Interposed Proportional Sharing for a Storage Service Utility[*]

Wei Jin
Department of Computer Science[†]
Duke University
Durham, NC 27708
jin@cs.duke.edu

Jeffrey S. Chase
Department of Computer Science
Duke University
Durham, NC 27708
chase@cs.duke.edu

Jasleen Kaur
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599
jasleen@cs.unc.edu

ABSTRACT

This paper develops and evaluates new share-based scheduling algorithms for differentiated service quality in network services, such as network storage servers. This form of resource control makes it possible to share a server among multiple request flows with probabilistic assurance that each flow receives a specified minimum share of a server's capacity to serve requests. This assurance is important for safe outsourcing of services to shared utilities such as Storage Service Providers.

Our approach interposes share-based request dispatching on the network path between the server and its clients. Two new scheduling algorithms are designed to run within an intermediary (e.g., a network switch), where they enforce fair sharing by throttling request flows and reordering requests; these algorithms are adaptations of Start-time Fair Queuing (SFQ) for servers with a configurable degree of internal concurrency. A third algorithm, Request Windows (RW), bounds the outstanding requests for each flow independently; it is amenable to a decentralized implementation, but may restrict concurrency under light load. The analysis and experimental results show that these new algorithms can enforce shares effectively when the shares are not saturated, and that they provide acceptable performance isolation under saturation. Although the evaluation uses a storage service as an example, interposed request scheduling is non-intrusive and views the server as a black box, so it is useful for complex services with no internal support for differentiated service quality.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Miscellaneous; C.3 [**Special-purpose and Application-based Systems**]: real-time and embedded systems; C.5 [**Computer System Implementation**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, performance

[*]This is a revised version of a paper appearing in SIGMETRICS '04. It clarifies several points, corrects a notational error in the proof of Theorem 3, and includes an improved and expanded discussion of the experimental results and conclusions. This version was released on 5/3/04. This research is supported by the National Science Foundation (EIA-99772879 and ANI-0126231), and by Hewlett-Packard, Network Appliance, and IBM.

[†]Wei Jin's present address: Department of Computer Information Science, Shaw University, Raleigh, NC 27601, *wjin@shawu.edu*.

Keywords

Performance isolation, differentiated service, proportional sharing, fair sharing, weighted fair queuing, multiprocessor scheduling, quality of service, storage services, utility computing

1. INTRODUCTION

Service providers and enterprises are increasingly mapping application workloads onto shared pools of computing and storage resources, such as utility computing environments in data centers. Multiplexing workloads onto shared infrastructure can help to improve resource efficiency and flexibility to adapt to changes in demand and resource status over time.

Resource sharing and consolidation create a need to control how competing customers consume the resources of the service. In these settings, negotiated Service Level Agreements may specify performance targets—such as response time bounds—for specific request classes. The workloads generate *streams* or *flows* of requests for each request class. The service handles the requests of all flows using shared physical resources, such as CPUs, memory, and disks. Thus, shared service infrastructures need effective support for performance isolation and differentiated *application* service quality: without proper scheduling and resource management, a load surge in one flow may unacceptably degrade the performance of another.

One approach to meeting these needs is to control and schedule the individual resources used to serve each flow. However, this approach has a number of limitations as described in Section 2; most importantly it requires support in the server software and/or the hosting platform.

The goal of our work is to support non-intrusive resource control externally to a service, without interfering with its operation in any way. In particular, our solutions are designed for existing services that have no internal support for differentiated service quality, such as commercial storage servers. Our approach is based on *interposed request scheduling*, as depicted in Figure 1. A request scheduler or controller is interposed on the network path between the service and its clients; the scheduler intercepts requests and forwards them to the server, delaying some requests and/or reordering requests from different flows according to some policy. This interposed approach views the server as a black box: in principle, it applies to a wide range of services.

This paper addresses two key questions pertaining to interposed request scheduling. First, *how effectively can an interposed request scheduler support proportional sharing of server resources?* With proportional sharing, the system reserves for each flow some specified minimum share of the server's capacity to handle requests, encoded as a *weight* or service rate. Surplus resources are shared among *active* flows—flows with outstanding requests—in proportion to their shares. Interposed request scheduling for storage has also appeared in Facade [24], which uses Earliest Deadline First (EDF) scheduling, and SLEDS [11], which uses a leaky bucket filter to shape and throttle I/O flows. Proportional sharing combines the advantages of these approaches: like leaky bucket it provides strong isolation under constraint independent of admission control, and like EDF it can use surplus resources to improve service quality for active flows. Section 2.4 discusses these systems and other related work.

Extensive research in scheduling for packet switching networks has yielded a group of algorithms for proportional sharing of network links [18, 21, 7, 8] based on Weighted Fair Queuing and Generalized Processor Sharing [15, 26]. Fair queuing has been adapted to other contexts such as disk scheduling [27], Web server resource management [23], and CPU scheduling [29]. Most fair queuing schedulers are bound tightly to a specific resource, such as a router's outbound link, a disk, or a CPU. This paper proposes two efficient fair queuing variants for interposed request scheduling with a configurable degree of concurrency within the server: depth-controlled SFQ(D) and a refinement called Four-tag Start-time Fair Queuing or FSFQ(D). These new algorithms extend Start-time Fair Queuing or SFQ [21] to approximate proportional sharing of an aggregate resource such as a complex server or even a distributed service. Like Facade, SFQ(D) and FSFQ(D) dispatch at most D outstanding requests to the server at any given time; the depth parameter D is a tradeoff between tight resource control and server resource utilization.

The second question addressed in this paper is: *can interposed proportional sharing be effective in a decentralized system with no central point of control for request scheduling?* Proportional sharing is attractive in a decentralized environment because shares may be partitioned across multiple request sources, with less coordination than would be required for distributed deadline scheduling. We propose and evaluate a distributed request throttling scheme called Request Windows (RW(D)) that bounds the outstanding requests for each flow independently. This idea is similar to the window concept used in TCP rate control.

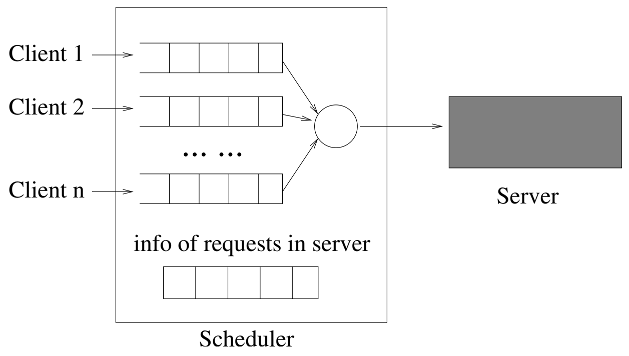


Figure 1: System model. A request scheduler is interposed on the network path between a black-box server and its clients. The scheduler controls server resource usage by intercepting and reordering requests as they flow to the server. The scheduler may reside in a virtualizing server switch.

We evaluate the effectiveness and fairness of SFQ(D), FSFQ(D) and RW(D) both analytically and experimentally using simulations validated by a prototype interposed request scheduler for the Network File Service (NFSv3). The results demonstrate the behavior of the algorithms, and show that they share resources fairly under appropriate values of D and provide acceptable performance isolation under constraint. The experiments are based on a simple storage service configuration and workload; the approach and results can generalize to a wider class of services, but we leave a full sensitivity analysis to future work.

This paper is organized as follows. Section 2 introduces introduces and motivates fair queuing and proportional sharing, and summarizes previous work including Start-time Fair Queuing, the starting point for our approach [21]. Section 3 defines variants of SFQ for interposed request scheduling, and Section 4 outlines the decentralized Request Windows approach. After evaluating the fairness properties of these algorithms analytically, Section 5 presents experimental results. Section 6 concludes.

2. OVERVIEW AND RELATED WORK

Consider any service that is shared by multiple clients: a storage service, an application server, or even a server cluster. Clients generate requests that may use server resources—disk arms, processor capacity, I/O bandwidth—in widely varying ways. Requests are grouped into service classes called *flows*, with service level objectives defined for each flow. (Without loss of generality we speak as if each flow is issued by a separate client, although requests from multiple clients may combine into a single flow, or the requests from a given client may be classified into multiple flows.) In this paper we consider the question: *how can a system control access to the server resources so as to provide predictable and controllable performance across the request flows?*

One way to provide performance isolation is to schedule access to each of the shared resources inside the server in a coordinated way. Many mechanisms and abstractions to provide this resource control (e.g., [6, 30, 17]) are just beginning to appear in general-purpose operating systems, and a new generation of virtual machine monitors supports resource control at a level below the operating system. These systems are designed to allow a service provider to control the resources allocated to each service and each flow or request class, with support from the low-level schedulers for each resource [27, 19, 29, 31]. However, they assume that each request class is served by a separate instance of the service running in a separate virtual machine or resource slice, which inhibits data sharing, or else they require modifications to add support for differentiated service quality into the application. Most importantly, this approach cannot be used for server appliances such as commercial storage servers without quality-of-service features, making performance isolation difficult to achieve for service providers (e.g., Storage Service Providers or SSPs) using these appliances.

2.1 Interposed Request Scheduling

In this paper, we consider a service model illustrated in Figure 1. We treat the service as a black box that is shared by multiple clients, and implement the resource control policy in an interposed request scheduler with no a priori knowledge of the server's internal structure. The scheduler intercepts requests and dispatches them to the server according to its internal policies to share resources fairly among the flows according to their assigned shares. To avoid overloading the server, the scheduler limits the maximum number or aggregate cost of simultaneous outstanding requests to D. It maintains a separate request queue for each flow, and selects the next request to dispatch whenever a request issue slot is open and one or more requests are queued. The scheduler dispatches requests from each flow in FIFO order, but in general we assume that the server may execute requests concurrently and may complete dispatched requests in any order.

The performance received by a client depends on the volume of requests, utilization of the shares, variability of resources within the system, variability of resource demands among the requests, and the effectiveness of the scheduler. Due to the complexity of the systems and the volatility of the workloads, the performance assurances for each flow can be probabilisitic at best [32]. This paper proposes novel algorithms for interposed request scheduling that provide probabilistic assurance that each client receives its *fair share* of the server resources, according to the estimated cost of each request. More precise cost estimates enable more precise resource control.

2.2 Fair Queuing

The proposed algorithms are variants of *fair queuing* [15], a class of algorithms to schedule a shared resource such as a network link or processor among competing flows. Many variants of fair queuing exist for different contexts [18, 21, 7, 8].

Each *flow* f consists of a sequence of requests or packets $p_f^0 ... p_f^n$ arriving at the router or server. Each request p_f^i has an associated *cost* c_f^i. For example, the requests may be packets of varying lengths, or requests for CPU service by threads. Fair queuing allocates the capacity of the resource in proportion to *weights* assigned to the competing flows according to some policy that is outside the scope of this paper. The weights may represent service rates, such as bits or cycles or requests per second. Only the relative values of the weights are significant, but it is convenient to assume that the weights ϕ_f for each flow f sum to the service capacity, or equivalently that they represent percentage shares of service capacity and request costs are normalized to a service capacity of one unit of cost per unit of time.

A flow is *active* if it has one or more outstanding requests, and *backlogged* if it has outstanding requests that have not yet been dispatched. Fair queuing algorithms are *work-conserving*: they schedule requests of active flows to consume surplus resources in proportion to the weights of the active flows. A flow whose requests arrive too slowly to maintain a backlog may forfeit the unconsumed portion of its share, and receive a slower service rate.

Formally, if $W_f(t_1, t_2)$ is the aggregate cost of the requests from flow f served in the time interval $[t_1, t_2]$, then a fair scheduling algorithm guarantees that:

$$\left|\frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g}\right| \leq U_{f,g} \tag{1}$$

where f and g are any two flows continuously backlogged with requests during $[t_1, t_2]$, and $U_{f,g}$ is a constant that depends on the flow and system characteristics. Smaller values of $U_{f,g}$ indicate tighter lag bounds and better fairness for the scheduling algorithm.

Practical fair queuing algorithms are approximations to Generalized Processor Sharing or GPS [15, 26], an idealized reference algorithm that assumes fluid flows served in units of bits or cycles. The practical algorithms consider requests as indivisible, although the per-request cost for each flow is bounded by some maximum cost, length, or quantum c_f^{max}, which affects $U_{f,g}$. Several analyses have derived delay and throughput guarantees when practical scheduling algorithms are used in conjunction with an admission control policy and shaped workloads [28, 20].

Two main characteristics make a service environment different from that of a network router. First, the cost of a request may be influenced by several factors, including cache hits, disk scheduling, etc., and service rates can fluctuate widely. Second, servers can handle multiple requests concurrently. Thus the problem is a form of *multiprocessor* scheduling, which is significantly more complex than multiplexing a single shared resource.

2.3 Start-time Fair Queuing and Virtual Time

We chose Start-time Fair Queuing (SFQ) [21] as the basis for our algorithms in part because it has been shown to be fair even for links or servers with fluctuating service capacity [21]. Like most fair queuing algorithms, SFQ assigns a tag to each request when it arrives, and dispatches requests in increasing order of the tags; ties are broken arbitrarily. The fairness properties of an algorithm result from the way in which tags are computed and assigned to requests of different flows. SFQ assigns two tags for every request: when the j^{th} request p_f^j of flow f arrives, it is assigned a *start tag* $S(p_f^j)$ and a *finish tag* $F(p_f^j)$. The tag values represent the time at which each request should start and complete according to a system notion of *virtual time* $v(t)$. Virtual time always advances monotonically and is identical to real time under ideal conditions: all flows are backlogged, the server completes work at a fixed ideal rate, request costs are accurate, and the weights sum to the service capacity. In practice, virtual time may diverge arbitrarily from real time without compromising fairness if the scheduler dispatches requests in virtual time order. For example, $v(t)$ advances faster than real time whenever surplus resources allow the active flows to receive service at a faster rate than their configured shares would allow.

SFQ assigns tags as follows:

$$S(p_f^j) = \max\{v(A(p_f^j)), F(p_f^{j-1})\}, \ j \geq 1 \tag{2}$$

$$F(p_f^j) = S(p_f^j) + \frac{c_f^j}{\phi_f}, \ j \geq 1 \tag{3}$$

where $A(p_f^j)$ is the actual arrival time of request p_f^j, $F(p_f^0) = 0$, $v(0) = 0$, c_f^j is the cost for the server to execute the request, and ϕ_f is the weight or share for flow f. During a busy period, $v(t)$ is defined to be equal to the start tag of the request in service at time t. When the server is idle, $v(t)$ is defined to be equal to the maximum finish tag of any request that has been serviced by time t. A key advantage of SFQ is that it determines $v(t)$ efficiently.

In [21], the authors derive the fairness guarantee of an SFQ server, stated as the following theorem:

THEOREM 1. *For any interval* $[t_1, t_2]$ *in which flows* f *and* g *are backlogged during the entire interval:*

$$\left|\frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g}\right| \leq \frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g} \tag{4}$$

where $W_f(t_1, t_2)$ *is the aggregate cost of requests from flow* f *served in the interval* $[t_1, t_2]$.

SFQ and other fair queuing algorithms are similar to Virtual Clock [34] in that the request tags for each flow advance according to the progress of that flow. The start tag of the flow's most recent request may be viewed as the flow's virtual clock. Flows with smaller tag values are "behind" and receive priority for service; flows with larger tag values are "ahead" and may be penalized. However, unlike Virtual Clock, the tag values of newly active flows advance to the system-wide virtual clock $v(t)$, so that their newly arriving requests compete fairly with other active flows. This avoids unfairly penalizing active flows for consuming surplus resources left idle by inactive flows.

A server with internal concurrency may have multiple requests in service simultaneously, so $v(t)$ is not well-defined for conventional SFQ in this setting. Moreover, even an active flow may lag behind $v(t)$ if it generates an insufficient number of concurrent requests to consume its assigned share. Section 3 presents and analyzes variants of SFQ to maintain virtual time and assign service tags for interposed request scheduling.

2.4 Other Approaches

Several other systems have used some form of interposed request scheduling to provide service quality assurances for storage arrays. Facade [24] proposes a storage switch that uses Earliest Deadline First (EDF) scheduling to meet response time objectives exposed directly to the scheduler. The key drawback of EDF is that it is unfair: EDF does not isolate request flows from unexpected demand surges by competing flows. Facade assumes effective admission

SYMBOLS	DESCRIPTION
p_f^j	Flow f's j-th request/packet
c_f^j	Cost of request p_f^j
c_f^{max}	Maximum request cost for flow f
ϕ_f	Weight or share for client f
$A(p_f^j)$	Arrival time of request p_f^j
$S(p_f^j)$	Start tag of request p_f^j
$F(p_f^j)$	Finish tag of request p_f^j
$v(t)$	Virtual time at time t
$W_f(t_1, t_2)$	Aggregate work/cost of requests served from flow f during interval $[t_1, t_2]$
$W(S)$	Aggregate cost of requests in set S

Table 1: Some symbols used in the paper.

control and incorporates priority scheduling to isolate each flow from the demands of lower-priority flows. Proportional sharing provides a more general and configurable solution with strong performance isolation. SLEDS [11] is a network adapter for network storage; it uses a leaky bucket filter to shape and throttle I/O flows from each client. The key drawback of the leaky bucket approach is that it is not work-conserving: it throttles flows to their configured service rate even if surplus resources are available to provide better service. This paper proposes proportional sharing algorithms for interposed request scheduling—SFQ(D), FSFQ(D), and Request Windows—that are both fair and work-conserving (for sufficiently large D). SFQ(D) and FSFQ(D) are designed to function in a switch environment similar to Facade, while Request Windows can function on a per-client basis such as SLEDS.

Our approach entails two key resource efficiency tradeoffs relative to other approaches. One advantage of Facade's EDF scheduling is that it schedules directly for response time targets. A fair proportional sharing system can meet service quality targets by sizing the shares under feedback control according to the workload profile and offered load levels [1, 5, 14, 16]. However, this may require conservative overprovisioning of shares, particularly in a storage service setting where per-request costs are difficult to estimate accurately. Second, like Facade and SLEDS, our approach does not control scheduling of requests once they are dispatched to the server. For example, request costs may vary based on the interaction of requests (e.g., due to disk seek), or a request pattern may result in load imbalances within the server, leading to unexpected queuing delays and leaving some resources unnecessarily idle. The alternative approach of integrating service quality targets into the server resource schedulers (e.g., disk schedulers) may use resources more efficiently [27, 10, 22].

Several works extend fair queuing to multiprocessors on the assumption that each flow corresponds to a thread making a sequence of service requests for a CPU [12, 13]. These systems differ from interposed request scheduling, in which multiple requests from a single flow may execute concurrently. Recent work has also derived theoretical properties for fair queuing over aggregated network links [9], and suggested that similar algorithms would be useful for network storage services. Those algorithms order requests by finish tags derived from a simulation of GPS. We use the efficient, practical SFQ algorithm as a starting point.

3. INTERPOSED PROPORTIONAL SHARING

This section develops request scheduling algorithms for proportional sharing of servers that can execute multiple outstanding requests concurrently—including requests from the same flow. Ideally, an interposed request scheduler would dispatch a sufficient number of concurrent requests to fully utilize the resources in the server. However, since the server defines the order in which dispatched requests complete, the scheduler cannot recall or reorder dispatched requests. Thus a competing goal is to delay request dispatching long enough to preserve the scheduler's flexibility to reorder requests and enforce fair sharing.

Our algorithms define a depth parameter D that controls the number of outstanding requests at the server. When a request completes, the scheduler selects the next queued request according to its scheduling policy, and dispatches it to maintain a concurrency level of D within the server. The value of D represents a tradeoff between server resource utilization and scheduler fairness. For instance, a larger D may allow better multiplexing of server resources, but it may impose a higher waiting time on an incoming client request. The policy to configure the D parameter or adapt it dynamically is outside the scope of this paper.

This section derives three scheduling algorithms from SFQ. First, we consider a direct adaptation of SFQ and show why it is unfair. Next, we define a depth-controlled SFQ variant called SFQ(D) and prove its fairness property in terms of a bound on lag given as a function of D. Finally, we present a refinement to SFQ(D), called Four-tag Start-time Fair Queuing or FSFQ(D), which can reduce the lag bound modestly by giving newly active flows preference for their fair share of the D request slots in dispatching requests.

3.1 Min-SFQ(D)

The simplest way to adapt SFQ to interposed request scheduling is to define $v(t)$ to be equal to the minimum start tag assigned to any outstanding request. A request is *outstanding* if it has arrived but has not yet completed, i.e., it is either queued in the scheduler or has been dispatched and occupies one of the D request issue slots. The scheduler may then compute start tags and finish tags of arriving requests using Equations (2) and (3), and dispatch requests in start-tag order as before. Call this scheduling algorithm Minimum-tag Start-time Fair Queuing, or Min-SFQ(D).

To see that MFSQ is not fair, consider the behavior of an active flow f that submits requests just fast enough for each request to arrive before its previous request completes. This flow allows little or no concurrency and may be unable to consume its share of resources, leaving some of those resources idle to be consumed by more aggressive flows. Since f has an outstanding request p_f^j when p_f^{j+1} arrives, $v(t) \le S(p_f^j)$ and its start tag is $S(p_f^{j+1}) = S(p_f^j) + \frac{c_f^j}{\phi_f}$. Thus the virtual clock for flow f advances according to its request arrival rate, and the tags for requests in f may lag arbitrarily behind more aggressive flows. Thus Min-SFQ(D) has the same flaw as Virtual Clock: a burst of requests arriving from f may unfairly penalize more aggressive flows that have obtained higher service rates by consuming resources previously left idle by f.

This example shows that it is no longer safe to assume that an active flow is consuming at least its fair share. More generally, if any flow persistently lags the others, then it holds back the system virtual time $v(t)$ so that the algorithm degrades to Virtual Clock, which is known to be unfair. If a flow lags behind its competitors due to insufficient concurrency, then it is necessary to advance the virtual clock $v(t)$ to catch up with more aggressive flows, to avoid penalizing those flows unfairly. On the other hand, if $v(t)$ advances too rapidly then the scheduler degrades to FIFO, which is also unfair.

3.2 SFQ(D)

One way to advance virtual time more fairly in the presence of lagging flows is to derive $v(t)$ from the progress of backlogged flows rather than lagging flows. The most direct way to do this is to derive $v(t)$ from the start tags of queued requests—requests that have arrived but have not yet been dispatched—without considering requests currently in service. Since the scheduler dispatches any arriving request from a lagging flow at the first opportunity, reducing the window of vulnerability for a lagging flow to dominate virtual time. If the request is dispatched before the flow's next request arrives, then $v(t)$ advances and the lagging flow forfeits its right to recoup any resources it left idle.

We consider a simple variant of this policy, in which $v(t)$ is defined as the start tag of the last request dispatched on or before time t, i.e., the queued request with the lowest start tag at the time of the last dispatch. Arriving requests are assigned start and finish tags according to Equations 2 and 3 as before. This algorithm is a depth-controlled variant of SFQ, referred to as SFQ(D). As in SFQ, virtual time in SFQ(D) advances monotonically on request dispatch events, but may not advance on every dispatch. This is because requests dispatch in order of their start tags, and the start tag of an arriving request $S(p_f^j) \geq v(t)$.

Observe also that the scheduler's request dispatch is driven by completion of a previously issued request, opening up one of the D issue slots. The algorithm used to dispatch requests is exactly SFQ, and thus Theorem 1 applies to define the fairness and bound the lag for requests *dispatched* by SFQ(D). It remains to determine fairness and lag bounds for requests *completed* under SFQ(D), which captures fairness from the perspective of the clients. Theorem 2 derives this bound.

THEOREM 2. *During any interval $[t_1', t_2']$, the difference between the amount of work completed by an SFQ(D) server for two backlogged flows f and g is bounded by:*

$$\left|\frac{W_f(t_1',t_2')}{\phi_f} - \frac{W_g(t_1',t_2')}{\phi_g}\right| \leq (D+1) * \left(\frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g}\right) \quad (5)$$

where ϕ_f and ϕ_g are the weights assigned to flows f and g.

[Proof]: Let the set S be the sequence of requests dispatched during a time interval $[t_1, t_2]$. Suppose S contains $|S| = N_S$ requests. Define t_1' as the earliest time when any request in S completes. Let the set S' be the sequence of N_S requests that complete at time t_1' or later. Define t_2' as the latest completion time for any request from S'. For any interval $[t_1', t_2']$ there is some corresponding interval $[t_1, t_2]$ defined in this way.

S consists of three disjoint subsets: $S = S_f + S_g + S_o$, where S_f contains the requests from flow f, S_g contains the requests from flow g, and S_o contains all requests in S that are not from flows f or g.

Consider the set of requests that were dispatched during $[t_1, t_2]$ and completed during $[t_1', t_2']$. This set $S' \cap S$ consists of three disjoint subsets: $S_f' = S' \cap S_f$, $S_g' = S' \cap S_g$ and $S_o' = S' \cap S_o$. Therefore, the set $S_f - S_f'$ represents the set of client f requests dispatched but not yet completed at time t_2', and similarly for S_g' and S_o'.

Now consider the set S_{resid} of requests dispatched during $[t_1, t_2]$ but still in service after time t_2': $S_{resid} = S - (S_f' + S_g' + S_o') = (S_f + S_g + S_o) - (S_f' + S_g' + S_o') = (S_f - S_f') + (S_g - S_g') + (S_o - S_o')$. Since there are at most D requests in service at any time, $|S_{resid}| \leq D$. Then trivially

$$|S_f - S_f'| \leq D$$

and

$$|S_g - S_g'| \leq D$$

Therefore

$$0 \leq W(S_f) - W(S_f') \leq D * c_f^{max} \quad (6)$$

$$0 \leq W(S_g) - W(S_g') \leq D * c_g^{max} \quad (7)$$

Of the N_S requests S' that completed service during $[t_1', t_2']$, consider the subset $S'_{straggle}$ that are not in S, i.e., they were dispatched before t_1 or perhaps after t_2. Note that $S' = (S_f' + S_g' + S_o') + S'_{straggle}$. $S'_{straggle}$ can be further decomposed into three disjoint sets $S'_{straggle} = S_f'' + S_g'' + S_o''$, where S_f'' is the set of requests from flow f, S_g'' from g and S_o'' otherwise. Since $S = (S_f' + S_g' + S_o') + S_{resid}$, $S' = (S_f' + S_g' + S_o') + S'_{straggle}$, and $|S'| = |S|$, we have $|S'_{straggle}| = |S_{resid}| \leq D$. Therefore,

$$|S_f''| \leq D \Rightarrow 0 \leq W(S_f'') \leq D * c_f^{max} \quad (8)$$

$$|S_g''| \leq D \Rightarrow 0 \leq W(S_g'') \leq D * c_g^{max} \quad (9)$$

From Equation (6) to (9), we have:

$$|(W(S_f) - W(S_f')) - W(S_f'')| \leq \quad (10)$$
$$D * c_f^{max} \quad (11)$$

and

$$|(W(S_g) - W(S_g')) - W(S_g'')| \leq \quad (12)$$
$$D * c_g^{max} \quad (13)$$

Note that $W_f(t_1', t_2') = W(S_f') + W(S_f'')$, and $W_g(t_1', t_2') = W(S_g') + W(S_g'')$. Therefore

$$\left|\frac{W(S_f)}{\phi_f} - \frac{W_f(t_1',t_2')}{\phi_f}\right| \leq D * \frac{c_f^{max}}{\phi_f} \quad (14)$$

$$\left|\frac{W(S_g)}{\phi_g} - \frac{W_g(t_1',t_2')}{\phi_g}\right| \leq D * \frac{c_g^{max}}{\phi_g} \quad (15)$$

Using $|A| - |B| \leq |A - B| \leq |A| + |B|$, we have

$$\left|\frac{W_f(t_1',t_2')}{\phi_f} - \frac{W_g(t_1',t_2)}{\phi_g}\right| - \quad (16)$$

$$\left|\frac{W(S_f)}{\phi_f} - \frac{W(S_g)}{\phi_g}\right| \quad (17)$$

$$\leq \quad D * \frac{c_f^{max}}{\phi_f} + D * \frac{c_g^{max}}{\phi_g} \quad (18)$$

$$(19)$$

According to Theorem 1,

$$\left|\frac{W(S_f)}{\phi_f} - \frac{W(S_g)}{\phi_g}\right| \leq \frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g} \quad (20)$$

Therefore,

$$\left|\frac{W_f(t_1',t_2')}{\phi_f} - \frac{W_g(t_1',t_2')}{\phi_g}\right| \quad (21)$$

$$\leq \quad (D+1) * \left(\frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g}\right) \quad (22)$$

□

3.3 Four-tag Start-time Fair Queuing (FSFQ(D))

With interposed request scheduling, an arriving request p_f^j may have up to D requests in service ahead of it. In some cases, a fair policy such as SFQ would have ordered some or all of those requests after p_f^j, but since they arrived earlier, the SFQ(D) scheduler dispatched them into free issue slots, and it is too late to recall them. In the worst case, D exceeds the available concurrency in the server, so p_f^j must wait for requests queued ahead of it at the server to complete before it can receive service. Moreover, a request from another flow may arrive, receive the same start tag, and issue first. Min-SFQ(D) does not suffer from this drawback.

We refine SFQ(D) to compensate a late-arriving flow by favoring it over other flows that hold more than their share of the D issue slots. Our approach—called Four-tag Start-time Fair Queuing (FSFQ(D))—combines the benefits of SFQ(D) and Min-SFQ(D).

FSFQ(D) associates two pairs of tags with each request. One pair is similar to the start and finish tag in SFQ(D). The other pair are the *adjusted start tag*, $\overline{S}$, and *adjusted finish tag*, $\overline{F}$, similar to the tag definition in Min-SFQ(D). Requests are scheduled in the increasing order of the start tags of the requests. Ties are broken according to the adjusted start tags.

In calculating the tags, two functions—the virtual time, $v(t)$, and the adjusted virtual time, $\overline{v}(t)$—are used. $v(t)$ is defined as the start tag of the last dispatched request, similar to the virtual time definition in SFQ(D). $\overline{V}(t)$ is defined as the minimum start tag of the requests in the system, similar to the virtual time definition in Min-SFQ(D). The four tags are calculated the following way:

If $F(p_f^{j-1}) < v(t)$, then

$$\overline{S}(p_f^j) = \max\{\overline{v}(A(p_f^j)), F(p_f^{j-1})\} \tag{23}$$

else:

$$\overline{S}(p_f^j) = \max\{\overline{v}(A(p_f^j)), \overline{F}(p_f^{j-1})\} \tag{24}$$

$$\overline{F}(p_f^j) = \overline{S}(p_f^j) + \frac{c_f^j}{\phi_f} \tag{25}$$

$$S(p_f^j) = \max\{v(A(p_f^j)), \overline{S}(p_f^j)\} \tag{26}$$

$$F(p_f^j) = S(p_f^j) + \frac{c_f^j}{\phi_f} \tag{27}$$

where $A(p_f^j)$ is the arrival time of request p_f^j, $F(p_f^0) = \overline{F}(p_f^0) = 0$, c_f^j is the request cost, and ϕ_f is the weight for flow f.

Initially $v(0)$ and $\overline{v}(0)$ are both 0. During a busy period, $v(t)$ is defined as the start tag of the last dispatched request. $\overline{v}(t)$ is defined as the minimum start tag of all outstanding requests. At the end of a busy period, $v(t)$ and $\overline{v}(t)$ are set to the maximum finish tag assigned to any request that has been serviced by time t.

When a request p_f^j arrives at time t, the scheduler checks if $F(p_f^{j-1}) < v(t)$. If so, then some requests from other flows have been dispatched with start tags greater than $F(p_f^{j-1})$, but have not yet completed. SFQ would serve the newly arrived request first, but it is too late. FSFQ(D) compensates f by giving it extra credits to make up for its slight late arrival. The rules above achieve this by assigning the start tags for p_f^j as the following:

$$\overline{S}(p_f^j) = \max\{\overline{v}(t), F(p_f^{j-1})\} \tag{28}$$

$$S(p_f^j) = \max\{v(t)), \overline{S}(p_f^j)\} = v(t) \tag{29}$$

The flow f receives surplus credits $S(p_f^j) - \overline{S}(p_f^j) = v(t) - \max\{\overline{v}(t), F(p_f^{j-1})\}$ to be used in breaking ties. Note that $F(p_f^j) = S(p_f^j) + \frac{c_f^j}{\phi_f} = v(t) + \frac{c_f^j}{\phi_f} > v(t)$. Therefore, only the first newly arrived request for f uses formula (23); subsequent requests use formula (24) until f becomes inactive.

For any request p_f^k arriving right after p_f^j, we can see that $S(p_f^k)$ and $F(p_f^k)$ are unchanged as long as $\overline{S}(p_f^j)$ has not caught up with the virtual time v (see formula (26)). However, $\overline{S}(p_f^k)$ and $\overline{F}(p_f^k)$ gradually increase with subsequent requests, using formula (24) and (25), until all the credits are used, that is, when $\overline{S}(p_f^k) = S(p_f^k) = v(A(p_f^k))$. At this point, we already have $\overline{F}(p_f^k) > v(A(p_f^k))$. After this $\overline{S}(p_f^k) > v(t)$, so using formula (26), we can see that $S(p_f^k) = \overline{S}(p_f^k)$.

4. REQUEST WINDOWS

The SFQ(D) and FSFQ(D) scheduling algorithms proposed in Section 3 require **(i)** per-client state variables (four tags, in case of FSFQ), and **(ii)** a priority queue to select the request with the smallest start tag. The scheduling overhead grows at best logarithmically in the number of flows, limiting the scalability of these mechanisms. Moreover, a full SFQ(D) or FSFQ(D) must interpose at a central point and intercept all requests from all flows. In this section, we ask the question: *what isolation guarantees are possible for a simple decentralized solution that throttles each flow independently of the others?*

We propose and analyze a simple credit-based server access scheme called *Request Windows* (RW). RW allocates a specified number of *credits*, n_f, to each flow f. An independent scheduler is interposed on each flow, e.g., residing at the clients. Each dispatched request p_f^j occupies a portion of the flow's credit allocation equal to the request's estimated cost c_f^j. We define the policy RW(D) by setting $n_i = D\phi_i$ for the case where all shares ϕ_i sum to one. That is, requests with total cost at most D may be outstanding from all flows, with each flow's share determining the portion of the total allowable outstanding work allocated to that client. Arbitrary weights may be used: in the general case $n_i = D\frac{\phi_i}{\sum_j \phi_j}$.

Request Windows can be implemented in a decentralized fashion with minimal per-flow state and without priority queuing. RW is closely related to request-based flow control in the Direct Access File System [25]; they are equivalent if requests have uniform cost. Note that if inactive flows leave surplus resources, then requests from an flow encounter less congestion at the server and complete at a faster rate, opening issue slots and allowing the flow to dispatch its requests at a faster rate. This is similar to the self-clocking behavior of protocols with window-based rate control, such as TCP. Even so, RW is not fully work-conserving for low values of D. The primary drawback of Request Windows is that it throttles each flow without knowledge of whether other flows are active; RW yields a tighter fairness bound, but it may limit the concurrency at the server under light load and/or limit an active flow's use of surplus resources.

THEOREM 3. *Consider an idealized FIFO server under RW(D). The difference between the amount of work completed by the server for two backlogged flows f and g during any interval $[t_1', t_2']$ is bounded by:*

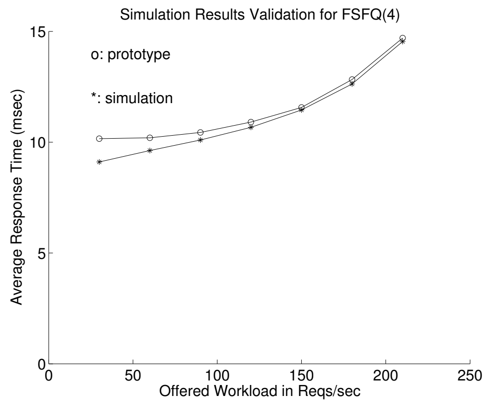
$$\left|\frac{W_f(t_1', t_2')}{\phi_f} - \frac{W_g(t_1', t_2')}{\phi_g}\right| \leq 2D \tag{30}$$

[Proof]: Let the sequence of requests sent to the server during a time interval $[t_1, t_2]$ be S, and let the total number of credits $W(S)$ consumed by S be N_S. Time t_1' is defined as the earliest time that any request in S completes. Let the set S' be the sequence of requests that complete at time t_1' or later, such that the total number of credits $W(S')$ consumed by S' is also N_S. Time t_2' is defined to be the latest time that any request in S' completes.

Assume that at time t_1 the total number of credits consumed by the outstanding client f requests is n_f', and n_g' for client g. Obviously $n_f' \leq n_f$, and $n_g' \leq n_g$.

Let the total number of credits of client f requests dispatched during $[t_1, t_2]$ be N_f. We can rewrite N_f as $N_f = (n_f - n_f') + x * n_f + n_f''$, where $n_f'' \leq n_f$. According to the way RW operates with a FIFO server, the number of client g requests sent to the server can be represented as $N_g = (n_g - n_g') + x * n_g + n_g''$, where $n_g'' \leq n_g$. Therefore,

$$W_f(t_1, t_2) = N_f = (x+1) * n_f - n_f' - n_f''$$
$$W_g(t_1, t_2) = N_g = (x+1) * n_g - n_g' - n_g''$$



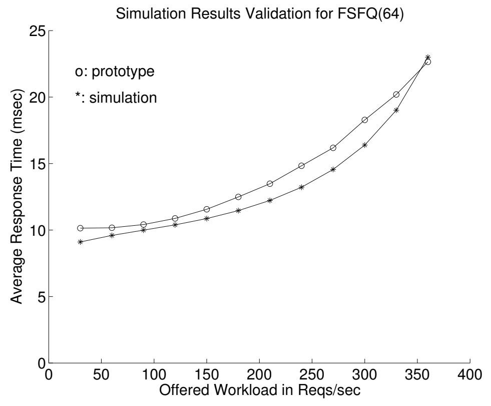


Figure 2: Validation of simulation results. In these validation experiments, we use two clients with equivalent weights. The x-axis shows the aggregate arrival rate for both clients.

Or

$$\frac{W_f(t_1,t_2)}{n_f} = (x+1) - \frac{n'_f + n''_f}{n_f} \tag{31}$$

$$\frac{W_g(t_1,t_2)}{n_g} = (x+1) - \frac{n'_g + n''_g}{n_g} \tag{32}$$

Therefore,

$$|\frac{W_f(t_1,t_2)}{n_f} - \frac{W_g(t_1,t_2)}{n_g}| = |\frac{n'_f + n''_f}{n_f} - \frac{n'_g + n''_g}{n_g}| \leq 2 \tag{33}$$

Since the server is FIFO,

$$W_f(t'_1,t'_2) = W_f(t_1,t_2) \tag{34}$$

And similarly,

$$W_g(t'_1,t'_2) = W_g(t_1,t_2) \tag{35}$$

Therefore,

$$|\frac{W_f(t'_1,t'_2)}{n_f} - \frac{W_g(t'_1,t'_2)}{n_g}| \leq 2 \tag{36}$$

Multiplying through by D proves the theorem. []

5. EXPERIMENTAL EVALUATION

The share-based algorithms discussed in this paper are intended to assure a proportional fair share of the server capacity for competing flows in proportion to their weights or shares. In this section, we evaluate and compare the performance isolation properties of these algorithms experimentally for a representative synthetic workload and storage server configuration. Our approach is to compare the response times for two competing request flows—client 1 and client 2—for varying weights (shares), request rates, and values of D.

Prototype and Simulation Environment

To evaluate our ideas, we have implemented a prototype interposed request scheduler for the Network File System protocol by extending an existing NFS proxy [3, 33] with about 1000 lines of code. The proxy maintains per-client queues for each server and uses SFQ(D) or FSFQ(D) to schedule requests, given a statically specified D parameter and shares associated with client IP addresses.

For ease of experimentation, most of the results presented in this section are obtained not from the prototype, but from experiments run in a simulated server environment with support for different client request arrival patterns. Requests arrive as an ON/OFF process with poisson arrivals during ON times. Each server is modeled as consisting of multiple service components (e.g., multiple hard drives). The request service time for any component is modeled as an exponential distribution; this model approximates the behavior of random access storage workloads.

We validated our simulation environment by comparing representative simulation results with experimental results from the prototype. Figure 2 presents one such validation result for the response times observed by two clients, with FSFQ(4) and FSFQ(64) controllers. The response times observed in the simulator and the prototype match closely. The proxy and NFS server ran on Dell 4400 servers running FreeBSD; the storage server uses a RAID on a concatenated disk (CCD) stripe consisting of six Seagate Cheetah drives. We used the fstress load generator [2] to generate a variable workload dominated by random reads on large files. Requests are assumed to have equivalent cost.

5.1 EDF vs. Share-based Scheduling

We first demonstrate the value of interposed proportional sharing for performance isolation under overload, relative to an interposed Earliest Deadline First (EDF) scheduler. This experiment simulates a setting in which client 1 has a constant demanding workload of 480 IO operations per second (IOPS or IOs/s), which is almost enough to saturate the server. Client 2 has an ON/OFF workload that is ON for 10 s with a constant 120 IOs/s and OFF for 10 s. Client 1 has a loose response time target of 1 second, while the target for client 2 is 10 ms.

Figure 3 plots the throughput and average response times observed by both clients for a two-minute simulation with an EDF scheduler. Requests from client 1 are queued and miss their deadlines, and are scheduled ahead of new requests from client 2, eventhough client 2 has a tighter response time target. The response times of both clients increase steadily. In practice, EDF schedulers depend on other mechanisms—such as priority or admission control—for performance isolation. Proportional sharing, however, does provide isolation, as shown in Figure 4, which plots results from similar experiments using FSFQ and Request Windows (RW) schedulers, with client 2 configured for a 33% share of the server. Client 1 exceeds its 67% share of the server; when client 2 demands service the response time for client 1 grows without bound, while the response time for client 2 remains stable at around 18 ms.

5.2 Differentiated Response Time

The next experiments investigate the effectiveness of interposed proportional sharing in differentiating the service quality received

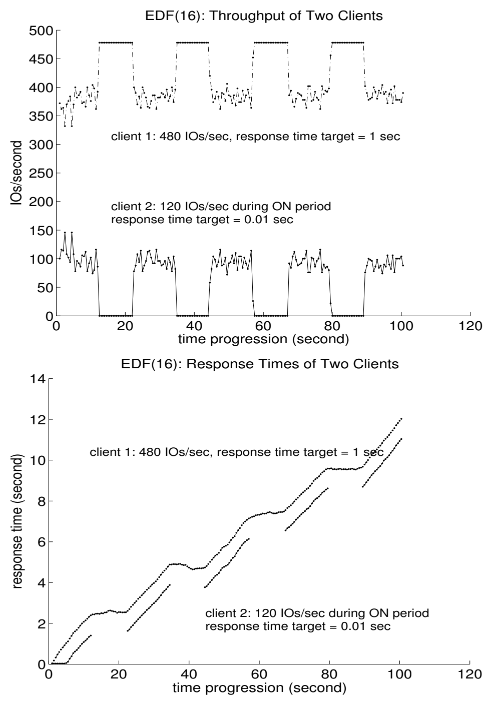


Figure 3: An EDF scheduler alone does not provide performance isolation when the server is overloaded. Client 1's high arrival rate causes client 2's response time to increase without bound.

by competing clients according to their configured shares. We fix client 1's arrival rate, but increase client 2's arrival rate, so that the sum of client 1 and client 2's arrival rate gradually approaches the system saturation rate. The goal is to understand the effect of share size on response time under resource competition, and in particular to illustrate the degree to which interposed proportional sharing can provide response time assurances for client 1 in spite of client 2's climbing rate. For these experiments we set $D = 8$.

5.2.1 FSFQ(D) vs. SFQ(D)

For each curve plotted in Figure 5, we fix client 1's rate (to either 30 or 240 requests/second) and the ratio $w_1 : w_2$ (2:1 or 8:1), and increase the arrival rate for client 2. We observe the following:

1. Client 2's increasing rate degrades response time for both clients. However, client 1's response time eventually stabilizes in all cases, while client 2's response time grows without bound as expected. This shows that both SFQ(D) and FSFQ(D) provide performance isolation: client 2's increasing rate has limited impact on client 1's performance. The impact on client 1 occurs in large measure because client 1 receives *better* than its configured service when client 2 has a low arrival rate, leaving surplus resources for use by client 1. As competition from client 2 increases, there is less surplus resource available for client 1; its response time initially degrades but ultimately stabilizes at a level that correlates with client 1's utilization of its configured share.

2. From the first pair of graphs in Figure 5, we can see that when client 1's arrival rate is low (30 IOs/sec), its response time is relatively insensitive to weight assignment. This is because requests from a flow that is not backlogged always receive priority when they arrive—independent of weight—but each request is scheduled behind up to D competing requests from

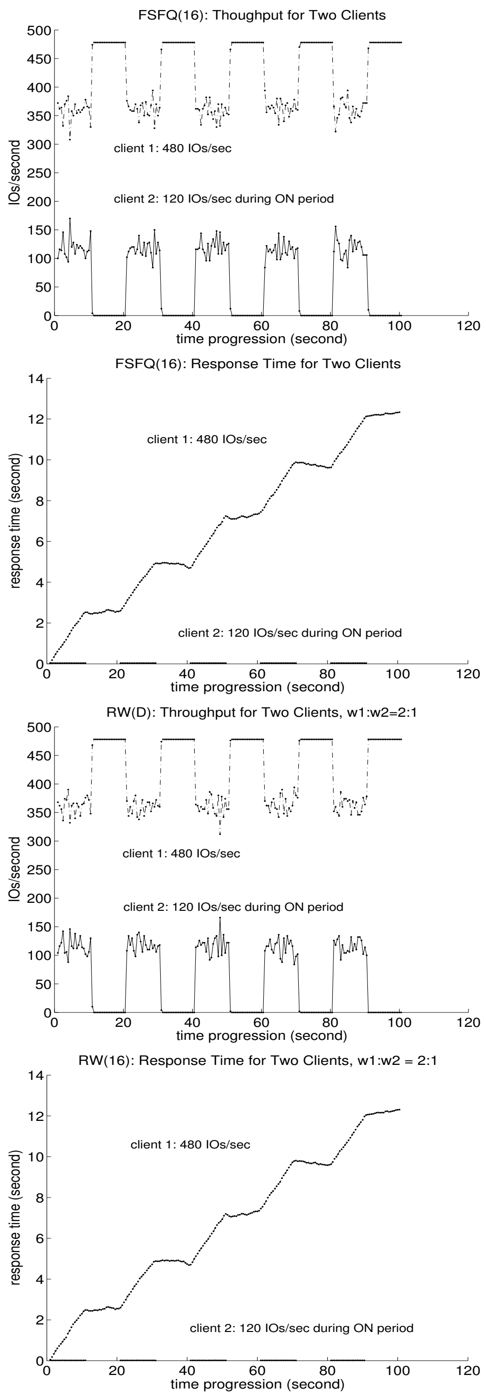
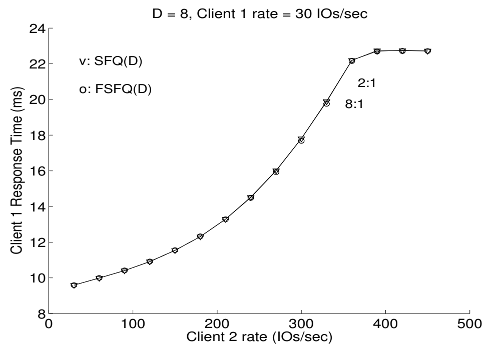
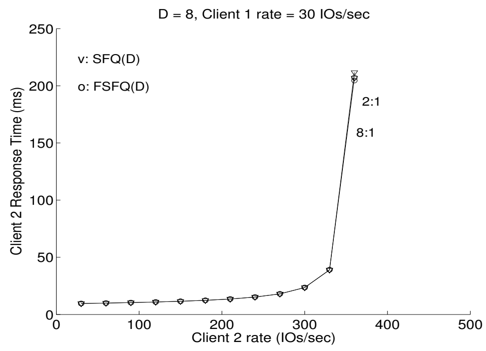
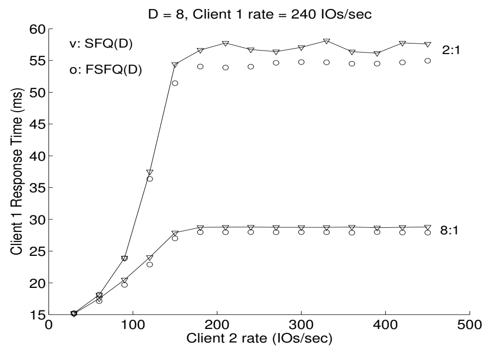


Figure 4: This is the same experiment as Figure 3, but using FSFQ(D) and RW(D) schedulers with client 2 configured for a 33% share of the server. Both FSFQ(D) and RW(D) can provide performance isolation: while client 1 exceeds its share and is penalized, client 2 receives service with a stable response time.







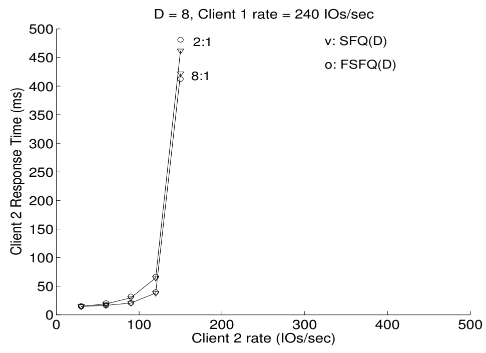


Figure 5: Response times under SFQ(D) and FSFQ(D) as load from client 2 increases along the x-axis. Each pair of graphs shows response times for each client when client 1 receives either a 67% (2:1) or 88% (8:1) share—note that the y-axes are scaled differently. All graphs show the expected behavior: client 2's response time degrades without bound as it exceeds its share, while fair sharing isolates client 1. Client 1 has light load in the top pair, but in the bottom pair it requests about 50% of the server capacity: this leaves less surplus resource for client 2, causing client 2's response time to spike at a lower arrival rate. In all cases client 1's response time stabilizes at a level related to its assigned share. The FSFQ refinement improves client 1's response time modestly when its share utilization is high.

backlogged flows. In contrast, the second pair of graphs in Figure 5 shows that when client 1's rate is relatively high (e.g. 240 IOs/sec), its response time is more sensitive to the weight assignment. In these cases, client 1 is often backlogged, and its response time is determined largely by the resource share assigned to it, as given by its weight.

3. FSFQ(D) offers modestly better isolation than SFQ(D). When client 2 is persistently backlogged, the FSFQ scheduler always issues client 1's requests ahead of any queued requests from client 2 that have the same start time tag. As a result, FSFQ(D) achieves better response time for client 1 than SFQ(D).

5.2.2 *FSFQ(D) vs. RW(D)*

Figure 6 presents results from similar experiments comparing FSFQ(D) and RW(D) with $D = 32$. The key result from this graph is that RW(D) isolates more effectively than even FSFQ(D), in that it yields lower response times for client 1 even under heavy competition from client 2. However, this advantage comes at a cost: RW limits the ability of client 2 to use surplus resources left idle by client 1. This can be seen from the bottom graph of each pair, which shows that the response time of client 2 increases at lower load levels with RW than with FSFQ, and that it is sensitive to the size of client 2's share. When client 2 has a smaller share, its response time increases rapidly even when client 1 leaves surplus resources that could serve client 2's increasing load. RW does make some use of these surplus resources: this can be seen by comparing the RW lines in the bottom graphs in each pair, which show that client 2 receives better service when client 1 has lighter load. Even so, client 2's response time is always better under FSFQ because FSFQ is fully work-conserving: FSFQ is more effective at using resources left idle by client 1. This experiment suggests that a hybrid of FSFQ and RW can balance the competing goals of fairness and resource efficiency (work conservation), combining elements of proportional sharing and reservations.

5.3 Impact of Depth D

By comparing the FSFQ(32) results Figure 6 with the FSFQ(8) results in Figure 5, we can see that FSFQ delivers weaker performance isolation as the depth parameter D increases. When there is competition for resources, higher D values increase average response time because each issued request may wait behind as many as D previously issued requests before it receives service. We now explore the impact of D on response time in more detail under FSFQ and Request Windows.

Figure 7 plots the average response times for both clients under FSFQ and RW with different values of D. As in the previous experiments, client 1 generates a fixed request rate (30 IOs/sec in the top pair of graphs and 240 IOs/sec in the bottom pair of graphs), while the arrival rate for client 2 increases along the x-axis. These experiments fix the weights of clients 1 and 2 to $w_1 : w_2 = 2 : 1$, i.e., client 1 is assigned a 67% share.

Figure 7 confirms that increasing the depth D weakens fairness for both RW and FSFQ. Weaker enforcement of the shares has the effect of increasing client 1's response time under competition from client 2, and reducing client 2's response time even at high load levels. In essence, higher D allows client 2 to steal resources unfairly from client 1, particularly when client 1's load is light. Lower values of D offer tighter resource control. However, if D is too low, then the service may have inadequate concurrency to use all of its resources efficiently. For example, the third graph shows that the response time for client 1 at 240 IO/s spikes at $D = 4$ because the system has inadequate concurrency to keep all of its disks busy, even as the scheduler delays issue of requests that could be served by the idle disks.

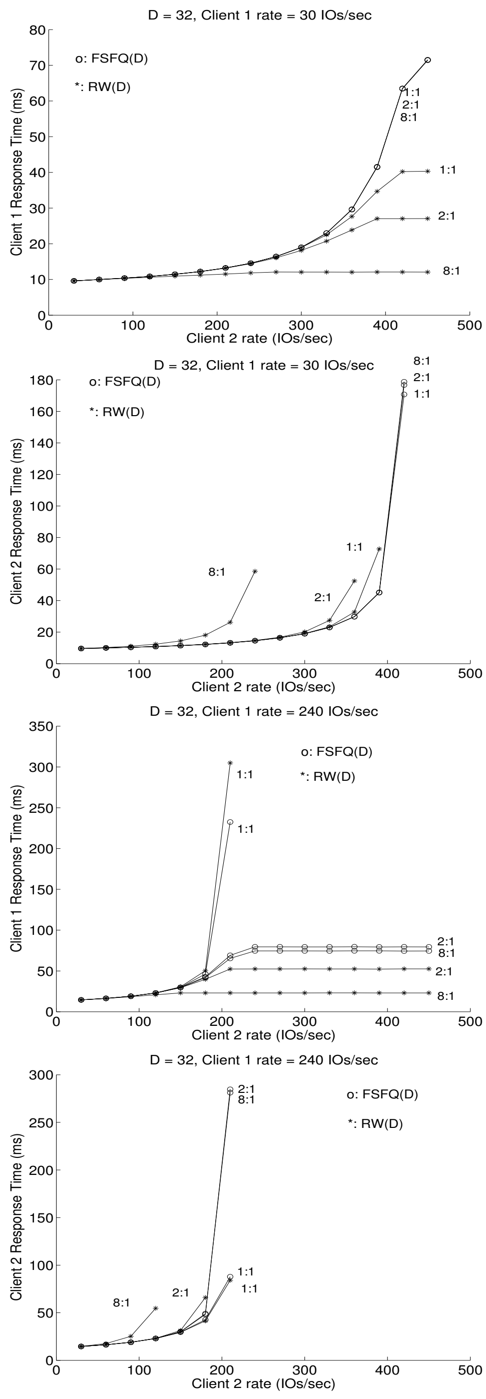


Figure 6: These graphs are from experiments similar to Figure 5, but compare FSFQ(D) and RW(D) for varying share sizes (weights) with a depth of $D = 32$. As in Figure 5, increasing load from client 2 along the x-axis causes its response time to increase without bound, while client 1's response time increases to a stable level related to the size of its share. The top graph shows that FSFQ(32) does not protect client 1 as effectively as FSFQ(8) in Figure 5. RW(D) always isolates more effectively than FSFQ(D) because it limits the ability of client 2 to consume issue slots left idle by client 1.

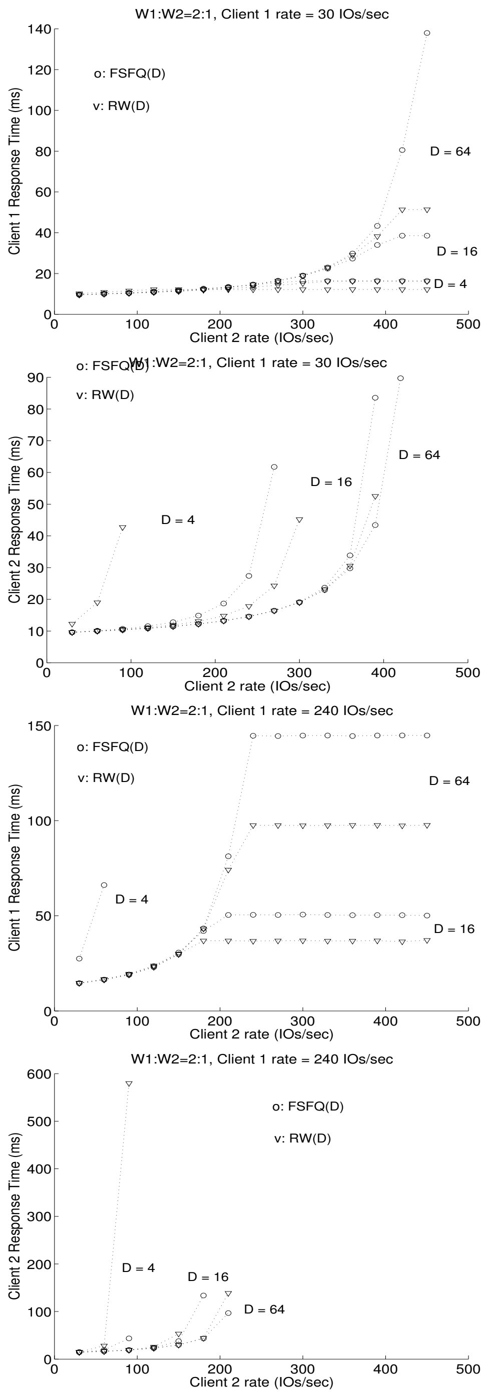


Figure 7: These graphs compare the average response times under FSFQ(D) and RW(D) for different values of D. These experiments are similar to Figure 6, except that the weights are constant at 2:1 while D varies. Increasing D may use resources more efficiently, but it weakens resource control (fairness). This weakened control is shown by the higher response times for client 1 under competition from client 2, and the lower response times for client 2 when its request rate exceeds its share. Note also that RW always offers tighter control than FSFQ—as shown by RW's lower response times relative to FSFQ for client 1 when client 2's load is high—but that RW also uses surplus resources less aggressively—as shown by RW's higher response times for client 2 when client 1's load is low.

Figure 7 also further illustrates the tradeoff of resource control vs. resource efficiency for RW and FSFQ. RW offers tighter resource control and improved fairness at every value of D, because it never permits any workload to consume more than its share of issue slots. However, RW is less effective than FSFQ at using surplus resources, i.e., it is not fully work-conserving. As result, client 2 always suffers higher response times under RW than FSFQ; while this is the desired result when client 2's load is high, it indicates that RW is not using server resources efficiently when the combined load is below the peak service rate.

5.4 Meeting Response Time Targets

A key premise of our approach is that proportional share scheduling allows an external policy to meet response time targets by controlling the share sizes. Indeed, the previous experiments illustrate how larger shares yield better response times under resource competition, when there are no surplus resources to exploit.

Figure 8 presents a simple example of how share-based scheduling algorithms, such as FSFQ(D) and RW(D), can be used to meet response time targets in this way. In this experiment client 1 generates a stable heavy load of requests with a response time target of 1 second, while client 2 generates bursts of requests with a much tighter response time bound of 10 milliseconds. The graph shows the expected behavior: bursts of requests from client 2 cause client 1's response time to degrade, but both request flows stay within their response time bound.

To achieve this result, it is necessary to set the depth D and weight w_i parameters appropriately. Section 5.3 demonstrates the importance of choosing a depth D that balances resource efficiency and resource control. For this experiment we used the approach proposed to adjust D for Facade [24], an interposed deadline scheduler for response time targets. The policy uses a feedback controller to converge on the maximum depth that meets the response time targets. The share sizes for this experiment are set statically using a rule of from queuing theory: response time is inversely proportional to the idle time of the resources assigned to a request flow. The idle time grows linearly with share size. For this experiment we simply set each w_i to the inverse of its target response time, although in practice it may be useful to adjust share sizes as load changes. We leave a full investigation of the role of share size and dept D on response time to future work.

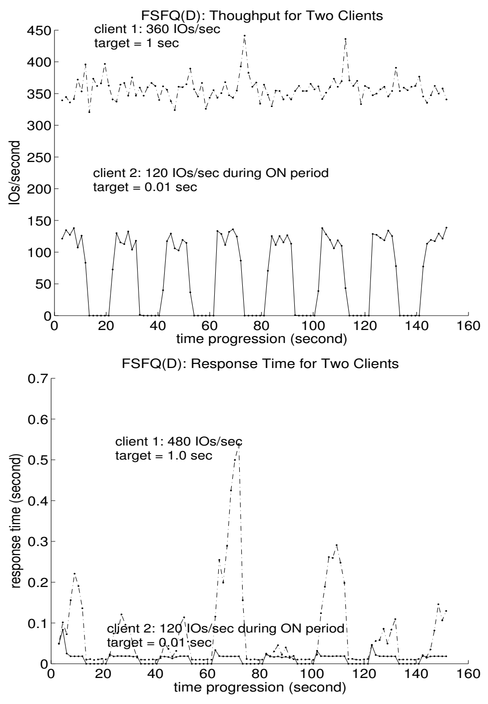


Figure 8: By adjusting weight assignment and D, we can use FSFQ(D) to satisfy both clients' response time targets.

6. CONCLUSION

This paper proposes an approach to proportional share resource control in shared services by interposed request scheduling. The approach is *non-intrusive* in the sense that it applies to network services with no internal support for fair sharing or differentiated service quality, such as commercial storage servers. The sole means to coordinate resource sharing is to control the entry of requests into the service, and reorder or delay them according to the sharing policy. We propose three proportional share scheduling algorithms, prove their fairness properties analytically, and present simulation results to illustrate their fairness behavior and efficiency under a selected workload. All of these algorithms are promising for use in practice to meet service quality targets for shared servers.

The first two algorithms extend a previous fair queuing algorithm, start-time fair queuing or SFQ, to *depth-controlled* variants SFQ(D) and FSFQ(D) that are applicable to shared services with internal concurrency. These algorithms are efficient (work-conserving) and they are fair over sufficient time scales, but they have two drawbacks. First, they require a central point of control for all requests entering the service. Second, their behavior exposes a tension between server efficiency and response time. Specifically, they release up to D concurrent requests into the service; the depth D defines a tradeoff between fairness and the degree of concurrency allowed within the server. Larger values of D allow more flexibility to the server to schedule its resources, but they also increase response time, particularly for request flows that have low utilization of their assigned resource shares. Thus these algorithms are not able to meet tight response time targets for services with a high degree of internal concurrency. Note also that internal load imbalances within the service may need more concurrency (greater depth) to avoid head-of-line blocking behind requests awaiting the bottleneck resources. To schedule for tight response time targets under these conditions it may be necessary to integrate some support for deadlines or priority scheduling into the service itself (e.g., [22]).

The third algorithm—Request Windows (RW(D))— throttles each flow independently to a configured share of the maximum total request cost permitted for release into the service at any time. The Request Windows scheme may be deployed in a decentralized fashion, and it supports tighter bounds on response time. However, unless the throttling of different flows is coordinated in some way, Request Windows is not work-conserving: it may throttle requests even if surplus resources are available to serve those requests. In this respect Request Windows behaves more like a share-based reservation scheme rather than a true proportional sharing algorithm that distributes surplus resources among the active flows in proportion to their weights.

Our results suggest several directions for future research. First, any form of share-based scheduling requires cost estimates for the requests; sharing is fair only with respect to these estimated costs. It is possible to adjust these costs to reflect observed imbalances within the server or interference among flows, as in Stonehenge [22] or Hippodrome [4]. Second, we have not investigated the behavior of interposed request scheduling under dynamic policies for adjusting depth, share sizes, or throttling thresholds (n_i for Request Windows) under feedback control, which is likely to be important in practice. Finally, our results indicate the potential of hybrid policies for interposed request scheduling that combine elements of FSFQ(D) and Request Windows. Hybrid policies would accommodate varying degrees of decentralization. They may also allow a configurable balance between proportional sharing and reservations, trading off work conservation for tighter response time bounds.

Acknowledgments

Ken Yocum and Darrell Anderson provided the NFS proxy and fstress load generator, and David Becker assisted with experimental setup. We thank John Wilkes and Guillermo Alvarez for early discussions on the idea of interposed request scheduling, and John Douceur, Magnus Karlsson, Christos Karamanolis, John Wilkes, Ken Yocum, and the anonymous reviewers for comments that helped us to improve the paper.

7. REFERENCES

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, January 2002.

[2] Darrell C. Anderson and Jeffrey S. Chase. Fstress: A flexible network file service benchmark. Technical Report CS-2002-01, Duke University, Department of Computer Science, January 2002.

[3] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems (TOCS) special issue: selected papers from the Fourth Symposium on Operating System Design and Implementation (OSDI), October 2000*, 20(1), February 2002.

[4] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the First Usenix Conference on File and Storage Technologies (FAST)*, January 2002.

[5] Mohit Aron. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Department of Computer Science, Rice University, October 2000.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[7] Jon C. R. Bennett and Hui Zhang. WF^2Q: Worst-case fair weighted fair queuing. In *Proceedings of IEEE INFOCOM '96*, San Francisco, CA, March 1996.

[8] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[9] Josep M. Blanquer and Banu Ozden. Fair queuing for aggregated multiple links. In *ACM SIGCOMM*, August 2001.

[10] John Bruno, Jose Brustoloni, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, June 1999.

[11] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, and Tzongyu P. Lee Jian Xu, Ram Menon. Performance virtualization for large-scale storage systems. In *22nd International Symposium on Reliable Distributed Systems (SRDS '03)*, October 2003.

[12] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[13] Abhishek Chandra, Micah Adler, and Prashant Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, May 2001.

[14] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doy le. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[15] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. *ACM SIGCOMM 89*, 19(4):2–12, August 19-22, 1989.

[16] Ron Doyle, Jeffrey S. Chase, Omer Asad, Wei Jin, and Amin Vahdat. Model-based resource provisioning in a Web service utility. In *Proceedings of the Fourth Symposium on Internet Technologies and Systems (USITS)*, Seattle, Washington, USA, March 2003.

[17] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[18] S. Jamal Golestani. A self-clocked fair queuing scheme for broadband applications. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking for Global Communication. Volume 2*, pages 636–646, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[19] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[20] Pawan Goyal and Harrick M. Vin. Generalized guaranteed rate scheduling algorithms: a framework. *IEEE/ACM Transactions on Networking*, 5(4):561–571, August 1997.

[21] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, October 1997.

[22] Lan Huang, Gang Peng, and Tzi cker Chiueh. Multi-dimensional storage virtualization. In *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '04)*, June 2004.

[23] Zhen Liu, Mark Squillante, and Joel Wolf. On maximizing service-level-agreement profits. In *Proceedings of the 3rd ACM Conference on Electronic Commerce (EC-01)*, pages 213–223, New York, October 14–17 2001. ACM Press.

[24] Christopher Lumb, Arif Merchant, and Guillermo A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003.

[25] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo Seltzer, Jeff Chase, Ri chard Kisley, Andrew Gallatin, Rajiv Wickremisinghe, and Eran Gabber. Structure and performance of the Direct Access File System. In *USENIX Technical Conference*, pages 1–14, June 2002.

[26] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[27] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems.

In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 44–55, 1998.

[28] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, October 1998.

[29] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002.

[30] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Symposium on Operating Systems Design and Implementation*, December 2002.

[31] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 1994.

[32] John Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Lecture Notes in Computer Science*, 2092:75–92, 2001.

[33] Kenneth G. Yocum, Darrell C. Anderson, Jeffrey S. Chase, and Amin Vahdat. Anypoint: Extensible transport switching on the edge. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[34] Lixia Zhang. Virtual Clock: A new traffic control algorithm for packet switching networks. In *SIGCOMM '90 Symposium: Communications Architectures and Protocols*, pages 19–29, September 1990.