HSS: A simple file storage system for web applications

Abstract

AOL Technologies has created a scalable object store for web applications. The goal of the object store was to eliminate the creation of a separate storage system for every application we produce while avoiding sending data to external storage services. AOL developers had been devoting a significant amount of time to creating backend storage systems to enable functionality in their applications. These storage systems were all very similar and many relied on difficult-to-scale technologies like network attached file systems. This paper describes our implementation and the operating experience with the storage system. The paper also presents a feature roadmap and our release of an open source version.

1. Introduction

All web applications require some type of durable storage system to hold the content assets used to present the look and feel of the application and in some cases the code that runs the application as well. Web scale applications require that the content be published from numerous sources and be delivered from numerous sources as well. There have been quite a few solutions to this problem over time but many of these past solutions have presented scalability problems, availability challenges, or both. We have implemented storage systems based on traditional network file systems like NFS[1] and we have also implemented storage systems based on more scalable cluster file systems like IBRIX Fusion[2], Lustre[3], and OrangeFS/PVFS[4,5]. These solutions have drawbacks related to the tight coupling of the server and client, the restricted ability to add and remove capacity from the system, the recovery behaviors of the system, the ability of the system to allow non-disruptive upgrades of software, single points of failure built into critical parts of the system, and limited if any multi-tenant capabilities.

The server and client dependencies in clustered file systems create a difficult-to-manage issue around updates of the software. In general, mismatched client and server versions do not work together, so the entire system must be shut down to perform upgrades. The multi-tenant requirement to use the system as a service makes isolation within the clustered file systems difficult other than through traditional file system layout structures which have limited flexibility. The interfaces that clustered file systems provide tend to look like traditional POSIX interfaces but almost all of them have unusual behaviors that many applications cannot understand or workaround without changes. The requirement to make application changes to accommodate the file system specifics makes clustered file systems unattractive to legacy applications.

The storage system we have created seeks to improve on the availability and operational characteristics of the storage systems we have used in the past while providing easily consumable front end semantics for the storage and retrieval of files by applications. We provide a minimal set of operations and rely on external components for any additional functionality that an application may need. We have built several mechanisms into the system that provide data durability and recovery. We have also added a number of load management features through internal and external mechanisms to reduce hotspots and allow for data migration within the system. The system is aware of its physical makeup to provide resilience to local failures. There are also autonomic behaviors in the system related to failure recovery. All of these

features are built and implemented using well-understood open source software and industry standard servers.

We illustrate here the design criteria, explain why the criteria were chosen, and the current implementation in the system. We follow up with a discussion of the observed behaviors of the system under production load for the past 24 months. We then present the feature roadmap and our invitation to add on to the system by open sourcing the project as it is today.

2. Design

These are the design criteria that we have selected to implement in the system as required features. They are in our opinion of general use to all web applications. They are all fully implemented in the system today.

2.1 Awareness of physical system configuration

In order to be resilient to physical component failure in the system as well as failures of the hosting site infrastructure the system has awareness of its physical makeup. This awareness is fairly granular with known components consisting of storage devices, servers which are containers of storage devices, and groups of servers referred to as a site.

Storage device awareness prevents storage of a file and its replicas on the same storage device which would be unsafe in the event of a failure of that device. Servers and the storage devices they contain have the same merit in the system for data protection. None of the copies of a file should reside on the same disk or server.

The grouping of servers into a site allows for the creation of a disaster recovery option where an application that chooses to store files in multiple sites can achieve higher availability in the event of a failure of an entire site. There is no requirement that the sites be physically separate. The preferred configuration does include multiple sites that do not share physical infrastructure.

2.2 Data protection through file replication

Data protection can be accomplished through hardware, software or a combination of both. The greatest flexibility comes from the software data protection. Software data protection improves the availability of the system and site failure resilience. File replication is the chosen method in the system. While space-inefficient compared to traditional RAID algorithms, the availability considerations outweigh the cost of additional storage.

Software control of the replication allows tuning based on a particular application's availability requirements independent of the hardware used and of other applications. In addition to tuning considerations, the software allows us the flexibility to specify how file writes are acknowledged. We can create as many copies as necessary before the write is acknowledged to the application. In general, the number of copies created synchronously is two, and then any additional copies, typically one more, are created by the system asynchronously. These additional asynchronous copies include any copies made to another site in the system.

2.3 Background data checking and recovery

Data corruption[6] is of concern for a system designed to store large numbers of objects for an indefinite amount of time. With the corruption problem in mind we have implemented several processes within the system that check and repair data both proactively and at access time.

The system runs a background process that runs constantly and compares file locations and checksums to the metadata. The background scrubber continuously reads files from the system and confirms that the correct number of copies

of a file are available and that the files are not corrupt by using a checksum. If the number of copies is incorrect, the number of file copies is adjusted by creating any missing copies or removing any additional copies. If a checksum fails on any file replica then the corrupt copy is deleted and the correct number of replicas is recreated by scheduling a copy process from one of the known good copies of the file.

In addition to the background process that constantly checks data, file problems can be repaired upon an access attempt of a missing or corrupt file. If the retrieval of a file fails due to a missing file or checksum error the request is served by one of the other file replicas that is available and checksums properly. The correct number of replicas is then created by scheduling a copy process from one of the known good copies of the file to a new location. Metadata reliability is handled by the metadata system.

2.4 Internal file system layout

Each HSS server is comprised of multiple file systems, one for each physical drive; file systems do not span multiple drives. A single top level directory will be the location of all the subdirectories that contain files. The directory structure of the file system will be identical on all hosts but file replica locations will vary by host. This is necessitated because each host *may* have different capacities, but the desire is to balance utilization across all hosts and spindles, precluding placement in the same directory on all systems.

Each node will have some number of physical disks. Each disk will be mounted on the top level directory as a subdirectory. Below the mount point of the directory are 4096 directories with names aaa to 999 using the hexadecimal notation. This structure is used to prevent potential problems with having a very large number of files in a directory. In the pathological case that all files on a given disk are 2KB with 1TB drives, we will not go over 125000 files in any directory with even file distribution before filling any individual drive. In fact, the majority of files stored have sizes between 10KB and 100KB, which reduces the possible number of files on a disk before it fills to somewhere between 2400 and 24000 files. This should be manageable for the system even as drives get larger. In the futures section there is a discussion of how the large numbers of files can be managed.

2.5 High performance database for system metadata

Metadata scalability and availability have direct affects on the system throughput and is ability to serve data. A number of options exist for systems that we believe are reliable enough to serve as the metadata engine in the system. Depending on the performance requirements, a well-scaled RDBMS system has been found to meet the needs of the system for a large variety of workloads. An additional benefit of using an RDBMS system is that they are widely available and well understood.

We have implemented the system with MySQL as the primary metadata engine. The data structure within the system is rather simple with almost no relation between the tables. Even though there are few relational aspects to the data, the use of a well understood RDBMS simplifies the metadata management aspects of the system and the development effort required to create the system. Most of the data is rarely changed and most of the tables are used for configuration of the system. This simplicity in the database can help maintain high throughput. There are only six tables in the database as shown below in figure 1.

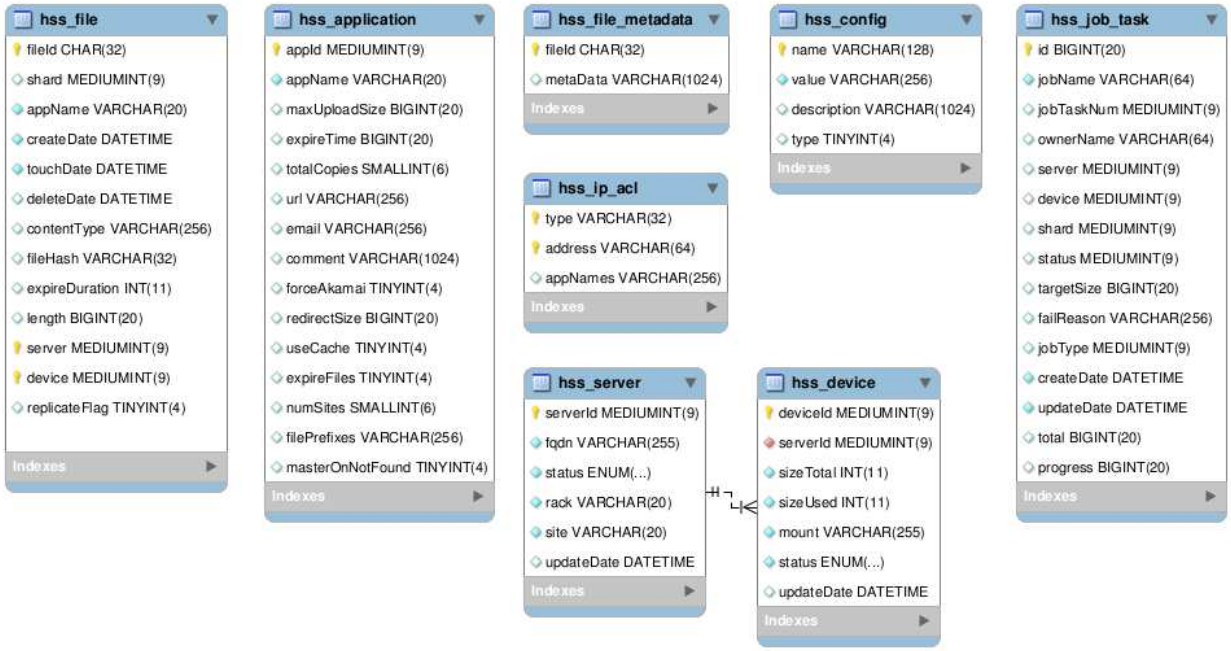


Figure 1

MySQL can be run in a partitioned or sharded fashion as the database grows to address potential performance concerns due to database growth. Replicas of the database are also maintained to create resilience to failures in the metadata system. A number of additional technologies exist from various sources to improve MySQL performance and availability while maintaining compatibility with the MySQL system. We have implemented several of those to improve the overall system.

We have also developed a version of the metadata system based on the VoltDB[7] database. VoltDB provides clustering for availability, an in-memory database engine for performance, and snapshots to durable storage for recovery and durability. The schema is unchanged when using VoltDB and the metadata engine is a configuration option for the storage nodes within the system.

2.6 Off the shelf open source components

The system is built from readily available open source software systems. The system is implemented on Linux servers running Apache Tomcat for the storage nodes and MySQL or VoltDB for the metadata system. These components were selected for their widespread adoption, functionality, transparency, and the deep knowledge about these systems prevalent in the technology community.

2.7 System scalability

The system has been designed to allow the addition or removal of capacity to increase both storage available and performance of the system. All of the changes to the system can be accomplished while the system remains online and completely available.

When additional storage nodes are added to the system, the new storage nodes are immediately available to take load from requesting applications. Additional write throughput from the new nodes is available immediately and read throughput of the new nodes comes online as files are stored on the new system through new writes or redistribution of data. Redistribution of stored data in the entire system or a subset

of the system is accomplished in the background to ensure that capacity utilization is roughly equal on all storage nodes in the system. It is anticipated that many files in the system are not commonly accessed and therefore those files can be moved without impact to the applications that own them.

Nodes can be removed from the system to scale down or migrate data. The system can be configured to disable writes to nodes that are being removed and then files are copied in an even distribution to all other remaining storage nodes. Once additional copies of the files have been made on the rest of the system, the files on the node being removed are deleted. This process continues until the node being removed has no files stored.

Metadata scalability tends to be more problematic but can still be accomplished within the constraints of the RDBMS being used for the metadata. In general, it is difficult to scale down the metadata engine. It is the expectation that the system will only increase metadata capacity and if the system is scaled down then the overhead in the metadata system will have to be tolerated.

2.8 Simple interface

The system is accessed though a very small number of web calls that, while somewhat restrictive, cover a large number of the use cases for web application storage systems. The system can write a file, read a file, delete a file, and replace a file. File object IDs are provided by the system in most cases but there is also a capability to allow applications to define their own object IDs. The system does not allow the updating of files. Examples of the system operations are discussed in section 3.

The external operations for the system are listed in Table

Operation	Example
Read	GET /hss/storage/appname/fileID
Write	POST /hss/storage/appname
Delete	DELETE /hss/storage/appname/fileId

Table 1

All of the external operations are atomic and return only success or failure. The client application is not expected to have to perform any operation other than perhaps a retry if appropriate. While the capabilities of the system may appear restrictive, we have found that adoption has been quite good. There are a number of internal operations that are used to manage the system behaviors.

The internal operations listed in Table 2 are only available to the storage nodes themselves. Administrative operations are an additional class of internal operations that are used for system configuration and batched file operations. The administrative operations are listed in table 3.

Operation	Example	Description
Copy	POST /hss/storage/internal_copy	Store replica on another storage node
Move	GET /hss/storage/internal_move	Move a file from one storage node to another
Migrate	GET /hss/storage/internal_migrate	Migrate a file from one server to another during healing process. The file is not removed from the old storage node

Table 2

Operation	Description
/hss/admin/addServer	add a server to a site configuration
/hss/admin/addDrive	add a drive to a server configuration
/hss/admin/changeServerStatus	set the server status to one of read-write, read-only, maintenance, or offline
/hss/admin/changeDriveStatus	set a drive status to read-write, read-only, or offline
/hss/admin/addApplication	add an application name to the configuration for files to be stored
/hss/admin/updateApplication	change application specific parameters about the way files are managed
/hss/admin/addJob	add a job to copy files to or from a device

Table 3

2.9 Network load balancer

The storage nodes in the system are accessed through a network load balancer to provide even distribution of incoming requests. Access to the metadata system is also provided by a network load balancer. This provides several benefits in addition to providing uniform request rates to all of the system components.

The load balancer implementation can remove unresponsive system nodes in a very short amount of time. The failure interval is also configurable providing a flexible tolerance for component failures. Unavailable system components do not cause requests to slow down or fail in the system except locally to a storage node for a very few requests that are in-flight during the actual failure.

Load balancers insulate client applications from the particulars of the system and allow changes to be made without having to update any external part of the system. The same is true of access to the metadata engine. The metadata is accessed through load balancers that provide access to the metadata system for reads, writes, or administrative background jobs independently and in some cases making use of replica copies for workload where data may not need to be up to date.

2.10 ACLs via IP address

The system has very little in the way of access controls. The system is designed to provide service to applications and the applications are expected to provide fine grained access controls and file security required. There are some basic controls available to restrict the system and assist applications with access control. The ACLs are provided not to prevent access to specific data or a subset of the data but to protect the system from bad external behaviors.

Read and write ACLs are defined for the system as a whole and on a per application basis. ACLs are specified by IP address. The system will reject write requests that do not come from systems that are explicitly defined within the ACL. The system will deny read request for systems that are explicitly defined within the ACL. The write ACL behavior is to only allow authorized systems to store files within the system. It is worth noting that only objects with file prefixes can be overwritten, so in the common use case for the system a file cannot be replaced and keep the same object ID. It is also possible for any allowed system to write files with any valid

application name. This can potentially become an accounting issue within the system but it doesn't place data at risk for overwrite or corruption in general. The read ACL behavior is to restrict reads from any clients that are found to be using the system improperly. The system is susceptible to bypassing application security to request an object if the object ID can be discovered.

2.11 Files served by proxy or redirect based on size

Any node may receive a request for any file in the system. Not all files will reside locally on every system node. The system serves file reads based on GET requests and those are treated like standard HTTP GET requests. If the file is not stored locally, a node will either retrieve the file from the node where it is stored and return the file itself or send a HTTP REDIRECT to the requester identifying a node that stores a copy of the file. The system implements a tunable size for the server redirect case. Files below the size threshold will be returned from the initial requesting node. Files above the size threshold will be redirected. This behavior allows the system to be tuned based on load and typical file size within and application.

2.12 Commonly served files cached

Typically we have seen that a small number of popular files are commonly requested. In order to avoid constant proxy or redirect requests a cache of commonly accessed files is implemented on the nodes. A tunable number of files can be cached on every node. The additional space consumed by the cache is more than offset by the read performance trade off of avoiding additional communication about file locations. The cached files also provide some insight into what files are popular and may be candidates for placement in an external CDN.

2.13 File expiration

The system has the ability to automatically delete files after a tunable interval has expired. The interval can be defined per application in the system. This enables the temporary storage of files for applications that make use of transient data. Deletion of files based on expiration offloads the need to keep track of file creation and lifetime from all applications that need this feature and implements it in the storage system consistently for all applications.

2.14 Application metadata stored with the file

The system provides a simple mechanism to store any additional metadata about a file along with the file record in the metadata system. This data is opaque to the storage system and simply exists as an assistive feature for applications. An application may store some metadata, perhaps a description, with the file and that metadata will be returned when the file is retrieved. The expectation is that this data is used within the storing and requesting application to provide some additional functionality. At some point it is expected that the system would be enhanced to make use of the application metadata stored to make the system more responsive and useful to application consumers.

2.15 Applications are system consumers with local configurations

The system assumes that files will be stored and retrieved by applications. Applications that make use of the system are expected to implement any necessary controls on files they store. The nature of the system keeps file storage as simple as possible and moves responsibilities for much of traditional content management and fine access control to external entities that make up the applications.

2.16 File accounting

While the system does have the concept of application to file relationships for multi-

tenancy concerns, it has no hierarchical namespace, so files stored are kept track of external to the storage system by the owning applications. This has the affect of allowing any file to be stored anywhere in the system for improved scalability and availability without concern for relationships between files. Multiple applications can store unique copies of the same file. While that duplication of content is space-inefficient, it prevents dependencies between applications within the storage system. The space overhead is preferred to potential conflicts between applications. It may be possible in a future update to the system to provide object level de-duplication to eliminate excessive copies of the same file.

The relationship between application owners and files allows the system to provide utilization reporting for capacity planning and usage metering. The system provides sufficient detail about files to report on capacity consumed per application, how much data is transferred per application, file specific errors, and several other usage based metrics. These metrics are used to predict when the system will require expansion or when access to files has become unbalanced for any reason. The metrics also enable the examination of usage trends within applications so that customers can be notified of observed changes to confirm that those changes are as expected.

2.17 Monitoring integrated with existing systems

There are several external mechanisms that are responsible for monitoring the system and its components. Based on events that occur in the system a number of external systems can be used to manage and adjust the system using the administrative operations. External systems are used to monitor throughput, software errors, and hardware errors. The external systems used are widely used industry standard systems.

Nagios[8], for example, is used to monitor the hardware components of the system. In the event of a disk failure, a Nagios monitor detects the occurrence. When that happens, an alert is sent to notify the system administrators, and the drive status is changed from read-write to offline automatically. This prevents any new requests for writes and any new read requests from using the failed disk drive and causes any at risk data to be re-protected by background processes.

3. Implementation

A high level diagram of the system and how the components interact is shown in Figure 2. A description of the external operations is provided to get a better feel for how the system functions from an application perspective. IP address access control checks are included in all operations.

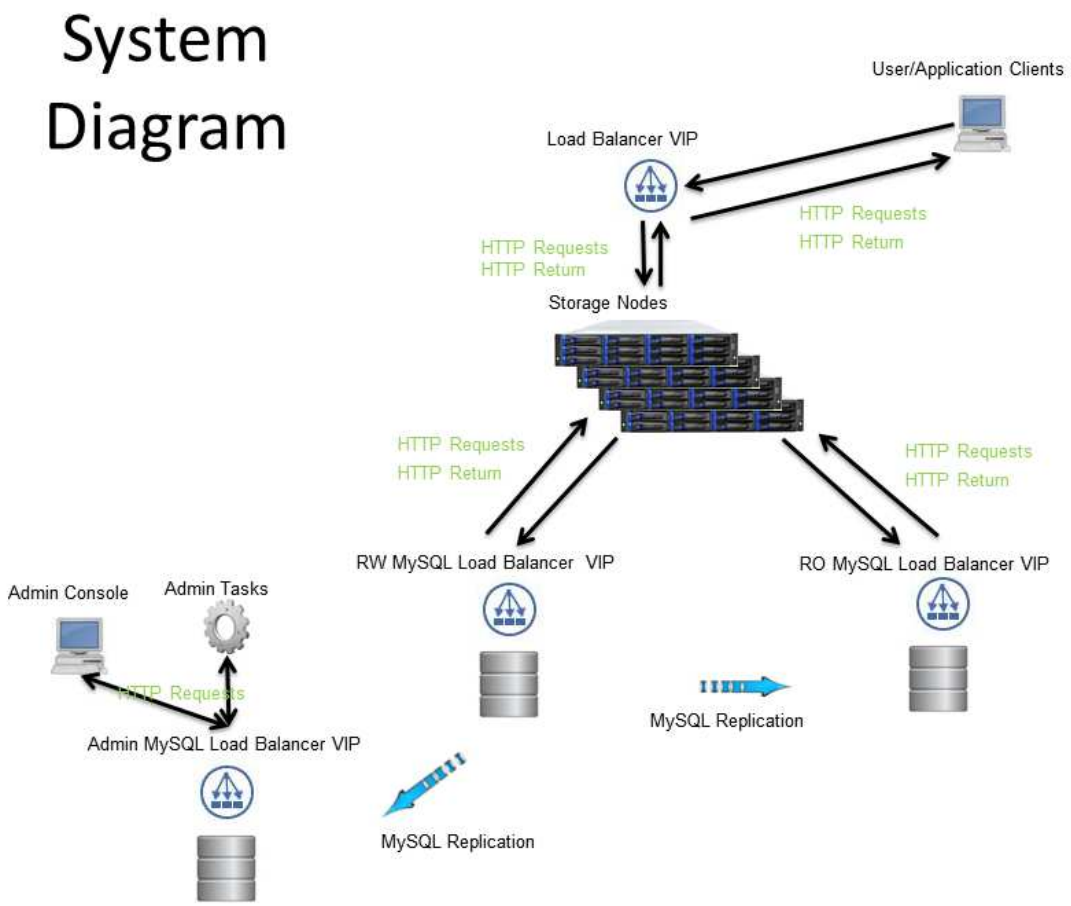


Figure 2

3.1 Write a file

When a POST request is sent to the system to write a file, the storage node that receives the request calculates the object ID and picks a local location to write the file. The storage node then inserts into the database the server, disk, and object ID of the file as well as some data about the file such as mime type and creation time. It then picks another node within the same rack if possible based on server availability and space free to write a second replica of the file. When the replica creation is successful the database is updated with the details that pertain to the replica. The write is then acknowledged as successful to the client once the minimum number of copies required is created. The default minimum number of copies is two but it is configurable per application. Based on the application configuration any additional replicas of the file are created at this time using the POST /hss/storage/internal_copy method from storage node that handled the original request. Once the minimum number of replicas is written a file is considered to be safe.

In the event of a failure before the first file write completes, the system retries the write and the process restarts. If the retries are unsuccessful after a limit is reached, the system returns a failure and the application is expected to retry the write. If any additional writes of file replicas fail, the system will retry the writes until the minimum replicas requirement is achieved and then any additional replicas will be created in the background.

If background replica creations fail, either by making too few or too many replicas, then the expectation is that the system background file check process will detect the improper number of replicas and take the appropriate action by making any additional replicas required or removing any additional replicas beyond what is required by the application configuration.

3.2 Read a file

A file read GET request specifies the application and objectID as part of the path sent to the system. The receiving storage node looks up the object ID and finds the location of all replicas. A replica is selected from the list of locations by preferring the replicas that are local if there are any, then within the same rack and then the same site. If the file is local it is simply served. If the file is not local, it is retrieved and served by the requesting storage node if it is smaller in size than the threshold specified for redirect by the application. If the file is larger than the redirect size, a redirect is served to the client that then requests the file for a server that has the file locally. In the case of a storage node serving a file it does not own, the file is placed in a local cache on the serving storage node with the expectation that commonly accessed files will be requested continuously from nodes that do not have a local copy due to the load balanced access to the system. This cache will avoid future lookups and file retrievals from other nodes for heavily accessed files. It also has the effect of making many copies of the file throughout the system to enable better concurrency when the file is requested.

In the case of a lookup failure from the database, an http 404 error is served to the requesting process. In order to avoid transient lookup failures a number of retry options are available. In the simple case a number of retries is attempted before returning the 404 error. In the case where a database replica is used to service some database lookups for read requests, a number of retries are attempted and then the lookup can be attempted on the master copy of the database if the application is configured to do so. The reasoning behind this second set of lookups is that replication lag between the master and the replica database may account for the lookup failure. If that is the case, the lookup in the master database should succeed if the replica database lookup fails. In general, the replication lag is less than 10 seconds within the current system, but some applications exhibit a usage pattern that causes an immediate read after a write that makes the lookup on the primary necessary. The metadata replication we use is asynchronous so it is not possible to require metadata replication to complete before a write is acknowledged to the client. Alternative metadata storage systems that have synchronous replication would also solve the problem of the replica metadata database being slightly behind the primary metadata database.

Occasionally a replica copy that is selected by the system will fail to be available either because the file is not found on the owning server or the file will fail the checksum when it is retrieved. If either of these cases occurs, the file will be served from one of the other replicas in the list returned by the lookup. The internal GET /hss/storage/internal_move method is then used to replace the missing or bad file replica.

3.3 Delete a file

The deletion of a file is the simplest of the external cases. The request to delete marks the files for deletion in the database and then deletes the replicas of the file and the records from the database. Any failures during the delete process are expected to be handled by the background checking processes of the system. If files are deleted

but the database entries remain for some reason, the records marked for deletion are removed by the background processes of the system. It is possible for the delete process to orphan files on storage nodes that are not referenced by the database. The process to recover from this occurrence requires looking up all files found on the storage nodes and comparing them to existing database entries. If the entries do not exist, the files can be safely deleted as there is no reference to them and they cannot be served.

3.4 Recovery of errors

A number of errors both well-understood and unknown are expected to occur within the system. These errors may include things like file system or storage node failures destroying data and causing the number of replicas to fall below the desired number, database lookup failures, storage nodes unavailable for extended periods due to hardware maintenance, network failures, and many others. The system is designed to be as resilient as possible to errors. The system has implemented recovery processes for the errors that are known to occur and updates are made to the system as new errors are detected and the root causes are discovered to prevent new errors from recurring.

3.5 Configuration

Application specific settings are managed from the administrative interface for the system. Application settings control the way files are stored in a number of ways. The specific parameters that can be set for an application are:

Expire Time – number of seconds a file will be kept before automatic deletion. Automatic deletion requires Expire Files to be set true.

Max Upload Size (MB) – The size limit of a file that may be written. There is no built in limit by default.

Copies – The number of replicas of a file required.

Sites – The number of sites that file replicas must be stored in.

Use Cache – Enables caching of often requested files on nodes that do not normally have local copies.

Expire Files – Enables or disables automatic deletion of files.

Use Master on 404 – Enables or disables queries for files that fail on read replicas of the metadata database to be retried on the master metadata database before returning a failure to the application.

Redir Size (MB) – The threshold size for a file when the request will be redirected instead of proxied by the storage node receiving the request.

File Prefixes – ObjectID prefixes that are specified by an application. An application may specify the ObjectID of a file and the system requires a consistent prefix for all of the files in that case. The prefix is usually the application name. This is useful for applications where the filename can be mapped directly to the file content such as a geographic location used in maps.

4. Results and observation

The system as described above has been in production since August of 2010. In that time it has served billions of files for dozens of applications. The system has good scalability and has serviced over 210 million requests in a 24 hour period. As of this time the system manages just over 380 million files. The file count includes all replicas. An analysis of the usage data from the system reveals that about one third of the current files in the system are active. The feature set is in-line with the intended use case. The feature set of the system also compares well with ongoing work on open storage systems designed for the same use case. Table 4 shows a feature comparison between the Openstack object storage system[9] (Swift) current and future features and the current features of the system presented here.

Open Stack Swift Storage System Current and Roadmap Feature Comparison

Feature	Openstack Swift	HSS
Store and manage files programmatically via API	Y	Y
Create public or private containers	Y	Y
Commodity hardware	Y	Y
HDD/Node failure agnostic	Y	Y
Unlimited storage	Y	Y
Multi-dimensional scalability	Y	Y
Account/Container/Object structure	Y	Y
Built-in replication	Y	Y
Easily add capacity	Y	Y
No central database	Y	N
RAID not required	Y	Y
Built in management utilities	Y	Y
Drive auditing	Y	Y
CLI	Y	Y
Improved client IP logging	N	Y
Option for replication	N	Y
Multi cluster syncing	N	Y
Large single uploads	N	Y
Self-destructing files	N	Y

Table 4

The load on the system is highly variable and dependent on external applications but the behaviors of the system are highly consistent as request load varies. Figure 3 shows the load over a thirty day period. Figure 4 shows the service time for all requests.

Daily Report

Each unit (▬) represents 8,000,000 requests or part thereof.

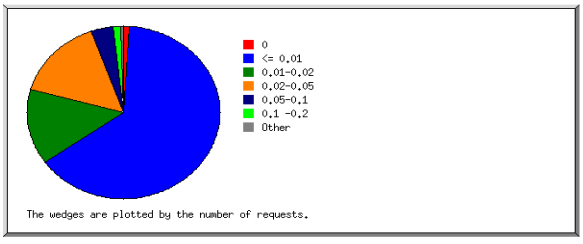
date	reqs	pages	Tbytes
14/Apr/12	34679957	117867	0.73
15/Apr/12	64603291	153938	1.36
16/Apr/12	82464950	180637	1.64
17/Apr/12	84348508	758508	1.64
18/Apr/12	85339078	356551	1.62
19/Apr/12	79331053	165346	1.58
20/Apr/12	89434038	227067	1.53
21/Apr/12	91277900	215410	1.28
22/Apr/12	71085800	209439	1.32
23/Apr/12	81083544	255295	1.62
24/Apr/12	77708581	300457	1.57
25/Apr/12	115020446	180783	1.50
26/Apr/12	102901308	130869	1.52
27/Apr/12	105428575	252088	1.50
28/Apr/12	92073035	185259	1.36
29/Apr/12	92852503	169668	1.25
30/Apr/12	111368504	194746	1.55
1/May/12	136847107	485617	1.54
2/May/12	158098801	508147	1.46
3/May/12	154312347	304603	1.48
4/May/12	141567232	268188	1.35
5/May/12	144097556	254431	1.21
6/May/12	152976182	232612	1.16
7/May/12	146743200	232870	1.58
8/May/12	119065960	378701	1.88
9/May/12	131036207	329123	2.15
10/May/12	109814165	187264	1.46
11/May/12	91329567	151346	1.34
12/May/12	148389620	115511	1.12
13/May/12	161159990	99295	1.14
14/May/12	212504364	121640	1.46
15/May/12	16494717	12697	0.09

Busiest day: 14/May/12 (212,504,364 requests).

Figure 3

In some cases days with fewer requests transfer more data than days with higher numbers of requests. This shows that not only is the request rate variable but file size is also quite variable.

Processing Time Report



seconds	reqs	%reqs
0	37514103	1.05%
<= 0.01	2296882027	64.00%
0.01-0.02	511234749	14.24%
0.02-0.05	547119494	15.24%
0.05-0.1	134610412	3.75%
0.1 -0.2	48795349	1.36%
0.2 -0.5	8474686	0.24%
0.5 -1	1974738	0.06%
1- 2	960778	0.03%
2- 5	599237	0.02%
5- 10	327617	0.01%
10- 20	299895	0.01%
20- 60	234591	0.01%
60-120	31508	
120-300	5914	
300-600	971	
> 600	388	

Figure 4

The overall observed response time of the system is quite acceptable with less than .5% of the requests taking more than 200ms and less than 2% of requests taking more than 100ms. This data includes all reads, writes and errors and we believe that the abnormally long response times are in actuality failures. The number of these abnormal events is very small and also quite acceptable.

Using only features of the system several infrastructures have been built and components have also been added and removed in existing infrastructures while the system has been under load and in production. The addition or removal of hardware components is simple and well-understood. The ability to update the system software components is more complicated but can also be done without taking the system offline. The existing systems have seen software updates, metadata database changes, and metadata database migrations without interruption in service.

5. Lessons learned

Many details of the system are implemented as conceived in the original design. There are still quite a few behaviors and functions of the system that can only be discovered though the use and maintenance of the system. The need to modify the way the system functions becomes apparent through its use by applications and administrators.

5.1 Data protection

There are several aspects of data protection that include both durability of the data and the availability of the data. Data replication within the system protects availability by ensuring that a file in the system is stored on multiple redundant components and it protects durability through that redundancy. There are some areas that require additional focus with file replication in practice.

The ability to rapidly recover large numbers of small files is limited due to several issues. Large numbers of small files in Linux file systems are difficult to access efficiently. Significant thought was given to creating a file system directory layout on the storage servers that would limit the impact of this problem but it still exists when very high numbers of small files are present even if they are efficiently stored in the file system. Additionally, small file transfers when files are replicated between storage servers are limited in the amount of bandwidth that can be utilized.

Another issue that occurs with the data protection availability aspects of the system is the I/O behaviors of the Tomcat servers when accessing a failed device of file system during the initial failure. There are a number of abnormal behaviors that prevent the system from returning the correct file to an application. Load spikes, crashes, and other oddities can occur when retrieving data. One response to the issue of failed physical devices can be to add local RAID data protection on a storage node to remove the possibility of a single device failure making a file system unavailable. This has the effect of increasing cost and complexity which are both to be avoided.

5.2 Metadata replication

Metadata replication is a requirement for the system in disaster scenarios as well as maintenance scenarios. It is temping to make use of the replicated metadata subsystem to assume some of the workload of the system. The benefits of this approach are that the replicated metadata subsystem is exercised to validate its proper functionality and that some load can be reduced on the primary metadata subsystem.

It is attractive to attempt to increase the total read capacity of the system by performing any reads from both the primary and the replicated metadata. The system is built with this capability in mind but the initial implementation suffered from the lag between the two systems. Applications that write a file and then immediately attempt a

read of that data initially received data access errors if the read was requested from the replicated metadata prior to the completion of replication. A retry case was added to always check the primary before returning a failure to the requesting client.

It is preferable to run administrative and maintenance jobs on the data from the replicated metadata subsystem. This prevents internal housekeeping from impacting the system. Many of the administrative jobs are long running and can be strenuous and having a secondary copy of the data isolates these jobs very effectively. Ad hoc jobs are also run against the replicated metadata subsystem.

5.3 Configuration tuning

The default configuration for the software components of the system are acceptable for integrating and bringing up the system. These configuration variables require adjusting to better manage the load and improve the overall experience with the system. There are many limits within the system based on the initial configuration that can safely be modified to increase the capabilities of the system.

Variables control memory utilization, thread limits, network throughput, and dozens of other functions within the system. These functions can be improved to the overall benefit of the system by examining the system and observing the limits. Many of the limits are imposed by the configuration and not by any physical or functional aspect of the system. Making informed changes to the configuration yields significant benefits.

6. Conclusion

To store and manage the large numbers of objects associated with web applications, the system is an extremely good fit. It offloads the need for application developers to manage where files are stored and how they will be protected. The system also provides a consistent interface so that application developers can avoid having to relearn how to store files for each new application that is developed.

From an operational point of view the system requires very little immediate maintenance due to its significant automation and redundancy. Repairs can be effected to subcomponents of the system in a delayed fashion without impacting the applications that use the system. Modifications to the system can be made while maintaining availability and durability of the data. These aspects make the system very lightweight for operators to manage.

The work done to develop this system and its usage in production applications for an extended period of time show that it is possible to build a system that is simple, reliable, scalable and extensible. The use of common industry standard hardware and software components makes the system economical to implement and easy to understand. There are certainly use cases where the system is not a good fit and should not be used but it is an excellent choice for the use around which it was designed.

There is room for improvement in the feature set and efficiency of the system. With that in mind there are some improvements that definitely should be investigated for integration into future versions of the system. A list of future planned improvements includes container files to address file management and performance concerns, lazy deletes to disconnect housekeeping operations from online operations, improved geographic awareness to improve access latency, and a policy engine to manage file placement and priority in the system.

References

[1] Network File system
http://tools.ietf.org/html/rfc1094
http://tools.ietf.org/html/rfc1813
http://tools.ietf.org/html/rfc3530
[2] IBRIX Fusion
http://en.wikipedia.org/wiki/IBRIX_Fusion
[3] Lustre
http://wiki.lustre.org/
[4] Orange File System Project
http://www.orangefs.org/
[5] PVFS
http://www.pvfs.org/
[6] Panzer-Steindel, Bernd Data Integrity April 2007 CERN/IT
[7] VoltDB
http://community.voltdb.com/
[8] Nagios
http://www.nagios.org/
[9] Openstack Object Storage (Swift)
http://openstack.org/projects/storage/