NIBFS

The Non-Indexed Blob File System
A High-Efficiency, Capability-Based, Storage System for Archival Websites

Jeremy T. Hilliker
University of British Columbia - Department of Computer Science
Breadth Project Essay
Friday, August 30th, 2008

Abstract

Image archival is a popular and high profile Web 2.0 service. We have examined the problem domain of internet archival websites, particularly image hosting sites, to discover the usage and characteristics unique to their problem domain. We have used this knowledge to determine how these services' needs differ from those offered by traditional posix filesystems, and then have constructed a tailored filesystem to better meet their needs. Particularly, our system reduces disk seeks caused by unnecessary meta-data and the domain's long tailed access distribution. Our filesystem offers a 40 to 55% improvement in throughput over standard filesystems, which is significant since these services are I/O bound. Increasing I/O efficiency for these services allows them to serve more content with fewer resources, and to scale better.

Introduction

Web 2.0 websites are built to allow users to collaborate with each other, share information, and to create content [5]. One application of Web 2.0 is online photo albums and image hosting. These websites allow users to post images to their accounts to be shared with their peers or their audience. This feature has bene adopted by many social networking websites which allow users to post images to be viewed by members of their social circle.

Some website place more or less of an emphasis on social networking or image hosting. Imageshack is strictly an image hosting site with no social component. Flickr is primarily an image archival site with a small social component (tagging, comments, friends, and groups). 4chan is an image message board with equal emphasis on image sharing and social interaction. Facebook is a social networking site with an image sharing component. Other sites such as Photobucket, Picasa, Blogger, Google Video, and Youtube provide differing mixes of social networking and content sharing and archival.

Facebook is currently the largest social networking site on the internet, and the 5th most popular website overall [6]. Though the primary function of Facebook does not appear to be image archival, it is nevertheless the largest image archival site on the internet with over 6.5 billion images (with 4 or 5 sizes each) occupying 540 terra-bytes of storage [2]. 475,000 images are served per second, (including 200,000 profile images per second) and 100 million images are uploaded per week.

Facebook currently serves 99.8% of its profile requests and 92% of its photo requests through content distribution networks (CDNs) such as Akamai and Limelight [2]. This results in approximately 452,600 images served through CDNs per second, and 22,400 images served per second by their own servers. The cost of using CDNs to this extent is prohibitive, but Facebook has had to use them due to their inability to scale their own servers and services fast enough.

They would like to reduce their reliance on CDNs to lower costs [2]. Facebook has 10,000 servers [3] and has had to borrow $100 million dollars to purchase more [4].

Facebook's scale of image hosting (as well as the other image hosts) makes them an interesting research case to discover if novel approaches to archival storage in their usage model can offer significant improvement (in either space or throughput) over existing approaches.

Problem

Facebook began with the naive approach of storing their image files in a traditional filesystem served over NFS by clusters of NetApp servers. These servers became heavily I/O bound, with as may as 15 disk reads required to serve each request [2]. The high number of reads was due to the meta-data (data about data) used by traditional file systems.

Every time a file is opened, the file's name and path must be resolved to an i-node which acts as a bookmark for the file. Each directory (a component of the path) has an i-node, and that i-node maps to the directory's contents on disk. When the final component of the path is resolved, the system gets an i-node which points to the file to be read. Only then can the filesystem begin to read the contents of the file. Each of these i-nodes is a piece of data that describes how to find the file on disk. In a deep directory structure, a large number of directory entries and their corresponding i-nodes are required to be read before the system can reach the requested file.

Facebook's first optimization was to develop a system to cache the mapping of filenames to their final i-node, and to modify the Linux kernel to allow files to be opened directly with a reference to that i-node. This reduced the number of required disks reads to approximately 3 reads per file [2]. They could perform this optimization because Facebook does not allow files to be moved, renamed or deleted within their storage system, so names would always resolve to the same i-node.

File meta-data can be cached in other ways, but most of it is unneeded for Facebook's application, so caching it wastes cache space which could be used for actual file data. Facebook, and most other image archival sites, are driven by databases which contain nearly all of the application's required meta-data. Having the filesystem duplicate this meta-data, or having it store un-needed meta-data, wastes resources.

Realizing that optimizations can be made by eliminating un-needed meta-data and by dropping support for unneeded filesystem functions can lead to a more efficient storage system for archival websites. The remainder of this paper will explore archival website's usage models (particularly image archives), and will build an efficient storage system for those models. Finally, the performance and characteristics of that system will be compared with traditional filesystems and with Facebook's solution to the same problem.

Analysis of Problem Domain

Image archival (and image portions of social networking) sites have some unique characteristics which differentiate them from regular filesystem usage models.

Access Pattern

On social sites such as Facebook, image access patterns follow a known distribution. Profile pictures (photo's identifying a user, shown in user listings, and as the main photo on a user's profile) are accessed the most frequently, followed by the most recently uploaded photos, followed by a very long tail [2].

This access pattern presumably translates to other archival sites. Identifying images are accessed most frequently as they are embedded in the most number of pages, followed by the most recent content as users distribute that content and view what is new, followed by the long tail of the website's archived content. Even on image boards such as 4chan, we can imagine a similar scenario. Images which start a thread of conversation will be the most accessed (as they are shown on the forum summary pages), the most recent images (which are on the first page of the forum) will be accessed next most often, followed by the long tail of old images in old conversation threads.

Filesystem Usage
The largest difference between archival storage systems and regular file systems is how the files are used and what is done to them. These systems are primarily driven by read operations where a file is submitted once and retrieved many times. This behavior makes an image archive like a write-once-read-many (WORM) storage system.

When the files are added, they are submitted as complete blocks (not streams) and they are not subsequently modified or appended. The systems are intended to be archives, so file deletion is rare. Related files are often submitted in batch (through a web-form with multiple file upload boxes), and related files are often retrieved close together (photos from the same album).

Filesystem Access
The files in the systems are always accessed by the user through a uniform resource locator (URL). These URLs have differing structures between systems, but they share some common characteristics. Example URLs are available at [1].

Analysis of these URLs reveal that they are most often not user-friendly. The URLs often contain some kind of volume identifier, an object identifier (OID), some kind of hash (presumably for replication and load-balancing), and a size indicator. The path is almost always completely meaningless to the user, and the filename component is sometimes meaningful, but often not. The following table presents a summary of the characteristics of archival site URLs.

	Facebook	Flickr	Photobucket	Picasa	Blogger	Imageshack	4chan	GoogleVideo	Youtube
volume ID	y	y	y	?	y	y	n/a	?	y
OID	y	y		?	?		y	y	y
hash	y			y	y	y		?	?
meaningful name			y	y	y	y			
meaningful path			?				?		

We can conclude that users are not intended to type these URLs themselves since the paths are meaningless to the user, and often long and complicated. Instead, they are intended to be embedded into generated web-pages for the user's web browser to fetch them automatically. The inclusion of user meaningful filenames appears to be a convenience feature to allow users to save images.

We also note that these URLs act as capabilities in the sense that knowing the URL grants access to the file. The URLs are tokens that give the holder of the URL the authority to retrieve the file. They do not care who holds them, they can be delegated, and they can be copied. None of the websites examined implement access control lists on the files or within the URLs. Simply knowing the URL is sufficient to retrieve the content, regardless of a user's privacy settings.

Threat Model

Archival websites are exposed to all kinds of users, including hostile ones. They must keep their services available and secure for regular users despite the efforts of active malicious adversaries. We will consider our system to be secure if it is at least as secure as the existing systems.

For archival websites, we want regular users to be able to store any image and to retrieve any images for which they have been given a valid URL. Adversaries are permitted to submit any data for storage into the system, including malicious data, and to submit any read request through any URL, including forgeries and maliciously crafted URLs. These users must be prevented from reading any files where they have not been given a valid URL generated by the system. In security parlance, we wish to prevent existential forgery when allowing the attackers to make chosen plaintext attacks, meaning that an adversary can chose to submit any data to the system and to receive valid URLs for that data, but they cannot create an URL to access any other data stored within the system.

We also want the system to be resistant to denial of service (DOS) attacks, but do not propose to make the system any more resilient than existing services. We merely want to ensure that our system does not create new avenues for DOS attacks.

Design Goals

Existing storage systems are I/O bound in this usage scenario, so our primary design goal is to maximize I/O efficiency. Images such as profile pictures and recent photos are more frequently accessed, so we can increase their throughput with replication. This is a well understood solution, so it will not be addressed in this paper. The other part of the problem is the large data-set and the long tail access pattern. Because of their wide distribution, less frequently accessed files cannot be efficiently replicated. For these files, minimizing the number of disk reads and disk seeks will be imperative to maintain throughput and I/O efficiency.

Another observation from our problem domain was that most file meta-data is not needed by these systems, so we will try to eliminate as much of it as possible. The elimination of the meta-data should allow the filesystem to be more efficient, and to conserve space for storage of actual data, both in memory caches and on disk.

Analysis of Requirements
From our above analysis, we can summarize the functional requirements specific to this domain in seven points:

1. Paths may be non-meaningful.
2. Filenames should be meaningful.
3. Submitted files may not be altered.
4. Files must be submitted as complete blocks.
5. Files must be "deleted" only in rare cases.
6. Files must only be retrievable via a valid URL generated by the system.
7. URLs must prevent existential forgery under chosen plaintext attacks.

From these seven points, we can make observations about our desired filesystem:

1. We can encode meta-data into the path component of the URL.
2. We should preserve original filenames.
3. Files can be contiguous on disk.
4. Files can be contiguous on disk.
5. Deleting is rare, so we can do it poorly.
6. We must detect and reject altered or forged URLs.
7. We must detect and reject altered or forged URLs.

Points three and four are the biggest advantage of our filesystem over traditional filesystems. The need to support file fragmentation is a large source of meta-data and disk seeks. Point one also allows us the chance to encode meta-data into the request URL itself, eliminating that data from disk and memory, allowing our system to be more efficient. Point seven requires us to verify any submitted URL, so we can trust that any meta-data contained in the URL is accurate.

Solution

The above design goals combined with the analysis of requirements suggest that the best way to build our file system is to eliminate all meta-data. This would ensure that any read which reaches the disk is used to retrieve actual user data, and not meta-data, maximizing disk efficiency. This is possible since we can encode the meta-data into the URL of the file.

When files are guaranteed to be contiguous, all that is required to read a file is a device (or volume), an offset into that volume for where the file is located, and the length of what is to be read. We propose to encode these values directly into the URL as a tuple: < volume, offset, length >. Since we wanted to preserve filenames, we ad that to the tuple < volume, offset, length, name >. Since the filename is in the tuple and we do not need it to read the file, we do not need to store it on disk.

Just using the tuple < volume, offset, length, name > would allow attackers to submit any request and have that request served. For example, and attacker could submit the request < v, 0, v.length, x > and read the entire volume. This violates the two security requirements of our system.

We could combat this by maintaining a list of valid start positions, but this would introduce meta-data that our system has to maintain and check for every request, which is contrary to one of our design goals. Alternatively, we could encode a header at the start of every file to indicate a starting position, but this won't work since we allow an attacker to submit malicious data for writing into the system. An attacker could craft a special file containing the start-of-file header, and then use it to read the rest of the volume. We could cryptographically secure the header so that it cannot be forged, but this is again adding meta-data to our system.

We observe that we have already shifted all of our meta-data from the filesystem into the URL given to the users, so we propose to move this cryptographic security to the URL as well. Rather than cryptographically securing the file header, we eliminate the idea of the file header and instead cryptographically secure the URL given to the users. This can be done by attaching a message authentication code (MAC) to the < volume, offset, length, name > tuple, giving us a new tuple of < volume, offset, length, name, mac >. If we use a MAC which is resistant to existential forgery under chosen plaintext attacks, then our URLs given to users will be resistant to the same attacks.

Message Authentication Codes

A message authentication code (MAC) is a short piece of information used to verify the integrity and authenticity of a message. A MAC paired with a message can be used to detect tampering (either deliberate or accidental) of that message. Any changes to the message would generate a different MAC.

A MAC is an easily computable function of the form: $mac(m) = d$ where m is a message, and d is a digest of that message. For a MAC to be secure, we insist that it is computationally infeasible to find any two messages which correspond to the same digest. MACs differ from digital signatures in that they do not provide non-repudiation.

For our application, we have chosen to use a keyed-hash MAC (HMAC) to provide URL authentication. Other MAC schemes can easily be substituted if desired. An HMAC combines a public message with a fixed hash function and a secret key to securely establish the integrity and authenticity of a message. An HMAC is a function of the form $HMAC(h,k,m) = d$, where h is a fixed hash function, k is a secret key, m is a public message, and d is the message digest. Secure HMACs are resistant to existential forgery under chose-plaintext attacks, and the security of an HMAC lays in the security of the underlying hash function h, and in the secret key k.

HMAC was chosen due to its wide-spread use in other protocols, the existence of universal standards, extensive cross-platform library support, and extensive review in the field of cryptography.

HMAC is implemented as the following function [7]:
$HMAC_{h,K}(m) = \mathrm{h} [(\mathrm{K} \oplus \mathrm{opad}) \parallel \mathrm{h}(\mathrm{K} \oplus \mathrm{ipad}) \parallel \mathrm{m})]$
where ⊕ is the XOR operator, || is the concatenation operator, and ipad and opad are constants.

For out initial version, we have chosen to use SHA-224. SHA-224 is the smallest hash function in the SHA-2 family, producing a 224 bit message digest, offering 112 bits of computational security (see [8] for details.)

The hash function can be easily changed later as it is simply a parameter to the HMAC function. SHA-2 was chosen over SHA-1 and MD5 due to recent attacks against the latter algorithms, and over RIPEMD due to SHA-2's wider use in government and industry, and more scrutiny in the field of cryptography. Extensive cross-platform libraries for SHA-224 are available.

URL Encoding
Our URL is now a tuple of the form: < volume, offset, length, filename, mac >. The size of each of the fields is specified in the table below.

field	bits		
	min	max	likely
volume	16	32	32
offset	32	64	64
length	32	64	32
mac	160	512	224

We cannot place the values in the URL as octets since octets are not URL safe [9], so the octets are required to be encoded in an URL safe format. Encoding requirements are listed in the table below for different encoding schemes.

bits	digits			
	decimal	hex	percent	base64
32	10	8	12	6
64	20	16	24	11
160	49	40	60	27
224	68	56	84	38
512	155	128	192	89

For small numbers (up to 64 bits), it appears completely feasible to encode the value in hexadecimal. Encoding the MAC in hexadecimal would require 56 digits which may result in undesirably long URLs. Using encoding methods such as quoted-printable, uuencoding, yenc, ascii85 or standard base64 would all generate characters that are not URL safe and therefore not suitable for our use. An URL safe and filename safe variant of base64 [10] could reduce the encoding length of the MAC to approximately 40 characters.

Support for File Deletion
The system as described so far does not support file deletion. The URL for a file acts as a capability to access that file. As such, once the capability is given out, it cannot be revoked. The solution to this in capability systems is to either have expiring capabilities (which must be renewed), or to introduce a level of indirection. Expiring capabilities are not suitable for our general use since URLs should be persistent. They may be useful in cases where the URLs are only encoded into generated web pages and not given to users directly.

Adding a layer of indirection unfortunately introduces meta-data to our system which cannot be removed. Either a list of deleted files must be maintained per volume, or each file must have a file header which indicates its availability/deletion status. We have chosen the latter approach as it requires less meta-data to be held in memory. Storing the deletion status in a file header should not introduce new reads or seeks to the filesystem since the read request for the header will cause the system to perform a read-ahead on disk to cache the data.

Implementation

This section of the report will summarize the implementation consequences of the proposed solution, and will solidify interface requirements.

Volume Header

We propose that each volume within our system specifies its MAC family and algorithm in its volume header. This will allow for later revisions to use different MAC families (such as UMAC, CBC-MAC, PMAC, or CMAC instead of HMAC) and different underlying algorithms (such as SHA-1, MD5, RIPEMD, or any member of the SHA-2 family instead of SHA-224). Each volume should specify its own secret key *K* to prevent birthday attacks against different volumes. This key can also be stored in the volume header.

As such, a volume header is required to store the following information:
< MAC family, MAC algorithm, MAC key, next free space >

The value "next free space" holds the offset on disk of the location of the next byte of free space. A new file is written to this location, and the free-space pointer is advanced to the first byte after that file.

MAC Computation

The MAC function takes the < offset, length, filename > part of the URL's tuple as the input message, *m*, along with the volume's hash function, *h*, and secret key, *K*, to generate the MAC for that URL. The < offset, length > components are binary values encoded in network byte order (little endian can be used as well, but the choice of encoding must be consistent across all nodes in the system). There is no additional value in encoding the < offset, length > into a human readable representation first since that representation would contain the same entropy as the binary representation.

The filename is not required by the system for opening a file, but it is still protected by the MAC for two reasons: to introduce additional entropy (though the value of this is minimal since the attacker can choose filenames), and to prevent user embarrassment through forged filenames. If the filename was not protected, then < volume, offset, length, mac, "meeting of math society.jpg" > would be as valid as < volume, offset, length, mac, "meeting of flat earth society.jpg" >.

URL Encoding

The < volume, offset, length > part of the tuple are encoded into the URL as hexadecimal values. The MAC component of the URL is encoded in the URL/filename safe variant of base64. An example URL is as follows:

```
/nibfs/v0x001/o0x9f9c1/l0x52ee/mZ1uIWeggibsSBO_EufTbL5jP6VG_edVb4dlS9i==/
          file.0x9f79a.21K.jpg
```

corresponding to the following values:

	encoded	meaning
volume ID	0x001	1
offset	0x9f9c1	653,761
length	0x52ee	21,230
mac	base64u(HMAC$_{h,K}$(nbo(offset) || nbo(length) || filename))	
filename	file.0x9f79a.21K.jpg	

The position of the path components are not significant, but we feel that the placement of volume and offset first will aid with replication and load-balancing using existing tools.

API

Write

The write call takes a collection of files as input (file descriptors, lengths, and filenames), and returns a mapping of those files to the generated URL's parameters. The side effect of the write call is that the files are stored into the volume, and the volume is prepared to accept the next files by updating the next-free-space pointer.

Read

The read call takes the URL's parameters and returns a reference to the file within the system. This reference includes a file descriptor, offset, and length (which will match the input parameters).

For every read request, the input parameters (the offset, length, and filename) are used to recompute the MAC with the volume's secret key and hash algorithm. If this computed MAC differs from the input MAC, then it indicates that the URL has been tampered with and should be rejected.

If the MAC validates, then a read request is made to retrieve the file header to see if the file has been deleted.

Result

Following this scheme requires only one disk seek to retrieve a file. All required meta-data can be decoded from the URL. The file deletion header must be verified for each request, but since deletion is rare, the system can preform a read-ahead to fetch the contents of the file on the same seek.

Related Work

Facebook has developed their own solution (named "haystack") to this problem [2]. Their solution uses a similar technique of having a single, large, non-block allocated volume for storage. The major difference is that Facebook's URLs are object identifiers (OIDs) which key into an index for each volume. This index maps the OID to the required length and offset values required to read the file from the volume. To mark a file as deleted, it is removed from the index.

Facebook achieves URL authentication by attaching a nonce to each OID. The nonce is verified against an on-disk copy of the nonce stored with each file for each read operation.

The index file requires approximately 20 bytes of storage per file, and the nonce is 2 bytes, requiring 22 bytes of extra data per file. This does not seem like much, but with an average file size of approximately 20 KB, and 540 TB of storage, it amounts to approximately 540 GB of memory to store the indexes (which must be in RAM to enable fast lookup), and 54 GB of storage for the nonce data. The space taken in RAM by the index is space which cannot be used to cache or preform read-ahead.

Memory requirement for file index:

Scale	Volume Size	Index Size
Facebook	540 TB	540 GB
Performance Server	16 TB	16 GB
Commodity Server	4 TB	4 GB
Sample Single Volume	16 GB	16 MB

As indicated in the above table, each server's entire RAM allocation can become occupied by the index data. Our solution's advantage is that it does not require the 540 GB of RAM required to store the indexes, but trades it for the extra computation cost in verifying the MAC. We feel that this is a better trade-off since the system is I/O bound with CPU time to spare.

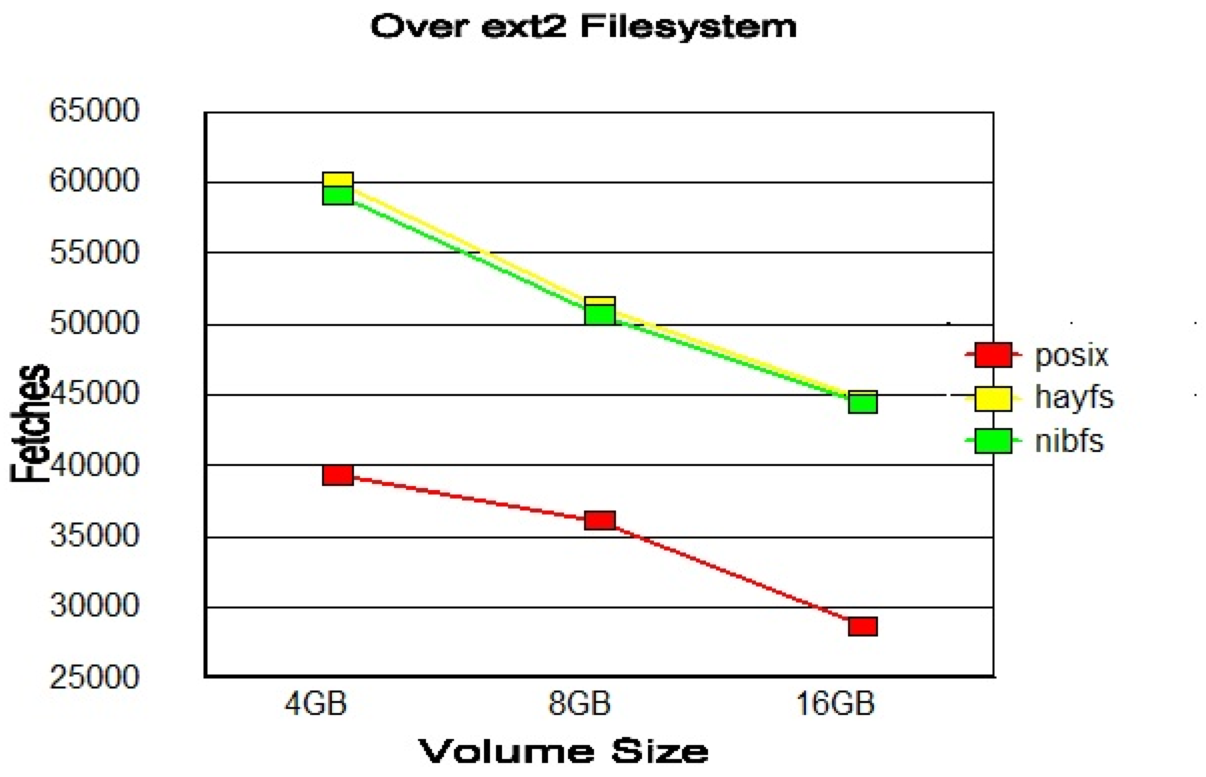
Performance

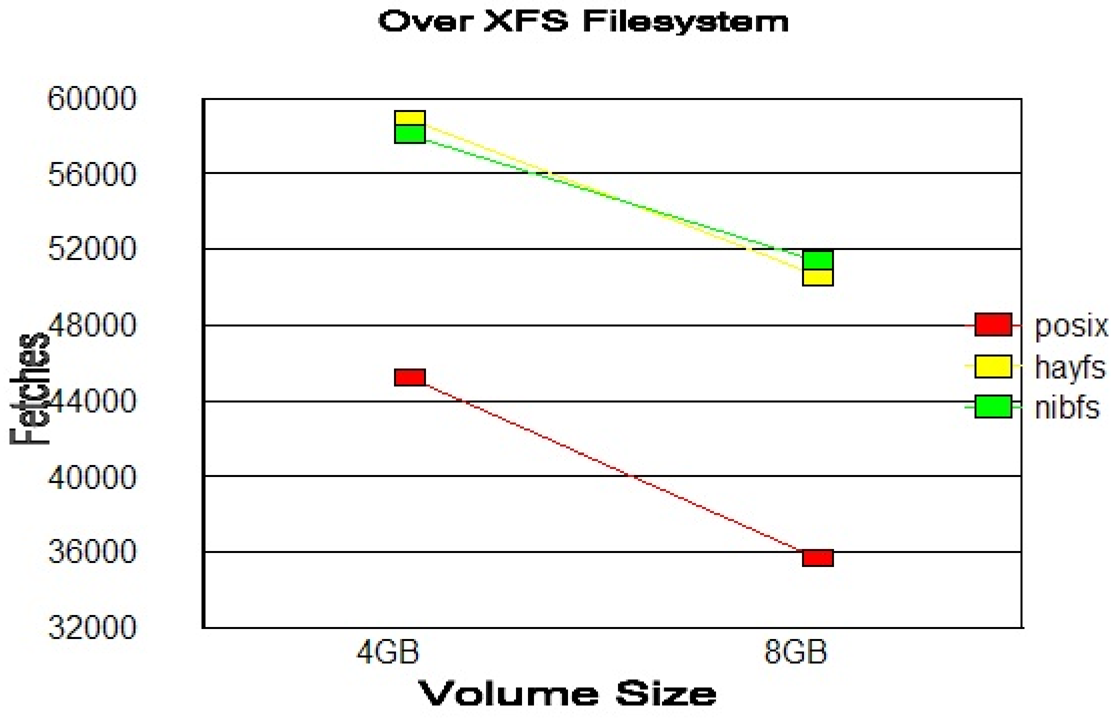
NIBFS was implemented along with Haystack for performance testing against regular file systems. The test environment was a simple multi-threaded web server, and an http load generator, chosen to match the problem domain. The server was fixed to operate with 20 threads, and the load generator maintained 20 concurrent requests.

Fresh filesystems were created at varying sizes and filled with test data representative of the operating environment (three images of 12K, 19K, and 22K). The http load generator would make concurrent requests to the http server which would fetch the file from the filesystem and send it back to the load generator as an http response. File were chosen at random over the entire volume to simulate the effect of the problem domain's long tail distribution.

NIBFS and Haystack can operate as files in existing volumes, so they were both tested on top of two underlying filesystems, ext2 (the Linux filesystem) and XFS (Sun's server filesystem). NIBFS can also operate with no underlying filesystem, but it's performance in this scenario is similar to when there is an underlying filesystem (provided that the underlying filesystem is not fragmented).

To prevent large directories when testing the posix filesystem, files were hashed into two levels of directories where each level contained 256 folders. Not performing this hash results in extremely poor performance for the posix filesystem.





NIBFS outperforms ext2 posix by 40 to 55%. NIBFS outperforms XFS posix by 30 to 40%. In all cases, NIBFS performs approximately the same as Haystack.

Aside: Reclamation of Deleted File’s Space

When a file is deleted, the file space can be reclaimed by maintaining a list of free blocks on disk. When a file is submitted, the free-space table can be searched to find a block larger than the new file. The new file (and it’s header) can be written into this block as long as the old file’s header remains at it’s original location. This will prevent the old URL from being able to read the new file in the reclaimed space.

We decided that since file deletion is rare in our problem domain, that this functionality was not worth it’s added complexity and meta-data. Haystack made the same design decision.

Conclusion

Our system has tackled the problem of how to minimize meta-data in a filesystem tailored for archival websites, thereby reducing the number of seeks required to retrieve a file to one. This optimization is necessary due to these site’s long tail access distribution pattern causing them to be I/O bound. Frequently accessed files can be cached or replicated, but the long tail cannot, so our solution is necessary to achieve and maintain high throughput and I/O efficiency as data sets grow.

Benchmarks of our solution show it to outperform regular filesystems by 30 to 55 percent under the problem domain’s usage model. This improvement will allow internet archival systems to serve more content using fewer resources, and to scale better.

Our solution performs as well as a competing related work from Facebook, but our solution requires far less RAM to operate. Freeing these resources can reduce costs, allow more services to be offered by the servers, or simply to allow the servers to perform larger read-ahead to fetch related data.

Construction of our access URLs supports replication, load balancing, and industry standard tools for content distribution and proxies.

Security of our scheme is maintained by industry and government standard techniques. The required functions are standardized and widely used, ensuring extensive, robust, library support across platforms. The extra computation required by our solution to maintain security is negligible, especially considering that these servers are I/O bound with plenty of idle CPU time.

References

[1] Hilliker, Jeremy. NIBFS Project Blog. 2008-08. http://nibfs.blogspot.com/

[2] Sobel, Jason. Needle in a haystack: efficient storage for billions of photos. Retrieved 2008-08. http://www.flowgram.com/f/p.html#2qi3k8eicrfgkv

[3] Miller, Rich. "Facebook Now Running 10,000 Web Servers." April 23rd, 2008. Data Center Knowledge. http://www.datacenterknowledge.com/archives/2008/04/23/facebook-now-running-10000-web-servers/

[4] Miller, Rich. "Facebook Borrows $100 Million to Buy Servers." May 12th, 2008. Data Center Knowledge. http://www.datacenterknowledge.com/archives/2008/05/12/facebook-borrows-100-million-to-buy-servers/

[5] "Web 2.0." Retrieved 2008-08-26. Wikipedia. http://en.wikipedia.org/wiki/Web_2.0

[6] "Alexa Top Sites." Retrieved 2008-08-26. http://www.alexa.com/site/ds/top_sites?ts_mode=global.

[7] RFC2104 - HMAC: Keyed-Hashing for Message Authentication. 1997. http://www.faqs.org/rfcs/rfc2104.html

[8] RFC4634 - US Secure Hash Algorithms (SHA and HMAC-SHA). 2006. http://www.faqs.org/rfcs/rfc4634.html

[9] RFC3986 - Uniform Resource Identifier (URI): Generic Syntax. 2005. http://www.faqs.org/rfcs/rfc3986.html

[10] RFC 4648 - The Base16, Base32, and Base64 Data Encodings. 2006. http://www.faqs.org/rfcs/rfc4648.html