ABSTRACT

SRINIVASAN, JAGANNATH. EMFS: An Email Based Distributed File System. (Under the direction of Xiaosong Ma.)

The pervasiveness of the Internet has fueled the massive growth of online cloud services. They are extremely varied, ranging from offering software as a service to online backup services to high performance computing solutions. Cloud services are very user friendly, extremely reliable and stable. Their most useful feature is accessibility - they allow access to the service through the Internet from any machine at any location. Recently, cloud storage services for personal data management and sharing have become very popular. However, existing cloud services like Google Docs, Microsoft Office Live, and Jungle Disk are not designed for general-purpose file accesses: they either tightly couple application services with storage, or are commercial applications which charge a fee to utilize their storage services.

In this thesis work, we explore the viability of using online email services to create a general purpose distributed filesystem and address an array of unique challenges associated with this approach. Performance of such a system depends on various factors, such as metadata management, use of specific protocols, as well as the format of storing the actual file data. SPAM control mechanisms of various email providers also play an important role. We implement a prototype system using the Filesystem in UserSpacE (FUSE) Python API and analyse the system using various synthetic and real-world workloads. Our results show that for individual file transfers, our prototype performs at about 90-95% of the level of performance of commercial storage solutions and at 60-70% of the level of performance of existing distributed file systems.

EMFS: An Email Based Distributed File System

by
Jagannath Srinivasan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Ting Yu	Vincent Freeh

Xiaosong Ma
Chair of Advisory Committee

DEDICATION

To my parents - *Amma* and *Appa* ...

BIOGRAPHY

Jagannath Srinivasan was born in Bhubaneshwar, India. He graduated from the Birla Institute of Technology and Science (BITS), Pilani, India with a Bachelor's degree in Computer Science in June 2008. He joined the North Carolina State University in Fall 2008 to pursue his masters in Computer Science. Jagan plans to graduate in December 2010, upon the completion of his research work.

ACKNOWLEDGEMENTS

First of all, I would like to offer special thanks to my advisor, Dr. Xiaosong Ma, for her insight, encouragement and patience in guiding me through this work as well as through my graduate studies. I also wish to thank Dr. Ting Yu and Wei Wei with whom I have collaborated closely for this work. I would also like to thank Dr. Vincent Freeh, for serving as a member of my thesis committee.

I am also grateful to the members of my research group - Amey Deshpande, Karthik Vijayakumar, Divya Dinakar, Feng Ji, Zhe Zhang and others for their cooperation and valuable inputs.

I am especially indebted to my friends, here at NCSU and elsewhere - Arjun Prasad Dasarakothapalli, Alphonse Hansel Anthony, Sandheep Kumar, Kasturirangan Parthasarathy, Arun Rajasekaran, Jayakumar Nanjappan, Pradeep Dorairaj, Sankarnarayanan Ravi, Adithya Raghavan, Arjun Ramakrishnan and Vikram Manoharan. Thanks for all the wonderful memories.

Finally, I wish to thank my parents and my sister, who have always encouraged, and guided me throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

The pervasiveness of the Internet has ensured the massive growth of cloud services. Cloud services are popular because of their scalability and reliability. In addition, they are very user friendly because they are accessible from any machine over the Internet. The most popular and widely used services are hosted online with little or no information being stored on the user's local computer. Users are provided with a wide range of cloud services. Gmail, MSN Hotmail and other email services, Google Docs, Facebook etc ... offer software as a service to users. Amazon S3 offers storage as a service and allows users to store and retrieve data through simple interfaces. Google AppEngine offers an infrastructure service which can be used to host our own applications on top of the Google cloud infrastructure.

There is a growing demand from users to easily access and edit their data from anywhere in the world. To this end, online storage and backup services like Drop Box, Jungle Disk and Microsoft Sky Drive are becoming extremely popular. Services like Google Docs and Microsoft's Office Live, provide a way to work with as well as upload documents online. Picasa Web and Flickr offer image storage solutions. However each of the above mentioned services have their own quirks and drawbacks. Some documents edited using desktop word processing software and uploaded to Google Docs often lose their formatting and are incompatible with other services. Image and video storage services like Picasa Web, although efficient at storing multimedia files, are not useful for storing other types of documents. Services like Jungle Disk and Drop Box offer limited free storage beyond which the user has to pay a monthly fee. However, the use of emails as a means of storing data has not been explored fully to date. In this thesis, we examine the viability of using email as a storage medium for personal users.

Table 1.1: Email Service Provider Statistics

Provider	**Storage**	**Message Size**	**Service Outages**
AOL Mail	Unlimited	25 MB Attachment, Body limit at least 32 MB	2 outages in 1997-1998, One outage in 2006
Gmail	7.4 GB	25 MB Attachment, Body limit at least 32 MB	At least 3 outages (failures) in 2009. At least one outage in 2008. No downtime for maintenance.
Hotmail	5 GB	10 MB Attachment, Body limit at least 32 MB	1 outage in Feb 2005, 1 outage in 2007, 1 outage in Feb 2010
Yahoo! Mail	Unlimited	25 MB, Attachment, Body limit at least 32 MB	At least 2 outages (failures) in 2008. 2 instances of downtime for maintenance in 2009.
Gawab	10 GB	50 MB Attachment, Body limit at least 32 MB	Outage stats unknown. However, maintenance work causes downtime.

1.2 Research Overview

1.2.1 Problem Definition

Online email services are highly available and reliable with a near zero percent chance of loss of data. In addition, these services offer a very large storage quota. For example, Gmail currently offers 7.4 GB of free space, Hotmail offers 5 GB and Yahoo Mail offers unlimited storage. Table 1.1 lists storage quotas, message size limits and server outage statistics of popular email service providers.

Most users of these services utilize only a very small percentage of the available storage leaving the rest unused. In addition, the log-structured file system nature of email storage would offer several advantages - optimized writes, versioning of files and simpler metadata management, to name a few. Hence there is a lot of potential in email services that can be exploited to create a highly reliable distributed file system. Despite their clear potential, emails do not lend themselves readily to storage of data files like traditional disk storage. They pose a completely different set of challenges and questions. For example, any file system based on web email services will not have any server side processing capability, thus leaving it completely up to the client machine to handle processing of data and metadata. Furthermore, activities like replication, garbage collection and ensuring consistency become more complicated since they have to be handled client side. Additionally, since this file system will be limited by the bandwidth of the client as well as the bottleneck of a single server, we design a RAID-like back end structure, increasing performance by fully exploiting client side bandwidth and preventing server side bottlenecks. In this work, we provide a comprehensive discussion on the challenges of such an email based file system, and conduct a comparison of our prototype file system with

existing distributed file systems using several suitable benchmarks.

In order to evaluate this file system, we develop a prototype using the Python API of Filesystem in User SpacE (FUSE). FUSE offers a simple API that allows a user to develop file systems that run in the user space. Python was chosen for its simplicity and flexibility in developing applications. A more detailed explanation of FUSE is discussed in chapter 2.

1.2.2 Summary of Contributions

We established the potential of email services in earlier sections and its possible suitability as a suitable cloud storage platform for personal users. This thesis makes the following contributions:

1. The author proposed a design for a distributed file system called EMFS with a RAID-like back end storage based on web email services.

2. The author implemented a complete, configurable nearly POSIX compliant file system with storage over web email services.

3. The author carried out several tests using synthetic and real-world workloads and compared EMFS to existing distributed file systems.

We have performed this work in collaboration with Wei Wei, a PhD student and Dr. Ting Yu, from North Carolina State University, Listed above are the individual contributions of the author, with inputs and guidelines from the collaborators about design and workload choices.

Chapter 2

Background

2.1 Filesystem in user space (FUSE)

2.1.1 Architecture

FUSE is a kernel module which lets developers create filesystems without editing kernel code. These filesystems run in user-space over the FUSE module which provides a link over the actual kernel filesystem interfaces. FUSE consists of a loadable kernel module along with a library API. It provides multithreaded operation along with an extremely efficiefnt user space-kernel interface and also allows non-privileged users to mount and use their filesystem. FUSE also allows the filesystem to be configured by passing different parameters to the kernel module and library, such as turning off multithreaded operation or enabling large writes. Since the FUSE library is in user space while the module is loaded in the kernel, communication between the two is achieved using a file descriptor which is obtained when /dev/fuse is opened. Any file system call issued from inside the user space file system is caught by the FUSE module and redirected back to user space to the FUSE library and the custom file system handler code. Because of this, FUSE introduces a small delay, which lowers performance slightly. The return value from this handler is again redirected by FUSE back to the user who issued the call. Figure 2.1 shows the architecture of FUSE along with the interactions that occur when a file system call is made. FUSE is particularly useful for developing virtual file systems, where resources not traditionally used for data storage, are exploited for that purprose. This virtual file system then acts as a translation of the actual medium of storage, providing users with a traditional file system view.

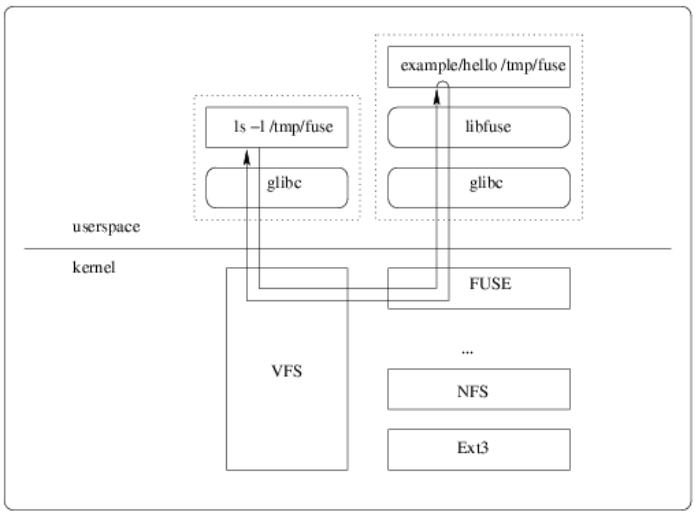


Figure 2.1: Typical path of a file system call of a user space file system running over FUSE

2.1.2 Interface

FUSE provides a comprehensive API that can be used to develop fully POSIX compliant file systems. The API has also been to ported to several languages for ease of development. EMFS has been implemented in the Python port of the FUSE API. The FUSE API provides interfaces for standard file system calls, such as *stat, open, read, write, readdir, create, mknod, rm, rmdir, fsync, flush*, and *release*.

2.2 Email Access Protocols

Most online email service providers provide provide access to their services through three major protocols namely, Internet Message Access Protocol (IMAP) [7], Simple Message Transfer Protocol (SMTP) [23] and Post Office Protocol (POP3) [30]. These three protocols are discussed in detail below.

2.2.1 Simple Mail Transfer Protocol

SMTP is a message transfer protocol and cannot be used to retrieve messages from the server. Messages from any account on any domain or service provider can be sent to any other account anywhere in the world. Messages sent using SMTP are usually transferred to the recipient's inbox. Users can modify this behaviour however in order to divert mails from certain domains to specific folders in their mailbox. SMTP is easy, simple to use and extremely reliable. However

there is one major drawback associated with the protocol. Almost all online email service providers have set up several control techniques in order to restrict SPAM. Many of these control techniques deal with restricting the heavy use of SMTP as messages can be sent between two accounts using only this protocol. Table 2.1 lists some of the common spam control mechanisms employed by major service providers. SMTP is supported by all email service providers as it is the standard way to transfer messages between two mail accounts.

2.2.2 Post Office Protocol

POP is a message access protocol. It can be used to retrieve messages from mailboxes but not send messages to other accounts. POP offers simple "download and delete" semantics, where messages once downloaded from the email servers, are usually deleted on the servers leaving the copy on the client as the only copy of the message. However, most modern email access client provide a method whereby the user can leave a copy of the messages on the server.

2.2.3 Internet Message Access Protocol

IMAP is another message access protocol like POP. IMAP is a much more complete protocol and offers more complex functionality than POP for remote message retrieval. It has several useful features. It allows separate access to individual parts of a message, like the headers, the subject, the body and the attachments. This allows users and email clients to manipulate these parts separately allowing for more efficient processing. Unlike POP, for message access IMAP requests only the ID of those messages that have arrived later than the message that was retrieved last. POP has to retrieve the IDs of all the messages in the inbox, which may cause problems for large mailboxes. Most importantly, IMAP allows users to "append" a message to their mailbox, like saving a draft email. This feature cannot be used to send emails to other mail accounts. Hence this feature is not controlled by anti-spam techniques like SMTP. Because of these complex and useful features, EMFS uses IMAP to both retrieve and upload (append) messages.

2.3 Related Work

The idea of using email as a storage medium is not new. Besides GmailFS, discussed in earlier sections, YaFS [26] is an extensible distributed file system which uses various heterogeneous online services (including email) as storage mediums. The possibility of using emails for data backup as well as using several accounts to mirror data has been discussed in [46]. However, to the best of our knowledge, this thesis is the first work to systematically explore the effects of various factors affecting the design and performance of email based file systems.

Table 2.1: Spam control mechanisms of different email service providers

Service Provider	Control Mechanisms	Details
Gmail	Daily limit on the number of messages sent, limit on the number of recipients per email	500 emails allowed to be sent daily, Maximum 500 recipients per email (from the web interface) and 100 per email (from client applications)
AOL	Challenge/Response system	CAPTCHA verification if too many emails are sent out quickly
Yahoo Mail	SPAM Guard	Complex application which uses a variety of techniques
Gawab Mail	Server disconnection, Tarpetting	Server disconnects user if too many emails are sent quickly. Tarpetting: Time delay introduced after every transaction.

Online cloud storage and data processing services like Google Docs [17] and Microsoft Live Office [32] are closely related areas. However, these services tightly bind data storage with application services, often causing incompatibility problems. Online general purpose storage services include DropBox [11], Jungle Disk [21], Microsoft SkyDrive [41] and Zumo Drive [51]. These services generally allow users to map a local volume to online storage disks for easy access. Our research indicates that it is possible to utilise free email services to obtain similar storage functionality and performance. The difference lies in the fact that EMFS targets at building a general purpose file system while focusing on email specific issues and taking advantage of increasingly powerful web email services.

Active research has been conducted in the area of distributed file systems for several years. Besides widely deployed systems such as NFS [34] and AFS [19], EMFS derives some of its ideas from other distributed file systems. Examples include consistency and replication mechanisms (Bayou [45], Coda [39], Dynamo [9], and TierStore [10], LogFS [38]), preserving communication bandwidth by exploiting similarities between versions of the same file (LBFS [29]), large data blocks and aggressive replication (GFS [14]), hash-based data distribution for scalable metadata management (Ceph [49]) and maximizing client side bandwidth utilization by striping data in parallel to several servers (Zebra FS [18]). EMFS has several similarities with these file systems. However, the main focus of this work is to examine the impact of mature techniques such as these in an environment like email services. These techniques can be used to solve some novel problems posed by email services such as bandwidth caps, email usage limits, and spam controls.

Chapter 3

EMFS Overview

3.1 Overview

3.1.1 Target Workload and Design Goals

EMFS provides an email-based cloud storage for personal users. Each user configures the system such that he has his own personal cloud storage space. It is assumed that users do not share this space with others. EMFS focuses on dealing with personal workloads of typical users [50], including reading, editing, and backing up documents such as word, powerpoint, excel, pdf, etc. Such workloads may have hundreds of files and directories, but the file size of most of these files may range only from several kilobytes to tens of megabytes [2]. Our design goals for personal cloud storage are summarized as follows:

1. **Usability** We aim at not only providing an easy-to-access interface for personal users to access the data stored on their personal cloud, but also enabling them to use existing applications to edit files on top of this storage.

2. **Scalability** We should be able to increase the capacity of personal cloud storage when needed, and efficiently handle the increasing amount of files as time goes by.

3. **Reliability** As with other cloud systems, EMFS should be highly reliable and available. In particular, EMFS should be able to work even if some email services are out of service.

We use existing email services and techniques to achieve the above goals.

3.1.2 EMFS Software Layers

Figure 3.1 illustrates the EMFS architecture, composed of two layers: the EMFS client and the email storage cloud. This email storage cloud provides storage resources by utilising web-based

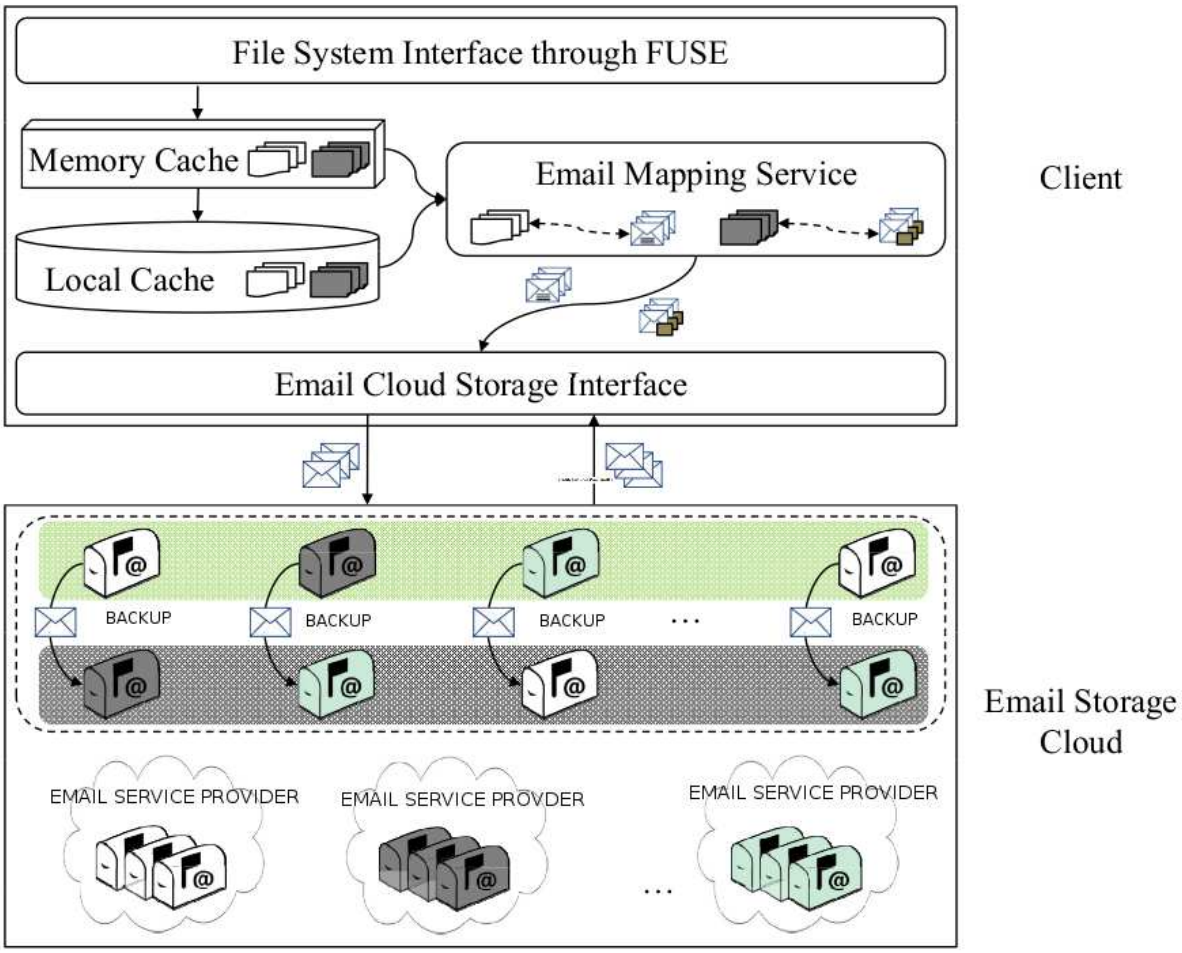


Figure 3.1: EMFS Design Overview

email accounts, from providers such as Google, Gawab, and AOL. The EMFS client is split into several components; the FUSE file system interface, the local disk and metadata caches, the email mapping service and the email cloud storage interface. The email mapping service manages distributed storage, treating each email account as a virtual disk. Data is stored as attachments as part of emails. The email mapping service also carries out optional data placement strategies, such as striping and replication, across multiple email accounts. The client also presents an approximate-POSIX file system interface to the user, which enables existing applications to run on top of EMFS without any modifications.

The EMFS package, comprising of the client layer, is implemented in around 3000 lines of Python code and runs on the users' client machine. Note that the EMFS package does not have any server side code. It uses existing email services and their interfaces to store, search for and retrieve data. Thus, since EMFS has a "dumb server", it is not able to make use of server side optimizations to provide extra features and improve performance. For example, with a smart server, a system will be able to easily handle consistency issues, provide concurrent access to data and possible conflict detection and resolution. The system will also be able to do background replication and garbage collection, processes which take up client side bandwidth without such a server.

Email Services EMFS ultimately stores its data in emails provided by service providers such as Google, Gawab and AOL. These email services usually run their own massive datacenters to store their customers' data. These services also provide high reliability by replicating data with their datacenters to guard against data loss. These datacenters also have fail safe mechanisms in order to provide high availability and guard against network connectivity problems and crashes. EMFS takes the full advantage of these potential benefits provided by existing email services to build a more reliable system.

Virtual Cloud Storage The virtual cloud storage (VCS) is a logical component in the EMFS system. The VCS presents to the user, virtual "disks" organized like a RAID [33] to store all data and metadata in the system. VCS does not place any restrictions on where data and metadata should be stored as they are treated the same way. Each virtual disk is actually an email account, created online with a web email service provider. The VCS exposes a thin interface which utilises email protocols to store and retrieve emails from these virtual disks. As mentioned before, the VCS organizes these virtual disks in RAID fashion. Multiple email accounts, not necessarily from the same service provider, are indexed in an array to simulate a RAID. Data is striped into different disks to improve performance.

Since the capacity of each email account is usually limited, the exact size varying according to different service providers, the VCS allows users to dynamically increase the size of their

storage, by adding more email accounts to the system. However, availability can still be an issue, as email servers may break down. Downtime due to maintenance also causes availability issues. Although this may not be a frequent occurrence, it can potentially render certain email accounts inaccessible for some time. For example Gmail had a few outages in 2009 [16], leading to users being unable to access their accounts for a few hours each time. Other service providers have also experienced some failures, as indicated in table 1.1.

To address this issue, EMFS is designed to remain available even if some email accounts are temporarily out of reach. This is achieved through the replication mechanism of EMFS. With replication enabled, each virtual disk may replicate data to other backup disks. The number of replicas can be configured to suit the user's needs. As shown in figure 1, each disk replicates the data it stores to other disks, with former disk being referred to as a *primary* and the latter disks as *backup disks*. The primary and backup disks together are referred to as a replication group. These disks are configured by the user. The primary servers are accessed for all uncached data and metadata requests unless there are availability issues. Whenever EMFS detects that a particular primary disk is unavailable, it immediately switches operations to a backup disk from the same replication group. It is advisable that email accounts in a replication group are hosted on different service providers, as this would ensure that if one service provider is unavailable, a backup email account which is hosted on a different service provider can be utilized instead.

Client As shown in Figure 3.1, the client layer consists of the File System Interface (FSI), the memory cache, the local disk cache, the email mapping service and the email cloud storage interface. The FSI provides an approximate-POSIX interface via FUSE [44]. We say the EMFS interface is approximate-POSIX because it selectively relaxes certain consistency semantics in order to optimize performance. For example, EMFS does not maintain the last accessed time. This time is set to the time of creation and is not changed on further accesses. This is to optimize performance by reducing the number of metadata updates that need to be sent. Otherwise, EMFS would need to send a metadata update for every file close, updating the last accessed time.

In EMFS, all file system calls invoked go to the FSI through the FUSE module. FSI communicates with a specific client component or coordinate with different components to fulfil a specific request. The metadata cache caches metadata on access, and also files which are currently open. The local disk cache caches files which have been accessed so far. The disk cache is emptied periodically. The default interval for emptying the cache is set to a large value to better serve read requests. The email mapping service keeps track of the file to virtual disk mapping, replication groups, as well as other email related configuration parameters. Whenever, a data request, such as *read* is made, the FSI interacts with the memory and disk caches to

see if the required metadata and file data can be retrieved from the local caches. In case the cache are unable to fulfil the request, they forward the request to the email messaging service. The email messaging service utilises the email cloud storage interface to retrieve data from the underlying email storage through a lean interface.

Single User Design As mentioned before, 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 partly 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 utilising just a client without severely hampering performance. These assumptions greatly simplify the design of EMFS, allowing us to effectively handle and optimize other more important issues.

3.2 EMFS Structure

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 discussed in chapter 2.2

3.2.1 Email Transfer and Retrieval Protocols

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 transfer of emails to the server while IMAP and POP are primarily used for retrieving emails. Each protocol has its own advantages and disadvantages which have been discussed in chapter 2. As mentioned before, EMFS uses the IMAP protocol to both send messages (using the append feature) and retrieve emails.

3.2.2 Metadata

Metadata management is one of the most important considerations of any filesystem. Since metadata operations make up as much as half of typical file system workloads [37], effective metadata management is critical to overall system performance. EMFS is designed with a simple but effective metadata structure which takes into consideration the need for scalability and reliability.

Data can be stored in many places in an email - the headers, the subject line, the email body and as attachments. The first 3 locations are ideal for storing metadata because we can then use IMAP to search these fields directly without having to download the email. In EMFS, metadata is stored in the body section of emails. Subject lines of these emails contain unique a set of unique identifiers, that can be used to search for and directly retrieve a single metadata email if needed. Metadata emails are created for directories only and no separate metadata email for files exist. Instead the metadata for files are stored in their parent directory's metadata email. Directory metadata emails contain the metadata for all subdirectories and files under the directory. Figure 3.2 illustrates what is contained in a typical metadata email.

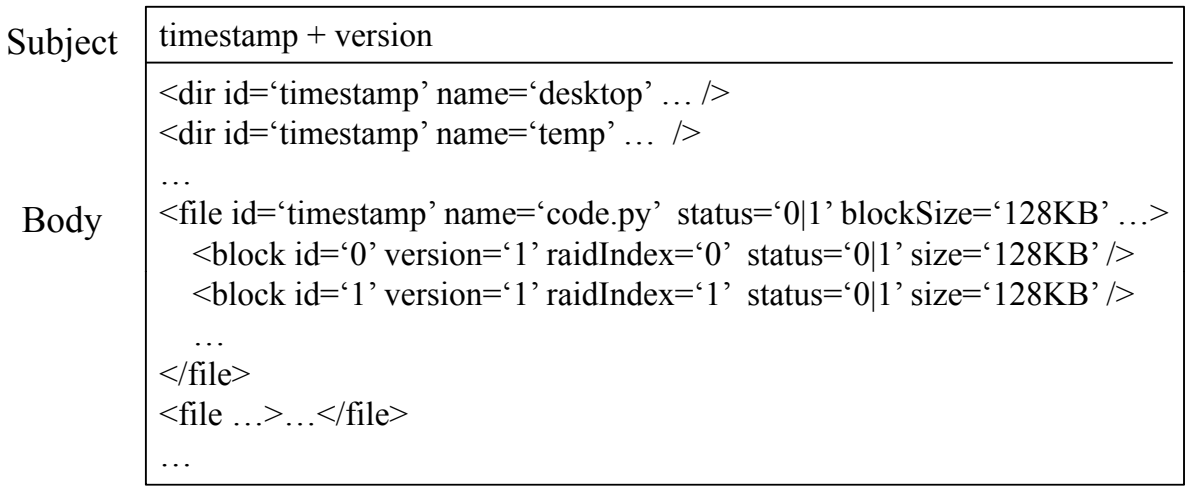


Figure 3.2: The Structure of Metadata Email

Metadata is stored in EMFS in XML format. The subject line lists the id of the directory under question along with its version number. As email storage is similar to a log fs, where data is not overwritten, the version number is used to identify the latest version of the metadata. The directory id is a unique id in the file system, which can be used to differentiate this directory from other directories and files. It is this id, along with the version number, which is used, to search for and retrieve the metadata email of a directory. The body of the email contains the metadata entries for all of the subdirectories and files under this directory. There are three types of entries, which are explained below.

- **<dir> entry** represents a directory in the file system. The directory entry contains attributes for a directory such as the unique id, name of the directory, and permissions.

- **<file> entry** represents a file in the file system. The file entry is more complex than a <dir> entry as a file naturally has more metadata attributes than a directory. The most important attributes listed in a file entry are id, name, status, block size, and file size. Each file entry will also contain several <block> entries, which are described below.
- **<block> entry** represents a single block in a file. These entries are used to identify and describe each block in the file. Each block entry describe a file block using several attributes - block index, block version number, RAID index, status, data size and so on. These entries are always associated a file since each block has to be a part of a file.

Please note that each metadata email contains the metadata for a directory's *children* and *not for that directory itself.* The metadata for that directory is stored in its parent directory's metadata email. The root directory of the file system has no metadata associated with itself and is only associated with a unique id which is known internally to the filesystem. Hence the metadata email describing the root's children can always be accessed using this id without any other lookups. The root directory's metadata email is fetched and cached at system start up. As mentioned before, the metadata email for a directory is searched using its unique id and its version number, both of which are found in its parent directory's metadata email. Thus, before the metadata email for a directory can be found, the email for its parent directory needs to be retrieved in order for us to find the id and version number of the target directory. A lookup for any directory's contents is performed in a series of steps. First the ids and version numbers of each directory along the requested path need to retrieved one by one, starting with the directory directly under the root. This information is then used to retrieve that directory's metadata email in order to determine the id and version number of the next directory along the path. Eventually, the id of the target directory is looked up from its parent directory's metadata email, following which, its metadata email is recovered and the contents listed. For example, if the path '/dir1/dir2/.../dirn' is requested, then the metadata emails for dir1 is first retrieved, using which the id for dir2 is identified. dir2 is then retrieved and the same process is followed till the email for dirn is retrieved. The same process is followed even if a file inside dirn is requested, except that at the end the file is retrieved. It should be noted that metadata is cached locally the first time it is retrieved, hence making subsequent requests much faster.

The metadata organization in EMFS is similar to that of the Linux filesystem to some extent. Like the separation of Inode numbers and names in Linux, EMFS separates ID numbers and file and directory names. This allows for the modification of file and directory names without the need to change the id number. This allows for more efficient metadata updates, since for these types of changes, only one metadata email (the parent directory's) needs to be updated. However, one major difference is that in Linux, apart from the file name and inode number, all other metadata describing a file is stored in a separate inode. In EMFS however, for performance reasons, we do not make this separation and instead store all metadata for files in the parent

directory's metadata email. Because of this choice, we need only to retrieve metadata emails of all the directories along the requested path to list the contents of the target directory, regardless of the number of files and directories it contains. If instead, we had separate metadata emails for each file and directory, we would need to retrieve several emails - one for each child in the target directory in addition to the emails of the parent directories along the path, thereby incurring a performance penalty and generally slowing down the system. Another advantage of this metadata structure is that the metadata for files and directories is stored across multiple emails, making it easy to handle hundreds of files and directories in the file system without considering the size limit of a single email. If directories grow to be really large, containing thousands of children, this design can also be modified the single metadata email for that directory into two or more emails for scalability and performance.

One possible disadvantage with this design is that any change to the metadata of a file or directory implies that the entire metadata email of its parent directory needs to be resent to the email server. However, because each metadata entry is around 70 bytes or less, and the number of files in each directory is not going to be very large for personal users [2], the size of each metadata email is not going to be large enough to slow down the performance of the system.

We feel this metadata stucture (where metadata for files and directories are not closely maintained together and not completely separated) is very scalable, easily extensible and efficient. With a global metadata design, where the metadata about all files and directories are stored in a single file, updates to metadata become very inefficient. The global metadata file could be cached on system startup for better performance. However, any updates to metadata would mean that the entire file would have to be transferred to the server. This would prove extremely inefficient for a system with a large number of files and directories.

3.2.3 File Data

File data organization in EMFS is simple and straightforward. Any file is treated as a byte stream and divided into chunks of size *blocksize*. Blocksize is a tunable parameter and can be adjusted to improve performance based on several factors such as the number of email accounts used, network bandwidth and network traffic. Each block of the file is sent with an email as an attachment. Online service providers limit the size of attachments allowed per email, but this value is very high (25 MB for Gmail and 50 MB for Gawab), and is certainly large enough for personal user workloads. As we discussed in section 3.2.2, there is a block metadata entry for each block of a file in a metadata email. The block id and the block version number 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. The block

identifer for every block is always present in the metadata email of the parent directory of the file. This email must be retrieved first in order to identify and read blocks of a file. For fast and efficient retrieval of the blocks, the subject line of each email containing a file block, contains the block identifier of that block. Thus, each time when a block is needed, the system directly retrieves the email containing the subject with the matching block identifier.

Each file metadata entry is also associated with a version number and status. These attributes are utilized to maintain consistency and facilitate failure recovery which is described in section 3.3.1. In addition, version numbers also facilitate versioning which is described in 3.2.4. Each block entry is associated with a RAID index which is described in 3.4.2.

3.2.4 Versioning

It is obvious that email storage is somewhat like a log structured file system, where all updates (data and metadata), are appended instead of overwritten. This characteristic of email storage can be used to create a versioning file system. Although an email based versioning file system would be straight forward to implement, it is currently not supported by EMFS. The metadata structure of EMFS already maintains version numbers for files, using which it would be possible to recover earlier versions of files. In addition, earlier versions of metadata, which would still be available on the email storage servers, can be used as a "snapshot" of the filesystem at various times of its lifetime. This metadata snapshotting would be useful, among other things to address consistency issues [42] when failures occur. By utilising the strong search features of IMAP, it would be easy to recover earlier versions of files and metadata at no additional cost.

3.3 Operations

3.3.1 Metadata and Data Access

Metadata Operations

Metadata Read Metadata needs to be accessed for several operations (e.g., *readdir, stat, open* and many more) Metadata access is efficiently implemented in EMFS using our metadata design. Any request for metadata is first referred to the metadata cache before the request is forwarded to the email servers. Any request for a directory's metadata, like *readdir*, will succeed when the directory's email has been retrieved either from the cache or the email server. If the requested metadata is not cached, several requests to the email server may need to be made, as described in the previous section. Any subsequent requests for *stat* or *open* on the children of this directory will be very fast as the required information would already have been cached.

Metadata Update Any updates to metadata update caused by operations such as *rename, rmdir, chmod, mkdir* are immediately synchronously committed to the server. To improve the performance for metadata update operations, the metadata structure is designed to minimize the number of metadata emails that need to be committed to the server. For example, *mkdir* creates a new directory under the current directory. To fulfil this operation, 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, no metadata email for that directory needs to be sent out immediately. This step is delayed until a file or directory is created under this directory. In this way, only one email update needs to be committed to the server. A similar strategy is followed for newly created files as their size is 0. The *rmdir* operation is another example. To delete a directory, we remove the target directory's entry from the metadata email of its parent directory. The version number of the parent directory's email is incremented and sent out to the server. There is no need to delete the metadata email of the target directory and, its files and subdirectories immediately. This can be cleaned up lazily,, during regular garbage collection, which is described in Section 3.4.5. Cleaning up these emails lazily also has an additional advantage, in that it may be useful to restore accidentally deleted files and directories. This feature is not currently implemented however.

Typical Data Access Operations

Any time a file is created in EMFS, a new metadata email for the parent directory is created and sent out. The size of the file is set to 0 on creation, and remains so until data is written to the file. If no data is written to the file, then no data emails are sent out. Opening a file involves fetching the metadata email of the parent directory to check whether the file exists and if so, identify its unique id. Following this, file data is fetched in the background. On subsequent read() calls, the fetched data is returned according to the request. If a read() call requests a data block which hasn't been retrieved yet, a request to retrieve that block is immediately sent out. The process which issued the read() call then needs to wait till the data block is received. No changes, except the file access times are made to the metadata on read() calls.

Whenever data is written to a file, one metadata email reflecting that the file has been changed, and several data emails, one per each data block, are sent out together. Following this, one more metadata email is sent out to confirm that the file transfer took place correctly. The purpose of the 2 metadata emails and the order in which they are sent out, is to ensure consistency. Further details on the consistency model of this system is explained in section 3.3.1. Changes to the metadata include file size, file status, access and modification times, and addition or deletion of block entries. Data is written to the email server only when flush() or fsync() is called. On simple write() calls, data is stored locally until flush() or fsync() is

invoked. This is for performance considerations as transferring data over the network on every write() call is expensive. Other systems like AFS also follow the same semantics for the similar reasons. Table 3.1 details some common file operations and how EMFS handles them.

Since repeatedly accessing email servers for every request will be very costly, EMFS uses a local cache to improve performance. The design and working of this cache is described in the following section.

Table 3.1: Common data and metadata operations in EMFS

Operation	**Number of Emails**	**Details**
Stat (getattr)	n emails retrieved	n is the depth of the file or dir being "stat"ed. One email for every directory along the path is retrieved
Open (open)	0 or k emails retrieved in the background	k emails, one per each data block is transferred in the background. 0 emails retrieved if data is cached
Read (read)	No emails retrieved	Data retrieved through the open call is used to serve requests
Write (write)	No email sent	Data is cached locally until fsync/flush/release is invoked
Flush (flush/fsync/release)	2 metadata emails + k data emails sent	k emails, one per block
Remove (rmdir/rm)	1 metadata email sent	1 metadata email sent to update metadata immediately. Actual data blocks are removed lazily.
Rename/Move (rename)	1 or 2 metadata emails sent	1 metadata email if target is renamed. 2 metadata emails if target is moved to a different directory.
Other operations (truncate, modtime, atime ...)	1 email sent per operation	1 email per operation for all other metadata changes

Client Cache Management

The EMFS client maintains a local disk cache to store file data. The use of a local disk cache is to reduce the number of requests sent to the email server as this tends to be a performance

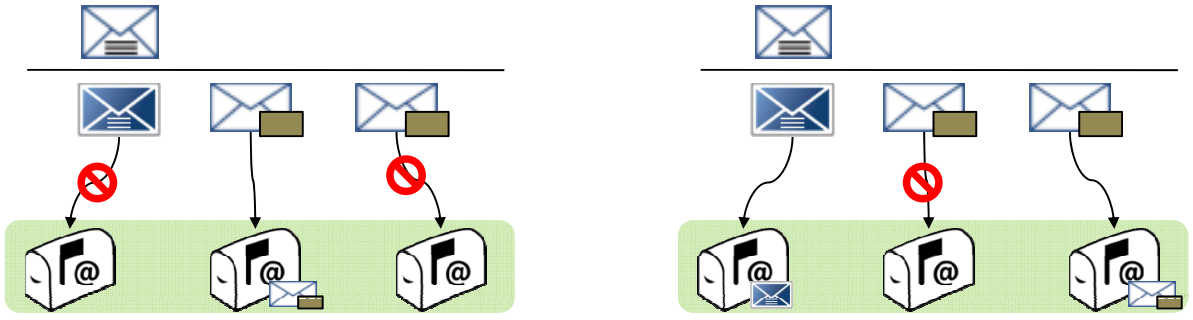
bottleneck. Any time a file is written to the server (whenever a fsync() or flush() is invoked), data is first written to the disk cache and then sent out to the email server. Similarly, whenever a block b of a file f is requested, the local cache is first searched to see if the block is cached. Only if the block is not found in the cache are requests sent out to the email server to retrieve data. On retrieval, the block is cached for future requests. EMFS assumes there is always enough space on disk to cache required files on disk. For most users, this won't be a problem as average files of a typical user are of size 189 KB and 90% of files on a disk are smaller than 1 MB [2]. Data in the disk cache is not deleted even if the system is unmounted. Hence even on subsequent mounts, data reads will be really quick. Additionally, whenever a file is opened, the full file is read into a memory buffer. Data from this memory buffer is used to service any read() calls. On write() calls, any data that is being written is stored in this memory buffer. When flush() or fsync() is invoked, this data is flushed to disk and then to the email servers. This memory buffer is cleared whenever the file is closed.

All metadata pertaining to a file or directory are cached upon first access. This metadata cache is maintained in memory and is deleted on system unmount. Although this system has been designed for single user access, the same user can access it from different computers. Because of this, metadata may become stale causing consistency problems. Hence the metadata cache is flushed periodically to maintain metadata cache consistency. The flushing interval may be configured by the user. Due to this same problem, cached data blocks are always checked against the version number in the metadata, which is ensured to be consistent, to make sure that they are indeed the latest copy of the file being accessed. If they are not, the cached block is deleted and the latest one is retrieved from email. Caching is an optional parameter of EMFS and can be turned off if needed.

Consistency and Failure Recovery

As mentioned earlier, EMFS is design for single user access. But the same user could access the system from different computers, causing cache incoherence. The previous section discussed how to handle this inconsistency. This section addresses the more difficult challenge of file system consistency and how to recover from inconsistent states. Essentially, this section addresses the following question: *How do we ensure that the transfer of data and metadata together is atomic?* To answer the previous question, we need to consider three scenarios where the filesystem may become inconsistent due to a system crash, network problem or other anomaly:

1. Both metadata and data emails are lost.
2. Data is transferred to the email server but metadata is lost.
3. Metadata is transferred to the email server but data is lost.



(a) Lost metadata update

(b) Lost part of data update

Figure 3.3: Two failure cases

EMFS assumes that data loss cannot occur once any data or metadata email has been transferred successfully to the email server. Except in the cases where the client system crashes, EMFS will always receive a confirmation 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. The first two cases listed above are pretty simple to handle. As in traditional systems, the file system cannot do anything to recover whenever metadata is lost. The system simply has to notify the user of the failure and prompt him to try again. It is the third case which is more complex. To protect against this type of failure, EMFS undertakes the following procedure. 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 which clears the status field which was set previously.

Now, if both metadata emails are lost, the system is still in a consistent state regardless of whether the data blocks were transferred successfully. In this case, any data blocks committed to the server are considered redundant and will eventually be cleaned up. If the first metadata email is lost, but the system receives confirmation from the server that the data emails have been received successfully, the successful transfer of the second metadata email will ensure that the system remains in a consistent state. If instead, the first metadata email and the data blocks are transferred successfully but the second metadata email fails, the system will still be in a consistent state. The next time the file is accessed from the server, EMFS will notice the status bit of the file is set and hence will check whether the data blocks for the corresponding version

number are available on the server. If they are, the status bit is cleared and the metadata updated. If the data blocks are not present on the server, the system assumes that the data has been lost, and reverts back to the previous version of the file. Thus file system consistency is always preserved in EMFS.

By default, EMFS makes all transfers to the email servers synchronous and atomic. The user is made to wait until the transfer ends. The transfer may be a success or failure, but in either case the user knows immediately whether the transfer was a success or whether something went wrong. Thus the system provides a "strong" guarantee about system consistency. However, the system is configurable to make all updates take place in the background. In this case, any update will first be made to the local cache following which it will be transferred asynchronously to the server. With this "weak" configuration, the system can only guarantee eventual consistency. It should be noted that the user will experience lesser latency and improved performance, as all transfers take place asynchronously. It is possible to switch from one consistency configuration to another any time the user wishes to do so.

3.4 Optimizations

In this section, we discuss the design and working of the Email Storage Cloud (ESC) in detail. ESC is built on top of multiple email services as shown in Figure 3.1. The ESC provides storage within the "cloud" and provides scalability and reliability through appropriate email storage organization, data striping and replication. Figure 3.4 shows an overview of how the ESC organizes data and metadata emails, and stripes and replicates data between multiple email accounts.

3.4.1 Email-based RAID

The ESC is structured as a RAID with email accounts in place of disks. There are three key concepts in RAID: mirroring, the writing of identical data to more than one disk; striping, the splitting of data across more than one disk; and error correction, where redundant parity data is stored to allow problems to be detected and possibly repaired (known as fault tolerance). Of these, ESC supports both mirroring and striping. Error correction has not been supported because it is assumed that once data has been successfully stored on an email server, it is up to that email server to guarantee data consistency and redundancy. In ESC, each email account is considered as a virtual disk. Multiple email accounts are organized into an array, with each account being identified by an index, referred to as a "RAID Index" from 0 to n. By organizing multiple email accounts in RAID fashion, the system achieves location independency and transparency. Data and metadata are not associated with a specific server or email account. If for some reason, one email account is unavailable, we can simply swap this account with

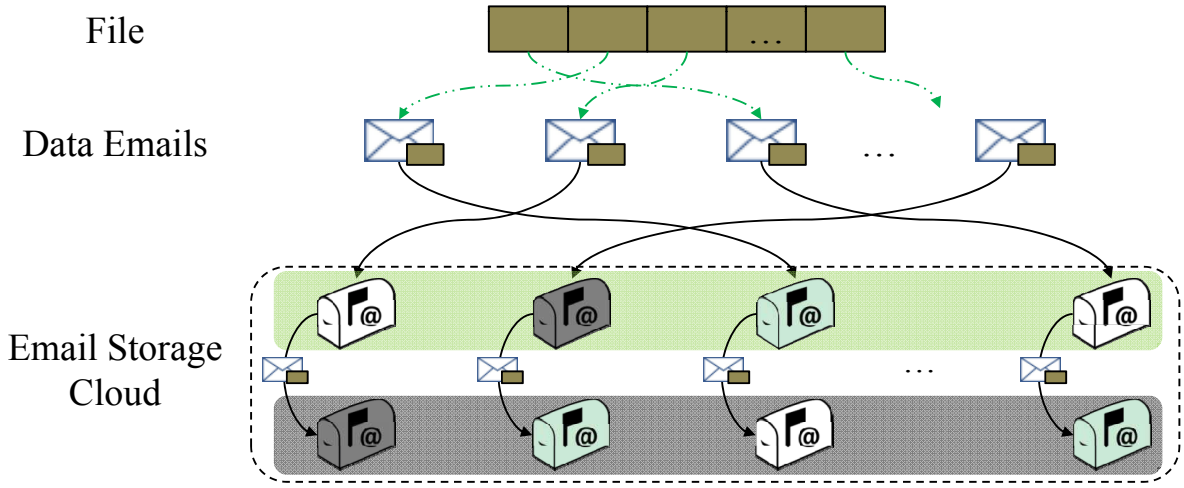


Figure 3.4: Virtual Cloud Storage

another working account. As long as the replacement has all the data that the unavailable account had, everything will continue running smoothly. With email-RAID, instead of storing all the blocks of a file in a single email account, we stripe different blocks into different disks to improve the throughput of the system and achieve high aggregate bandwidth. Metadata is not striped in EMFS and is written to a single disk. Metadata emails are usually small and no additional advantage is gained by striping them. Furthermore, striping metadata complicates their retrieval and may slow down the system. To enhance the reliability and availability, each virtual disk in ESC has backup disks. Each backup disk is an exact replica of the primary disk in our system.

3.4.2 File Striping

Data striping is a general technique to improve the performance by achieving high aggregate bandwidth, which is a key concept in RAID [33]. In EMFS, a file is split into multiple blocks according to a *blocksize* parameter. There are many ways by which data can be striped. A simple way is to store blocks of a file into disks in round-robin fashion, starting from the first disk. With this method, no additional metadata needs to be stored for each block to identify the email account they are stored in. For example, with n virtual disks, where the disk indices range from 0 to $n-1$, a block i can be found on disk $i\%n$. However, the major disadvantage of this method is that the first few disks can quickly be saturated as they are always likely to receive a few blocks to be stored regardless of size of the file. This will lead to unequal data distribution causing a few disks to run out of space sooner than others. A simple modification

to address this issue would be to randomly choose the disk where the first block is to be stored and then continue in round-robin fashion. If the first disk index is x, block i can be found on the disk with index $x + i\%n$. With this method, the only additional piece of metadata we need to store is the position of the first block of the file. However, since "virtual" disks here are email accounts, it is possible that each email account may have a different storage quota depending on the service provider, leading to possible storage problems with the round-robin approach.

EMFS takes on a more flexible approach but incurs the cost of additional metadata. In this scheme, each block can be stored on any disk as long as the disk has enough free space. However, to identify where each block is stored, we have to know the index of the disk that stores the block. The *RAIDIndex* parameter in each <*block*> metadata entry is used for this purpose. Even though any block can be stored on any disk, EMFS tries to maximize client side bandwidth by trying to write to all n servers. For example, with n servers and more than n blocks, EMFS tries to write to all n servers at least once, provided that each server has enough space. This is so that we can fully utilize the available client side bandwidth. It should be noted that once EMFS has been configured for a set of virtual disks, the indices or the order of the disks cannot be changed. Additional disks can be added as needed to increase storage, but disks cannot be removed from the configuration without loss of data (unless a backup disk is inserted in place of the disk that has been removed). A block is assigned a disk to be written to, with its corresponding RAID Index, on creation. Further writes and rewrites to the block always go to the same RAID Index. This is in order to simplify EMFS operations and also to facilitate manual versioning of files. With this design, previous versions of blocks can be found on the same disk without any need to maintain additional metadata.

Although we use the same block size for blocks in a file, we can choose different block sizes for different files according to the file size, the number of available disks, and other factors like network bandwidth. There is no single block size that would be appropriate for different network environments and system workloads. In Chapter 4, the influence of these factors on the *blocksize* and the performance of the system is discussed.

3.4.3 Data Replication

Replication in EMFS is mainly to ensure availability of the system. It can be safely assumed that once data has been stored on an email server, data loss is impossible as online email service providers take a variety of steps to ensure the safety of the data stored on their servers. However, email services do go down from time to time due to a variety of reasons. It is during these periods that data stored on EMFS systems must be available to their users. In EMFS each virtual disk is associated with a *replication group*. A replication group consists of two or more disks, with each disk being an exact replica of the other disks in the group. All disks

in a replication group are assigned the same disk. One disk from the group is considered as the "primary" disk and the rest are called "backup" disks. With a simple configuration, all data writes go to the primary disk. Data from this disk is then replicated to the backup disks using one of two replication methods discussed below. In case of failure or unavailability of the primary disk, it can be replaced by any other disk from the same replication group. Once a replication group has been configured, disks from the group can be removed as long as there is at least one disk to read and write to. Additional disks can be added to the group. Any new data written to this group will be replicated on the new disk. However data written earlier will not be copied to the new disks by EMFS currently. However, it would be easy to design a daemon which keeps track of and completes tasks such as these.

Data in EMFS can be replicated through two different methods: client-side parallel replication (CSPR) and server-side chain replication (SSCR). In CSPR, the responsibility lies with EMFS to replicate explicitly deciding when and where replicas must be stored. With this design, the EMFS client will send copies of the same file to the primary disk as well as its backup disks. The disadvantage of this method is that it increases the client's responsibility, and more importantly, uses up already limited bandwidth on the client side. This type of explicit replication may be done synchronously, so that replicas are created whenever a block is written to the primary disk or asynchronously, with replication taking place in the background to reduce the burden on the client network. Synchronous replication, while being more reliable, reduces performance by taking up more bandwidth. It may also be overkill, considering that we are replicating to ensure availability due to email server down time and not to guard against data loss. Downtime statistics for popular email services are given in table **??**. As shown in the table, email services do not fail frequently, are highly reliable and hence replication can be carried out asynchronously or lazily.

An alternative solution is server side chain replication (SSCR), shown in Figure 3.4. SSCR is extremely simple and replication is done completely by the email servers, placing no extra burden on the client machine. SSCR exploits automatic email forwarding, a well-known functionality provided by most email service providers. By setting up email-forwarding between different email accounts, we can construct a chain to connect different email accounts. When the first email account receives an email, the email server will automatically forward the email to the next email account in the chain. In this way, the update email is propagated to the last email account through the chain, thus creating exact replicas of each other. This approach simplifies the implementation of the client and data replication comes without any additional cost imposed on the current system. However, the disadvantage is that email forwarding works only if messages are sent using the SMTP protocol to the email account's inbox. Messages appended to a folder in the email account using IMAP are not picked for email forwarding by the service provider. However, as discussed in previous sections, messages sent using SMTP are

blocked by SPAM guards, thus making the entire system unreliable.

EMFS adopts lazy replication of data using the IMAP append feature. Replication is done manually by the client and this ensures that the system is reliable and does not suffer from performance loss.

Replication Strategies Rather than naively mirroring each account by copying emails to ensure availability, we can use different strategies to improve performance and availability.

- **Read-one and Write-one** In this strategy, all reads and writes from EMFS go to the same email account, the primary email account. For replication, a set of backup email accounts are configured. Any email sent to the primary email account will be forwarded to the backup email accounts. If the primary email account becomes unavailable, one of the backup email accounts can be used as the new primary email account and accept read and write requests.

  In this case, replication may be achieved using either of the two replication methods discussed above However, it is possible with SSCR or with asynchronous CSR that some emails have not been replicated to all backup accounts. In this case when the primary account goes down, the backup accounts may have to be synchronized with the primary account to ensure consistency. Thus, a "replication pending" list needs to be maintained by the client to keep track of what emails still need to replicated.

- **Read-one and Write-all** With this strategy, all remote reads go to one email account, which is treated as the primary email account, and every write goes to all email accounts in the replication group, including the primary email account. Since we're writing to all email accounts, strong consistency is ensured at the loss of performance. If the primary email server goes down, any backup server can immediately replace the primary server, without the need for further updates.

- **Read-fast and Write-fast** This strategy has been developed from our observation that some email services provide better performance for reads and some better performance for writes. Hence, all writes go to the email servers which provide higher write speeds. All reads are requested from servers which provide higher read bandwidth. Hence, replication must occur from the servers which service writes to the service which service reads. Theoretically, this strategy can achieve the highest performance. But since EMFS utilizes lazy replication, it cannot guarantee that all updates are propagated to the backup servers immediately. However, since data is cached by the client, the user will always read the latest data, if he accesses the system from the same computer.

3.4.4 Fast Recovery

Email services provided by third-party service providers are supposed to be highly available and extremely reliable, but failures do happen from time to time regardless of the efforts of service providers. For example, Google had a few outages over the past few years [16]. As discussed earlier, EMFS is configured to handle email server failures such as these. In EMFS, each email account may be configured with a few backup email accounts, each of which is a mirror of the primary account. These accounts together form the replication group. If the primary disk fails somehow, the system can quickly identify which disk fails. We cache a connection to each primary disk and send no-op messages to the server from time to time to keep the connection alive. Once there is no response from a server, we assume that the server breaks down and that the email account is not available temporarily. To recover the system, we just simply treat a backup disk in the same replication group as the new primary disk.

Like primary accounts, backup accounts may also fail. A backup failure is not as critical as a primary disk failure and hence dealing with this type of failure is not as critical. If any backup server is found to be unresponsive, the system can adopt different ways to deal with it. A simple way to handle this situation would be to first inform the client of the backup failure, and then ask the user to manually configure a new backup account. An automatic configuration tool may be designed to help the user set up a new backup and transfer data from the primary disk to the new backup. EMFS currently does not support this automatic configuration tool.

3.4.5 Garbage Collection

In EMFS, any metadata update or data update will generate a new version of metadata or data, which will be appended to the physical storage of email servers as new emails. The older versions of data and metadata are not immediately reclaimed by the system. EMFS reclaims storage space occupied by redundant data lazily as this increases performance, while simplifying design.

There are different cases where redundant data is generated. First, since any modification to data or metadata generates a new version, many versions of metadata, files, blocks and directories exist in our file system. Old versions are not necessary to maintain the system in a consistent state when versioning is not supported. Second, when a file or directory is deleted, the system only updates the metadata to indicate that that file or directory doesn't exist. The contents of the actual file or directory will still exist on the server untild eleted explicitly from the server. These emails can be deleted later on to reclaim some space on the system. In addition, redundant data may also appear on email servers due to system or network crashes.

To reclaim storage from redundant data, the system regularly scans the whole file system by utilizing old versions of metadata. By comparing different versions of metadata, the system

knows what has been modified. For example, if a file entry is in an old metadata email but not in a new metadata email, it indicates that the file was deleted. Similarly if the file entry is present, but the version numbers differ, it indicates that the file has been modified. Therefore the older version of the file may not be needed. First, all emails related to the deleted file or older version of the file are removed completely. This can be easily achieved as the unique id of the file is known from the old metadata email. Then the metadata email itself is deleted, once it has been completely scanned and processed.

Orphaned blocks, which are created during network or system crashes when a file has been partially transferred to the server, are detected the next time the file is opened. The *status* bit for the file entry will indicate that the file is in an inconsistent state and at this point, the server will be checked to see if the whole file is present. If not, blocks of the partial file are deleted and the file is reverted back to the previous version.

3.5 Security

The main focus of this work has been the design and optimization of an email based system. System security has not been given due consideration so far. This type of system may be vulnerable to "spoofing" attacks, where attackers may send emails resembling a metadata or data email, thus corrupting the file system. This type of attack may be avoided by having users supply a unique string, which may be included in every metadata and data email. The presence of this unique string may be enough to verify that an email has been sent by the system and not by an attacker. Additionally, EMFS can utilise an IMAP feature to create a separate mailbox or "folder", to which, messages from the file system can be uploaded. The name and id of this mailbox can be randomly generated by the system to ensure security. With this mechanism, attackers would still be able to send spoofed emails to the user's account, but the messages would be delivered only to the account's inbox and not to the special file system folder. However, hacked email accounts can still lead to corruption and loss of data confidentiality. Although it might be impossible to prevent hacking attacks, data privacy and confidentiality can still be ensured through encryption of all data sent to email servers using EMFS. It should be noted that EMFS does not support these features as yet.

Chapter 4

Performance Evaluation

4.1 Experiment Setup

We evaluate our prototype using a combination of microbenchmarks, macrobenchmarks and synthetic benchmarks. We compare our results against 4 other existing systems, JungleDisk, GmailFS, NFS, and AFS, and report the feasibility and effectiveness of our approach. Experiments were run on an Intel duo-core machine (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. Email accounts from three service providers, Gmail, Gawab Mail, and NCSU Campus Mail (Google Apps) were used for evaluation.

4.1.1 Evaluation Workloads

Evaluation of our system is done in two phases. First, we use synthetic benchmarks to show how the performance of our system varies with different parameters. We then use the most optimal configuration, derived from the results of these benchmarks, to compare our prototype against existing systems.

The following parameters of the system were varied to show their effects on performance: *block size*, *stripe width*, *number of connections*, *stripe servers*, and *mode of replication*. The *block size* is the size of the data chunk being transferred in one email. A file is split into several chunks of size *block size* and transferred to the servers. *stripe width* indicates the number of email accounts being used for data access. In all performance evaluation tests, a separate email account was used for dedicated metadata access. Several IMAP connections can be created and used simultaneously per email account. For example, Gmail allows seven simultaneous connections per account. The number of connections used is indicated by the *number of connections* parameter. Finally we use different *stripe servers* and different *modes*

of replication to show their impact on performance. Different service providers provide varying levels of performance and this variation can be taken advantage of for replication purposes (as discussed in 3.4.3). We also assume that all service providers provide the same level of service, without discrimination, to all email accounts of the same class, hosted on their servers.

Comparison of our system against Jungle Disk and Gmail FS was done using 5 benchmarks: Postmark [22] - a macrobenchmark, , IOZone [20], an Open Office workload [5] (suitably modified to simulate "real user" access) and synthetic benchmarks.

Postmark: Postmark is a macro benchmark that measures performance for network based systems by simulating access on short lived small files. It is used to create a large number of files of varying sizes and too measure performance on a series of transactions conducted on these small files. Version 1.51 of the benchmark is used for evaluation.

IOZone: IOZone is a popular benchmark used to test filesystem performance. IOZone is highly flexible and can be configured to simulate several different workloads such as random reads and writes, sequential reads and writes, and stride reads. Version 3.287 has been used for evaluation of EMFS.

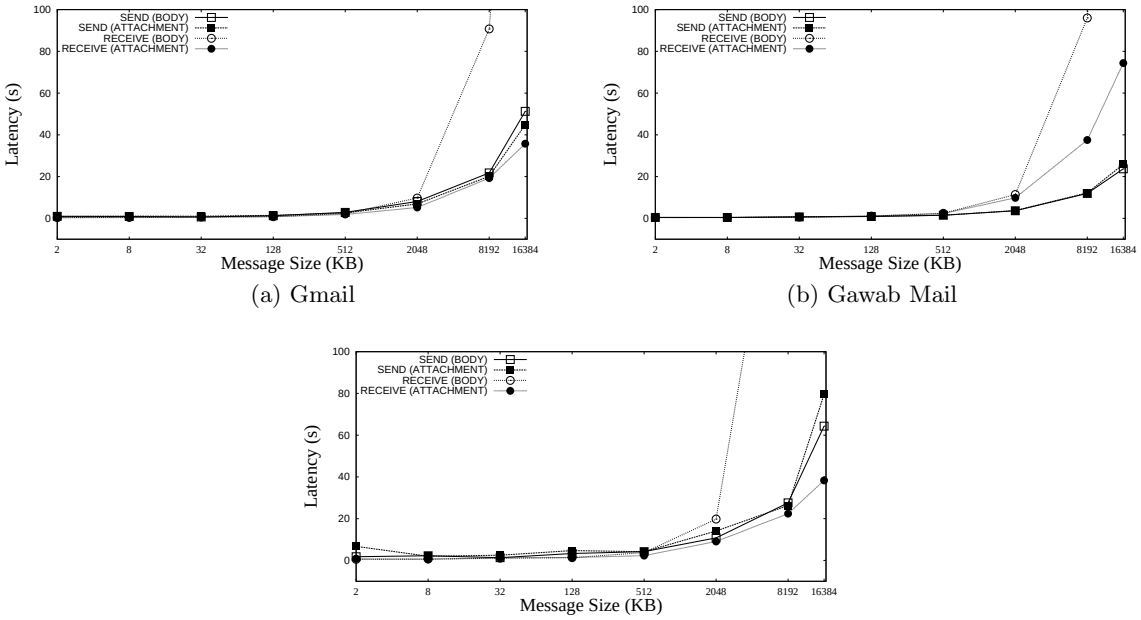
Office Workload: The Office workload is part of the Battery Life Toolkit (BLTK) which measures battery performance. The workload has been suitably modified to do the following: open Writer from the Open Office package, input data into it from a specified file, perform common work processing tasks such as save and repllace, save the file and close it.

Synthetic Benchmark: This benchmark is used to write and read files of varying sizes. It is used to test the performance of the system for writes, and reads for varying file and block sizes and stripe widths.

Since this system is currently designed for a single user scenario, multi-threaded benchmarks were not considered. However, FUSE itself is inherently multi-threaded and initializes a pool of 10 threads to service concurrent file system requests. For all performance evaluation tests either the time taken for data transfer to or from a server or the throughput is measured. Higher throughput and lower transfer times are indications of good performance.

4.1.2 Basic Parameter Variation

Figure 4.1 shows the time taken to send and retrieve a single email from one email account of each of the three service providers. The latency was measured when messages with payloads of size between 2 KB to 16 MB were sent to or retrieved from one email account on each of the three different service providers. The payload contained binary data mixed with ascii text and was always bound to only one email. Furthermore, the payload was sent in two different ways - as an attachment and in the body of the email. The time measured for sending the payload includes the time taken to create and format a message and send it completely to



(a) Gmail

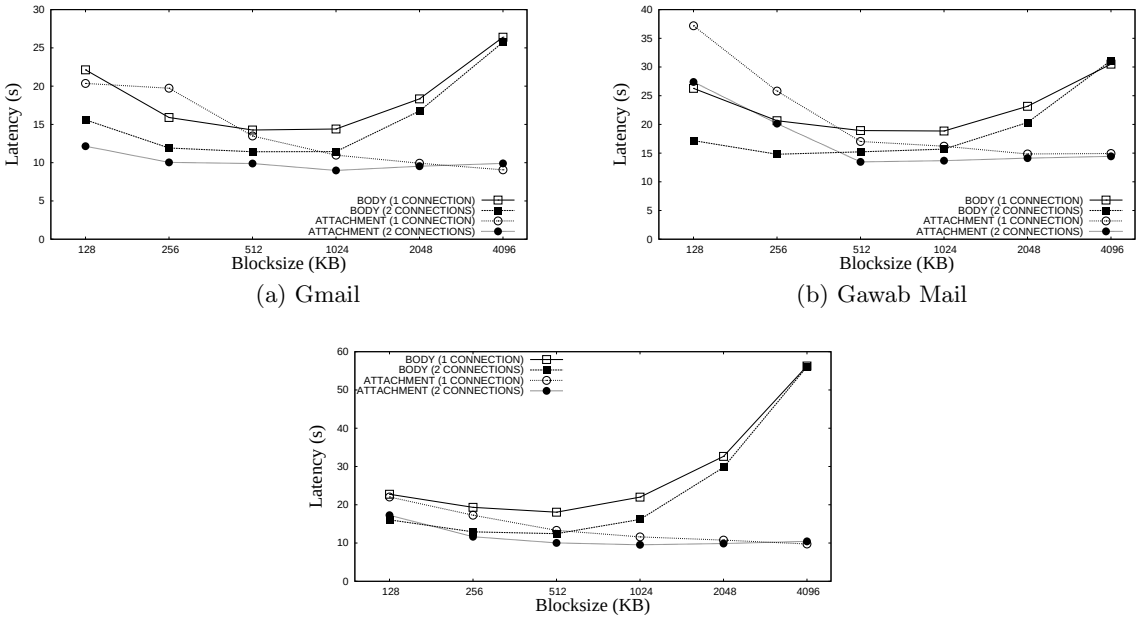
(b) Gawab Mail

(c) NCSU Gmail Apps

Figure 4.1: Single email sending/retrieving performance

the server; time measured when the payload is retrieved from the server includes the time for searching, retrieving and formatting the downloaded string. As expected, the results clearly show that as the payload size increases, the time taken to send or retrieve the data increases. In addition, it can be seen that that the latency when the payload was sent using the body is always worse (Gmail) or at best equal to the latency when an attachment was used as the means of delivery. When data stored in the body of an email is read back from a server, the time taken for the transfer is equal to or less than that of the time taken when the data is stored in an attachment, for smaller file sizes. However, as the payload size increases, the performance for the body degrades badly when the file size exceed 2MB. The degradation in performance is not due to the time taken for the actual download of the data (which is comparable to the time taken to transfer attachments from the server), but due to the time taken to transform the downloaded string into an email message format. This is probably due to the fact that email bodies were never meant to carry large sizes of ascii data, much less large sizes of binary data.

Figures 4.2 and 4.3 both vary the block size (128KB to 4MB) and the number of IMAP connections (1,2) to an account with a fixed file size of 4MB. The results again confirm that using the email body to store data for transfer gives, at best, the same level of performance as when attachments are used to store data. Furthermore we can see that for smaller block sizes, using more connections provides better performance. When the block size is small in



(a) Gmail

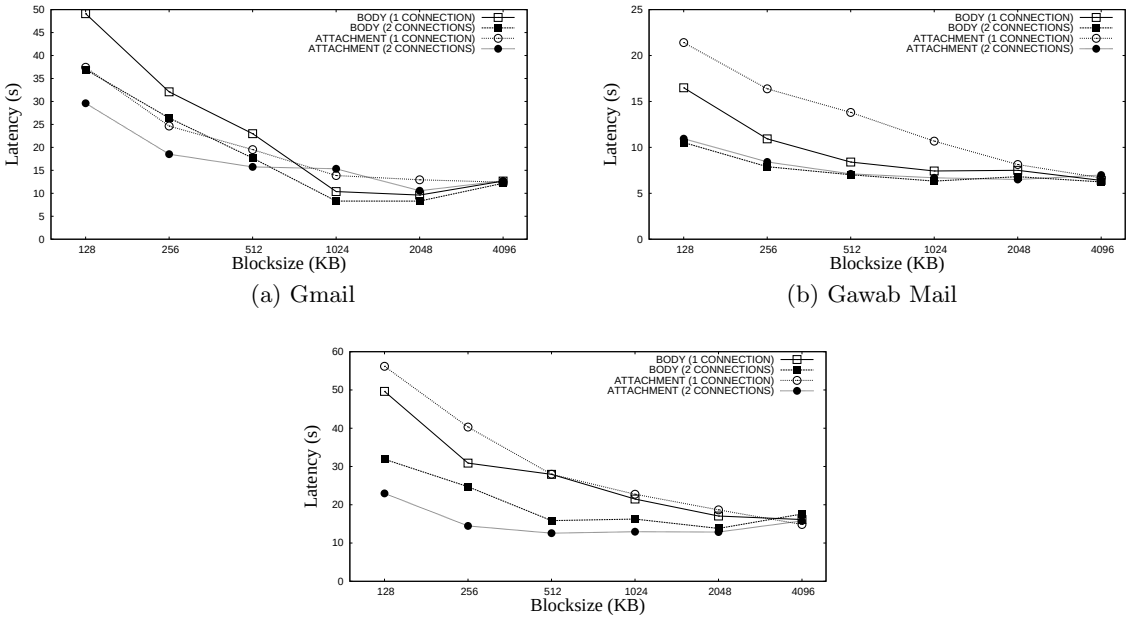
(b) Gawab Mail

(c) NCSU Gmail Apps

Figure 4.2: Single email read performance

comparison to the file size, several blocks need to be sent to/received from the email servers. In this case, all connections to an email account are used simultaneously, utilizing more client size bandwidth and thereby reducing transfer time. As the block size approaches the file size, less number of blocks need to be accessed, and the difference in performance, between using one connection and two connections, is lowered. For both reads and writes, having smaller blocksizes actually increases the response time and hampers performance. This is because with smaller blocksizes more number of emails need to be sent or retrieved, thereby incurring overhead due to having more round trips to the servers and also due to email protocols (headers and handshakes). The results also indicate that some service providers are faster when emails are sent to them (Gawab), while others are faster when emails are retrieved (Gmail). This fact can be used to implemented the read-fast/write-fast mode of operation discussed earlier to improve performance. The write performance of the Gawab servers is much better than that of the Gmail servers. This is probably due to a variety of potential reasons, like more complex security mechanisms in Gmail than Gawab and differing implementations of the IMAP RFC by each service provider.

Based on the results shown so far, we see that overall data transfer is faster when data is stored as attachments in emails. Hence, for the configuration for the results shown in figures 4.4 and 4.5, data was stored in attachments and striped across varying number of servers.



(a) Gmail

(b) Gawab Mail

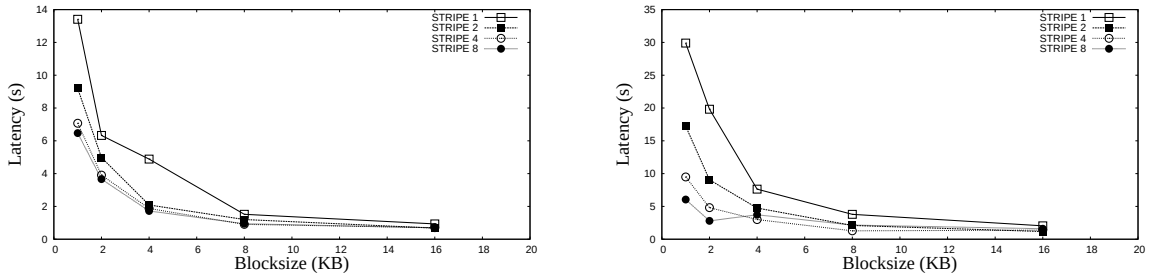
(c) NCSU Gmail Apps

Figure 4.3: Single email write performance

In addition the block size was varied to judge its effect on the data transfer time. Email accounts from Gmail were used. For both reads and writes, increasing the stripe width definitely increases the performance. This is because, we write data to all servers in the stripe width simultaneously. So with stripe width n, we write n times more data, with the total transfer time being approximately equal to just one round trip time. This effect is more profoundly seen in figures 4.4a and 4.4b for smaller files than for larger files. Furthermore, block sizes of 512 KB or 1024 KB give the lowest transfer times for larger files.

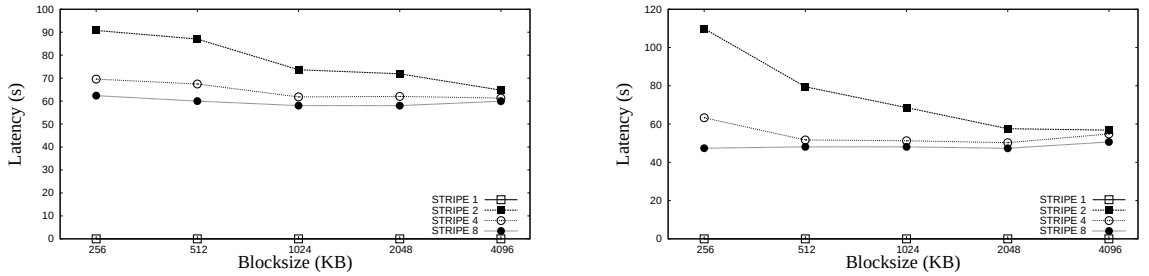
In summary, we see that with a single account and one connection, as the file size increases, the data transfer time also increases. Furthermore, data stored in attachments can be transferred faster than data stored in body. Striping data to multiple accounts certainly helps boost performance, and lowers the response times for both reading and writing. Having multiple connections per email account also lowers the data transfer times, especially when the block size is small when compared to the file size. We also see that among the servers tested, Gawab provides optimal speeds for writing, while Gmail is better for reading.

Blocksizes also affect the transfer times of files. Small blocksizes increase the response times as more emails need to be sent or received, increasing the overhead of data transfer. Similarly, large blocksizes tending towards the file size, are also unhelpful, as they do not allow the system to utilize all configured stripe servers and/or multiple connections to an account.



(a) Read (b) Write

Figure 4.4: Whole file access latency with different block size and stripe width (Gmail). File Size 32 KB



(a) Read (b) Write

Figure 4.5: Whole file access latency with different block size and stripe width (Gmail). File Size 32 MB

From the results shown in 4.5, we see that blocksize 512 KB produces the best performance.

4.1.3 Comparison with Other Systems

Based on the conclusions derived from the previous results, we configure EMFS with the following parameters: *stripe width* = 8, *block size* = 512 KB, *IMAP mode of transfer*. This configuration was used in all of the following experiments, where EMFS was compared with NFS, AFS, and Jungle Disk. The authors wished to compare EMFS with Gmail FS IMAP as well, as that would have provided a like for like comparison. However Gmail FS IMAP is unstable and often crashed with the benchmarks used. We also compare two different configurations of EMFS, namely EMFS- NORMAL and EMFS- WRITE FAST. With the first configuration, we use a set of 8 gmail accounts for data transfers. Gmail accounts were chosen for the EMFS-NORMAL configuration, as these accounts are widely used and popular. With the second configuration, we use 8 Gawab accounts, called write-fast servers, for writing data and 8 Gmail accounts or read-fast servers, for reading data from. Read requests are serviced

from the local disk cache for both of the above configurations. However, lazy replication does take place with the write-fast configuration.

Both NFS and AFS servers were configured on dedicated machines, similar in configuration to our test machine. Jungle Disk, which stores data on the Amazon S3 servers, was configured such that background or asynchronous transfers were disabled. All three systems employed a local disk cache to cache data.

Postmark

Postmark creates several files initially and performs several transactions on these files. Before the benchmark finishes, any remaining files created by the benchmark are deleted completely. A single transaction involves two operations in total, one from reading/appending and one more from creating/deleting a file. The biases for each of these operations are configurable. Files created as part of transactions are not included in the number of files configured when the benchmark is started. Two configurations of Postmark were used to conduct the comparison. Any particular configuration of Postmark, regardless of the number of times the benchmark is run or the underlying file system it is run on, always reads and writes the same amount of data and performs the same set of transactions.

When a single run is finished, the read and write throughputs, as well as the time taken to complete the benchmark is reported. However, Postmark reports throughput in a strange manner. Read and write throughputs are calculated as the total amount of data read or written, divided by the total time taken to run the benchmark. The total time taken to run the benchmark includes the time taken for the initial creation of files, time for performing all transactions and the time for finally deleting any remaining files. Hence, the read to write throughput ratio is always constant for any particular configuration, for any file system. Therefore, we have decided to report the total time taken to complete a single run of the benchmark, instead of the throughput reported by Postmark, as we feel this would be a more accurate measure for comparison.

The results shown in figure 4.6 were generated using the following Postmark configuration: Equal bias for reading and appending, equal bias for creation and deletion, with two hundred transactions in total involving files of size ranging between 4 KB and 16 MB. Any settings not mentioned above were set to the Postmark default values. The number of files initially created by Postmark during each of its runs was varied to see how each system scales with increasing number of files. The results show that NFS and AFS are highly scalable and are faster than Jungle Disk or EMFS. This is because both are highly optimized systems which run in the kernel, utilize and optimize raw connections for data transfer, and in addition have smart servers. Additionally, both NFS and AFS were set up inside the campus network, meaning that data transfers took place over the local network rather than the Internet. EMFS clearly struggles

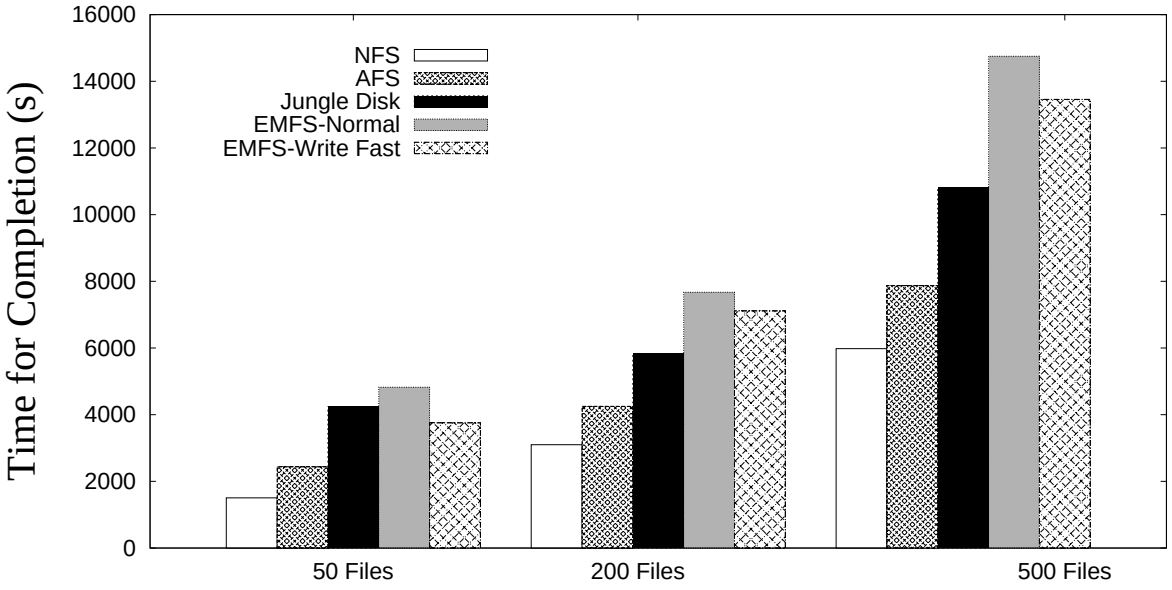


Figure 4.6: Varying number of files using Postmark

when large numbers of files are created and transferred immediately. Jungle disk is better than EMFS but still pales in comparison to NFS and AFS. The reason for the poor performance of EMFS is due to the fact that email protocols were never meant for large data transfers. Additionally, EMFS was tested using strict synchronous writes as were the other systems. However, because of the handicap of a "dumb" server, EMFS has to send two metadata emails for every write, to ensure consistency. This becomes expensive when a lot of files are written consecutively. Jungle Disk is proprietary software and hence we cannot conclusively state why it performs slower than AFS or NFS.

Figures 4.7 shows results generated from using a second Postmark configuration: 200 files, 200 transactions, file sizes ranging between 4 KB and 16 MB. The bias for creation, reading, writing and deletion was varied to judge their effect on the different systems. The bias for each pair of operations (read/append and create/delete) should always total 10. Four different configurations were used and compared - equal bias (read bias 5, append bias 5, create bias 5, delete bias 5), read heavy (read bias 8, append bias 2, create bias 5, delete bias 5), append heavy (read bias 2, append bias 8, create bias 5, delete bias 5), create heavy (read bias 5, append bias 5, create bias 8, delete bias 2). Any settings not mentioned above were set to the Postmark default values. The equal bias configuration is the same as the 200 files configuration discussed previously. Again, for all variations in bias, AFS and NFS perform better than EMFS and Jungle Disk due to the same reasons as mentioned above. The read-heavy configuration

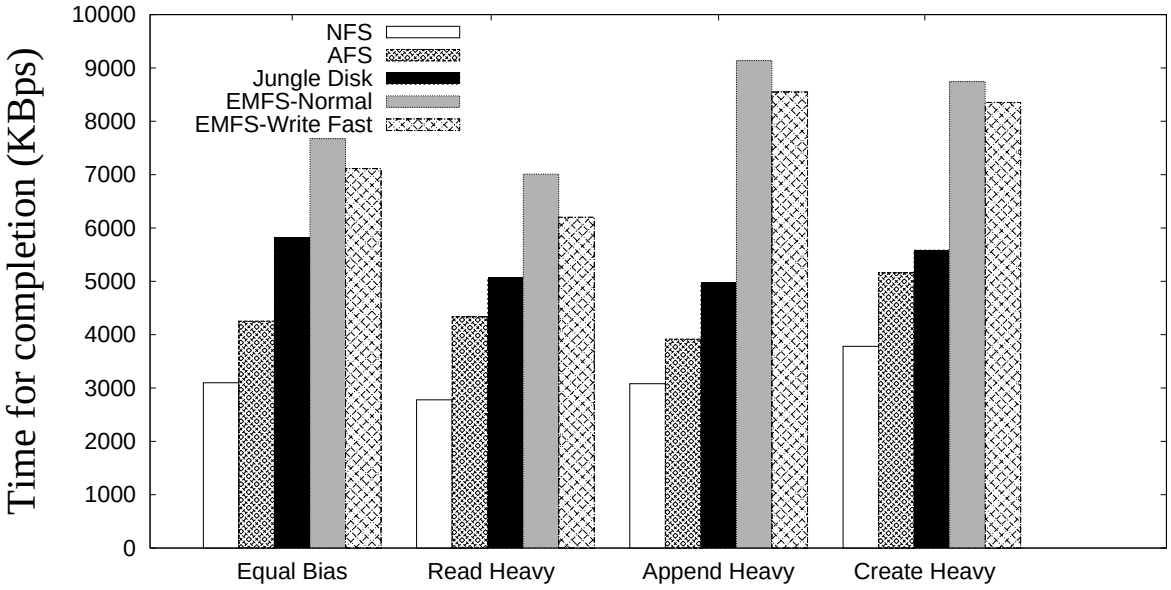


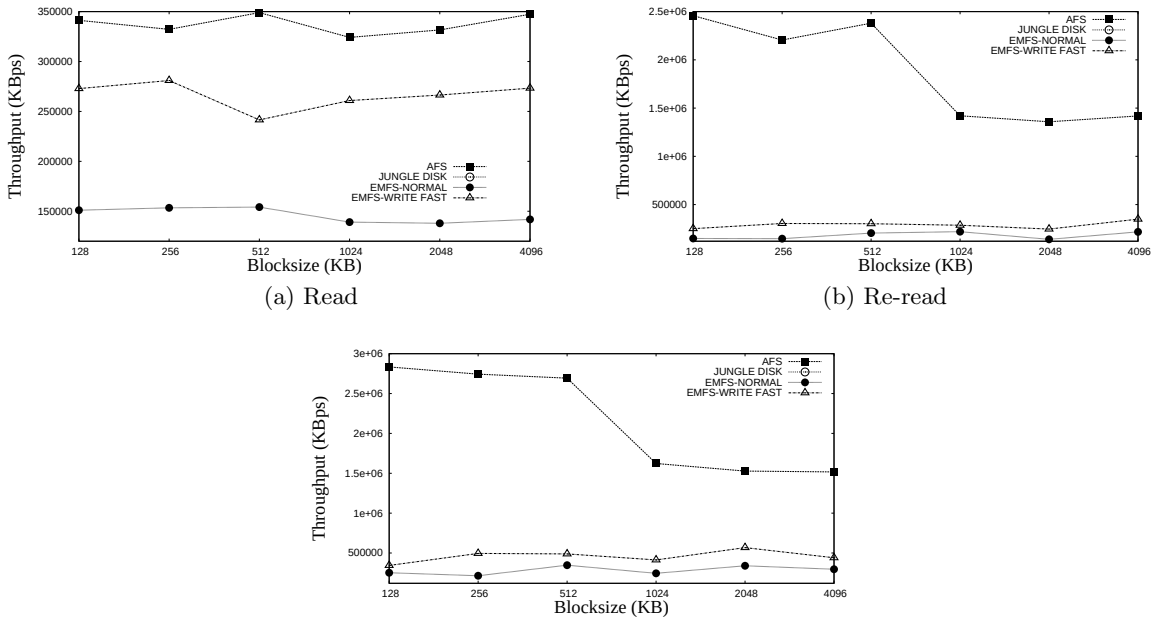
Figure 4.7: Varying transaction bias using Postmark

finishes faster than all the other configurations for all systems, because less data is written to the servers. All systems utilize their own local caches as well as the OS page and buffer caches. Hence the difference in performance arises mostly due to the difference in write throughput. Both the append heavy and create heavy configurations transfer nearly write the same amount of data to the server and hence their performance is very similar. For these two settings, 2743 MB and 3134 MB of data respectively is transferred over the network and EMFS struggles with such large quantities of data. This is again due to the nature of the email protocols.

IOZone

IOZone can be used to thoroughly test a file system's performance using several different workloads. For benchmarking EMFS, IOZone was used to create a file of size 16 MB and test the performance of the system using varying block sizes with three different workloads: write/re-write, read/re-read and random read/random write. Furthermore IOZone was configured so that every write was synchronous (using the O_SYNC flag) and such that the time taken for the file close was also included in calculations. IOZone performs these tests sequentially, one immediately after the other, for a given file and block size.

Figures 4.8 shows the read performance of a 16 MB file in block sizes ranging from 128 KB to 4 MB. The read performance shown is extremely high for all systems shown except Jungledisk. There are two reasons for this. One, all systems cache their data on the client machine and two,



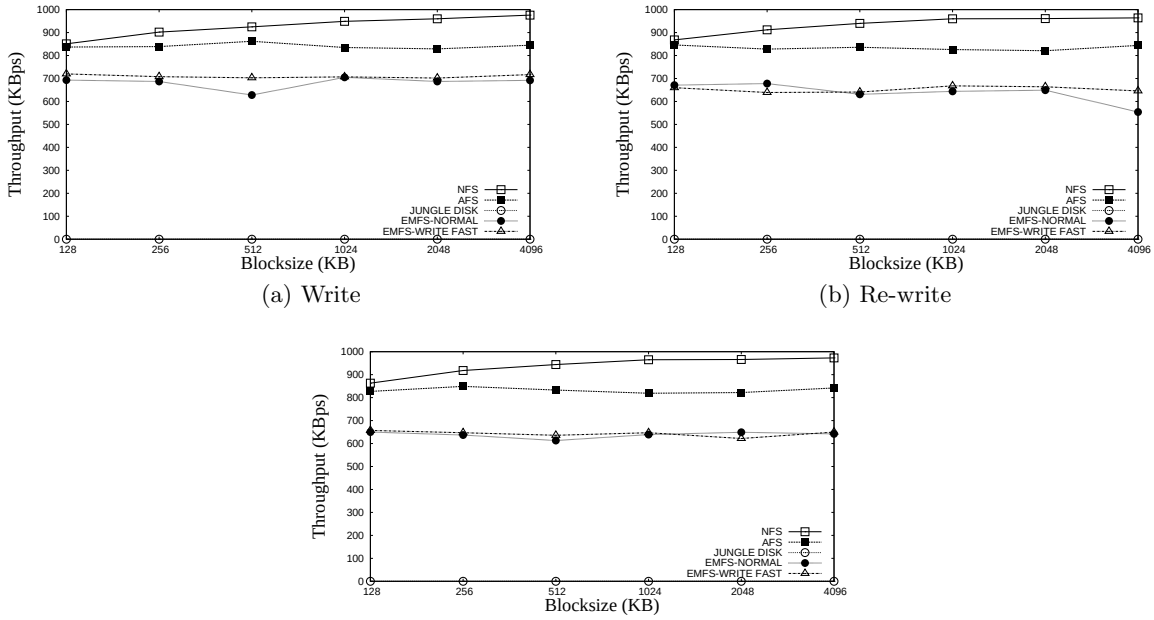
(a) Read

(b) Re-read

(c) Random Read

Figure 4.8: Read Performance of a 16 MB file using IOZone

the OS level disk and buffer caches also come into play. IOZone conducts its tests sequentially in the following order: write and re-writes, reads and re-reads and finally random-reads and then random-writes. Hence, when the read performance is tested, the file has already been cached by the filesystem itself as well as the OS level page and buffer caches. This results in very high performances of 200 MBps to 3 GBps as shown above. However, Jungle Disk still reports only around 30-35 MBps, which is the typical transfer rate on a normal hard drive. The read performance for Jungle Disk suffers due to the fact that it sends out metadata updates, through a setattr operation, when a file is closed. More specifically, Jungle Disk updates the last accessed time of the file on file close(). This update is not sent out on every file close, but whenever there is a significant difference (about 3-8 seconds from our observations) between the current time and the last accessed time. For a file as large as 16 MB, read operations take long enough such that the last accessed time is updated every time on file close. This extra delay of about 0.5 to 1 second significantly lowers Jungle Disk performance. NFS and AFS report extremely high read throughput, greater than 1 GBps. This is again due to their own caching mechanisms as well as the OS caches. EMFS doesn't report as high a throughput due to two reasons. First of all, NFS and AFS are both implemented in C and running inside the kernel, whereas EMFS runs in userspace. This results in more context switches than NFS and AFS, slightly lowering performance. Secondly, Fuse limits each read request to a maximum of



(a) Write

(b) Re-write

(c) Random write

Figure 4.9: Write Performance of a 16 MB file using IOZone

128 KB at a time. Hence, even larger requests made by the application, are broken down into this size before being passed to the EMFS call handler library. This results in more calls to the read() function, and generally increases overhead. Additionally, the EMFS prototype has been implemented in Python, in which function call overhead is generally expensive. However, one advantage of EMFS is that, the file data is cached in memory on file open(). This results in very stable performance regardless of the data access pattern. Hence, EMFS reports the same throughput even when the OS buffer and page caches are cleared. When the page and buffer caches are cleared, NFS reports a throughput of only about 900 KBps, whereas EMFS reports the same value as when the caches were not cleared. This implies that NFS doesn't cache locally but instead retrieves from the network server instead. NFS caching is not in-built within the system but users need to set up external caching mechanisms by themselves. A similar trend is noted for AFS when the caches are cleared. However, AFS has a local cache and hence AFS read speeds will be much higher than NFS, but still comparable to EMFS. Furthermore, Jungle Disk reports very low speeds (about 550-600 KBps) for random reads. The reason for this is partly due to how IOZone carries out the test. IOZone opens up the test file in r+ mode for reading. 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 hence the drop in performance.

Figures 4.9 shows the write performance of a 16 MB file in block sizes ranging from 128 KB to 4 MB. Clearly, all systems display the same trend for all file writes, regardless of the write access pattern (write, re-write, and random write). In order to accurately reflect the time taken for the system to transfer the file to the server, the IOZone test was configured such that the file closing time was included in throughput calculation. All the systems under comparison transfer the entire file to the server only on close() or flush() and not on every write() call. Any changes made to the file are cached locally regardless of the access pattern of writes (write, rewrite or random writes). The changed file is then transferred to the server and hence the write throughput is dominated by the network bandwidth rather than anything else. The results also show that EMFS writes are slightly better than Jungle Disk in terms of Write Performance. This is possibly because Jungle Disk sends out additional metadata updates, such as the last accessed time, separately on file close(), as mentioned above in the discussion on read performance. This introduces an extra delay of about 0.5 to 1 second, resulting in the slightly lower performance. As before, NFS and AFS are much faster due to their being highly optimized systems utilizing protocols used for fast and efficient data transfer.

Open Office

The BLTK Open Office benchmark simulates a real user office document editing trace. It was originally designed to test battery life on laptops running typical user workloads. To compare these systems, it has been modified to test out keystroke and local processing delay on Open Office Writer files. The automated scenario opens an existing Open Office Writer file from the target file system and proceeds to perform typical office document editing functions, such as typing in new pages of text, deleting existing text, find and replace etc. . . for about ten minutes. Finally, the edited document is saved back to the target file system and the scenario ends. The main aim of this benchmark is to show that the local processing time on EMFS is negligible when compared to any other distributed file system. We compare the running time of the benchmark against other distributed file systems as well as the local file system. Any delays, be it due to keystrokes, slow reads, slow writes would be accurately reflected in the running time of the benchmark. Table 4.1 reports the percentage scores of each file system, normalized over the score of the local file system. The difference in scores seen is mainly due to the document being saved at the end of the scenario. This involves network operations and hence introduces a small delay of 3 to 4 seconds, lowering the score.

Summary

The results from the variation of basic parameters shows that data transfer is fastest when attachments are used. Storing data in an email body provides comparable performance to attachments for data writes. However, data read times when data is stored in an email body

Table 4.1: File System Performance for a typical Office Workload

File System	Normalized Score
Local	100
NFS	92.07
AFS	91.23
Jungle Disk	90.04
EMFS - Normal (Gmail)	88.36
EMFS - Read-Write Fast	90.2

rapidly increases when the file size increases. Striping to more email servers reduces the data transfer times and increases performance. The results indicate that a block size of 512 or 1024 KB is optimal. Smaller block sizes increase the overhead because more emails have to be transferred. Larger block sizes also provider poorer performance as they defeat the purpose of striping. Increasing the number of connections per email account reduces transfer times. EMFS does not scale well when large numbers of files of varying size are written and read continuously. Email protocols were not designed for large data transfers. Furthermore, the handicap of a dumb server results in EMFS having to send more email to ensure atomicity. However, EMFS provides comparable file write performance on individual files with Jungle Disk and slightly lower performance when compared with AFS and NFS, as shown by the IOZone results. EMFS provides nearly 5 to 6 times better performance than Jungle Disk on individual file reads. The read performance, however, is well below the performance of NFS and AFS due to small latency introduced by FUSE, the fact that FUSE restricts read requests to a maximum of 128 KB and function call overhead in Python in which the EMFS prototype is implemented. However, we see that EMFS is comparable in performance to other systems for individual file reads and writes. Network factors, email server implementation, router byte level caching, location based routing, network congestion, and email server load could all potentially influence performance. EMFS is primarily meant for single user personal file access. Hence the expected workload is not large, with files of small sizes. EMFS performs extremely well for this type of target workload, as shown in figure 4.4. Overall EMFS is cost effective, and is convenient as it utilises users' personal email accounts to store data.

Chapter 5

Conclusions

This work examined the viability of an email based distributed file system. We design and implemented such a system and analyzed its performance using various synthetic and real world benchmarks. Our results indicate that such a system is effective and comparable in terms of performance to systems of similar scale and size, such as Jungle Disk. However it lags in performance behind classic distributed file systems such as NFS and AFS. Our results show that as we increase the number of servers, we utilize more client side bandwidth, leading to higher throughput. Furthermore, we established flexible limits on the minimum and maximum block sizes through experiments for maximizing performance. We believe there are several future directions to this work. An interesting direction would be to pursue client side data de-duplication which would help greatly improve performance. Smaller improvements in performance can also be achieved by batching metadata updates together. It has been shown through studies [2] that most files are either modified or deleted within ten minutes of creation. Thus, batching metadata updates would reduce the number of metadata emails sent, thereby slightly increasing performance. Effective data differencing would also help the system and reduce the traffic on the network. The ultimate aim is to design a complete and effective system, incorporating all of these features, without any loss in performance.

REFERENCES

[1] Adobe buzzword. https://www.adobe.com/acom/buzzword/.

[2] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3):9, 2007.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[4] T. Bray. The bonnie benchmark. *http://www.textuality.com/bonnie*.

[5] L. Brown, K.A. Karasyov, V.P. Lebedev, A.Y. Starikovskiy, and R.P. Stanly. Linux laptop battery life. In *Proceedings of the Linux Symposium*, pages 127–146, 2006.

[6] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 63–65, 2006.

[7] Mark Crispin. Internet message access protocol - version 4rev1. RFC 3501, March 2003.

[8] Curlftpfs. http://curlftpfs.sourceforge.net/.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings Of The Fifteenth Acm Symposium On Operating Systems Principles*, 2007.

[10] Michael Demmer, Bowei Du, and Eric 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*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[11] Dropbox. http://www.dropbox.com/.

[12] Comparison of webmail providers. *http://en.wikipedia.org/wiki/Comparison_of_webmail_providers*.

[13] Expandrive. http://en.wikipedia.org/wiki/ExpanDrive.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[15] Gmailfs. *http://sr71.net/projects/gmailfs/*.

[16] Google outages damage cloud credibility. *http://en.wikipedia.org/wiki/Gmail*, 2009.

[17] Google Docs.

[18] J.H. Hartman and J.K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, 13:274–310, 1995.

[19] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[20] Iozone filesystem benchmark. http://www.iozone.org.

[21] Jungle disk. http://www.jungledisk.com/.

[22] Jeffrey Katcher. Postmark: A new file system benchmark. *Technical report TR3022. Network Appliance*, October 1997.

[23] J Klensin. Simple mail transfer protocol. RFC 2821, April 2001.

[24] Eliezer Levy, Eliezer Levy, Avi Silberschatz, and Avi Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4:529–540, 1992.

[25] Lftpfs. http://lftpfs.sourceforge.net/.

[26] Yutong Lu, Huajian Mao, and Jie Shen. A distributed filesystem framework for transparent accessing heterogeneous storage services. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[28] Elinor Mills. Yahoo mail to offer unlimited storage. *http://news.cnet.com/2100-1038_3-6171111.html*, 2007.

[29] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM.

[30] J. Myers and M. Rose. Post office protocol - version 3. RFC 1939, May 1996.

[31] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, 2008.

[32] Microsoft office live. http://www.officelive.com/.

[33] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM.

[34] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. Nfs version 3 - design and implementation. In *In Proceedings of the Summer USENIX Conference*, pages 137–152, 1994.

[35] Vijayan Prabhakaran. Iron file systems. pages 206–220. ACM Press, 2005.

[36] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, New York, NY, USA, 2010. ACM.

[37] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[38] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[39] M. Satyanarayanan, J. Kistler, and E. Siegel. Coda: A resilient distributed file system. In *IEEE Workshop on Workstation Operating Systems*, 1987.

[40] Amazon Web Services. Amazon simple storage service. http://aws.amazon.com/s3/.

[41] Skydrive. http://windowslive.com/online/skydrive.

[42] Livio B Soares, Orran Y Krieger, and Dilma 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*, pages 41–52, New York, NY, USA, 2003. ACM.

[43] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, Berkeley, CA, USA, 2003. USENIX Association.

[44] Miklos Szeredi. File system in user space. *http://fuse.sourceforge.net/*, 2006.

[45] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995.

[46] Avishay Traeger, Nikolai Joukov, Josef Sipek, and Erez 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.

[47] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):1–56, 2008.

[48] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5(4):1–28, 2009.

[49] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[50] Min Zhou and Alan Jay Smith. Analysis of personal computer workloads. In *MASCOTS '99: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 208, Washington, DC, USA, 1999. IEEE Computer Society.

[51] Zumodrive. http://www.zumodrive.com/.