Quota enforcement for high-performance distributed storage systems

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

Abstract

Storage systems manage quota to ensure that each user gets the storage they need, and that no one user can—even by accident—use up all available storage. This is difficult for large, distributed systems, especially those used for high-performance computing applications, because resource allocation occurs on many nodes concurrently. We present a scheme where quota is enforced asynchronously by intelligent storage servers: storage clients contact a shared management service to get *vouchers*, a capability-like certificate that the clients can redeem at participating storage servers to allocate storage space. This approach produces low load on the shared management service, promotes good scaling, and allows the client to make decisions about which storage server(s) to use without communicating with the management service for further approval. Storage servers and the management service periodically reconcile 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 shows that this approach gives performance nearly as good as not enforcing quota at all, and that the load on the shared management server is remarkably low.

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. In a file system, for example, users must not use more than their storage quota. Many scientific applications involve tens of thousands of nodes all cooperating on a problem, all writing to shared files and consuming from the same pool of quota. The file system is typically built as a small cluster of metadata servers and a larger number of storage servers or disk arrays. We concentrate here on systems that use storage servers that provide intelligence similar to object storage, and so can track local storage allocation and enforce access control.

Existing quota systems trade off scalability and accuracy. A centralized quota tracking server can be byte-accurate as long as each client informs the server on each resource allocation, which inhibits scalable performance. Other systems use a centralized server but relax accuracy, either by tracking quota only in large granules or by using time-limited escrow mechanisms (which set aside a certain amount of resource for a client for a limited duration), both of which reduce the frequency with which a client must interact with the quota tracking server.

The existing quota systems also provide for a single quota-related policy for all clients. In a distributed file system like SanFS [10], for example, the centralized metadata server decides which logical disks in a storage pool a client should be allocated when it needs storage. This requires that the policy not only have an accurate record of how much quota each user has consumed, but also an accurate map of how much resource is available on every server on which that resource could be allocated.

We propose an alternative approach to tracking and enforcing resource limits. This approach borrows from microcash mechanisms: there is a centralized server that acts as a *bank* that issues *vouchers* to clients, which the client can spend to allocate resources on whatever 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.

This approach provides a different tradeoff than other quota servers. It provides excellent scalability—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 centralized tracking service well below that of other mechanisms. 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

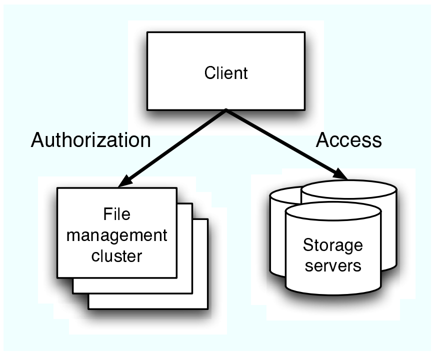

Figure 1: Basic distributed storage system architecture.

need to be concerned with where that consumption has occurred, 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* [11] 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 [8, 5].

We are investigating quota management as part of the K2 distributed storage system. 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," [6] 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 aggregate 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 minimise 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 [9]. Our design assumes that clients can crash-fail. While some clients can also be malicious, we do not focus on them; for example, we do not provide data access that can survive Byzantine behavior [1]. However, for resource quota management we do bound the effect that any client can have on the system.

3 Protocol operation

In this section 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

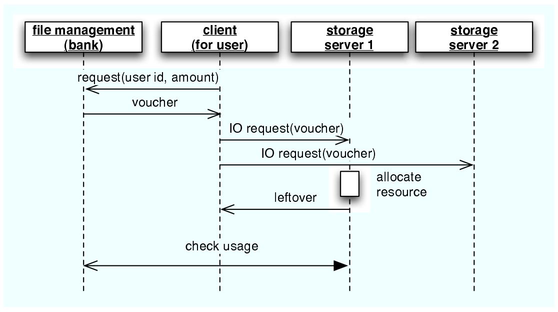

Figure 2: Sequence of operations. A client begins by obtaining a voucher for a user from the quota server, then spending that voucher during IO requests to different storage nodes. Later, the quota server and storage nodes check that the client did not overuse a voucher.

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.

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:

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

similar to capabilities used to authorize actions in Amoeba [11] and the T10 OSD [8]. *User* and *amount* are obvious. 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. However, it does not prevent one client from eavesdropping on another, and so vouchers must be transmitted only over private channels. Issues like avoiding replay attacks do not require special mechanisms in vouchers if they are used in conjunction with authorization capabilities that provide defense against replay.

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.

Getting 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.

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 go over the user's quota.

While there are many possible policies for deciding when and how much to ask for, we focus on a client requesting quota from the management server on a regular schedule. 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 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 have to 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, 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 determines how much to grant to a client based on its global information, including the total amount 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.

One reasonable heuristic is for the management server to give each client an amount proportional to its consumption rate, while reserving some quota in case new clients begin using quota. The client policy discussed above makes requests approximately proportional to consumption rate, and so the management server can give each client the same fraction f of their requested amount. When there is plenty of quota, $f = 1$. As the number of clients increases or the amount of remaining quota decreases, each client gets a fraction $f = r/((n+1)\bar{r})$ of their requests, where r is this client's consumption rate, n is the number of clients consuming from that quota, and $\bar{r}$ is the average consumption rate over all active clients.

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.

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 the request, and may send the client a new voucher for any balance in its reply. The storage server keeps track of how much each user has consumed, 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.

Vouchers can also be used in a somewhat different way to solve a long-standing problem in object storage systems—ensuring that an operation will succeed when multiple clients could be consuming resources in the storage server. A client can use a voucher to *reserve* resources at the storage server to ensure that its later operations will have the resources to complete. This is particularly important for object stores because it is hard for a client to predict how much resource any one operation will consume.

Tracking and reconciling usage. The storage server keeps track of how much a user has consumed, periodically reconciles voucher usage with the management server to catch cheaters and recover from failures.

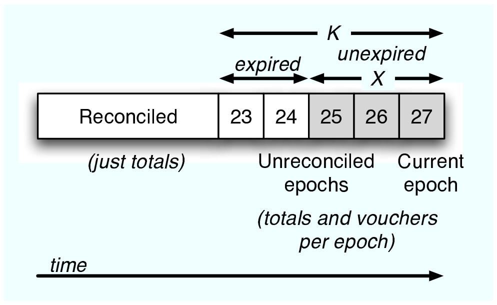

Figure 3: How the storage server maintains consumption information over multiple epochs. X is the number of epochs before vouchers expire; K is the number of epochs in the past when reconciliation happens.

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. It also tracks how much resource was consumed against those vouchers in the epoch. At some point there can be no more activity associated with an epoch—because all vouchers from that epoch will have expired—and the storage server can reconcile the list of vouchers used for that epoch with the management server. After reconciliation, the storage server can get rid of the list of vouchers and merge that epoch's consumption information into the record of overall reconciled consumption.

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 allocation A is the amount of storage used on all storage servers plus the amount of any unredeemed vouchers: $A = \sum_{\forall d} S_d + U$, where S_d is the amount used on storage server d and U is the amount of unredeemed vouchers.

For the management server to know A accurately using this definition, it would need to be involved synchronously on every resource allocation, which defeats the intent of the voucher approach. Instead, the management server uses a conservative estimate of A: instead of the actual current usage $\sum S_d$, it uses the usage determined at the last reconciliation, and instead of the actual unredeemed vouchers, it uses all vouchers issued since the last reconciliation. Formally, the management server knows the amount of resource the user had consumed as of the last reconciliation, which covered all epochs up to and including epoch e: $\sum_{\forall d} S_d^e$, where e denotes which epoch the reconciliation was for. V^e is the amount issued in vouchers in epoch e. The estimate is then $A \approx \sum_{\forall d} S_d^{c-K} + \sum_{0 \leq i < K} V^{c-i}$, where c is the current epoch, and K is number of epochs in the past when reconciliation occurs. (Note that reconciliation for an epoch e cannot occur until all the vouchers issues for that epoch have expired, so $K > X$, where X is the number of epochs before a voucher expires.)

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* by using a voucher more than once.

The protocol addresses this issue in two ways: first, each storage server protects against a client using a voucher twice at that server; and second, the management server catches multiple usage across storage servers during reconciliation. Since each voucher has a unique serial number and the storage server records the vouchers have been sent to it during the last K epochs, a 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 management server a list of all the vouchers it received in that epoch, so that the manager can cross-check for duplicate usage by serial number. Storage servers can also send used vouchers to the management server earlier in order to catch cheaters sooner.

The cross-checking is not strictly necessary for correctness. If a client uses more than a user's quota by using a voucher from epoch e at multiple storage servers, the management server will detect that the user went over quota in epoch e when that epoch is reconciled. Cross-checking for voucher duplication, however, allows the management server to determine which client (or clients) misbehaved. 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. Several responses are possible—notifying a human, disallowing further allocation, or automatically compressing or removing redundancy.

Failures. When one of the system components fails, the quota management system must maintain its integrity. For this paper, we treat management server failure as catastrophic, and so do not model recovery from it. In practice, this can be addressed by clustering.

When a client fails, it may be holding unused vouchers. The management server learns what vouchers were actually used as it reconciles with storage servers.

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.

Optimizations. 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. This somewhat complicates the reconciliation mechanism but does not change the basic protocol design.

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 fraction 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. 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.

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. In this section we discuss how we evaluated these comparisons, and the results.

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. 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 several different approaches for tracking quota. It 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 the voucher approach, the simulator modeled two different policies for how clients request vouchers. The *fixed request* policy 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. We test fixed amounts of 5 and 10 times the average request size. The *adaptive request* policy requests the amount of quota they predict a user will need for some window of time based on a moving average of the user's recent throughput.

The simulator also modeled two different policies for how the management server should respond to voucher requests. The *revocation* policy always grants every voucher request until a user runs out of quota. If a request cannot be satisfied because the user does not have enough quota left, the management server sends out revocation messages to all of the clients asking that vouchers be returned for the user that has run out of quota. The request is 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. The *limiting* policy satisfies every quota request fully until a user begins to come close to running out of quota. The amount granted is capped 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 the block size to minimize client thrashing in requesting the final bytes.

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 [16]. 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 [19] 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.

For all tests, vouchers expired after two epochs and epochs were 10 minutes long. Storage servers reconciled allocation information with the management servers within two epochs. 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.

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 4 and 5 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 4 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

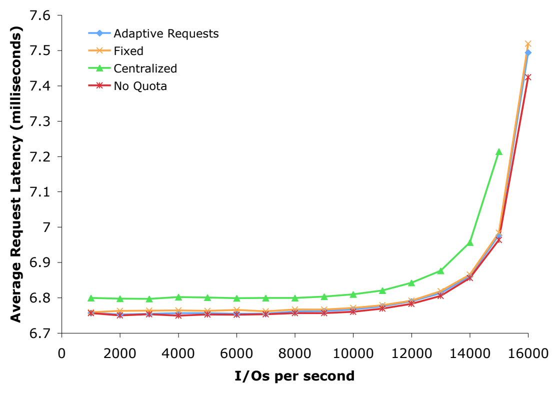

Figure 4: Average I/O request latency for the user workload, under an increasing user load. The system cannot support more than 1600 users using 10 quota servers with or without quota enforcement, and for the centralized method it is limited to 1500 users.

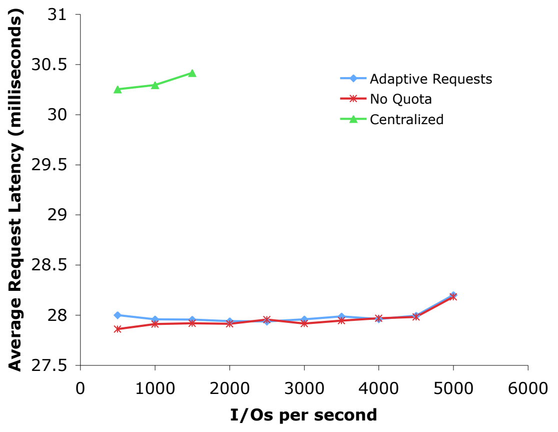

Figure 5: Average I/O request latency for the scientific workload, under increasing I/Os per second. The system cannot support more than 5500 I/Os per second with or without quota enforcement, and using the centralized method it cannot support more than 1500 I/Os per second.

with these metadata operations. As a result, both centralized and voucher quota management schemes impose only a small overhead compared to no quota management.

Figure 5 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 7% 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 contributing factors is that the system using centralized quota tracking require 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

Clients must determine how large a voucher to request each time they request more quota from the management server, using one of two policies: requesting a fixed amount, or basing the request on recent usage. Clients need to request large enough vouchers to cover multiple I/O requests in order to effectively piggyback quota traffic on regular metadata requests. Figure 6 shows the extra message overhead for quota enforcement using the fixed and adaptive policies, defined as the number of voucher requests that could not be combined with other necessary metadata operations.

The fixed request policy is sensitive to its amount parameter, which can cause it to allocate too little for large files. Using a larger fixed amount decreases the chances that a client will need to request more quota before it is finished with a file, but the tradeoff is that at the limit large fixed requests inhibit sharing quota among multiple clients. This problem is exacerbated when file size or I/O request rates vary over time.

The adaptive request policy determines how much quota it will request based on a moving average over a recent window of requests; the window size is the parameter for this policy. This policy should cause the client to go to the management server for quota at a constant rate. However, the lag in the moving average can cause extra requests when the workload changes. Figure 7 shows that the adaptive policy is insensitive to its window size parameter for the workloads we tested. The figure shows the result for the user workload; results for the bursty scientific workload were similar.

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.

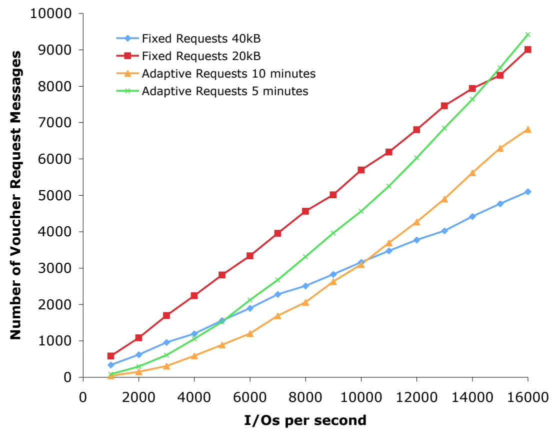

Figure 6: Total number of messages clients make to the management server to request vouchers.

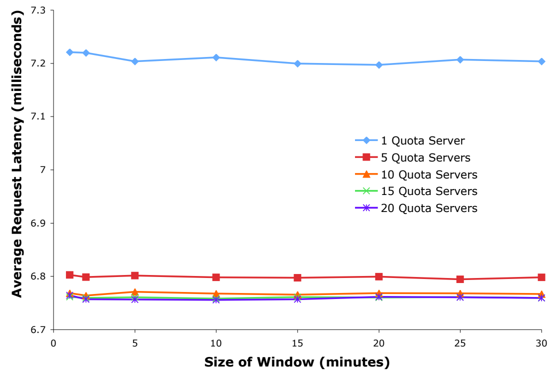

Figure 7: Average I/O request latency for the user workload under the adaptive request method where the window of prediction time is varied.

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 special user separated from all other users.

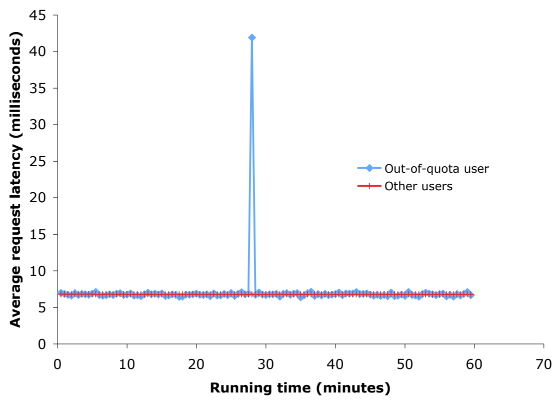

Figure 8: Average I/O request latency over time for the user workload using the revocation server method, where one user runs out of quota.

Figure 8 shows the response when using the revocation policy. 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 9 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.

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. Figure 10 shows that, as expected, the requests to the quota server are near zero when consumption and deletion are balanced. For comparison, a second curve shows the num-

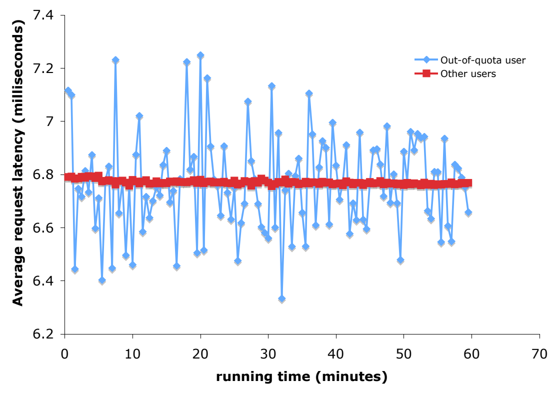

Figure 9: Average I/O request latency over time for the user workload using the limiting server method, where one user runs out of quota.

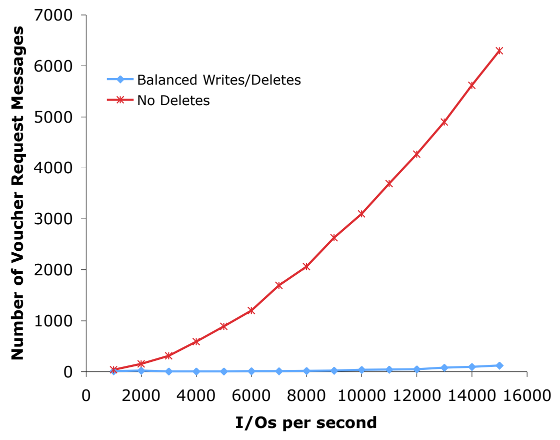

Figure 10: Total number of messages clients make to the management server to request vouchers.

ber of voucher requests for clients that are only consuming storage.

5 Related work

The voucher approach to quota management is obviously inspired by the extensive literature on digital cash systems [14]. 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 [11], which implemented capabilities as cryptographically-protected sequences of bytes. This model was taken up for the NASD [5] and T10 OSD [8] object storage model, which added the notion of expiration.

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 [18] and CIFS [15]. AFS [7] 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 [17] and SanFS [10] are two examples of file systems that perform block allocation in the file management path. Object-based file systems derived from the NASD [5] model, including the Panasas [12] 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 [3], 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 [2] and SHARP [4] provide a way to ensure that users of a peer-to-peer storage system contribute as much storage 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 central servers quota enforcement can be done in a simpler and more efficient manner allowing centralized control for changing and viewing quotas and current usage.

Another peer-to-peer system [13] 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.

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 storage servers periodically reconcile actual usage with the management server in order to verify that all clients have behaved properly. The vouchers can be used any time after issued until they expire, thus trading temporal granularity for performance.

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' reqeusts 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] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

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

[3] 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.

[4] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM Symposium on Operating Systems Principles (SOSP)*. October 2003.

[5] 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.

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

[7] 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, pp. 51–81, February 1988.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] T. Okamoto and K. Ohta. Universal electronic cash. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pp. 324–337, London, UK, 1992.

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

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

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

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

[19] 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.