The University of Sydney

Hierarchical Fair Queuing

Technical Report Number 506

May, 1996

David Hogan

ISBN 1 86451 024 2

Basser Department of Computer Science
University of Sydney NSW 2006

Hierarchical Fair Queueing

David Hogan
dhog@plan9.cs.su.oz.au

Basser Department of Computer Science
University of Sydney

ABSTRACT

In this paper, we present a new queueing algorithm for networks, called Hierarchical Fair Queueing (HFQ). This shares the bandwidth between classes of users, and (within each class) between subclasses. HFQ provides a guaranteed share of bandwidth to each class, but bandwidth is not wasted when a class is inactive. Unlike the Link Sharing scheme of Floyd-Jacobsen, HFQ is provably fair, in that throughput is a close approximation to the ideal sharing scheme for an infinitely divisible resource. HFQ extends the techniques of Fair Queueing of Demers-Keshav-Shenker, to deal with a hierarchy of classes. HFQ also provides bounded delays for real time traffic. We implemented HFQ as part of a real IP gateway, and performed a number of tests which demonstrate its fairness.

1. Introduction

The design of a service discipline to use in network gateways has received much attention in recent years. The traditional First Come First Served (FCFS) discipline used in most gateways has proved to be inadequate. With best-effort data applications, it allows ill-behaved users to obtain excessive bandwidth and starve others whose requests are more moderate. Similarly, FCFS does not provide a guaranteed rate of service to real-time multimedia applications. A number of alternatives have been suggested, some designed specifically for best-effort data applications, some for for real-time applications, and some for both.

Recent work by Floyd and Jacobsen [4] and colleagues [18] has drawn attention to the case where bandwidth on a link must be shared between users who are arranged in a hierarchy of classes. At each level of the hierarchy, the classes have predefined *shares*, such that the bandwidth allocated to the parent class is to be divided among the children in proportion to their shares. For example, the top-level classes might represent organisations which have contributed to the cost of the link; the shares then represent the financial stake of each class. At a lower level, the classes might correspond to different protocol suites; with different end-point pairs defining the lowest level. Of course one could permanently reserve part of the bandwidth for each class, but this is very wasteful when a class is inactive. Instead we seek to divide the service fairly between those classes which are requesting it at any time.

One can easily recognise some service mechanisms as *unfair*; for example, FCFS will allow a class to obtain more service, simply by being more greedy in its requests. However, to allow rigorous evaluation of proposed queueing algorithms, we need a precise definition of fairness. For the case where there is a single level of classes, rather than a hierarchy, the natural definition was proposed (in the context of CPU scheduling) by Kleinrock [11]. Kleinrock's Processor Sharing model (PS) is an idealised, fluid flow model which serves as an intuitive definition of fairness: all active users are serviced at an equal rate, which requires that the resource be arbitrarily divisible. For situations where different users are actually entitled to different proportions of the resource, PS readily generalises to include a concept of weights which control the division of the instantaneous bandwidth. In this paper we present a natural extension of PS to deal with hierarchical classes. We call this Hierarchical Processor Sharing (HPS).

Of course, a realistic queueing algorithm needs to choose a single packet to send, and then wait till

that has finished before choosing another. For a single level of classes, the Fair Queueing (FQ) service discipline [2] has been proposed. This technique amounts to running a simulation of PS, and choosing the next packet based on the start (or finish) of transmission within PS. FQ approximates PS closely, as proved in [5]. It is trivial to generalise FQ to include shares, as in [2]. This generalisation is commonly called Weighted Fair Queueing (WFQ).

The limitation of WFQ is that it is not hierarchical. That is, it is designed only to share bandwidth between a set of users (or classes of traffic), and provides no facilities for splitting the total bandwidth between a bunch of organisations, then dividing each organisation's bandwidth between users in that organisation. This is not equivalent to dividing the total bandwidth amongst all the users; HPS does not degenerate to the non-hierarchical case.

The **main contribution** of this paper is to propose an algorithm called Hierarchical Fair Queueing (HFQ). This uses ideas inspired by WFQ to solve the problem of sharing a link fairly between users arranged in a hierarchy of classes. HFQ does approximate HPS to within a (theoretically guaranteed) bounded error. The theory behind the algorithm will be described in a future paper; this paper concentrates on the definition of the algorithm, and its implementation and testing in a real TCP/IP system.

As an example of why hierarchical allocations are important, consider a hypothetical situation where two organisations share the cost of a network connection. We will suppose for concreteness that they both pay equal portions of this cost. It seems reasonable that they should receive equal shares of the bandwidth, when they are both presenting sufficient requests, and further that this should be independent of the actual composition of the classes of users making the requests, or the requests themselves. Thus we wish the bandwidth allocations to the two companies to approximate PS, treating the two companies as ''users''. We could do this with WFQ, but then within each company the requests are serviced in FCFS order, making the network unfair to the individual users. Hence it is desirable to further subdivide the bandwidth of each company amongst the individual users, according to PS. This gives us an instance of HPS with a two level hierarchy. There is no reason to stop with two level hierarchies, either our company may have departments, which are further subdivided in one way or another; also, we may wish to subdivide each user's requests according to connection, and even discriminate between different classes of connection. Thus we see the usefulness of an HPS system which allows a general hierarchy.

This work also has implications for real-time service disciplines. Here the usual approach is to assume that each connection has made prior negotiations for a particular proportion of the available bandwidth. The role of the service discipline here is to provide each connection with its reserved bandwidth (when it actually needs it), generally with some kind of constraint on the delay, which may be included in the negotiation process. If a connection tries to use more than its allocated bandwidth, it may be subject to packet loss or to delays worse than allowed by the constraint. In real time applications, it is usually the variation in delay (also known as jitter) which is important, rather than the delay itself. Large variations may result in buffer starvation or overflow at the receiving end of the real time data. Keeping the delay bounded guarantees that the variation is bounded too. Of particular interest here is Virtual Clock [21]. This discipline is actually know to be equivalent to WFQ, under a suitable mapping between allocated bandwidths and shares. Virtual Clock provides no explicit control over delays or jitter. However, it has been shown [3] that under the assumption of leaky bucket constrained arrivals, VC provides bounded delays. The bounds are dependent on the allocated bandwidths. Since HFQ is a generalisation of WFQ, we expect similar results to apply to it. Other disciplines exist which decouple the delay bounds from the bandwidth allocations, providing separate control over each, eg delay-EDD, jitter-EDD, etc [20]. Of course, if we are only interested in delay bounds, then WFQ (or some variant) will do; the advantage of HFQ is that it adds flexibility for bandwidth allocation without losing the ability to provide delay bounds.

It is tempting to think that we could achieve the same result as HFQ by combining a number of WFQ schedulers. This would have the added bonus of allowing us to take advantage of techniques for implementing WFQ efficiently, which do not generalise to HFQ. However, it is unclear how such a combination of WFQ schedulers could actually be produced. Existing Fair Queueing algorithms are based on the assumption that the resource that they provide access to runs at a constant rate, which breaks down when we try to ''nest'' WFQ schedulers. This considerably complicates the implementation of such a system. In addition, we have doubts that such a system can provide error bounds comparable to those provided by HFQ.

Another competitor for HPS/HFQ, as far as hierarchical bandwidth allocation algorithms go, is the Link Sharing methodology [4]. Rather than providing a specific algorithm, Link Sharing specifies a general framework for combining real-time traffic scheduling with bandwidth control. The bandwidth control provided by Link Sharing is only applied at times of congestion. An actual algorithm which implements this kind of policy is Class Based Queueing (CBQ), described in [18].

We believe that HFQ provides superior bandwidth control to Link Sharing, because HFQ is an approximation to the ideal discipline, HPS, with the concept of sharing instantaneous rate. In contrast Link Sharing is based on average rate over some interval, typically measured by a decaying average. This makes Link Sharing unfair in the presence of traffic patterns tailored to the particular interval length or decay rate.

In section 2, we explore the background literature in more detail. In section 3, we describe the HPS service discipline, with examples. In section 4, we give an informal summary of how the HFQ algorithm works (the detailed description is relegated to the appendix). Section 5 describes our practical implementation, including some empirical results obtained from it.

2. Background

2.1. Processor Sharing and Fair Queueing

Processor Sharing (PS) is defined by Kleinrock [11] as a limiting case of Round Robin (RR) where the service quantum is allowed to approach zero. This limiting case forms an idealised, ''fluid flow'' service discipline, in which at every instant of time, all active users are simultaneously serviced, with equal (instantaneous) rates. Thus, if the resource services requests at rate r, and there are k users active at time t, then each of these active users receives an instantaneous rate of service r/k. Using this as our definition of PS, and given some arrival sequence, we may compute each user's cumulative service as a piecewise linear function of time. See [5] for further details.

Of course we cannot use PS as a practical service discipline, since we cannot, in general, service all the active users simultaneously. However, PS encapsulates the intuitive concept of ''fair service'' simply and precisely. Furthermore, it has the useful property that for a given number of users (say N), each user has a guaranteed minimum rate of service, r/N, irrespective of the behaviour of the other users (ie this is the worst case value). This lower bound on throughput implies an upper bound on delay for a given user's arrival sequence, irrespective of all other users' arrivals. These are very desirable properties. They become even more desirable when we generalise PS, as described below, since they then provide a useful, practical method of bandwidth allocation.

Since we cannot implement PS directly, we are interested in ways of approximating PS service with a practical, non-preemptive service discipline, such that the differences in cumulative throughput and delay remain bounded. This restriction is quite important, as it transfers the guarantees provided by PS to the practical algorithm, albeit in a slightly weakened form.

Fair Queueing (FQ) [2] was developed in response to the need for an approximation to PS. It is in some sense the best possible approximation. As shown in [5], the error in the cumulative throughput is at most the value of the maximum packet length. No smaller bound is possible with a non-preemptive approximation.

The basic idea of FQ is to order the requests by their finishing time in PS, essentially by using an embedded simulation of PS. Sufficient state information is kept which enables the finishing times to be determined. An abstraction called ''virtual time'' helps us here it is a monotonic function of time, with the useful property that it maps each packet's starting and finishing times into values that can be determined as soon as the packet has arrived [2]. This makes it possible to implement FQ quite efficiently [10]. It is shown in [5] that starting times can be used instead of finishing times, and that in either case the error in approximating PS remains bounded.

As defined so far, PS and FQ both assume an ''equal rights'' policy for all of their users, in that all users are to be serviced at equal rates. A trivial extension is to introduce per-user numbers, referred to variously in the literature as weights, priorities, or shares (we will call them shares), which represent the users' relative entitlements. That is, the instantaneous rate of service of each active user is proportional to its shares. To calculate the instantaneous rate of an active user, we divide their number of shares by the sum of

the shares of all the active users. This gives us their proportion of the bandwidth, which we multiply by the resource capacity to get their actual instantaneous bandwidth. FQ generalises to this case easily, as described in [2], and provides bandwidth guarantees with control over the individual allocations, by changing the shares.

2.2. Link Sharing and CBQ

Link Sharing [4] is a general framework for providing hierarchical bandwidth sharing in a network gateway. Class Based Queueing, aka CBQ [18], is a particular algorithm designed within the context of this framework. This is analogous to HPS and HFQ, in that Link Sharing provides a policy, and CBQ implements it. However, the policy provided by Link Sharing is much more loosely defined.

Link Sharing operates on a hierarchy of user classes, each of which has has attributes controlling its allocation. Whereas HPS has a single attribute per class (the number of shares), Link Sharing has multiple attributes, and is a more complex model.

Before describing the Link Sharing model in more detail, we must first describe the conditions under which the model is applied. Unlike HPS, which is meant to be applied at every instant in time, Link Sharing is only applied when the system is congested. Rather than having a single scheduler, Link Sharing stipulates two: a general scheduler, to be applied when the system is not congested, and the Link Sharing scheduler, which is applied when the system is congested. The behaviour of the general scheduler is not defined by the Link Sharing model, although some of the possibilities for its behaviour are described in [4]. In practice, there may really be one scheduler, with the functions of both ''schedulers'' intertwined. This is the case with CBQ. It could be argued that HPS/HFQ does nothing when the system is not congested, however the definition of congestion that must be used is different in this case. If more than one user is active at a given time in HPS/HFQ, the system is considered ''congested'', and the discipline acts appropriately, ensuring bandwidth and delay guarantees. The same execution may be considered ''uncongested'' by Link Sharing, which evaluates the congestion status over a period of time. Thus Link Sharing relies on the general scheduler to provide delay guarantees when the system is not seen to be congested.

The actual allocation model employed by Link Sharing is considerably different to that of HPS. Bandwidth allocation is controlled by absolute bandwidth allocations to each node, rather than relative weights. Typically the root node is allocated 100% of the bandwidth, and each internal node's allocation is equal to the sum of its children's allocations. This allocation is static and long-term, rather than dynamic and instantaneous, as in HPS. That is to say, it is intended that Link Sharing provide each node with its specified allocation over a long-term interval, assuming sufficient demand. When there are nodes which do not make sufficient demands, Link Sharing has to redistribute the surplus somehow, to any nodes requesting more than their share. The exact behaviour is not specified by Link Sharing, although it is stated as a secondary goal that the scheduler should ''do something reasonable''. We believe that the instantaneous, relative allocations provided by HPS provide the most reasonable behaviour for such situations. Furthermore, we note that the concept of ''sharing over an interval'' is not well defined, in that it depends on the size (and position) of the interval.

Link Sharing has other per-node attributes besides the bandwidth allocation. In particular, a priority is assigned to each node, which provides some control over the delay allocation. Real-time traffic can be given a higher priority, to improve its delay characteristics. HFQ does not provide such a feature, although it should be possible to add one. Link Sharing also provides the ability for a node to nominate a class which it can borrow from, in the event that it exceeds its allocation. Different actions may be defined for nodes to execute when they are overlimit. There are also flags for making nodes as exempt, bounded, or isolated. An exempt node is not affected by the Link Sharing scheduler. If effectively receives an allocation of 100%. A bounded class cannot receive more than its allocation. An isolated class does not borrow from the rest of the tree, outside of its descendants, and vice versa. Such features are not provided by HPS/HFQ, which is a simpler model. It could be argued that they are unnecessary and/or undesirable. In the case of the ''exempt'' attribute, this kind of behaviour can be approximated in HPS by giving that node a very large number of shares, and making in a child of the root node. This approach is actually preferable, since it does not completely lock out other users in the event that the node actually tries to use its allocation. The ''bounded'' attribute could be simulated by introducing admission control. We consider the ''isolated'' attribute to be undesirable, because it implies that the service discipline has to be non-work

conserving.

2.3. Problems with Long-Term Sharing

Any system which performs bandwidth allocation must include in its design, either explicitly or implicitly, a definition of rate. When a user is allocated a particular share of a resource, they are being given access to that resource at a particular rate.

Rate is not a very well defined concept. Multiple definitions of rate are possible; the two main ones are instantaneous rate, and average rate over an interval. When instantaneous rate is used as the basis for bandwidth allocation, idealised service disciplines such as PS and HPS result; we refer to these collectively as *Instantaneous Sharing*. When the average rate over an interval is used, a model is obtained which we will refer to as *Long-Term Sharing*. Long-Term Sharing is the model used by Link Sharing [4], CBQ [18], Cambridge Share [12], [13], and Basser Share [8], [7], [1], [9]. It suffers from the following flaws: *arbitrariness*, *unpredictability*, and *unfairness*.

Long-Term Sharing is arbitrary, in that the size of the interval, and the type of averaging used, are completely arbitrary, and different choices for these parameters yield different results. There are two main types of averaging used: arithmetic mean, and exponentially decayed averaging. In the case of the exponentially decayed average, the decay rate replaces the interval lengths as the variable parameter. Most practical systems based on Long-Term Sharing use the exponentially decayed average (eg Basser Share, CBQ) although the arithmetic mean is often invoked as an informal justification.

We will give an example of the arbitrariness of this method. Consider a system with two users with equal shares, who are continuously backlogged with packets requiring 1 second of service each. Suppose that in a sample execution, the packets are serviced in the order "1 2 2 2 1 1 1 1". Let us now analyse this using the arithmetic mean model, with two different interval lengths. If we use an interval length of 8 seconds, we see that user 2 only receives 3/8 of the total bandwidth, and thus it appears that user 1 is being favoured. If however, we use an interval of 4 seconds, user 2 receives 3/4 of the bandwidth in the first interval, but none in the second. This example shows a lack of consistency the interpretation of an experiment depends on the sampling interval chosen. The situation is no better for the decayed average method.

Our second objection to Long-Term Sharing is that it is unpredictable. In particular, systems based on decayed averages tend to be too difficult to analyse in full generality (as is possible with Instantaneous Sharing).

Lastly, we claim that Long-Term Sharing is unfair. Consider again a system with 2 users who have equal shares. Suppose that user 1 is continually active, while user 2 only becomes active halfway through the sampling interval. According to the arithmetic mean model, we have to give user 2 undivided attention for the remainder of the interval. During this time, user 1 will experience massive delays (assuming the interval is long enough), while the system only services user 2. It is as if user 1 is punished for having used the resource when nobody else was! Again, the decayed average method exhibits the same unfairness.

For these reasons, we believe that Long-Term Sharing is inferior to Instantaneous Sharing, a la PS and HPS, which is intuitively fair, predictable, and independent of the interval over which its behaviour is observed.

2.4. Hierarchical Classes in CPU Schedulers

There is a close analogy between mechanisms for sharing a network link, and those for scheduling processes in a CPU. There have been a number of hierarchical CPU schedulers eg: Cambridge Share [12], [13], Basser Share [8], [7], [1], [9], the VSTa scheduler [17], and Lottery Scheduling [19]. None of these is based on approximating an ideal fair discipline; Cambridge Share and Basser Share are both based on Long-Term Sharing, and VSTa and Lottery Scheduling use Monte Carlo methods, hence do not provide guarantees. Restricted Fairness results were found for Basser Share in the steady state [6], but little is known about the general case, except that it suffers the drawbacks described in the previous section.

3. Defining Fairness Precisely for Hierarchical Policies

Hierarchical Processor Sharing (HPS) is a generalisation of PS: not only do we allow different weights to be assigned to each user, but we also generalise the flat array of users, replacing it by a tree. Each node of this tree has a weight (although the root node's weight is never used). The leaf nodes are the actual users. The rule for calculating instantaneous rates of service is a straightforward recursive application of that of (weighted) PS: starting at the root, with the full rate of the resource as its allocation, we recursively divide each interior node's allocation amongst its active children in proportion to its shares, until each active user has been assigned a portion of the bandwidth. An internal node is defined to be active if any of its descendants are active.

An example will make this clearer. Consider the tree in figure 1. This consists of 6 leaf nodes, labelled user0 through user5, and 3 internal nodes: root, Group1, and Group2. The weight of each node is recorded next to the link from that node's parent (the root node's weight is of course irrelevant). Underneath the weight is the percentage of total bandwidth that each node would get if all the users were active. Since this makes all the internal nodes active, we can easily calculate user0 = 50%, Group1 = 20% and Group2 = 30%. Now the allocation given to the groups is further parcelled out according to shares, and the result is as shown.

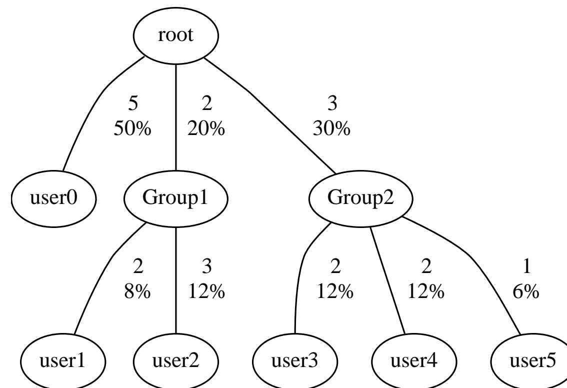

Figure 1: Example of HPS (all users active)

In figure 2, we have the same basic structure, but users 0, 2 and 4 are now inactive (represented by dashed outlines). Each of these now receives 0%. No internal nodes have been inactivated in this case. We just hand out the bandwidth as before, as if the inactive users were not there, and the result is as shown.

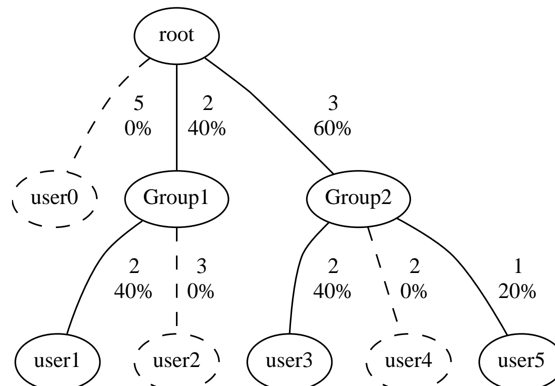

Figure 2: Example of HPS (some users active)

We will give one further example. Consider figure 3, where only users 0 and 3 are active. This makes Group1 inactive, and we see how inactivity propagates up the tree. Allocations are user0 = 62.5%,

Group2 = 37.5%, and thus user3 = 37.5%, with all the inactive users receiving 0%.

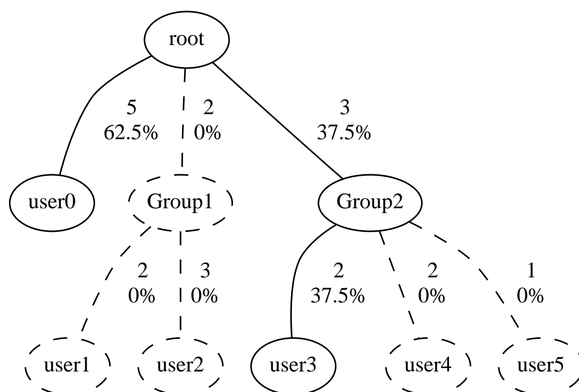

Figure 3: Example of HPS (even less users active)

Like PS, HPS provides guaranteed bandwidth allocations. It is quite straightforward to calculate a user's minimum guaranteed bandwidth: assume that every user is active, and calculate bandwidth allocations as above. This gives us every user's minimum guaranteed bandwidth observe that in the cases where not every user is active, making them all active reduces the bandwidth allocation for those users who were already active.

Since HPS provides guaranteed bandwidth allocations, it also provides delay guarantees for suitably constrained flows. If a user's arrival process is constrained by leaky bucket, with an average rate value equal to the minimum bandwidth guarantee, then it follows that the user's queue length at the HPS discipline will be bounded, and, as the transmission time is also bounded (since there is a minimum rate) the total delay experienced by one of the user's requests is bounded.

HPS is thus a very desirable service discipline to approximate. Not only does it provide bandwidth and delay guarantees, as did PS (and thus FQ), but it also provides the added flexibility of hierarchical bandwidth allocation. We have developed an algorithm called Hierarchical Fair Queueing (HFQ), which basically consists of ordering the requests by their starting times in HPS. We have shown that this yields bounded approximations to delay and cumulative bandwidth (note that the actual bounds depend on the maximum request length, and the values of all the weights). This makes HFQ an algorithm of practical significance for network queueing, providing hierarchical bandwidth allocations, and guaranteed behaviour for both bandwidths and delays.

4. The HFQ Algorithm

In this section we will explain the basic ideas behind the algorithm. A detail description is contained in the Appendix. The goal of the algorithm is to approximate the HPS service discipline, which, as we have already mentioned, we cannot implement directly, due to its requirement that we service all active users simultaneously. Instead, what we do is to simulate the HPS discipline, using the actual arrival times and packet lengths as inputs to the simulation. The simulation is used to calculate the starting time of each request under HPS. We then service the actual requests in the order given by the starting times under HPS. Note that we have used starting times, rather than finishing times, as in FQ [2]. This is because we have theoretical results which describe the behaviour of the system when we use starting times, but no corresponding results for the system which uses finishing times. Contrast this again with FQ, for which the behaviour is known for both variants [5].

Our ability to simulate HPS depends on the availability of the arrival time information which forms the input to the simulation. Note that we do not know a packet's arrival time until that time. This fact creates a problem which does not exist for the FCFS discipline. In the case of FCFS, packets are serviced in the order that they arrive in. Thus the arrival of a future packet cannot affect the starting time of any packets already in the system. HPS is not like that. Since HPS services all users simultaneously, the arrival of a new packet can change the starting times of packets already in the system, effectively by slowing down prior packets belonging to the same user. Thus there may be requests in the system for which we do not

know the starting time under HPS.

Note that in the non-hierarchical case, ie PS, the order of the outstanding requests is unaffected by new arrivals. This is related to the concept of *virtual time*, which is used in the implementation of Fair Queueing [2]. Although the real starting and finishing times of a request may not be known when it arrives, the corresponding virtual times are known. Since the mapping from real to virtual time is monotonic, FQ merely has to order its requests by virtual starting (or finishing) time. Only one queue is required in such a system: a priority queue sorted according to virtual starting (or finishing) time.

There is no analogous concept of virtual time in the general case of HPS. Implicit in its definition for PS is the fact that a pair of users under PS always receive service at rates with a fixed ratio when both users are active. This invariant enables the specifics of which users are active to be ''factored out'', resulting in the definition of virtual time. In HPS, the ratio is no longer fixed, eg consider the allocations given to users 1 and 3 in figures 1 and 2. Thus we do not have the simplification afforded by virtual time, along with its useful property that the ordering of requests may be determined as soon as they have arrived. It is in fact possible for a new request to change the order in which the currently queued requests should be serviced. Because of this, an HFQ implementation requires more than one queue.

A single priority queue is needed to hold requests which have already ''started'' in the parallel HPS simulation. These requests are held in order of HPS starting time. Requests which have not yet started in HPS are kept in separate, per-user queues. These queues are maintained in FIFO order. As the HPS starting times of packets on these per-user queues become known, they are stamped with their times, and placed on the priority queue.

It might be thought that a FIFO queue would be sufficient for holding the requests that have started in HPS after all, we are simulating HPS to obtain the ordering for the requests in HFQ, so those requests should be generated in the right order. However, our algorithm doesn't generate the HPS starting events one at a time, but rather calculates the longest period of time that it can get away with advancing the state of the simulation by, and generates all HPS starting events with a single sweep of the tree. This, naturally, generates the events out of order, but their associated times are known, and can be used to put the events back in order.

An important question is whether the priority queue can become empty even when there are requests in the system, such that the system is not work-conserving. This cannot happen, because it implies that the system is servicing requests at a rate faster than the HPS simulation runs at. Of course, it is crucial that the rate of the HPS simulation match the actual resource rate, otherwise the algorithm may not behave correctly.

A full description of the algorithm, including data structures and pseudocode, is contained in the Appendix.

5. An Experimental HFQ Implementation

5.1. The Test Environment

The HFQ algorithm was implemented as a modification to the user mode PPP server included with the Plan 9 Operating System [14]. Two PCs running Plan 9 were connected via a serial link operating at a rate of 9600 baud. One of these machines ran the program `aux/pppclient`, which is designed to give a machine with no other network connection access to the internet via the Point-to-Point Protocol (PPP) [16]. The other machine ran a modified version of `aux/pppserver`, which acts as an IP gateway, providing a PPP implementation on a machine which already has IP connectivity. The modifications were to replace the standard FCFS queueing behaviour of the PPP server with an implementation of HFQ. This second machine was also connected to an ethernet, thus acting as an experimental HFQ gateway.

Plan 9 has a novel PPP implementation. Rather than implementing it as part of the kernel, it is a user mode program. This is possible because of Plan 9's ''Streams'' implementation [15], [14]. Under Plan 9, many of the devices are streams, which are bidirectional data channels with an associated stack of protocol modules, known as streams modules. The streams modules are built into the kernel, but the commands to ''push'' them onto the streams can be issued from user mode. Thus, for example, to obtain IP access via ethernet, one opens the ethernet device, then pushes the `arp` and `internet` modules onto the stream.

The internet streams module acts as an IP protocol multiplexor, providing TCP and UDP streams through an associated device. Thus, one way to implement PPP under Plan 9 would be to add a new streams module to the kernel, which could be pushed onto a serial line stream before the internet module. The module would perform PPP encapsulation for packets passing in one direction, and unencapsulate the data flowing in the other direction. Rather than take this approach, Plan 9 implements PPP as a program. This program creates a pipe (a stream under Plan 9) and pushes the internet module onto one end. IP packets are read from the other end of this pipe, converted to PPP format, and written to the serial line. A separate process reads PPP packets from the serial line, turns them back into IP packets, and writes them to the pipe. Thus we have an implementation of PPP in user mode.

Using Plan 9's PPP implementation as our test bed for the HFQ algorithm has a major advantage over kernel-based approaches, in that the compile-run cycle time is greatly reduced, so debugging is made easier. Furthermore, being a user mode program, it has easy access to other services available in user mode, so that configuration of the HFQ system is made simpler.

5.2. The Implementation

The Plan 9 program aux/pppserver is written in a concurrent programming language called Alef [14]. It consists of a number of Plan 9 processes which share memory and communicate using message passing. In the case where there is just one serial line, there are three processes: ipmuxencode, which receives the ip packets from the system and writes them as PPP packets to the serial line, pppdecode, which takes incoming PPP packets and decodes them, and doalarms, which handles timer events. The program communicates with the kernel IP device by reading and writing a pipe which has the internet streams module pushed on the other end. Plan 9 is designed to handle multiple input sources using true concurrency, ie separate processes, rather than the select system call (there is none). Hence separate processes are actually required to read the pipe and the serial line. This is not a hardship in Plan 9, since processes are cheap, and Alef makes their use simple.

The point at which congestion occurs in pppserver is in the ipmuxencode process. Given that ipmuxencode must write each encoded packet onto a slow serial line, and that these packets can be supplied by the pipe at a much faster rate, ipmuxencode spends most of its time blocked, writing to the serial device. While this is happening, packets bank up in the pipe, which acts as a FIFO queue there is no explicit queueing in the process itself.

We modified pppserver by adding an additional process, writerproc, to handle writes to the serial line. This freed ipmuxencode to read the pipe as quickly as possible, with the purpose of enqueueing the packets on the queues maintained by HFQ, for writerproc to dequeue at its leisure. We considered using a 2 process structure, where each of these processes would participate in the HFQ algorithm, but decided that the locking and communication issues would complicate and obscure the algorithm. Instead, we put all of the HFQ operations in a third process, called fqproc.

Communication between the processes is via three channels. The first of these is used by ipmuxencode to pass new arrivals to fqproc. It sends a tuple (pkt, qnum) consisting of the new packet and the number of the queue that it is to be placed on, as determined by a packet classifier. The classifier must be run before the packet has been PPP encoded, in order for it to be able to recognise the packet. On the other hand, the PPP encoded version of the packet must be used in determining the packet length in the HFQ algorithm. ipmuxencode calls the classifier code prior to encoding the packet, and then sends the encoded packet along with the queue number to fqproc.

The other two channels are used for communication between fqproc and writerproc. There is one channel for writerproc to signal that it is idle, requesting a new packet to service, and a channel for fqproc to pass the next packet to writerproc.

fqproc loops endlessly, waiting for something from one of its input channels. It performs the actions described as new_arrival and next_service in the appendix (and the functions called by them). In the implementation, instead of next_service blocking and new_arrival waking it up, next_service actually waits for the next arrival itself, and calls new_arrival to process it.

Floating point arithmetic was used for all of the HPS-related variables, such as csrv and creq (see appendix for definitions). This requires some care, as numeric errors can cause the various comparisons

used in the algorithm to yield incorrect results, and the algorithm to behave erroneously. This actually happened in the early stages of testing, until a ''slop factor'' was added to some of the floating point comparisons. Further work is needed in applying principles of numeric analysis to make the algorithm more robust. This problem is also a potential pitfall for WFQ implementations.

The packet queues were implemented as linked lists. All the per-user queues are FIFO, so linked lists are the correct choice here. In the case of the priority queue, it may actually be better to use some other data structure, such as a heap, though further data is needed on the likely length of the queue in order to choose the best data structure. It was expected that this queue would remain relatively short, in which case there is little advantage in using a heap, but this may not be true for a production system.

Although the theory behind the algorithm guarantees that the priority queue will never be empty when there is work in the system, the implementation can not safely assume this. Slight numerical errors in the HPS simulation, in time keeping, and in the rate of the serial line, can cause the implementation to deviate from the ideal theoretical behaviour, and enter a state of temporary starvation. When this occurs, fqproc sleeps for one tick, and then tries again. It should probably make adjustments to the state of the simulation to bring forward the time of the next event.

5.3. Test Methodology

The tests of the implementation were performed using two programs, called `source` and `sink`.

The program `source` was run on the computer with the modified `pppserver`. It generates a series of packets to be sent to the machine on the other end of the PPP link. The details of the packets that are sent are controlled by a text file, which is given to the program as the first argument. Each line of the program describes a packet to be generated. The lines are expected to be in the following format:

t user seq len

where *t* is the arrival time in milliseconds, *user* controls the UDP port that the packet is sent on, *seq* is a sequence number to be imbedded in the packet, and *len* is the length of the packet. The file is assumed to be sorted in increasing order of arrival time. All packets are sent to UDP port 5555 on the remote machine (whose IP address is hardwired) using local port (3200+*user*). The HFQ implementation currently classifies UDP packets based on the last 4 bits of the UDP source port, allowing for up to 16 users. The time $t = 0$ is assumed to be when the program starts up. Source attempts to deliver the packets at the times specified, however the delays associated with writing to the network device, plus the possibility of blocking, mean that the actual arrival sequence may differ considerably from the requested one. For this reason, `source` writes the actual arrival sequence generated to standard output. This data is formatted in the same way as the input data, and is normally captured in a file and used instead of the input data when analysing the results.

The program `sink` was run on the other machine, which ran the unmodified `pppclient`. Its job is simply to bind to UDP port 5555, and capture all the packets sent to the port, producing a one line summary of each on standard output. The output is in the same format as the input to `source`. Here *t* is the time that the packet is received (with origin at the time that the first packet arrived), *user* is the last 4 bits of the UDP source port, *seq* is the sequence number extracted from the packet, and *len* is the length of the packet. The output of `sink` is captured in a file, which is used in conjunction with the output of `source` to analyse the experiment. There are a couple of subtleties associated with the use of `sink`. First, there is no way for `sink` to know when the test is over. It could be instructed to wait until a certain number of packets have been received, or certain set of sequence numbers are reached, but unfortunately packet loss does occur, and renders such methods unreliable. The method used was to manually watch the output file, and stop `sink` when it was observed to be quiescent. The other problem is in the synchronisation of the time between the two machines. Plan 9 has no facilities for doing this, particularly not at the millisecond level. The output of `sink` is such that the first packet is seen to be received at $t = 0$. This packet would typically have been sent by source at $t = 0$ in source's timescale. To synchronise the two timescales, the length of the first packet and the known rate of transmission were used to calculate an adjustment factor to be added to each of the times output from `sink`. Suppose the first packet is of length 960 bytes. This will take 1 second to transmit at 9600 baud. Thus if it is sent at time $t = 0$, it will arrive at time $t = 1$, which `sink` takes to be time $t = 0$. In this case, the adjustment consists of adding 1 to all times output by `sink`.

In general, we add $len_0/960$, where len_0 is the length of the first packet. This gives us the time that each packet finishes service. To get the time that a packet starts service, we can then add *len*/960, where *len* is the length of that packet.

Analysis of the results is complicated by the overheads in transmitting the packets. There are two sources of overheads: protocol headers, and HDLC encapsulation. A typical UDP packet will have an 8 byte UDP header, and a 20 byte IP header. Thus, to send *len* bytes via UDP, we actually have to send a total of $len + 28$ bytes. If we were using TCP, then we would also have to worry about the effects of Van Jacobsen compression, which save some of the overheads, however UDP is not compressed. Further overheads are added by PPP itself. PPP adds a header and a CRC to each frame that it sends, adding 7 bytes to the length of the data transmitted. Large packets will be fragmented by the IP implementation (Plan 9 PPP uses an MTU of 1500, the same as ethernet, thus avoiding further fragmentation at the PPP level when speaking to another Plan 9 machine). PPP uses HDLC framing to delimit packets on the serial link. This involves the use of a special character chosen as an escape character. If the escape character appears in the input, then it has to be escaped. This effectively adds a variable number of bytes to the length of the packet, based on the number of times that the escape code appears. This number is difficult to predict, though it can be minimised by initialising the buffer that source sends to (say) all zeros. Another overhead added by PPP is is the control information (LCP and IPCP), which basically constitutes a class of traffic which we are unable to measure, since it is internal to the PPP client and server. In summary, a packet of length *len* will take $(len + 28 + 7 + nesc)/960$ seconds to transmit, where *nesc* is the number of escaped characters in the completed PPP frame.

Our use of simple text files for the input and output of the test programs enables us to use awk scripts for much of the analysis, and also to generate the input data for the arrival sequences. Each experiment starts with an awk script which generates arrival sequences for each user, which is then piped through `sort -n`. An auxiliary program `poisson` is used to generate poisson arrival time distributions, and exponential distributions of packet lengths (where needed). The output of `sink` is processed by another awk script, called `fqgraph`, which calculates cumulative throughputs for each user and turns them into suitable data for the system `graph` command. A program called `psgraph` simulates the behaviour of HPS, given the output of `source`, and generates corresponding cumulative throughput data in the same format as `fqgraph`. Thus the behaviour of HFQ and HPS can be compared visually, by graphing their cumulative throughputs on a common set of arrival data. We are also interested in the delays experienced by a user with leaky-bucket constrained arrivals. The program/script `regulator` takes an arrival sequence and constrains it according to the given leaky bucket parameters. Packet arrivals which would occur too soon are merely delayed, not discarded. This program is typically used in conjunction with `poisson` to generate suitable leaky-bucket constrained arrivals. A script called `delay` is then used to calculate the delays experienced by each packet owned by a given user, and generate a graph against time.

5.4. Tests and Results

We will present results from two different tests here. The first test was designed to produce a relatively simple execution, where the changes in activity status are clearly seen, so as to illustrate the behaviour of the algorithm. The second test was designed to show how the algorithm provides bandwidth guarantees to all users, and delay guarantees to those users who are suitably well-behaved. From our theoretical work, we expect the difference in cumulative throughput for a user in HFQ and HPS to be bounded. By plotting the cumulative throughput for each user in the actual HFQ algorithm, and in an HPS simulation, we can verify whether their difference remains bounded. Similarly, we may plot the delay versus the arrival time for a number of users with a variety of arrival characteristics, and observe that the delay remains bounded when the appropriate leaky bucket constraints are observed. Thus, for the second test we allowed some users to be constrained by appropriate leaky bucket flow control, and others left unconstrained, the object being to show that delay bounds were maintained for well-behaved users, in spite of the presence of badly behaved users.

The class tree for the first test was as shown in Figure 4. There were four users (user1 through user4), and one group, which had two of the users as children. The other two users were children of the

root node.

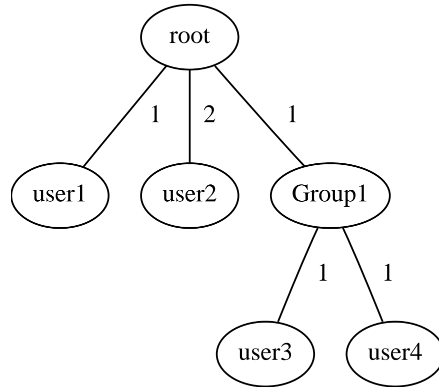

Figure 4: Class Tree for Test 1

A very simple set of arrival times was chosen for the test. Each user had a particular time at which all of its arrivals were to occur simultaneously. This starting time was at $t = 0$ for user 1, $t = 100$ for user 2, $t = 200$ for user 3, and $t = 300$ for user 4 (all times in seconds). Each request had a data payload of 1280 bytes. The total number of requests generated for user 1 was 200, for user 2 was 160, for user 3 was 109, and for user 4 was 128. These differences in starting time and number of requests were designed to cause a variety of activity states to occur, but without the full complexity that could occur if each user's arrivals were spread out in time.

Figure 5 shows the cumulative throughput graphs for the 4 users, both in the actual HFQ execution (represented by the solid curve) and for the separate HPS simulation via `psgraph` (dashed curve).

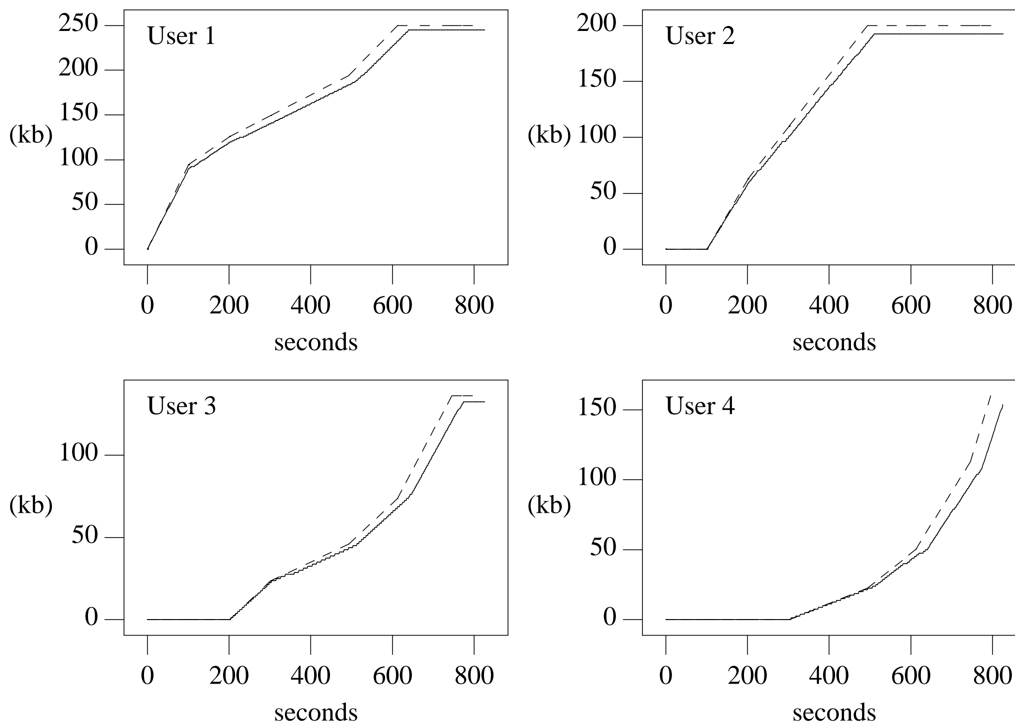

Figure 5: Cumulative Throughput for HFQ (solid) and HPS (dashed), test 1

Note that the curves are similar in shape, however do not match each other exactly. Aside from the fact that HFQ is at best an approximation to HPS, there are two other sources of discrepancy, corresponding in fact to the two axes of the graphs. Note how each solid curve comes to an end lower than its

corresponding dashed curve, and also further on in time. It is lower because packet loss occurred during the experiment. This packet loss was caused by receive overruns in the duart of the machine running `sink`. There was a surprisingly large number of these (around 70 for an 800 second test) given that the duart was only operating at 9600 baud. The overruns caused those packets containing the bytes that were lost to not make it past the error detection on the receiving machine, after using up some of the bandwidth.

The other discrepancy was caused by the fact that `psgraph` used the length of the data contents of each UDP packet as the request length, and did not take any of the overheads into account (whereas HFQ did). This could have been done naively by just adding the overheads to the request lengths, but this would have meant that the ordinate would have been further distorted, since for HFQ the ordinate does not take the overheads into account. `psgraph` was designed to perform its simulation as simply as possible, without keeping track of the individual requests (which were just added to a cumulative request tally on each simulated arrival), and to treat each user's cumulative throughput as a continuous fluid flow. Corrections for the overheads would not make much sense in this context. Thus no attempt was made to compensate for them, with the net effect that the HFQ curves appear to be running at a slightly lower baud rate than the HPS ones.

The two sources of discrepancy mentioned above yield errors of the right order of magnitude to explain the differences between the HPS and HFQ curves. Taking these into account, we can see that HFQ follows HPS quite closely. The slope of each HPS curve changes in response to changing conditions of user activity we can actually see the HPS policy in action.

The second test employed more realistic user arrival processes, where the arrivals tended to be spread out in time, and with particular rate and burstiness characteristics. A different class tree was used, with 6 users divided up between 2 groups, as shown in figure 6 (astute readers will note that it is based on one of the class trees used in [4]).

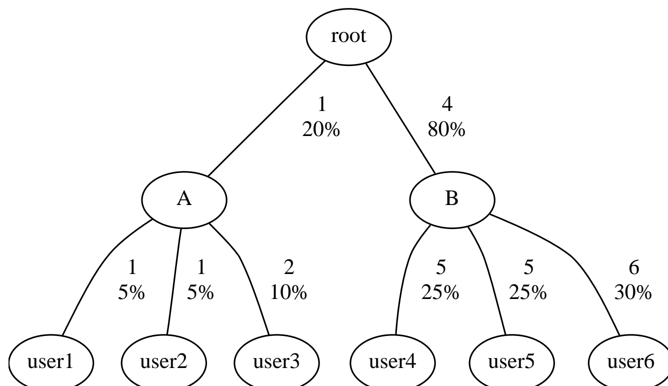

Figure 6: Class Tree for Test 1

Each link in figure 6 is marked with the corresponding number of shares and percentage of the total bandwidth that will be received by that user when all users are active, ie the minimum guaranteed bandwidth as a percentage. By multiplying this percentage by 960 bytes/second, we obtain the guaranteed bandwidth rate for each user. Each user was given a different arrival process, with some chosen to demand more than their share (on average) and others constrained to be within their guaranteed allocation. The following table summarises these arrival processes. User0 was a deterministic arrival process (with very low bandwidth), the others were based on poisson arrival processes with constant request lengths. Two of these were constrained by leaky bucket.

user	guaranteed bw	avg bw	max burst	length	avg interarrival	num pkts
1	48	10	-	1000	5000	240
2	48	48	2	1000	21584	54
3	96	200	-	1200	6245	92
4	240	120	5	1000	8741	137
5	240	300	-	800	2830	339
6	288	300	-	900	3163	364

The calculation of average interarrival time was made under the assumption that $nesc = 14$ (experimentally determined as a likely value). Since this is only an approximation, the average bandwidth demand of each user in the table is only approximate. In particular, user1 was supposedly operating at its guaranteed bandwidth, but may have gone over it. Given these inputs, we would expect to see low delays for users 1, 2, and 4, and high delays for the others, particularly user 3, who is trying to operate at over twice the guaranteed bandwidth.

Figure 7 shows the cumulative bandwidth under HFQ and HPS for each user. These remain quite close, except in the case of user 3, where significant deviation is present. This deviation is greatest around the time when the user's end-to-end delay is largest. Note that the discrepancy is still bounded, but the actual bound for this user is larger than the rest. Figure 8 shows the delay encountered by each packet, graphed against its arrival time. In interpreting this graph, note that the average rate of service in the experiment is about 1 packet per second, hence a delay of a few seconds is to be considered ''small''. The graph of user 1's delay shows a maximum delay of less than 4 seconds for this low volume (interactive) user. In the case of user 2, there is a single peak of almost 10 seconds, but all other packets remain lower than 4 seconds. This peak is probably caused by the arrival process briefly exceeding its guaranteed bandwidth. User 3's delay grows quite steadily to almost 100 seconds, then decays again. This shows quite dramatically how the system penalises users who try to obtain more than their fair share. User 4 is constrained to at most half its available bandwidth, and with little burstiness, and so we see that its delay remains bounded. Users 5 and 6 are both unconstrained, and trying to use slightly more than their guaranteed bandwidth. As user 4 is using less than its guarantee, it is easy for 5 and 6 to soak up the extra bandwidth, thus their delay remains relatively small (eg compared to user 3). A few major peaks, particularly for user 5, occur as a result of their lack of constraint.

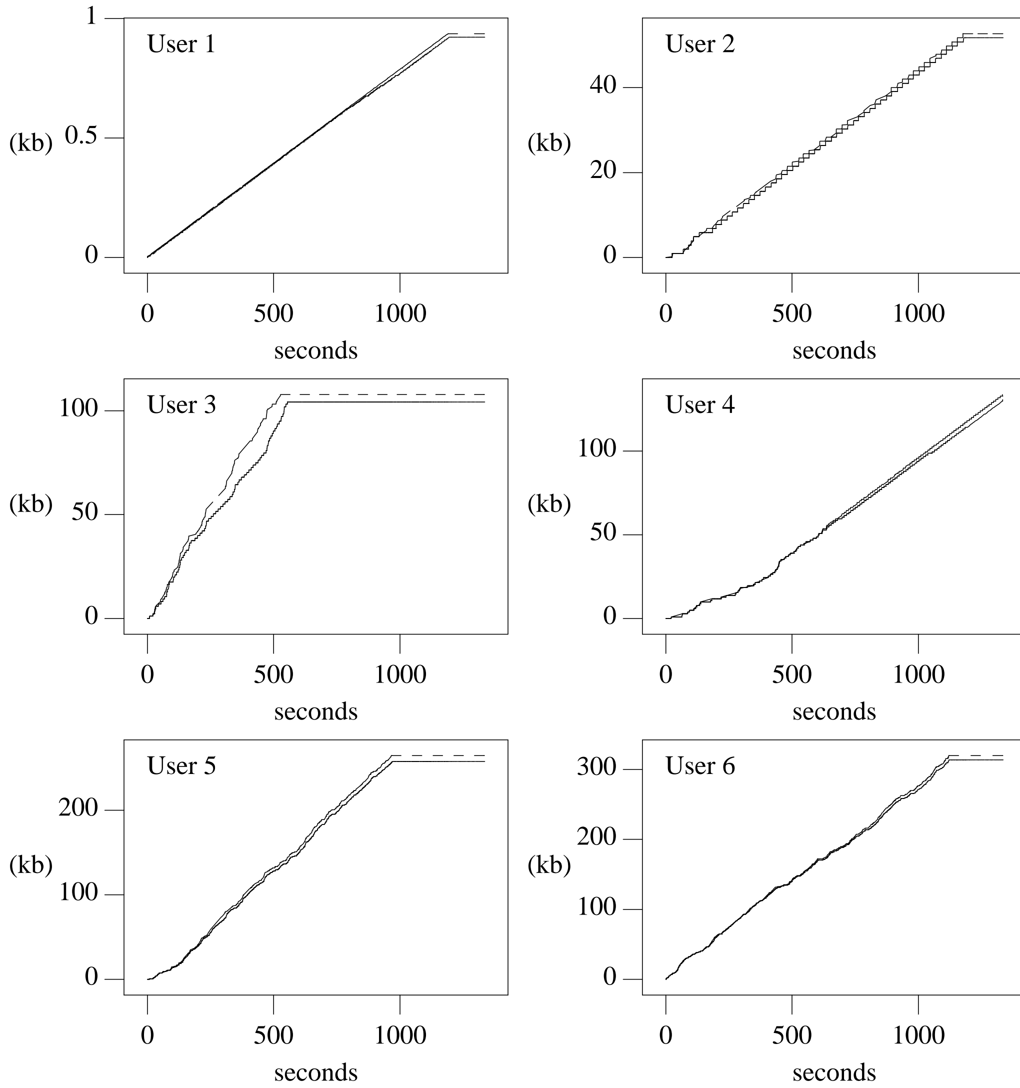

Figure 7: Cumulative Throughput for HFQ (solid) and HPS (dashed), test 2

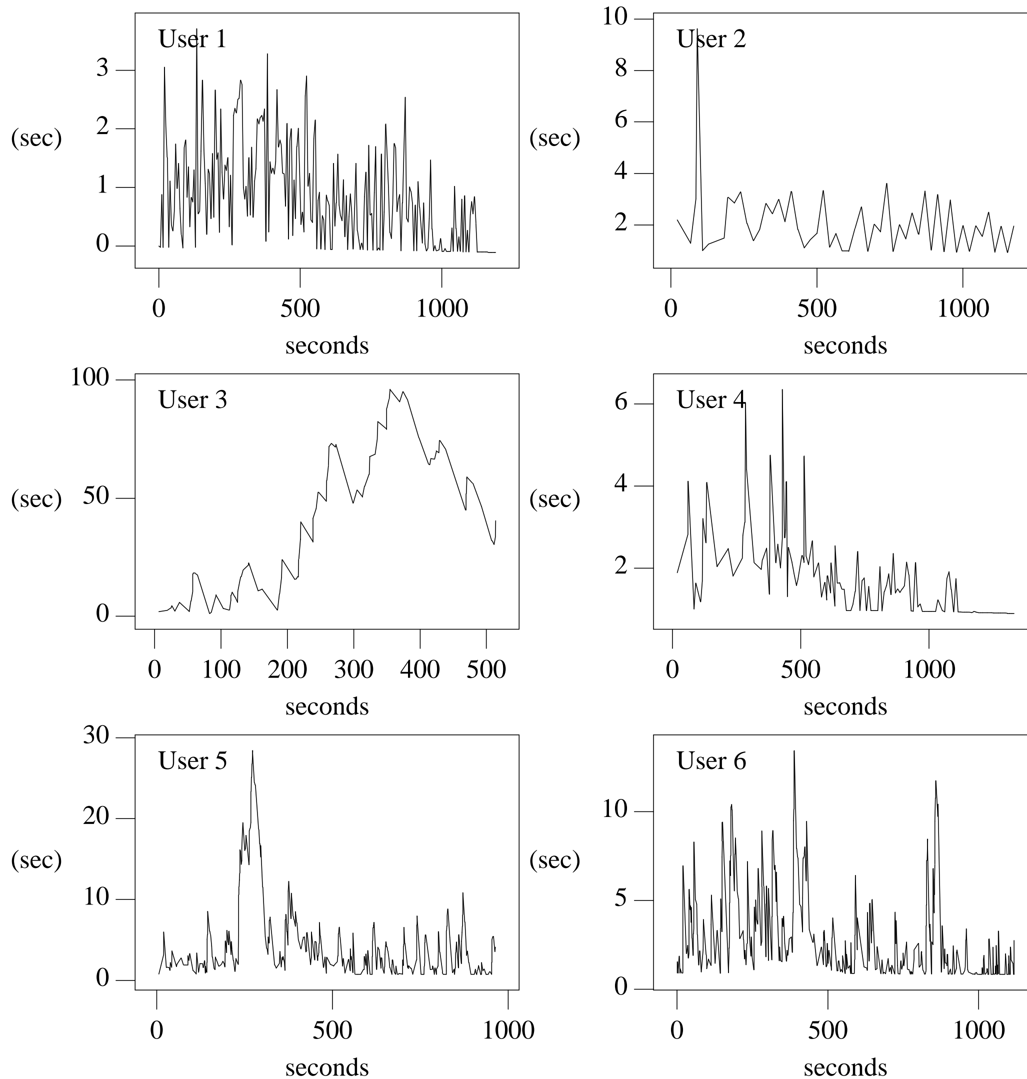

Figure 8: Delay vs time elapsed, test 2

6. Conclusion

We have defined a new queueing algorithm called Hierarchical Fair Queueing. HFQ extends WFQ to the case where the bandwidth of a link should be divided between classes of users, and between subclasses within each class. Unlike CBQ or other Link Sharing schemes (which also address this problem) HFQ is provably fair: there are bounds on the difference between arrival times in HFQ and those in the idealised fair discipline HPS. Indeed HFQ operates by keeping a simulation of HPS, and approximating that.

We have implemented HFQ in an IP gateway for the Plan 9 operating system, and we gave measurements showing how the throughput approximates HPS, and how low delay is achieved for well-behaved users, even when others are trying to swamp the system.

HFQ has the disadvantage of not being able to decouple delay bounds from bandwidth allocations, as is possible in CBQ. HFQ does give delay bounds, but they are fixed for a given set of bandwidth allocations. It may be possible to generalise HFQ to allow separate control over delays, as is done with delay-EDD [20]. It is not know at present what effect this would have on the effectiveness with which HFQ approximates HPS. This is an important area for future research.

Appendix

A.1. Data Structures

The central data structure is the *class tree*. The leaves of this tree represent individual users. The internal nodes represent groups of users. Each node has a number of *shares* associated with it. The shape of the tree, together with the values of all the shares, determines the *policy* that HPS is to implement, as described in section 3.

The other data structures to consider are the *queues*. There are two kinds of these: a single *priority queue*, and a *fifo queue* for each user. The priority queue is used to hold requests which have already started service in the HPS simulation. The other requests are held in the queue corresponding to the user owning the request.

Some mechanism by which incoming requests are classified and associated with particular leaf nodes (ie users) is assumed.

A.2. Notation

Given α, a node of the class tree, the following are defined:

`α.parent`	the node's parent
`α.children`	the set of all the node's children
`isleaf(α)`	true is node is a leaf node, false otherwise
`α.queue`	the per user queue associated with the node (leaf nodes only)
`α.creq`	cumulative requests received by this node (leaf nodes only)
`α.csrv`	cumulative service received by this node (leaf nodes only)
`α.creq1`	cumulative requests passed to priority queue (leaf nodes only)
`α.shares`	the number of shares of the node
`α.sashares`	sum of active shares of the node's children (if any)

A number of global variables are also defined. These are:

`root`	the root of the class tree
`pri_q`	the global priority queue
`t`	current time
`tlast`	time of last event in HPS simulation
`rmax`	the resource capacity
`rendez`	rendezvous used for communication between `new_request` and `next_service`

A.3. Pseudocode Walkthrough

In this presentation of the HFQ algorithm, there are two entry points: `new_request` and `next_service`. `new_request` is called whenever a new request arrives at the system, to enqueue the request on the appropriate queue. `next_service` is called whenever a request finishes service (and at startup time) to choose the next request to be serviced. These two functions are (conceptually) located in different threads; if the system is idle, `next_service` blocks until `new_request` has been called. Note that this treatment does not deal with the problem of locking between the two threads (which must exist to prevent both functions trying to update the tree simultaneously).

`new_request` takes as arguments the user that the request belongs to, α, and the request itself, `req`. The pseudocode for the function is as follows:

```
/*
 *      Entry point: a new request "req" has arrived at node "α"
 */
new_request(α, req)
{
        update(root, t);
        enqueue(α.queue, req);
        wasidle = (root.sashares == 0);
        if (α.creq == α.csrv)
                active(α);
        α.creq += len(req);
        if (wasidle)
                wakeup(rendez);
}
```

The first action taken is to call a function called `update`, described in full later, which advances the state of the HPS simulation to the current time, `t`. Since we are processing an arrival occurring at time `t`, there can be no arrivals prior to `t` left unaccounted for, and thus we have sufficient information to determine the state of the HPS simulation at time `t`. Next the request is enqueued one the queue belonging to user α. Then a flag `wasidle` is set if the system was idle when this request arrived. This flag is a local variable, used later in the function. Then, if the user α was idle prior to this arrival, we call `active` to update the state of the class tree to reflect the change in α's activity state. Then we update α's cumulative request value, `α.creq`. Last of all, if the system was idle before this arrival, we wake up the service thread, which may be blocked in `next_service`.

`next_service` takes no arguments. Its purpose is to determine the next request to be serviced at time `t`, which it returns. Here is the pseudocode:

```
/*
 *      Entry point: called whenever the resource becomes idle, to schedule
 *      the next request to be serviced
 */
next_service()
{
        while (idle)
                sleep(rendez);          /* wait for call to new_request() */

        update(root, t);
        req = dequeue(pri_q);   /* pri_q cannot be empty */
        return req;
}
```

We first test to see whether there are any requests enqueued in the system. If not, then we must wait until the next call to `new_request` is made before we can proceed, at which point the system is no longer idle. Having satisfied this condition, we now call `update` to advance the HPS simulation to the present time. We then merely dequeue the first request on the priority queue, which we return. We are assured that at this stage there must be a request on the queue, as discussed in section 4.

The remainder of the algorithm consists of helper functions used to update the HPS simulation. The first of these is `active`. This function is called when a user (α) makes the transition from idle to active, to update the activity state of the class tree. Since this activity state is represented using the `sashares` property on each internal node, it is only necessary to update this property on (at most) the ancestors of α. The pseudocode is as follows:

```
/*
 *      Given a node "α", adjust its ancestors' sum-of-active-shares (sashares)
 *      attribute to reflect α's transition from idle to active
 */
active(α)
{
        if (α == root)
                return;
        if (α.parent.sashares == 0)
                active(α.parent);
        α.parent.sashares += α.shares;
}
```

`active` is a recursive function. It is initially called with a leaf node as argument, and recurses up through the ancestors of that node until either an active node is found (represented by a non-zero `sashares` property) or the root of the tree is reached. For idle ancestor encountered, the shares value of that node is added to its parent's `sashares` attribute. Note that the recursion takes place before updating the attribute, so that the test is not invalidated.

`inactive` is similar to `active`, except that it represents the transition from active to idle. The pseudocode is as follows:

```
/*
 *      Given a node "α", adjust its ancestors' sum-of-active-shares (sashares)
 *      attribute to reflect α's transition from active to idle
 */
inactive(α)
{
        if (α == root)
                return;
        α.parent.sashares -= α.shares;
        if (α.parent.sashares == 0)
                inactive(α.parent);
}
```

Here the parent's `sashares` attribute is decremented before testing; if this makes it zero, then the parent is now idle, and recursion occurs.

Next we describe a function used by `update` to predict the time of the next change in activity status, assuming no further arrivals. `predict` is called with a node α and a rate of service `r`, and recursively searches the node's descendents for the next node to become idle (in HPS), returning a tuple `(β, tnext)`, consisting of the predicted node β, and the amount of time `tnext` until it will become idle. If no such change is possible, because all the leaf node are idle, then `(nil, 0)` is returned. `predict` is passed `root` as the node in order to search the whole tree. The node argument is provided for the purpose of recursion. The pseudocode is:

```
/*
 *      Given a node "α", and a rate "r", search α's descendants for the next
 *      change in active status, assuming no intervening arrivals.
 *
 *      returns: (β, tnext), where "β" is the leaf node of the next predicted
 *      transition, and "tnext" is the amount of time until this transition
 *      is predicted to occur (assuming no intervening arrivals).
 */
predict(α, r)
{
        if (isleaf(α)) {
                if (α.creq > α.csrv)
                        return (α, (α.creq-α.csrv)/r);
                else
                        return (nil, 0);
        }
        if (α.sashares == 0)            /* idle -- no departures possible */
                return (nil, 0);

        /* search children for next transition */
        r /= α.sashares;
        (β, tnext) = (nil, 0);
        for (γ in α.children) {
                (δ, ttry) = predict(γ, r*γ.shares);
                if (β == nil || δ != nil && ttry < tnext)
                        (β, tnext) = (δ, ttry);
        }
        return (β, tnext);
}
```

The function falls into three cases. First, if it is called with a leaf node, then it merely has to determine whether it is idle or not, returning `(nil, 0)` if it is idle, and the node along with the amount of service time required, at rate `r`, to complete its service, ie `(α.creq-α.csrv)/r`. The second case is when we have an internal node α with `α.sashares == 0`. This indicates that all of α's descendants are idle, so we waste no further time searching beneath this node, and just return `(nil, 0)`. In the third case, we are given an active internal node. Here `predict` recursively calls itself on each of its children, and finds the return value `(β, tnext)` with the smallest `tnext`, but `β != nil`. This is the first node under α to become idle, and hence the required return value. Note that if α is being serviced at rate `r`, that the child β is serviced at rate `r*β.shares/α.sashares`, according to HPS policy. This value is calculated and passed as the rate in the recursive call to `predict`.

We now come to `update`. The task here is to advance the current simulated time (`tlast`, the time of the last simulated event) to the current time. This advance is broken down into steps. `predict` is called first to find the next change in activity, if any. As it is assumed that no arrivals occur between `tlast` and `time`, so the only changes can be departures. For each such change in activity found, `tlast` is updated to that time using a further helper function `update1`, described below. This is repeated until no further changes in activity are found, at which point `tlast` is updated one last time, to the current time. Here is the pseudocode:

```
/*
 *      Given "root", the root of the class hierarchy, and "time", the current
 *      time, update the state of the HPS simulation such that "time" is now
 *      the current simulated time ("tlast").  "rmax" is the line rate.
 */
update(root, time)
{
        for (;;) {
                (β, tdep) = predict(root, rmax);
                if (β == nil || tlast+tdep > time)
                        break;
                update1(root, rmax, tdep);
                tlast += tdep;
        }
        if (tlast < time) {
                update1(root, rmax, time-tlast);
                tlast = time;
        }
}
```

Note that if the time returned by `predict` is beyond the current time, then its value is not accurate, since it could be affected by further arrivals. But in this case we don't care, since it signals the fact that there are no further changes in activity to simulate; the actual value does not matter.

The actual dirty work is performed by `update1`. Here we are concerned with the class tree over an interval of time, during which no changes in activity occur (except at the endpoints) and thus every node is serviced at a constant rate. `update1` takes three arguments: a node α, a rate of service `r`, and an amount of time `deltat`. It is a recursive function, and is structured similarly to predict:

```
/*
 *      Given a node "α", a rate "r", and an amount of time "deltat",
 *      simulate the effect of "deltat" seconds of HPS service at rate "r",
 *      on "α" and its descendants
 */
update1(α, r, deltat)
{
        if (isleaf(α)) {
                if (α.creq > α.csrv) {
                        newsrv = α.csrv + r*deltat;
                        while (newsrv > α.creq1) {
                                req = dequeue(α.queue); /* cannot be empty */
                                enqueuepri(pri_q, req, tlast + (α.creq1-α.csrv)/r);
                                α.creq1 += len(req);
                        }
                        if (newsrv >= α.creq) {
                                newsrv = α.creq;
                                inactive(α);
                        }
                        α.csrv = newsrv;
                }
        }
        else if (α.sashares != 0) {
                r /= α.sashares;
                for (β in β.children)
                        update1(β, r*β.shares, deltat);
        }
}
```

The first case is when α is a leaf node. This is the most complex case, and its description will be left until last. Supposing now that α is an internal node. If α is idle (`α.sashares == 0`) then there is nothing

to do: all of α's descendants are idle, and their state does not change over the interval. If, however, α is active, then we must call `update1` recursively on each of its children. Here we use the same device as in `predict`, whereby the appropriate value for `r` is pass in the recursive call.

We now consider the case where α is a leaf. If this leaf is idle `(α.creq == α.csrv)` then there is nothing to do. Otherwise, we have to update `α.csrv` to its new value, `α.csrv+r*deltat`, and handle any departures caused by this change. Here we use `α.creq1` to check for individual requests that need to be dequeued. If we find any, they are moved to the priority queue, stamped with their HPS starting time, which is now known. Then `α.creq1` is updated, so that it always represents the cumulative request time up until the first request on α's queue. We iterate until all such requests have been dealt with. There may be more than one; although we are assured by `update`'s use of `predict` that no node becomes idle in HPS before the end of the interval, multiple individual packets are allowed to depart. Having dealt with these packets, we then check whether α is becoming idle at the interval. If so, we call `inactive` to register the fact. The tests used err on the side of caution, lest a numeric error cause `α.csrv` to become larger than `α.creq`.

References

1. Brownie, J. Analysis and simulation of share systems. *Honours Thesis* (1983).
2. Demers, A., Keshav, S., and Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. *Internetworking: Research and Experience 1* (1990), 3-26. (An earlier version may be found in *Computer Communications Review 19*, 4 (Sept 1989), 1-12.)
3. Figueira, N.R. and Pasquale, J. An Upper Bound on Delay for the VirtualClock Service Discipline. *IEEE/ACM Transactions on Networking 3*, 4 (August 1995), 399-408.
4. Floyd, S. and Van Jacobsen Link-Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking 3*, 4 (August 1995), 365-386.
5. Greenberg, A.G. and Madras, N. How Fair is Fair Queueing?. *Journal of the Association for Computing Machinery 39*, 3 (July 1992), 568-598.
6. Hogan, D. Analysis and Simulation of SHARE. In *Fifteenth Australian Computer Science Conference*, Vol. 14, No. 1, Pt. 1, Australian Computer Science Communications, Jan 1992, pp. 351-363.
7. Hume, A. A Share scheduler for Unix. *AUUG News.* (1979).
8. Kay, J. and Lauder, P. A Fair Share Scheduler. *Communications of the ACM 31* (January 1988).
9. Kay, J., Lauder, P., Maltby, C., and Tollasepp, S. The SHARE Charging and Scheduling System. *Technical Report 174* (May 1982).
10. Keshav, S. On the Efficient Implementation of Fair Queueing. *Internetworking: Research and Experience 2* (1991), 157-173.
11. Kleinrock, L. Time-shared Systems: A Theoretical Treatment. *Journal of the Association for Computing Machinery 14*, 2 (April 1967), 242-261.
12. Larmouth, J. Scheduling for a share of the machine. *Software Practice and Experience 5* (Jan. 1975), 29-49.
13. Larmouth, J. Scheduling for immediate turnaround. *Software Practice and Experience 8* (Sept.-Oct. 1978), 559-578.
14. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. *Plan 9 - The Documents*. Vol. 2. Harcourt Brace, 1995. URL http://plan9.att.com/plan9/vol2.html.
15. Presotto, D. Multiprocessor Streams for Plan 9. In *Summer 1990 UKUUG Conference*, July 1990, pp. 11-19.

16. Simpson, W. The Point-to-Point Protocol. rfc1548, Dec 1993.

17. Valencia, A. An Overview of the VSTa Microkernel. URL http://www.cen.uiuc.edu/~jeske/VSTa/vsta_intro.ps.

18. Wakeman, I., Ghosh, A., Crowcroft, J., Van Jacobsen, and Floyd, S. Implementing Real Time Packet Forwarding Policies using Streams. *Usenix 1995 Tech. Conf.* (Jan. 1995), 71-82. URL ftp://cs.ucl.ac.uk/darpa/usenix-cbq.ps.

19. Waldspurger, C.A. and Weihl, W.E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *First Symposium on Operating Systems Design and Implementation (OSDI)*, Usenix Association, , pp. 1-11.

20. Zhang, H. and Keshav, S. Comparison of Rate-Based Service Disciplines. In *Proc. ACM SIGCOMM*, 1991, pp. 113-121.

21. Zhang, L. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks. *Computer Communication Review 20*, 4 (Sep 1990), 19-29. (Special issue: Proceedings of SIGCOMM '90, Communications Architectures & Protocols).