2011 Sixth IEEE International Conference on Networking, Architecture, and Storage

EMFS: Email-based Personal Cloud Storage

Jagan Srinivasan
EMC[2]
jaganvasan@gmail.com

Wei Wei*, Xiaosong Ma‡*, Ting Yu*
*North Carolina State University, ‡Oak Ridge National Laboratory
wwei5@ncsu.edu, ma@cs.ncsu.edu, yu@csc.ncsu.edu

***Abstract*—Though a variety of cloud storage services have been offered recently, they have not yet provided users with transparent and cost-effective personal data storage. Services like Google Docs offer easy file access and sharing, but tie storage with internal data formats and specific applications. Meanwhile, services like Dropbox offer general-purpose storage. Yet they have not been widely utilized, partly due to their fee-charging nature and long-term service availability concerns.**

Web-based email services, on the other hand, have been offering growing email storage capacity, reliable service, and powerful search capability, making them appealing as storage resources. In this paper, we examine the efficacy of leveraging web-based email services to build a personal storage cloud. We present EMFS, which aggregates back-end storage by establishing a RAID-like system on top of virtual email disks formed by email accounts. In particular, by replicating data across accounts from different service providers, highly available storage services can be constructed based on already reliable, cloud-based email storage. This paper discusses the design and implementation of EMFS, focusing on unique challenges and opportunities associated with utilizing email services for file transfer and storage, such as email-based data organization, metadata format and management, and handling provider-imposed anti-spam usage restrictions. We evaluated EMFS extensively with multiple benchmarks, and compared its performance with NFS, AFS, and a non-free cloud storage service built upon Amazon S3. Our results indicate that while EMFS cannot match the performance of highly optimized distributed file systems with dedicated servers, it performs quite closely to the commercial cloud storage solution.

I. Introduction

Recently, a variety of cloud storage services have emerged and provided different levels of storage abstractions. Web applications, such as Google Docs and Adobe Buzzword, offers not only various applications but also online storage to support file upload and backup. However, they tightly bind cloud storage with specific applications, and have to convert existing documents into certain internal formats before storing them. Besides potential compatibility issues, such storage services often have limited functionalities compared with general-purpose file systems. Migration between different service providers also presents a challenge.

Other commercial solutions do provide a file system interface for the back-end cloud storage. Representative examples include Jungle Disk [1], Dropbox [2] and Windows Live SkyDrive. These services allow users to access the cloud by mounting a virtual disk on the client machines. Unfortunately, these services have not been widely utilized. This might be partly attributed to their fee-charging nature. Another reason is that users tend to have long-term service availability concerns. Should a company go down, a user's data may become unavailable. These services typically solely rely on one underlying cloud, and cannot yet guarantee data reliability/availability. For example, in our experiments using Jungle Disk in a period of two weeks, even as a paid user, we have twice encountered service interruptions.

Meanwhile, many users today have been explicitly or implicitly using web-based email services as a portable storage/backup tool. There are several reasons that web-based email services might offer an appealing solution to personal storage. First, the capacity of a single email account has increased dramatically in recent years, ranging from several GB to even unlimited storage. Considering that a single user can easily own multiple accounts from different providers, the aggregated storage creates a sizable space for personal data backup and processing. Second, many email services are provided by reliable and reputable providers, such as Google and Yahoo. Email services, even for uncharged accounts, are rather stable and long-lasting, offering additional merits for as personal data repositories. Though email services are not immune to technical failures, as users can easily obtain multiple accounts from different providers, replication techniques can be more naturally adopted for better reliability.

Based on the above observation, we present EMFS, an email-based personal cloud storage system. While the design of EMFS shares many common challenges and solutions as existing distributed file systems, we focus on email-specific issues and opportunities. For example, email protocols are not optimized for synchronous data transfer, creating obstacles for fast interactive personal data usage. Also, email service providers often impose restrictions on the frequency of emails, making it difficult to provide reliable file access. Meanwhile, email services also offer functionalities not available in traditional disks. For instance, email protocols like IMAP support quite flexible content-based email retrieval, which can be leveraged to simplify metadata design. Further, email services are inherently append-only, allowing us to leverage log-structured file system techniques. Our design must take such challenges and opportunities into consideration.

We do recognize that though online email services are often free of charge to users, they are not really free. The access to a large amount of user's personal data is a great asset to email service providers, for purposes such as personalized advertising. A personal storage system leveraging online email service infrastructures would also benefit service providers as it extends their access to valuable customer data in terms of both volumes and variety. Note that our approach does not cause more privacy concerns than existing online email services do.

978-0-7695-4509-7/11 $26.00 © 2011 IEEE
DOI 10.1109/NAS.2011.55

Our major contributions are summarized as follows:

- We propose EMFS, a personal cloud storage solution based on online email services (Section II). While this idea has been exploited previously [3], [4], [5], EMFS is novel in that it views email accounts as virtual disks and employs RAID-like approaches for space aggregation, data striping, and data replication.
- We examined the feasibility of using email transfer protocols for general-purpose file access and providing traditional file system interfaces. In doing so, we addressed many unique design challenges and issues, such as anti-spam usage restrictions, metadata and file data organization, and data placement (Sections III and IV).
- We implemented a proof-of-concept prototype of EMFS via FUSE [6], and evaluated it with comprehensive experiments, using two widely used file system benchmarks and a synthetic personal file access workloads. Performance comparison is conducted with NFS, AFS, and Jungle Disk, a charging cloud storage service (Section V).

II. EMFS Overview

A. Target Workload and Assumptions

EMFS provides an email-based cloud storage, where a user construct his/her personal storage space from web-based email accounts for easy, portable accesses. We focus on dealing with the typical personal workload [7], including reading, editing, and backing up documents such as Word, pdf, etc. While media files such as pictures and music are often candidates of uploading in today's home computing setting, gigabyte level files such as large movies are less common contents for email-based portable storage. Therefore our design targets file sizes ranging from several KBs to tens of MBs.

EMFS targets personal file accesses, and is designed to be accessed by a single user, with the assumption that this user will not share storage with others or allow concurrent access to his/her data. This is also due to the consideration that EMFS does not have a full fledged server to handle issues such as real time synchronization, conflict detection and resolution, etc. These issues cannot be solved utilizing just a client without severely hampering performance. These assumptions simplify the design of EMFS and this paper's discussion focuses more on email-specific challenges and opportunities.

B. EMFS System Architecture

Figure 1 illustrates the EMFS architecture, composed of two layers: the EMFS client and the email storage cloud. This email storage cloud provides storage through email accounts from providers such as Google, Gawab [8], and AOL. The EMFS client presents an approximate-POSIX file system interface based on FUSE [6] for the email storage cloud. Note that the EMFS package does not have any server side code. It utilizes existing email services and standard email protocols such as SMTP and IMAP to store and retrieve data.

Email services: Email services are shown at the bottom of Figure 1. EMFS ultimately stores its data in emails provided by email service providers such as Google, Gawab and AOL.

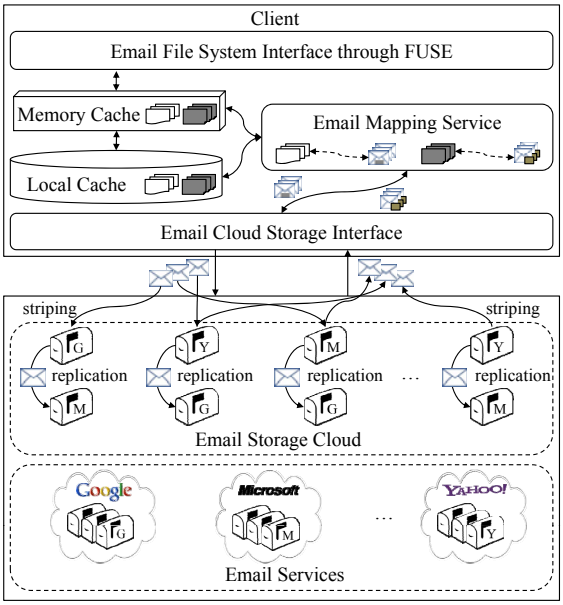


Fig. 1. EMFS design overview

These email services usually run their own massive data centers. EMFS builds a reliable personal email storage cloud on top of such existing infrastructure.

Email storage cloud: This is a logical component in EMFS, where we treat email accounts as virtual disks. By striping and replicating data across these "email disks", especially accounts with different service providers, we receive benefit in several aspects: performance, reliability, and capacity.

Client: At the client side, EMFS presents an approximate-POSIX file system interface via FUSE [6], which enables existing applications to run on top of EMFS without any modifications. Two layers of cache, a local disk cache and a memory cache, are used to speed up accesses. An email mapping component performs the translation between files/directories and emails. It also carries out storage management tasks, such as data placement in striping and replication. The EMFS client is implemented in around 3000 lines of Python code. The rest of the paper discusses in more details its design issues.

III. Data Organization and Access

EMFS shares the design goals and challenges of typical distributed file systems. In this paper, we focus on unique challenges presented by email-based file system construction.

EMFS is built on top of email services provided by third-party email service providers. Data and metadata are stored as contents of emails, either as attachments or as part of the body of the emails. In EMFS, there are two types of emails, metadata emails and data emails. As their names indicate, a metadata email stores metadata for files and directories, and a data email is used to stored file data. These emails are sent to and received from email servers through standard email protocols such as IMAP and SMTP.

EMFS utilizes a special property of email services: the server-side search feature. Most major metadata and data operations require EMFS to search for and retrieve specific emails. This search and retrieval is carried out using IMAP commands. For example, when a file open call is invoked, EMFS searches for the required emails, including metadata emails, on the server using certain unique identifiers (to be discussed in the following sections). Once the server confirms that the requested emails exist, EMFS can retrieve them to the local machine. Deletions are a bit cheaper and are completely server-side operations. The emails to be deleted are searched in the same fashion as above. Once found, IMAP commands are sent to the server to authorize their deletion.

A. File Organization

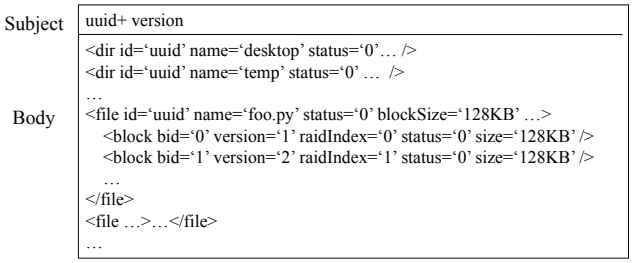


Fig. 2. Sample metadata email structure

Metadata: Metadata operations make up as much as half of typical file system workloads [9]. Considering EMFS's target workload – personal file editing, processing, and storage – and the email access latency, it is desired to reduce the number of metadata emails. Hence one metadata email is created for each directory, which contains the metadata entries for all subdirectories and files directly under it.

Figure 2 illustrates a typical metadata email, which stores metadata in XML. Each file/directory is assigned a unique ID (in the form of a version 3 UUID [10]). and a version number. The version number in EMFS serves three purposes. First, email storage is similar to log structured file systems [11], where all updates are appended. A version number helps with locating the emails containing the current data. The root directory has a special ID, and the latest root directory email in the server contains the up-to-date root metadata, which can be located by a email search supported by email servers. Secondly, it allows for consistency check and recovery from failures (to be discussed later). Earlier versions of metadata, available on the email servers (unless voluntarily deleted), can be used to construct a "snapshot" of the file system at various times of its lifetime. Such metadata snapshotting would be useful, among other things, to address consistency issues when failures occur [12]. Finally, though not implemented in this prototype, a versioning file system can be built rather easily on EMFS using these version numbers. The above usages of version number are further facilitated by the strong search features of today's email services. Note that the latest metadata email for a directory is retrieved using email search (which retrieves the latest email using given search criteria)and not by using version numbers. Thus, the metadata entry for a directory in its parent metadata email does not include a version field, which avoids the problem of cascading updates when a file or directory is modified. However, each directory carries and updates its own version number in its own metadata email, without propagating it to the parent directory's metadata email. When a full versioning system is needed, email search will be able to search the email subjects and bodies of emails simultaneously to locate a certain version of metadata.

The body of the email contains the metadata entries for all the subdirectories and files under this directory. There are three types of entries:

- <**dir**> **entry** contains attributes for a directory, such as its ID, directory name, and status.
- <**file**> **entry** contains attributes for a file, among which the most important ones are ID, name, status, block size, and file size. Each file entry will also contain one or more <block> entries, as described below.
- <**block**> **entry** describes a single block in a file, containing attributes such as the block index, block version number, RAID index, status, and data size. A directory metadata email contains the block entries for all the blocks of a file in that directory. This allows quick location of the data of a file. More details regarding the block settings will be discussed in Section IV.

The metadata organization in EMFS is similar to that of the Unix file system in the sense that it also decouples inode numbers and names. By separating ID numbers and file/directory names, metadata operations such as renaming becomes more efficient, as only the parent directory's metadata email needs to be updated. It is significantly different, however, in that EMFS combines the metadata contents of files and directories under a common directory in one single metadata email, rather than storing such data separately within a separate email (inode) for each of them. This greatly reduces the number of metadata message retrievals, and receives more benefit from local metadata caching. With typical personal file system image size and composition [13], and the mail body size allowed by today's web-based email systems, we do not expect to be concerned with the space limit problem, as each metadata entry averages around 70 bytes. This design choice is further supported by the fact that email send/receive efficiency is low with small messages, as shown in Section IV-A.

One drawback of this design is that apparently, any change to the metadata of a file/directory causes the resending of the parent directory's metadata email. However, considering the small sizes of typical metadata emails, the latency is tolerable in our experience.

Another potential problem with the above design is the hierarchical lookup overhead over emails, as we need to go through a sequence of metadata emails along the path of a certain file/directory when we first retrieve it. EMFS's dumb server handicaps the use of a path based lookup as rename or move operations will prove to be extremely expensive due to cascading emails. To this end, EMFS explores two

optimizations.

First, we experiment with a rather conservative metadata prefetching policy, which prefetches metadata for directories directly under the current directory.

Second, we also evaluate the effectiveness of a client-side global lookup table (GLT), which performs translation between a given full path of a file or directory and its unique identifier within EMFS. Besides server-side search, this approach leverages another unique feature of email services: custom email flags. Entries are added to the table upon creation of the corresponding file or directory. A miss in the GLT for a particular path implies that the corresponding object does not exist in the file system. The GLT is also stored on the server side in the form of an email, where lazy updates are adopted. Previous versions of the GLT are deleted whenever a new update to the table is sent out. Note that only operations that change the full path of a file or directory object, such as rename or move, trigger a corresponding change in the GLT. To ensure system recovery after failures, metadata emails that reflect changes to the path of a file or directory are marked as "GLT-DIRTY" using email flags. These flags are deleted following a successful synchronization of the GLT with the email servers. When EMFS starts up, the system searches the email servers for emails with these "GLT-DIRTY" flags set. Upon finding such emails, the system knows that a possible failure has occurred and its GLT is inconsistent. EMFS updates the table accordingly by scanning the "GLT-DIRTY" emails to restore consistency. The setting and deletion of email flags are cheap server-side operations: they do not require the download of email messages to the local system. Our results show that the use of the GLT helps in improving the lookup performance of the system.

File data: File data organization in EMFS is rather simple and straightforward. Any file is treated as a byte stream and divided into blocks of size *blocksize*, a tunable parameter. We discuss in more details about setting this parameter in Section IV-C. Each block of the file is sent in a separate email. As mentioned earlier, there is a block metadata entry for each block of a file in a metadata email. The block id, version and file id are all required, in order to uniquely identify a block of a file. These three pieces of information together, known as a *block identifier*, are always unique globally in the file system, and need to be looked up in the parent directory's metadata email before a file block can be accessed. For fast and efficient retrieval of the blocks themselves, the subject line of each file block email contains its identifier.

B. Metadata and Data Access

Client cache management: The EMFS client maintains a local disk cache to store file data. Any time a file is written to the server (whenever an `fsync()` or `flush()` is invoked), data is first written to the disk cache and then sent out to the email servers. On retrieval, a block is cached for future requests. EMFS assumes there is always enough space on local disk to cache required files, which appears reasonable considering personal file access workload characteristics [13]. Note that file data in the disk cache will not be deleted even when the EMFS root is unmounted: updated metadata, plus the version numbers, will indicate whether file blocks are stale due to the user's access from another machine. This way, data reads can be greatly expedited on subsequent mounts. Additionally, whenever a file is opened, the full file is read into a memory buffer. Writes are buffered in memory until `flush()` or `fsync()` is invoked, when data will be synchronously flushed to disk and the email servers. This memory buffer is cleared when the file is closed.

All metadata pertaining to a file or directory are cached in memory upon first access. This in-memory metadata cache is deleted on system unmount. To ensure consistency considering users' access from different computers, the metadata cache is flushed periodically, with a frequency configurable by the user. In our experiments, we used a frequency of once every 12 hours. Caching is an optional parameter of EMFS and can be turned off if needed.

Metadata update: Metadata update is immediately and synchronously committed to the server, with a design to reduce the number of such update emails. For example, for `mkdir`, a new directory metadata entry is added to the metadata email of the current directory, and the version number of the metadata email is increased by 1. This email is then immediately committed to the server. Since the new directory does not have any files or subdirectories, the creation of its own metadata email is delayed until a file or directory is created under it. A similar strategy is followed for newly created files as their size is 0. `rmdir` is another example. To delete a directory, we mark the target directory's as "deleted" in the metadata email of its parent directory, and this email is committed to the server. The actual cleanup of its files and subdirectories, and the removal of the target entry from the metadata can be performed lazily as a part of garbage collection, to be discussed in Section III-D. Doing so has another potential advantage, in that it may be useful to restore accidentally deleted files and directories. This feature is not currently implemented however.

Data access operations: Above we discussed the semantics of directory and file creation. Here we briefly describe other common file access operations. Opening a file involves validating its existence against the parent directory metadata email. If successful and file data has not been cached, EMFS fetches file data in the background, which facilitates subsequent reads.

As mentioned earlier, with `write()` calls, updates are stored locally until `flush()` or `fsync()` is invoked. These operations cause the concerned metadata email and file block emails to be sent out. At the end of the write operation, EMFS sends one more metadata email asynchronously to confirm that the file transfer takes place correctly, to ensure consistency. On file release, the file is checked to see if new data has been written to it. If so, the file is flushed to the email server. If no update has been made, the file is simply closed and deleted from the in-memory cache.

Finally, when the FUSE `release()` interface is invoked, `flush()` or `fsync()` will be called if the file is found dirty.

C. Consistency and Failure Recovery

EMFS assumes that data loss cannot occur once any data or metadata email has been transferred successfully to the email server. Except when the client system crashes, EMFS expects to receive acknowledgment from the email server confirming successful transmissions. In the absence of such a confirmation, EMFS assumes that the email has not been received by the server and will resend updates. EMFS tries to resend updates three times (a configurable parameter) before notifying the user of failure. EMFS also uses *noop* or heartbeat operations to check server status and keep connections to the server alive.

Considering the email system's append-only nature, EMFS adopts a mechanism similar to that used in LFS [11] to ensure the atomicity of updates. Whenever a file needs to be written to the server, one metadata email and as many data emails as needed, are sent out to the email server at the same time. This metadata email has a "status" field set for the file that is being transferred, which indicates that the file is dirty. Once the system receives confirmation that the data blocks have been transferred successfully, it sends out another metadata email with the status field cleared. Such a pair of metadata emails help EMFS check the consistency of files and roll back if necessary.

By default, EMFS makes all transfers to the email servers synchronous and atomic, blocking users till the confirmation has been received from the servers. However, the system can be configured to a "lazy" mode, where all updates take place in the background. With the single user assumption of the current EMFS implementation, the performance gain of using this weaker mode may outweigh the risk of losing updates.

D. Garbage Collection

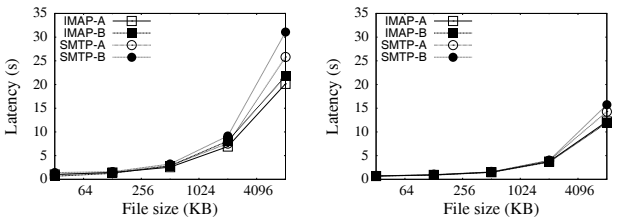
When a full versioning system is not needed, garbage collection has to be performed to reclaim storage space. EMFS does this lazily by regularly scanning the whole file system, checking all versions of metadata, and delete redundant emails from the servers. Again the garbage collection aspect is similar to log-structured file system. However, with email storage on the web-based services, we do not need to worry about the disk space fragmentation problem, which simplifies the garbage collection design.

IV. Email-based File System Design

A. Email Protocol Selection

The most common protocols in use today for transfer and retrieval of emails are the Simple Mail Transfer Protocol (SMTP), the Internet Message Access Protocol (IMAP), and the Post Office Protocol (POP). SMTP can be used only for transferring emails to the server, while IMAP and POP are primarily used for retrieving emails.

In selecting email protocols, we have found that the key constraint comes from the usage control imposed by email service providers. While message size is not a problem, popular services all have rather strict policies regarding the number of messages an account can send in a certain amount



(a) Gmail (b) Gawab

Fig. 3. Email sending and appending performance

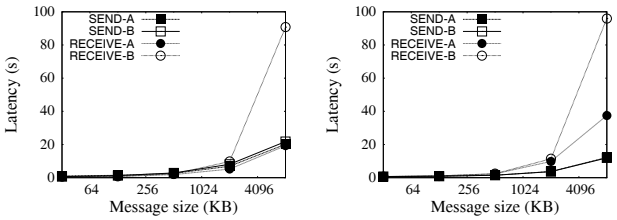
of time through SMTP, for purposes such as spam control. We found such limits quite easy to reach even with a single-user, personal file access workload and with EMFS's scheme to use per-directory metadata emails. IMAP, on the other hand, allows users to "append" a message to their own mailbox, and is not limited by traffic restrictions. Performance wise, Figure 3 shows the latency of sending a file via the two protocols, using mail body and attachment respectively. IMAP is faster than SMTP in almost all cases, by 5.5% on average and up to 42.64%.

For retrieving emails, IMAP is also a much more powerful protocol compared to POP, providing complex remote access to mailboxes, such as multiple client connections to the same mailbox, flexible message retrieval, and access to individual parts of a message.

Therefore, EMFS employs IMAP for both sending and retrieving messages. This does have one drawback: messages sent using SMTP can be automatically forwarded to other accounts, which is an easy way for data replication. (In fact, when sending from the user's one account to another, replication is achieved by having a copy in the sender's sent-box and the receiver's inbox). With IMAP, replication has to be explicitly performed by uploading redundant messages to different accounts.

B. Data Placement Within Emails

Data can be stored in multiple places in an email - the headers, the subject line, the email body, and as attachments. The first three locations are ideal for storing metadata because we can then use IMAP to search these fields on the server, without download the emails. In EMFS, metadata is stored in the body section, while the unique identifiers are stored in the subject line.



(a) Gmail (b) Gawab

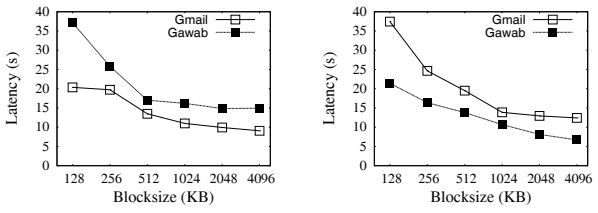
Fig. 4. Single email sending/retrieving performance

For file data blocks, the only possible placement options are bodies or attachments. Today's web-based email service providers have rather generous message size limits (e.g., 10MB for Hotmail, 25 MB for Gmail, and 50MB for Gawab), certainly large enough for file blocks. To choose between the two, we conducted experiments to measure the latency of sending and retrieving a single email, with varied message payload sizes as email body or attachment (Figure 4). The payload contained binary data mixed with ascii text. For sending mails (appending via IMAP), the performance is similar regardless of whether the payload is placed in the body or the attachment. However, the placement of the payload in the attachment slightly outperforms the placement in the body with Gmail. For retrieving mails, when the payload size grows beyond 2MB, latency of using mail body dramatically increases. We found that the increase is mainly due to the time taken to transform the downloaded string into an email message format. As attachment in general outperforms body, EMFS stores file blocks as email attachments.

C. Block Size and File Striping

Each email account used in EMFS is viewed as a virtual disk and RAID [14] systems can be built on top of a group of such email disks. In our prototype implementation, we experimented with simple RAID composition, with striping and mirroring. n email accounts are organized into an array, with each account identified by a "RAID Index" from 0 to $n-1$. Data blocks are striped across email accounts to improve the aggregate throughput. Metadata emails are usually small, so they are not striped, which also simplifies email retrieval.

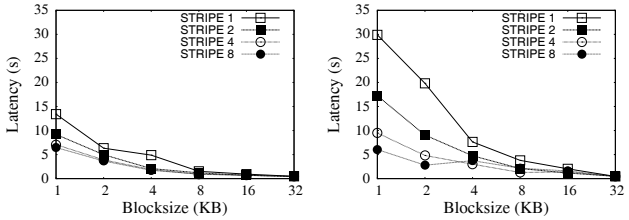
Instead of having a fixed array of email disks and striping data in a round-robin manner, EMFS takes on a more flexible approach at the cost of additional metadata. Blocks are stored on randomly chosen disks, provided the disk has enough free space. The *RAIDIndex* parameter in every <*block*> metadata entry carries the index of its destination disk. As metadata size is not a major concern, saving such complete mapping from file blocks to email disks allows for easy capacity expansion when new email accounts are added to the system.



(a) Read (b) Write

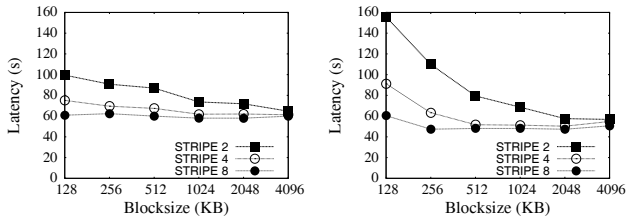
Fig. 5. File read/write performance without striping

The striping setting is intertwined with block size selection, which greatly influences EMFS's performance since it decides how many emails should be sent or retrieved for a file. To examine the trade off in using different file block sizes, we first conducted a group of tests to measure a 4MB file's read/write latency, where emails, each with an attachment (block) of size ranging from 128KB to 4MB, are sent or retrieved from a single email account (Figure 5). The results show that the file access latency steadily decreases when we increase the file block (attachment) size, for both Gmail and Gaweb mail.



(a) Read (b) Write

Fig. 6. Gmail access latency of 32KB file



(a) Read (b) Write

Fig. 7. Gmail access latency of 32MB file

Using very large file block sizes, however, could risk losing the transfer parallelism brought by striping, when data to be read/written cannot be split into enough number of blocks. Figure 6 and Figure 7 show the effect of striping with different blocksizes on EMFS's performance, with a small (32KB) and a large (32MB) file respectively. Clearly, striping provides a significant performance improvement. However, for the small file, the benefit of using large block sizes seems to outweigh the loss in parallelism. For the larger file, as the stripe width grows, file read/write performance increases in general, while the performance sensitivity to block size decreases, due to the saturation of network bandwidth. Increasing the stripe width beyond 8 or the block size beyond 1MB does not help the performance. Block sizes smaller than 256KB, on the other hand, degrades performance in almost all cases. Based on these results, EMFS uses 512KB as its default block size and 8 as the default stripe width. Note that messages smaller than the block size are not padded in transferring or processing.

D. Data Replication

As mentioned earlier, EMFS assumes that data received by the email servers will not be lost permanently. On the other hand, it has to prepare for the unusual occasions of service interrupt. As it is highly unlikely that multiple service providers experience down time concurrently, EMFS employs replication through IMAP to mirror data across multiple email accounts from different providers. Considering the rather low failure rates of major web-based email services [15], lazy replication is chosen for apparent performance advantages. In

EMFS, each virtual disk is associated with a *replication group*, which consists of two or more disks mirroring the same data. All disks in a replication group are assigned the same RAID index. Updates will be written to one of the email disks within the group synchronously, and to the others by a client-side replication daemon lazily, as IMAP is not able to directly send messages from one account to another. Email disks (accounts) can be added or removed from a group.

With multiple email accounts carrying mirrored data, we have a choice in selecting which account to use in read/write operations. In EMFS, we examine two replication strategies:

- *Read-one and Write-one*: all reads and writes from EMFS go to the same email account, which acts as a primary email account. The other accounts are not used in file accesses till the primary account fails, in which case another account will be selected to become the primary.
- *Read-fast and Write-fast:* reads and writes go to different accounts based on their uploading and downloading performance. This optimization is based on our observation that some email services provide better performance for reads and some for writes. Hence, EMFS writes to the email servers with higher write speeds, and reads from those with higher read speeds. Replication occur lazily to all accounts, which is not a problem for reads due to local caching.

The replication daemon is also in charge of data recovery when a failed server (account) is back into service. It maintains a pending replication job list, which can be used for both lazy replication and recovery. With this list, not only can a failed account be restored, another account can also be updated into the current state if the primary account (the account EMFS reads from within a replication group) suddenly becomes unavailable.

V. Performance Evaluation

A. Experiment Setup and Workloads

We use three benchmarks to evaluate different aspects of our system, and compare it with three existing distributed file systems, JungleDisk [1], NFS and AFS[1]. Experiments were run on an Intel duo-core desktop (2.66 Ghz) with 3 GB of RAM running Ubuntu 8.10. The machine was connected to the Internet using a wired connection with a download speed of 6.3 Mbps and an upload speed of about 4.8 Mbps, measured using online internet speed testing sites. Both NFS and AFS servers were configured on dedicated machines inside the campus network, similar in configuration to our test machine. Jungle Disk, a commercial service that stores data on the Amazon S3 servers (and charges for both data storage and transfer), was configured such that background or asynchronous transfers were disabled.

We evaluated EMFS with accounts from Gmail and Gawab Mail. We also performed experiment with a university webmail system. The results are quite similar to those with Gmail, and were omitted. In our tests, 8 accounts from both Gmail and Gawab are used, allowing a stripe width of 8. Considering the mirroring overhead across two services, the smaller quota from Gmail will impose a space limit. Therefore, this setting aggregates around 7.5GB$\times$8, i.e., 60GB of free storage space.

In our evaluation we used two widely used file system benchmarks (Postmark Version 1.51 [16] and IOZone Version 3.287 [17]), as well as one synthetic personal file access benchmark. Details of these benchmarks will be described later. Since EMFS is currently designed for personal usage, multi-threaded benchmarks were not considered. However, FUSE itself is inherently multi-threaded and initializes a pool of 10 threads to serve concurrent file system requests. All experiments were run for at least three times and the average value was reported. Error bars have been shown for experiments whose performance variance is over 6%.

B. Performance Results

Postmark: Postmark measures performance for network based systems by simulating access on short lived small files. It creates a large number of files of various sizes and measures the time to finish a series of transactions (read, append, creation and deletion), so it tests both data and metadata accesses. Before the benchmark finishes, any remaining files are deleted completely. A single transaction involves two operations: read/append and create/delete. The bias parameter of Postmark can be configured to adjust the ratio among these operations. We report the time that each system takes to finish all the transactions.

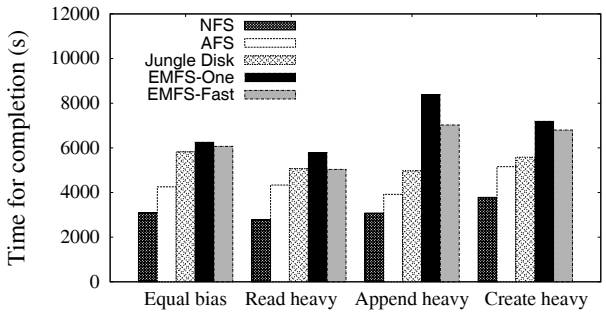


Fig. 8. Postmark performance

Figures 8 shows the results in campus network. The tests were configured to use 200 files, 200 transactions, with file sizes ranging between 4 KB and 16 MB. We generated four workloads (equal bias, read heavy, append heavy, and create heavy) by varying the operation bias. For EMFS, we tested two strategies, Read-One-Write-One (EMFS-One) and Read-Fast-Write-Fast (EMFS-Fast), as described in Section IV-D. EMFS-One reads from and writes to Gmail accounts considering the service's popularity. EMFS-Fast reads from Gmail and writes to Gawab accounts. With both strategies, EMFS does lazy replication between Gmail and Gawab accounts.

[1]We had planned to compare EMFS with GmailFS IMAP [3], which is probably the closest to our system. However GmailFS IMAP is unstable and often crashed with the benchmarks used. So the comparison cannot be finished.

Not surprisingly, AFS and NFS perform better than EMFS and Jungle Disk – both are highly optimized systems running in the kernel, utilizing and optimizing raw connections for data transfer, and with smart servers. Further, both NFS and AFS servers were set up inside the campus network, meaning that data transfers took place over the local network rather than the Internet.

EMFS offers comparable performance to Jungle disk, especially for balanced or read-heavy workloads, as both systems get to utilize their own local caches as well as the OS page and buffer caches. For append-heavy and create-heavy workloads (where 2743 MB and 3134 MB of data respectively is transferred over the network), EMFS is not as efficient as Jungle disk. The reason is that email protocols are not designed for fast continuous transfer of large amounts of data. Though all systems use strict synchronous writes, due to its "dumb" server, EMFS has to send a second metadata email for every write to ensure consistency. Although it is sent in the background to minimize the performance penalty while providing consistency, it becomes expensive when a lot of files are written consecutively. Jungle disk, on the other hand, has dedicated servers.

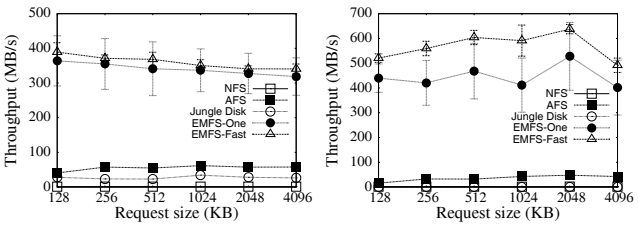
We also observe that EMFS-Fast does offer better performance than EMFS-One, especially for update-heavy workloads, due to its selective use of accounts for reads and writes.

To test EMFS's scalability, we also performed a 500-file, 200-transaction experiments, which last around 5 hours and upload around 5GB of data. The results obtained are very similar to those from the smaller tests.

Finally, personal users are more likely to utilize asymmetric networks, such as home internet connections, where uploads are much slower than downloads. We conducted the same experiments in home network settings with ADSL connection at different times during a day. The performance trends are almost identical to those shown in Figure 8. Due to space limit, we only report the results of the rest of our performance evaluation with the campus network setting.

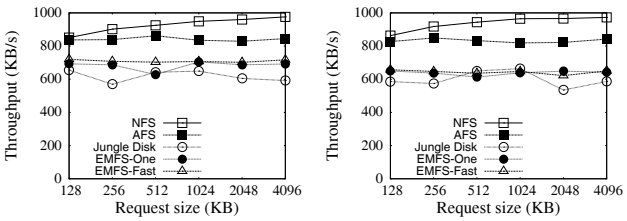
IOZone: IOZone is a popular benchmark for file system performance evaluation. Unlike Postmark, IOZone mainly focuses on file data access. It can be configured to test different file access patterns. This can be useful in evaluating EMFS's performance as a backup service. In our experiments, we created a file of size 16 MB and compare EMFS with other systems with two different workloads: sequential read/sequential write, and random read/random write, using varying request sizes. Furthermore, IOZone was configured so that every write was synchronous (using the O_SYNC flag) and the time taken by file close was also included in throughput calculations.

Figures 9 shows the read performance of a 16 MB file with read request sizes ranging from 128 KB to 4 MB. The test was conducted with a cold cache for all systems. AFS, Jungle Disk, and EMFS all download the file upon file open. However, from our examination of the IOZone source code, the benchmark does not include the file open time in throughput calculations. Therefore, AFS and Jungle Disk both serve read requests directly from their disk cache, leading to a transfer rate of



(a) Sequential Read (b) Random Read

Fig. 9. IOZone read performance. Error bars show 95% confidence levels.



(a) Sequential Write (b) Random write

Fig. 10. IOZone write performance

between 25 to 50 MB/s. The transfer rate for Jungle Disk is lower than that of AFS, mainly due to that Jungle Disk invokes a `setattr()` operation on file close to update the last accessed time. This is a network operation and restricts the Jungle Disk throughput to a maximum of 34 MB/s. The read transfer rate for NFS is only around 1 MB/s as NFS retrieves the entire file from the server before servicing the read requests. This is because with NFS3, disk caching is advisory and an external cache manager (such as CacheFS) is needed to enable caching, which was not configured on our test client. EMFS reports very high transfer rates (between 390 and 600 MB/s) for both sequential and random reads. This is due to that in addition to disk caching, EMFS caches the entire file in memory on file open, which is acceptable for personal workloads, where file sizes are relatively small.

Interestingly, Jungle Disk reports very low throughput (about 550-600 KB/s) for random reads. This is because IOZone opens up the test file in r+ mode for random reads. On file close, even though no changes have been made, Jungle Disk writes the whole file back to the server since it has been opened in r+ mode. This is extremely inefficient and significantly lowers the throughput for random reads.

Figures 10 shows the write performance of a 16 MB file in request sizes ranging from 128 KB to 4 MB. All systems display the same trend for all file writes, regardless of the write access pattern (sequential or random). This is because for all of them, any changes made to the file through `write()` calls are cached locally until a `flush()` is invoked. The changed file is then transferred to the server and hence the write throughput is dominated by the time taken to transfer the data across the network. In order to accurately reflect the time taken for a system to transfer a file to its server, the IOZone test was configured such that the time to close a file was included in

throughput calculation.

We see that EMFS is slightly better than Jungle Disk in terms of write throughput. This is because Jungle Disk sends out additional metadata updates, such as the last accessed time, separately on file close as mentioned before. This introduces an extra delay of about 0.5 to 1 second, resulting in the slightly lower throughput. NFS and AFS are faster due to their high file transfer performance and low overhead.

Editing Workload: This is a synthetic benchmark that simulates a document editing task. On each of the systems evaluated, we created a filesystem image, containing about 100 files (with sizes ranging between 8KB and 4MB) within 14 directories (with a maximum depth of 3). First, on a cold cache, lookup (`ls -l`) operations are executed, starting with directories at the deepest level (to assess EMFS's hierarchical lookup overhead), followed by operations performing `cd` to and `ls -l` in other directories. A total of 38 metadata operations are executed. After that, an Open Office document of size is opened and edited using a OpenOffice benchmark (part of the Linux Battery Life Toolkit [18]. It plays a typical document editing trace, containing operations such as inputting, formatting and editing text, search and replace, and finally saving the edited document. The document's size grows from 8.7KB to 26KB after playing this trace.

Figure 11 shows the results, categorized and normalized against the Jungle Disk measurement. Lookup operations for AFS is lightning fast, as its client downloads enough metadata to perform a direct local translation between the full pathname to file ID. In NFS, lookup operations for a single path may involve several lookup RPC calls to the server. We are not clear about Jungle Disk's lookup behavior as it is proprietary. With EMFS, lookup operations occur in a hierarchical manner proceeding from the top most directory in the path to the target directory. Hence, the lookup overhead is considerably higher. In addition, the retrieval of even just 1 KB of data from email servers via IMAP takes 0.25 seconds due to protocol overhead. Furthermore, the latency of the AFS and NFS servers determined via ping packets of size 1K is only 2.7 milliseconds, as opposed to 21 milliseconds for gmail.com and 87 milliseconds for gawab.com. The hierarchical lookup overhead of EMFS can be partially alleviated by more aggressive metadata prefetching. The "EMFS-Prefetch" method prefetches metadata for one more level of directories under the current directory. Results in Figure 11 shows that this help reducing the total lookup time by 17.4%. The lookup-table design of EMFS ("EMFS-GLT") also helps in improving performance. It does not prefetch metadata, therefore saving both client- and server-side bandwidth, yet performing roughly as well as the "EMFS-Prefetch" method. Given the short total execution time (EMFS spends 0.72 seconds on the 38 metadata operations), we do not expect that users performing everyday personal file processing tasks notice a slowdown when using EMFS.

The bulk of the time taken for execution of the entire workload is taken up by the "editing" part of the workload, which costs around 11 minutes (650 seconds) in this benchmark. The "Edit" bar in Figure 11, however, shows the total response time to keystrokes and other editing operations. Here all systems perform nearly the same, as the editing operations are executed locally.

Only file open, and file save involve network operations. For save, EMFS-Fast does bring an improvement of 31%. Its performance is quite close to Jungle Disk, but both are far behind NFS and AFS, mainly due to the inefficiency of using un-optimized protocols to transfer small files to/from Internet-based servers. While the relative performance differences between systems is large, we consider the open and save overhead of EMFS or Jungle Disk rather safe from causing user frustration, as it tends to be hidden by much higher application overhead. For example, in this benchmark, the Open Office takes around 50 and 30 seconds to startup and exit, respectively.

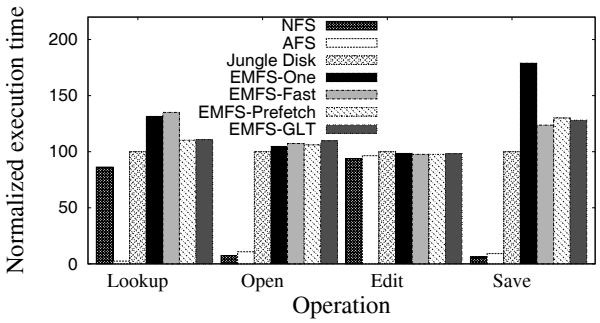


Fig. 11. Editing workload results

VI. Related Work

We are not the first to study email-based file systems. GmailFS [3] implements a file system using Gmail services, although it seems to be under development and no stable version is available for performance comparison. YaFS [4] is an extensible distributed file system using heterogeneous online storage services (including email) as back-ends. In addition, free email accounts have been explored for data backup [5], where the authors also discussed the possibility of mirroring data across multiple email accounts. However, to our best knowledge, our work is the first that systematically examines email-based file system design issues, and thoroughly evaluates the effectiveness of features such as multi-account space aggregation, file striping, and data replication.

Several other existing client-server systems are similar to EMFS in the sense that they only require client-side installation, which accesses servers via standard network protocols such as FTP and SFTP. For example, LftpFS [19] is a read-only network file system with caching for smart mirror sites. ExpanDrive [20] is a network file system that maps a local volume to an SFTP server. The major difference is that our work aims to build a general-purpose file system capable of partial file accesses. Also, EMFS focuses on email-specific design issues and enables users to take advantage of widely available and increasingly powerful web-based email services.

Distributed file system has been an active research area for many years. Besides widely deployed systems such as NFS [21] and AFS [22], other distributed file systems provide many building blocks and techniques that EMFS can leverage. Examples include consistency and replication mechanisms (Coda [23], Dynamo [24], and TierStore [25]), communication bandwidth preserving by exploiting cross-file similarities (LBFS [26]), large data blocks and aggressive replication (GFS [27]), and hash-based data distribution for scalable metadata management (Ceph [28]). EMFS, in turn, complements existing studies on distributed file/storage systems by exploring the feasibility of supporting file operations on top of free, pervasive, yet rather opaque email services.

Certain aspects of EMFS design are similar to existing file systems, such as striping (universal in parallel file systems [29], [30], [31], as well as distributed file systems [27]). Similarly, EMFS leverages metadata consistency and failure recovery methods from log-structured file systems [11]. However, our exploration and discussion focus on examining the application and impact of such mature techniques in email environments, especially in exploiting their novel uses in addressing unique problems such as the usage limit enforced by mail service providers.

VII. Conclusion

We presented EMFS, a personal cloud storage solution on top of multiple web-based free email accounts. By viewing email accounts as virtual disks and applying techniques such as striping and replication, EMFS is able to provide cost-effective, efficient, and highly available storage, by leveraging the cloud infrastructure of leading web-based email service providers. Though email protocols are not designed for file transfer, we have found that EMFS achieves a significant fraction of NFS/AFS performance (with the latter running on dedicated servers within local networks) and approximately matches or outperforms Jungle Disk, a non-free commercial cloud storage service.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback and suggestions. This work is partially supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), by the NSF under grants CNS-0716210, CNS-0747247, CNS-0546301 (CAREER) and CNS-0915861, an IBM faculty award, a joint faculty appointment between NC State University and Oak Ridge National Laboratory, a senior visiting scholarship from Tsinghua University, and by K. C. Wong Education Foundation. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government and funding agencies.

References

[1] "Jungle disk," *http://www.jungledisk.com/*.
[2] "Dropbox," *http://www.dropbox.com/*.
[3] "Gmailfs," *http://sr71.net/projects/gmailfs/*.
[4] Y. Lu, H. Mao, and J. Shen, "A distributed filesystem framework for transparent accessing heterogeneous storage services," in *IPDPS '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.
[5] A. Traeger, N. Joukov, J. Sipek, and E. Zadok, "Using free web storage for data backup," in *StorageSS '06: Proceedings of the second ACM workshop on Storage security and survivability*, New York, NY, USA, 2006.
[6] M. Szeredi, "File system in user space," *http://fuse.sourceforge.net/*, 2006.
[7] M. Zhou and A. J. Smith, "Analysis of personal computer workloads," in *MASCOTS '99*. Washington, DC, USA: IEEE Computer Society, 1999, p. 208.
[8] "Gawab," *http://www.gawab.com/*.
[9] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 4–4.
[10] P. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," http://www.ietf.org/rfc/rfc4122.txt, 2005.
[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.
[12] L. B. Soares, O. Y. Krieger, and D. Da Silva, "Meta-data snapshotting: a simple mechanism for file system consistency," in *SNAPI '03: Proceedings of the international workshop on Storage network architecture and parallel I/Os*. New York, NY, USA: ACM, 2003, pp. 41–52.
[13] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *Trans. Storage*, vol. 3, no. 3, p. 9, 2007.
[14] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, 1988.
[15] "Google outages damage cloud credibility," *http://en.wikipedia.org/wiki/Gmail*, 2009.
[16] J. Katcher, "Postmark: A new file system benchmark," *Technical report TR3022. Network Appliance*, October 1997.
[17] "Iozone filesystem benchmark," http://www.iozone.org.
[18] "Linux battery life toolkit," *http://www.lesswatts.org/projects/bltk/*.
[19] "Lftpfs," *http://lftpfs.sourceforge.net/*.
[20] "Expandrive," *http://en.wikipedia.org/wiki/ExpanDrive*.
[21] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "Nfs version 3 - design and implementation," in *In Proceedings of the Summer USENIX Conference*, 1994, pp. 137–152.
[22] 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," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 51–81, 1988.
[23] M. Satyanarayanan, J. Kistler, and E. Siegel, "Coda: A resilient distributed file system," in *IEEE Workshop on Workstation Operating Systems*, 1987.
[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.
[25] M. Demmer, B. Du, and E. Brewer, "Tierstore: a distributed filesystem for challenged networks in developing regions," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.
[26] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *SOSP '01*. New York, NY, USA: ACM, 2001, pp. 174–187.
[27] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.
[28] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *OSDI '06*. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.
[29] F. Schmuck and R. Haskin, "GPFS: a shared-disk file system for large computing clusters," in *Proceedings of the First Conference on File and Storage Technologies*, 2002.
[30] J. Hartman and J. Ousterhout, "The Zebra striped network file system," in *Proceedings of the 14th Symposium on Operating Systems Principles*, 1993.
[31] P. Carns, W. L. III, R. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.