterns that determine the membership of a particular class are compiled in user space, and lookup engines are chosen to optimise the per-packet lookup code. The classes are then downloaded to the CBQ filter module, which schedules the packets using the downloaded information. The scheduling code is largely based upon the work of Lawrence Berkeley Laboratory (LBL), whilst the classifier and the surrounding infrastructure are the work of University College London (UCL). The particular choices made in the design of the classes and the classifier are described in Section 2, the design of the Streams module and its virtual interface are described in Section 3 and performance measurements reported in Section 4. We conclude in Section 5 with ruminations on the feasibility of this design and the limitations of the design from the Streams mechanism and the pointers to future work.

2 Link Sharing Policies and the Classifier

The problem is to map the high level specification of policies onto a classifier that maps a packet into a particular class so that the necessary information can be found for scheduling the packet for output.

The high level specification of policies should be at a level of abstraction that can be used in the negotiation of legally binding sharing of the link. For instance, the agencies should be named as NASA or UK academic organisations, packet types should be specified by the services they provide - interactive data, bulk data, video data, audio data, specific sites, protocol suites (DECnet, IP etc.), and specific protocols.

Our initial class structure definition has the following hierarchical ordering when we are allocating the shares of the bandwidth:-

1. Agency
2. Protocol Suite
3. Protocol
4. Service

This partitioning first allows the bandwidth to be divided up by organisation, allowing arbitrary levels in this level. The next level is a suggestion that bandwidth should be given to protocol suites separately, so, for instance, OSI and IP packets are separated out in treatment. However, an actual implementation would reverse the process in classifying the packets, since the agency could only be discovered after knowing the structure of the packet. In this paper, we consider only IP. The partitioning by protocols allows protection against various forms of link sharing at the congestion control level - e.g. none vs slow start [6] vs responsible second order sharing [7] [8], [9] and to specify appropriate actions when the classes are exceeding their bandwidth allocation when the link is loaded. The final division by service allows us to divide up bandwidth amongst the applications. By adding a priority level at this point and implementing sensible borrowing policies in the scheduler, we can ensure that when there is congestion the service offered to the applciations is degraded according to their importance, such as video packets being dropped before audio in a video conference.

2.1 What's a class?

Having derived the abstractions used to map a packet onto a class, we then want to consider the actual attributes that will be attached to a class. A class is :-

- A share of the bandwidth
- A priority
- A parent class
- A set of pattern tuples over a packet, $P_j = \{\{A_i, V_i\}\}$,where a pattern is:-
  - A pattern A_i.
  - A mask of significant bits V_i.

the usage see [5]

As happens in the solution to many computing problems, we define a hierarchy of classes. Rather than attempting to map a packet onto a class in one step, our pattern matching proceeds in order down the tree. We thus compare the packet against the patterns of the children of the current class. To ensure that a packet maps unambiguously only onto a single class, the patterns specified should follow the following constraint in the general case:-

- For each child of a given parent, each pattern of each child should be distinguishable from the patterns of all other children, ie for children i and j, there should exist no pattern tuples in i and j such that $A_i \wedge V_i \equiv A_j \wedge V_j$.

To simplify the implementation, and because the mapping abstractions correspond nicely to the packet headers we are using, the patterns are further constrained to map directly onto fields in the protocol header. For our initial IP implementation, the agency maps onto the destination and source addresses[2], the protocol maps onto the protocol field, whilst the application maps onto the port numbers in UDP or TCP [10,10]. The abstraction of a class and the tree structure of the classes leads to the number of patterns we have to compare a packet against being exponential in the depth of the tree, using Figure 2. In addition, the number of patterns at a particular level may be very large - e.g. at the agency level, the current design of protocols insists that we compare against the destination and source net numbers of the composite networks of the agency. For the initial case this can be as large as 10,000 or more (UK and US academic institutions). Furthermore, since the patterns will be repeated at each level (because each agency will want to partition traffic along similar lines), this will require a large number of patterns to be constructed if we attempt to classify the pattern in one step. Making a comparison at each level of the tree reduces the total number of patterns that need to be stored, but increases the number of comparison operations.

However, this is not a problem, since the class tree will be shallow - generally three or four levels - and we can select a lookup engine for a given class level according to the nature of the patterns to be compared against. The use of IP and simple classes allows the classification to proceed by a sequence of hash table or simple table lookups. Note that there is a default class as a child of the root of the tree into which a packet falls if it does not map to any other class. This default class gives the lowest quality of service, since it is not being used by any of the agencies paying for the link. It should be noted that the existence of a default class will encourage agencies to offer their link as a transit link for other agencies, and promote greater connectivity.

The use of sequenced lookups for the classes thus adds additional parameters to the class - a lookup engine function, and the data structures for the lookup engine.

Its then becomes natural for the network manager to specify the full set of parameters for the classes in a file, which is then compiled. Simple parenthesised definitions are used to define the tree. Patterns are defined very simply at present, just as a specification of the field within the IP packet and the pattern to compare it against.

The classifier compiler types the set of patterns that define the child classes of a class, checking that they can all be used to create a single engine. It then creates the data structures for the engine, placing the engine type in the class data, along with functions to lookup, insert and relocate the data, along with other parameters to describe a class.

Relocation of the data structures is necessary to allow the compiler to work in user space and then pass the data structures down to the kernel driver. The design is split in this manner so that we could experiment with reservation strategies - adjustments to the bandwidth and the priorities of classes can be worked out in user space and then passed down to the kernel. Additional management functionality, such as Management Information Base (MIB) creation etc., can be easily designed and added at a later date. An example configuration file is shown in Figure 3, which is used in the tests described below. An alternative approach to packet classification is to use a generalised patricia lookup engine [11], [12]. Patricia constructs a tree which is traversed by testing only against the bits of the key which differentiate the patterns stored

[2] If a member of agency A is temporarily on walkabout, yet still wishes to use the share of the link, an additional pattern can be added to the class to cope with the temporary addresses.

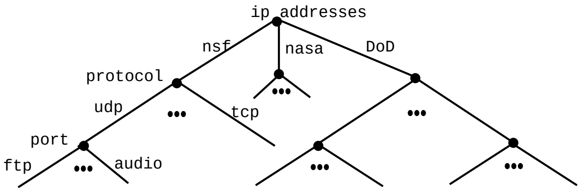

Figure 2: Example Class tree hierarchy

```
# bandwidth of wire in
# nanoseconds per byte
linkspeed 20000
{ name  ucl-cs
# percentage of bandwidth
  bandwidth 90
  pattern [addr 128.16.0.0]
  { name ucl-cs-tcp
# percentage of total bandwidth
    bandwidth 50
# action when overlimit
    overlimit drop
    pattern [proto tcp]
    { name highTCP
# high number = high priority (max 7)
      priority 6
      bandwidth 29
      overlimit drop
      pattern [port 5001]
    }
    { name lowTCP
      bandwidth 19
      overlimit drop
      pattern [port 5002]
    }
  }
  { name ucl-cs-udp
    bandwidth 39
    overlimit drop
    pattern [proto udp]
    { name CBR
      priority 7
      bandwidth 38
      overlimit drop
      pattern [port 4579]
    }
  }
}
```

Figure 3: Classes used in Testing the Streams Module

in the tree. The technique has been generalised to cope with masked keys by Halpern [13] and Tsuchiya [14]. However, because our patterns may have no bits in common, such as matching on one of either destination or source address, multiple passes are still required through the tables. In addition, tests against a generalised Patricia tree lookup engine show the sequenced table lookup as more efficient (Section 4).

Another common mechanism for classifying packets is the automata used inside the Berkeley Packet Filter (BPF) and other similar entities [15]. These are designed to match a narrow range of patterns across a packet, and to do this very efficiently. However, the range of patterns we expect the classifier to handle is very wide, which would end up with a number of automata which would all need to be tried sequentially. Therefore we have generalised the framework of the classifier to use generalised engines. This does not preclude use of the a BPF automata as a particular instance of an engine, if that is the most suitable.

2.2 The scheduling properties of a Class

Each of the classes maintains an exponentially weighted moving average (EWMA) (avgidle in Figure 4) of the idle period between packets, updated on every packet using the time it would take to send the same size packet using the percentage of the bandwidth allocated to the class. When the average is less than zero, a class is transmitting more than its share of the bandwidth and is thus “overlimit”. If the class of a packet is overlimit, the “borrow” class of the packet is examined to see if the original class can borrow bandwidth to transmit the packet, and so on up the hierarchy. If the class goes overlimit and can’t borrow,

```
struct rm_class {
  mbuf_t *tail; /* tail of circularly linked output q */
  struct timeval last; /* time last packet sent */
  struct timeval undertime; /* time can next send */
  int sleeping;      /* != 0 if delaying */
  int qcnt; /* # packets in queue */
  int avgidle; /* EWMA of idle time between pkts */
  struct rm_class *peer; /* Linked list of same priority classes */
  struct
  rm_class *borrow; /* Class to borrow bandwidth from */
  struct rm_class *parent; /* Parent class */
  struct rm_ifdat *ifdat; /* Output Device data structure */
  int priority; /* Class priority */
  int maxidle; /* Roof of avgidle */
  int offtime; /* Penalty added to class when overlimit */
  int qmax; /* Maximum queued pkts */
  void (*overlimit)(); /* Action to take when we can't borrow */
}
```

Figure 4: The Class Structure related to Scheduling (after Sally Floyd)

then the overlimit action is invoked, such as dropping the packet, or delaying the packet. The maximum size of the avgidle estimator is limited by the maxidle parameter which prevents the class from building up credit when it isn't transmitting, and so limits the maximum burst size from the class.

The scheduling code maintains equal priority classes with traffic to send in circular queues. The packet to send is either from the highest priority class which is underlimit, or from the highest priority class which is overlimit but can borrow from classes above it in the borrowing hierarchy of classes. After a packet is sent from a particular class the queue pointer is advanced to the next position in the queue. Thus we implement prioritised round robin as long as classes are underlimit. When the class goes overlimit, it allows other underlimit classes to send first and so prevent starvation of any class. In this way, real time flows can ensure low delay by setting the priority of the class high and ensuring that the bandwidth share requested is sufficient such that they are always underlimit. When the queues are full or when the overlimit action specified is to drop a packet, the current implementation drops from the tail. However, other mechanisms such as dropping from a random position within the queue are possible [16]. The details of scheduler are described more fully in [1].

3 Implementation of a Streams-based Packet Forwarding Engine

The Streams "plumbing" used for the CBQ filter can be seen in Figure 5. The CBQD module is used to communicate with the CBQ filter module and to download the class buffers and to obtain statistics on usage.

Our target scenario was for the workstation, a SparcClassic running Solaris 5.2, to sit transparently in front of the router whose output serial line we wished to protect[3]. The workstation and the router would be connected by Ethernet or some high bandwidth link. Thus we had to emulate the speed of the router interface in the CBQ software. This is done by tracking the time that the packet would be sent from the interface if it were the speed of the serial line in virtual time and suspending transmission if the virtual time gets too far in front of real time - ie the queue in the router builds up. We rely on incoming packets and a backstop timer to ensure that the transmission is continued at some point after being

[3] The obvious place to implement this code is in the router, but we had no way of modifying the router.

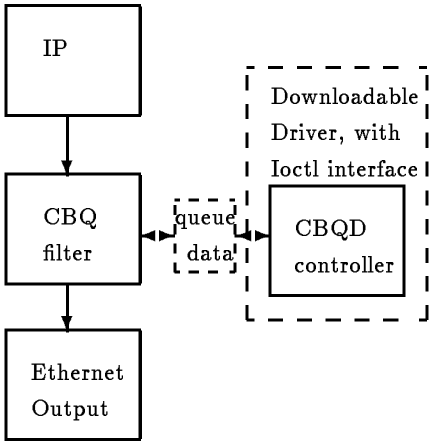

Figure 5: The Streams Plumbing of the CBQ filter modules.

suspended. An additional advantage of running in "virtual" time is that the code becomes independent of output completion interrupts. The original LBL code used the completion interrupt to service another packet, but device drivers in Solaris do not provide this interrupt.

Implementing the CBQ filter as a Streams module and driver had some excellent benefits. We could introduce code into the protocol stack without requiring changes to the kernel source, with one major proviso. At the time the work was started the Streams plumbing code was built into a program ("ifconfig"); it was therefore necessary to take a copy of the "ifconfig" program and modify the source to enable us to insert the CBQ module at the correct point in the Stream. It would have been more convenient if the Streams implementations had used a configuration file to set up the plumbing, as some other implementations do.

As Solaris only supports dynamically loadable drivers we had hoped that it would be possible to write and debug the CBQ module without ever having to reboot the development machine. Unfortunately there was no obvious way of tearing down a Stream and creating a new one with a new CBQ module. Therefore it was necessary to reboot each time we wished to test a new module. If there were any serious bugs in the CBQ module the only way of recovering was to boot the machine from the boot prompt and edit the relevant startup files to not use the CBQ module. One way we attempted to circumvent this problem was to try and create a file before attempting to load the CBQ module - if this file already existed then the CBQ module would not be placed in the Streams stack. Unfortunately the network is configured so early in the Solaris boot sequence that the filestores are still readonly, so this strategy failed.

The scheduling code used was written at LBL and had been written for a BSD derived kernel. The first task was to change the code to use the Streams interface. This turned out to be quite easy; the BSD interface structure mapped quite simply onto a Streams queue and the "mbuf" structures mapped onto the Streams message buffers with the aid of some macros. The major change was the addition of some locks. The classification code was then integrated into the CBQ module.

A mechanism was required to control the CBQ module, to enable/disable the classification and to change the tables. The standard mechanism for sending information to drivers/modules is by obtaining a file descriptor to the driver/module and then issuing an "ioctl" which is understood by the driver. However the CBQ module is below a Streams multiplexor, and there is no way that the multiplexor layer can correctly deliver the ioctl message without rewriting the multiplexor code. The only way around this problem was to write a CBQ driver which accepts "ioctl" requests and makes a direct connection to the CBQ module, as in Figure 5, using a shared piece of memory in the queue structure.

Code	Lookups	Total Time spent /s	Mean ms/Call
UCL Classifier	15186	0.20	0.01
Patricia	15186	0.47	0.03

Table 1: Classifier test results

4 Performance measurements

4.1 Classifier performance

The classifier is designed to be flexible in the lookup engines used on the packets. The alternative design choice would have been to use a single engine, such as the modified Patricia code from Joel Halpern [13]. We compared the performance of the two approaches, using a classifier with multiple hash and normal table lookups with a modified Patricia engine. We specified the classes to consist of 1920 addresses culled from the NSFNet acceptable use database, with child classes of UDP and TCP traffic, and child classes of Telnet and FTP-data for the TCP class. We profiled the engines using the gprof profiler on a sparcStation10 under Solaris 2.3, and in all tests, the UCL classifier code was faster, even though the patricia code ignored the destination address and port fields, due to the limitations described in Section 2. We present sample results from a test designed to exercise all paths of the engine in Table 1.

4.2 Streams Module Performance

In the experiments described below, we use the setup described in Figure 7. Both interfaces of the CBQ filter machine are on the same Ethernet, so we can see both input and output packets using a single tcpdump [17].

To measure the throughput of the Streams module, we measured the time taken to forward a series of packets well below the specified output rate of the filter for one of the connections on its own, and compared these with the performance of IP forwarding without the CBQ module. The results can be seen in Figure 6. These suggest that a sparcClassic can forward 700 Kbit/s of minimum sized packets, or 7 Mbit/s of maximum sized packets on an Ethernet, and that traversing the CBQ module takes in the order of 300-450 microseconds. It should be noted that the current code is unoptimised - it does an extra copy to ensure alignment of the headers, and will pull up buffers without hesitation. The next version of the code will remove these inefficiencies.

To illustrate the performance of the Streams code in separating real packet streams, we set up an experiment using the classes in Figure 3. The three flows are routed into the CBQ filter which exists on a dual-homed host with both ethernet outputs on the same physical segments. The flows are then routed to their target hosts via an ordinary cisco router. The return path does not pass through the CBQ filter. The medium priority TCP flow starts first, followed twenty seconds later by the low priority TCP flow. After a further twenty seconds, we get a minute of high priority constant bit rate UDP packets. The constant bit rate traffic is intended to simulate audio traffic.

The data gathered from the tcpdump were analysed and can be seen in Figure 8. The data are clumped in five second buckets for clarity. It is important to realise that the constant bit rate stream suffered no loss, whilst the TCP streams incurred loss when the congestion window was opened too large. In addition, the streams are kept to within some margin of their allocated shares of the 62.5 KByte/s that we have set the virtual Interface to.

The time series illustrating the delays suffered by the packets, measured by the time difference between the IP packets entering the filter and emerging on the wire again can be seen in Figure 9. The data are bucketed in 1 second buckets. The high priority CBR stream suffers low delay, whilst the TCP streams suffer variable delays, dependent upon whether they are underlimit and thus transmitted in priority order, or overlimit and thus transmitted after any underlimit classes with traffic to send have been sent.

Non-intuitively the average delay experienced by the higher priority TCP class is larger

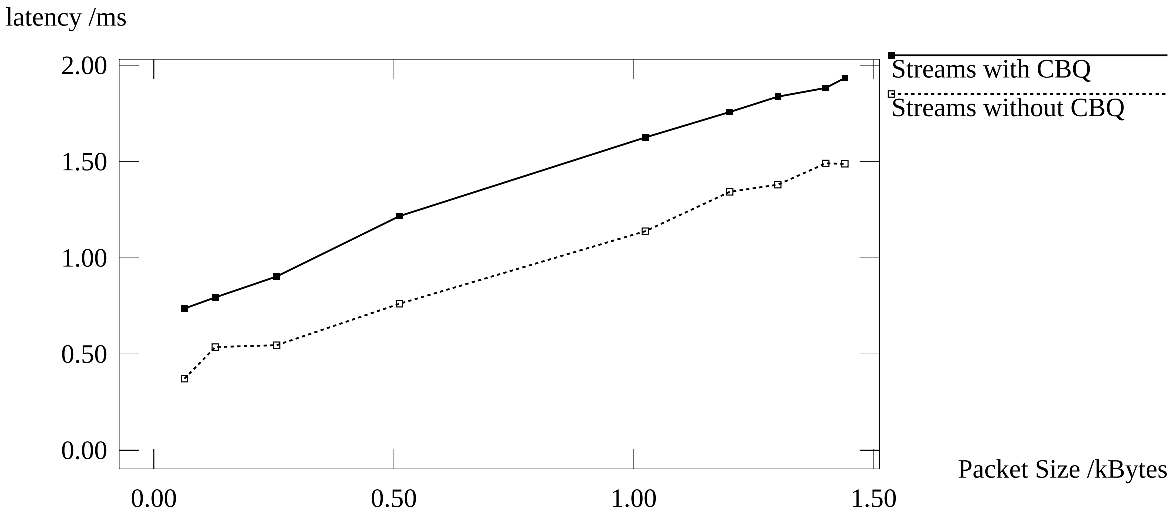

Figure 6: Latency of Streams module against size of packet

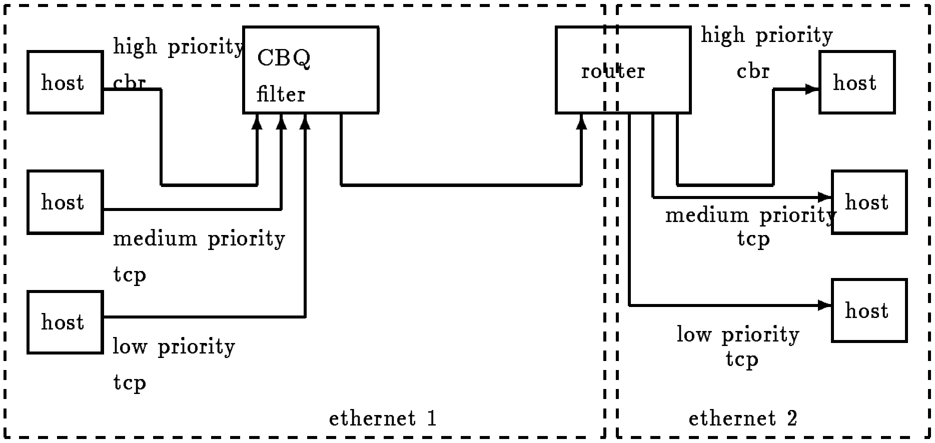

Figure 7: Experimental Setup for Testing the Streams Module

than the average delay of the lower priority TCP class. This is an artifact of the particular borrowing strategy we've implemented. Packets that are sent in the higher priority TCP class when this class is underlimit are at higher priority and thus suffer low delay. When this class is overlimit, it has greater claim on unused bandwidth than any other overlimit class, but at lower priority than any underlimit class, so the packets sent using this bandwidth have a higher delay, contributing to an overall higher delay $(\bar{delay} = (delay_{low} \times packets_{low} + delay_{high} \times packets_{high})/packets_{total})$. Alternative borrowing schemes are detailed in [1].

5 Conclusion

We have described an implementation of the class based queueing strategy to provide link sharing, suggested by Sally Floyd and Van Jacobson. The Streams installation provided a modular environment in which to work. However, a number of factors proved a hindrance. The Solaris 2.3 release currently does not allow modules to be downloaded dynamically below the IP code. This slowed the development somewhat. In addition, some problems were encountered in determining when the packet had left the output driver.

The code is currently in use on part of the traffic passing over the UK-US FATpipe. This partitions traffic amongst a number of agencies, and allows the guaranteeing of multicast traffic for real-time applications a minimum of bandwidth and low delay scheduling.

We plan to extend this work to optimise the lookup engines and to allow more sophisticated pattern matching within the classifier. A remote management option will be added so that classes can be added, changed and deleted dynamically, without having to download all classes again. With this in place, we shall be able to experiment with real-time reservation, such as RSVP [18] and traffic management within the Internet.

An unsupported version of the code can be found on ftp://cs.ucl.ac.uk/darpa/cbq.tar.Z.

Acknowledgements

We are deeply indebted to Joel Halpern and Paul Francis for their code and help on Patricia derived algorithms, to the folks at University of London Computer Centre, SuraNet and BBN for their work and help in putting the CBQ code on the FATpipe and to Greg Minshall and the anonymous referees for their helpful comments.

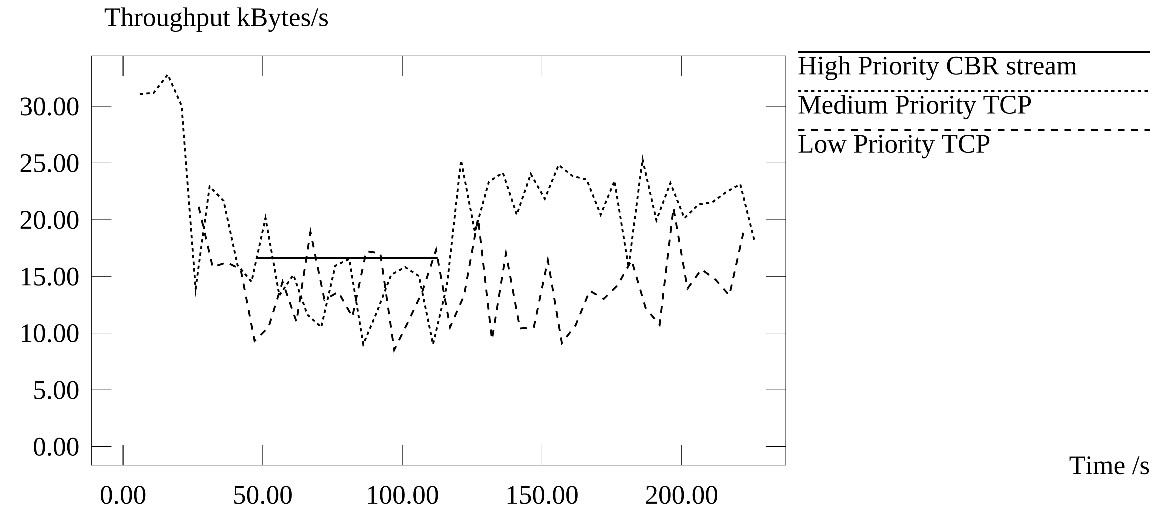

Figure 8: Throughput for the illustrative Streams Experiment

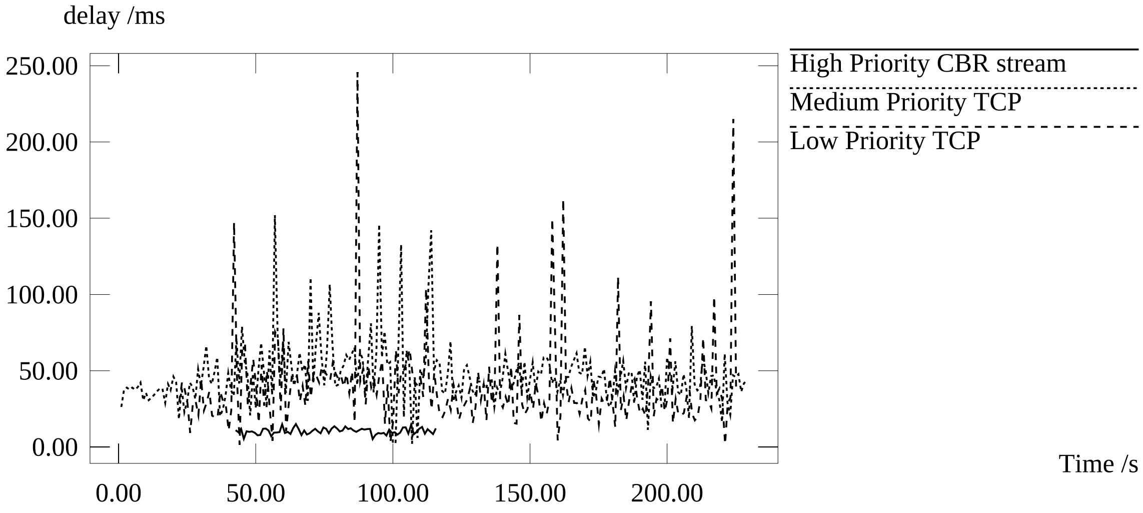

Figure 9: Delay for the illustrative Streams Experiment:Average delays - CBR stream - 0.01s, high priority TCP - 0.045s, low priority TCP - 0.038s

References

[1] Sally Floyd, "Link-sharing and Resource Management Models for Packet Networks," *Submitted to ACM/IEEE Transactions on Networking*.

[2] Sally Floyd & Van Jacobson, "Class based queueing for policy based resource sharing," 1993, Internal presentation and private email.

[3] R. Braden, D. Clark & S. Shenker, "Integrated Services in the Internet Architecture: an Overview," *rfc1633* (September 1994).

[4] Ian Wakeman, "Packetised Video: Options for interaction between the User, the Network and the Codec," *The Computer Journal* 36,1 (February 1993).

[5] Ian Wakeman, Dave Lewis & Jon Crowcroft, "Traffic Analysis of trans-Atlantic traffic," *Computer Communications* 16 (June 1993), 376,388.

[6] Van Jacobson, "Congestion Avoidance and Control," *Proceedings ACM SIGCOMM Symposium* (August 1988).

[7] Ian Wakeman & Jon Crowcroft, "A Combined Admission and Congestion Control Scheme for Variable Bit Rate Video," *To be published in Journal of Distributed Systems Engineering* (October 1992).

[8] Wang & Crowcroft, "A New Congestion Control Scheme: Slow Start and Search (Tri-S)," *Computer Communications Review* 21 (January 1991), 32–43.

[9] Lawrence S. Brakmo, Sean W. O'Malley & Larry L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," *Proceedings of ACM Sigcomm94*, London (September 1994).

[10] D. Comer, *Interworking with TCP/IP, Principles, Protocols and Architecture*, Prentice Hall International, ISBN 0 13 468505 9, 1988.

[11] Donald R. Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded In Alpha-numeric," *Journal of the ACM* 15,4 (October 1968).

[12] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1988.

[13] Joel Halpern, "Modifications of Patricia Trees for Handling values with discontiguous masks with emphasis on internet routing applications," *Private Communication*.

[14] Paul Tsuchiya, "Cecilia: A Search Algorithm for Table Entries with Non-contiguous Wildcarding," *ftp://thumper.bellcore.com/pub/tsuchiya /cecilia.tar.Z* (January 1992).

[15] Steven McCanne & Van Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," *Proceedings of Winter Usenix Conference*, San Diego Ca. (January 1993).

[16] Sally Floyd & Van Jacobson, "Random Early Drop Gateways," *IEEE/ACM Transactions on Networking* 1,1 (August 1993), 397,413.

[17] Van Jacobson & Steve McCanne, "TCP-DUMP," *ftp://ftp.ee.lbl.gov/tcpdump-*.tar.Z* (1990).

[18] L. Zhang, S. Deering, D. Estrin, S. Shenker & D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network* (September 1993.).