Proportional-Share Scheduling for Distributed Storage Systems

Yin Wang*
University of Michigan
yinw@eecs.umich.edu

Arif Merchant
HP Laboratories
arif@hpl.hp.com

Abstract

Fully distributed storage systems have gained popularity in the past few years because of their ability to use cheap commodity hardware and their high scalability. While there are a number of algorithms for providing differentiated quality of service to clients of a centralized storage system, the problem has not been solved for distributed storage systems. Providing performance guarantees in distributed storage systems is more complex because clients may have different data layouts and access their data through different coordinators (access nodes), yet the performance guarantees required are global.

This paper presents a distributed scheduling framework. It is an adaptation of fair queuing algorithms for distributed servers. Specifically, upon scheduling each request, it enforces an extra delay (possibly zero) that corresponds to the amount of service the client gets on other servers. Different performance goals, e.g., per storage node proportional sharing, total service proportional sharing or mixed, can be met by different delay functions. The delay functions can be calculated at coordinators locally so excess communication is avoided. The analysis and experimental results show that the framework can enforce performance goals under different data layouts and workloads.

1 Introduction

The storage requirements of commercial and institutional organizations are growing rapidly. A popular approach for reducing the resulting cost and complexity of management is to consolidate the separate computing and storage resources of various applications into a common pool. The common resources can then be managed together and shared more efficiently. Distributed storage systems, such as *Federated Array of Bricks* (FAB) [20], Petal [16], and IceCube [27], are designed to serve as large storage pools. They are built from a number of individual storage nodes, or bricks, but present a single, highly-available store to users. High scalability is another advantage of distributed storage systems. The system can grow smoothly from small to large-scale installations because it is not limited by the capacity of an array or mainframe chassis. This satisfies the needs of service providers to continuously add application workloads onto storage resources.

A data center serving a large enterprise may support thousands of applications. Inevitably, some of these applications will have higher storage performance requirements than others. Traditionally, these requirements have been met by allocating separate storage for such applications; for example, applications with high write rates may be allocated storage on high-end disk arrays with large caches, while other applications live on less expensive, lower-end storage. However, maintaining separate storage hardware in a data center can be a management nightmare. It would be preferable to provide each application with the service level it requires while sharing storage. However, storage systems typically treat all I/O requests equally, which makes differentiated service difficult. Additionally, a bursty I/O workload from one application can cause other applications sharing the same storage to suffer.

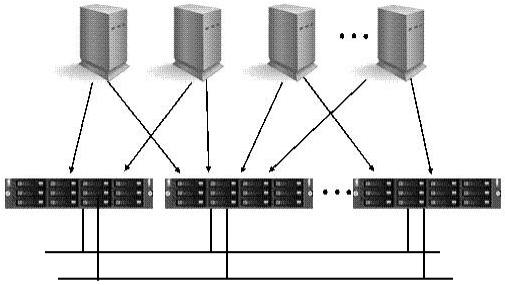


Figure 1: **A distributed storage system**

One solution to this problem is to specify the performance requirement of each application's storage workload and enable the storage system to ensure that it is met. Thus applications are insulated from the impact of workload surges in other applications. This can be achieved by ordering the requests from the applications appropriately, usually through a centralized scheduler, to coordinate access to the shared resources [5, 6, 24]. The scheduler can be implemented in the server or as a separate *interposed request scheduler* [2, 12, 17, 29] that treats the storage server as a black box and applies the resource control externally.

Centralized scheduling methods, however, fit poorly with distributed storage systems. To see this, consider the typical distributed storage system shown in Figure 1. The system is composed of bricks; each brick is a computer with a CPU, memory, networking, and storage. In a symmetric system, each brick runs the same software. Data stored by the system is distributed across the bricks. Typically, a client accesses the data through a *coordina-*

*This work was done during an internship at HP Laboratories.

tor, which locates the bricks where the data resides and performs the I/O operation. A brick may act both as a storage node and a coordinator. Different requests, even from the same client, may be coordinated by different bricks. Two features in this distributed architecture prevent us from applying any existing request scheduling algorithm directly. First, the coordinators are distributed. A coordinator schedules requests possibly without the knowledge of requests processed by other coordinators. Second, the data corresponding to requests from a client could be distributed over many bricks, since a logical volume in a distributed storage system may be striped, replicated, or erasure-coded across many bricks [7]. Our goal is to design a distributed scheduler that can provide service guarantees regardless of the data layout.

This paper proposes a distributed algorithm to enforce *proportional sharing* of storage resources among *streams* of requests. Each stream has an assigned *weight*, and the algorithm reserves for it a minimum share of the system capacity proportional to its weight. Surplus resources are shared among streams with outstanding requests, also in proportion to their weights. System capacity, in this context, can be defined in a variety of ways: for example, the number of I/Os per second, the number of bytes read or written per second, etc. The algorithm is work-conserving: no resource is left idle if there is any request waiting for it. However, it can be shown easily that a work-conserving scheduling algorithm for multiple resources (bricks in our system) cannot achieve proportional sharing in all cases. We present an extension to the basic algorithm that allows per-brick proportional sharing in such cases, or a method that provides a hybrid between system-wide proportional sharing and per-brick proportional sharing. This method allows total proportional sharing when possible while ensuring a minimum level of service on each brick for all streams.

The contribution of this paper includes a novel distributed scheduling framework that can incorporate many existing centralized fair queuing algorithms. Within the framework, several algorithms that are extensions to Start-time Fair Queuing [8] are developed for different system settings and performance goals. To the best of our knowledge, this is the first algorithm that can achieve total service proportional sharing for distributed storage resources with distributed schedulers. We evaluate the proposed algorithms both analytically and experimentally on a FAB system, but the results are applicable to most distributed storage systems. The results confirm that the algorithms allocate resources fairly under various settings — different data layouts, clients accessing the data through multiple coordinators, and fluctuating service demands.

This paper is organized as follows. Section 2 presents an overview of the problem, the background, and the related work. Section 3 describes our distributed fair queueing framework, two instantiations of it, and their properties. Section 4 presents the experimental evaluation of the algorithms. Section 5 concludes.

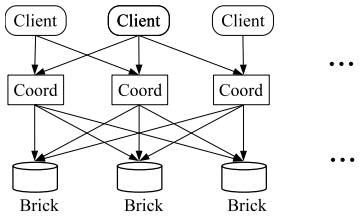


Figure 2: **Data access model of a distributed storage system.** Different clients may have different data layouts spreading across different sets of bricks. However, coordinators know all data layouts and can handle requests from any client.

2 Overview and background

We describe here the distributed storage system that our framework is designed for, the proportional sharing properties it is intended to enforce, the centralized algorithm that we base our work upon, and other related work.

2.1 Distributed Storage Systems

Figure 2 shows the configuration of a typical distributed storage system. The system includes a collection of storage *bricks*, which might be built from commodity disks, CPUs, and NVRAM. Bricks are connected by a standard network such as gigabit Ethernet. Access to the data on the bricks is handled by the coordinators, which present a *virtual disk* or *logical volume* interface to the clients. In the FAB distributed storage system [20], a client may access data through an arbitrary coordinator or a set of coordinators at the same time to balance its load. Coordinators also handle data layout and volume management tasks, such as volume creation, deletion, extension and migration. In FAB, the coordinators reside on the bricks, but this is not required. We consider local area distributed storage systems where the network latencies are small compared with disk latencies. We assume that the network bandwidths are sufficiently large that the I/O throughput is limited by the bricks rather than the network.

The data layout is usually designed to optimize properties such as load balance, availability, and reliability. In FAB, a logical volume is divided into a number of *segments*, which may be distributed across bricks using a replicated or erasure-coded layout. The choice of brick-set for each segment is determined by the storage system. Generally, the layout is opaque to the clients.

The scheduling algorithm we present is designed for such a distributed system, making a minimum of as-

SYMBOLS	DESCRIPTION
ϕ_f	Weight of stream f
p_f^i	Stream f's i-th request
$p_{f,A}^i$	Stream f's i-th request to brick A
$cost(\cdot)$	Cost of a single request
$cost_f^{max}$	Max request cost of stream f
$cost_{f,A}^{max}$	Max cost on brick A of f
$W_f(t_1, t_2)$	Aggregate cost of requests served from f during interval $[t_1, t_2]$
$batchcost$ $(p_{f,A}^i)$	Total cost of requests in between $p_{f,A}^{i-1}$ and $p_{f,A}^i$, including $p_{f,A}^i$
$batchcost_{f,A}^{max}$	Max value of $batchcost(p_{f,A}^i)$
$A(\cdot)$	Arrival time of a request
$S(\cdot)$	Start tag of a request
$F(\cdot)$	Finish tag of a request
$v(t)$	Virtual time at time t
$delay(\cdot)$	Delay value of a request

Table 1: **Some symbols used in this paper.**

sumptions. The data for a client may be laid out in an arbitrary manner. Clients may request data located on an arbitrary set of bricks at arbitrary and even fluctuating rates, possibly through an arbitrary set of coordinators.

2.2 Proportional Sharing

The algorithms in this paper support proportional sharing of resources for clients with queued requests. Each client is assigned a weight by the user and, in every time interval, the algorithms try to ensure that clients with requests pending during that interval receive service proportional to their weights.

More precisely, I/O requests are grouped into service classes called *streams*, each with a weight assigned; e.g., all requests from a client could form a single stream. A stream is *backlogged* if it has requests queued. A stream f consists a sequence of requests $p_f^0 ... p_f^n$. Each request has an associated service cost $cost(p_f^i)$. For example, with bandwidth performance goals, the cost might be the size of the requested data; with service time goals, request processing time might be the cost. The maximum request cost of stream f is denoted $cost_f^{max}$. The weight assigned to stream f is denoted ϕ_f; only the relative values of the weights matter for proportional sharing.

Formally, if $W_f(t_1, t_2)$ is the aggregate cost of the requests from stream f served in the time interval $[t_1, t_2]$, then the unfairness between two continuously backlogged streams f and g is defined to be:

$$\left| \frac{W_f(t_1, t_2)}{\phi_f} - \frac{W_g(t_1, t_2)}{\phi_g} \right| \quad (1)$$

A fair proportional sharing algorithm should guarantee that (1) is bounded by a constant. The constant usually depends on the stream characteristics, e.g., $cost_f^{max}$. The time interval $[t_1, t_2]$ in (1) may be any time duration. This corresponds to a "use it or lose it" policy, i.e., a stream will not suffer during one time interval for consuming surplus resources in another interval, nor will it benefit later from under-utilizing resources.

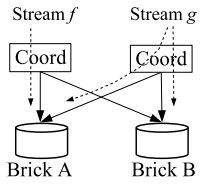


Figure 3: **Distributed data.** Stream f sends requests to brick A only while stream g sends requests to both A and B.

In the case of distributed data storage, we need to define what is to be proportionally shared. Let us first look at the following example.

EXAMPLE 1. Figure 3 is a storage system consisting of two bricks A and B. If streams f and g are equally weighted and both backlogged at A, how should we allocate the service capacity of brick A? ∎

There are two alternatives for the above example, which induce two different meanings for proportional sharing. The first is *single brick proportional sharing*, i.e., service capacity of brick A will be proportionally shared. Many existing proportional sharing algorithms fall into this category. However, stream g also receives service at brick B, thus receiving higher overall service. While this appears fair because stream g does a better job of balancing its load over the bricks than stream f, note that the data layout may be managed by the storage system and opaque to the clients; thus the quality of load balancing is merely an accident. From the clients' point of view, stream f unfairly receives less service than stream g. The other alternative is *total service proportional sharing*. In this case, the share of the service stream f receives on brick A can be increased to compensate for the fact that stream g receives service on brick B, while f does not. This problem is more intricate and little work has been done on it.

It is not always possible to guarantee total service proportional sharing with a work-conserving scheduler, i.e., where the server is never left idle when there is a request queued. Consider the following extreme case.

EXAMPLE 2. Stream f requests service from brick A only, while equally weighted stream g is sending requests to A and many other bricks. The amount of service g obtains from the other bricks is larger than the capacity of A. With a work-conserving scheduler, it is impossible to equalize the total service of the two streams. ∎

If the scheduler tries to make the total service received by f and g as close to equal as possible, g will be blocked at brick A, which may not be desirable. Inspired by the example, we would like to guarantee some minimum service on each brick for each stream, yet satisfy total service proportional sharing whenever possible.

In this paper, we propose a distributed algorithm framework under which single brick proportional sharing, total service proportional sharing, and total service proportional sharing with a minimum service guarantee are all possible. We note that, although we focus on distributed storage systems, the algorithms we propose may be more broadly applicable to other distributed systems.

2.3 The centralized approach

In selecting an approach towards a distributed proportional sharing scheduler, we must take four requirements into account: i) the scheduler must be work-conserving: resources that backlogged streams are waiting for should never be idle; ii) "use it or lose it", as described in the previous section; iii) the scheduler should accommodate fluctuating service capacity since the service times of IOs can vary unpredictably due to the effects of caching, sequentiality, and interference by other streams; and iv) reasonable computational complexity—there might be thousands of bricks and clients in a distributed storage system, hence the computational and communication costs must be considered.

There are many centralized scheduling algorithms that could be extended to distributed systems [3, 4, 8, 28, 30, 17, 14, 9]. We chose to focus on the Start-time Fair Queuing (SFQ) [8] and its extension SFQ(D) [12] because they come closest to meeting the requirements above. We present a brief discussion of the SFQ and SFQ(D) algorithms in the remainder of this section.

SFQ is a proportional sharing scheduler for a single server; intuitively, it works as follows. SFQ assigns a *start time* tag and a *finish time* tag to each request corresponding to the normalized times at which the request should start and complete according to a system notion of *virtual time*. For each stream, a new request is assigned a start time based on the assigned finish time of the previous request, or the current virtual time, whichever is greater. The finish time is assigned as the start time plus the normalized cost of the request. The virtual time is set to be the start time of the currently executing request, or the finish time of the last completed request if there is none currently executing. Requests are scheduled in the order of their start tags. It can be shown that, in any time interval, the service received by two backlogged workloads is approximately proportionate to their weights.

More formally, the request p_f^i is assigned the *start tag* $S(p_f^i)$ and the *finish tag* $F(p_f^i)$ as follows:

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

$$F(p_f^i) = S(p_f^i) + \frac{cost(p_f^i)}{\phi_f}, i \geq 1 \quad (3)$$

where $A(p_f^i)$ is the arrival time of request p_f^i, and $v(t)$ is the virtual time at t; $F(p_f^0) = 0, v(0) = 0$.

SFQ cannot be directly applied to storage systems, since storage servers are concurrent, serving multiple requests at a time, and the virtual time $v(t)$ is not defined. Jin *et al.* [12] extended SFQ to concurrent servers by defining the virtual time as the maximum start tag of requests in service (the last request dispatched). The resulting algorithm is called *depth-controlled start-time fair queuing* and abbreviated to SFQ(D), where D is the queue depth of the storage device. As with SFQ, the following theorem [12] shows that SFQ(D) provides backlogged workloads with proportionate service, albeit with a looser bound on the unfairness.

THEOREM 1. *During any interval $[t_1, t_2]$, the difference between the amount of work completed by an SFQ(D) server for two backlogged streams f and g is bounded by:*

$$\left| \frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g} \right| \leq \left(\frac{cost_f^{max}}{\phi_f} + \frac{cost_g^{max}}{\phi_g} \right) * (D+1) \quad (4)$$

While there are more complex variations of SFQ [12] that can reduce the unfairness of SFQ(D), for simplicity, we use SFQ(D) as the basis for our distributed scheduling algorithms. Since the original SFQ algorithm cannot be directly applied to storage systems, for the sake of readability, we will use "SFQ" to refer to SFQ(D) in the remainder of the paper.

2.4 Related Work

Extensive research in scheduling for packet switching networks has yielded a series of fair queuing algorithms; see [19, 28, 8, 3]. These algorithms have been adapted to storage systems for service proportional sharing. For example, YFQ [1], SFQ(D) and FSFQ(D) [12] are based on start-time fair queueing [8]; SLEDS [2] and SARC [29] use leaky buckets; CVC [11] employs the virtual clock [30]. Fair queuing algorithms are popular for two reasons: 1) they provide theoretically proven strong fairness, even under fluctuating service capacity, and 2) they are work-conserving.

However, fair queuing algorithms are not convenient for real-time performance goals, such as latencies. To address this issue, one approach is based on real-time schedulers; e.g., Façade [17] implements an Earliest

Deadline First (EDF) queue with the proportional feedback for adjusting the disk queue length. Another method is feedback control, a classical engineering technique that has recently been applied to many computing systems [10]. These generally require at least a rudimentary model of the system being controlled. In the case of storage systems, whose performance is notoriously difficult to model [22, 25], Triage [14] adopts an adaptive controller that can automatically adjust the system model based on input-output observations.

There are some frameworks [11, 29] combining the above two objectives (proportional sharing and latency guarantees) in a two-level architecture. Usually, the first level guarantees proportional sharing by fair queueing methods, such as CVC [11] and SARC [29]. The second level tries to meet the latency goal with a real-time scheduler, such as EDF. Some feedback from the second level to the first level scheduler is helpful to balance the two objectives [29]. All of the above methods are designed for use in a centralized scheduler and cannot be directly applied to our distributed scheduling problem.

Existing methods for providing quality of service in distributed systems can be put into two categories. The first category is the distributed scheduling of a single resource. The main problem here is to maintain information at each scheduler regarding the amount of resource each stream has so far received. For example, in fair queuing algorithms, where there is usually a system virtual time $v(t)$ representing the normalized fair amount of service that all backlogged clients should have received by time t, the problem is how to synchronize the virtual time among all distributed schedulers. This can be solved in a number or ways; for example, in high capacity crossbar switches, in order to fairly allocate the bandwidth of the output link, the virtual time of different input ports can be synchronized by the *access buffer* inside the crossbar [21]. In wireless networks, the communication medium is shared. When a node can overhear packages from neighboring nodes for synchronization, distributed priority backoff schemes closely approximate a single global fair queue [18, 13, 23]. In the context of storage scheduling, Request Window [12] is a distributed scheduler that is similar to a leaky bucket scheduler. Services for different clients are balanced by the windows issued by the storage server. It is not fully work-conserving under light workloads.

The second category is centralized scheduling of multiple resources. Gulati and Varman [9] address the problem of allocating disk bandwidth fairly among concurrent competing flows in a parallel I/O system with multiple disks and a centralized scheduler. They aim at the optimization problem of minimizing the unfairness among different clients with concurrent requests. I/O requests are scheduled in batches, and a combinatorial optimization problem is solved in each round, which makes the method computationally expensive. The centralized controller makes it unsuitable for use in fully distributed high-performance systems, such as FAB.

To the best of our knowledge, the problem of fair scheduling in distributed storage systems that involve both distributed schedulers and distributed data has not been previously addressed.

3 Proportional Sharing in Distributed Storage Systems

We describe a framework for proportional sharing in distributed storage systems, beginning with the intuition, followed by a detailed description, and two instantiations of the method exhibiting different sharing properties.

3.1 An intuitive explanation

First, let us consider the simplified problem where the data is centralized at one brick, but the coordinators may be distributed. An SFQ scheduler could be placed either at coordinators or at the storage brick. As fair scheduling requires the information for all backlogged streams, direct or indirect communication among coordinators may be necessary if the scheduler is implemented at coordinators. Placing the scheduler at bricks avoids the problem. In fact, SFQ(D) can be used without modification in this case, provided that coordinators attach a stream ID to each request so that the scheduler at the brick can assign the start tag accordingly.

Now consider the case where the data is distributed over multiple bricks as well. In this case, SFQ schedulers at each brick can guarantee only single brick proportional sharing, but not necessarily total service proportional sharing because the scheduler at each brick sees only the requests directed to it and cannot account for the service rendered at other bricks.

Suppose, however, that each coordinator broadcasts all requests to all bricks. Clearly, in this case, each brick has complete knowledge of all requests for each stream. Each brick responds only to the requests for which it is the correct destination. The remaining requests are treated as *virtual requests*, and we call the combined stream of real and virtual request a *virtual stream*; see Fig. 4. A virtual request takes zero processing time but does account for the service share allocated to its source stream. Then the SFQ scheduler at the brick guarantees service proportional sharing of backlogged virtual streams. As the aggregate service cost of a virtual stream equals the aggregate service cost of the original stream, total service proportional sharing can be achieved.

The above approach is simple and straightforward, but with large-scale distributed storage systems, broadcast-

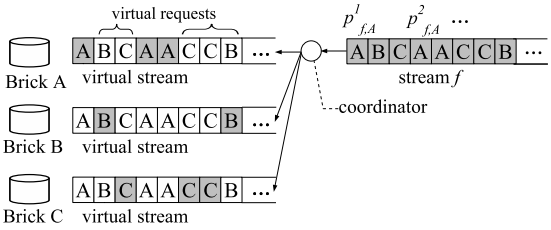


Figure 4: **The naive approach.** The coordinator broadcasts every request to all bricks. Requests to incorrect destination bricks are *virtual* and take zero processing time. Proportional scheduling at each local brick guarantees total service proportional sharing.

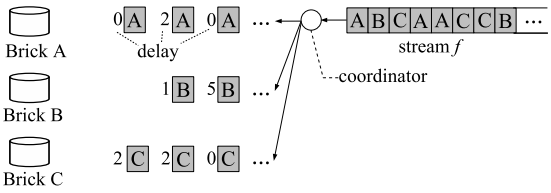


Figure 5: **The improved approach.** Only the aggregate cost of virtual requests is communicated, indicated by the number before each request (assuming unit cost of each request). Broadcasting is avoided yet total service proportional sharing can be achieved.

ing is not acceptable. We observe, however, that the SFQ scheduler requires only knowledge of the cost of each virtual request, the coordinators may therefore broadcast the cost value instead of the request itself. In addition, the coordinator may combine the cost of consecutive virtual requests and piggyback the total cost information onto the next real request; see Fig. 5. The communication overhead is negligible because, in general, read/write data rather than requests dominate the communication and the local area network connecting bricks usually has enough bandwidth for this small overhead.

The piggyback cost information on each real request is called the *delay* of the request, because the modified SFQ scheduler will delay processing the request according to this value. Different delay values may be used for different performance goals, which greatly extends the ability of SFQ schedulers. This flexibility is captured in the framework presented next.

3.2 Distributed Fair Queuing Framework

We propose the distributed fair queuing framework displayed in Fig. 6; as we show later, it can be used for total proportional sharing, single-brick proportional sharing, or a hybrid between the two. Assume there are streams $f, g, ...$ and bricks $A, B, ...$ The fair queueing scheduler is placed at each brick as just discussed. The scheduler has a priority queue for all streams and orders all requests by some priority, e.g., start time tags in the case of an SFQ scheduler. On the other hand, each coordinator has a separate queue for each stream, where the requests in a queue may have different destinations.

When we apply SFQ to the framework, each request has a start tag and a finish tag. To incorporate the idea presented in the previous section, we modify the computation of the tags as follows:

$$S(p^i_{f,A}) = \max \left\{ v(A(p^i_{f,A})), F(p^{i-1}_{f,A}) + \frac{delay(p^i_{f,A})}{\phi_f} \right\} \quad (5)$$

$$F(p^i_{f,A}) = S(p^i_{f,A}) + \frac{cost(p^i_{f,A})}{\phi_f} \quad (6)$$

The only difference between SFQ formulae (2-3) and those above is the new delay function for each request, which is calculated at coordinators and carried by the request. The normalized delay value translates into the amount of time by which the start tag should be shifted. How the delay is computed depends upon the proportional sharing properties we wish to achieve, and we will discuss several delay functions and the resulting sharing properties in the sections that follow. We will refer to the modified Start-time Fair Queuing algorithm as Distributed Start-time Fair Queuing (DSFQ).

In DSFQ, as in SFQ(D), $v(t)$ is defined to be the start tag of the last request dispatched to the disk before or at time t. There is no global virtual time in the system. Each brick maintains its own virtual time, which varies at different bricks depending on the workload and the service capacity of the brick.

We note that the framework we propose works with other fair scheduling algorithms [28] as long as each stream has its own clock such that the delay can be applied; for example, a similar extension could be made to the Virtual Clock algorithm [30] if we desire proportional service over an extended time period (time-averaged fairness) rather than the "use it or lose it" property (instantaneous fairness) supported by SFQ. Other options between these two extreme cases could be implemented in this framework, as well. For example, the scheduler can constrain each stream's time tag to be within some window of the global virtual time. Thus, a stream that underutilizes its share can get extra service later, but only to a limited extent.

If the delay value is set to always be zero, DSFQ reduces to SFQ and achieves single brick proportional sharing. We next consider other performance goals.

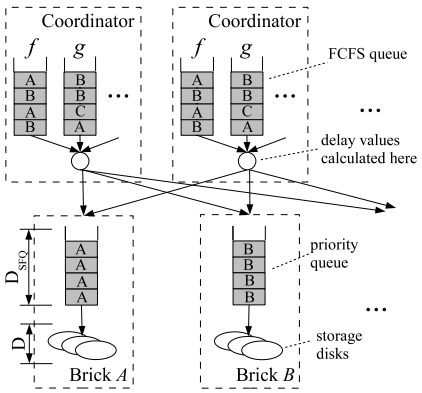


Figure 6: **The distributed fair queuing framework**

3.3 Total Service Proportional Sharing

We describe how the distributed fair queueing framework can be used for total proportional sharing when each stream uses one coordinator, and then argue that the same method also engenders total proportional sharing with multiple coordinators per stream.

3.3.1 Single-Client Single-Coordinator

We first assume that requests from one stream are always processed by one coordinator; different streams may or may not have different coordinators. We will later extend this to the multiple coordinator case. The performance goal, as before, is that the total amount of service each client receives must be proportional to its weight.

As described in Section 3.1, the following delay function for a request from stream f to brick A represents the total cost of requests sent to other bricks since the previous request to brick A.

$$delay(p^i_{f,A}) = batchcost(p^i_{f,A}) - cost(p^i_{f,A}) \quad (7)$$

When this delay function is used with the distributed scheduling framework defined by formulae (5-7), we call the resulting algorithm TOTAL-DSFQ. The delay function (7) is the total service cost of requests sent to other bricks since the last request on the brick. Intuitively, it implies that, if the brick is otherwise busy, a request should wait an extra time corresponding to the aggregate service requirements of the preceding requests from the same stream that were sent to other bricks, normalized by the stream's weight.

Why TOTAL-DSFQ engenders proportional sharing of the total service received by the streams can be explained using virtual streams. According to the formulae (5-7), TOTAL-DSFQ is exactly equivalent to the architecture where coordinators send virtual streams to the bricks and bricks are controlled by the standard SFQ. This virtual stream contains all the requests in f, but the requests that are not destined for A are served at A in zero time. Note that SFQ holds its fairness property even when the service capacity varies [8]. In our case, the server capacity (processing speed) varies from normal, if the request is to be serviced on the same brick, to infinity if the request is virtual and is to be serviced elsewhere. Intuitively, since the brick A sees all the requests in f (and their costs) as a part of the virtual stream, the SFQ scheduler at A factors in the costs of the virtual requests served elsewhere in its scheduling, even though they consume no service time at A. This will lead to proportional sharing of the total service. The theorem below formalizes the bounds on unfairness using TOTAL-DSFQ.

THEOREM 2. *Assume stream f is requesting service on N_f bricks and stream g on N_g bricks. During any interval $[t_1, t_2]$ in which f and g are both continuously backlogged at some brick A, the difference between the total amount of work completed by all bricks for the two streams during the entire interval, normalized by their weights, is bounded as follows:*

$$\begin{aligned} &\left| \frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g} \right| \\ \leq\ & ((D + D_{SFQ}) * N_f + 1) \frac{cost^{max}_{f,A}}{\phi_f} + \\ & ((D + D_{SFQ}) * N_g + 1) \frac{cost^{max}_{g,A}}{\phi_g} + \\ & (D_{SFQ} + 1) \left(\frac{batchcost^{max}_{f,A}}{\phi_f} + \frac{batchcost^{max}_{g,A}}{\phi_g} \right) \end{aligned} \quad (8)$$

where D is the queue depth of the disk[1]*, and D_{SFQ} is the queue depth of the Start-time Fair Queue at the brick.*

PROOF. The proof of this and all following theorems can be found in [26]. □

The bound in Formula (8) has two parts. The first part is similar to the bound of SFQ(D) in (4), the unfairness due to server queues. The second part is new and contributed by the distributed data. If the majority of requests of stream f is processed at the backlogged server, the $batchcost^{max}_{f,A}$ is small and the bound is tight. Otherwise, if f gets a lot of service at other bricks, the bound is loose.

As we showed in Example 2, however, there are situations in which total proportional sharing is impossible with work conserving schedulers. In the theorem above, this corresponds to the case with an infinite $batchcost^{max}_{g,A}$, and hence the bound is infinite. To delineate more precisely when total proportional sharing is possible under TOTAL-DSFQ, we characterize when the total service rates of the streams are proportional to their weights. The theorem below says that, under TOTAL-DSFQ, if a set of streams are backlogged together at a

[1] If there are multiple disks (the normal case), D is the sum of the queue depths of the disks.

set of bricks, then either their normalized total service rates over all bricks are equal (thus satisfying the total proportionality requirement), or there are some streams whose normalized service rates are equal and the remainder receive no service at the backlogged bricks because they already receive more service elsewhere.

Let $R_f(t_1,t_2) = W_f(t_1,t_2)/(\phi_f * (t_2 - t_1))$ be the normalized service rate of stream f in the duration (t_1,t_2). If the total service rates of streams are proportional to their weights, then their normalized service rates should be equal as the time interval $t_2 - t_1$ goes to infinity. Suppose stream f is backlogged at a set of bricks, denoted as set S, its normalized service rate at S is denoted as $R_{f,S}(t_1,t_2)$, and $R_{f,other}(t_1,t_2)$ denotes its normalized total service rate at all other bricks. $R_f(t_1,t_2) = R_{f,S}(t_1,t_2) + R_{f,other}(t_1,t_2)$. We drop (t_1,t_2) hereafter as we always consider interval (t_1,t_2).

THEOREM 3. *Under* TOTAL-DSFQ*, if during* (t_1, t_2)*, streams* $\{f_1, f_2, ...f_n\}$ *are backlogged at a set of bricks* S*, in the order* $R_{f_1,other} \leq R_{f_2,other} \leq ... R_{f_n,other}$*, as* $t_2 - t_1 \to \infty$*, either* $R_{f_1} = R_{f_2} = ...R_{f_n}$ *or* $\exists k \in \{1,2,...n-1\}$*, such that* $R_{f_1} = ...R_{f_k} \leq R_{f_{k+1},other}$ *and* $R_{f_{k+1},S} = ...R_{f_n,S} = 0$.

The intuition of Theorem 3 is as follows. At brick set S, let us first set $R_{f_1,S} = R_{f_2,S} = ... = R_{f_n,S} = 0$ and try to allocate the resources of S. Stream f_1 has the highest priority since its delay is the smallest. Thus the SFQ scheduler will increase $R_{f_1,S}$ until $R_{f_1} = R_{f_2,other}$. Now both f_1 and f_2 have the same total service rate and the same highest priority. Brick set S will then increase $R_{f_1,S}$ and $R_{f_2,S}$ equally until $R_{f_1} = R_{f_2} = R_{f_3,other}$. In the end, either all the streams have the same total service rate, or it is impossible to balance all streams due to the limited service capacity of all bricks in S. In the latter case, the first k streams have equal total service rates, while the remaining streams are blocked for service at S. Intuitively, this is the best we can do with a work-conserving scheduler to equalize normalized service rates.

In Section 3.4 we propose a modification to TOTAL-DSFQ that ensures no stream is blocked at any brick.

3.3.2 Single-client Multi-coordinator

So far we have assumed that a stream requests service through one coordinator only. In many high-end systems, however, it is preferable for high-load clients to distribute their requests among multiple coordinators in order to balance the load on the coordinators. In this section, we discuss the single-client multi-coordinator setting and the corresponding fairness analysis for TOTAL-DSFQ. In summary, we find that TOTAL-DSFQ does engender

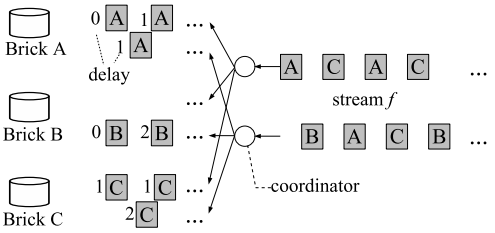


Figure 7: **Effect of multiple coordinators under TOTAL-DSFQ.** Delay value of an individual request is different from Fig. 5, but the total amount of delay remains the same.

total proportional sharing in this setting, except in some unusual cases.

We motivate the analysis with an example. First, let us assume that a stream accesses two coordinators in round-robin order and examine the effect on the delay function (7) through the example stream in Fig. 5. The result is displayed in Fig. 7. Odd-numbered requests are processed by the first coordinator and even-numbered requests are processed by the second coordinator. With one coordinator, the three requests to brick A have delay values 0, 2 and 0. With two round-robin coordinators, the delay values of the two requests dispatched by the first coordinator are now 0 and 1; the delay value of the request dispatched by the second coordinator is 1. Thus, although individual request may have delay value different from the case of single coordinator, the total amount of delay remains the same. This is because every virtual request (to other bricks) is counted exactly once.

We formalize this result in Theorem 4 below, which says, essentially, that streams backlogged at a brick receive total proportional service so long as each stream uses a consistent set of coordinators (i.e., the same set of coordinators for each brick it accesses).

Formally, assume stream f sends requests through n coordinators $C_1, C_2, ..., C_n$, and coordinator C_i receives a substream of f denoted as f_i. With respect to brick A, each substream f_i has its $batchcost^{max}_{f_i,A}$. Let us first assume that $batchcost^{max}_{f_i,A}$ is finite for all substreams, i.e., requests to A are distributed among all coordinators.

THEOREM 4. *Assume stream* f *accesses* n *coordinators such that each one receives substreams* $f_1, ..., f_n$*, respectively, and stream* g *accesses* m *coordinators with substreams* $g_1, ..., g_m$*, respectively. During any interval* $[t_1,t_2]$ *in which* f *and* g *are both continuously backlogged at brick A, inequality (8) still holds, where*

$$batchcost^{max}_{f,A} = max\{\ batchcost^{max}_{f_1,A}, ... \\ batchcost^{max}_{f_n,A}\} \quad (9)$$

$$batchcost^{max}_{g,A} = max\{\ batchcost^{max}_{g_1,A}, ... \\ batchcost^{max}_{g_m,A}\} \quad (10)$$

An anomalous case arises if a stream partitions the

bricks into disjoint subsets and accesses each partition through separate coordinators. In this case, the requests served in one partition will never be counted in the delay of any request to the other partition, and the total service may no longer be proportional to the weight. For example, requests to B in Fig. 7 have smaller delay values than the ones in Fig. 5. This case is unlikely to occur with most load balancing schemes such as round-robin or uniformly random selection of coordinators. Note that the algorithm will still guarantee total proportional sharing if *different* streams use separate coordinators.

More interestingly, selecting randomly among multiple coordinators may smooth out the stream, and result in more uniform delay values. For example, if $batchcost(p^j_{f,A})$ in the original stream is a sequence of i.i.d. (independent, identically distributed) random variables with large variance such that $batchcost^{max}_{f,A}$ might be large, it is not difficult to show that with independently random mapping of each request to a coordinator, $batchcost(p^j_{f_i,A})$ is also a sequence of i.i.d. random variables with the same mean, but the variance decreases as number of coordinators increases. This means that under random selection of coordinators, while the average delay is still the same (thus service rate is the same), the variance in the delay value is reduced and therefore the unfairness bound is tighter. We test this observation through an empirical study later.

3.4 Hybrid Proportional Sharing

Under TOTAL-DSFQ, Theorem 3 tells us that a stream may be blocked at a brick if it gets too much service at other bricks. This is not desirable in many cases. We would like to guarantee a minimum service rate for each stream on every brick so the client program can always make progress. Under the DSFQ framework, i.e., formulae (5-6), this means that the delay must be bounded, using a different delay function than the one used in TOTAL-DSFQ. We next develop a delay function that guarantees a minimum service share to backlogged streams on each brick.

Let us assume that the weights assigned to streams are normalized, i.e. $0 \leq \phi_f \leq 1$ and $\sum_f \phi_f = 1$. Suppose that, in addition to the weight ϕ_f, each stream f is assigned a brick-minimum weight ϕ^{min}_f, corresponding to the minimum service share per brick for the stream.[2] We can then show that the following delay function will guarantee the required minimum service share on each brick for each stream.

$$delay(p^i_{f,A}) = \frac{\phi_f/\phi^{min}_f - 1}{1-\phi_f} * cost(p^i_{f,A}) \quad (11)$$

[2]Setting the brick-minimum weights requires knowledge of the client data layouts. We do not discuss this further in the paper.

We can see, for example, that setting $\phi^{min}_f = \phi_f$ yields a delay of zero, and the algorithm then reduces to single brick proportional sharing that guarantees minimum share ϕ_f for stream f, as expected.

By combining delay function (11) with the delay function (7) for TOTAL-DSFQ, we can achieve an algorithm that approaches total proportional sharing while guaranteeing a minimum service level for each stream per brick, as follows.

$$delay(p^i_{f,A}) = \min \{\ batchcost(p^i_{f,A}) - cost(p^i_{f,A}), \frac{\phi_f/\phi^{min}_f - 1}{1-\phi_f} * cost(p^i_{f,A})\} \quad (12)$$

The DSFQ algorithm using the delay function (12) defines a new algorithm called HYBRID-DSFQ. Since the delay under HYBRID-DSFQ is no greater than the delay in (11), the service rate at every brick is no less than the rate under (11), thus the minimum per brick service share ϕ^{min}_f is still guaranteed. On the other hand, if the amount of service a stream f receives on other bricks between requests to brick A is lower than $(\phi_f/\phi^{min}_f - 1)/(1-\phi_f) * cost(p^i_{f,A})$, the delay function behaves similarly to equation (7), and hence the sharing properties in this case should be similar to TOTAL-DSFQ, i.e., total proportional sharing.

Empirical evidence (in Section 4.3) indicates that HYBRID-DSFQ works as expected for various workloads. However, there are pathological workloads that can violate the total service proportional sharing property. For example, if a stream using two bricks knows its data layout, it can alternate bursts to one brick and then the other. Under TOTAL-DSFQ, the first request in each burst would have received a large delay, corresponding to the service the stream had received on the other brick during the preceding burst, but in HYBRID-DSFQ, the delay is truncated by the minimum share term in the delay function. As a result, the stream receives more service than its weight entitles it to. We believe that this can be resolved by including more history in the minimum share term, but the design and evaluation of such a delay function is reserved to future work.

4 Experimental Evaluation

We evaluate our distributed proportional sharing algorithm in a prototype FAB system [20], which consists of six bricks. Each brick is an identically configured HP ProLiant DL380 server with 2x 2.8GHz Xeon CPU, 1.5GB RAM, 2x Gigabit NIC, and an integrated Smart Array 6i storage controller with four 18G Ultral320, 15K rpm SCSI disks configured as RAID 0. All bricks are running SUSE 9.2 Linux, kernel 2.6.8-24.10-smp. Each brick runs a coordinator. To simplify performance com-

parisons, FAB caching was turned off, except at the disk level.[3]

The workload generator consists of a number of clients (streams), each running several Postmark [15] instances. We chose Postmark as the workload generator because its independent, randomly-distributed requests give us the flexibility to configure different workloads for demonstration and for testing extreme cases. Each Postmark thread has exactly one outstanding request in the system at any time, accessing its isolated 256MB logical volume. Unless otherwise specified, each volume resides on a single brick and each thread generates random read/write requests with file sizes from 1KB to 16KB. The number of transactions per thread is sufficiently large so the thread is always active when it is on. Other parameters of Postmark are set to the default values.

With our work-conserving schedulers, until a stream is backlogged, its IO throughput increases as the number of threads increases. When it is backlogged, on the other hand, the actual service amount depends on the scheduling algorithm. To simplify calculation, we target throughput (MB/sec) proportional sharing, and define the cost of a request to be its size. Other cost functions such as IO/sec or estimated disk service time could be used as well.

4.1 Single Brick Proportional Sharing

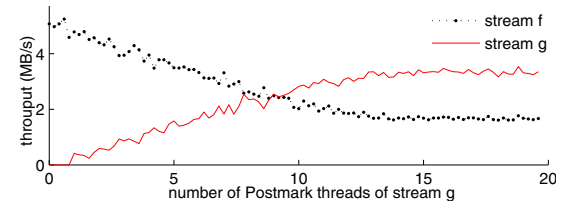
We first demonstrate the effect of TOTAL-DSFQ on two streams reading from one brick. Stream f consistently has 30 Postmark threads, while the number of Postmark threads for stream g is increased from 0 to 20. The ratio of weights between f and g is at 1:2. As the data is not distributed, the delay value is always zero and this is essentially the same as SFQ(D) [12].

Figure 8 shows the performance isolation between the two clients. The throughput of stream g is increasing and its latency is fixed until g acquires its proportional share at around 13 threads. After that, additional threads do not give any more bandwidth but increase the latency. On the other hand, the throughput and latency of stream f are both affected by g. Once g gets its share, it has no further impact on f.

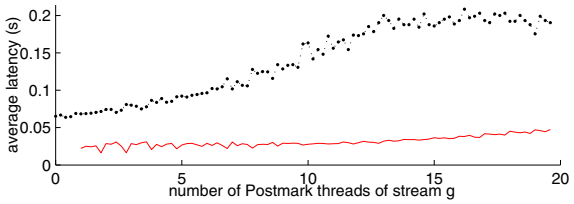
4.2 Total Service Proportional Sharing

Figure 9 demonstrates the effectiveness of TOTAL-DSFQ for two clients. The workload streams have access patterns shown in Fig. 3. We arranged the data

[3]Our algorithms focus on the management of storage bandwidth; a full exploration of the management of multiple resources (including cache and network bandwidth) to control end-to-end performance is beyond the scope of this paper.



(a) throughputs for two streams



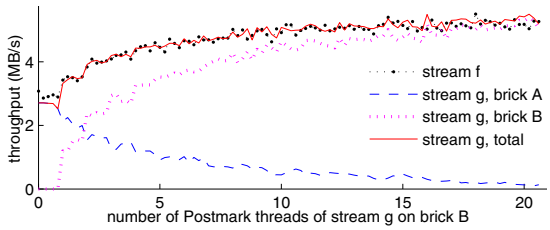
(b) latencies for two streams

Figure 8: **Proportional sharing on one brick.** ϕ_f:ϕ_g=1:2; legend in (a) also applies to (b).

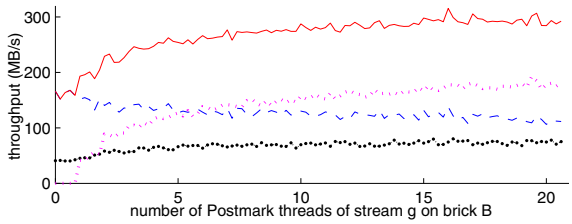
layout so that each Postmark thread accesses only one brick. Stream f and stream g both have 30 threads on brick A throughout the experiment, meanwhile, an increasing number of threads from g is processed at brick B. Postmark allows us to specify the maximum size of the random files generated, and we tested the algorithm with workloads using two different maximum random file sizes, 16KB and 1MB.

Figure 9(a) shows that as the number of Postmark threads from stream g directed to brick B increases, its throughput from brick B increases, and the share it receives at brick A decreases to compensate. The total throughputs received by streams f and g stay roughly equal throughout. As the stream g becomes more unbalanced between bricks A and B, however, the throughput difference between streams f and g varies more. This can be related to the fairness bound in Theorem 2: as the imbalance increases, so does $batchcost_{g,A}^{max}$, and the bound becomes a little looser. Figure 9(b) uses the same data layout but with a different file size and weight ratio. As g gets more service on B, its throughput rises on B from 0 to 175 MB/s. As a result, the algorithm increases f's share on the shared brick A, and its throughput rises from 40 MB/s to 75 MB/s, while g's throughput on A drops from 160 MB/s to 125 MB/s. In combination the throughput of both streams increases, whole maintaining a 1:4 ratio.

The experiment in Figure 10 has a data layout with dependencies among requests. Each thread in g accesses all three bricks, while stream f accesses one brick only. The resource allocation is balanced when stream g has three or more threads on the shared brick. As g has a RAID-0 data layout, the service rates on the other two bricks are limited by the rate on the shared brick. This experiment shows that TOTAL-DSFQ correctly controls the



(a) Random I/O, ϕ_f:ϕ_g=1:1, max file size=16KB



(b) Sequential I/O, ϕ_f:ϕ_g=1:4, max file size=1MB

Figure 9: **Total service proportional sharing.** f's data is on brick A only; g has data on both bricks A and B. As g gets more service on the bricks it does not share with f, the algorithm increases f's share on the brick they do share; thus the total throughputs of both streams increase.

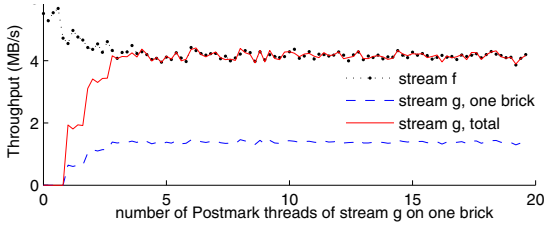
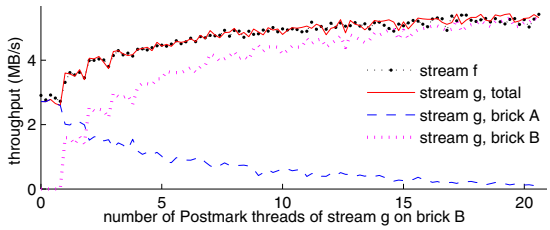


Figure 10: **Total service proportional sharing with striped data.** ϕ_f:ϕ_g=1:1. g has RAID-0 logical volume striping on three bricks; f's data is on one brick only.

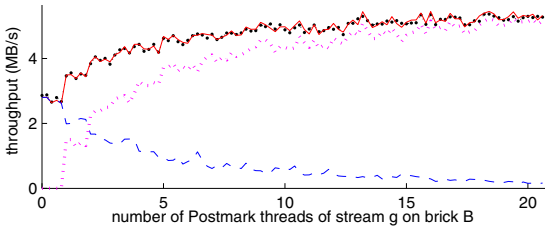
total share in the case where requests on different bricks are dependent. We note that in this example, the scheduler could allocate more bandwidth on the shared brick to stream g in order to improve the total system throughput instead of maintaining proportional service; however, this is not our goal.

Figure 11 is the result with multiple coordinators. The data layouts and workloads are the same as in the experiment shown in Figures 3 and 9: two bricks, stream f accesses only one, and stream g accesses both. The only difference is that stream g accesses both bricks A and B through two or four coordinators in round-robin order.

Using multiple coordinators still guarantees proportional sharing of the total throughput. Furthermore, a comparison of Fig. 9, 11(a), and 11(b) indicates that as the number of coordinators increases, the match between the total throughputs received by f and g is closer, i.e., the unfairness bound is tighter. This confirms the observation in Section 3.3.2 that multiple coordinators may smooth out a stream and reduce the unfairness.



(a) Two coordinators



(b) Four coordinators

Figure 11: **Total service proportional sharing with multi-coordinator**, ϕ_f:ϕ_g=1:1

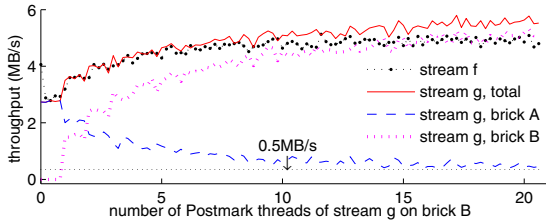
4.3 Hybrid Proportional Sharing

The result of HYBRID-DSFQ is presented in Fig. 12. The workload is the same as in the experiment shown in Figures 3 and 9: stream f accesses brick A only, and stream g accesses both A and B. Streams f and g both have 20 Postmark threads on A, and g has an increasing number of Postmark threads on B. We wish to give stream g a minimum share of $1/12$ on brick A when it is backlogged. This corresponds to $\phi_g^{min} = 1/12$; based on Equation 12, the delay function for g is

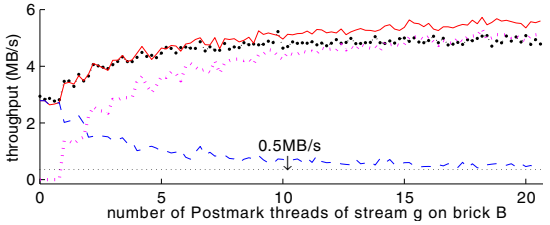
$$delay(p^i_{g,A}) = \min \{\ batchcost(p^i_{g,A}) - cost(p^i_{g,A}),\ 10 * cost(p^i_{g,A})\}$$

Stream f is served on A only and thus the delay is always zero.

With HYBRID-DSFQ, the algorithm reserves a minimum share for each stream, and tries to make the total throughput as close as possible without reallocating the reserved share. For this workload, the service capacity of a brick is approximately 6MB/sec. We can see in Fig. 12(a) that if the throughput of stream g on brick B is less than 4MB, HYBRID-DSFQ can balance the total throughputs of the two streams. As g receives more service on brick B, the maximum delay part in HYBRID-DSFQ takes effect and g gets its minimum share on brick A. The total throughputs are no longer proportional to the assigned weights, but is still reasonably close. Figure 12(b) repeats the experiment with the streams selecting between two coordinators alternately; the workload

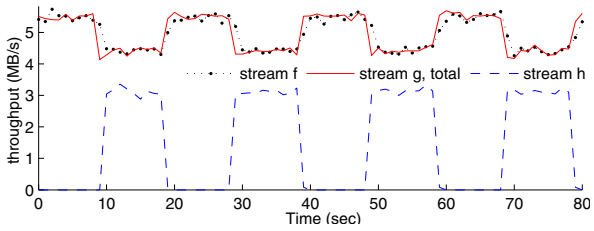


(a) throughputs with HYBRID-DSFQ

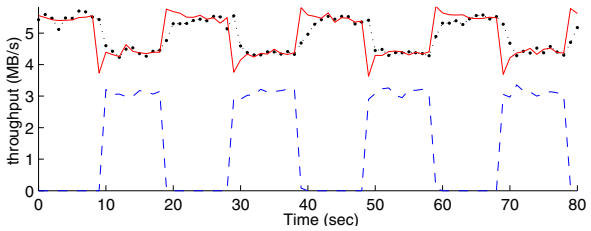


(b) throughputs with HYBRID-DSFQ, two coordinators

Figure 12: **Two-brick experiment using HYBRID-DSFQ**



(a) Streams f and g both have 20 Postmark threads on brick A, i.e., Queue depth $D_{SFQ} = 20$



(b) Streams f and g both have 30 threads, $D_{SFQ} = 30$

Figure 13: **Fluctuating workloads.** Streams f and g both have the same number of Postmark threads on brick A, and stream g has 10 additional Postmark threads on brick B. In addition, there is a stream h that has 10 on/off threads on brick B that are repeatedly on together for 10 seconds and then off for 10 seconds. The weights are equal: $\phi_f : \phi_g : \phi_h = 1 : 1 : 1$.

and data layout are otherwise identical to the single coordinator experiment. The results indicate that HYBRID-DSFQ works as designed with multiple coordinators too.

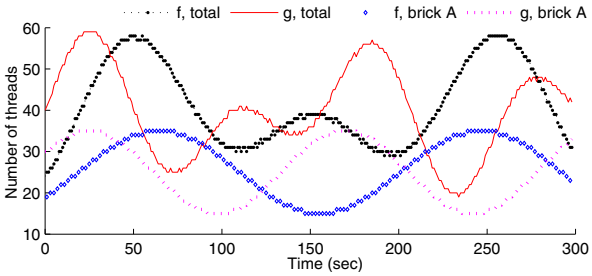
4.4 Fluctuating workloads

First we investigate how TOTAL-DSFQ responds to sudden changes in load by using an on/off fluctuating workload. Figure 13 shows the total throughputs of the three streams. Steams f and g are continuously backlogged at brick A and thus the total throughputs are the same. When stream h is on, some bandwidth on brick B is occupied by h (h's service is not proportional to its weight because of insufficient threads it has and thus it is not backlogged on brick B). As a result, g's throughput drops. Then f's throughput follows closely after a second, because part of f's share on A is reallocated to g to compensate its loss on B. Detailed throughputs of g on each brick is not shown on the picture. We also see that as the number of threads (and hence the SFQ depth) increases, the sharp drop in g's throughput is more significant. These experimental observations agree with the unfairness bounds on TOTAL-DSFQ shown in Theorem 2, which increase with the queue depth.

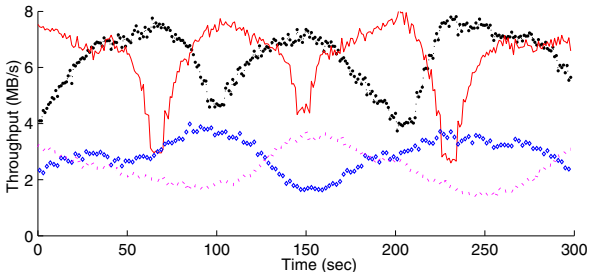
Next we examine the effectiveness of different proportional sharing algorithms through sinusoidal workloads. Both streams f and g access three bricks and overlap on one brick only, brick A. The number of Postmark threads for each stream on each brick is approximately a sinusoidal function with different frequency; see Fig. 14(a). To demonstrate the effectiveness of proportional sharing, we try to saturate brick A by setting the number of threads on it to a sinusoidal function varying from 15 to 35, while thread numbers on other bricks take values from 0 to 10 (not shown in Fig. 14(a)). The result confirms several hypotheses. Figure 14(b) is the result on a standard FAB without any fair scheduling. Not surprisingly, the throughput curves are similar to the thread curves in Fig. 14(a) except when the server is saturated. Figure 14(c) shows that single brick proportional sharing provides proportional service on brick A but not necessarily the total service. At time 250, the service on A is not proportional because g has minimum threads on A and is not backlogged. Figure 14(d) displays the effect of total service proportional sharing. The total service rates match well in general. At times around 65, 100, 150, and 210, the rates deviate because one stream gets too much service on other bricks, and its service on A drops close to zero. Thus TOTAL-DSFQ cannot balance the total service. At time around 230-260, the service rates are not close because stream g is not backlogged, as was the case in Fig. 14(c). Finally, Fig. 14(e) confirms the effect of hybrid proportional sharing. Comparing with Fig. 14(d), HYBRID-DSFQ proportional sharing guarantees minimum share when TOTAL-DSFQ does not, at the cost of slightly greater deviation from total proportional sharing during some periods.

5 Conclusions

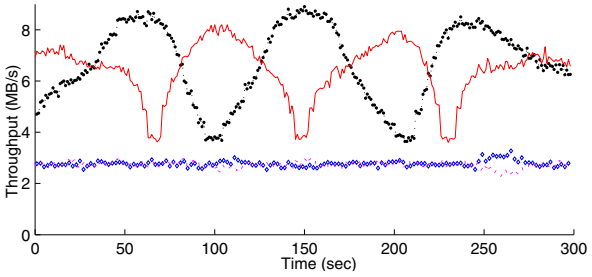
In this paper, we presented a proportional-service scheduling framework suitable for use in a distributed storage system. We use it to devise a distributed scheduler that enforces proportional sharing of total service



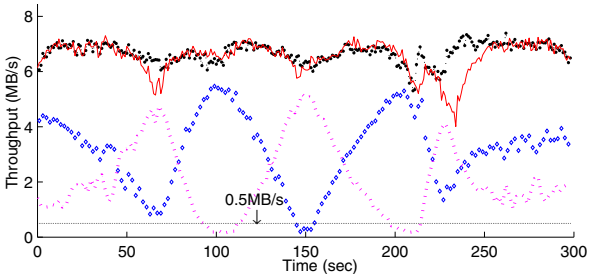
(a) Number of Postmark threads



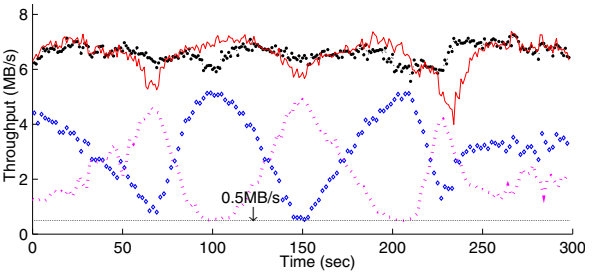
(b) Without any proportional sharing scheduling



(c) Single brick proportional sharing



(d) Total service proportional sharing



(e) Hybrid proportional sharing

Figure 14: **Sinusoidal workloads**, ϕ_f:ϕ_g=1:1. The legend in (a) applies to all the figures.

between streams to the degree possible given the workloads. Enforcing proportional total service in a distributed storage system is hard because different clients can access data from multiple storage nodes (bricks) using different, and possibly multiple, access points (coordinators). Thus, there is no single entity that knows the state of all the streams and the service they have received. Our scheduler extends the SFQ(D) [12] algorithm, which was designed as a centralized scheduler. Our scheduler is fully distributed, adds very little communication overhead, has low computational requirements, and is work-conserving. We prove the fairness properties of this scheduler analytically and also show experimental results from an implementation on the FAB distributed storage system that illustrate these properties.

We also present examples of unbalanced workloads for which no work-conserving scheduler can provide proportional sharing of the total throughput, and attempting to come close can block some clients on some bricks. We demonstrate a hybrid scheduler that attempts to provide total proportional sharing where possible, while guaranteeing a minimum share per brick for every client. Experimental evidence indicates that it works well.

Our work leaves several issues open. First, we assumed that clients using multiple coordinators load those coordinators equally or randomly; while this is a reasonable assumption in most cases, there may be cases when it does not hold — for example, when some coordinators have an affinity to data on particular bricks. Some degree of communication between coordinators may be required in order to provide total proportional sharing in this case. Second, more work is needed to design and evaluate better hybrid delay functions that can deal robustly with pathological workloads. Finally, our algorithms are designed for enforcing proportional service guarantees, but in many cases, requirements may be based partially on absolute service levels, such as a specified minimum throughput, or maximum response time. We plan to address how this may be combined with proportional sharing in future work.

6 Acknowledgements

Antonio Hondroulis and Hernan Laffitte assisted with experimental setup. Marcos Aguilera, Eric Anderson, Christos Karamanolis, Magnus Karlsson, Kimberly Keeton, Terence Kelly, Mustafa Uysal, Alistair Veitch and John Wilkes provided valuable comments. We also thank the anonymous reviewers for their comments, and Carl Waldspurger for shepherding the paper.

References

[1] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of ser-

vice guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, June 1999.

[2] D. D. Chambliss, G. A. Alvarez, P. Pandey, and D. Jadav. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, Oct. 2003.

[3] H. M. Chaskar and U. Madhow. Fair scheduling with tunable latency: A round-robin approach. *Proceedings of the IEEE*, 83(10):1374–96, 1995.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM: Symposium proceedings on Communications architectures & protocols*, Sept. 1989.

[5] Z. Dimitrijević and R. Rangaswami. Quality of service support for real-time storage systems. In *International IPSI-2003 Conference*, Oct. 2003.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2003.

[7] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Int. Conf. on Dependable Systems and Networks*, June 2004.

[8] P. Goyal, M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *ACM Transactions on Networking*, 5(5):690–704, 1997.

[9] A. Gulati and P. Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2005.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[11] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[12] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[13] V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. Knightly. Distributed priority scheduling and medium access in ad hoc networks. *Wireless Networks*, 8(5):455–466, Nov. 2002.

[14] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the 12th International Workshop on Quality of Service (IWQoS)*. IEEE, June 2004.

[15] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, Oct. 1997.

[16] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of ASPLOS*. ACM, Oct. 1996.

[17] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, Mar. 2003.

[18] H. Luo, S. Lu, V. Bharghavan, J. Cheng, and G. Zhong. A packet scheduling approach to QoS support in multihop wireless networks. *Mob. Netw. Appl.*, 9(3):193–206, 2004.

[19] A. K. Parekh and R. 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, 1993.

[20] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: Building distributed enterprise disk arrays from commodity components. In *Proceedings of ASPLOS*. ACM, Oct. 2004.

[21] D. C. Stephens and H. Zhang. Implementing distributed packet fair queueing in a scalable switch architecture. In *Proceedings of INFOCOM*, Mar. 1998.

[22] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*. IEEE, Aug. 2001.

[23] N. H. Vaidya, P. Bahl, and S. Gupta. Distributed fair scheduling in a wireless lan. In *MobiCom: Proceedings of the 6th annual international conference on Mobile computing and networking*, Aug. 2000.

[24] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[25] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[26] Y. Wang and A. Merchant. Proportional service allocation in distributed storage systems. Technical Report HPL-2006-184, HP Laboratories, Dec. 2006.

[27] W. Wilcke et al. IBM intelligent bricks project—petabytes and beyond. *IBM Journal of Research and Development*, 50(2/3):181–197, Mar. 2006.

[28] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–96, 1995.

[29] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel. An interposed 2-level I/O scheduling framework for performance virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2005.

[30] L. Zhang. Virtualclock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, 1991.