Quota enforcement for high-performance distributed storage systems

Kristal T. Pollack, Darrell D. E. Long, Richard A. Golding, Ralph A. Becker-Szendy
IBM Almaden Research Center, San Jose, CA

Benjamin Reed*
Yahoo Research, Sunnyvale, CA

Abstract

Storage systems manage quotas to ensure that no one user can use more than their share of storage, and that each user gets the storage they need. This is difficult for large, distributed systems, especially those used for high-performance computing applications, because resource allocation occurs on many nodes concurrently. While quota management is an important problem, no robust scalable solutions have been proposed to date. We present a solution that has less than 0.2% performance overhead while the system is below saturation, compared with not enforcing quota at all. It provides byte-level accuracy at all times, in the absence of failures and cheating. If nodes fail or cheat, we recover within a bounded period.

In our scheme quota is enforced asynchronously by intelligent storage servers: storage clients contact a shared management service to obtain vouchers, which the clients can spend like cash at participating storage servers to allocate storage space. Like a digital cash system, the system periodically reconciles voucher usage to ensure that clients do not cheat by spending the same voucher at multiple storage servers. We report on a simulation study that validates this approach and evaluates its performance.

1. Introduction

Tracking and enforcing resource usage limits in a large distributed system is difficult because it requires maintaining a consistent view of total usage when consumption is occurring in several places concurrently. Storage quota enforcement mechanisms are essential for shared storage, especially when resources are limited. In a file system, for example, users must not use more than their storage quota. In the case of scientific applications that involve tens of thousands of nodes all cooperating on a problem, all writing to shared files, nodes often consume from the same pool of quota in order to bound the amount of storage that a runaway application can consume.

We concentrate here on distributed storage systems that use storage servers that provide intelligence similar to object storage, which track local storage allocation and enforce access control. For these types of systems clients make storage requests on behalf of users, receiving and caching metadata and capabilities from management servers in order to perform I/O requests directly with the storage servers. Many systems with this type of distributed architecture have been proposed [1, 2, 6, 7, 12, 14, 23]. These provide scalable methods for security, load balancing, reliability, and allocation for this type of architecture; a scalable solution for quota management has not yet been proposed. We focus on a distributed quota management technique that can be used for enforcing resource usage limits for systems such as these.

In order to enforce quota a storage system must verify before each resource allocation that the requesting user can consume the required storage space without exceeding their assigned quota. For local or network file systems such as CIFS [17] and NFS [20] this can be easily verified since requests must go through a single machine, where quotas can then be checked. Similar techniques apply to distributed or cluster file systems such as SanFS [12], GPFS [19], and AFS [9] that use block storage devices, as quota changes imply allocation changes, and block allocation is typically handled centrally or by delegation from a central entity, so quota can be tracked together with allocation. In the case of object-based storage devices allocation is completely decentralized to the storage servers. For such a system, requiring that all quota update requests be coordinated through a central server at allocation time is an unacceptable bottleneck. Quota enforcement decisions must be able to occur in parallel, across multiple machines, even when requests are for the same user.

*Work done while at IBM Almaden Research Center

We propose a novel approach to tracking and enforcing resource limits. This approach borrows from digital cash mechanisms: there is a management server that acts as a *bank* that issues *vouchers* to clients, which the client can spend to allocate resources on any server they want. The client can withdraw enough vouchers to cover their needs for some period, during which time the client does not need to contact the bank. Servers are able to check the vouchers for authenticity. The vouchers are valid for a limited time, in order to handle clients that fail, and servers periodically reconcile their transactions with the bank to check that clients have behaved correctly. Since storage servers operate independently it is possible for a misbehaving client to cheat by spending the same voucher at multiple storage servers. However, we provide mechanisms similar to digital cash systems that allow us to detect cheaters within a guaranteed amount of time and bound the amount of storage they can allocate during this period.

This approach provides a different trade-off than other quota management methods. It provides excellent performance—in most cases indistinguishable from not tracking resource usage at all—while providing byte-accurate but temporally-coarse accuracy similar to time-limited escrow. It reduces load on the tracking service well below that of other mechanisms, thereby achieving excellent scalability. It also decouples quota tracking from allocation policy, so that the quota server *only* needs to track how much quota a user has consumed, and does not need to be concerned with where that consumption will occur, which reduces the load on the quota server and makes it easier to partition the quota tracking work across multiple servers. Further, each client can decide for itself where to allocate resources based on its own needs, which allows a client to customize its allocation based, for example, on how a particular file will be used.

2. System context

Figure 1 shows the architecture of the distributed storage systems we are investigating. In these systems, *clients* act on behalf of users. The clients communicate with a file system *management service* cluster to locate files and authorize actions. The authorization includes both checking permission to access data and permission to consume resources. The authorization is expressed using location-independent *capabilities* [13] and *vouchers*, which encode the client's rights to access files and to allocate resources respectively. Once the client has the capabilities and vouchers it needs, it communicates directly with *storage servers* to read and write data and to create and delete files. The storage server has the intelligence to manage internal resource allocation and to check capabilities for validity, similar to object store model [6, 10].

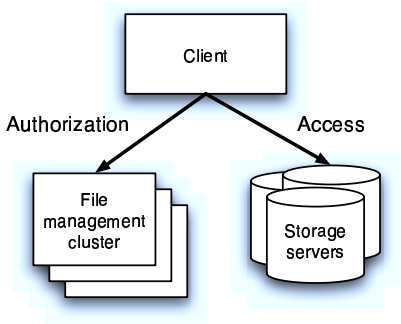

Figure 1. Basic distributed storage system architecture.

We are investigating quota management as part of the K2 distributed storage system [7]. K2 provides virtualized storage across multiple servers. Files are striped and/or replicated across servers, and a client can perform consistent I/Os to the pieces of a file. The system balances resource consumption between servers both as files are created, by placing files well, and later in response to changes in load, by migrating files as needed. K2 also provides security mechanisms to ensure both traditional file-level access control and authorization for management operations.

For scalability reasons, K2 pushes decentralization as far as possible. Each node is an autonomous agent, acting in its own interest as much as possible while respecting community needs. We make this possible by each node acting as an enlightened rational agent, with algorithms that work to meet the node's needs while avoiding the "tragedy of the commons," [8] in which the limits on shared resources are not considered. In concrete terms for a storage system, this means that we want each client to be able to make its own allocation decisions that give the best result for the application the client is running (self-interest), while ensuring that users do not go over quota and that storage resources are not over-used (community interest).

Different files can have significantly different needs, and so the system allows a different layout for each file. One file may be located on a single storage server; another may be mirrored and striped across many storage servers. The client decides what the layout should be, based on expected needs derived either from application hints or inferred from file attributes. Peak file creation rates can be high in some scientific applications, and so it is important for good scalability to minimize the dependence on the shared file management service during file creation.

Scientific applications have characteristics different from what studies of end-user workstations have shown. The absolute numbers are several orders of magnitude larger: petabytes of data are being deployed now, with ag-

gregate transfer rates of gigabytes per second, files in the terabytes, all being accessed by tens of thousands of clients. The clients are cooperating to run one application, and both read- and write-share files. The applications are bursty, as the clients synchronize as they move through phases of a computation and write out checkpoints concurrently or read the results of previous phases. Some of the files are only temporary, for communication between computation phases, while others are results of days of computation and must be carefully protected.

Because K2 is built for large distributed environments, its design works to minimize the trust required in any one component—in particular, the client. While many clients may be part of a single homogeneous compute cluster, some clients will be different and potentially not under careful administrative care—for example, user workstations used for visualizing results. The system assumes that clients authenticate the users that run on them, and that the clients can provide evidence of that authentication when communicating with the file management service and with storage server [11]. Our design assumes that clients can crash-fail. While some clients can also be malicious, we do not focus on them; we do provide mechanisms that ensure that cheaters will *always* be caught, and we bound the amount of storage a cheating client can allocate for a user.

3. Protocol operation

Here we give an overview of the operation of the voucher-based quota system. Figure 2 shows the general flow of usage. A client first requests a voucher for storage resources from the quota server for the user, then sends IO requests to storage nodes, including the voucher when those requests may consume resources. If a client frees resources on a storage server, the storage server gives the client a "refund" voucher for the amount freed. The quota server and storage nodes periodically reconcile the set of vouchers that have been spent against those that have been issued, in order to detect clients that overuse a voucher. We will describe in detail what a voucher is, how vouchers are obtained, and when vouchers are used. We then show that this method always provides temporally-coarse byte-level accuracy, even in the face of cheating and device failures.

Vouchers. A voucher is a record of a decision to allow a client to consume resources on behalf of a particular user. It is represented as a cryptographically-protected sequence of bytes:

$$\{epoch, expiry, user, value, serial\}_{auth}$$

similar to capabilities used to authorize actions in Amoeba [13] and the T10 OSD [10], the exact authorization mechanism is outside the scope of this paper. The

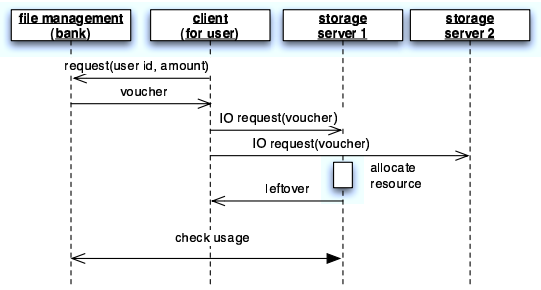

Figure 2. Sequence of operations. A client begins by obtaining a voucher for a user from the quota server, then spending that voucher during IO requests to different storage nodes. Later, the quota server and storage nodes check that the client did not overuse a voucher.

value encodes the amount of storage that may be purchased.Our system can track quota by the identity of the writer, or by the owner of the file; traditional file systems use the latter. The *user* field needs to encode the entity to which quota and allocation shall be charged, depending on that choice. The voucher has a unique *serial* number, which is used when storage servers reconcile voucher usage with the management server. Each voucher also records when it was issued (the *epoch*) and when it *expires*. The voucher includes a signature or MAC generated using a secret key known only to the management and storage servers, which ensures that a client cannot forge a voucher.

Each voucher is valid for only a limited duration, as recorded in its *expiry* field. This is used in handling failure—if a client crashes while holding an unused voucher, other clients can use that quota once the voucher expires—and in reconciling storage and management servers. These are discussed further below.

Obtaining vouchers. While a client could ask the management server for quota authorization on every I/O, this would put an unreasonable load on the management server. Instead, the client maintains a pool of vouchers, and only periodically communicates with the management server. The client tries to maintain enough vouchers to cover any allocation it expects to do in the near future, while allowing for other clients to share quota. This approach reduces load on the management server and improves client response latency. We can further reduce the load by piggy-backing quota requests on other metadata requests.

The client has to decide when to request vouchers and how much resource to ask for; the management server has to decide how much of that request to grant. The

management server must maintain the invariant that the vouchers granted for a user to any client—which represent potentially-used resources—plus the amount actually allocated do not exceed the user's quota.

The simplest policy for requesting vouchers would be a *fixed request* policy which always requests the same fixed amount of quota every time it must allocate more quota. The fixed amount should be larger than the expected average request size in order to amortize interactions with the management server over many I/O requests.

Another approach is an *adaptive request* policy where a client requests quota from the management server on a regular schedule based on its usage. This generally makes the load on the management server proportional to the number of clients, rather than to the intensity of workload on those clients. The client uses a moving average based on its history of recent resource consumption to estimate how much it will likely use between the current request and the next request, and asks the management server for the difference between the estimated usage and the vouchers it already has on hand. If actual usage is higher than anticipated, then the client will ask the management server for extra vouchers before its next scheduled request. The client can estimate the management server's response time and the short-term voucher usage rate to predict when to send a request to the management server before the client runs out of vouchers. The client may have extra vouchers when net consumption is lower than anticipated—perhaps because it has been freeing rather than allocating storage. In that case the client can return some vouchers to the management server or just let the vouchers expire, making the quota available for other clients. This matters, for example, when one client is cleaning up old files while other clients are writing new data.

The management server can determine how much to grant to a client based on its global information, including the total value of vouchers outstanding for a user and estimated demand from all clients consuming that user's quota. Granting more to a client can reduce the number of request messages that the management server must process, but giving too much to one client can inhibit sharing across multiple clients. The management server must also not issue enough vouchers that a user could go over quota.

In order to ensure that sharing can be done without starving a client of vouchers, and reserving some quota in case new clients begin using quota, the management server can give each client a maximum number of vouchers based on the remaining amount of quota. This *limiting* policy satisfies every quota request fully until a user begins to come close to running out of quota by capping the amount granted to $r/(n+1)$, where r is the remaining unauthorized quota and there are n active clients, defined as clients that have requested quota for the user in the last X epochs (meaning they may still hold valid vouchers). A minimum amount of quota is set to minimize client thrashing in requesting the final bytes.

When there are multiple clients competing for a user's quota, and that quota is running low, the policy presented above will work to give each client a portion of the remaining quota. An alternative is to try to give one client enough quota to get its work done, rather than spreading the remainder too finely. This can be done by *revocation:* the management server contacts the other clients who are holding unexpired vouchers, and requests that they return any unused vouchers. Requests waiting for these vouchers are satisfied as soon as the management server receives enough returned vouchers from the clients, and all subsequent quota requests are queued at the server until they can be satisfied, or they time out. Revocation allows active clients to quickly use any remaining quota, even when some other clients have stopped their activity but still hold vouchers. The cost is a possible burst of messages between the management server and clients to return the vouchers.

Using vouchers. Once a client has obtained a voucher, it can use the voucher to consume resources. The client picks which storage server it will use; the problem of selecting the server is outside the scope of this paper, and is addressed by other components in the K2 system [7] .

In the simplest way of using vouchers, the client sends its I/O request to the storage server, along with one or more vouchers that will cover any resource allocations the I/O request might require. The storage server keeps track of how much resource was actually consumed by this request, and may send the client a new voucher for any balance in its reply. It also keeps track of how much each user has allocated in total, plus any recently-spent vouchers. The vouchers are periodically reconciled with the management server in order to handle failure or to catch cheaters, as discussed below.

Consider a simple scenario: a client is trying to write 1 MB of data into an existing file. The client obtains a voucher for (say) 2 MB from the management server, then sends a write request to the storage server along with the voucher. The storage server determines how much resource is consumed. It might consume nothing, if the request only overwrites already-allocated blocks, or it might consume a full 1 MB, or something in between. The storage server will reply with a refund voucher for 2 MB minus the amount actually allocated. A refund is made by appending this amount to the voucher outside of the signature:

$$\{\{epoch, expiry, user, value, serial\}_{auth},\ amount\}$$

When the storage servers and quota servers validate vouchers they will keep track of the vouchers they have

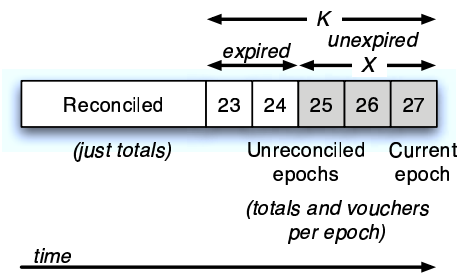

Figure 3. How the storage server maintains consumption information over multiple epochs. $X = 3$ is the number of epochs before vouchers expire; $K = 5$ is the number of epochs in the past when reconciliation happens.

seen by serial number and flag a cheater if the user spends more than 100% of a voucher.

The discussion so far has assumed that a client must use a voucher in its entirety, just as a person cannot divide a high-denomination coin themselves. However, it is possible to allow a client to split a voucher by appending an indication of what fraction of the voucher it is using on any one operation, the same way a storage server splits a voucher when refunding the unused fraction of a voucher. This allows a user to maximize the number of storage servers on which it can allocate resources concurrently, before having to request more vouchers.

Tracking and reconciling usage. The storage server keeps track of how much a user has consumed, and periodically reconciles voucher usage with the management server to catch cheaters and recover from failures. It is important that I/O operations that allocate storage can continue while reconciliation occurs. If they could not, applications would observe an unacceptable interruption in performance. As a result, we avoid synchronously polling every server, and instead use a method that asynchronously reconciles vouchers that have expired and are no longer in use.

We summarize the formal model for tracking and reconciling quota for a single user as follows. The exposition for a single user is clearer than for multiple users, but the rules are the same. The user has a quota Q, and the management server has authorized some allocation A; the system works to keep $A \leq Q$.

The system divides time into *epochs*, as illustrated in Figure 3. Each voucher is associated with the epoch in which it was issued, and the storage server keeps a list of vouchers that it has received for each epoch e, and the total amount of storage allocated using vouchers up to and including each epoch e, S_e. At some point there can be no more activity associated with an epoch, because all vouchers from that epoch will have expired. If vouchers expire after X epochs then the storage server has the final value for S_{c-X-1}, where c is the current epoch. Since this value will not change it can now be sent to the quota server for reconciliation. After reconciliation, the storage server can forget this value since the storage is accounted for in S_e for all $e > c - X$ since S_e includes the total amount of storage allocated using vouchers up to and including e. We allow the storage servers an additional $K - X$ epochs to reconcile S_{c-X-1} with the quota server, so they do not need to be aggressive about updating the quota server and we can handle transient communication problems as long as they do not last more than $K - X$ epochs. Note that reconciliation for an epoch e cannot occur until all the vouchers issued for that epoch have expired, so $K \geq X + 1$.

The quota server tracks V_e, the total value of all vouchers issued in epoch e. From reconciliation with the storage servers the quota server knows how much has been stored on them before epoch $c - K$; however, the quota server can't accurately estimate how much storage has been allocated since then. The quota server must make a conservative estimate of the current allocation A for a user for this period by assuming that all the vouchers issued by the quota server during the past K epochs, $\sum_{0 \leq i < K} V_{c-i}$, have been spent. Therefore the quota server must maintain the amount of vouchers it issued for the past K epochs, to calculate the current allocation for a user as $A = \sum_{\forall d} S_{c-K}(d) + \sum_{0 \leq i < K} V_{c-i}$, where $S_e(d)$ is the storage allocated in epoch e on device d. Using this method to calculate A we can guarantee $A \leq Q$ (except in the case of cheating, which is discussed in detail in the next section), meaning no user will ever exceed their quota using this approach.

Deletions and refunds. When a user deletes data we need to credit them back for the space they have deallocated. The simplest way to do this would be to update all of the values of S being tracked for the storage server the deallocation takes place on. At the next reconciliation A would be updated to accurately reflect the change for the user. The problem with this method is that it takes usually one and at most K epochs for a user to be credited back their space, which can be problematic if a user is near the end of their quota. To solve this we allow storage servers to issue vouchers in order to refund deallocated storage. The user can use this voucher to allocate storage on any of the other storage servers, or equivalently refund it to the quota server, or simply let it expire after X epochs. In order to maintain an accurate measure of A at the quota server the refund voucher is set to the current epoch c, and the refunding storage server d decrements its value for $S_c(d)$.

If the voucher is spent at another storage server d' it will be added to that server's value for $S_c(d')$ (as described in the previous section) balancing the refund when S_c for all involved storage servers is ready to be reconciled with the quota server. There is no scheme for which a user can spend more than it actually deletes since reconciliation for the deletion and the refund (if spent) will be for the same epoch c. If the refunded voucher is not spent and voucher expires at $c+X+1$ the storage server that issued it will have just finalized its $S_c(d)$ value, which it decremented earlier for the value of the voucher, so at reconciliation with the quota server the refund will be reflected in the total allocation A of the user.

Cheating. Since the management server does not track where a voucher is spent, and the storage servers do not update the management server after each allocation, a client can cheat *briefly* by using a voucher more than once on different storage servers. A cheater will *always* be caught within K epochs, where K is the number of epochs storage servers have to reconcile vouchers with the quota management server, as discussed in the previous section.

Similar to digital cash, the protocol addresses this issue in two ways: first, the quota management server catches multiple usage across storage servers during reconciliation, and second, each storage server protects against a client using a voucher twice at that server. The quota management server detects cheating at reconciliation for an epoch by checking if the total allocation for a user using vouchers of the reconciled epoch is greater than the amount of storage allocated using vouchers granted during that epoch. Since each voucher has a unique serial number and the storage server records the vouchers that have been sent to it during the last K epochs, a storage server can reject any I/O request that that reuses a voucher. Then, during reconciliation for a particular epoch, each storage server sends to the quota server a list of all the vouchers it received in that epoch, so that the quota server can cross-check for duplicate usage by serial number. Storage servers can also send used vouchers to the quota server earlier in order to catch cheaters sooner. The cross-checking is not strictly necessary for correctness; however, it allows the quota server to determine which client (or clients) misbehaved, in addition to the user. Cheaters can be caught sooner if the storage servers send their voucher information earlier.

Once cheating has been detected, the system must decide how to respond. Many responses are possible, including notifying a human, disallowing further allocation, or automatically compressing or removing redundancy.

Though cheating is possible in the system for a brief amount of time, the amount of damage cheaters can do is bounded. Since storage servers immediately detect voucher reuse the amount is bounded by $(Q-A)n$ where $Q-A$ is the amount of quota the cheating user has left according to the quota server, and n is the number of storage servers it has permission to store data on. The amount of storage it could consume is also bounded by $K\mu$ where K is the number of epochs storage servers have to reconcile vouchers with the quota management servers, and μ is the maximum throughput available to the cheating user (this would depend on the number of clients it could corrupt to cheat). In practice the limit would be the minimum between these two bounds. If the system is sensitive to cheating K could be adjusted to be very short in order to reduce the amount of storage a client can over allocate with a fairly small effect on performance. This trade-off is explored quantitatively in the experimental results section. In order to slow a possible denial-of-service attack the length of an epoch could be a function of the amount of storage space left, therefore decreasing the amount of time it takes to detect a cheater; however, the investigation of this idea is the topic of future work.

The time in which it takes to detect cheaters is a performance trade-off. If we shorten the time in which we catch cheaters the load on the quota management server increases since voucher requests and reconciliation will be more frequent. Our experiments in the next section illustrate this trade-off and show that by allowing the risk of temporary cheating for a very brief period we get a significant performance gain over a centralized method.

For our system we consider cheating to be a rare event. Incentives not to cheat are very clear since cheating users and clients will *always* be caught quickly, within at least K epochs.

Failures. When one of the system components fails, the quota management system will always maintain its integrity. When storage servers fail, all information about what was allocated on them disappears. The management server will reliably learn of the failures and exclude those storage servers from its computation of the total authorized allocation A, which will make additional quota available for clients to allocate (presumably for recovering the data by rebuilding redundancy or restoring from backup). Once all epochs are reconciled up to and including the one when the failure occurred, the estimate will have decreased by exactly the amount that had been used on the failed servers.

When a client fails, it may be holding unused vouchers. The management server will learn what vouchers were actually used as it reconciles with storage servers. Any unredeemed vouchers the client held at the time it failed will not be counted against the user after they expire.

When a quota server fails, the clients will not be able to obtain more vouchers for users. During the failure clients can still use the vouchers they already have and storage devices will continue to work as normal. In practice the quota

server will be made highly reliable, preferably using a cluster of servers. However, in the highly unlikely case that the quota server has to be entirely rebuilt using only the quota limits of the users, the quota server can run for the first K epochs (where K is the number of epochs storage servers have to reconcile vouchers with the quota management server) issuing only a very conservative number of vouchers based on each reconciliation, and after K epochs the server will be completely up-to-date on the system-wide consumption. In order to speed the quota server recovery epochs can be issued at a higher rate until K epochs have passed.

4. Experimental results

The voucher approach to maintaining quota is designed to promote good scalability by minimizing the amount of work that the shared management service should have to do on behalf of clients, and so we have evaluated the performance as various system scale factors increase. In addition, there are several design choices to be made, such as the policy for how a client determines how large a voucher to request. We have evaluated the performance, scalability and design choices for the proposed system; the results are presented in this section.

4.1. Simulation

We implemented a discrete event simulation to evaluate the voucher approach. We chose to use simulation for two reasons: first, we wanted to evaluate different options quickly without the effort of implementing them in our full storage system; and second, we wanted to evaluate performance at scale points far larger than we could achieve with our actual testbed cluster. All of the I/O operations a system would normally perform, such as reads and metadata-only requests, were run in the system in addition to writes and voucher requests, in order to better model the overall impact that quota enforcement would have on a real system.

The simulator used simple models of the client, network, and storage server since the focus of the experiments are on the scalability of the management service for quota enforcement. The client cache was represented by a uniform probability of a cache hit or miss on I/O requests. The network was modeled as a constant transfer time, without contention. The storage server included a cache, modeled as a uniform hit/miss probability, and a simple disk, with fixed seek time and transfer rate, and queuing at the disk.

The simulator modeled the approaches for tracking quota described in the previous section. For client voucher request policies we compared the *fixed* and *adaptive* methods, and for the quota management server we compared the *limiting* and *revocation* policies for issuing vouchers. The

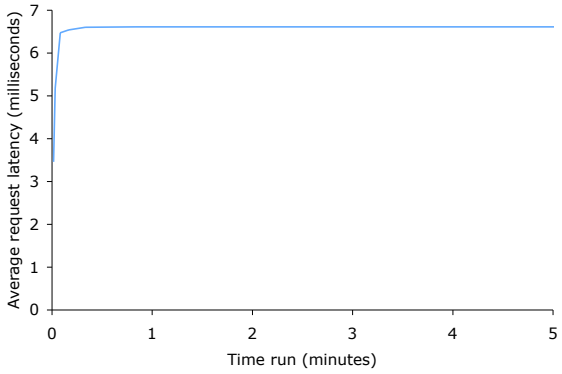

Figure 4. Average request latency over simulated time. The system stabilizes quickly in the first minute.

simulation also included a centralized management service for comparison, where all quota allocation decisions are made at the management server cluster, and quota is assigned for each I/O operation. The management server tracks quota in a centralized database, and the simulation models the overhead for processing transactions. Clients stripe data sequentially across storage servers in 4 MB stripes.

For all tests, vouchers expired after two epochs and epochs were 10 minutes long. Storage servers reconciled allocation information with the management servers after two epochs. Each test was run 10 times, and the 95% confidence intervals are plotted in the graphs. Every test was run for an hour in simulated time to allow the system to stabilize, and measurements were recorded over the last 10 minutes of the run. This allowed vouchers to expire and required reconciliation between the quota management servers and the storage servers during this period. Figure 4 demonstrates that the system reaches equilibrium in less than a minute in simulated time. In order to verify that our simulation correctly modeled the management server bottleneck we verified our results against an M/D/1 queuing system. Clients sent requests exponentially (M) and the quota management server responded deterministically (D). For an M/D/1 queuing system the average time a request waits in the queue at the server is:

$$\frac{\lambda}{\mu - \lambda} \cdot \frac{1}{2\mu},$$

where μ is the mean service rate and λ is mean interarrival rate. Figure 5 shows this theoretical time and simulated time under increasing load. The number of clients and storage servers were scaled with the I/O load in order to keep their queues empty, while the number of management servers was held constant.

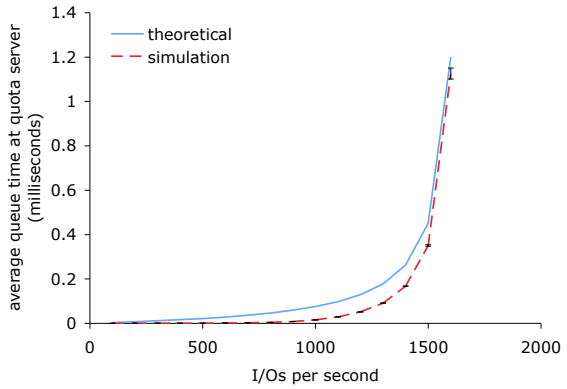

Figure 5. Average queue time at the quota management server for the simulated system and for the system modeled as an M/D/1 queuing system.

4.2. Workloads

We evaluated the system using two different workloads. The *user* workload provides a fairly steady flow of I/O requests with low sharing, while the *scientific* workload is highly bursty and involves significant write sharing. We implemented both workloads using synthetic workload generators based on published measurements.

For our user workload, each user generated 10 IO requests per second using the system call statistics taken from a set of institutional machines with mounted home directories for a few hundred (student) users [18]. This workload was heavily dominated by *stat*s and *read*s, with only 2% of the total load being writes. The average request size used was 4 KB. Each user in this workload spread its work over 10% of the available clients. This workload can be scaled by increasing the number of users.

Our scientific workload was modeled after a physics application [22] where each node has similar responsibilities and alternates between a computation phase and large write-intensive phases, in which it writes out a checkpoint of intermediate results. This workload modeled a single user using all clients to run the application. This workload scales by increasing the I/O rate for the user.

4.3. System overhead

The first evaluation answers the the main question: which approach gives the lowest overall latency for user I/O requests while scaling to handle heavy loads? Figures 6 and 7 show the main results.

These experiments vary the I/O load on the system, and measure the resulting I/O request latency. The load varies from a low baseline to the point where the system saturated. To ensure that the bottleneck causing saturation comes from the quota mechanism, the number of clients and storage servers are scaled with the I/O load, while the number of management servers is held constant.

Figure 6 shows the overall results for the user workload. Performance is dominated by ordinary file metadata traffic, including retrieving file layout for both reads and writes and checking authorization, and by communicating with the storage server. Since most files in this workload are small, most quota-related operations can be combined with these metadata operations. As a result, both centralized and voucher quota management schemes impose less than 0.2% overhead compared to no quota management, while the system is below saturation. However it is important to note that when using the centralized method the system saturates at around 1500 users (15000 I/Os per second), while the system using no quota enforcement and the voucher-based method saturate around 1600 users (16000 I/Os per second).

Figure 7 shows the results for the scientific workload. This workload is bursty and involves large files, which places more importance on the performance of quota management. At loads below saturation, the voucher approach gives performance essentially identical to no quota enforcement, while centralized quota management imposes about 8% overhead. More important, using the voucher approach saturates at the same load as does no quota checking, while the centralized approach saturates at about half the load. One of the contributing factors is that the system using centralized quota tracking requires more quota-related messaging than the system using vouchers because it must make quota requests for every I/O. In addition, the centralized management server takes longer to satisfy quota requests since it must commit transactions to a centralized database. Using the voucher approach very few extra messages must be sent to request quota since clients cache vouchers, and the management server only needs to update a very small set of data for tracking quota.

4.4. Client adaptivity

For the voucher-based method to be effective clients must request enough vouchers to satisfy some number of future requests without requesting so many vouchers that it hinders sharing of quota between clients. We compared the adaptive and fixed policies discussed earlier for how clients decide on the number of vouchers to request. Recall that in the adaptive request policy the client tries to predict the user's allocation rate over some window of time using a moving average in order to make quota requests only periodically. For the fixed request policy the client always requests the same fixed amount of vouchers. We also com-

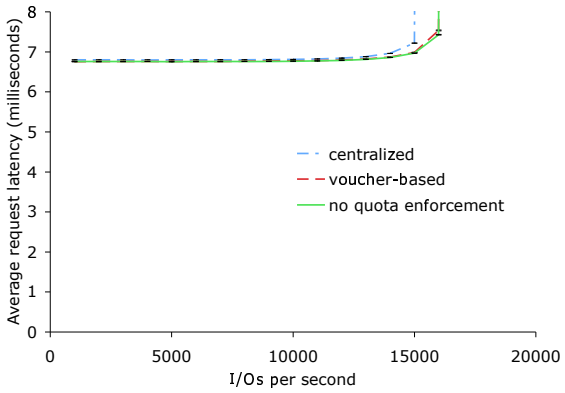

Figure 6. Average I/O request latency for the user workload, under an increasing user load. The system cannot support more than 1600 users issuing 10 I/Os per second using 10 quota servers with or without quota enforcement, and for the centralized method it is limited to 1500 users.

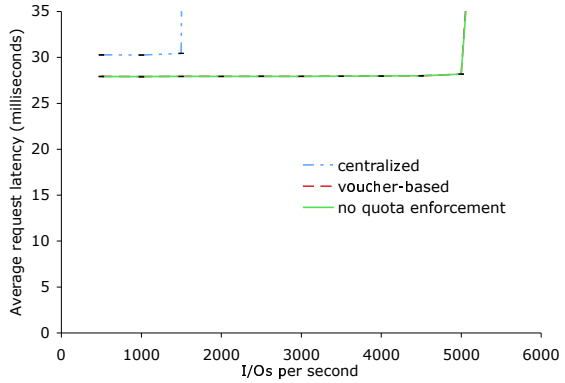

Figure 7. Average I/O request latency for the scientific workload, under increasing I/Os per second. The system cannot support more than 5000 I/Os per second using 10 quota servers with or without quota enforcement, and using the centralized method it cannot support more than 1500 I/Os per second.

pared these policies to an *expected* method in which the client requested the average amount a user consumed over a period of time based on the known rate and distribution the user generated requests from, so it can be thought of as what the adaptive request policy attempts to predict for this set of experiments.

Figure 8 shows the number of withdrawal messages the client must send to the management server that could not

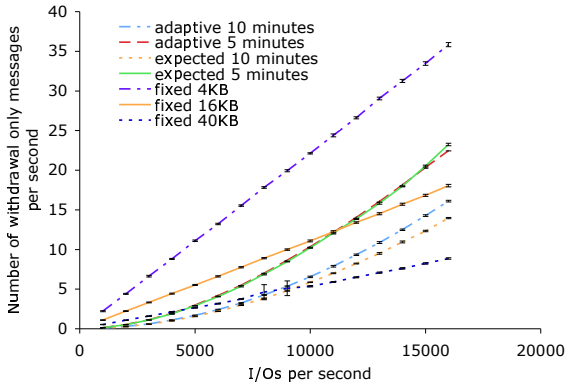

Figure 8. Total number of messages clients make to the management server to request vouchers under increasing load for the fixed and adaptive client request policies.

be grouped with a metadata request. As in the previous experiments, to ensure that the bottleneck causing saturation came from the quota mechanism, the number of clients and storage servers were scaled with the I/O load, while the number of management servers were held constant. The adaptive policy performs close to the expected measurement that is set at the actual average request size for the time window used, showing that the adaptive request policy predicts the expected usage well. For a real system the adaptive policy is likely preferable since the exact consumption rate of storage for each user per client is usually unknown beforehand, and will probably vary over the lifetime of the system.

For the fixed request policy the performance overhead of quota enforcement is heavily dependent on the fixed amount chosen and the write load on the system, while the adaptive policy depends on the size of the prediction window and the accuracy of the prediction. Figure 9 shows that the adaptive policy is fairly insensitive to its window size parameter for the workloads we tested. There are diminishing returns for larger window sizes as vouchers are more likely to expire before a client has the need to spend them when the prediction is not exact. The above results were all for the user workload, but results for the scientific workload exhibited the same behavior as the user workload in these cases.

4.5. Running out of quota

Quota systems behave differently when a user has plenty of quota left and when they are running low. In the first case, the system just has to keep track of usage, while in the second, it has to actually enforce the quota limit.

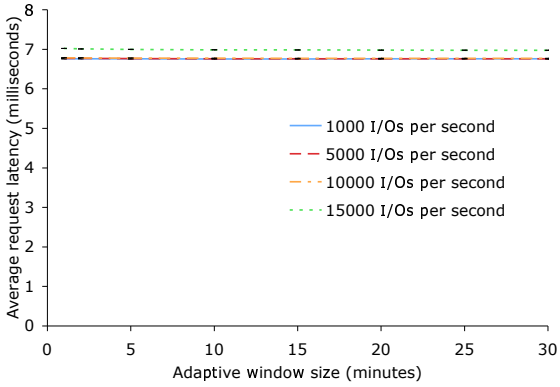

Figure 9. Average I/O request latency for the user workload under the adaptive request method where the window of prediction time is varied.

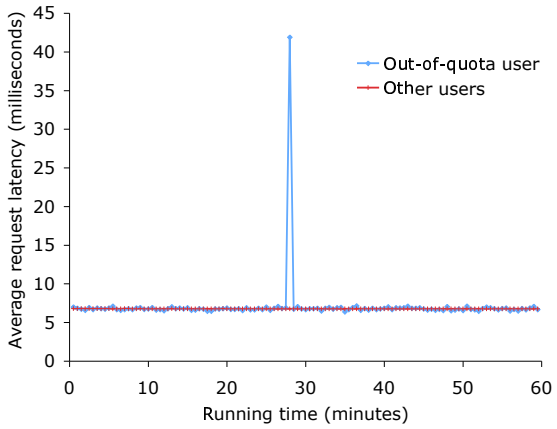

Figure 10. Average I/O request latency sampled every 30s for the user workload using the revocation server method, where one user runs out of quota after 30 minutes. The average request latency is plotted separately for this user while all other users average request latency is averaged together.

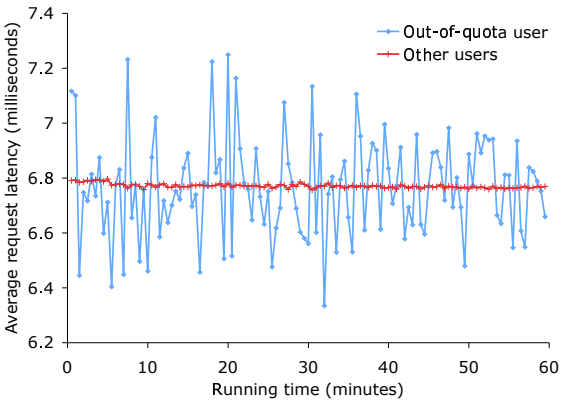

Figure 11. Average I/O request latency sampled every 30s for the user workload using the limiting server method, where one user runs out of quota after 30 minutes.The average request latency is plotted separately for this user while all other users average request latency is averaged together.

In the voucher approach, the management server has to determine how much of a client's voucher request to grant in order to maintain the invariant that authorized use is bound by the user's quota. The management server also should ensure that one client does not starve another. Both these goals are more difficult when there is little available quota.

To compare the two policies we have proposed for the management server, we simulated a scenario where one user consumes all their quota. This scenario used the same workload as in previous experiments, with 10 management servers, 1000 users, and 100 clients (using adaptive requests), and 100 storage servers. The quota for one user is set so that it will run out after about 30 minutes. After that user runs out of quota, it stops issuing writes that could consume space and continues with all other operations. Recall that each user's activity is spread over 10 clients. We report the average I/O request latency taken at 30-second intervals, with the statistics for the user that runs out of quota separated from those for all other users.

Figure 10 shows the response when using the revocation policy, where unused vouchers are revoked from clients when a user needs more vouchers to spend at other clients. The user that runs out of quota suffers a large increase in latency during its final write requests. This occurs because the last writes wait for vouchers to be revoked from other clients, or time out. This is the worst possible case for the revocation server, since all clients are trying to consume resources steadily and so must all contend for the last bits of quota.

Figure 11 shows the response obtained using the limiting management server policy. This policy does not show any noticeable performance difference as the user runs out of quota. All the user's clients run out of quota together gradually as their requests are tapered down by the management server.

Note that in both cases traffic from other users is not much affected.

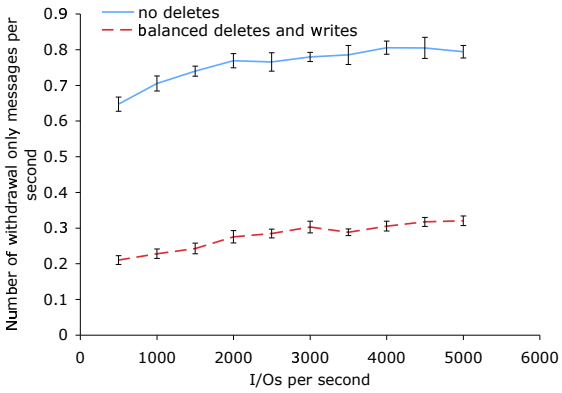

Figure 12. Total number of messages clients make to the management server to request vouchers.

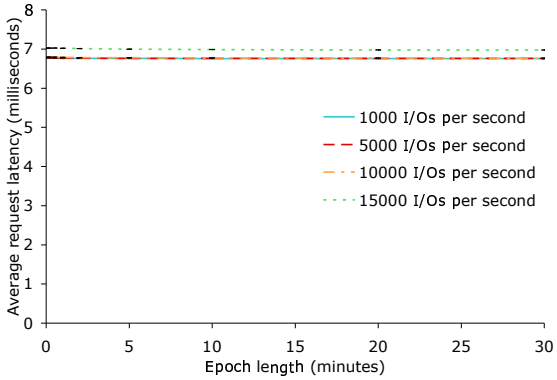

Figure 13. Average I/O request latency for the user workload using different epoch lengths.

4.6. Balanced deletes and writes

We expected the voucher approach to cause a notable reduction in quota-related traffic between clients and management server when a client's consumption is approximately balanced by the resources it frees, because the client can use the vouchers it gets from storage servers when deleting one file to allocate for another file. This may happen in the case of a scientific workload that writes out checkpoints, and after writing a checkpoint it deletes the previous checkpoint to conserve storage. Figure 12 shows that even when using the adaptive request policy on the clients, the requests to the quota server are drastically reduced when consumption and deletion are balanced for the scientific workload. The number of requests would be even lower if the the load was not distributed across multiple clients since deletes and writes do not always happen evenly on each client.

4.7. Granularity of quota enforcement

Our quota enforcement method requires several configuration parameters to determine the granularity of quota tracking, the length of an epoch, the number of epochs a voucher is valid for, and the number of epochs the storage servers have to update the quota server. Figure 13 shows that performance improves very slightly for longer epochs. This is because vouchers can be requested in larger batches since they can be cached at the clients longer, and reconciliation doesn't happen as frequently. Both of these factors should reduce load on the quota server. Similar tests were run for varying the number of epochs a voucher is valid for and the number of epochs the storage servers have to update the quota server. These showed no significant effect on performance and have been omitted for space.

4.8. User distribution

The more clients used by each user, the less effective caching vouchers may become as the load is more distributed. Figure 14 demonstrates the effects of users distributing a fixed load across an increasing number of clients for the user workload with 1,000 users and 100 clients. We see less than a 0.5% increase in average latency as we increase the number of clients each user is using from 0 to 100. By fixing the load of the user the voucher method behaves more similar to the centralized method as the number of clients increase since clients are less likely to make enough requests per epoch for caching to be beneficial. If the users were to increase their load on any of the clients then caching at that client should become more effective as long as there is enough quota to support the demand. Another effect of sharing is that when a user is low on quota the more clients it uses, the more competition there is for the last of the vouchers, leading to poorer performance as well.

5. Related work

The voucher approach to quota management is obviously inspired by the extensive literature on digital cash systems [16]. The voucher approach is considerably simpler than those systems, however, because vouchers do not provide anonymity. Vouchers only require that a storage server can verify that they have not been forged or corrupted.

Vouchers are also similar to capabilities. In particular, they are similar to the kinds of capabilities used in Amoeba [13], which implemented capabilities as cryptographically-protected sequences of bytes. This model was taken up for the NASD [6] and T10 OSD [10] object storage model, which added the notion of expiration.

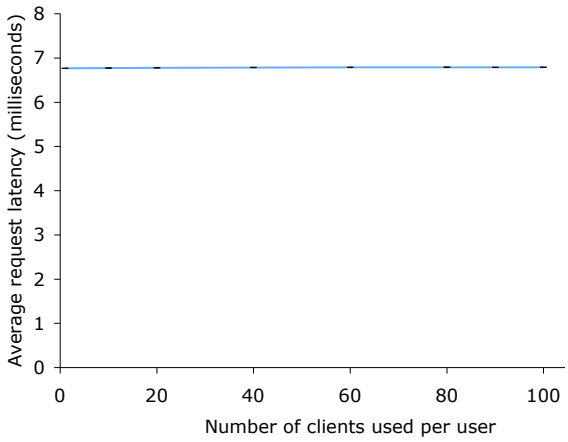

Figure 14. Average I/O request latency for the user workload where the load is distributed among an increasing number of clients.

One application of vouchers is the use of distributed quota to control spam [21]; that system uses a *stamp* mechanism very similar to our vouchers, but lacks the reconciliation necessary for a robust storage system.

Many file or storage systems implement quotas. They can be divided into three classes: those that perform file management in band with I/O request processing; those that perform file management out of band but have file management decide which resources are to be allocated; and those where file management and resource consumption are completely separated.

Most network-oriented file systems perform file management in band, including NFS [20] and CIFS [17]. AFS [9] would appear to have a more complex quota management scheme, but in reality, quota management is done on a volume group granularity, which allows quotas to be managed with the storage allocation.

Several more recent file systems allow clients to perform I/O directly to storage devices, while dealing with file management out of band. GPFS [19] and SanFS [12] are two examples of file systems that perform block allocation in the file management path. GPFS, being optimized for very large clusters, uses a quota management system that resembles our vouchers: clients request quota from a server, and independently update their quota information locally. The GPFS system lacks epochs and reconciliation, and can not correctly handle either failures or cheating of clients, without the manual use of repair tools.

Object-based file systems derived from the NASD [6] model, including the Panasas [14] file system, determine how much resource on which OSD a client should be able to use. Although the clients can make requests directly to the storage devices, quota enforcement is still done at the storage manager. Capabilities are for specific devices and are created with offset and size limitations to restrict the storage clients from exceeding their quotas.

Peer-to-Peer storage systems have much more difficult problem because they are not able to manage quota and allocation together. PAST [4], for example, uses smart cards to manage storage quotas. These cards are trusted by the peer storage devices to reliably keep track of the allocations and quotas of their owners. As storage is used the cards will increment the allocated storage. The cards are also able to process reclaim receipts that will decrement the allocated storage. The use of smart cards in PAST binds the storage user to a single client machine. Such an architecture is not well suited for storage users that use multiple machines concurrently.

Samsara [3] and SHARP [5] provide a way to ensure that users of a peer-to-peer system contribute as much resources as they use. Both systems use cryptographic signature chains to enable peers that contribute storage to use storage on other peers. In peer-to-peer systems this is roughly analogous to quota enforcement; however, in systems with trusted servers quota enforcement can be done in a simpler and more efficient manner, and catch cheaters in a guaranteed fixed time rather than a probabilistic time.

Another peer-to-peer system [15] focuses on fair sharing by performing random audits of resource usage. Each peer lists the storage available, the storage it is using, and the storage used by other peers. The information is structured in such a way that a peer can check claims of storage usage for random peers that it is using storage from or providing storage to. If lying is detected, appropriate action can be used to eject the peer from the system. For our quota enforcement technique the risk of being caught cheating is not dependent on random checking, a cheater will always be caught within a bounded period of time. The incentive not to cheat is clear as cheaters will always be caught quickly at which time some corrective action will be taken.

6. Conclusions

We have presented a system for scalable tracking and enforcing quota in distributed storage systems. This system minimizes the load on a central management server by issuing clients *vouchers* that the clients can use to consume storage on whichever storage server is appropriate. The vouchers can be used any time after issued until they expire, thus trading temporal granularity for performance. The storage servers periodically reconcile actual usage with the management server in order to verify that all clients have behaved properly. Misbehaving clients are always caught within a short period of time, and the amount of damage they can do is bounded. In the future, we can envision generalizing this quota mechanisms to control usage

of other storage QoS metrics, such as bandwidth or server utilization.

The simulation we have conducted indicates that this approach will work well. For workloads characterized by low I/O rates per user and small files, in which ordinary metadata operations like checking file permissions and getting file layout dominate, the voucher approach gives performance essentially as good as not checking quota at all. In a workload with larger files, there are more I/O requests for each metadata operation and so the difference between a centralized quota system and the voucher approach is more pronounced. For one class of workload, where there a client frees resource at about the same rate that it consumes, the client can use the "refund" vouchers it gets for its allocations and thus needs to get new vouchers from the management server only rarely.

There are several design options, and this study explored a few of them. The storage client has a policy for when it will request vouchers, and how much it will request. We found that an adaptive policy, which uses a moving average of recent consumption to predict near future consumption, works well and is insensitive to a broad range of moving average windows. In the future we intend to look at other ways to predict near future consumption, especially ways that can react quickly when scientific applications finish an I/O phase.

The management server also has a policy for how much of clients' requests to grant. We found that the revocation policy induces a large burst of traffic when a user runs out of quota, while the limiting policy produced a smoother result. Finding policies that predict changes to the degree of sharing are future work.

References

[1] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, (Vol. 4, No. 4):405–436, 1991.

[2] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. White paper, November 2002.

[3] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[4] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems (HotOS) VIII*, May 2001.

[5] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, October 2003.

[6] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks. Technical Report CMU-CS-97-118, CMU SCS, 1997.

[7] R. A. Golding and T. M. Wong. Walking toward moving goalposts: agile management for evolving systems. In *First Workshop on Hot Topics in Autonomic Computing (HOTAC-1)*, Jun 2006.

[8] G. Hardin. The tragedy of the commons. *Science*, (162):1243–1248, 1968.

[9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, volume 6, pages 51–81, February 1988.

[10] INCITS Technical Committee. Information technology - SCSI object-based storage device commands - 2 (OSD-2). http://www.t10.org/ftp/t10/drafts/osd2/osd2r00.pdf.

[11] J. Kohl and C. Neuman. The Kerberos network authentication service (V5). RFC 1510, September 1993.

[12] J. Menon, D. Pease, B. Rees, L. Duyanovich, and B. Hillsberg. IBM StorageTank—a heterogeneous scalable SAN file system. *IBM Systems Journal*, 4(2):250, 2003.

[13] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[14] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—Delivering scalable high bandwidth storage. In *ACM/IEEE Conference on Supercomputing*, Nov. 2004.

[15] T.-W. Ngan, D. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *International Peer to Peer Symposium*, February 2003.

[16] T. Okamoto and K. Ohta. Universal electronic cash. In *11th Annual International Conference on Advances in Cryptology (CRYPTO)*, pages 324–337. Springer-Verlag, Aug 1991.

[17] The Open Group. *Protocols for X/Open PC Internetworking: SMB, Version 2*, September 1992.

[18] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *USENIX Annual Conference*, June 2000.

[19] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Usenix File and Storage Technologies Conference (FAST)*, pages 231–44, Jan. 2002.

[20] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, December 2000.

[21] M. Walfish, J. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 281–296, May 2006.

[22] F. Wang, B. Hong, S. Brandt, E. Miller, and D. Long. File system workload analysis for large scale scientific computing applications. In *21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2004.

[23] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Conference on Operating Systems Design and Implementation (OSDI)*, November 2006.