Hierarchical Fair Queueing

By D. Hogan

Technical Report 513
February 1997

Basser Department of Computer Science
University of Sydney
Hierarchical Fair Queueing

by

David Hogan

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Basser Department of Computer Science

University of Sydney

February 1997
ABSTRACT

Hierarchical Fair Queueing (HFQ) is a new queueing algorithm for networks, which 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 et al., to deal with a hierarchy of classes. HFQ also provides bounded delays for real time traffic.

We provide extensive analysis of the throughput and delay discrepancy bounds that exist on the approximation of HFQ to an idealised, fluid flow discipline called Hierarchical Processor Sharing. We implemented HFQ as part of a real IP gateway, and performed a number of tests which demonstrate its fairness. Results of these tests are presented.
ACKNOWLEDGEMENTS

I would like to thank my supervisor Bob Kummerfeld, and my co-supervisor Alan Fekete for their patience, support and encouragement. This thesis would never have been finished but for their motivation and direction.

I would also like to thank Danny Yee and Geoff Bailey for proof-reading drafts of this thesis, and providing many suggestions for the improvement of the style. Michael Usher and Carole Cusack provided much helpful advice and encouragement when it was most needed.

Many thanks to all my friends and my family, who have continued to give me their love and have helped me to make it through a difficult time in my life.
# Contents

1. Introduction 1

2. Background Literature 4
   2.1. Early Work on Queueing Disciplines 4
   2.2. Fair Queueing and its Implementation 8
   2.3. Variations on Fair Queueing 10
   2.4. Theory of Fairness 19
   2.5. Hierarchical Sharing 23
   2.6. Alternative Approaches to Sharing 25
   2.7. Summary 28

3. Defining Fairness 29
   3.1. The Non-Hierarchical Case 29
   3.1.1. Long-Term Sharing 30
   3.1.2. Problems with Long-Term Sharing 31
   3.1.3. Max-Min Fairness Criterion 32
   3.1.4. Processor Sharing 34
   3.2. Hierarchical Processor Sharing 35

4. The HFQ Algorithm 39
   4.1. Overview 39
   4.2. Data Structures 41
   4.3. Notation 42
   4.4. Pseudocode Walkthrough 42
   4.5. Complexity of HFQ 49

5. Analysis of Hierarchical Fair Queueing 51
   5.1. Overview 51
   5.2. Service Discipline Executions 52
   5.3. Comparing Two Service Disciplines 56
   5.4. Rate of Service and Work Conservation 57
1. Introduction

Modern computer networks integrate many different types of physical communication media which provide a wide variety of transmission speeds. Different subnetworks are connected together using gateways. Two users in such a network can communicate by sending data over a number of links, through the gateways connecting them.

Network applications are similarly diverse—delay sensitive applications such as network audio and video on demand require strong guarantees on throughput and end-to-end delays, while best-effort data applications adapt easily to prevailing conditions. In the past, voice and data have been kept separate, but the trend is towards integration of these different types of applications into a unified network.

Telecommunications networks have traditionally used fixed bandwidth connections and Time Division Multiplexing. Such techniques provide service guarantees simply and efficiently, but are wasteful of bandwidth. Data networks allow users to send and receive data in packets without such wastage of bandwidth, but without any guarantees either. It is possible on a data network for users to send data at a higher rate than the link that it must travel over. This possibility is handled by maintaining queues within the gateways. When a link’s capacity is temporarily exceeded, packets are held in the appropriate queue. When the situation becomes more permanent it is referred to as congestion, which is characterized by increased queueing delays and eventual depletion of buffer space for storing the queues. Congestion control methods are used to stop users from overtransmitting.

Each queue is managed by a service discipline, which is an algorithm which controls the order in which the packets are taken from the queue and serviced. The traditional First Come First Served (FCFS) service 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 real-time applications, and some for both.

Recent work by Floyd and Jacobsen [11] and colleagues [45] 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 organizations 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 recognize 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, a natural definition was proposed (in the context of CPU scheduling) by Kleinrock [25]. Kleinrock's Processor Sharing model (PS) is an idealized, 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 generalizes to include a concept of weights which control the division of the instantaneous bandwidth. In this thesis 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 until that has finished before choosing another. In this light, PS is not a realistic queueing algorithm, because of its assumption that we can service all active users simultaneously. We can not implement PS directly; the best we can do is to approximate it in some way.

For a single level of classes, the Fair Queueing (FQ) service discipline [9] has been proposed. This technique amounts to running a simulation of PS, and choosing the next packet based on the finish (or start) of transmission within PS. FQ
approximates PS closely, as proved by Greenberg and Madras [15]. It is trivial to generalize FQ to include shares, as shown by Demers et al. [9], and Parekh and Gallager [30]. This generalization is commonly called Weighted Fair Queueing (WFQ).

The limitation of FQ and WFQ is that they are not hierarchical. That is, they are designed only to share bandwidth between a set of users (or classes of traffic), and provide no facilities for splitting the total bandwidth between a bunch of organizations, then dividing each organization’s bandwidth between users in that organization. This is not equivalent to dividing the total bandwidth amongst all the users; the hierarchical extension of WFQ does not degenerate into WFQ.

The main contribution of this thesis is an algorithm called Hierarchical Fair Queueing (HFQ). This uses ideas inspired by FQ to solve the problem of sharing a link fairly between users arranged in a hierarchy of classes. We prove that HFQ approximates HPS to within a (theoretically guaranteed) bounded error. The proof of this approximation result is based on a careful theory of service disciplines for hierarchical classes, itself another important contribution of the thesis. The HFQ algorithm has been implemented and tested in a real TCP/IP system. The results of the tests form the third significant contribution of this thesis.
2. Background Literature

In this chapter, we examine the existing literature on the problem of Fair Queueing in networks. First we examine early work on queueing disciplines which lead to the development of Fair Queueing. Then we examine the original Fair Queueing algorithm of Demers et al., along with papers which explore the associated implementation issues. Next, we describe some variations on Fair Queueing which provide a variety of trade-offs between efficiency and throughput/delay bounds. Following this we examine the theoretical background for Fair Queueing. Then we cover papers on hierarchical sharing schemes. We look briefly at alternative approaches to bandwidth sharing. We conclude with a summary of the state of the field with regard to Hierarchical Fair Queueing.

Although the main focus here is on bandwidth sharing, we cannot ignore the influence which ideas from CPU sharing have had. Certain CPU scheduling ideas have anticipated the development from Processor Sharing (PS) to Generalized Processor Sharing (GPS) and thence to Hierarchical Processor Sharing (HPS) which has occurred in the area of network queueing algorithms. We therefore include such work in our survey.

2.1. Early Work on Queueing Disciplines

Most of the early work relevant to Fair Queueing occurred in the domain of CPU scheduling. There are sufficient differences between network queueing and CPU scheduling that, in general, one can not take an algorithm intended for one such purpose and use it without modification for the other. Nevertheless, the two areas are sufficiently closely related that ideas from one area may be relevant to the other. And indeed we find that ideas originally developed for CPU scheduling have influenced the development of techniques for network queueing.

The Round Robin (RR) scheduler, and the Processor Sharing (PS) discipline derived from it, have been of particular importance in the development of Fair Queueing algorithms. PS was introduced by Kleinrock [25], who defined it as the limiting case of Round Robin when the service quantum is allowed to approach zero. This paper also anticipated the development of GPS, where users are assigned priorities ("shares") such that service is received in proportion to the priority of the task.
Kleinrock's model differs from that used in network queueing in that all jobs are serviced simultaneously. Network queueing uses 'head of line' Processor Sharing [15], where only the jobs at the head of the per-user queues are serviced. Kleinrock's paper includes detailed analyses of PS (with and without priorities) for M/M/1 queues and a comparison with the FCFS (First Come First Served) discipline: it demonstrates that the scheme provides two properties which are desirable for timesharing systems—rapid service for short jobs and virtual appearance of a fractional capacity processor on a permanent basis.

Processor Sharing forms the basis of the FQ algorithm of Demers et al. [9], and is also central to a number of theoretical works [15], [30], [29]. A hierarchical version of PS forms the basis of the algorithm presented in this thesis (see Chapter 3).

A number of alternative sharing mechanisms have been devised for CPU scheduling, some of which use a hierarchical model. Cambridge Share [27], [26] is one such system. Cambridge Share was developed for batch processing, to control the "turnaround" time of jobs so as to provide fairness. Each user is assigned a number of shares and a decaying average of usage is kept for each user. New jobs are scheduled on the basis of shares and usage. The second paper [26] includes a version of the algorithm with a hierarchical scheme.

The basic ideas of Cambridge Share were an inspiration for Basser Share [21], [18], [4], [22], [17], [16], a scheduler for interactive processing. Basser Share is a replacement for the scheduler used by the Unix Timesharing System [36], [1]. It includes per-user shares and decayed usage, as in Cambridge Share, but the mechanism by which these are used to control scheduling is quite different. Each process has an associated priority, which in the standard Unix scheduler is effectively a decayed average of work done by that process, incremented once per scheduling tick. In Basser Share, the constant increment is replaced by a formula based on the per-user usage and shares. Basser Share is also hierarchical; the usage and shares of nodes higher up in the tree are used to adjust the priority calculation. This system has been simulated and analysed in a number of papers [4], [17], [16]. In general its behaviour is complex and difficult to predict, and it does not approximate Processor Sharing. The reason for this is in part due to its origins in a student environment, where it was held to be desirable to "punish" users who used the machine "too
much” by reducing their effective share of the machine—thus effectively pushing the problem of regulating terminal access into the scheduler! Bassel Share is based on a paradigm that we refer to as “Long-Term Sharing”, which we will explore in Chapter 3.

Another approach is Lottery Scheduling [46]. This method attempts to approximate PS probabilistically. Every time that a scheduling decision is made, a random number generator is invoked and the value generated is used to choose which process to run. The method used gives a probability of being chosen for service which is proportional to shares. The system also includes some interesting ideas for transferring shares during transactions.

The method used by Lottery Scheduling easily generalizes to hierarchical policies. Such a scheme has been implemented as the VSTa scheduler [42], [43].

The “Monte Carlo” approach used by Lottery Scheduling and the VSTa scheduler provide at best a probabilistic approximation to the ideal service discipline, be it PS or HPS. Hence they are inferior to FQ, HFQ and related algorithms, which provide strongly bounded, deterministic approximations to the ideal disciplines. Providing bounded delays is of particular importance in networking applications, but no such guarantee can be obtained with a probabilistic method.

Some early attempts to deal with network unfairness and congestion used end to end (transport layer) transmission control. An example of some significance is the DECbit congestion avoidance algorithm [20], [19], [33], [5], [34]. This method requires modifying the gateways to set a special bit in packets (called the congestion avoidance bit) whenever it is determined that a source is overtransmitting. The source monitors these bits (duplicated in the destination’s replies) and adjusts its transmission behaviour accordingly, using an additive increase/multiplicative decrease algorithm. This provides a feedback loop to find the optimal operating point of the network. The gateway throttles its users when a certain point is reached, before congestion has actually started—hence providing congestion avoidance rather than congestion control. The choice of which users to throttle is done in such a way that some sort of fairness results.

Since the advent of DECbit (and systems similar to it, such as Qbit [37]) it has been realized that fairness cannot be achieved by transmission control alone. Rather,
a combination of fair service discipline and good transmission control is required—the service discipline must enforce fairness, otherwise a rogue service can "cheat". Transmission control is still important for obtaining good utilization of the network resource. DECbit is not dead—it was very influential over the development of the Van Jacobson TCP congestion avoidance algorithm [44].

The first tentative steps towards Fair Queueing were taken by Nagle [28], in a paper on congestion in network gateways. In this paper, the unfairness of FCFS is recognized, and a solution is proposed. Nagle uses a game-theoretic view of datagram networks to explain why changing the service discipline is necessary. The unmodified, FCFS-based network is seen to be an unstable multiplayer game. The optimal strategy for each user is suboptimal for all users—a user must increase packet transmission to gain a greater share of the bandwidth, but this leads to congestion and increased end-to-end delays for everyone. Such a system naturally tends towards a suboptimal state. Nagle proposes to change the rules of the game, such that well-behaved users are the ones that get optimal throughput. Nagle's solution is to keep per-user queues, and service them in round robin fashion. Under this regime, users who overtransmit will only succeed in driving their queue lengths up, and increasing their end-to-end delays. The other users are insulated from the deleterious effects of such users' ill-behaviour by the fairness properties of the Round Robin algorithm. This paper also raises the problem of fair buffer allocation, and suggests that it be solved by discarding packets from the longest per-user queue when there is a shortage.

It is now realized that Round Robin is unfair in the presence of variable packet sizes. Since packets have to be transmitted in full, one cannot use a fixed quantum of service as is the case with standard Round Robin in a CPU scheduling environment. Instead, Nagle's solution services a packet from each active connection per "round". This leads to unfairness if packet sizes are not fixed, since a user transmitting larger packets will automatically receive a greater portion of each "round". FQ was developed to overcome this problem.
2.2. Fair Queueing and its Implementation

Fair Queueing (FQ) [9] was developed to overcome the unfairness of Nagle’s proposal to apply Round Robin on a packet by packet basis. Demers et al. reasoned that the unfairness could be removed (in theory) by breaking each packet into its constituent bits, and servicing them in Round Robin order, using a service quantum of 1 bit. The resulting "Bitwise Round Robin" would be fair in spite of varying packet sizes, but impractical to implement, since packets can not be subdivided in such a way in a practical system. To finesse the problem of this impracticability, their Fair Queueing algorithm (allegedly) simulated Bitwise Round Robin, using the (virtual) finishing times from the simulation to order the actual transmission of packets by the algorithm. In this way, a practical approximation to BRR could be obtained, by servicing the packets in roughly the same order. Demers et al. actually took the abstraction a step further—their "simulation of BRR" is actually a simulation of PS, as noted by Greenberg and Madras [15]. Thus it is usual nowadays to regard FQ as being based on a simulation of PS, and indeed an approximation to PS.

An important feature of FQ is the way in which the simulation of PS is carried out. Demers et al. [9] introduced the concept of "virtual time", a continuous, monotonic function of real time which intuitively provides a count of the number of "rounds" completed. By expressing the starting and finishing times of packets in the simulation in virtual time, it is possible to calculate them using a pair of simple recurrence relations. Furthermore it turns out that the virtual finishing time of a packet may be determined as soon as the packet arrives, enabling it to be placed immediately in a priority queue. The main complexity of the algorithm consists in keeping track of virtual time.

As originally defined, FQ assumed an "equal rights" policy for all of their users, in that all users are to be serviced at equal rates. Just as PS allowed for the unequal rights in the form of priorities or shares, it is trivial to extend FQ in this way. The extension is described briefly by Demers et al. [9], and provides bandwidth guarantees with control over the individual allocations by changing the shares. The extended version of FQ is commonly referred to as Weighted Fair Queueing (WFQ), although the name Packet-by-packet Generalized Processor Sharing (PGPS) is also used [30].
Implementation of FQ has been discussed in a few papers. A crucial component in any FQ implementation is the PS simulation, used to calculate virtual time. Greenberg and Madras [15] describe how to simulate Processor Sharing in $O(\log N)$ time. Simulating PS involves calculating the finishing times of all requests, which helps us to calculate the virtual time function. The key to their technique is in the use of the recurrence relations for the virtual starting and finishing times of requests in PS, which were first described by Demers et al. [9]. These recurrence relations make it possible to calculate the virtual finishing time of a request when it first arrives in the system, without needing to know any of the subsequent arrivals. By keeping all of the (at most $N$) active requests in a priority queue, ordered by virtual finishing time, we can find out the time of the next service completion (assuming no intervening arrivals) in $O(1)$ time. The cost of maintaining the PS simulation is thus the cost of maintaining the priority queue, which is $O(\log N)$ for any decent priority queue implementation.

Srinivasan Keshav [24] has further examined the issue of implementing the FQ algorithm efficiently. FQ is conceptually divided into three components: bid number computation, round-number computation (i.e., virtual time) and packet buffering. The main issues for bid number computation are determining the conversation ID of each packet as it arrives, and using it to look up the state for that conversation. These issues are fairly well understood, and thus are dealt with only briefly.

Round number computation is examined next. An algorithm is described which uses a heap to keep track of finishing times in the associated Processor Sharing discipline. This is essentially the same as that described by Greenberg and Madras [15], as mentioned above. Use of the heap yields $O(\log N)$ time per operation.

The remainder of the paper deals with packet buffering issues. It is assumed here that when the router runs out of buffers, the packet with the greatest bid number (priority) is dropped. The requirements for the packet buffering algorithm are formulated as an abstract data structure called a bounded heap. This has an operation to insert a new packet with a given priority, an operation to remove the packet with the lowest priority, and a maximum number of packets. Four implementations for the data structure are considered in the paper: an ordered list, a binary tree, a double heap, and a per-conversation ordered list. After comparing the theoretical worst case
costs, a detailed performance evaluation using simulated workloads is described. This shows that the per-conversation ordered list provides the best performance, but also has the highest implementation cost. On the other hand, if the network is designed such that it never becomes overloaded, then the single ordered list performs the best (and has the lowest implementation cost). The double heap method provides good performance with a low implementation cost, and the tree method performs badly.

The use of virtual time in PS simulations is of crucial importance to deriving efficient FQ implementations. Unfortunately, such techniques do not generalize to Hierarchical Processor Sharing. Our HFQ algorithm, therefore, is not as efficient as the FQ implementations described above.

2.3. Variations on Fair Queueing

A variety of alternatives to FQ have been proposed. Some of these are modifications to the existing FQ algorithm, whereas others are completely new replacements. One motivation for modifying or replacing the algorithm is to improve the efficiency of the algorithm. Some of the variants achieve a significant improvement in this area. Another reason is to improve the bounds on the approximation to PS. Although FQ without shares achieves the best possible bound, when shares are introduced this is no longer the case. Another measure of fairness, called the Worst-Case Fairness Index (WFI), has been introduced by Bennett and Zhang [3]. WFQ does not yield a very good WFI; other algorithms have been developed which improve upon it. The WFI is closely related to the delay guarantees provided by a discipline; a better (smaller) WFI is desirable.

There are a number of ways of evaluating alternatives to FQ. In general, each algorithm provides a compromise between the various competing factors: one algorithm may provide a more efficient implementation at the expense of the throughput bounds, another may sacrifice delay bounds, etc. Ultimately the choice of algorithm depends on the application. In this section we survey alternatives to the original FQ algorithm.

A paper by Davin and Heybey [8] describes some simulation experiments performed based on three different router algorithms, using the Internet Protocol Suite
as the basis for the simulations. The algorithms simulated were First Come First Served (FCFS) and two variations on the FQ algorithm of Demers et al. [9], referred to as FQNP and FQFQ.

Both FQ variations used in the simulation contain a modification designed to make FQ more efficient to implement. The "virtual time" function used by FQ is approximated by the virtual finishing time of the current (or most recent) packet under service. (This variation has subsequently been analysed in detail under the name "Self Clocked Fair Queueing" (SCFQ) [12]). The two variations differ in their buffer management strategy, which specifies their behaviour when the router runs out of buffers. In FQNP (Fair Queueing with No Punishment), the user with the most packets has the packet from the end of its queue dropped, and future packets from that user are treated as if the dropped packet never arrived (as opposed to "punishing" the user for the excessive number of packets, by allowing the dropped packet to influence later packets' priorities). FQFQ, or Fair Queuing with Fixed Quota, uses an alternative method, where every user is preallocated a fixed, equal number of buffers. Packets are discarded when a user tries to exceed its quota. This strategy is designed to simplify the task of management, making it easier to implement and more efficient.

The simulations all contained 4 user classes. Each user class had multiple source-sink pairs, all of which used a model based on TCP. Both well-behaved and ill-behaved TCP sources were used: well-behaved sources used the slow-start congestion control algorithm, and ill-behaved sources did not. A variety of experimental scenarios were used, combining different allocation policies (equal shares versus unequal shares), different demand characterizations and different delay characterizations. The results of the simulations show that the two FQ variants enforce the resource allocation policies better than FCFS.

Golestani [12] proposes an alternative to the Fair Queueing algorithm of Demers et al., called Self-Clocked Fair Queueing (SCFQ). This is motivated by the observation that FQ is impractical for high-speed applications (such as ATM). The problem with FQ is that it requires the simulation of a hypothetical fluid-flow system, in order to calculate the "virtual time" function which FQ uses. In a worst case scenario, this simulation may generate an event for every user in the time that it takes
to service a single packet. For high-speed networks such as ATM, this order of complexity is unacceptable.

SCFQ is based on the same calculations as FQ except that instead of using a fluid-flow reference system to generate virtual time, a function analogous to virtual time is generated from SCFQ itself (in fact it is the same idea as that used by Davin and Heybey [8]). The resulting algorithm is much simpler to implement, and suitable for high-speed applications. The trade-off is that it is slightly less fair than FQ.

The fairness of the system is analysed by comparing the normalized service received by two users over an interval for which both users are backlogged. A bound is derived on the difference between the two, which is shown to be at most twice the bound obtainable by any packet-based system. (Normalized here means divided by shares). Thus it is concluded that SCFQ is "near optimal" in its fairness.

In the concluding remarks, the existence of a delay bound for Leaky Bucket constrained arrivals is mentioned, but deferred to a later publication. A bound on the disparity in service between SCFQ and GPS is also claimed.

The efficiency problems shown to exist in FQ are also present in HFQ. However, it is not at all clear how the technique used by SCFQ to circumvent this problem could be applied to the hierarchical case.

The analysis of the fairness of SCFQ in the original paper of Golestani [12] is augmented by a second paper [13] with a detailed analysis of SCFQ's delay characteristics. The results obtained in the first paper are used as a springboard for defining a whole class of Fair Queueing algorithms. This class includes SCFQ as well as GPS and WFQ. The definition constitutes an alternative definition of fairness to that introduced by Greenberg and Madras [15]; namely, bounded throughput discrepancies when compared with GPS. The new definition is based around the concept of virtual time, with the bounds being on the discrepancy between the (normalized) service provided to users backlogged over some interval of time, and the change in the virtual time over that interval. Here virtual time is defined as an arbitrary function of time. SCFQ and GPS are shown to satisfy this definition (indeed, the case for SCFQ follows from the proofs in the first paper [12]).

The delay analysis is based on the assumption that each user's arrivals conform to a Leaky Bucket characterization. Given that the Leaky Bucket parameters satisfy
appropriate resource allocation constraints, delay bounds are proved for the class of "Fair" algorithms, in both the single node case, and the multiple node case (for a network of fair queueing algorithms). These results are applied to the special case where the algorithm is SCFQ, and it is shown that reasonable delay bounds are obtained for an example involving ATM.

The "Deficit Round Robin" (DRR) algorithm proposed by Shreedhar and Varghese [38] provides a very efficient alternative to FQ. The basic idea is to modify the Round Robin algorithm to make it fair. Round Robin is unfair because of different packet lengths—if one user consistently sends larger packets than another, then a correspondingly greater proportion of the bandwidth will be allocated to that user. To remedy this problem, each user is assigned a quantum of service each round, and the deficit—the amount of the quantum which could not be used due to the next packet being too large—is stored in a variable and added to the quantum on the next round. Different users may be assigned different quanta, providing a form of bandwidth control (directly analogous to "shares"). This algorithm only requires O(1) time, which is a massive improvement over all previous alternatives. The use of closed hashing is suggested in conjunction with this scheme, as was described in the Stochastic Fair Queueing scheme [23], to bring the cost of finding the connection (user) associated with a packet down to constant time, with a small probability of loss of fairness.

The fairness of DRR is analysed under the assumption that all users are backlogged at all times. Under such conditions, the algorithm provides throughput allocations in proportion to the per user quanta. This does not tell us, however, how fairly the algorithm behaves under more general conditions, when users are only backlogged for some of the time. Simulation results are provided which compare DRR with FCFS, demonstrating how DRR provides isolation between flows, protecting against the effects of an ill-behaved user. Further results demonstrate the independence of bandwidth allocation on packet sizes, and on traffic patterns.

Unfortunately, the theoretical results in the paper by Shreedhar and Varghese [38] do not apply to the general case (where users are not necessarily backlogged all of the time). Shreedhar and Varghese provide a conjectured bound on the difference in throughput allocations between DRR and PS, but do not prove it. Although not as
good as the bound for FQ, the bound should be adequate for most applications (if only it can be proved). The delay behaviour is less favourable, though—a small packet can be delayed by up to a quantum's worth for every other flow. This suggests that DRR is ideal for applications where throughput fairness is required but latency is unimportant. Shreedhar and Varghese suggest dividing the flows into "best-effort" and "latency critical" flows, and using DRR for best-effort flows, and some other FQ algorithm for latency critical flows.

Bennett and Zhang [3] observe that the throughput and delay bounds provided by WFQ are sub-optimal, and suggest a replacement for WFQ which provides superior bounds. The bounds provided by Parekh and Gallager [30] only look at one side of each of the throughput and delay discrepancy bounds. A simple thought experiment shows that the bounds on the other side are much less tight, resulting in deleteriously bursty output under some conditions. The Worst-case Fairness Index (WFI) is defined, and it is shown that for WFQ, the WFI increases in proportion to the number of users. This is undesirable, because it represents a corresponding increase in delays encountered, in comparison with GPS.

Bennett and Zhang's solution is to make use of both starting and finishing times of packets in the GPS simulation. Rather than choosing the next packet to transmit out of all the queued packets, only packets which have already started in GPS are considered. The criterion of choosing the packet with the smallest (virtual) finishing time is still used, but restricted to the set of packets which have started in GPS. This has the effect of improving the weaker bounds on throughput and delay, without compromising the strong ones. The resulting algorithm is called WF$^2$Q. Their analysis shows that strong bounds on throughput discrepancy exist, that the WFI is small, and that WF$^2$Q is work conserving. Their algorithm is thus much preferable to WFQ.

The shortcomings of WFQ demonstrated by Bennett and Zhang also apply to our HFQ algorithm. Unfortunately, it does not seem possible to apply their technique to improve HFQ, because in HFQ, the finishing order of requests is not known ahead of time as it is with WFQ. Thus HFQ orders requests by starting time in HPS. It is possible that better bounds might be obtained from HFQ by estimating the finishing order (in HPS) of those requests which have already started in HPS, by
disregarding the effect of future arrivals.

Bennett and Zhang further refine their technique in a second paper [2], with an algorithm which they call $WF^2Q+$. This algorithm is based around an alternative function for virtual time (as is SCFQ), which is designed to be able to be calculated in (worst case) $\mathcal{O}(\log N)$ time. Although this is less efficient than SCFQ, which boasts a worst case time of $\mathcal{O}(1)$ for calculating its virtual time, it is better than WFQ, which has a worst case time of $\mathcal{O}(N)$ [12]. The reason for preferring $WF^2Q+$ over SCFQ is that it achieves a lower WFI than SCFQ (in fact, it achieves the best possible WFI for a packet based discipline). This is of particular significance to their results on Hierarchical Fair Queueing. Bennett and Zhang show that HPS (which they independently discovered and refer to as H-GPS) may be approximated by nesting GPS-approximating servers. Their analysis shows that the delay seen by Leaky Bucket traffic is bounded. The expression for the bound is based on the WFI of each server, such that a smaller WFI will yield a smaller bound. Of course, this shows that small WFI’s are sufficient, but not that they are necessary (which may well be the case, but they have not shown it). We believe that our methodology of Chapter 5 may prove useful in exploring this problem further.

It is interesting to compare Bennett and Zhang’s approach to our own. We have taken a different approach to implementing HFQ: rather than approximate each GPS server in the decomposition of HPS separately, we approximate the whole HPS as a single unit. Our approach clearly has the potential to provide better bounds on the approximation, since in a system made up of many GPS approximations the errors are cumulative. On the other hand, their approach is more efficient to implement. Again we are faced with a tradeoff between accuracy and efficiency.

The paper by Rexford et al. [35] examines the problem of implementing an FQ-like algorithm efficiently in hardware, for high-speed networks such as ATM. SCFQ [12] was chosen as the basis of the architectures developed in the paper, since it is substantially more efficient and simple to implement compared to FQ. The paper describes three architectures based on SCFQ. In the first, the weights are restricted to values which are the reciprocal of integers less than some maximum. (This restriction is justified on the basis that it is only the ratio between the weights that is important). Under this restriction, the virtual finishing times calculated by
SCFQ are always integers. Furthermore, the virtual finishing time at the head of an active queue is always within the maximum reciprocal of the currently executing packet's virtual finishing time. This enables the calculations to be performed using modulo arithmetic, and furthermore allows an efficient hardware solution to the problem of sorting the requests by finishing time. The architecture maintains a set of per-connection FIFOs, and a finite set of FIFO sorting bins. The request at the head of each active connection is placed in the appropriate sorting bin, based on the modulo arithmetic calculations mentioned above. This consists of a single addition for each arrival and departure. The sorting mechanism requires a small amount of logic to locate the next non-empty bin. The number of sorting bins places a limit on the range of allowable weights.

The restriction placed on the weights is a bit too limiting, so the other two architectures provide ways of increasing the range of possible weights without further increasing the number of sorting bins. One technique is to only approximate SCFQ, by allowing each bin to represent a range of virtual finishing times. The fractional portion is held in a per-user variable, and updated when each packet reaches the head of its queue.

The other technique is to use a two-level hierarchical scheme, which is referred to as Hierarchical Fair Queueing. (The object of this scheme is to make the scheme capable of handling a wider range of connection rates, rather than to increase the flexibility of bandwidth allocation, as with other hierarchical schemes such as our own). The first stage in their scheme consists of a number of schedulers using the finite sorting bins approach outlined above, with weights assigned to each connection. The second stage chooses the next packet to service out of these “groups” of connections, with weights assigned to the groups. There are two variants of the scheme, one with static group weights (set to approximate the desired GPS policy when all queues are backlogged) and a second with dynamic group weights, where the group weights try to adjust to match the current activity pattern. The scheme allows a wider range of weights to be simulated, by putting connections with similar rates in the same group, and allowing the groups to have a much wider range of rates, but with a small number of groups, allowing efficient hardware implementation, say by a tree of comparators.
The various schemes have been evaluated by simulation. A surprising result is that the dynamic hierarchical scheme introduces less distortion than the non-hierarchical schemes, even the "exact" version of SCFQ. The (much needed) analysis of the approximate SCFQ architectures is left as future work.

The techniques described by this paper are clearly of importance to the problem of obtaining an efficient implementation of a non-hierarchical GPS approximation. They are also of interest to us for a few reasons. They show an alternative strategy for producing an HFQ system. They show how HFQ-like systems may be of use in dealing with the problem of efficient implementations of GPS approximations. And they demonstrate that algorithms such as our HFQ have a long way to go before they can compete with the fastest non-hierarchical algorithms. It is still an open question whether the techniques similar to those of Rexford et al. [35] can be applied to our HFQ algorithm, and one for further research.

An impressive improvement on the efficiency of the algorithms described so far is provided by Leap Forward Virtual Clock [40]. This algorithm provides throughput and delay bounds close to those of WF$^2$Q, with a processing time per packet of $O(\log \log N)$. An advantage of this scheme is that it does not require special hardware. It also (like WF$^2$Q) avoids the bursty behaviour exhibited by WFQ and SCFQ.

The basic technique used by LFVC takes the existing Virtual Clock algorithm and modifies it to eliminate the throughput unfairness which is characteristic of standard VC. VC tags packets with the time that they should be transmitted by, as described in section 2.6. The authors have formulated two conditions which are required for a tag-based service discipline to provide delay and throughput guarantees. A "backlog inequality" captures the necessary condition for the server to complete each request by its due-time tag. The paper shows that this condition is also sufficient and that an algorithm satisfying the inequality provides delay guarantees. The other condition is the "throughput condition", which ensures that a flow's current tag does not deviate too far from the current server clock. In order to maintain this constraint, LFVC maintains two queues—a priority queue for "well-behaved" flows, and a holding area for "oversubscribed" flows. Flows which attempt to break the throughput condition are "quarantined" by moving them to the holding
queue. Some time later, when they are no longer in violation of the throughput condition, they are moved back to the main queue. This is done such that the backlog inequality is never violated, so the delay condition is still met. The problem of what happens when all of the active flows end up in the holding area is dealt with by the "leap forward" part of the algorithm—the server clock is advanced as far as possible without violating the backlog inequality. This enables at least one flow to be moved back to the main queue.

The efficient implementation is achieved by using tag coarsening. Tags are calculated and maintained exactly, but the priority queues use rounded tags, which have been rounded up to a multiple of some basic quantum. This restricts the number of possible tag values to a finite set, thus enabling the use of a finite universe priority queue. The van Emde Boas data structure is one such queue, which achieves $O(\log \log N)$ behaviour on insertions and deletions. All other parts of the LFVC algorithm require constant time to execute, thus the overall cost of the algorithm is $O(\log \log N)$. Tag coarsening adds a small constant to each of the throughput and delay bounds.

LFVC has been analysed, using the Guaranteed Rate Clock framework of Goyal et al. [14], to demonstrate a finite delay bound for suitably Leaky Bucket constrained arrivals. Its throughput fairness has been analysed using two measures of fairness, one due to Golestani [13], and the Worst-case Fairness Index (WFI) of Bennet and Zhang [3]. Both measures are shown to be bounded by finite constants independent of the number of users. The analysis includes the effects of tag coarsening, which is shown to add a small constant to each of the bounds.

In the conclusion, comments are made to the effect that tag coarsening may be applicable to other service disciplines; of course its effect on each discipline needs to be analysed on a case-by-case basis. It is interesting to compare this tag coarsening strategy with the methods used by Rexford et al. [35] to restrict the set of possible priorities to a finite set. As future work, the authors are planning to use LFVC in a Hierarchical Fair Queueing system.

As the papers discussed in this section show, there are a large variety of methods which may be used to improve the efficiency of Fair Queueing. We noted at the end of section 2.2 that HFQ is naturally harder to implement efficiently than FQ, due
to the added complexity of the hierarchical policy. Although we have not managed to find any way of significantly closing the gap, the variants of FQ described above give us some hope that the efficiency problem for HFQ is soluble. One approach which shows some promise is the approximation of HPS by nested instances of an FQ-like algorithm. Such systems have been analysed using the WFI measure [2]; however, there are still unanswered questions. The techniques described in Chapter 5 may be of use here.

2.4. Theory of Fairness

The paper "How Fair is Fair Queueing" by Greenberg and Madras [15] contains important contributions to the understanding of Fair Queueing algorithms. The bulk of the paper is concerned with the theoretical correctness of the FQ algorithm, but the paper also looks at the question of efficiency, the applicability of a variant of Fair Queueing to CPU scheduling, and includes some simulation results.

Greenberg and Madras link the work done by Demers et al. on Fair Queueing to the Processor Sharing discipline introduced by Kleinrock [25]. Where Demers et al. talk about "Bitwise Round Robin", it is actually Processor Sharing that they are looking for, as becomes clear on examination of the equations that they use. Greenberg and Madras have recognized this fact, along with the realization that Processor Sharing provides a good definition of fairness, in that the goal of any Fair Queueing algorithm should be to "emulate" Processor Sharing, subject to the constraint that jobs already in service can not be preempted. They discuss what it means to emulate PS, first looking briefly at the "Max-Min Fairness Condition", and then making the observation that FQ emulates PS in a much stronger way, in that it provides bounded discrepancies in cumulative throughput for each user.

The paper extends the definition of Demers et al.'s FQ algorithm to provide two variants: FQS and FQF. FQF is the original FQ algorithm, which orders requests based on finishing times in PS. FQS is a new variant, using starting times instead of finishing times.

As the paper shows, both FQS and FQF provide good approximations to the PS discipline. FQS has an additional advantage: it may be implemented on a system where the job length is not known at the time of arrival. As is pointed out, this
makes FQS useful for CPU scheduling applications. Greenberg and Madras may have underrated its importance slightly, reserving the technique for systems where the cost of switching between jobs is large. We believe that it is useful even for systems where the cost is small, given the importance in modern operating systems of providing fast response times to "interactive" processes. Fair Queueing provides a natural way of achieving this goal without compromising fairness.

The main section of the paper of Greenberg and Madras [15] consists of the mathematical analysis, where it is proved that both FQS and FQF approximate PS in a rigorously defined manner. The technique used is to compare the behaviour of PS and FQS/FQF when confronted with identical arrival sequences. Subject to the assumption that request lengths are bounded, they show that the discrepancies between throughputs provided by PS and FQ for each user remain bounded. Hence, the practical method (FQ) provides essentially the same throughputs as the ideal method (PS). In addition, they prove that the discrepancy in delays also remains bounded. These proofs do not assume a particular arrival distribution; the results are true for an arbitrary arrival sequence (subject to the bound on packet lengths).

FQ is based on a simulation of PS. A more efficient method for simulating PS is provided, requiring O(log N) time to simulate each event, rather than the usual O(N) time. This uses a priority queue to keep track of virtual finishing times of active jobs. A description of how to use this technique to implement FQF is also given; it has a cost of O(log N) per job.

An important step in their mathematical analysis of Fair Queueing consists of proving that the FQ algorithm always has enough information to choose the next packet to service; i.e., that the algorithm is consistent and well-behaved. It is here that the different requirements of FQS and FQF come out: FQS doesn't need to know a request length until that request has finished, whereas FQF needs to know it at the time that the request arrives.

The paper also includes some results of simulations performed on PS, FQS and FQF, which show how the algorithms behave under a range of traffic conditions, and support the theoretical results.

Some other variations are mentioned at the end of the paper. They give a generalization of FQ to provide per-user weights (now known as Weighted Fair
Queueing) along with some conjectured bounds on the throughput discrepancies (which they state that they have not proved). They also refer to a variation of FQ by Heybey and Davin [8], which does not achieve the same discrepancy bound, but which avoids the need to simulate PS.

In a pair of papers, Parekh and Gallager [30], [29] provide a theoretical analysis of Generalized Processor Sharing (GPS) in the presence of Leaky Bucket constrained traffic. The first paper [30] deals with the single node case. A definition of GPS is given, which can be shown to be equivalent to Kleinrock's "Processor Sharing with priorities" [25]. Leaky Bucket [41] provides a model for specifying constraints on incoming traffic based on average rate, peak rate and burstiness, and is the main "flow control" algorithm favoured for real-time applications. It has been further examined by Cruz [6], [7].

The main result of Parekh and Gallager's first paper [30] is that, under the assumption of Leaky Bucket constrained traffic, tight bounds on the worst case packet delay, output burstiness and queue length (backlog) are provided by GPS, as long as the sum of the average rates does not exceed the resource capacity. A feature of their analysis is that they do not assume that the average rate in the Leaky Bucket constraints does not exceed the guaranteed rate under GPS. This provides a more general, flexible approach than one which does—it is possible, for instance, to give a non-delay sensitive session a smaller guaranteed rate than its average sending rate, and give delay sensitive sessions a much larger guaranteed rate. The analysis shows that delay and burstiness still remain bounded under these conditions. (Of course, the sources have to be trusted to obey their Leaky Bucket constraints in order for the analysis to be valid). Some parts of the analysis are actually applicable to arbitrary (work conserving) service disciplines—they show that the system busy period (maximal interval over which at least one session is always backlogged) is bounded, provided that the sum of the average source rates does not exceed the server capacity. This implies that the delay is bounded under these circumstances, for arbitrary service disciplines. A better bound on the delay is obtained in the case of GPS.

Parekh and Gallager also show how GPS may be approximated by a practical algorithm which they call PGPS. This is actually the same as WFQ defined by Demers et al. [9]—they developed it independently. The paper includes a derivation
of bounds on the delay and throughput discrepancies between PGPS and GPS. (These bounds are one-sided, unlike the bounds of Greenberg and Madras for PS, which give both sides of the bounds.)

The multiple node case (networks of GPS servers) is dealt with in a second paper [29]. This case is considerably more complex than the single node case. The main source of this extra complexity is that cycles in the network can lead to a phenomenon called “virtual feedback”, which can make the system unstable. Parekh and Gallager provide two solutions to this problem. The first is an analysis of “locally stable” sessions, which are sessions for which the average (Leaky Bucket) rate does not exceed the guaranteed (GPS) rate. For such sessions, simple bounds on the end-to-end delay and queue length exist, which are independent of the behaviour of the other sessions. Further, these bounds are superior to the result of adding the worst case bounds at each node along the path.

Of course, sessions which are not locally stable are of interest, as described in the single node case. Parekh and Gallager have developed a methodology called the Consistent Relative Session Treatment (CSRT) which allows such systems to be analysed, provided that they meet a certain constraint. This constraint is that the sessions can be ordered in such a way that no session “impedes” a later session in the ordering at any node in the network. The notion of “impedance” is defined by comparing the ratios between shares and average rates. The definition of CSRT captures the need for a session to be treated “consistently” well at each node in the network. Special cases of CSRT include Uniform Relative Session Treatment (USRT), where each session receives the same number of shares at each node along its path, and Rate Proportional Processor Sharing (RPPS), where each session has shares equal to average rate at every node (thus every session is locally stable). Parekh and Gallager show that under the CSRT bounds exist on the delay and burstiness. They include an examination of the effects of using PGPS (WFQ) instead of GPS in their analysis.

The Guaranteed Rate Clock framework of Goyal, Lam and Vin [14] provides end-to-end delay bounds for a general class of service disciplines. The class GR introduced in the paper includes Virtual Clock, PGPS (WFQ) and SCFQ. GR is defined in terms of a very simple constraint which captures the property of service rate guarantees as provided by the service discipline. Bounds are then derived on the
end-to-end delay in a network of GR servers, for two source traffic models: Leaky Bucket and Exponentially Bounded Burstiness (EBB) (a statistical model defined in a paper by Zhang et al. [49]). The analysis assumes that the session in question conforms to the RPPS constraint; that is, the average rate in the Leaky Bucket (or EBB characterization) is equal to the guaranteed rate in the GR characterization. Equivalently, the session is locally stable. The authors claim that this loss of generality is not a problem and that relaxing the constraint is inadvisable due to the possibility of misbehaving sources violating their constraints and hence negating the guarantees provided to other sessions. The GRC framework is general enough to be applicable to both HPS and HFQ. We make use of it in Chapter 5, to derive end-to-end delay bounds for these disciplines.

2.5. Hierarchical Sharing

A number of hierarchical sharing schemes have been described so far in the previous sections. The discussion of these methods has concentrated on features of the algorithms apart from the hierarchical sharing itself. In this section we collect all of the hierarchical schemes together and discuss their hierarchical nature, comparing their approaches.

Cambridge Share was extended [26] to provide a form of hierarchical sharing. At each node is kept the number of shares belonging to that node along with a decayed usage and various housekeeping variables. Three allocation “modes” are available, which can be set on a per-node basis. The basic strategy of the system is to define “real shares” associated with each node based on a recursive calculation (which depends on the mode). The “real shares” of the leaf nodes are then used in the place of shares to convert the original, non-hierarchical system into a hierarchical one. The “fixed” allocation mode divides the resource up according to shares, without taking user inactivity into account. The “sliding” allocation system does take inactivity into account, allowing the unused capacity to be redistributed to other users within a group (this is preferred). The “automatic” allocation system effectively allocates shares in proportion to the amount of work being done, as if all users were on the one node. The paper is vague about such details as how frequently the “real shares” need to be recalculated, and whether the proposed calculations actually achieve their objective. No theoretical justification is provided for the
algorithm.

Basser Share [21] also employs a hierarchical structure with shares and decayed usage at every node in the tree. A similar technique is used to Cambridge Share, with the “fixed” allocation method—the “machine share” of each node is calculated by recursively dividing up the resource according to shares. This “machine share” is then used in the place of ordinary shares in a non-hierarchical scheduler. In an attempt to deal with the problem of user inactivity, a heuristic is used to adjust the charges recorded at each node (used in the priority calculations). The hierarchical part of Basser Share has never been analysed; even without the hierarchical part, Basser Share is analytically intractible in the general case.

VSTa [42], [43] employs a hierarchical version of the probabilistic Lottery Scheduling [46] approach. At each scheduling decision, the scheduler starts at the root node, recursively picking an active child at random, until an active leaf node (process) is reached (which is then run). The random choice is weighted by shares—a random number between 0 and the sum of the active children’s shares (minus 1) is generated, and used to select a child. VSTa appears to provide a probabilistic approximation to HPS. The main drawback of the approach is that it is only probabilistic, so there are no guarantees.

Recent network service disciplines such as Bennett and Zhang’s Hierarchical Packet Weighted Fair Queueing [2] and Rexford et al.’s Hierarchical Fair Queueing [35] nest GPS-approximating service disciplines to create an HPS-approximating service discipline. The technique has been analysed by Bennett and Zhang [2]. Their analysis only provides a delay bound for Leaky Bucket constrained traffic; there is clearly a need for further analysis to establish other bounds, such as throughput discrepancy bounds. It should be noted that the size of Bennett and Zhang’s delay bound grows with the depth of the tree.

The CBQ service discipline [45] and the associated Link Sharing framework [11] are described in detail in the next section.

Our survey of the literature reveals that the concept of sharing a resource hierarchically has been around for some time; however, few of these systems have made use of a rigorous definition of hierarchical fairness.
2.6. Alternative Approaches to Sharing

In this section, we examine some approaches to bandwidth which are not based on the Processor Sharing fluid flow paradigm (or its generalizations). Two such approaches are described: the VirtualClock service discipline, and the Link Sharing methodology (with its associated CBQ algorithm).

Zhang's VirtualClock [48] is a rate-based traffic control algorithm for packet-based networks. It is intended to be used in conjunction with explicit resource reservation. Each user must specify an average rate of throughput at connection setup time. VirtualClock is designed to enforce the throughput reservation by controlling the order in which packets are serviced by a network resource. This protects users from the effects of a mis-behaving source.

VirtualClock is conceptually derived from TDM (Time Division Multiplexing), which allocates fixed time slots for network conversations. This has the drawback of wasting the resource if a conversation is idle when it is its turn to transmit. VirtualClock attempts to provide the bandwidth guarantees of TDM without the wastage.

To each conversation \( i \) is associated a "virtual clock" value \( \text{auxVC}_i \). The average rate, defined at connection setup time, is referred to as \( AR_i \). Upon each packet arrival, the virtual clock value is updated as follows

\[
\text{auxVC}_i \leftarrow \max(\text{arrival time}, \text{auxVC}_i) + \frac{1}{AR_i}
\]

and the packet assigned a priority equal to the new value of \( \text{auxVC}_i \). Packets are serviced in increasing order of priority.

Simulation results are given, which show that the algorithm provides the required bandwidth allocations, and protects against mis-behaving sources. It has been shown elsewhere [10], [14] that VirtualClock provides delay guarantees for Leaky Bucket constrained arrivals.

A weakness of the method is that it only applies to real-time traffic, and is not suited for data applications. Furthermore, it has been shown to be unfair [40]. Zhang [48] compares VirtualClock with Fair Queueing, and points out that Fair Queueing (as originally defined) does not allow for non-uniform bandwidth allocations, but gives every user an equal share of the bandwidth. This shortcoming of FQ
has since been remedied by the addition of per-user weights to the algorithm. Another possible advantage to VirtualClock is that it is simpler to implement and more efficient than FQ. However, faster alternatives to FQ now exist which also provide throughput fairness (e.g., SCFQ).

Link Sharing, as introduced by Floyd and Jacobsen [11], is a general framework for providing hierarchical bandwidth sharing in a network gateway. Class Based Queueing (CBQ) [45] 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 attributes controlling its allocation. Whereas HPS has a single attribute per class (the number of shares), Link Sharing has multiple attributes and is more complex.

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 the paper by Floyd and Jacobsen [11]. In practice, there may really only 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, but 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 (see section 3.1.2).

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 it. Link Sharing also provides the ability for a node to nominate a class from which it can borrow, 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 marking nodes as exempt, bounded, or isolated. An exempt node is not affected by the Link Sharing scheduler. It 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 it 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 (an undesirable state of affairs). The “bounded” attribute could be simulated by introducing admission control. We consider the “isolated” attribute undesirable, because it implies a non-work conserving service discipline.
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 (as demonstrated in Chapter 3).

2.7. Summary

Examination of the literature reveals a need for good HFQ algorithms. The ideal HFQ algorithm would be as efficient as possible, while providing tight delay bounds and throughput guarantees. A variety of solutions have been proposed for the non-hierarchical case, providing various tradeoffs between efficiency, delay bounds and throughput bounds. Apart from our HFQ algorithm, one other HFQ algorithm has been published, using a different approach to ours. Although our algorithm is less efficient, we believe that it will provide superior delay bounds. Other approaches not based on HPS have also been suggested, but the evidence suggests HPS is the most appropriate fairness paradigm for networking.
3. Defining Fairness

In this chapter we examine what it actually means to allocate a resource fairly, in particular for hierarchical policies. Our main focus is the underlying policy used by a service discipline, rather than the specific service discipline. We start by looking at some existing attempts to answer the question in the non-hierarchical case, and give reasons why Processor Sharing is the superior approach. We then present an informal definition of Hierarchical Processor Sharing (HPS), using examples to explain how HPS works.

For the purpose of discussion, we will assume that a resource provides a constant rate of service. We generalize this in Chapter 5 to allow the service rate to vary over time.

3.1. The Non-Hierarchical Case

Two basic approaches have been used to define fairness. On the one hand, we have definitions which attempt to define fairness in terms of behaviour over an interval, such as the underlying policy of Basser Share [22], [21] or Link Sharing [11]. We refer to such schemes generically as Long-Term Sharing. On the other hand are definitions based on instantaneous allocation of the resource, such as Processor Sharing (PS) [25]. We refer to these schemes as Instantaneous 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, Instantaneous Sharing results. When the average rate over an interval is used Long-Term Sharing results.

To these two approaches we add the Max-Min Fairness Condition [9], which appears to be an attempt to fix some of the problems with Long-Term Sharing. However, the definition does not actually state how “rate” should be measured, and so can be considered to belong to either class. Hence we consider it separately.
3.1.1. Long-Term Sharing

Long-Term Sharing is the model used by Link Sharing [11], CBQ [45], Cambridge Share [27], [26], and Basser Share [21], [18], [4], [22].

Long-Term Sharing, as the name implies, seeks to share out the resource amongst its users over a period of time. The period may be seconds or days; we call it “Long-Term” because these are long periods of time when compared with Instantaneous Sharing. An typical example of this approach is found in an early technical report on Basser Share [22]:

The final requirement that the system seem fair was interpreted thus: each user is entitled to a certain proportion of the machine’s power and that he sees as the right to do a certain amount of work over a period of a few days.

Clearly this definition is equivalent to saying that each user is entitled to receive service at a certain rate, where rate is measured as an average (arithmetic mean) over a few days. This definition seems quite straightforward, yet it is virtually unusable as it stands. How is the server to enforce this policy without knowing about future arrivals? What does it mean when a user gets more than their share because it is the only active user for the first half of the interval? And should they receive any service for the remainder of the interval if another user becomes active?

It is perhaps because of these problems that Basser Share does not actually use the “arithmetic mean” version of rate, even though the above quote implies it. Instead, it uses an exponentially decayed average of past usage to decide what level of service a user is entitled to at present. By making a user’s instantaneous rate inversely proportional to this usage history, and proportional to the square of the user’s shares, Basser Share attempts to push the system towards an ideal operating point, where this exponentially averaged usage is proportional to the number of shares [21], [16]. This has been shown to work in a special case [16], but in the general case the system is intractable analytically.

Most practical systems based on the concept of Long-Term Sharing use an exponentially decayed average. CBQ [45] is the most recent example, using this technique to calculate the rate at which each user is working, and adjusting service accordingly. The method is different to that of Basser Share, but the basic idea is
similar.

Closely related to this approach are congestion control techniques such as DECbit [20], [19], [33], [5], [34] and Van Jacobsen's algorithm [44]. Such algorithms provide a great improvement in fairness over networks without any fairness controls; however, they are no substitute for a Fair service discipline with a guaranteed rate of service.

3.1.2. Problems with Long-Term Sharing

We have determined that Long-Term Sharing suffers from the following flaws: 

*arbitrariness, analytic intractability, 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 (e.g., 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 analytically intractible. In particular, systems based on decayed averages tend to be too difficult to analyse for arbitrary arrival sequences (as is possible with Instantaneous Sharing). There are
no throughput or delay guarantees available for any Long-Term Sharing based systems.

Lastly, we claim that Long-Term Sharing is unfair. Consider again a system with two 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 no-one 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, which is intuitively fair, able to be analysed, and independent of the interval over which its behaviour is observed.

3.1.3. Max-Min Fairness Criterion

We noted some problems with the Long-Term Sharing paradigm in the previous section. One of these is unfairness, as a consequence of failing to deal with variations in the users' activity levels. Long-Term Sharing does not include the effects of a user's inactivity in the definition; each user is presumed to get a certain amount of service, whether they want it or not.

An attempt to overcome this difficulty is found in the Max-Min Fairness Criterion [9], [34]. This definition supposes a server with a fixed capacity $\mu_{total}$, and $N$ users requesting amounts of service $\rho_1, \ldots, \rho_N$. The task of the server is to allocate amounts of service $\mu_1, \ldots, \mu_N$ in a fair way. The Max-Min Fairness Criterion states that the allocation is fair if the following 3 conditions hold (taken from Demers et al. [9]):

1. no user receives more than their request ($\mu_i \leq \rho_i$)
2. no other allocation scheme satisfying (1) has a higher minimum allocation
3. condition 2 remains recursively true if we remove the minimal user and adjust the total resource accordingly, $\mu_{total} \gets \mu_{total} - \mu_{min}$

Note that the first condition is the boundary condition which ensures that the
allocation is "physically" possible. The second condition is the source of the name "Max-Min".

The same 3 conditions may be used to define a "fair" allocation for the multiple resource case [34]. We restrict our attention to the single resource case here. The conditions have a "solution" of the form

$$
\mu_i = \min(\mu_{fair}, \rho_i)
$$

where $\mu_{fair}$ is chosen so as to ensure that

$$
\sum_{i=1}^{N} \mu_i = \min(\mu_{total}, \sum_{i=1}^{N} \rho_i)
$$

This definition appears to deal with varying levels of activity, by taking the request amount ($\rho_i$) into account; however, there are still problems with the definition. The "amount" of service requested by each user is assumed to be a scalar; no attempt is made to relate this to real-life scheduling problems. If the scalar is to be interpreted as a rate, what type of rate measurement is to be used? We note that if one uses a Long-Term definition of rate, such as the average over an interval, then the definition suffers from the same problem as Long-Term Sharing described in the thought experiment in the previous section. Exponentially decayed averages fare no better.

Suppose we choose to use instantaneous rates in conjunction with Max-Min Fairness—at each instant, a scheduling decision is made based on the instantaneous rates of service requested. Then the definition reduces to the Processor Sharing discipline described in the next section. To see this, note that at any instant, each user can only request either 0 or $\mu_{total}$ (there may be more choice for a multiprocessor/multi-channel resource; we are implicitly assuming a single resource). Under this constraint, $\mu_{fair} = \frac{\mu_{total}}{N_{ac}}$, where $N_{ac}$ is the number of active users, which are the users with non-zero $\rho_i$. $\mu_i = \mu_{fair}$ for these users. Thus, at each instant the resource is split equally amongst the active users, which is how Processor Sharing is defined.
3.1.4. Processor Sharing

In its simplest form, Processor Sharing divides the available bandwidth at each instant amongst all of the active users. Thus, if at time $t$ there are $N_{ac}(t)$ active users, and the resource capacity is $R$, then each active user $\alpha$ receives service simultaneously at a rate of

$$\frac{d}{dt} \mu_{\alpha}(t) = \frac{R}{N_{ac}(t)}$$

where $\mu_{\alpha}(t)$ is the cumulative service received by user $\alpha$ at time $t$ (note that this formula is for active users only). This readily extends to the concept of shares—if user $\alpha$ has $w_{\alpha}$ shares, then active users receive service simultaneously in proportion to shares:

$$\frac{d}{dt} \mu_{\alpha}(t) = \frac{Rw_{\alpha}}{\sum_{\beta \text{ active}} w_{\beta}}$$

Again, this is for active $\alpha$ only. The service discipline described by this equation is called Generalised Processor Sharing (GPS). GPS provides throughput guarantees—each user is assured a minimum rate of service; for user $\alpha$, it is

$$r_{\alpha} = \frac{Rw_{\alpha}}{\sum_{\beta = 1}^{N} w_{\beta}}$$

GPS has also been shown to provide delay bounds to traffic which is Leaky Bucket constrained (with suitable parameters) [30], [29]. It is thus a desirable service discipline for use in networks. Unfortunately it cannot be implemented exactly, due to the requirement that all active users be serviced simultaneously. Nevertheless, it can be approximated quite well, as has been shown [30]. This provides us with a useful Fairness paradigm for non-hierarchical policies—we call an algorithm “Fair” if it approximates GPS closely. The relative Fairness of two service disciplines can be compared by comparing their throughput and delay discrepancy bounds. This definition is intuitively fair, non-ambiguous and susceptible to analysis—in short, all that Long-Term Sharing and Max-Min Fairness are not.
3.2. Hierarchical Processor Sharing

We have compared the available definitions of Fairness, and seen that (Generalised) Processor Sharing is preferable as an idealisation of Fairness for non-hierarchical policies ("idealisation", in that we can at best only approximate it in real life). We need a corresponding definition for hierarchical policies. Hierarchical Processor Sharing (HPS) is a natural extension of PS to hierarchical policies.

HPS is a generalization of PS: not only do we allow different weights to be assigned to each user (as in GPS), but we also generalize 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 3.2.1. 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.

```
+---+---+---+
| 5 | 2 | 3 |
| 50%| 20%| 30%|
+---+---+---+
```

```
user0
+---+---+---+
| 2 | 3 |
| 8%| 12%| 12%|
+---+---+---+
user1 user2 user3
```

```
Group1
+---+---+---+
| 2 | 2 |
| 12%| 12%| 6%|
+---+---+---+
user4 user5
```

```
Group2
+---+---+---+
| 1 |
| 16%|
+---+---+---+
user6
```

**Figure 3.2.1: Example of HPS (all users active)**

In figure 3.2.2, 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.

```
+---+---+---+
| 5 | 2 | 3 |
| 0%| 40%| 60%|
+---+---+---+
```

```
user0
+---+---+---+
| 2 | 1 |
| 40%| 10%| 20%|
+---+---+---+
user1 user2 user3
```

```
Group1
+---+---+---+
| 2 |
| 40%|
+---+---+---+
user4
```

```
Group2
+---+---+---+
| 2 |
| 0%|
+---+---+---+
user5
```

**Figure 3.2.2: Example of HPS (some users active)**

We will give one further example. Consider figure 3.2.3, 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%.

![Tree diagram]

Figure 3.2.3: 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 queuing, providing hierarchical bandwidth allocations, and guaranteed behaviour for both
bandwidths and delays.
4. The HFQ Algorithm

This chapter describes the Hierarchical Fair Queueing (HFQ) algorithm. This algorithm contains a simulation of the Hierarchical Processor Sharing (HPS) service discipline, which was defined in Chapter 3. The starting times of packets in the simulation are used to order the actual transmission of packets in the network. We have combined this basic, abstract description of the algorithm with various implementation decisions to produce the concrete algorithm presented here.

Section 4.1 provides an overview of the algorithm and explains the rationale behind various choices made in its design. Section 4.2 discusses data structures used in the algorithm. Section 4.3 introduces notation for describing the algorithm. Section 4.4 then presents the algorithm in pseudocode form. In Section 4.5 we consider the question of complexity.

4.1. Overview

We begin by explaining the basic ideas behind the HFQ algorithm. The goal of the algorithm is to approximate the HPS service discipline, which we cannot implement directly due to its requirement that we service all active users simultaneously. Instead, we 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 [9]. 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. The situation is further complicated by the fact that future arrivals can affect the order in which requests finish service. Thus it may not be possible to service the requests in the correct order if we attempt to order them by finishing times. Contrast this again with FQ, in which the order of finishing times is fixed once the packets in question have arrived, and the theoretical behaviour has been analysed for both variants [15].

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 it arrives. This fact creates a problem which does not exist
for the FCFS discipline. In FCFS, packets are serviced in the order in which they arrive. Thus the arrival of a future packet cannot affect the starting time of any packets already in the system. This is not true in HPS. Since HPS services all users simultaneously, the arrival of a new packet can change the starting times of packets already in the system, 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 Processor Sharing (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 [9]. 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.

There is no analogous concept of virtual time in HPS. Implicit in the definition of virtual time 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—consider the allocations given to user1 and user3 in figures 3.2.1 and 3.2.2. When all users are active, user3 receives a greater proportion of the bandwidth than user1. In the example of figure 3.2.2, however, they receive equal portions. Thus we do not have the simplification afforded by virtual time, 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 does not generate the HPS starting events one at a time, but rather calculates the longest period of time that it can advance the state of the simulation by without waiting for further arrivals, and generates all HPS starting events within this interval with a single sweep of the tree. This 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, making the system non-work-conserving. This cannot happen, because it implies that the system is servicing requests at a rate faster than the HPS simulation runs at. A rigorous proof of this may be found in Chapter 5. Of course, it is crucial for this result that the rate of service used in the HPS simulation matches the actual resource rate, otherwise the algorithm may not behave correctly.

A full description of the algorithm, including data structures and pseudocode, constitutes the remaining sections of this chapter.

4.2. 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 Chapter 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. It may be a heap or similarly efficient structure. The other requests are held in the queue corresponding to the user owning the request. These are processed in FIFO order, hence a simple list with pointers to the head and tail is sufficient for these queues.

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

4.3. Notation

Given $\alpha$, a node of the class tree, the following are defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$.parent</td>
<td>the node's parent</td>
</tr>
<tr>
<td>$\alpha$.children</td>
<td>the set of all the node's children</td>
</tr>
<tr>
<td>isleaf($\alpha$)</td>
<td>true if node is a leaf node, false otherwise</td>
</tr>
<tr>
<td>$\alpha$.queue</td>
<td>the per user queue associated with the node (leaf nodes only)</td>
</tr>
<tr>
<td>$\alpha$.creq</td>
<td>cumulative requests received by this node (leaf nodes only)</td>
</tr>
<tr>
<td>$\alpha$.csrv</td>
<td>cumulative service received by this node (leaf nodes only)</td>
</tr>
<tr>
<td>$\alpha$.creql</td>
<td>cumulative requests passed to priority queue (leaf nodes only)</td>
</tr>
<tr>
<td>$\alpha$.shares</td>
<td>the number of shares of the node</td>
</tr>
<tr>
<td>$\alpha$.sasheares</td>
<td>sum of active shares of the node's children (if any)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>the root of the class tree</td>
</tr>
<tr>
<td>pri_q</td>
<td>the global priority queue</td>
</tr>
<tr>
<td>t</td>
<td>current time</td>
</tr>
<tr>
<td>tlast</td>
<td>time of last event in HPS simulation</td>
</tr>
<tr>
<td>rmax</td>
<td>the resource capacity</td>
</tr>
<tr>
<td>rendez</td>
<td>rendezvous used for communication between new_request and next_service</td>
</tr>
</tbody>
</table>

4.4. 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, $\alpha$, and the request itself, req. The pseudocode for the function is as follows:
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 on 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:

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 4.1 and proved in Chapter 5.

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:

```c
/*
 * Given a node "α", adjust its ancestors' sashares
 * (sum of active shares) 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 'a', adjust its ancestors' sashares
* (sum of active shares) attribute to reflect a's transition
* from active to idle
*/

inactive(a) {
    if (a == root)
        return;
    a.parent.sashares -= a.shares;
    if (a.parent.sashares == 0)
        inactive(a.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 a and a rate of service r, and recursively searches the node's descendants for
the next node to become idle (in HPS), returning a tuple (b, tnext), consisting
of the predicted node b, 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 \( \alpha \), and a rate \( r \), search \( \alpha \)'s descendants for the next change in active status, assuming no intervening arrivals.

returns: \((\beta, t_{\text{next}})\), where \( \beta \) is the leaf node of the next predicted transition, and \( t_{\text{next}} \) is the amount of time until this transition is predicted to occur assuming no intervening arrivals.

```
predict(\alpha, r)
{
    if (isleaf(\alpha)) {
        if (\alpha.creq > \alpha.csrv)
            return (\alpha, (\alpha.creq-\alpha.csrv)/r);
        else
            return (nil, 0);
    }
    if (\alpha.sshares == 0) /* idle -- no departures possible */
        return (nil, 0);

    /* search children for next transition */
    r /= \alpha.sshares;
    (\beta, t_{\text{next}}) = (nil, 0);
    for (\gamma in \alpha.children) {
        (\delta, t_{\text{try}}) = predict(\gamma, r*\gamma.shares);
        if (\beta == nil || \delta != nil && t_{\text{try}} < t_{\text{next}})
            (\beta, t_{\text{next}}) = (\delta, t_{\text{try}});
    }
    return (\beta, t_{\text{next}});
}
```

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 \((\text{nil}, 0)\) if it is idle, and the node along with the amount of service time required (at rate \( r \)) to complete its service, which is \((\alpha.creq - \alpha.csrv)/r\). The second case is when we have an internal node \( \alpha \) with \( \alpha.sshares == 0 \). This indicates that all of \( \alpha \)'s descendants are idle, so we waste no further time searching beneath this node, and just return \((\text{nil}, 0)\). 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 \((\beta, t_{\text{next}})\) with the smallest \( t_{\text{next}} \) and \( \beta != \text{nil} \). This is the first node under \( \alpha \) to become idle, and hence the required return value. Note that if \( \alpha \) is being serviced at rate \( r \), then the child \( \beta \) is serviced at rate \( r*\beta.shares/\alpha.sshares \) 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 (t\text{last}, the time of the last simulated event) to the current time. This advance is broken down into steps. \texttt{predict} is called first to find the next change in activity (if any). As it is assumed that no arrivals occur between \texttt{t\text{last}} and \texttt{time}, the only changes can be departures. For each such change in activity found, \texttt{t\text{last}} is updated to that time using a further helper function \texttt{update1}, described below. This is repeated until no further changes in activity are found, at which point \texttt{t\text{last}} is updated one last time, to the current time. Here is the pseudocode:

```c
/*
  * 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 ("t\text{last}").
  * "r\text{max}" is the line rate.
  */
update1(root, time)
{
  for (;;) {
    (β, t\text{dep}) = predict(root, r\text{max});
    if (β == nil || t\text{last}+t\text{dep} > time)
      break;
    update1(root, r\text{max}, t\text{dep});
    t\text{last} += t\text{dep};
  }
  if (t\text{last} < time) {
    update1(root, r\text{max}, time-t\text{last});
    t\text{last} = time;
  }
}
```

Note that if the time returned by \texttt{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 \texttt{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. \texttt{update1} takes three arguments: a node \(\alpha\), a rate of service \(r\), and an amount of time \(\text{deltat}\). It is a recursive function, and is structured similarly to \texttt{predict}:
Given a node `$\alpha$`, a rate "$r$", and an amount of time "deltat", simulate the effect of "deltat" seconds of HPS service at rate "$r$", on "$\alpha$" and its descendants.

```c
update(\alpha, r, deltat)
{
    if (isleaf(\alpha)) {
        if (\alpha.creq > \alpha.csrv) {
            newsrv = \alpha.csrv + r*deltat;
            while (newsrv > \alpha.creq) {
                req = dequeue(\alpha.queue);
                tag = tlast + (\alpha.creq-\alpha.csrv)/r;
                enqueuepri(pri_g, req, tag);
                \alpha.creq += len(req);
            }
            if (newsrv >= \alpha.creq) {
                newsrv = \alpha.creq;
                inactive(\alpha);
            }
            \alpha.csrv = newsrv;
        }
    } else if (\alpha.sshares != 0) {
        r /= \alpha.sshares;
        for (\beta in \alpha.children) update(\beta, r*\beta.shares, deltat);
    }
}
```

The first case is when $\alpha$ is a leaf node. This is the most complex case, and its description will be left until last. Suppose now that $\alpha$ is an internal node. If $\alpha$ is idle ($\alpha.sshares == 0$) then there is nothing to do: all of $\alpha$'s descendants are idle, and their state does not change over the interval. If, however, $\alpha$ is active, then we must call `update` recursively on each of its children. Here we use the same device as in `predict`, whereby the appropriate value for $r$ is passed in the recursive call.

We now consider the case where $\alpha$ is a leaf. If this leaf is idle ($\alpha.creq == \alpha.csrv$) then there is nothing to do. Otherwise, we have to update $\alpha.csrv$ to its new value, $\alpha.csrv+r*deltat$, and handle any departures caused by this change. Here we use $\alpha.creq$ to check for individual requests that need to be dequeued. If we find any, then they are moved to the priority queue, stamped with their HPS starting time (which is now known). Then $\alpha.creq$ is updated, so that it always represents the cumulative request time up until the first request on $\alpha$'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 $\alpha$ 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 $\alpha.csrv$ to become larger than $\alpha.creq$.

4.5. Complexity of HFQ

We now examine the complexity of HFQ. In this section, we use $N$ to represent the number of leaf nodes (users), $M$ to represent the number of nodes in the class tree, and $D$ the maximal depth of the class tree.

Both `active` and `inactive` have a worst case cost of $O(D)$. Depending on the pattern of usage, the average cost may be better. For example, suppose that $\alpha$ usually has at least one active sibling; then neither of these functions ever needs to recurse, and the average cost is $O(1)$. In any case, $D$ is usually equivalent to a small constant.

`predict` has a worst case cost of $O(M)$, since it may have to visit every node in the tree (when every internal node is active). `update1` also (potentially) visits every node of the tree; however, there are additional costs associated with the processing of each leaf node which is reached. In particular, there are the calls to `enqueuepri` and `inactive`. We have seen that `inactive` has a worst case cost of $O(D)$, but a single execution of `update1` can inactivate each node of the tree at most once. Thus the extra cost introduced by the call to `inactive` is at most $O(M)$. Assuming that an efficient implementation of the priority queue is used (such as a heap), `enqueuepri` costs at most $O(\log N)$ to execute, since each user can have at most one request in the priority queue. Since `update1` may call `enqueuepri` once for each user (leaf node), the cost of this operation is $O(N\log N)$. Hence the `update1` function has an overall worst case cost of $O(M + N\log N)$. If, on average, `update1` moves one packet to the priority queue, then the cost is at most $O(M + \log N)$, which is $O(M)$, since $\log N << M$. This gives us an amortized cost of $O(M)$ per packet.

`update` may call `predict` and `update1` multiple times in a single invocation; however, each of these calls to `update1` moves at least one packet to the
priority queue. update may determine from the call to predict that no requests are ready to be moved, but update itself is called at most twice for every request. Thus there is an amortized cost of $O(M)$ per packet for update. new_request includes a call to active, with worst case cost $O(D)$, but $D < M$. both new_request and next_service call update. Hence the algorithm has an amortized cost of $O(M)$ per packet.
5. Analysis of Hierarchical Fair Queueing

In this chapter we present our analysis of the HFQ algorithm. Our main results are a set of bounds on the throughput and delay discrepancies between HFQ and HPS, when presented with identical arrival sequences. These results enable the guarantees provided by the HPS discipline to be converted into guarantees provided by the practical HFQ algorithm.

Our methodology is inspired by that in [15], albeit considerably generalized. The proofs in [15] are tied quite closely to the concept of “virtual time”, and are thus not applicable to algorithms such as HFQ. We have discovered generalizations of some of those results which apply to arbitrary service disciplines; in particular, that two disciplines which always start servicing requests in the same order satisfy certain “intermediate bounds”, which lead to delay and throughput discrepancy bounds under certain circumstances.

Our analysis not only generalizes on the results known for FQ by extending them to HFQ, but it also improves our knowledge about WFQ, and generalizes the results for FQ and WFQ such that servers providing variable rates of service are permissible. This is an important step along the way to analysing the effect of combining FQ-like algorithms to produce an HPS approximation. We hope that future researchers will be able to make use of these results.

5.1. Overview

This section presents an overview of the analysis, which will be developed over the remainder of the chapter. We start by defining the concept of Service Discipline Execution (SDE), a mathematical model of what happens during the execution of a service discipline. This includes functions describing arrival of requests and service of same. The concept of multiple users/request streams is built into the model. The model is designed to be equally applicable to practical service disciplines such as HFQ, and theoretical idealizations such as HPS. Arrivals are discrete; a request is not available until the whole request has arrived.

We define what it means for a service discipline to be work conserving in the context of servers which provide a variable rate of service. We then show how HPS and HFQ may be modelled using this notation, and demonstrate that HFQ is
consistent, work conserving and achieves the same ordering of service start times as HPS. This “ordering constraint” is the means by which we prove the discrepancy bounds. From it we are able to derive two intermediate bounds, and thence derive the delay and throughput discrepancy bounds.

We finish up this chapter by comparing the bounds for HFQ/HPS with known bounds for FQ/PS and WFQ/GPS, utilising the fact that these systems are degenerate cases of HFQ/HPS. The bounds obtained for HFQ/HPS are slightly inferior to those for FQ/PS. We note the asymmetry of the bounds, and the fact that, since HFQ is based on starting times, this asymmetry is oriented in the opposite direction to that of WFQ, which is based on finishing times.

5.2. Service Discipline Executions

We begin by introducing a mathematical model of the execution of a service discipline. This model needs to be sufficiently general to describe, on the one hand, idealized, fluid-flow service disciplines such as HPS, and on the other hand, practical, non-preemptive algorithms such as HFQ. Thus we must represent each user’s cumulative service as a continuous function and allow the possibility of multiple users being serviced simultaneously, as they are in HPS. We must also represent the discrete nature of a user’s requests as entities which may need to be serviced one at a time, as they are in HFQ.

The execution of a service discipline (SDE) is described by a number of symbols. Firstly, $N$ represents the number of user classes. $N$ is always finite. Greek letters $\alpha, \beta, \gamma$ are used as indices into the user classes, e.g., $\alpha = 1, \ldots, N$.

Within each user class, there is a sequence of “requests”. This sequence may be finite or infinite in number. We write $\text{maxreq}(\alpha)$ for the number of requests belonging to user class $\alpha$; if the sequence is infinite, we write $\text{maxreq}(\alpha) = \infty$.

The requests belonging to user class $\alpha$ are numbered from 1 to $\text{maxreq}(\alpha)$. We typically use subscripts $i, j, k$ to index into this set. We use $\text{req}(\alpha, i)$ to denote the $i$-th request of user class $\alpha$.

In our model, each request has a definite arrival time and a packet length, which is known at the time of arrival. We denote the arrival time of $\text{req}(\alpha, i)$ by $\tau_{\alpha, i}$, and the packet length by $P_{\alpha, i}$.
It is convenient to describe arrival and service processes by means of monotonic functions representing the cumulative arrivals/service over $[0,t)$. Accordingly, we set

$$
\rho_{\alpha,i}(t) = \begin{cases} 
  P_{\alpha,i} & \text{if } t \geq \tau_{\alpha,i} \\
  0 & \text{if } t < \tau_{\alpha,i}
\end{cases}
$$

$\rho_{\alpha,i}(t)$ is the amount of $\text{req}(\alpha,i)$ which is ready for service at time $t$. In our model, the change from none of the request being ready to all being ready is always a single discrete step, representing the property of many network gateways (including ours) that the entire request has to arrive before any of it can be serviced.

Similarly, we can look at the cumulative arrivals for a whole user class. We write

$$
\rho_{\alpha}(t) = \sum_{i=1}^{\max\text{req}(\alpha)} \rho_{\alpha,i}(t)
$$

$\rho_{\alpha}(t)$ is the cumulative arrival for user class $\alpha$ over $[0,t)$. It is a staircase function, with each step corresponding to one or more discrete arrivals. An example of the cumulative arrival function is provided below. Four arrivals are shown. The second and third arrivals are simultaneous ($\tau_{\alpha,2} = \tau_{\alpha,3}$).

![Cumulative Arrival Function](image)

Figure 5.2.1: A Cumulative Arrival Function

We can take this one step further, and define
\[ \rho(t) = \sum_{\alpha=1}^{N} \rho_\alpha(t) \]

\( \rho(t) \) is the total cumulative arrival function over all user classes on \([0, t]\). Like \( \rho_\alpha(t) \), \( \rho(t) \) is a staircase function, with steps occurring at the time of every arrival to the system.

Having described the arrival process of the execution, we next turn our attention to the service process. For each req(\( \alpha, i \)), we define a continuous, monotonic function \( \mu_{\alpha, i}(t) \), to be the amount of service received by req(\( \alpha, i \)) on \([0, t]\).

Note that the service process is assumed to be continuous, in direct contrast to the arrival process. This is clearly appropriate for idealized fluid flow service disciplines like HPS; for practical systems which process a request as an indivisible entity, which occupies the network interface exclusively during a time interval, the model can represent them by defining the amount of service received in the middle of the interval as a linear function of time. This function starts at 0, and ramps up until it reaches the full length of the request \( (P_{\alpha, i}) \), increasing at the rate of service provided by the server (typically assumed to be constant for the duration of each request, and lying somewhere on \([r_{\text{min}}, r_{\text{max}}]\), although we make no such constant rate assumptions for the fluid-flow discipline).

The diagram below shows a sample service function \( \mu_{\alpha, i}(t) \). A request arrives at time \( \tau_{\alpha, i} \), commences service some time later, and is serviced at three different rates until the full request has completed service, at which point \( \mu_{\alpha, i}(t) = P_{\alpha, i} \). Note that for each request, service can only be received for what has arrived so far. Thus \( \mu_{\alpha, i}(t) \leq \rho_{\alpha, i}(t) \), as is illustrated by the diagram. Further, by summation, \( \mu_\alpha(t) \leq \rho_\alpha(t) \) and \( \mu(t) \leq \rho(t) \) also. The expression \( \rho_\alpha(t) - \mu_\alpha(t) \) represents the amount of work queued up waiting for service for user \( \alpha \) at time \( t \) (the "backlog" for that user) and \( \rho(t) - \mu(t) \) represents the total system queue length/backlog. When \( \mu_\alpha(t) = \rho_\alpha(t) \), user \( \alpha \) has no outstanding requests to be serviced, and we say that \( \alpha \) is idle. When \( \rho(t) = \mu(t) \), the system has no work to do, and is said to be idle.
Figure 5.2.2: A Cumulative Service Function

By analogy with the arrival process, we construct these functions:

\[ \mu_\alpha(t) = \sum_{i=1}^{\max\text{req}(\alpha)} \mu_{\alpha,i}(t) \]

\[ \mu(t) = \sum_{\alpha=1}^{N} \mu_\alpha(t) \]

\( \mu_\alpha(t) \) is the cumulative service for user class \( \alpha \) on \([0,t)\) and \( \mu(t) \) is that for the whole system.

We are interested in the times at which particular requests start and finish service. We write

\[ \sigma_{\alpha,i} = \sup \{ t \mid \mu_{\alpha,i}(t) = 0 \} \]

\[ \phi_{\alpha,i} = \inf \{ t \mid \mu_{\alpha,i}(t) = P_{\alpha,i} \} \]

\( \sigma_{\alpha,i} \) is the time at which \( \text{req}(\alpha,i) \) starts service, and \( \phi_{\alpha,i} \) is the time at which it finish service. These times are illustrated on the diagram of our example service function above. Note the general pattern of arrival, followed by start of service, followed by service completion; that is, \( \tau_{\alpha,i} \leq \sigma_{\alpha,i} < \phi_{\alpha,i} \).

Now we have all of the symbols required to describe a service discipline execution. We write this formally as

\[ D = (N, \max\text{req}, \tau_{\alpha,i}, P_{\alpha,i}, \rho_{\alpha,i}, \mu_{\alpha,i}, \sigma_{\alpha,i}, \phi_{\alpha,i}) \]
5.3. Comparing Two Service Disciplines

We will see shortly how HPS and HFQ can be described by this formal model. Our goal is to compare the behaviour of HPS and HFQ when confronted with the same arrival process; i.e.:

$$HFP = (N, \maxreq, \tau_{\alpha,i}, P_{\alpha,i}, \rho_{\alpha,i}, \mu_{\alpha,i}, \sigma_{\alpha,i}, \phi_{\alpha,i})$$

$$HPS = (N, \maxreq, \tau_{\alpha,i}, P_{\alpha,i}, \rho_{\alpha,i}, \mu'_{\alpha,i}, \sigma'_{\alpha,i}, \phi'_{\alpha,i})$$

The property of HFQ which interests us is how well it approximates HPS in terms of throughput and delay characteristics. Accordingly, we define:

$$T^{+}_{\alpha} = \sup \{ \mu'_{\alpha}(t) - \mu_{\alpha}(t) \} , \quad T^{-}_{\alpha} = \sup \{ \mu_{\alpha}(t) - \mu'_{\alpha}(t) \}$$

$$L^{\sigma^{-}_{\alpha}} = \sup \{ \sigma'_{\alpha,i} - \sigma_{\alpha,i} \} , \quad L^{\sigma^{+}_{\alpha}} = \sup \{ \sigma_{\alpha,i} - \sigma'_{\alpha,i} \}$$

$$L^{\phi^{-}_{\alpha}} = \sup \{ \phi'_{\alpha,i} - \phi_{\alpha,i} \} , \quad L^{\phi^{+}_{\alpha}} = \sup \{ \phi_{\alpha,i} - \phi'_{\alpha,i} \}$$

So that

$$\forall \alpha,i: T^{-}_{\alpha} \leq \mu'_{\alpha}(t) - \mu_{\alpha}(t) \leq T^{+}_{\alpha}$$

$$\forall \alpha,i: -L^{\sigma^{-}_{\alpha}} \leq \sigma'_{\alpha,i} - \sigma_{\alpha,i} \leq L^{\sigma^{+}_{\alpha}}$$

$$\forall \alpha,i: -L^{\phi^{-}_{\alpha}} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq L^{\phi^{+}_{\alpha}}$$

We may well ask whether such bounds exist. For an arbitrary pair of service discipline executions, the answer is clearly no. But for particular instances of HFQ and HPS, with bounded request lengths and satisfying our generalization of work conservation (see below), the answer is yes.

Before we go any further with the theoretical concerns, we take a look at the practical significance of these bounds. Note that $\mu_{\alpha}(t)$ and $\mu'_{\alpha}(t)$ are the cumulative throughputs of user $\alpha$ at time $t$, in HFQ and HPS. Thus the first bound tells us how well HFQ approximates HPS in terms of bandwidth allocation.

The other 2 bounds are concerned with delay approximation. $\sigma'_{\alpha,i}$ and $\sigma_{\alpha,i}$ are the starting times of req($\alpha,i$) in HFQ and HPS. Thus the second bound tells us how well HFQ approximates HPS in terms of starting times. Similarly, the third bound
tells us the analogous bound for finishing times, \( \phi_{\alpha,i} \) and \( \phi_{\alpha,i} \). (The third bound, which deals with finishing times, is of greater practical use than the second bound, which deals with starting times, since a request has to complete service before it is considered to have arrived at the next node in the network. Other analyses generally only provide a bound for finishing times, but we will provide both, since the starting time bound arises naturally from our methodology).

Some real world constraints are required in order to prove that bounds do exist. Firstly, we require that request lengths are bounded:

\[
P_{\alpha,i} \leq P_{\alpha}^{\text{max}} \leq P^{\text{max}}
\]

This bound is true of all real-life networking systems. Indeed, if it were not true, then arbitrarily bad "unfairness" would be possible—for example a single user could send a packet so large that all other users are deprived of service.

We have a global packet length maximum, \( P^{\text{max}} \). We also define per-user packet length maxima, \( P_{\alpha}^{\text{max}} \). In cases where tighter bounds are known for some users than for others, better bounds on throughput and delay characteristics may be found.

The second constraint concerns the rate of service provided to the two service disciplines. This must be the same, otherwise divergent behaviour can be expected. Our analysis allows the rate of service to vary over time, as long the instantaneous values are the same for both disciplines being compared, and there is a minimum rate of service. The existence of a minimum rate of service is important for proving the delay bounds.

5.4. Rate of Service and Work Conservation

An important property of HPS and HFQ is that they are work conserving. This means that they are never idle when there are requests waiting to be serviced; rather, the full capacity of the resource is used whenever possible.

As mentioned in the last section, we do not assume that the resource provides a constant rate of service. Instead, we suppose that the rate of service is described by a function \( r(t) \), giving the instantaneous available bandwidth at time \( t \). We assume that this function is bounded, i.e., that there are positive constants \( r_{\text{min}} \) and \( r_{\text{max}} \) such
that for all $t$,

$$r_{\text{min}} \leq r(t) \leq r_{\text{max}}$$

Now work conservation of HPS says that if there is work available, it is serviced at the maximum available rate; i.e., if $\mu'(t) < \rho(t)$, then $\frac{d}{dt} \mu'(t) = r(t)$. Clearly, when $\mu'(t) = \rho(t)$ there is no work to do, and $\frac{d}{dt} \mu'(t) = 0$. Work conservation for HPS may thus be summarized as

$$\frac{d}{dt} \mu'(t) = \begin{cases} r(t) & \mu'(t) < \rho(t) \\ 0 & \mu'(t) = \rho(t) \end{cases}$$

Similarly, as HFQ is work conserving,

$$\frac{d}{dt} \mu(t) = \begin{cases} r(t) & \mu(t) < \rho(t) \\ 0 & \mu(t) = \rho(t) \end{cases}$$

from which we conclude that $\mu'(t) = \mu(t)$, an identity that we will use later on in our proofs of the bounds. Note that the work conservation of HPS is part of the definition of HPS, whereas we have to prove this property for HFQ.

The existence of $r_{\text{min}}$, when coupled with the work conservation property, leads to another useful property. If there is work available throughout an interval $[t_1, t_2)$, then

$$\mu(t_2) - \mu(t_1) \geq r_{\text{min}} (t_2 - t_1)$$

since by work conservation $\frac{d}{dt} \mu(t) = r(t) \geq r_{\text{min}}$ throughout $[t_1, t_2)$.

Of course, if our resource has a constant rate of service $r$, then $r_{\text{min}} = r_{\text{max}} = r$.

The assumption of constant rate is sufficient to cover the majority of practical systems, including the implementation described in this thesis, so some words of explanation are required as to why we consider the generalization worth making. It turns out that the generalization does not increase the complexity of the mathematical analysis, so that we may as well solve the more general problem. There are examples of practical systems where the constant rate assumption does not hold.
For example, consider a collection of parallel network links, which are to be treated as a single logical resource. The instantaneous capacity of such a resource varies depending on the number of outstanding requests: since a request can not be split between two links, if the number of outstanding requests is less than the number of links then the full capacity of the system is not used. Here the instantaneous service rate function, $r(t)$, is dependent on the arrival process.

Another possible example is furnished by PPP, which uses HDLC encapsulation. Depending on the number of characters in the request packet which need to be escaped, the apparent service rate can be reduced by up to 50%. We could treat this as a variable rate server, but it is probably better to count the escape characters as part of the request itself and include them in the request length (this is the approach used in our implementation).

An even more compelling reason for introducing the concept of a variable rate server is that it allows us to decompose a given instance of HPS into smaller HPS systems, and ultimately into GPS servers. In order to make such a decomposition possible, we require our model of HPS to include the possibility of variable rates of service because such a phenomenon occurs at the internal nodes of an HPS tree, even though the system as a whole receives a constant rate of service. We will make no explicit use of this decomposition principle in this work. We believe that it may prove to be a useful tool in future work, for studying the behaviour of systems which attempt to approximate HPS by nested instances of an FQ-like algorithm. Such systems have recently been proposed [40], [3], [35].

5.5. The HPS Service Discipline

We now apply the mathematical model which we have built up to understanding the HPS service discipline. We start by defining HPS formally. Then we show that HPS provides throughput and delay guarantees (for suitably well-behaved sessions). Finally, we describe how HPS may be simulated.
5.5.1. Describing HPS Formally

The HPS service discipline is based on the concept of a “class tree”. A class tree is a tree where each node is labelled with a number of shares. The leaves of the class tree represent actual users ($N$ in number). The internal nodes represent groups of users. We extend the numbering $\alpha = 1, \ldots, N$ of users to the internal nodes, which we number $\alpha = N + 1, \ldots, M$, where $M$ is the root node. Each node $\alpha$ is assigned a number of shares $w_\alpha$. For example

![Class Tree Diagram]

Figure 5.5.1.1: Example HPS Class Tree

We define $C_\alpha$ to be the set of $\alpha$’s children:

$$C_\alpha = \{ \beta \mid \beta \text{ is a child of } \alpha \}$$

For example, in the diagram above, $C_6 = \{3, 4\}$ and $C_2 = \{\}$. 

HPS is defined by recursive application of GPS (PS with shares) to each internal node. That is, starting at the top of the class tree, we recursively split up the available bandwidth amongst the active children according to shares. “Active” here means that unfinished work is available either at the node (if it is a leaf) or at a descendant of the node (for internal nodes).

More formally, we define

$$\alpha = 1, \ldots, N: \quad A_\alpha(t) = \begin{cases} 1 & \rho_\alpha(t) > \mu'_\alpha(t) \\ 0 & \rho_\alpha(t) = \mu'_\alpha(t) \end{cases}$$

$$\alpha = N + 1, \ldots, M: \quad A_\alpha(t) = \begin{cases} 1 & \exists \beta \in C_\alpha: A_\beta(t) = 1 \\ 0 & \text{otherwise} \end{cases}$$
Note the recursive nature of the second half of this definition; an internal node is defined to be active if any of its children are active. Repeated application of the recursive case to an active internal node must eventually yield an active leaf node as a descendant. Thus, for an internal node \( \alpha \), \( A_\alpha(t) = 1 \) precisely when some leaf descendant \( \gamma \) of \( \alpha \) has \( A_\gamma(t) = 1 \), and \( A_\alpha(t) = 0 \) otherwise.

We also extend the definition of cumulative throughput to internal nodes, defining \( \mu'_\alpha(t) \) to be the sum of the cumulative throughputs of all of the leaf-node descendants of \( \alpha \). A recursive definition is furnished by

\[
\alpha = N + 1, \ldots, M: \quad \mu'_\alpha(t) = \sum_{\beta \in C_\alpha} \mu'_\beta(t)
\]

Note that since user \( M \) is the root node of the class tree, it follows that \( \mu'_M(t) = \mu'(t) \).

Now we describe the division of bandwidth amongst the competing users by the following equations:

\[
\forall \alpha, \forall \beta \in C_\alpha: \quad \frac{d}{dt} \mu'_\beta(t) = \begin{cases} 
\frac{w_\beta}{\sum_{\gamma \in C_\alpha} w_\gamma A_\gamma(t)} \frac{d}{dt} \mu'_\alpha(t) & A_\beta(t) = 1 \\
0 & A_\beta(t) = 0
\end{cases}
\]

We are careful to avoid division by zero—in the case where \( A_\alpha(t) = 0 \), then by definition \( A_\gamma(t) = 0 \) for each \( \gamma \in C_\alpha \), and the denominator is undefined. But this possibility is excluded by the requirement that \( A_\beta(t) = 1 \). Note that this formula is consistent with the definition of cumulative service function for internal nodes by summation of their children’s cumulative service, as given above.

HPS is defined by the above set of equations, plus the requirement that HPS be work conserving, i.e., given the instantaneous service rate function \( r(t) \),

\[
\frac{d}{dt} \mu'_M(t) = \begin{cases} 
\mu'(t) < \rho(t) \\
0 & \mu'(t) = \rho(t)
\end{cases}
\]

Note that the division of bandwidth amongst the actual users is determined entirely by the activity states of the users. That is, there exist \( N \) functions of \( N \) "boolean" variables, which we write as \( E_\alpha(a_1, \ldots, a_N) \), such that
\[ \alpha = 1, \ldots, N: \quad \frac{d}{dt} \mu'_\alpha(t) = E_\alpha(A_1(t), \ldots, A_N(t)) \frac{d}{dt} \mu'(t) \]

For example, let us calculate \( \frac{d}{dt} \mu'_1(t) \) for the sample class tree in Figure 5.5.1.1.

When \( A_1(t) = 0 \), then \( \frac{d}{dt} \mu'_1(t) = 0 \). When \( A_1(t) = 1 \), then \( A_5(t) = 1 \), and

\[ \frac{d}{dt} \mu'_5(t) = \frac{w_5}{w_5A_5(t) + w_6A_6(t)} \frac{d}{dt} \mu'_7(t) \]

thus,

\[ \frac{d}{dt} \mu'_1(t) = \frac{w_1}{w_1A_1(t) + w_2A_2(t)} \frac{w_5}{w_5A_5(t) + w_6A_6(t)} \frac{d}{dt} \mu'_7(t) \]

(note that we have left \( A_1(t) \) and \( A_5(t) \) in the denominators to preserve the symmetry; they are both known to be 1 under the current assumption). This gives us a formula for \( E_1 \):

\[ E_1(a_1, a_2, a_3, a_4) = \begin{cases} \frac{w_1}{w_1 + w_2a_2} & \frac{w_5}{w_5 + w_6(a_3 \lor a_4)} \quad a_1 = 1 \\ 0 & a_1 = 0 \end{cases} \]

Similar formulae can be derived for the other users. The principle is the same for arbitrary class trees—one applies the formula for dividing a node’s bandwidth amongst its children recursively from the root node down through the chain of ancestors of the node in question, until the node is reached. Let us define \( \alpha^{(i)} \) to be the \( i \)-th ancestor of \( \alpha \), such that \( \alpha^{(0)} = \alpha \). Furthermore, let \( d_\alpha \) be the depth of \( \alpha \), whence \( \alpha^{(d_\alpha)} = \emptyset \). Then we can write:

\[ E_\alpha(A_1(t), \ldots, A_N(t)) = \begin{cases} \prod_{j=1}^{d_\alpha} \frac{w_{\alpha^{(j)}}}{\sum_{\beta \in C_{\alpha^{(j)}}} w_\beta A_\beta(t)} \quad A_\alpha(t) = 1 \\ 0 \quad A_\alpha(t) = 0 \end{cases} \]
5.5.2. Throughput Guarantees for HPS

We observed in Chapter 3 that HPS provides throughput (and delay) guarantees. We will now express these formally. Assume that $A_\alpha(t) = 1$. Then

$$E_\alpha(A_1(t), \ldots, A_N(t)) = \prod_{j=1}^{d_\alpha} \frac{w_{\alpha^{(j-1)}}}{\sum_{\beta \in C_{\alpha^{(j)}}} w_\beta A_\beta(t)}$$

$$\geq \prod_{j=1}^{d_\alpha} \frac{w_{\alpha^{(j-1)}}}{\sum_{\beta \in C_{\alpha^{(j)}}} w_\beta}$$

(which is in fact $E_\alpha(1, \ldots, 1)$). Hence we have the following minimum guaranteed rate of service for user $\alpha$, when $\alpha$ is backlogged:

$$\frac{d}{dt} \mu_\alpha'(t) \geq r_{\min} \prod_{j=1}^{d_\alpha} \frac{w_{\alpha^{(j-1)}}}{\sum_{\beta \in C_{\alpha^{(j)}}} w_\beta}$$

While we are looking at throughput guarantees, we should note that there is a maximum rate of service:

$$\frac{d}{dt} \mu_\alpha'(t) \leq \frac{d}{dt} \mu_M(t)$$

$$\leq r_{\max}$$

Thus $\frac{d}{dt} \mu_\alpha'(t)$ has a maximum as well as a minimum (for $\alpha$ active), just as the total service rate $r(t)$ does. We can use this fact to nest HPS disciplines, treating any given user of one HPS discipline as a virtual resource to be shared by another HPS discipline. The formulas for $\mu_\alpha'(t)$ and $E_\alpha(A_1(t), \ldots, A_N(t))$ clearly compose correctly under such a circumstance, to give us a large HPS system out of two smaller ones—we don’t get anything new by nesting HPS disciplines, whereas we get HPS by nesting GPS.
5.5.3. Delay Guarantees for HPS and Leaky Bucket Arrivals

We now present a delay bound for HPS with Leaky Bucket constrained arrivals, based on the Guaranteed Rate Clock framework [14]. To simplify the exposition, we only look at the delay caused by a single node, ignoring inter-node propagation delays. The end-to-end delay bound for multiple nodes is much better than that obtained by adding the worst case delay provided by each node; the interested reader is referred to the paper by Goyal et al. [14].

First we must define the Leaky Bucket constraint. We say that \( \rho_{\alpha}(t) \sim (q_{\alpha}, r_{\alpha}) \) if for all \( t_1, t_2 \) satisfying \( t_1 < t_2 \)

\[
\rho_{\alpha}(t_2) - \rho_{\alpha}(t_1) \leq q_{\alpha} + r_{\alpha}(t_2 - t_1)
\]

This is the Leaky Bucket constraint with average rate \( r_{\alpha} \) and burstiness \( q_{\alpha} \). For the purpose of using the GRC framework, the average rate must be the same as the guaranteed rate provided by HPS, hence

\[
r_{\alpha} = r_{\min} L_{\alpha}(1, \ldots, 1)
\]

The Guaranteed Rate Clock \( GRC_{\alpha,i} \) is associated with each request as follows

\[
GRC_{\alpha,0} = 0
\]

\[
GRC_{\alpha,i} = \max(\tau_{\alpha,i}, GRC_{\alpha,i-1}) + \frac{P_{\alpha,i}}{r_{\alpha}}
\]

A service discipline is in the class GR if there is some constant \( C \) such that each request \( \text{req}(\alpha, i) \) finishes by time \( GRC_{\alpha,i} + C \). For example, HPS is in class GR if a \( C \) exists such that

\[
\phi'_{\alpha,i} \leq GRC_{\alpha,i} + C
\]

We show that HPS is in class GR with \( C = 0 \). To see this, note first that since HPS guarantees user \( \alpha \) a minimum rate of \( r_{\alpha} \) when active, and since user \( \alpha \) is active on \( [\sigma'_{\alpha,i}, \phi'_{\alpha,i}) \),

\[
\mu'_{\alpha}(\phi'_{\alpha,i}) - \mu'_{\alpha}(\sigma'_{\alpha,i}) \geq r_{\alpha}(\phi'_{\alpha,i} - \sigma'_{\alpha,i})
\]
Noting that $\mu'_\alpha(\phi'_{\alpha,i}) - \mu'_\alpha(\sigma'_{\alpha,i}) = P_{\alpha,i}$,

$$\phi'_{\alpha,i} - \sigma'_{\alpha,i} \leq \frac{P_{\alpha,i}}{r_{\alpha}}$$

Now, since $\sigma'_{\alpha,i} = \max(\tau_{\alpha,i}, \phi'_{\alpha,i-1})$,

$$\phi'_{\alpha,i} \leq \max(\tau_{\alpha,i}, \phi'_{\alpha,i-1}) + \frac{P_{\alpha,i}}{r_{\alpha}}$$

From this it is easy to show by induction that

$$\phi'_{\alpha,i} \leq GRC_{\alpha,i}$$

The significance of one of the delay discrepancy bounds defined in section 5.3 may now be seen. Suppose that HFQ and HPS satisfy the delay discrepancy bounds. Then

$$\phi_{\alpha,i} \leq GRC_{\alpha,i} + (\phi_{\alpha,i} - \phi'_{\alpha,i})$$

$$\leq GRC_{\alpha,i} + L_\alpha^\phi$$

This gives us a characterization of HFQ as a member of the GR class. The analysis of Goyal et al. [14] gives us the following delay bound for the single node case

$$\phi_{\alpha,i} - \tau_{\alpha,i} \leq \frac{q_{\alpha}}{r_{\alpha}} + L_\alpha^\phi$$

We will derive a value for $L_\alpha^\phi$ later in this chapter.

### 5.5.4. Simulating HPS

We next turn our attention to the question of how HPS may be simulated in real time. We will assume for this section that the resource provides a constant rate of service ($r_{\text{min}} = r_{\text{max}} = r$). We see that under HPS, each cumulative throughput function $\mu'_\alpha(t)$ is piecewise linear in $t$. The changes in slope are restricted to times at which the activity state ($A_{\alpha}(t)$) of one or more users changes; between two such times, each $\mu'_\alpha(t)$ is a linear function.

Thus an HPS simulation consists of a series of events, each of which
corresponds to an activity state change. When more than one user changes activity state at the same time, the changes may be treated as separate events occurring at the same time.

There are two types of events: ‘‘arrivals’’ and ‘‘completions’’. An arrival is when a user becomes active, in response to an actual arrival; it is the transition \( A_\alpha(t) = 0 \rightarrow A_\alpha(t) = 1 \). These only occur when \( t = \tau_{\alpha,i} \), for some \( i \). (Note that not all actual arrivals show up as transitions in the activity state, since the user may already be active when an arrival occurs).

A completion is a transition \( A_\alpha(t) = 1 \rightarrow A_\alpha(t) = 0 \), which corresponds to some service completion \( t = \phi_{\alpha,i} \). Again, not all actual service completions cause activity state transitions.

To simulate HPS, we need to keep a certain amount of state information. This includes the current time of the simulation, the activity state of each user in the simulation and the cumulative request and service amounts for each user. The current time will be updated on every arrival to the system, and also at every real time that we need to know the state of the system (generally whenever a request finishes service in the HFQ discipline that we use the HPS simulation to drive). Whenever we update the current time, we update the state of the whole system. Because each arrival triggers an update, there are no arrivals in between updates. Thus the only activity state changes which can occur between updates are service completions. Thus the main task of an update operation is to check for intervening service completions. The way to do this is to search for the first intervening completion, then update the system to the time at which it occurred. This is repeated until there are no more intervening service completions.

Suppose the current time is \( t_0 \), and we are trying to update it to time \( t_1 \). Assume that there are no arrivals on \([t_0,t_1)\). If user \( \alpha \) is active at time \( t_0 \) \( (A_\alpha(t_0) = 1) \), then \( \mu_\alpha(t) \) increases at a rate of

\[
\frac{d}{dt} \mu_\alpha(t) = E_\alpha(A_1(t_0), \cdots, A_N(t_0))r
\]

until it reaches \( \rho_\alpha(t_0) \) (or until \( t = t_1 \)). If user \( \alpha \) is the first to complete within the interval, then this completion occurs at time
\[ t = t_0 + \frac{p_\alpha(t_0) - \mu_\alpha(t_0)}{E_\alpha(A_1(t_0), \ldots, A_N(t_0)) r} \]

We find the first completion by calculating this value for each user, and taking the minimum. If the minimum does not lie on the interval \([t_0, t_1]\), then there are no service completions on the interval. Otherwise, we update the system to the time of the first service completion, adjusting each \(\mu_\beta(t)\) according to the linear increase formula:

\[ \mu_\beta(t) = \mu_\beta(t_0) + (t - t_0) E_\beta(A_1(t_0), \ldots, A_N(t_0)) r \]

Then we repeat the process, until there are no more service completions to process, updating all users one more time to \(t = t_1\).

An important consideration is the calculation of the functions \(E_\alpha(A_1(t), \ldots, A_N(t))\). If \(N\) is small, then we can store the functions in a lookup table. When \(N\) becomes large, this become impractical. Another approach is to keep track of the activity state of all of the nodes, not just the leaf nodes; \(E_\alpha(A_1(t), \ldots, A_N(t))\) can then be calculated by recursing over the class tree. In the practical version of the HFQ algorithm described in Chapter 4, we use this method with a few shortcuts. The sum of the shares of the active children of each node is kept track of, enabling efficient calculation of \(E_\alpha(A_1(t), \ldots, A_N(t))\). The recursive calculation of the function is combined with the search for the first service completion. See Chapter 4 for further details.

### 5.6. The HFQ Service Discipline

We now turn our attention to the HFQ service discipline. We start by defining HFQ formally as a packet-based algorithm driven by a simulation of HPS. Having defined HFQ, we then prove that the service discipline is work conserving. Next, we show that HFQ and HPS achieve the same order of service starting times (a crucial result for the proof of the bounds in section 5.7). Finally, we show that the algorithm presented in chapter 4 is equivalent to the theoretical model of HFQ given here.
5.6.1. Describing HFQ Formally

HFQ is a non-preemptive service discipline—it services requests one at a time, finishing each request before starting another. Once a request is chosen to be serviced, it is serviced at the maximum possible rate until it is finished; formally, for $t \in (\sigma_{\alpha,i}, \phi_{\alpha,i})$, \( \frac{d}{dt} u_{\alpha}(t) = r(t) \).

HFQ uses a simulation of HPS to order requests. As we have seen, the starting time $\sigma'_{\alpha,i}$ for a request in HPS is known at time $t = \sigma'_{\alpha,i}$, but generally not before then. (Recall from section 5.3 that the primed quantities refer to HPS, and the unprimed ones refer to HFQ). HFQ divides the queue of pending requests into two separate queues: a queue for requests which have started in HPS, and a queue for those which have not. We represent the contents of these queues at time $t$ by $K(t)$ and $U(t)$ respectively. That is, $K(t)$ is the set of pending requests (in HFQ) which have started in HPS, and $U(t)$ is the set of pending requests which have not started in HPS. Formally:

$$K(t) = \{ \text{req}(\alpha,i) \mid t \geq \tau_{\alpha,i} \land t < \sigma_{\alpha,i} \land t \geq \sigma'_{\alpha,i} \}$$

$$U(t) = \{ \text{req}(\alpha,i) \mid t \geq \tau_{\alpha,i} \land t < \sigma_{\alpha,i} \land t < \sigma'_{\alpha,i} \}$$

The reason for this segregation of requests is that HFQ attempts to service requests in order of starting time in HPS; ordering cannot take place until the starting time is known, which is ensured by restricting attention to members of $K(t)$.

HFQ schedules requests to start service under two conditions: when a prior request completes service ($t = \phi_{\alpha,i}$) and when a new request arrives while the system is idle ($t = \tau_{\alpha,i}$). In the latter case, there is only one request to choose from—the newly arrived packet. Since HFQ was idle just prior to its arrival, HPS must have been too, since HPS always services requests at the maximum possible rate. Hence the new request starts service in HPS, and its starting time in HPS is known.

In the case where a prior request has completed ($t = \phi_{\alpha,i}$) HFQ chooses from $K(t)$ a request $\text{req}(\beta,j)$ so as to minimize $\sigma'_{\beta,j}$. If $K(t)$ is empty, HFQ waits until it is non-empty (we will see that $K(\phi_{\alpha,i})$ empty implies that the system is idle, thus in this case HFQ actually waits for the next arrival).
5.6.2. Proof that HFQ is Work Conserving

We will now show that \( K(\phi_{\alpha,i}) \) empty implies that \( U(\phi_{\alpha,i}) \) is empty also, which suffices to prove that HFQ is work conserving. We suppose that \( K(\phi_{\alpha,i}) \) is empty, but that \( U(\phi_{\alpha,i}) \) is not, and derive a contradiction.

Since we are assuming \( K(\phi_{\alpha,i}) \) to be empty, any request \( req(\beta,j) \) satisfying \( \tau_{\beta,j} \leq \phi_{\alpha,i} \) and \( \sigma_{\beta,j} > \phi_{\alpha,i} \) must satisfy \( \sigma_{\beta,j} > \phi_{\alpha,i} \) (otherwise we have a member of \( K(\phi_{\alpha,i}) \), according to the definition of \( K(\phi_{\alpha,i}) \)). Now for any request \( req(\beta,j) \) satisfying \( \sigma_{\beta,j} \leq \phi_{\alpha,i} \), it follows (since \( \tau_{\beta,j} \leq \sigma_{\beta,j} \)) that \( \tau_{\beta,j} \leq \phi_{\alpha,i} \), and since \( K(\phi_{\alpha,i}) \) is empty, that \( \sigma_{\beta,j} \leq \phi_{\alpha,i} \). Thus we have shown that

\[
\tau_{\beta,j} \leq \phi_{\alpha,i} \rightarrow \sigma_{\beta,j} \leq \phi_{\alpha,i}
\]

Now since HFQ is non-preemptive, any request starting service in HFQ before \( \phi_{\alpha,i} \) must finish by \( \phi_{\alpha,i} \), thus

\[
\sigma_{\beta,j} \leq \phi_{\alpha,i} \rightarrow \tau_{\beta,j} \leq \phi_{\alpha,i}
\]

Putting these two statements together, we have

\[
\tau_{\beta,j} \leq \phi_{\alpha,i} \rightarrow \phi_{\beta,j} \leq \phi_{\alpha,i}
\]

Now we observe that

\[
\mu'(\phi_{\alpha,i}) \leq \sum_{\sigma_{\beta,j} \leq \phi_{\alpha,i}} P_{\beta,j}
\]

by the definition of \( \mu'(t) \). And by the implications derived above,

\[
\sum_{\sigma_{\beta,j} \leq \phi_{\alpha,i}} P_{\beta,j} \leq \sum_{\phi_{\beta,j} \leq \phi_{\alpha,i}} P_{\beta,j}
\]

But from the definition of \( \mu(t) \) and the fact that HFQ is non-preemptive,

\[
\sum_{\phi_{\beta,j} \leq \phi_{\alpha,i}} P_{\beta,j} = \mu(\phi_{\alpha,i})
\]

Putting all this together,

\[
\mu'(\phi_{\alpha,i}) \leq \mu(\phi_{\alpha,i})
\]
But since HPS is work conserving, we have that \( \mu'(\phi_{\alpha,i}) \geq \mu(\phi_{\alpha,i}) \); thus it must be the case that \( \mu'(\phi_{\alpha,i}) = \mu(\phi_{\alpha,i}) \), and the inequalities above are equalities. Hence

\[
\mu'(\phi_{\alpha,i}) = \sum_{\beta,j} P_{\beta,j}
\]

which means that the requests which have started in HPS by \( \phi_{\alpha,i} \) have also finished, thus

\[
\sigma'_{\beta,j} \leq \phi_{\alpha,i} \Rightarrow \phi'_{\beta,j} \leq \phi_{\alpha,i}
\]

The contraposition is

\[
\phi'_{\beta,j} > \phi_{\alpha,i} \Rightarrow \sigma'_{\beta,j} > \phi_{\alpha,i}
\]

We now derive a contradiction from the assumption that \( U(\phi_{\alpha,i}) \) is non-empty. Let \( \text{req}(\beta,j) \) be the member of \( U(\phi_{\alpha,i}) \) with the smallest value of \( j \). Since HPS always provides service to active users, the following identity holds:

\[
\sigma'_{\beta,j} = \max(\tau_{\beta,j}, \phi'_{\beta,j-1})
\]

Now, since \( \text{req}(\beta,j) \in U(\phi_{\alpha,i}) \), \( \sigma'_{\beta,j} > \phi_{\alpha,i} \) and \( \tau_{\beta,j} \leq \phi_{\alpha,i} \), by definition of \( U(t) \). Substituting the identity into the first of these inequalities gives

\[
\max(\tau_{\beta,j}, \phi'_{\beta,j-1}) > \phi_{\alpha,i}
\]

but since \( \tau_{\beta,j} \leq \phi_{\alpha,i} \),

\[
\phi'_{\beta,j-1} > \phi_{\alpha,i}
\]

But this implies that \( \sigma'_{\beta,j-1} > \phi_{\alpha,i} \), by the result we have just derived from the emptiness of \( K(\phi_{\alpha,i}) \). Since \( \text{req}(\beta,j-1) \) has not started in HPS, it cannot have started in HFSQ either, and hence must be a member of \( U(\phi_{\alpha,i}) \), contradicting the choice of \( \text{req}(\beta,j) \).

Thus we have shown that as long as there are requests to service, HFSQ will always be able to find the next request (in \( K(t) \)), and is thus work conserving.
5.6.3. Proof that HFQ and HPS Satisfy Ordering Constraint

We will now show that HFQ and HPS start serving requests in the same order, that is,

\[ \sigma'_{\alpha,i} > \sigma'_{\beta,j} \Rightarrow \sigma_{\alpha,i} \geq \sigma_{\beta,j} \]

Suppose to the contrary, that there are requests \text{req(}\alpha,i\text{)} and \text{req(}\beta,j\text{)} such that \( \sigma'_{\alpha,i} > \sigma'_{\beta,j} \) but \( \sigma_{\alpha,i} < \sigma_{\beta,j} \). Consider the behaviour of HFQ at time \( \sigma_{\alpha,i} \). Since at this time it chose \text{req(}\alpha,i\text{)} to be serviced, it must be that the request has started service in HPS, which is expressed formally as \( \sigma'_{\alpha,i} \leq \sigma_{\alpha,i} \). Since we are assuming that \( \sigma'_{\alpha,i} > \sigma'_{\beta,j} \), it follows that \( \sigma'_{\beta,j} < \sigma_{\alpha,i} \), and thus that \text{req(}\beta,j\text{)} has arrived by time \( \sigma_{\alpha,i} \). But it has not started in HFQ, since \( \sigma_{\alpha,i} < \sigma_{\beta,j} \) (our assumption). Hence \text{req(}\beta,j\text{)} is a pending request, in fact a member of \( K(\sigma_{\alpha,i}) \) since \( \sigma'_{\beta,j} < \sigma_{\alpha,i} \). Then HFQ should have chosen \text{req(}\beta,j\text{)} for service at time \( \sigma_{\alpha,i} \), instead of \text{req(}\alpha,i\text{)}, since \( \sigma'_{\alpha,i} > \sigma'_{\beta,j} \). This contradiction proves that HFQ and HPS always start serving requests in the same order.

5.7. Proof of the Bounds

In this section we derive the main results of the chapter. We take the ordering property established in the previous section and use it to prove two intermediate bounds. We then use these results to prove the bounds on throughput and delay discrepancy, which are expressed in terms of the values which appear in the intermediate bounds.

These proof techniques are quite general; the first intermediate bound applies to any pair of SDEs which are linked by the constraint on the ordering of starting times (which can actually be generalized to allow limited reorderings), and the second intermediate bound follows from the first as long as both SDEs satisfy a bound on the relative rates of service, which is true for all non-preemptive disciplines and a broad class of PS-like disciplines. Given these two intermediate bounds (which could potentially be derived in other ways for other types of SDEs), the delay and throughput bounds follow without making further assumptions.
5.7.1. Intermediate Bound 1

Our proof of the delay and throughput efficiency bounds for HFQ and HPS is based on the fact that both disciplines produce start-of-service times with the same ordering, as proved above and expressed formally as:

\[ \forall \alpha, \beta, i, j: \sigma'_{\alpha, i} > \sigma'_{\beta, j} \rightarrow \sigma_{\alpha, i} \geq \sigma_{\beta, j} \]

Note the inherent symmetry between the two disciplines here: the contrapositive of the statement is:

\[ \forall \alpha, \beta, i, j: \sigma_{\beta, j} > \sigma_{\alpha, i} \rightarrow \sigma'_{\beta, j} \geq \sigma'_{\alpha, i} \]

which corresponds to swapping the two disciplines.

This ordering constraint leads to a useful intermediate result, a bound on the expression \( \mu'_{\beta}(\sigma'_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i}) \). Let us fix arbitrary \( \alpha, \beta \in \{1, \cdots, N\} \) and \( i \in \{1, \cdots, \text{maxreq}(\alpha)\} \). We will show that:

\[ |\mu'_{\beta}(\sigma'_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i})| \leq P_{\beta}^{\max} \]

This says that the amount of service that \( \beta \) has received when \( \alpha \)'s \( i \)-th request begins being serviced is almost the same in both disciplines; in fact, it differs by at most one request of \( \beta \).

We start by assuming that we can find \( j \) such that \( \sigma_{\alpha, i} \in [\sigma_{\beta, j}, \sigma_{\beta, j+1}) \). The two exceptions to this assumption will be dealt with later. We distinguish two cases:

**Case 1:** \( \sigma_{\alpha, i} \neq \sigma_{\beta, j} \)

In which case we can say:

\[ \sigma_{\beta, j} < \sigma_{\alpha, i} < \sigma_{\beta, j+1} \]

and using the ordering constraint yields

\[ \sigma'_{\beta, j} \leq \sigma'_{\alpha, i} \leq \sigma'_{\beta, j+1} \]

Now, since \( \mu_{\beta} \) and \( \mu'_{\beta} \) are monotonic functions (they measure cumulative throughput):

\[ \mu_{\beta}(\sigma_{\beta, j}) \leq \mu_{\beta}(\sigma_{\alpha, i}) \leq \mu_{\beta}(\sigma_{\beta, j+1}) \]
\[ \mu'_\beta(\sigma'_{\beta,j}) \leq \mu'_\beta(\sigma'_{\alpha,i}) \leq \mu'_\beta(\sigma'_{\beta,j+1}) \]

Note that \( \mu'_\beta(\sigma_{\beta,j}) = \sum_{k=1}^{j-1} P_{\beta,k} \), since the first \( j-1 \) requests of user \( \beta \) have been serviced at time \( \sigma_{\beta,j} \), and the remaining requests have received no service. Similarly,

\[ \mu'_\beta(\sigma'_{\beta,j+1}) = \sum_{k=1}^{j} P_{\beta,k}, \quad \mu'_\beta(\sigma'_{\beta,j}) = \sum_{k=1}^{j-1} P_{\beta,k}, \quad \text{and} \quad \mu'_\beta(\sigma'_{\beta,j+1}) = \sum_{k=1}^{j} P_{\beta,k}. \]

Substituting into the inequalities yields:

\[ \sum_{k=1}^{j-1} P_{\beta,k} \leq \mu'_\beta(\sigma_{\alpha,i}) \leq \sum_{k=1}^{j} P_{\beta,k} \]

\[ \sum_{k=1}^{j-1} P_{\beta,k} \leq \mu'_\beta(\sigma'_{\alpha,i}) \leq \sum_{k=1}^{j} P_{\beta,k} \]

Now, subtracting these inequalities yields:

\[ -P_{\beta,j} \leq \mu'_\beta(\sigma'_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i}) \leq P_{\beta,j} \]

and hence

\[ |\mu'_\beta(\sigma'_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i})| \leq P_{\beta}^{\max} \]

**Case 2: \( \sigma_{\alpha,i} = \sigma_{\beta,j} \)**

This gives us, for \( j > 1 \),

\[ \sigma_{\beta,j-1} < \sigma_{\alpha,i} < \sigma_{\beta,j+1} \]

from which the ordering constraint yields

\[ \sigma'_{\beta,j-1} \leq \sigma'_{\alpha,i} \leq \sigma'_{\beta,j+1} \]

Since \( \mu'_\beta \) is monotonic, we have

\[ \mu'_\beta(\sigma'_{\beta,j-1}) \leq \mu'_\beta(\sigma'_{\alpha,i}) \leq \mu'_\beta(\sigma'_{\beta,j+1}) \]

and thus

\[ \sum_{k=1}^{j-2} P_{\beta,k} \leq \mu'_\beta(\sigma'_{\alpha,i}) \leq \sum_{k=1}^{j} P_{\beta,k} \]
Now, since $\sigma_{\alpha,i} = \sigma_{\beta,j}$,

$$\mu_\beta(\sigma_{\alpha,i}) = \mu_\beta(\sigma_{\beta,j}) = \sum_{k=1}^{j-1} P_{\beta,k}$$

hence

$$-P_{\beta,j-1} \leq \mu_\beta'(\sigma_{\alpha,i}) - \mu_\beta(\sigma_{\alpha,i}) \leq P_{\beta,j}$$

and once again

$$|\mu_\beta'(\sigma_{\alpha,i}) - \mu_\beta(\sigma_{\alpha,i})| \leq P_{\beta}^{\max}$$

In the special case $j=1$, we can not say that $\sigma_{\beta,j-1} < \sigma_{\alpha,i}$. Instead, we rely on the fact that $\mu_\beta'(t)$ can not be negative. Thus,

$$0 \leq \mu_\beta'(\sigma_{\alpha,i}) \leq \mu_\beta'(\sigma_{\beta,j+1}) = \mu_\beta'(\sigma_{\beta,2}) = P_{\beta,1}$$

and

$$\mu_\beta(\sigma_{\alpha,i}) = \mu_\beta(\sigma_{\beta,j}) = \mu_\beta(\sigma_{\beta,1}) = 0$$

Hence, in this case,

$$0 \leq \mu_\beta'(\sigma_{\alpha,i}) - \mu_\beta(\sigma_{\alpha,i}) \leq P_{\beta,1}$$

proving the bound for the case where $j=1$.

So far we have assumed that we can find a $j$ such that $\sigma_{\alpha,i} \in [\sigma_{\beta,j}, \sigma_{\beta,j+1})$. This is not possible in either of two cases: $\sigma_{\alpha,i} < \sigma_{\beta,1}$ and $\sigma_{\alpha,i} \geq \sigma_{\beta,\text{maxreq}(\alpha)}$.

Now, if $\sigma_{\alpha,i} < \sigma_{\beta,1}$, then it follows that $\sigma_{\alpha,i} \leq \sigma_{\beta,1}$. Then

$$\mu_\beta'(\sigma_{\alpha,i}) \leq \mu_\beta'(\sigma_{\beta,1}) = 0$$

and

$$\mu_\beta(\sigma_{\alpha,i}) \leq \mu_\beta(\sigma_{\beta,1}) = 0,$$

and thus

$$\mu_\beta'(\sigma_{\alpha,i}) = \mu_\beta(\sigma_{\alpha,i}) = 0,$$

whence $\mu_\beta'(\sigma_{\alpha,i}) - \mu_\beta(\sigma_{\alpha,i}) = 0$. Hence in this case the bound is true.

The remaining special case to dispose of is where $\sigma_{\alpha,i} \geq \sigma_{\beta,\text{maxreq}(\alpha)}$. This follows by a similar argument as the main proof, with $j=\text{maxreq}(\beta)$, except that there is no $\text{req}(\beta,j+1)$ with which to impose bounds $\mu_\beta(\sigma_{\alpha,i}) \leq \mu_\beta(\sigma_{\beta,j+1})$ and $\mu_\beta'(\sigma_{\alpha,i}) \leq \mu_\beta'(\sigma_{\beta,j+1})$. Instead, one merely notes that as there are no more requests to service after $\text{req}(\beta,j)$, that $\mu_\beta(\sigma_{\alpha,i}) \leq \sum_{k=1}^{j} P_{\beta,k}$ and
\[ \mu'_{\beta}(\sigma'_{\alpha, i}) \leq \sum_{k=1}^{j} P_{\beta, k}. \]

Therefore,

\[ |\mu'_{\beta}(\sigma'_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i})| \leq P_{\beta}^{\text{max}} \]

### 5.7.2. Intermediate Bound 2

The bound that we have just derived can be considered as a constraint on the relative progress of two users in the two service disciplines. We will write it in the following generalized form:

\[ -B_{\alpha, \beta}^{\sigma} \leq \mu'_{\beta}(\sigma'_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i}) \leq B_{\alpha, \beta}^{\sigma} \]

We have shown that, subject to the ordering constraint satisfied by HFQ and HPS, this bound is true with \( B_{\alpha, \beta}^{\sigma} = B_{\alpha, \beta}^{\sigma} = P_{\beta}^{\text{max}} \). We will next explore the corresponding bound for finishing times:

\[ -B_{\alpha, \beta}^{\phi} \leq \mu'_{\beta}(\phi'_{\alpha, i}) - \mu_{\beta}(\phi_{\alpha, i}) \leq B_{\alpha, \beta}^{\phi} \]

It turns out that we can derive such a bound from the first one, given the presence of two more bounds which arise naturally in the context of HFQ and HPS. These bounds are:

\[ \mu_{\beta}(\phi_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i}) \leq U_{\alpha, \beta} \]

\[ \mu'_{\beta}(\phi'_{\alpha, i}) - \mu'_{\beta}(\sigma'_{\alpha, i}) \leq U'_{\alpha, \beta} \]

We have written these bounds in the same form for convenience; the actual values are quite different for HPS and HFQ, except in the special case \( \alpha = \beta \). Note that since \( \mu_{\alpha}(\phi_{\alpha, i}) - \mu_{\alpha}(\sigma_{\alpha, i}) = P_{\alpha, i} \), that \( U_{\alpha, \alpha} = P_{\alpha}^{\text{max}} \). Similarly, \( U'_{\alpha, \alpha} = P_{\alpha}^{\text{max}} \).

This deals with the case \( \alpha = \beta \). We shall now look at the two bounds for \( \alpha \neq \beta \).

Let us first look at HFQ. The relevant bound is \( U_{\alpha, \beta} \). To derive the bound, we merely have to note that HFQ only services one request at a time. Now, since user \( \alpha \) is under service throughout \([\sigma_{\alpha, i}, \phi_{\alpha, i}]\), it follows that, for \( \beta \neq \alpha \), user \( \beta \) receives no service on this interval, so that \( \mu_{\beta}(\phi_{\alpha, i}) - \mu_{\beta}(\sigma_{\alpha, i}) = 0 \). Hence
\[ U_{\alpha,\beta} = 0 \quad (\text{when } \alpha \neq \beta) \]

For HPS, there can be more than one request serviced at a time. However, their relative rates of service are precisely controlled. In particular:

\[
\frac{d}{dt} \mu_\beta(t) = \frac{E_\beta(A_1(t), \cdots, A_N(t))}{E_\alpha(A_1(t), \cdots, A_N(t))} \frac{d}{dt} \mu'_\alpha(t)
\]

Note that we have used the fact that \( \alpha \) is active on \( (\sigma_{\alpha,i}, \phi_{\alpha,i}) (A_\alpha(t) = 1) \). This guarantees that \( E_\alpha(A_1(t), \cdots, A_N(t)) \neq 0 \). Now there are a finite number of values for the quotient in the above equation, since each \( A_\beta(t) \) is always either 0 or 1. Thus there must be a maximum value, which allows us to write:

\[
\mu'_\beta(\phi'_{\alpha,i}) - \mu'_\beta(\sigma'_{\alpha,i}) \leq \max_{a_\alpha = 1} \frac{E_\beta(a_1, \cdots, a_N)}{E_\alpha(a_1, \cdots, a_N)} (\mu'_\alpha(\phi'_{\alpha,i}) - \mu'_\alpha(\sigma'_{\alpha,i}))
\]

where the maximum is taken over the set of \( 2^{N-1} \) valid combinations \( (a_1, \cdots, a_N) \) with \( a_\alpha = 1 \). Now substituting \( \mu'_\alpha(\phi'_{\alpha,i}) - \mu'_\alpha(\sigma'_{\alpha,i}) = P^{\alpha}_{\max} \) gives us the required bound with

\[
U'_{\alpha,\beta} = \max_{a_\alpha = 1} \frac{E_\beta(a_1, \cdots, a_N)}{E_\alpha(a_1, \cdots, a_N)} P^{\alpha}_{\max}
\]

We will now derive the \( B^{\phi^+}_{\alpha,\beta}, B^{\phi^-}_{\alpha,\beta} \) bounds from the existing ones. We have:

\[
-B^{\sigma^-}_{\alpha,\beta} \leq \mu'_\beta(\sigma'_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i}) \leq B^{\sigma^+}_{\alpha,\beta}
\]

\[
0 \leq \mu'_\beta(\phi_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i}) \leq U_{\alpha,\beta}
\]

\[
0 \leq \mu'_\beta(\phi'_{\alpha,i}) - \mu'_\beta(\sigma'_{\alpha,i}) \leq U'_{\alpha,\beta}
\]

(the left hand side of the last two inequalities is a result of the monotonicity of the cumulative throughput functions). Subtracting the last two equations and rearranging gives:

\[
-U_{\alpha,\beta} \leq (\mu'_\beta(\phi_{\alpha,i}) - \mu'_\beta(\phi_{\alpha,i})) - (\mu'_\beta(\sigma_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i})) \leq U'_{\alpha,\beta}
\]

and thus

\[
-U_{\alpha,\beta} + (\mu'_\beta(\sigma'_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i})) \leq (\mu'_\beta(\phi'_{\alpha,i}) - \mu'_\beta(\phi_{\alpha,i})) \leq U'_{\alpha,\beta} + (\mu'_\beta(\sigma'_{\alpha,i}) - \mu'_\beta(\sigma_{\alpha,i}))
\]
which yields

$$-U_{\alpha, \beta} - B_{\alpha, \beta}^\sigma \leq (\mu'_\beta(\phi_{\alpha,i}) - \mu_\beta(\phi_{\alpha,i})) \leq U_{\alpha, \beta} + B_{\alpha, \beta}^\sigma$$

Hence we have proved that the bound is true with

$$B_{\alpha, \beta}^{\phi, +} = B_{\alpha, \beta}^{\sigma, +} + U_{\alpha, \beta}$$

$$B_{\alpha, \beta}^{\phi, -} = B_{\alpha, \beta}^{\sigma, -} + U_{\alpha, \beta}$$

Substituting the values that we derived for HFQ and HPS above, we have, for $\alpha \neq \beta$:

$$B_{\alpha, \beta}^{\phi, +} = P_{\beta}^{\max} + \max_{a_{1}, \ldots, a_{N}} \frac{E_\beta(a_{1}, \ldots, a_{N})}{E_\alpha(a_{1}, \ldots, a_{N})} P_{\alpha}^{\max}$$

$$B_{\alpha, \beta}^{\phi, -} = P_{\beta}^{\max}$$

Observe that, since $\mu'(\phi_{\alpha,i}) = \mu_\alpha(\phi_{\alpha,i})$,

$$B_{\alpha, \alpha}^{\sigma, +} = B_{\alpha, \alpha}^{\phi, -} = 0$$

5.7.3. Delay Discrepancy Bound

We are now in a position to derive bounds on the delay discrepancies. Starting with

$$-B_{\alpha, \beta}^{\sigma, -} \leq \mu'_\beta(\sigma_{\alpha,i}) - \mu_\beta(\sigma_{\alpha,i}) \leq B_{\alpha, \beta}^{\sigma, +}$$

and summing over all $\beta$ yields

$$-\sum_{\beta=1}^{N} B_{\alpha, \beta}^{\sigma, -} \leq \mu'(\sigma_{\alpha,i}) - \mu(\sigma_{\alpha,i}) \leq \sum_{\beta=1}^{N} B_{\alpha, \beta}^{\sigma, +}$$

Recall that $\mu'(t) = \mu(t)$ for all $t$. Also recall that there is a minimum rate of service $r_{\min}$, such that if the service disciplines are not idle at any point in time over $[t_1, t_2]$, then $\mu(t_2) - \mu(t_1) \geq r_{\min}(t_2 - t_1)$.

Suppose that $\sigma'_{\alpha,i} > \sigma_{\alpha,i}$. Since $\tau_{\alpha,i} \leq \sigma'_{\alpha,i}$, HPS has work to do throughout $(\sigma_{\alpha,i}, \sigma'_{\alpha,i})$ (namely the request req($\alpha_i$)), and is thus not idle at any stage of this interval. Thus

$$\mu'(\sigma'_{\alpha,i}) - \mu(\sigma_{\alpha,i}) = \mu'(\sigma'_{\alpha,i}) - \mu(\sigma_{\alpha,i})'$$
\[
\geq r_{\min}(\sigma'_{\alpha,i} - \sigma_{\alpha,i})
\]

So

\[
0 < \sigma'_{\alpha,i} - \sigma_{\alpha,i} \leq r_{\min}^{-1}(\mu'(\sigma'_{\alpha,i}) - \mu(\sigma_{\alpha,i}))
\]

\[
\leq r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\sigma,+}
\]

Similarly, if \(\sigma'_{\alpha,i} - \sigma_{\alpha,i}\) is negative, we get

\[
0 \geq \sigma'_{\alpha,i} - \sigma_{\alpha,i} \geq -r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\sigma,-}
\]

Combining the two cases gives us

\[
-r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\sigma,-} \leq \sigma'_{\alpha,i} - \sigma_{\alpha,i} \leq r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\sigma,+}
\]

By a similar argument,

\[
-r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\phi,-} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq r_{\min}^{-1} \sum_{\beta=1}^{N} B_{\alpha,\beta}^{\phi,+}
\]

If we now substitute the relevant values for HPS/HFQ, we get

\[
-r_{\min}^{-1} \sum_{\beta \neq \alpha}^{N} P_{\beta}^{\max} \leq \sigma'_{\alpha,i} - \sigma_{\alpha,i} \leq r_{\min}^{-1} \sum_{\beta \neq \alpha}^{N} P_{\beta}^{\max}
\]

\[
-r_{\min}^{-1} \sum_{\beta \neq \alpha}^{N} P_{\beta}^{\max} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq r_{\min}^{-1} \sum_{\beta \neq \alpha}^{N} (P_{\beta}^{\max} + \max_{a_\alpha=1}^{E_{\alpha}(a_1, \cdots, a_N)} \frac{E_{\beta}(a_1, \cdots, a_N)}{E_{\alpha}(a_1, \cdots, a_N)} P_{\alpha}^{\max})
\]

### 5.7.4. Bandwidth Discrepancy Bound

We will now prove the following bound on the (cumulative) throughput discrepancy:

\[
\forall \alpha, t: \max_{\beta} \left( B_{\beta,\alpha}^{\phi,-}, P_{\alpha}^{\max} \right) \leq \mu'_{\alpha}(t) - \mu_{\alpha}(t) \leq \max_{\beta} \left( B_{\beta,\alpha}^{\phi,+}, P_{\alpha}^{\max} \right)
\]

We will concentrate on the upper bound. Fix \(\alpha\) and \(t\). Suppose that a request \(\text{req}(\beta,i)\) exists, for which \(\phi_{\beta,i} \leq t \leq \phi'_{\beta,i}\). Then it immediately follows from the monotonicity of \(\mu'_{\alpha}(t)\) and \(\mu_{\alpha}(t)\) that
\[
\mu_\alpha'(t) - \mu_\alpha(t) \leq \mu_\alpha'(\phi_{\beta,i}) - \mu_\alpha(\phi_{\beta,i}) \\
\leq B_{\beta,\alpha}^{0+} \\
\leq \max_{\beta} (\max \{ B_{\beta,\alpha}^{0+}, P^{\max} \})
\]

and so in this case, the bound is proven. Henceforth we will assume that no such request exists. Now this assumption implies that, if for any req(\beta,i) we have that \(\phi_{\beta,i} \leq t\), then \(\phi_{\beta,i}' < t\) (otherwise we would have an instance of a request satisfying \(\phi_{\beta,i} \leq t \leq \phi_{\beta,i}'\)).

Now, since HFQ services one request at a time, there is at most one request req(\gamma,j) satisfying \(\sigma_{\gamma,j} < t < \phi_{\gamma,j}\). Given such a request, we can prove that

\[
\mu_{\gamma}'(t) - \mu_{\gamma}(t) \geq -P^{\max}
\]

To see this, note that since \(\phi_{\gamma,j-1} \leq \sigma_{\gamma,j} < t\), it must follow that \(\phi_{\gamma,j-1}' < t\) by our earlier assumption. Thus,

\[
\mu_{\gamma}'(t) - \mu_{\gamma}(t) \geq \mu_{\gamma}'(\phi_{\gamma,j-1}) - \mu_{\gamma}(\phi_{\gamma,j}) \\
= \sum_{k=1}^{j-1} P_{\gamma,k} - \sum_{k=1}^{j} P_{\gamma,k} \\
= -P_{\gamma,j} \\
\geq -P^{\max}
\]

Now suppose that \(\gamma\) is any of the other users, for which no request req(\gamma,j) satisfies \(\sigma_{\gamma,j} < t < \phi_{\gamma,j}\). In this case, if a request satisfies \(\sigma_{\gamma,j} < t\), then \(\phi_{\gamma,j} \leq t\), and by our assumption, \(\phi_{\gamma,j}' \leq t\). Thus, any request of user \(\gamma\) which starts before \(t\) in HFQ must finish before \(t\) in HPS, and hence

\[
\mu_{\gamma}'(t) - \mu_{\gamma}(t) \geq 0
\]

Now we recall that \(\mu'(t) = \mu(t)\), and apply the definitions of \(\mu(t)\) and \(\mu'(t)\) to give:

\[
\mu_\alpha'(t) - \mu_\alpha(t) = (\mu'(t) - \mu(t)) - \sum_{\gamma \neq \alpha} \mu_{\gamma}'(t) - \mu_{\gamma}(t)
\]
\[
- \sum_{\gamma \neq \alpha} \mu'_{\gamma}(t) - \mu_{\gamma}(t) \\
\leq P_{\text{max}}
\]

Since for at most one \( \gamma \neq \alpha \), \( \mu'_{\gamma}(t) - \mu_{\gamma}(t) \geq -P_{\text{max}} \), and for all other users \( \mu'_{\gamma}(t) - \mu_{\gamma}(t) \geq 0 \). Hence the bound is true in this case also.

The lower bound follows by a similar argument.

We have shown that

\[
T_{\alpha}^{+} \leq \max \left( \max_{\beta} \left( B_{\beta, \alpha}^{\phi^{-}} \right), P_{\text{max}} \right)
\]

\[
T_{\alpha}^{-} \leq \max \left( \max_{\beta} \left( B_{\beta, \alpha}^{\sigma^{-}} \right), P_{\text{max}} \right)
\]

Let us substitute the upper bounds for \( B_{\alpha, \beta}^{\phi_{+}} \) and \( B_{\alpha, \beta}^{\sigma_{-}} \) proved above:

\[
T_{\alpha}^{+} \leq \max \left( \max_{\beta} \left( P_{\alpha}^{\text{max}} \right), \max_{a_{1}, \ldots, a_{N}} \frac{E_{\alpha}(a_{1}, \ldots, a_{N})}{E_{\beta}(a_{1}, \ldots, a_{N})} P_{\beta}^{\text{max}} \right), P_{\text{max}} \right)
\]

\[
T_{\alpha}^{-} \leq \max \left( \max_{\beta} \left( P_{\alpha}^{\text{max}} \right), P_{\text{max}} \right) = P_{\text{max}}
\]

Hence,

\[
-P_{\text{max}} \leq \mu'_{\alpha}(t) - \mu_{\alpha}(t) \leq \max \left( \max_{\beta} \left( P_{\alpha}^{\text{max}} \right), \max_{a_{1}, \ldots, a_{N}} \frac{E_{\alpha}(a_{1}, \ldots, a_{N})}{E_{\beta}(a_{1}, \ldots, a_{N})} P_{\beta}^{\text{max}} \right), P_{\text{max}} \right)
\]

5.8. Comparison with Existing Results

We conclude this chapter by comparing our results with the bounds previously obtained for FQ and WFQ. Both PS and GPS are degenerate cases of HPS, and, furthermore, the corresponding special cases of HFQ are variants of FQ and WFQ, based on starting times instead of finishing times. Greenberg and Madras refer to this variant of FQ as FQS, and use FQF to refer to the standard FQ algorithm. We will extend this notation to WFQ, using WFQS to denote the variant of WFQ which uses starting times, and WFQF to denote standard WFQ. (We will also follow Greenberg and Madras in using FQ to mean either FQS or FQF, and thus WFQ to mean either WFQS or WFQF). Since FQS and WFQS are degenerate cases of HFQ,
we can use our results to prove discrepancy bounds for these service disciplines. Comparing these bounds with the existing bounds gives us a way of checking how good our bounds are.

Note that Greenberg and Madras [15] proved their bounds for both FQS and FQF, making direct comparison possible. However, Parekh and Gallager's results [30] only apply to WFQF, and only provide one side of each of the delay and throughput bounds. Thus comparing our results with those of Parekh and Gallager may be seen as a slightly dubious exercise. However, it is still a useful comparison to make, since WFQS could be used as an alternative to WFQF, and we are not aware of any existing analyses of WFQS. Thus it can be viewed as a comparison between two rival systems, at the level of what is known about them.

For the purpose of making the comparison, we note that the analyses of FQ and WFQ both assume a constant rate of service. Hence we set \( r_{\min} = r_{\max} = r \). In addition, they make no distinction between the maximum request length for the various users. We will thus set \( P_{\alpha}^{\max} = P^{\max} \).

For both FQ and WFQ, we have a simple, non-hierarchical policy. That is, each of the \( N \) users is a child of the root of the class tree, and has as associated entitlement function of the form

\[
E_{\alpha}(A_1(t), \ldots, A_N(t)) = \begin{cases} 
\frac{w_{\alpha}}{\sum_{\alpha} w_{\alpha} A_{\alpha}(t)} & \text{if } A_1(t) = 1 \\
0 & \text{if } A_1(t) = 0 
\end{cases}
\]

FQ satisfies the additional constraint that \( w_{\alpha} = 1 \) for all \( \alpha \), which gives us the following formula for FQ

\[
E_{\alpha}(A_1(t), \ldots, A_N(t)) = \begin{cases} 
\frac{1}{\sum_{\alpha} A_{\alpha}(t)} & \text{if } A_1(t) = 1 \\
0 & \text{if } A_1(t) = 0 
\end{cases}
\]

\( \sum_{\alpha} A_{\alpha}(t) \) is merely the number of users active at time \( t \).

We will now substitute the first formula for entitlement above (that for
GPS/WFQ) into our discrepancy bounds. This yields, on simplification,

\[ -P_{\alpha}^{\text{max}} \leq \mu'_\alpha(t) - \mu_\alpha(t) \leq P_{\alpha}^{\text{max}} \left( 1 + \max_{\beta} \left( \frac{w_\alpha}{w_\beta} \right) \right) \]

\[ -r^{-1} (N-1) P_{\alpha}^{\text{max}} \leq \sigma'_{\alpha,i} - \sigma_{\alpha,i} \leq r^{-1} (N-1) P_{\alpha}^{\text{max}} \]

\[ -r^{-1} (N-1) P_{\alpha}^{\text{max}} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq r^{-1} P_{\alpha}^{\text{max}} \sum_{\beta \neq \alpha} \left| \frac{w_\beta}{w_\alpha} \right| \]

Our analysis yields two delay bounds: one for starting times, and one for finishing times. The existing analyses for FQ and WFQ only look at finishing times. Of course, the bounds for starting times are not very useful any way. Thus we will concentrate on the first and third bounds above.

We will first compare our results with those proven for FQ in [15]. Let us further reduce the above bounds according to the substitution \( w_\alpha = 1 \):

\[ -P_{\alpha}^{\text{max}} \leq \mu'_\alpha(t) - \mu_\alpha(t) \leq 2P_{\alpha}^{\text{max}} \]

\[ -r^{-1} (N-1) P_{\alpha}^{\text{max}} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq r^{-1} 2(N-1) P_{\alpha}^{\text{max}} \]

The corresponding results for FQS are:

\[ -P_{\alpha}^{\text{max}} \leq \mu'_\alpha(t) - \mu_\alpha(t) \leq P_{\alpha}^{\text{max}} \]

\[ -r^{-1} N P_{\alpha}^{\text{max}} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \leq r^{-1} N P_{\alpha}^{\text{max}} \]

We see that our results are close to those of Greenberg and Madras, the main difference being the extra factor of 2 on the right hand side of both bounds.

For WFQF, the known bounds are

\[ \mu'_\alpha(t) - \mu_\alpha(t) \leq P_{\alpha}^{\text{max}} \]

\[ -r^{-1} P_{\alpha}^{\text{max}} \leq \phi'_{\alpha,i} - \phi_{\alpha,i} \]

Comparing this with the results for FQS obtained above, we see that WFQF does considerably better, at least for the two bounds for which something is known. However, it is known that the discrepancies for WFQF are considerably larger on the side where no bound is known [3]. WFQS does achieve a lower bound on throughput
discrepancy of $-P^\text{max}$, which is no better or worse than WFQF’s upper bound of $P^\text{max}$. Evidently, the change from finishing times to starting times has exchanged the good upper bound and poor lower bound of WFQF, to produce a good lower bound and a poor upper bound. The same cannot be said for the delay bounds. WFQF yields a lower bound for delay which is much better than either of the known delay bounds for WFQS, and it is the lower delay bound for WFQS which is the best of the two. It is possible that a better upper bound is possible for WFQS (and presumably for HFQ). This is an important area for future research.
6. An Experimental HFQ Implementation

In the previous chapter we presented a theoretical analysis of the throughput and delay characteristics of the new HFQ algorithm. To enhance our understanding of the algorithm, we have also implemented it in a real system and measured its behaviour. This chapter describes the implementation and presents the measurements we gathered.

We begin by describing the test environment, which consisted of a modified PPP service running on a Plan 9 machine. Next we deal with the way in which HFQ was implemented, and how this fitted into the existing PPP implementation. We then describe our test methodology. The tests themselves are described next, followed by a presentation of the results. We conclude the chapter with a discussion of what the test results show.

6.1. The Test Environment

The HFQ algorithm was implemented as a modification to the user mode PPP server included with the Plan 9 Operating System [31]. Two PCs running Plan 9 were connected via a serial link operating at a rate of 9600 baud. (This was the highest speed that we could run our hardware at without encountering serious levels of packet loss due to serial overruns). One of these machines ran the program aux/pppsclient, which is designed to give a machine with no other network connection access to the internet via the Point-to-Point Protocol (PPP) [39]. The other machine ran a modified version of aux/pppsserver, 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 PPP as part of the kernel (as is usual in Unix based systems), Plan 9 has PPP as a user mode program. This is possible because of Plan 9's 'Streams' implementation [32], [31]. 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.

6.2. The Implementation

We now discuss the implementation. This can be logically divided into two parts: implementation structure and HFQ implementation. Implementation structure refers to how we modified the existing PPP implementation to introduce our own service discipline. This involves changing the process structure (the existing PPP implementation is a concurrent program) and adding hooks to call the entry points of the HFQ implementation. The section on the HFQ implementation deals with issues internal to the service discipline itself, such as the data structures used for the HFQ priority queue, and the way in which problems with numeric accuracy are dealt with.
6.2.1. Implementation Structure

The Plan 9 program `aux/pppserver` is written in a concurrent programming language called Alef [31]. 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. This involves the use of 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 into 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 recognize 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.

6.2.2. The HFQ Implementation

Floating point arithmetic was used for all of the HPS-related variables, such as csrv
and creq (see section 4.3 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 the comparisons. Further work
is needed in applying principles of numeric analysis to make the algorithm more
robust.

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, although
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 cannot 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, fcproc 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.

6.3. Test Methodology

We now describe the way in which we tested the HFQ implementation. Testing consisted of three components. These were the generation of arrivals to the system, the collection of results, and the processing of the results. We describe each of these in turn.

6.3.1. Test Data Generation

In this section we describe the method used to generate sequences of arrivals for each user. Each test was driven by a simple text file (described further in the next section), which controlled the times and lengths of the arrivals for each user. This file was generated prior to the test using a collection of scripts and special programs. A variety of user behaviours were simulated. These included deterministic distributions, poisson distributions and Leaky Bucket constrained distributions.

Our use of simple text files for the input to the test software enabled us to use awk scripts to generate the input data for the arrival sequences. Each experiment started with an awk script which generated arrival sequences for each user, which was then piped through sort -n. An auxiliary program poisson was used to generate poisson arrival time distributions, and exponential distributions of packet lengths (where needed). We are also interested in the delays experienced by a user with Leaky Bucket constrained arrivals. A program called regulator was written, which took an arrival sequence and constrained it according to the given Leaky Bucket parameters. Packet arrivals which would occur too soon were merely delayed, not discarded. This program was typically used in conjunction with poisson to generate suitable Leaky Bucket constrained arrivals.
6.3.2. Data Collection

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. source 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 file describes a packet to be generated. The lines are expected to be in the following format:

\[ t \text{ user seq len} \]

where \( t \) is the arrival time in milliseconds, \( \text{user} \) controls the UDP port that the packet is sent on, \( \text{seq} \) is a sequence number to be imbedded in the packet, and \( \text{len} \) 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 was made to classify UDP packets based on the last 4 bits of the UDP source port, allowing for up to 16 users. The time \( t = 0 \) was defined as the time 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 was captured to 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 was to bind to UDP port 5555 and capture all 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 was received (with origin at the time that the first packet arrived), \( \text{user} \) is the last 4 bits of the UDP source port, \( \text{seq} \) is the sequence number extracted from the packet, and \( \text{len} \) is the length of the packet. The output of sink was 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. Firstly, 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 a certain set of sequence numbers are
reached, but unfortunately packet loss does occur and renders such methods un reli able. 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 synchronization 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 synchronize 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 (each byte is
sent as 8 bits plus 1 start bit and 1 stop bit). 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 adjust-
ment 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 subtract
$len/960$, where $len$ is the length of that packet.

6.3.3. Processing of Results

The results were processed and analysed in a couple of ways. The throughput
behaviour of our implementation was compared with that of HPS by graphing them
both on the same axes. The output of sink was processed by an awk script called
fggraph, which calculates cumulative throughputs for each user and turns them
into suitable data for the system graph command. A program called psgraph
simulated the behaviour of HPS, given the output of source, and generated corre-
sponding cumulative throughput data in the same format as fggraph. Thus the
behaviour of HFQ and HPS could be compared visually, by graphing their cumula-
tive throughputs on a common set of arrival data.

The delay behaviour of the system was also examined. A script called delay
was used to calculate the delays experienced by each packet owned by a given user,
and generate a graph against time.
Analysis of the results was complicated by the existence of 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 saves some of the overheads; UDP is not compressed, however. 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 frames 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 frame, based on the number of times that the escape code appears. This number is difficult to predict, though it can be minimized by initializing the buffer that source sends to (say) all zeros. Another overhead added by PPP 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.

6.4. The Tests

We present 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.
6.4.1. Test 1

The class tree for the first test was as shown in Figure 6.4.1.1. 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.

![Class Tree for Test 1](image)

Figure 6.4.1.1: 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.

6.4.2. Test 2

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 6.4.2.1 (astute readers may note that it is based on one of the class
trees used in [11]).

```
Figure 6.4.2.1: Class Tree for Test 2

Each link in Figure 6.4.2.1 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, i.e., 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 summarizes 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.

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

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.

6.5. The Results

6.5.1. Test 1

Figure 6.5.1.1 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 pagraph (dashed curve).

![Cumulative Throughput Graphs](image_url)

Figure 6.5.1.1: Cumulative Throughput for HFQ (solid) and HPS (dashed), Test 1

Note that the curves are similar in shape, but 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 serial hardware of the machine running sink. There were a surprisingly large number of these (around 70 for an 800 second test) given that the serial hardware 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 pgraph 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. pgraph 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.

6.5.2. Test 2

Figure 6.5.2.1 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. Figure 6.5.2.2 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 penalizes 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, and thus their delay remains relatively small (e.g., compared to user 3). A few major peaks, particularly for user 5, occur as a result of their lack of constraint.

![Graphs of cumulative throughput for HFQ (solid) and HPS (dashed), Test 2](image)

Figure 6.5.2.1: Cumulative Throughput for HFQ (solid) and HPS (dashed), Test 2
6.6. Conclusion

The HFQ algorithm of Chapter 4 was implemented in a practical system, and tested. The algorithm performed as expected in the tests.
7. Conclusion

We have defined a new queueing algorithm called Hierarchical Fair Queueing. HFQ extends WFQ (weighted Fair Queueing) 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 service completion times in HFQ and those in the idealized fair discipline HPS, and bounds on the difference in cumulative throughputs for the two disciplines. 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.

We have applied analytical techniques to demonstrate HFQ's fairness. These show that, under a set of reasonable constraints, if we feed the same arrival sequence to the HFQ and HPS disciplines, explicit bounds on the delay and throughput discrepancies may be obtained.

7.1. Further Work

There are a number of areas we see where further research is required. The first of these concerns the efficiency of the algorithm. The HFQ algorithm described in this thesis is based on a simulation of the HPS discipline. As such, it is considerably less efficient than FQ. Implementations of FQ have been suggested [15], [24] which give $O(\log N)$ behaviour per event (events being packet arrival and service completion). Some variants exist which are less fair (the errors in approximating GPS are larger) but even more efficient, some boasting $O(\log \log N)$ or even $O(1)$ behaviour [40], [38]. In contrast, the HFQ algorithm is $O(M)$, where $M$ is the number of nodes in the class tree. Furthermore, HFQ suffers from the same efficiency problem as FQ which is described by Golestani [12], namely that it is possible for the HPS simulation to generate an event for every user during the time it takes to service a single packet. Both these problems make HFQ unsuitable for high speed applications.

Developing a more efficient version of HFQ is a difficult problem, which will
need to be solved before this approach to bandwidth sharing can gain widespread acceptance. The most promising direction seems to be to make use of the various efficient FQ-like algorithms in some way. Since HPS can be thought of as being composed of a number of GPS disciplines, and we can approximate a GPS discipline efficiently with an FQ-like algorithm (such as SCFQ), it is possible to combine the appropriate FQ-like algorithms to create an efficient implementation of HPS. This has been proposed by Bennett and Zhang [2]. A weakness of this method is that the errors introduced by each FQ-like algorithm are cumulative.

Another question for further research is whether better bounds on the throughput and delay discrepancies are possible. If we apply our theoretical results to simple FQ as a degenerate case (using a flat class tree where all nodes have equal shares), then the upper bound on the throughput discrepancy that we obtain is twice that obtained by Greenberg and Madras [15]. Preliminary investigation suggests that it is possible to derive another upper bound here which gives the same bound as [15] in this case, but in general this bound is not always superior. In [15] the bound obtained for simple FQ is shown to be the best possible. It is regrettable that an analogous result is not available for HFQ; it would be useful to know what the best obtainable bound is. Our delay bounds for HFQ do not compare well with those known for WFQ either. It remains to be seen whether they can be improved.

Difficulties arose during the implementation of HFQ associated with numeric underflow. Due to the inexactness of floating point arithmetic, it was possible for the HPS simulation to miscalculate the time of the next event slightly, with the result that the simulation never reached the next event, but instead looped continually. This problem was solved by introducing a "fudge factor" into the comparison of event times. Further examination of the numeric robustness issues associated with the algorithm is clearly needed. For instance, we are not sure what effect the approximateness of the practical algorithm has on the theoretical bounds. Numeric robustness is a problem for all Fair Queuing algorithms, not just HFQ.

HFQ has the disadvantage of not being able to decouple delay bounds from bandwidth allocations, as is possible in CBQ as well as some non-hierarchical algorithms. HFQ does give delay bounds, but they are fixed for a given set of bandwidth allocations. It may be possible to generalize HFQ to allow separate control over
delays, as is done with delay-EDD [47]. It is not known at present what effect this would have on the effectiveness with which HFQ approximates HPS. This is another area for future research.

Powerful theoretical results concerning end-to-end delays in networks of GPS disciplines are known [30], [29], where arrivals are constrained by Leaky Bucket. This helps to justify the use of GPS as the ideal, maximally fair discipline for non-hierarchical policies, and to convert delay discrepancy bounds (when an algorithm is compared with GPS) into practical delay bounds. The results for locally-stable sessions generalize easily to HFQ/HPS through the use of the Guaranteed Rate Clock framework [14] (see section 5.5.3). However, nothing is known about the general case for HPS servers. Analysis of non-locally-stable sessions under HPS could be useful for increasing the flexibility with which HPS may be used.
References


20. Jain, R., Ramakrishnan, K.K., and Chiu, D. Congestion Avoidance in


31. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey,


40. Suri, S., Varghese, G., and Chandranmenon, G. Leap Forward Virtual Clock: A New Fair Queueing Scheme with Guaranteed Delays and Throughput Fairness. WUCS-96-10, Department of Computer Science. Washington University, St Louis, June 1996.


Appendix—Code of the Implementation

/*
 *  tq.h -- Fair Queuing adts
 */

enum {
    NQ = 64,           /* maximum number of queues */
};

typedef adt Classifier;
typedef adt Classtrree;
typedef adt Buf;
typedef adt Pktq;

tagagr ArrivalMsg {
    Buf *b;
    int class;
};

adt Pktq {
    Buf *head;
    Buf *tail;
    void init(*Pktq);
    void enqueue(*Pktq, Buf*);
    Buf *dequeue(*Pktq);
    void enqueuepri(*Pktq, Buf*, float);
    int isempty(*Pktq);
};

adt FQ {
    Classtrree *root;
    int nq;
    Classtrree **class;
    Pktq gpri;
    chan(ArrivalMsg) arrival;
    chan(int) reqsrv;
    chan(Buf*) nextsrv;
    void init(*FQ, int);
    void writerproc(*FQ, int);
    void enqueue(*FQ, Buf*, int);
    void fgproc(*FQ);
    int classify(*FQ, byte*, int);
};
adt Buf {
    extern byte *data;
    extern int len;
    extern Buf *next;
    extern float pri;

    Buf *new(int);
    void free(*Buf);
}

dt Classtree {
    int shares;
    int sashares; /* sum of shares of active children */
    float creq; /* cumulative requests - leaf */
    float creql; /* requests up to pkt under service */
    float csrv; /* cumulative service - leaf */
    Pktq q;

    extern Classtree *children;
    Classtree *nextchild;
    Classtree *parent;

    extern byte *name; /* for debugging */

    Classtree *init(Classtree*, int);
    void arrival(*Classtree, Buf*);
    void update(*Classtree, float, Pktq*);
    void update(*Classtree, float, float, Pktq*);
    (Classtree*, float) predict(*Classtree, float);
    void active(*Classtree);
    void inactive(*Classtree);
    int isactive(*Classtree);
    void sanity(*Classtree);
}
/*
 * fq.1 -- adt FQ
 */
#include <sleef.h>
#include "ppp.h"

#define USED(x) if(x);

int msecs(void);

void
FQ.init(FQ *fq, int fd)
{
    Classtree *c;
    int i;

    /* todo: allow dynamic configuration */
fq->nq = NQ;
fq->class = malloc(fq->nq*sizeof( Classtree* ));
fq->root = .Classtree.init(nil, nil);
fq->root->name = "root node";
fq->class[0] = .Classtree.init(fq->root, nil);
fq->class[1] = .Classtree.init(fq->root, nil);
c = .Classtree.init(fq->root, 1);
c->name = "udp class 1";
fq->class[2] = .Classtree.init(c, 1);
fq->class[3] = .Classtree.init(c, 1);
fq->class[4] = .Classtree.init(c, 2);
c = .Classtree.init(fq->root, 4);
c->name = "udp class 2";
fq->class[5] = .Classtree.init(c, 5);
fq->class[6] = .Classtree.init(c, 5);
fq->class[7] = .Classtree.init(c, 6);
    for (i = 8; i <= 17; i++)
    
        fq->class[i] = .Classtree.init(fq->root, 1);
c = .Classtree.init(fq->root, 1);
c->name = "tcp class";
    for (i = 18; i <= 33; i++)
    
        fq->class[i] = .Classtree.init(c, 1);

    fq->qpri.init();

    for (i = 0; i <= 33; i++) {
        fq->class[i]->name = malloc(10);
        sprintf(fq->class[i]->name, "%d", i);
    }

    alloc fq->reqsrv, fq->nextsrv, fq->arrival;
    proc fq->fpforward();
    proc fq->writerproc(fd);
}

void
FQ.writerproc(FQ *fq, int fd)
{
Buf *buf;
int ctlfld;

cctlfd = open("/dev/eialctl", O_RDWR);
for (; ;) {
    fq->reqsrv <= 1;
    buf = <-fq->nextsrv;
    if (write(fd, buf->data, buf->len) != buf->len)
        break;
    buf->free();
    write ctlfd, "f", 2); /* flush output */
}
exits("bad write");

void
FQ.enqueue(FQ *fq, Buf *b, int class)
{
    fq->arrival <= (b, class);
}

void
FQ.fqproc(FQ *fq)
{
    Buf *b;
    float t;
    int i, npkts, tot;

    /* debugging */
    Class tree *cdep;
    float tdep;

    npkts = 0;
    tot = 0;
    for (; ;) {
        t = msecs() / 1000.;
        alt{
            case (b, i) = <-fq->arrival:
                npkts++;
                tot++;
                fq->root->update(t, &fq->qpri);
                fq->class[i]->arrival(b);
                break;
            case <-fq->reqsrv:
                if (npkts == 0) {
                    /* idle, wait for arrival */
                    (b, i) = <-fq->arrival;
                    tot++;
                    fnextsrv <= b;
                    t = msecs() / 1000.;
                    fq->root->update(t, &fq->qpri);
                    fq->class[i]->arrival(b);
                    b = fnextsrv->queue();
                    if (b != nil) {
                        fnextsrv <= b;
                        continue;
                    }
                }
        }
    }
}


```c

npkts++;

for (;;) {
    fq->root->update(t, &fq->qpri);
    b = fq->qpri.dequeue();
    if (b != nil)
        break;
    (cdemp, tdemp) = fq->root->predict(9600/10);
    fq->root->sanity();
    sleep(1); /* BUG: arrivals are delayed */
    t = msecs()/1000.0;
}
npkts--;
fq->nextsrv = b;
break;
}

Classtree *
Classtree.init(Classtree *parent, int shares)
{
    Classtree *c;
    alloc c;
    c->parent = parent;
    c->children = nil;
    if (parent == nil)
        c->nextchild = nil;
    else {
        c->nextchild = parent->children;
        parent->children = c;
    }
    c->shares = shares;
    c->sshares = 0;
    c->creq = c->creql = 0;
    c->csrv = 0.;
    c->q.init();
    return c;
}

intern int msecfd = -1;

int
msecs(void)
{
    byte buf[12];

    if (msecfd < 0) {
        msecfd = open("/dev/mssec", OREAD);
        if (msecfd < 0){
            fprintf(2, "msecs: cannot open: %s\n");
            exits("msecs");
        }
    }
}
```


seek(msecfd, 0, 0);
read(msecfd, buf, sizeof(buf));
return strtoui(buf, nil, 10);
}
/ * hfg.1 -- HFG implementation */
#include <alef.h>
#include "ppp.h"

#define LineRate  (96000/10)

float tlast;

void Classtree.arrival(Classtree *c, Buf *b) {
    c->q.enqueue(b);
    if (c->creq - c->csr < 0.)
        c->active();
    c->creq += b->len;
}

void Classtree.update(Classtree *root, float time, Pktq *qpri) {
    Classtree *cdep;
    float tdep;

    for (; ; ) {
        (cdep, tdep) = root->predict(LineRate);
        if (cdep == nil || tlast+tdep - time > 0.)
            break;
        root->update(LineRate, tdep, qpri);
        tlast += tdep;
    }
    if (time- tlast > 0.) {
        root->update(LineRate, time-tlast, qpri);
        tlast = time;
    }
}

void Classtree.update(Classtree *c, float rate, float deltat, Pktq *qpri) {
    float newsrv, pri;
    Buf *buf;

    if (c->children == nil) {
        /* leaf node */
        if (c->creq - c->csr > 0.000001) {
            newsrv = c->csr + rate*deltat;
            while (newsrv - c->creql > -0.000001) {
                buf = c->q.dequeue();
                if (buf == nil)
                    break;
                pri = tlast+(c->creql-c->csr)/rate;
                qpri->enqueuepri(buf, pri);
                c->creql += buf->len;
            }
        }
    }
}
if (newsrv - c->creq >= -0.00001) {
    newsrv = c->creq;  /* idle -- clamp at creq */
    if ((buf = c->q.dequeue()) != nil) {
        pri = tlast*(c->creql-c->csrv)/rate;
        qpri->enqueuepri(buf, pri);
    }
    c->inactive();
}
c->csrv = newsrv;
}
return;

if (c->shares == 0) {  /* this internal node is idle */
    return;
    rate /= c->shares;
    for (c = c->children; c; c = c->nextchild)
        c->update!(rate*c->shares, deltat, qpri);
}

(Classstree *, float)
Classstree.predict(Classstree *c, float rate)
{
    Classstree *nextc, *tryc;
    float nextt, tryt;

    if (c->children == nil) {  /* leaf node */
        if (c->creq - c->csrv > 0.)
            return (c, (c->creql-c->csrv)/rate);
        else {
            if (c->q.dequeue() != nil) check 0, "predict confused!";
            return (nil, 0);
        }
    }
    if (c->shares == 0) {  /* this internal node is idle */
        return (nil, 0);
    }
    rate /= c->shares;
    (nextc, nextt) = (nil, 0);
    for (c = c->children; c; c = c->nextchild) {
        (tryc, tryt) = c->predict(rate*c->shares);
        if (nextc == nil || tryc != nil && tryt < nextt)
            (nextc, nextt) = (tryc, tryt);
    }
    return (nextc, nextt);
}

void
Classstree.active(Classstree *c)
{
    Classstree *p;
    p = c->parent;
    if (p == nil)
        return;
    p->shares += c->shares;
    if (p->shares == c->shares)
become p->active();
}

void
Classtree.inactive(Classtree *c)
{
    Classtree *p;
    p = c->parent;
    if (p == nil)
        return;
    p->sashares -= c->shares;
    if (p->sashares == 0)
        become p->inactive();
}

int
Classtree.isactive(Classtree *c)
{
    Classtree *p;

    if (c->children == nil)
        return !c->qisempty();

    for (p = c->children; p; p = p->nextchild)
        if (p->isactive())
            return 1;

    return 0;
}

void
Classtree.sanity(Classtree *c)
{
    Classtree *p;
    int sum;

    if (c->children == nil) {
        if (c->isactive())
            check c->creq - c->csrv > 0.0,
            "sanity: node active but no csrv left";
        else
            check c->creq - c->csrv <= 0.0,
            "sanity: node idle but has csrv left";
        return;
    }

    sum = 0;
    for (p = c->children; p; p = p->nextchild) {
        p->sanity();
        if (p->isactive())
            sum += p->shares;
    }

    if (sum != c->sashares) {
        fprintf(2, "node %s has sashares %d, should be %d.
"
c->name, c->shares, sum);
    fprintf(2, "first child is %s with %d shares0,
    c->children, c->shares);
}

check sum == c->shares,
    "sanity: incorrect shares value";
}

/*
 * classify.1 -- identify owner of a packet
 */
#include "alef.h"
#include "ppp.h"

enum {
    IP_HDR = 20,
};

int
FQ.classify(FQ *, byte *buf, int len) /* 0..33 */
{
    if (len < IP_HDR+8)
        return 0;
    if (buf[0] != 0x45)
        return 0;
    if (buf[9] == 17) /* udp */
        return 2*(buf[IP_HDR+1]&0x0f); /* 2..17 */
    if (buf[9] == 6) /* tcp */
        return 18+(buf[IP_HDR+1]&0x0f); /* 18..33 */
    return 1;