# The Design and Implementation of AQuA: An Adaptive Quality of Service Aware Object-Based Storage Device

Joel C. Wu Scott A. Brandt

*Department of Computer Science*
*University of California, Santa Cruz*
{*jwu,sbrandt*}*@cs.ucsc.edu*

## Abstract

*As storage systems are becoming larger and more complicated, they must handle an increasing variety of data storage needs. Storage system Quality of Service (QoS) has therefore become an important issue. Storage systems must be able to deliver satisfactory performance under both expected and unexpected workloads and, increasingly, must handle data (such as multimedia data) with real timeliness requirements. We are developing a petascale high-performance distributed object-based storage system called Ceph. Ceph's object storage architecture and other unique features have given rise to new QoS issues that require new solutions. In this paper, we first discuss the issues involved with the design of the overall QoS framework, we then present the design and implementation of AQuA, an adaptive QoS-aware object-based storage device (OSD) suitable for use as the fundamental building block in an overall QoS-capable object-based storage system.*

## 1. Introduction

With the explosive growth of digital information, storage systems are becoming larger and more complicated in order to store the ever-increasing amount and increasingly diverse types of data. The performance of the storage system is crucial to meeting the needs of both applications and users. To satisfy the performance requirements, in addition to developing higher-performing storage systems, the ability to *ensure* performance (provide Quality of Service (QoS) support) has become essential.

To provide consistent and usable performance, the storage system must be able to deliver satisfactory performance under expected (and unexpected) workloads. Many storage-bound applications such as multimedia, scientific, or visualization applications have timing constraints associated with their storage access [8]. These types of soft real-time applications can benefit from performance assurances. In addition, large storage systems are also likely to serve different groups of users and different workloads that have different characteristics and priorities. In such environments, it would be useful to have the ability to allocate performance (bandwidth) dynamically according to the needs of the current applications and users.

Without QoS support, different and unrelated workloads sharing the same storage system may interfere with each other's performance. To ensure Quality of Service, the storage must be provisioned in such a way as to ensure that it can meet the performance requirement of all expected workloads. However, storage workloads are dynamic and transient in nature, and provisioning for the worst case scenario is prohibitively expensive. Ensuring performance isolation by over-provisioning can be twice as expensive [18, 2]. Although automated design tools exist that can assist designers in provisioning the system [2, 3], they require detailed knowledge of the expected workload and system configuration, and are slow to adapt to changes. Adequate provisioning is essential but, in addition, it is highly desirable to have the ability to assure performance under different load conditions.

The object-based storage model [19] is a distributed storage architecture that overcomes many of the limitations of traditional distributed storage. It achieves high capacity, throughput, reliability, availability, and scalability. We are currently developing *Ceph*, a petabyte distributed object-based storage system optimized for high performance computing environments [26]. The three major components of Ceph are shown in Figure 1. The Client Interface (CI) provides file system API to applications on the client nodes. The Metadata Server (MDS) cluster is responsible for metadata operations such as namespace and directory management and authentication [28]. The Storage Managers (SMs) encapsulate low-level storage allocation, scheduling, and management within each OSD.

A design goal of Ceph is to have QoS capability that can provide performance assurances. Toward this end, there

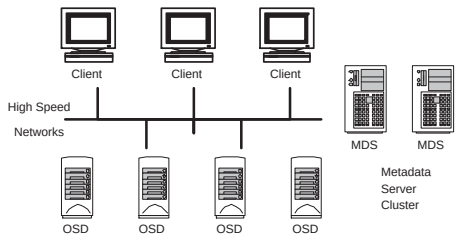



**Figure 1. Ceph's object-based architecture**

are two main aspects of Ceph's QoS architecture. First, each of the OSDs must be QoS-aware, and second, there must be an overall framework aggregates the performance assurances provided by the individual OSDs into an overall QoS architecture. In this paper, we first discuss the issues of the overall QoS framework. We then present the design and implementation of our adaptive QoS-aware OSDs, or AQuAs.

## 2. Storage QoS Architecture

The desire to build multimedia storage systems capable of video streaming [11] has generated considerable interest in distributed storage QoS. Generally, the goals of these systems are to ensure jitter-free delivery of continuous media data. In this context, the assurances provided are relatively fine-grained and at the transactional level, such as per disk request, per movie/file, or per client job (as in Swift [16]). One such example is the MARS project [5, 6]. To attain parallelism, many of these systems stripe data across different storage nodes. However, they are designed with homogeneous continuous media data in mind, and the storage architecture and the techniques they employ to achieve QoS often exploit the nature of video delivery and properties of continuous media data (mostly sequential access with identifiable frame rate) and can not be readily applied to systems with heterogeneous data or access patterns.

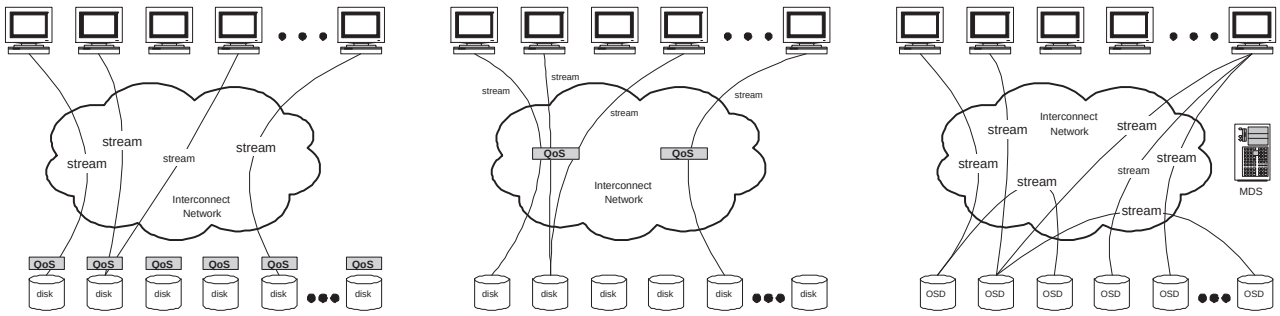
The type of QoS-capable distributed storage systems that we are interested are those designed for general purpose use (not multimedia-specific). The goal of the QoS mechanism in this context is to allocate bandwidth among different workloads. The assurance provided is often at a coarser grain, such as per workload or per initiator-target stream. While many distributed storage systems stripe data across different storage nodes, most QoS-aware non-multimedia specific distributed storage systems are not striped. They utilize a collection of *per device* mechanisms (a *device* in this context is a logical storage node). Although the file system spans multiple storage nodes, the QoS assurance are provided individually for each storage node (or for streams starting or terminating at a node). For example, Facade [18], Zygaria [29], and the GRIO [22] in XFS have a QoS mechanism for each node. The system described by Sundaram [24] associates QoS mechanism with a logical volume. Stonehenge [13] associates QoS attributes with a logical volume at volume creation time. SLEDS [7] has special gateways between client and storage nodes that associate each initiator-target streams with a QoS property (per-stream).

The use of per-device (or per-stream) QoS mechanisms is natural because traditional distributed storage systems store related data (such as a single large file) on the same storage node. However, on Ceph, files are broken up into objects and striped across different OSDs for load balancing and parallelism [12, 28]. The object distribution is expected to distribute the objects evenly across all of the OSDs [25] and a very large file may have its contents spread across the entire OSD cluster. Because of this feature, the locality of data as seen by the file system is destroyed above the device level. Therefore, for Ceph, it is not sufficient to have a collection of per-device QoS mechanisms, as multiple devices may be involved in even a single transfer request.

The difference in QoS mechanisms and data flow is depicted in Figure 2. Figure 2(a) shows a QoS framework with a collection of per-device QoS mechanisms. The per-stream mechanism for SLED is shown in Figure 2(b). Figure 2(c) shows the flow of data in Ceph, where a read or write stream between a client and a file can be broken up into multiple streams going between the client and different OSDs. In addition to the flow of data between clients and OSDs, there are also flows of data between OSDs. The OSDs in Ceph are intelligent and autonomous devices that have peer-to-peer capability and can independently perform data management functions such as replication and recovery.

Storage QoS is complicated by the stateful nature of disk scheduling, caused by the physical and mechanical properties of the disks themselves. The time it takes to serve a request not only depends on the location of that request, but also depends on the current location of the disk head as the result of the previous request. Consequently, the throughput of a disk depends on both its workload and data layout. True isolation of performance at the level of a shared disk arm is not possible, as different workloads sharing the same disk arm may interfere with each other. Thus, unlike network bandwidth, the amount of available disk bandwidth is not fixed. Storage QoS must contend with the issue of assuring resource allocation when the total amount of resource varies. The way different QoS mechanisms deal with the stateful nature of the disk provides us with another way to characterize the QoS mechanisms.

There are three ways to address this issue. The first method is proportional sharing. In proportional sharing of disk, each allocation gets a fraction of the disk's bandwidth.




(a) Per-device QoS mechanism (b) Per-stream QoS mechanism (SLEDS) (c) Data flow in Ceph

**Figure 2. Per-device and per-stream QoS mechanisms and their data flows vs. data flows in Ceph.**

The system described by Sundaram [24] uses the Cello [23] disk scheduler as the per-device mechanism. Cello allocates portions of the total disk bandwidth to different classes proportionally. YFQ [4] is another disk scheduler that uses proportional sharing. As the total disk bandwidth varies, the actual bandwidth received by each reserved portion also varies. Because proportional sharing can't make assurance in terms of the actual data rate, its usefulness is limited.

The second method is to use an estimated value as the total bandwidth for a disk. Admission control and performance assurance are made based on treating this value as the amount of the total resource, in the same way that network QoS is achieved over a link with fixed bandwidth. Typically, a profiling program is used to determine a fixed average bandwidth of the disk. DFS [1] and Zygaria [29] uses this approach, as well as the GRIOv2 [22] in XFS, which refers to this value as the "qualified bandwidth." The choice of this value is usually a trade-off between the "tightness" of the QoS assurance and the total throughput

The third way in which QoS mechanisms deal with the stateful nature of the disk is by adaptation. This method attempts to adapt to the changing total bandwidth. Regardless of the actual techniques implemented, it can be generalized into a *throttling model*. We present this throttling model in more detail, as it simplifies the overall storage QoS framework. There is actually a fourth approach for dealing with the stateful nature of the disk, by carefully profiling and extracting the disk parameters such as seek time, rotational latency, and access time. With these parameters, a disk model can be constructed in which the disk scheduler will try to calculate the exact service time of a request and schedule it accordingly. The disk scheduler is expected to anticipate and control every movement of the disk drive. This method is used by real-time disk schedulers [21] and is largely limited to special-purpose systems. Increasingly intelligent disk drives with increasingly many layers of intelligence and interface make this approach infeasible in our system [30] and we do not consider it further.

We now describe a generalized throttling approach that works by throttling workload streams. Generally, QoS can be abstracted and viewed as redistributing resource among consumers of the resource to meet certain criteria. The throttling approach is essentially a feedback-based technique that allocates bandwidth based on a comparison of the current loading condition with the desired loading condition. The rate of a stream is ensured by throttling the rate of competing streams. This throttling approach can be simplified and generalized into a generic model with four major components.

**Specification**—The *Specification* mechanism allows the declaration of the desired quality level that a particular entity (client, group of clients, class of applications, etc.) is to receive, and the system attempts to assures that the entity receives this minimum level of service. SLED [7] uses average response time (IOPS, bytes/sec) as its specification. Facade [18] uses a pair of curves specifying read and write latency as a function of request rate.

**Monitor**—The *Monitor* is the component that monitors the rate at which different entities are receiving service. Different parameters of the system can be monitored (and the values compared to the specification) to extract the status of the system and derive the actions to be taken. Status such as queue length, average completion time, response time, as well as throughput can be monitored. SLEDS monitors I/O service time; Facade monitors a number of statistics including I/O arrival and completions, average latency, and read/write request arrival rates.

**Enforcer**—The *Enforcer* is the mechanism that shapes bandwidth by throttling resource consumption. To manage bandwidth by rate-limiting, the system must have the ability to place caps on over-demanding clients. Facade uses an I/O scheduler, and SLED uses leaky buckets to control the rate that SCSI commands can be issued.

**Controlling Technique**—The *Controlling Technique* decides how much and when to throttle the bandwidth of over-limit clients as well as how much and when to release the throttling. Triage [14] uses a control-theoretic approach; Facade and SLED uses heuristic-based approaches.

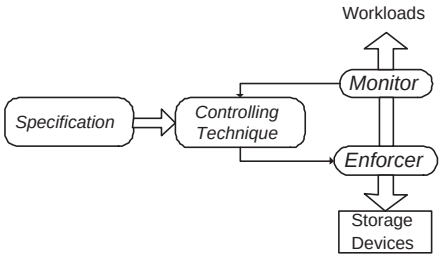



**Figure 3. The throttling approach**

Figure 3 shows the components of this throttling-based approach. These systems work by allowing specifications to be associated with workload streams. The actual performance is monitored. If some entities are not receiving their desired bandwidth, the enforcer throttles back the bandwidth of competing over-limit entities' streams in order to boost the bandwidth of the under-limit entities. The details of the throttling are determined by the controlling technique.

## 3. Ceph Global QoS Objective

Most traditional QoS mechanisms allow the partitioning of performance of one single storage node at a time. In Ceph, the traditional association between a file and storage node is destroyed because objects are expected to spread evenly across OSDs. Any object can potentially end up on any OSD, and a large file will have its content spread across many OSDs. This property of Ceph requires that the QoS framework move beyond the local per-device QoS mechanisms; global coordination is required. This section describes the global QoS framework that provides the context for our AQuA QoS-aware OSDs. The design and implementation of AQuA is described in the next section.

¡¡¡¡¡¡¡ msst06-qos.tex The QoS objective for Ceph is global performance partitioning. The Ceph QoS framework is intended to allow the storage system to be partitioned into different performance *classes*. A class is an aggregate of storage traffic sharing the same QoS attribute, analogous to a traffic class in DiffServ for network QoS. The semantics and granularity of the classification is administrator-defined. It can be applications, groups of applications, groups of users, type of task, etc. Regardless of the scheme, each request to the underlying storage is tagged by its class identifier. The tagging of request class is done by the client component.

Consider the scenario where the storage system is used in a storage-intensive high priority distributed scientific application. While the scientific application is running, there are other less important background tasks that must be performed such as backup or copying the results to another system. However, these less important tasks may negatively impact the performance of the scientific application. The current practice is to suspend all other tasks while important high-performance applications are running. The Ceph QoS architecture allows the administrator to assure the bandwidth for the scientific application at the global level, without going to each storage device individually for resource allocation. Other tasks can proceed at the same time without negatively impacting the performance of the scientific application ======= The QoS objective for Ceph is global performance partitioning. The Ceph QoS framework is intended to allow the storage system to be partitioned into different performance *classes*. A class is an aggregate of storage traffic sharing the same QoS attribute, analogous to a traffic class in DiffServ [9] for network QoS. The semantics and granularity of the classification is administrator-defined. It can be applications, groups of applications, groups of users, type of task, etc. Regardless of the scheme, each request to the underlying storage is tagged by its class identifier. The tagging of request class is done by the client component.

Consider the scenario where the storage system is used in a storage-intensive high priority distributed scientific application. While the scientific application is running, there are other less important background tasks that must be performed such as backup or copying the results to another system. However, these less important tasks may negatively impact the performance of the scientific application. The current practice is to suspend all other tasks while important high-performance applications are running. The Ceph QoS architecture allows the administrator to assure the bandwidth for the scientific application at the global level, without going to each storage device individually for resource allocation. Other tasks can proceed at the same time without negatively impacting the performance of the scientific application ¿¿¿¿¿¿¿ 1.33

While the potential total throughput of the system can be determined under ideal situations, in reality it is difficult to assure a specific bandwidth to a class by just using the throttling of the QoS mechanism alone. Although objects are expected to be distributed evenly across all OSDs, in heterogeneous workloads the popularity of objects will vary and at times only a subset of the total OSDs will be involved in data transfer. As a result, the total throughput will vary and be determined by workload of the subset of OSDs involved. Because of this property, the per-class assurances are administrator configured, and we expect the administrator to tune the system to obtain optimal results.

Figure 4 shows the proposed QoS architecture of Ceph. As viewed from the client at the file system level, the QoS layer encapsulates the complexity of the striping with the enforcement done at the OSDs.

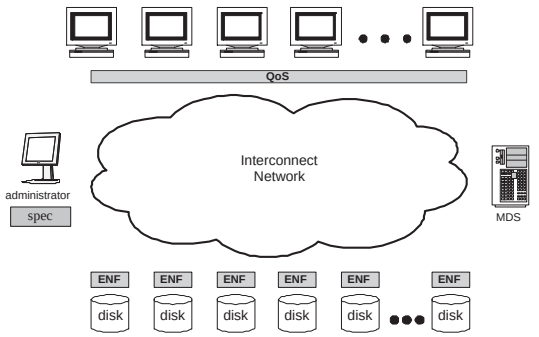



**Figure 4. The global QoS framework encapsulates the striping and the bandwidth sharing is enforced at the OSDs.**

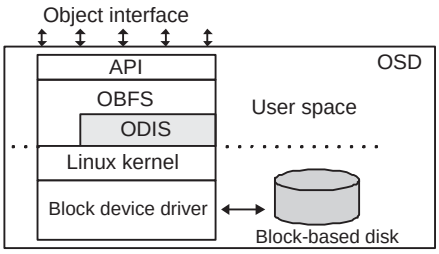



**Figure 5. Internal Structure of an AQuA OSD with the Object Disk I/O Scheduler (ODIS)**

## 4. Ceph QoS-Aware OSD

The basic building blocks of the Ceph system are the object-based storage devices (OSDs). Each OSD is an intelligent and autonomous storage device that offloads disk block allocation and scheduling functions from the traditional storage server and provides an abstract data object interface to the client system software. All local storage issues are managed within the OSDs. The QoS capabilities of the OSDs are used to support the overall global QoS framework. Ceph's OSDs are built with commodity block-based hard drives; each OSD consists of a CPU, network interface, local cache, and storage device (disk or small RAID configuration).

The block-based hard disk in the OSD is managed by Object-Based File System (OBFS) [25], a small and efficient file system designed to optimize disk layout (and thereby performance) based on the expected workloads of Ceph. OBFS manages all low-level storage functions and presents an object interface to the clients. OBFS is implemented in the Linux 2.6 user space and manages the raw block-device directly. The initial implementation of OBFS uses a simple elevator scheduler for disk request scheduling.

The fundamental capability provided by the AQuA is to throttle traffic. Regardless of the higher-level QoS goals, eventually they can all be decomposed into why, when, how, which, and how much to shape the disk traffic. The underlying QoS framework simply shapes the traffic to support the higher level QoS goals.

All requests arriving at the OSD are associated with a class ID. Requests with invalid or no class ID are put into a default best-effort class. The determination of the class ID for the requests are done outside of the OSD. To add the traffic shaping capability, we replaced the elevator scheduler in OBFS with a QoS-aware scheduler ODIS (Object Disk I/O Scheduler). The internal structure of the OSD is shown in Figure 5.

## 5. The OSD I/O Scheduler

The QoS-assuring capability of OBFS is provided by ODIS (Object Disk I/O Scheduler). ODIS enables the OSD's ability to allocate bandwidth between different classes of traffics. This section describes the design and implementation details of ODIS.

### 5.1. Reservation

The generic term *class* is used to represent a collection of requests with the same QoS requirements (a QoS entity). This classification of requests allows the differentiated handling of disk traffic. ODIS allows reservations to be made. A reservation request includes a class identifier, the requested bandwidth, and an optional duration parameter. When an OSD receives a reservation request, admission control is performed to determine if it can support this new reservation. The admission control is a simple linear additive admission control based on a pre-determined value of the total disk bandwidth. The choice of this value is discussed further in Section 5.4.

ODIS maintains the reservation information. For each admitted reservation, ODIS creates the associated data structure and thread. Taking advantage of the computing power available at the OSD, the ODIS scheduler utilizes multi-threading to increase parallelism. ODIS maintains a QoS queue and a thread for each class reservation. The functions `quofs_reserve()` and `quofs_release()` are used to explicitly setup and remove the reservations. Reservations can also be removed automatically if an expiration time is supplied.

At the top level, data on an OSD is accessed through OBFS's `quofs_read()` and `quofs_write()` calls. The calls contain a parameter indicating the QoS class of the request.

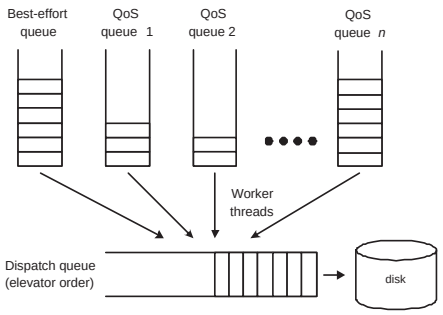



**Figure 6. ODIS structure**

## 5.2. Queueing Structure

Requests going to the disk are sorted into different QoS queues according to their class ID. A default best-effort class with an ID of 0 aelways exist for best-effort requests (and any requests with invalid class IDs). For an OSD with $n$ class reservations, there will be $(n+1)$ QoS queues. The bandwidth requirement information is maintained for each queue. In addition to the QoS queues, there is a sorted queue (dispatch queue). Figure 6 shows the internal structure of ODIS.

As requests arrive, they are sorted into their respective QoS queues. The best-effort queue is also considered a QoS queue since it has a minimal bandwidth requirement for starvation prevention. Each QoS queue has a worker thread that moves the request from the QoS queue to the dispatch queue. The QoS requirement is enforced by controlling the rate at which requests are moved from each of the QoS queues to the dispatch queue. Requests are inserted into the dispatch queue in elevator order. The request at the head of the dispatch queue is removed and sent to the disk as fast as the disk allows. This is done by a worker thread associated with the dispatch queue. The sorting of incoming requests into different queues by class ID and the controlling of how fast requests from each of the QoS queue are moved to the dispatch queue forms the basis of the bandwidth assurance framework.

## 5.3. Token Bucket Implementation

Conceptually, the queueing structure slices up the total bandwidth of the disk and assigns them to different classes. The QoS queues rate-limit the bandwidth each class can obtain according to its reservation, preserving the desired sharing and preventing interference between different classes of traffic. However, under realistic usage, not all reserved bandwidth will be used. ODIS must be able to reclaim reserved but unused bandwidth, as well as unreserved bandwidth. This is accomplished by using hierarchical token bucket filter (HTB). The principle of hierarchical sharing [10] allows a class to receive more than its reserved share if there is unused bandwidth available.

¡¡¡¡¡¡¡ msst06-qos.tex Using HTB, a disk's bandwidth is divided between classes. Each leaf node is associated with a class and represents a point of control for accessing the disk. In our implementation a leaf node corresponds to a QoS queue. When a reservation for a class is first made, a QoS queue (leaf node) is created and added to the tree structure. When the reservation expires or is deleted, its QoS queue (and its corresponding place in the tree) is removed.

The token bucket filters serve to regulate disk bandwidth. In order to move a request from a QoS queue to the dispatch queue, the class associated with the queue must possess enough tokens. In ODIS, each token represents 1 KB of data, meaning a request for 256 KB of data requires 256 tokens. Each leaf node in the tree has an associated bucket, which may hold up to a maximum number of client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. The root (global) token rate represents the entire bandwidth of a disk. For example, if a disk supports an average throughput of 30 MB/second, the root token rate $T_0 = 30K$ tokens/second. ======= Using HTB, a disk's bandwidth is divided between classes. Each leaf node is associated with a class and represents a point of control for accessing the disk. In our implementation a leaf node corresponds to a QoS queue. When a reservation for a class is first made, a QoS queue (leaf node) is created and added to the tree structure. When the reservation expires or is deleted, its QoS queue (and its corresponding place in the tree) is removed.

The token bucket filters serve to regulate disk bandwidth. In order to move a request from a QoS queue to the dispatch queue, the class associated with the queue must possess enough tokens. In ODIS, each token represents 1 KB of data, meaning a request for 256 KB of data requires 256 tokens. Each leaf node in the tree has an associated bucket, which may hold up to a maximum number of tokens. When a client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. The root (global) token rate represents the entire bandwidth of a disk. For example, if a disk supports an average throughput of 30 MB/second, the root token rate would be $30K$ tokens/second. ¿¿¿¿¿¿¿ 1.33

Since every class has its own supply of tokens, ODIS guarantees that every class is always able to freely request up to its reservation in disk requests. The sharing of unused bandwidth is facilitated by the token bucket of the root node. In addition to its own tokens, a class can consume any extra bandwidth by using tokens from the root.

The effect is that unused bandwidth is shared among the classes.

When a class drains tokens from its own bucket, it also drains from the root bucket. The result is that when a class makes disk requests, the root tokens will be drained as well. If some of the classes are not fully using their reservation, the root will have surplus tokens. These tokens are available to other classes when their own supply runs out, so that a class that has exceeded its reservation may still be able to proceed. This hierarchical structure provides a method for a class to use unreserved bandwidth, as well as reclaim bandwidth that is reserved, but not used, by others.

### 5.4. Trade off between QoS assurance and total throughput

The throughput of a disk during overload depends on the workload. Since the totality of resource may vary, proportional sharing is often used to reserve and allocate a fraction of the bandwidth to different shares [23, 4]. However, because the total disk bandwidth varies, these fractional guarantees do not result in fixed data rate. In many circumstances it would be more useful to have assurances in terms of data rates, and not in terms of a fixed fraction of a dynamically changing total bandwidth.

ODIS is able to provide assurance in terms of data rates with *absolute* reservations and knowledge of the root node bandwidth. However, the unpredictable nature of storage also presents a problem for ODIS. Thus far we have assumed that the disk bandwidth is fixed. Although convenient, this assumption is not true; the throughput of a disk during overload depends on the workload. ODIS uses a conservative estimate of total bandwidth. This underutilizes the disk, but allows ODIS to provide the reserved bandwidth. The more aggressively ODIS utilizes the disk, the less effective it is at providing isolation and sharing. Trade-offs must be made between ”tightness” of the QoS assurance and total throughput.

The GRIOv2 (Guaranteed Rate I/O version 2) [22] mechanism of XFS also allows applications to reserve a specific bandwidth from the file system in terms of actual data rates. Similar in concept to our global token rate, GRIOv2 requires the specification of a ”qualified bandwidth,” which is defined as ”the maximum I/O load that it (file system) can sustain while still satisfying requirements for delivered bandwidth and I/O service time.” This value is to be set by the administrator and it is suggested that this value be determined iteratively from running realistic test workloads, monitoring the performance, and refining the value. In a large storage system with varying workloads, this is unlikely to be optimal.

### 5.5. Bandwidth Maximizer

¡¡¡¡¡¡¡ msst06-qos.tex We seek to address this problem through the use of a *bandwidth maximizer*. The object distribution causes objects in each OSD to have little inter-object locality. The workload variation experienced within a single OSD would be less as locality in data is destroyed intentionally for system-wide load-balancing purposes, and small accesses are absorbed by caching at the client. Given these properties, the approach taken by AQuA is a combination of: (1) Conservative estimate of the global token rate to ensure stringent QoS assurance; and (2) Minimizdinge underutilization of disk with adaptive heuristics. A conservative estimate of the global token rate is used for admission control and bandwidth assurance. However, the total throughput of the disk is not limited by this value. The bandwidth maximizer will attempt to increase the aggregate throughput as long as the QoS assurance are not violated. ======= We seek to address this problem through the use of a *bandwidth maximizer*. The object distribution causes objects in each OSD to have little inter-object locality. The workload variation experienced within a single OSD would be less as locality in data is destroyed intentionally for system-wide load-balancing purposes, and small accesses are absorbed by caching at the client. Given these properties, the approach taken by AQuA is a combination of: (1) Conservative estimate of the global token rate to ensure stringent QoS assurance; and (2) Minimizing underutilization of disk with adaptive heuristics. A conservative estimate of the global token rate is used for admission control and bandwidth assurance. However, the total throughput of the disk is not limited by this value. The bandwidth maximizer will attempt to increase the aggregate throughput as long as the QoS assurance are not violated. ¿¿¿¿¿¿¿ 1.33

To derive an estimate of global token rate, we use a small program that profiles the disk to determine a conservative global token rate. The profiling program simulates the type of expected workloads and determines a conservative estimate of the disk throughput. The workload it simulates is tunable by the system administrator and the global token rate for ODIS can be modified by distributing the new value to the OSDs.

As already stated, the maximum achievable throughput of a disk at any given time is dependent on the workload. The global token rate represents a conservative estimate of this value. The hypothetical maximum achievable throughput at any given time may be higher or lower than the global token rate. Let $G$ denote the maximum total throughput as specified (and limited) by the global token rate, and let $A$ denote the hypothetical maximum achievable total throughput under a given workload. Figure 7 depicts a hypothetical scenario showing the relationship between $G$ and $A$. With

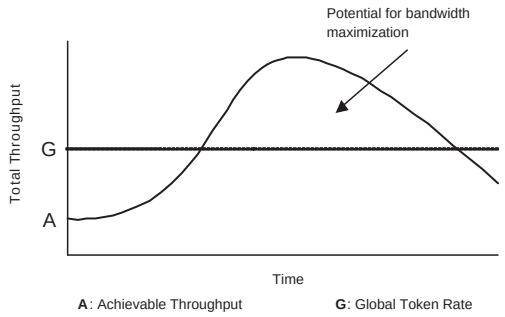



**Figure 7. Global token rate and the hypothetical maximum achievable rate**

a conservative estimate of $G$, we expect $G < A$ most of the time.

Let $O$ be the actual total throughput experienced by the disk (the observed rate). Its upper bound will be limited by the value of $G$ and $A$: $O \leq min(G,A)$. There are three scenarios of interest:

1. When the demand is not capped by either $A$ or $G$. $O < A$ and $O < G$
2. When the demand is capped by $A$, and $A < G$
3. When the demand is capped by $G$, $G < A$.

In the first scenario, when the demand on disk bandwidth is less than the supply, no throttling is needed and all requests are able to proceed at disk speed. Even if one or more class is demanding more than their reservation, there is sufficient spare bandwidth available (at the root node) to satisfy the extra demands. In this case, $(O \leq G)$ and $(O \leq A)$. The value of $G$ does not result in underutilization of the disk given the demand.

In the second scenario, the actual maximum achievable throughput under the given workload is less than the global token rate, and the actual total throughput is limited by this value. In this scenario, the reservation mechanism has over-committed, and the QoS assurance may or may not hold. For example, if all the reservations are demanding their maximum reserved rate, there is simply not enough resource (tokens) to go around. But it may be possible that one reservation is using less than its reserved share, and the other reservation is using more than its reserved share (unused share from the first reservation is reclaimed). In this case the QoS assurance would still be met. In either case, the total throughput of the disk is limited by $A$. Although the QoS assurance may be compromised, the disk is not underutilized.

The last scenario occurs when $G < A$ and $O$ is capped by $G$. In this case, the actual throughput of the disk is limited by the value of the global token rate. Some requests are throttled to maintain the QoS assurance, but the possible maximum achievable throughput is higher than that specified by the global token rate. In this case it is desirable to overcome the limit of $G$ and maximize the throughput. This potential is depicted as the center area in in Figure 7. To maximize disk throughput without violating the QoS goals, the bandwidth maximizer—an adaptive heuristic—is used. The bandwidth maximizer attempts to maximize the throughput by dynamically increasing the global token rate. By increasing the global token rate, nodes that have demands beyond their allocated tokens can borrow from the root token bucket, increasing the total throughput. This is desirable as long as the QoS goals are not violated. The extra tokens are available for use by any classes that have demands beyond their reservation on a first-come-first-serve basis.

The heuristic adjusts $G$ by monitoring the status: If disk throughput is capped by $G$ and no QoS commitments are violated, it increase $G$. If disk throughput is capped by $G$, $G$ has been increased to greater than its original value, and some QoS commitments are violated, decrease $G$. $G$ will not drop below its original value.

## 6. Experimental Results

Our test system uses a 1.5 GHz AMD Sempron 2200+ CPU with 512MB of RAM. It runs Linux kernel 2.6.11. The test disk is a Maxtor DiamondMax 6Y080P0. It is a 7200 RPM IDE drive with a capacity of 80GB and 8 MB of onboard buffer. The disk is divided into two equal partitions, the experiments are run using one of the partitions (placed under control of OBFS). The experiments are conducted using a synthetic workload generator that we developed (no real object workloads are currently available). The purpose of these experiments is to highlight the capabilities of our AQuA OSDs rather than to demonstrate exhaustively that the mechanisms work in all cases, and we therefore focus on a few illustrative examples. For the experiments, we use the large block size (512 KB) of OBFS because most small accesses are absorbed by the buffer at the client, and streaming and other applications that need bandwidth assurance typically access large files (which have large blocks).

We show the ability of ODIS to provide performance isolation by allocating portions of the disk to different classes of traffic. In this experiment, we have three separate threads all generating write requests to the OSD. The global token rate is set at 40 MB/s. Figure 8 shows the performance of OBFS without QoS assurance. The X-axis is the passage of time in seconds, and the Y-axis is the measured throughput by the synthetic application program. All the workloads attempt to write as much as possible and start at 30 second interval. As more workloads are introduced, we

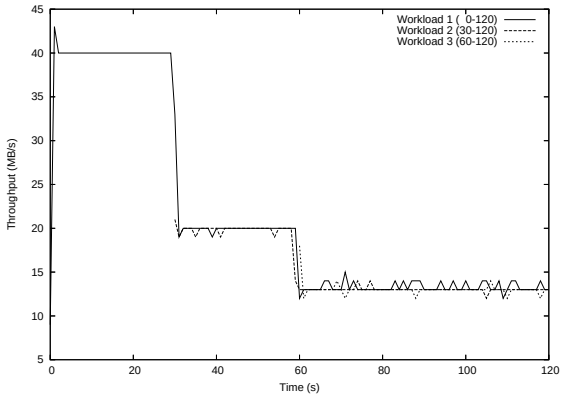



**Figure 8. No QoS reservation**

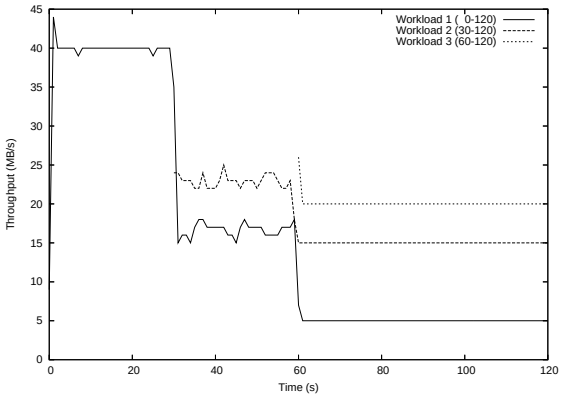



**Figure 9. With QoS reservation at 5 MB/s, 15 MB/s, and 20 MB/s for workloads 1, 2, and 3.**

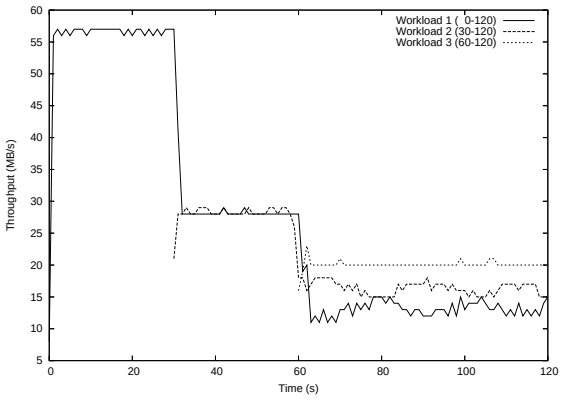



**Figure 10. Same experiment as depicted in Figure 9, but with bandwidth maximizer at work.**

can see that they interfere with each other's performance. The throughputs received by each of the workloads are approximately equal. There is little loss of total throughput as we move from one stream to three, this is due to the fact that the writes are at 512KB each, and with more streams, the disk interleaves requests from different streams.

In the next experiment shown in Figure 9, we make reservation for the three classes (workloads) at 5 MB/s, 15 MB/s, and 20 MB/s. From time 0 to time 30, workload 1 is the only workload experienced by ODIS, and although it only has a reservation of 5 MB/s, it is able to consume the unused bandwidth, resulting in a throughput of 40 MB/s, equal to the global token rate. At time 30, the second workload is introduced. They each receive more than their reserved share because there are unused tokens available. At time 60, the third workload is introduced. Under this condition, all tokens are being used up, and all three workloads are capped at their reserved rate and there is no interference.

Figure 10 shows the same experiment again, but this time with the bandwidth maximizer at work. We see that all QoS assurances are met, and the aggregate throughput of the entire disk is increased.

## 7. Related Work

The importance of storage QoS has been recognized. Our work builds and extends upon the contributions of many previous works. The hierarchical token bucket filter used in ODIS is based on the concept of hierarchical link sharing [10] from the networking field. However, the total available bandwidth for a network link is constant, while it varies for a disk according to the workload. Hence many QoS solutions developed for networking cannot be applied directly to disk. ODIS mitigates this issue with the bandwidth maximizer adaptation technique.

Disk schedulers are often used to manage the disk bandwidth in order to provide QoS assurance. These schedulers must satisfy QoS assurance in addition to minimize response time and maximize throughput. They range from special purpose real-time disk schedulers [20, 21] to mixed-workload disk schedulers that can allocate bandwidth among different classes of applications. It is generally accepted that there is a trade-off between QoS assurance and total throughput. The pure real-time disk schedulers assume that all requests have an associated deadline. The requests are typically ordered based on some combination of real-time scheduling techniques and disk seek optimization schemes (e.g. SCAN-EDF). They are more suitable for specialized systems such as multimedia server with homogeneous request types. The mixed workload schedulers such as Cello [23] and YFQ [4] use proportional sharing of the disk bandwidth, which can be an issue of assurance in data rate is required. Cello supports a limited and static numbers of classes, while YFQ allows for dynamic creation and deletion of classes.

A number of QoS-capable distributed storage systems have been developed. Systems such as Facade [18], SLED [7], Stonehenge [13], Zygaria [29], DFS [1], and GRIO [22] have already been mentioned and compared to our system in earlier sections of the paper, we will not describe them again in this section. Of these systems, Zygaria [29] is most similar to our work. It is a object-based storage system that uses the concept of a storage bricks. The Zygaria mechanism allows hierarchical sharing of individual bricks enforced with token buckets. It is a per device (brick) mechanism that assumes a fixed value for total disk throughput.

Lu *et al.* [17] presented a three level QoS specification for OSD and proposed extensions to the iSCSI specification to support the OSD QoS framework. Studies have also been done on how to communicate the QoS requirements to the OSD [15].

## 8. Conclusion and Future Work

¡¡¡¡¡¡¡ msst06-qos.tex Quality of Service is an important issue for large distributed storage systems. We first discussed the issues in achieving QoS on our Ceph distributed object-based storage system. We then presented the design and implementation of AQuA, an adaptive QoS-aware OSD, and present proof-of-concept results demonstrating the effectiveness of AQuA's mechanisms. AQuA combines the ODIS disk scheduler with the OBFS file system, enabling the differentiated handling of disk traffic belonging to different classes. The ODIS disk scheduler is implemented with hierarchical token buckets. It is able to rate-limit disk requests and reclaim unused bandwidth. The total disk throughput can be increased by the use of the bandwidth maximizer, a heuristic algorithm that attempts to maximize total disk throughput without violating the QoS goals. Ultimately, AQuA OSDs are intended to serve as the basic building blocks in an overall QoS framework for the Ceph object-based storage system.

Future work on AQuA will shift from utilizing OBFS to EBOFS (Extend-Based Object File System) [27]. EBOFS is a new object-based file system that will form the mainstay of future Ceph OSD. With EBOFS, more testing will be conducted and analyzed to better understand the characteristics of an AQuA OSD. In addition, the global QoS framework presented in the first part of the paper will be implemented.

The per-class assurance currently implemented by AQuA allows performance to be partitioned among different classes of traffic. However, if a class is sufficiently general, overloading within a class is possible and there may be times in which it is desirable to assure a particular traffic flow within a class. To provide more flexibility, we will investigate the benefit of adding finer-grained transaction-style QoS assurance to AQuA. A transaction is a particular sequence of read or write request issued by a client (similar to a*client job* in Swift [16]). It is associated with a class and uses the resource of that class. Because of the use of hierarchical sharing, the mechanism enabling *per-class* and *per-transaction* assurance should fit together seamlessly as a two layered hierarchical sharing. ======= Quality of Service is an important issue for large distributed storage systems. We first discussed the issues in achieving QoS on our Ceph distributed object-based storage system. We then presented the design and implementation of AQuA, an adaptive QoS-aware OSD, and present proof-of-concept results demonstrating the effectiveness of AQuA's mechanisms. AQuA combines the ODIS disk scheduler with the OBFS file system, enabling the differentiated handling of disk traffic belonging to different classes. The ODIS disk scheduler is implemented with hierarchical token buckets. It is able to rate-limit disk requests and reclaim unused bandwidth. The total disk throughput can be increased by the use of the bandwidth maximizer, a heuristic algorithm that attempts to maximize total disk throughput without violating the QoS goals. Ultimately, AQuA OSDs are intended to serve as the basic building blocks in an overall QoS framework for the Ceph object-based storage system.

Future work on AQuA will shift from utilizing OBFS to EBOFS (Extend-Based Object File System) [27]. EBOFS is a new object-based file system that will form the mainstay of future Ceph OSD. With EBOFS, more testing will be conducted to better understand the characteristics of an AQuA OSD. In addition, the global QoS framework presented in the first part of the paper will be implemented.

The per-class assurance currently implemented by AQuA allows performance to be partitioned among different classes of traffic. However, if a class is sufficiently general, overloading within a class is possible and there may be times in which it is desirable to assure a particular traffic flow within a class. To provide more flexibility, we will investigate the benefit of adding finer-grained transaction-style QoS assurance to AQuA. A transaction is a particular sequence of read or write request issued by a client. It is associated with a class and uses the resource of that class. Because of the use of hierarchical sharing, the mechanism enabling *per-class* and *per-transaction* assurance should fit together seamlessly as a two layered hierarchical sharing. ¿¿¿¿¿¿¿ 1.33

## Acknowledgments

This research was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. We are also grateful for our sponsors Intel Cor-

poration and Veritas Software. Thanks to Feng Wang for assistance with OBFS.

## References


[1] C. Akinlar and S. Mukherjee. Bandwidth guarantee in a distributed multimedia file system using network attached autonomous disks. In *Proceedings of the IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 237, Washington, D.C., 2000.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, 2001.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies*, pages 175–188, Berkeley, CA, USA, 2002. USENIX Association.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[5] M. Buddhikot and G. Parulkar. Efficient data layout, scheduling and playout control in mars. *ACM/Springer Multimedia Systems Journal*, 5(3):199–211, 1997.

[6] M. M. Buddhikot, G. M. Parulkar, and J. R. Cox, Jr. Design of a large scale multimedia server. *Journal of Computer Networks and ISDN Systems*, 27(3):503–517, 1994.

[7] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 109–118, 2003.

[8] Z. Dimitrijevic and R. Rangaswami. Quality of service support for real-time storage systems. In *Proceedings of the International IPSI-2003 Conference*, October 2003.

[9] S. B. et al. *An Architecture for Differentiated Services*. IETF DiffServ Working Group, December 1998.

[10] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[11] J. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, 1995.

[12] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[13] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[14] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. Technical Report HPL-2004-40, HP Laboratories, March 2004.

[15] K. KleinOsowski, T. Ruwart, and D. Lilja. Communicating quality of service requirements to an object-based storage device. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST 2005)*, April 2005.

[16] D. D. E. Long and M. N. Thakur. Scheduling real-time disk transfers for continuous media applications. In *Proceedings of the 12th IEEE Symposium on Mass Storage Systems (MSST 1993)*, April 1993.

[17] Y. Lu, D. H. Du, and T. Ruwart. Qos provisioning framework for an osd-based storage system. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST 2005)*, April 2005.

[18] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *USENIX Conference on File and Storage Technology (FAST'03*, 2003.

[19] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–900, August 2003.

[20] A. L. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of ACM Conference on Multimedia*, pages 225–233. ACM Press, 1993.

[21] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*. IEEE, December 2003.

[22] SGI. Guarantee-rate i/o version 2 guide, 2004.

[23] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.

[24] V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *Proceedings of 11th International Workshop on Quality of Service (IWQoS 2003)*, pages 479–497, June 2003.

[25] F. Wang, S. A. Brandt, E. L. Miller, and D. D. Long. OBFS: A file system for object-based storage devices. In *Proceedings of 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, April 2004.

[26] F. Wang, Q. Xin, B. Hong, E. L. Miller, D. D. Long, S. A. Brandt, and T. T. McLarty. File system workload analysis for large scientific computing applications. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST 2004)*, April 2004.

[27] S. A. Weil. Leveraging intra-object locality with ebofs. UCSC CMPS-290S Project Report, May 2004.

[28] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004. ACM.

[29] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: storage performance as a managed resource. In *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 06)*, April 2006.

[30] J. C. Wu and S. A. Brandt. Storage access support for soft real-time applications. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, Toronto, Canada, May 2004.