BeanFS: A Distributed File System for a Large Number of Immutable Files

Wook Jung Daewoo Lee Eunji Pak
Youngjae Lee Sang-Hoon Kim Jin-Soo Kim
Computer Science Department
Korea Advanced Institute of Science and Technology
{wjung,dwlee,pakej,yjlee,sanghoon}@calab.kaist.ac.kr
jinsoo@cs.kaist.ac.kr

Taewoong Kim Sungwon Jun
NHN Corporation
{twkim,swjun}@nhncorp.com

CS/TR-2008-289

K A I S T
Department of Computer Science

BeanFS: A Distributed File System for a Large Number of Immutable Files

Wook Jung Daewoo Lee Eunji Pak
Youngjae Lee Sang-Hoon Kim Jin-Soo Kim
Computer Science Department
Korea Advanced Institute of Science and Technology
{wjung,dwlee,pakej,yjlee,sanghoon}@calab.kaist.ac.kr
jinsoo@cs.kaist.ac.kr

Taewoong Kim Sungwon Jun
NHN Corporation
{twkim,swjun}@nhncorp.com

Abstract

The amount of immutable files, such as images, video clips, audio files, and e-mail messages, is expected to grow significantly, as users actively generate, distribute, share, and re-use digital contents. In this paper, we present BeanFS, a distributed file system for a large number of immutable files. A key features of BeanFS include (1) the volume-based replication scheme to alleviate the metadata management overhead of the central metadata server, (2) a lightweight consistency maintenance protocol tailored to the simple access patterns of immutable files, and (3) volume synchronization and re-replication mechanisms which provide high availability against transient and permanent failures.

BeanFS has been evaluated using a microbenchmark and a synthetic workload which reflects the characteristics of the real-world e-mail service system. Our measurement results indicate that the performance of BeanFS is scalable as the number of servers increases. In addition, volume synchronization and re-replication are quite effective in enhancing the availability of BeanFS.

1 Introduction

The amount of digital information has grown exponentially over the past decade and it is expected that the trend will continue with the proliferation of broadband Internet connectivity. Recently, IDC (International Data Corporation) forecasts that the amount of information that is either created, captured, or replicated in digital form will increase by almost 60% annually until 2011 [13]. IDC identifies that the major sources of this growth are images, video clips,

audio files, and e-mail messages, all of which have an *immutable* property. The immutable property of a file indicates that once the file is written, it is not modified afterwards. Most Internet service companies will have to deal with an ever increasing number of immutable files, especially in the upcoming Web 2.0 [17] era where users are actively generate, distribute, share, and re-use UGCs (User Generated Contents) as exemplified in YouTube, Flickr, Facebook, and so on.

Distributed file systems running on a cluster of inexpensive commodity hardware are being recognized as an effective solution to support the explosive growth of storage demand in large-scale Internet service companies. For example, Google has built their own Google File System (GFS) [11] to cope with increasing storage needs of various Google services, and Amazon has developed Amazon Simple Storage Service (Amazon S3) [1] to provide storage service to third party services as well as to themselves. Yahoo is also known to use open source Hadoop Distributed File System (HDFS) [2] for handling web search queries. All of these file systems achieve scalability by distributing data to numerous nodes, while providing availability by replicating data to multiple nodes and ensuring consistency among replicas.

In this paper, we focus on a distributed file system for immutable files, which occupies the vast majority of the total storage capacity required by large-scale Internet services. Our study reveals that the existing distributed file systems are not suitable for a large number of immutable files in the following three respects.

First, the metadata management scheme needs to be reexamined. Usually, file systems are required to manage a large amount of metadata and to process a lot of metadata operations. Traditional distributed file systems take either centralized or decentralized approach to metadata management. The decentralized metadata management scheme [10, 21, 12] seems to be suitable for a lot of metadata operations because they can distribute metadata operations over all nodes. However, it is difficult to balance load among nodes, keep global data without central servers, and coordinate system-wide activities. The centralized scheme [11, 8, 26, 2] makes the system simple and flexible, but central servers can easily become a performance bottleneck.

Second, it is expensive and even unnecessary to provide a strong consistency guarantee to immutable files. The existing distributed file systems employ strong consistency maintenance mechanisms such as primary replica technique [15, 11, 26] and quorum protocol [10]. Although there is little difference in supported consistency levels, they basically maintain consistency among replicas against concurrent writers to the same file. Ensuring strong consistency among replicas is expensive, as file systems should serialize every operation or check whether write conflicts exist during normal operations. Since there are no concurrent writes to immutable files, a failure, which makes a replica miss some update requests, is the only source of inconsistency among replicas. Due to their simple access patterns, a consistency protocol for immutable files can be designed to be simple and fast.

Third, recovering from *transient failures* can be simplified. A transient failure denotes the situation where the failure can be resolved in a prompt and

timely manner. The persistent data in the failed node can be made consistent simply by applying operations which occurred during the failure. Distributed file systems such as GFS [11] and HDFS [2] do not handle transient failures separately; they copy all the data in the failed node to another live node whenever they detect a failure, which results in significant disk and network traffic as well as excessive load in the central server to coordinate the recovery process. Ceph [26] and Harp [15] are considering transient failures, but their recovery mechanisms are based on primary replica technique, which is expensive to use for immutable files.

This paper presents the design and implementation of BeanFS, a distributed file system for a large number of immutable files. The key features of BeanFS can be summarized as follows.

- The volume-based replication scheme alleviates the metadata management overhead of the central metadata server.
- A lightweight consistency maintenance protocol for immutable files enables the system to be simple and fast.
- Transient and permanent failures are treated separately. Recovering from transient failures can be done quickly and has less overhead.

The remainder of the paper is organized as follows. We begin by giving our motivation in the next section. Section 3 describes the design goals and the overall architecture of BeanFS. Section 4 discusses possible failure types in BeanFS and Section 5 elaborates upon recovery mechanisms from transient and permanent failures. The availability of our system is analyzed in Section 6. Section 7 presents the evaluation results and Section 8 compares BeanFS to other approaches. Finally, we conclude in Section 9.

2 Motivation

2.1 Target System Overview

NHN Corp. is a company that is running `naver.com`, one of the largest Web portal sites in South Korea. The site provides an integrated search service as well as a wide range of Internet services including webmail, blogs, communities, games, etc. to more than thirty million users. NHN Corp. is ranked as the world's fifth largest search service provider by comScore in December 2007 [9]. Our work is initially started to provide a cost-effective storage solution to a large-scale, web-based e-mail service system for NHN Corp.

The target system architecture of BeanFS is depicted in Figure 1. The traditional web-based e-mail system consists of a number of Mail servers and Webmail clients, one or more Index database servers, and a huge storage system for storing e-mail messages. Mail servers process incoming and outgoing SMTP (Simple Mail Transfer Protocol) sessions, and Webmail clients service HTTP requests generated as users read, send, or delete mails. Sent and received e-mail

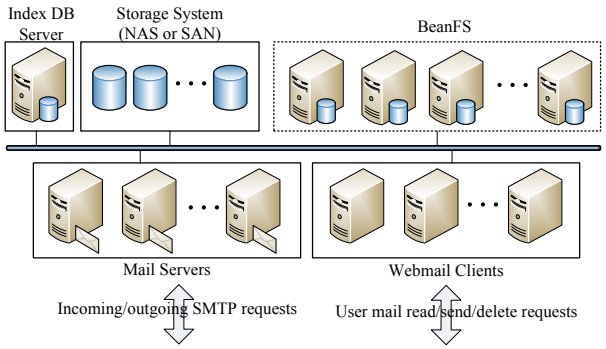


Figure 1: A web-based e-mail system architecture

messages are stored in the storage system and their metadata (such as sender, recipients, subject, etc.) are managed by Index database servers.

In its simplest form, the storage system for e-mail messages can be constructed with conventional NAS (Network-Attached Storage) or SAN (Storage Area Network). However, using general-purpose storage systems based on NAS or SAN is likely to be overkill for immutable files such as e-mail messages, as full POSIX compliance and strong consistency guarantee they provide are not necessary for immutable files, let alone the cost they require. The main goal of BeanFS is to add complementary storage to the target system just for immutable files (as shown with the dotted box in Figure 1), thus reducing the overall cost while providing higher level of scalability, availability, and aggregated performance. The pioneering work of Google [11] has already demonstrated that an application-specific storage system running on a cluster of low-cost servers can be a cost-effective alternative to general-purpose storage systems.

2.2 Workload Characteristics

In order to obtain storage requirements for BeanFS, we have investigated the characteristics of the target workload. Mail servers and Webmail clients write e-mail messages sequentially when processing incoming or outgoing mails. There are no concurrent writes to the same file since Index database servers assign a unique file name to each session. Read or delete operations are generated by users or an automatic spam classifier. Due to spam mails, file creation and deletion are very frequent which induces heavy directory updates as well.

Figure 2 shows the distribution of e-mail sizes for 507,306,028 SMTP session logs which we collect from one of Mail servers from Jan. to Sep. 2007. We can see that the e-mail service system mostly deals with small files whose sizes are several tens of KB (Kilobytes). 95.0% of files are less than 55 KB, and 98.5% of files are less than 100 KB. Note that files less than 100 KB occupy 62.3% of the total storage. Therefore, the storage for e-mail messages should handle small-sized read and write requests efficiently.

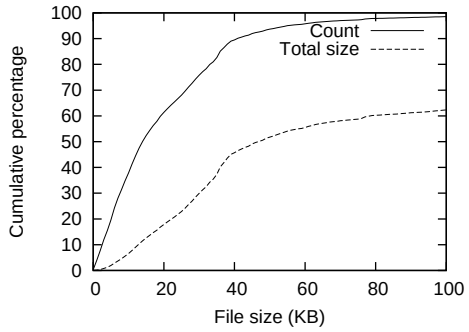


Figure 2: The distribution of e-mail sizes

2.3 Storage Requirements

From the workload characteristics investigated in the previous subsection, we can summarize storage requirements for BeanFS as follows. Many of these requirements are also applicable to other immutable files such as images, audio files, and video clips.

First, it is necessary to support a very large number of small files that are created and deleted frequently. Second, there are primarily three file access patterns: create, read, and delete. When a file is created, it is written sequentially. Any mutations or concurrent writes to a file do not exist. Reading from a file is allowed only after all writes to the file finish, and concurrent reads from the same file are allowed. A file is not accessible anymore if a delete operation is performed on the file. Finally, full POSIX compliance is not necessary. Since applications running on Mail servers and Webmail clients are developed in-house, they can be modified to use our own APIs instead of POSIX-compliant APIs.

3 BeanFS Design

In this section, we describe design goals and the overall architecture of BeanFS.

3.1 Goals

During the design step, we have clarified the following design goals for BeanFS to meet the storage requirements presented in Section 2.3.

- The main goal of BeanFS is to support a very large number of files. Most of files in BeanFS can be as small as tens of KB in size. Therefore, BeanFS should be able to deal with several orders of magnitude larger number of files than the existing distributed file systems can handle. The volume-based replication scheme in BeanFS is an attempt to lessen the metadata management overhead.

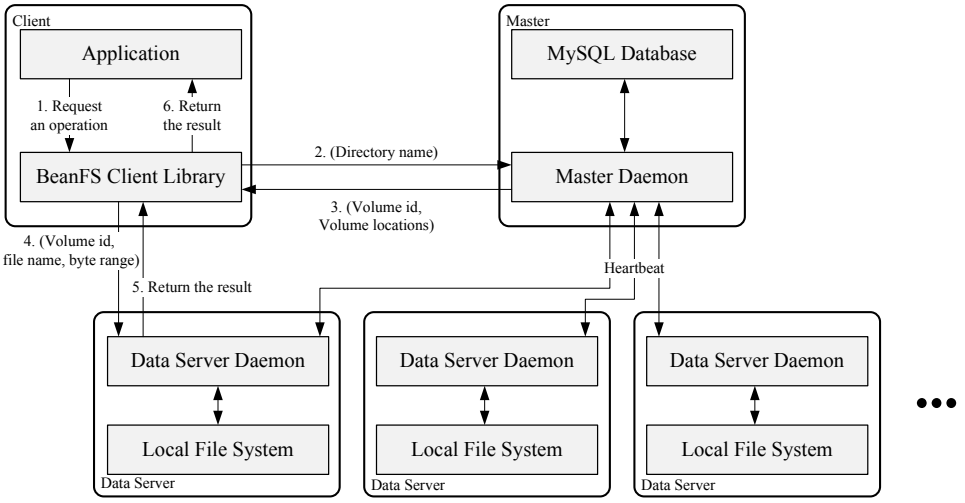


Figure 3: The overall architecture of BeanFS

- BeanFS should provide a cost-effective storage solution. As the storage capacity required by large-scale Internet services is increasing explosively, cost-effectiveness is becoming one of the most important features that any storage system should have. To achieve this, BeanFS utilizes a large number of inexpensive commodity hardware instead of expensive, proprietary storage hardware.

- BeanFS should maintain high availability under component failures, which are the norm rather than the exception in commodity hardware [11]. Although BeanFS relies on the same replication technique to cope with component failures as other distributed file systems do, it is necessary to develop a simple and lightweight consistency maintenance protocol and recovery mechanisms which are tailored to the characteristics of immutable files.

- Easy manageability is another feature we focus on. Since the storage capacity grows rapidly and components break down frequently, it should be possible for storage servers to be added to or removed from the running system. In addition, BeanFS should provide automatic storage rebalancing, monitoring infrastructure, and management tools to reduce administrative costs.

3.2 Overall Architecture

A BeanFS cluster consists of a single master server, numerous data servers, and a number of clients that access files stored in BeanFS, as illustrated in Figure 3. Communications among these components are performed through FlexRPC [14],

a customized RPC (Remote Procedure Call) layer built for BeanFS.

The architecture of BeanFS resembles those of the existing distributed file systems with a central server such as GFS and HDFS [11, 2]. The main difference is that the master in BeanFS manages metadata information not on the basis of files, but on the basis of *volumes*. A volume is a collection of files that reside in the same directory. This volume-based replication scheme reduces the amount of metadata information that should be maintained by the master and, in fact, a key design to support a very large number of files with a single master.

The master manages volume location information, monitors the states of data servers, and administrates system-wide activities. Files in the same volume are replicated on a specified number of (three, by default) data servers. To process a file system request, a client first queries the master for data server locations of the volume to which the target file belongs, and forwards the request to the corresponding data server(s).

The decentralized architecture, as adopted in some distributed file systems [21, 10], could be a solution to support a large amount of file metadata. However, the decentralized approach complicates system design since activities requiring system-wide knowledge, such as failure detection, recovery, load balancing, and file placement, have to be done in a distributed way without centralized control. The centralized approach eases system design and metadata management [11, 2]. To prevent the central server from being a bottleneck or a single point of failure, several distributed file systems employ multiple central servers [8, 26]. However, the overall performance is still bounded by the number of central servers, and in practice, the performance is degraded due to load imbalance among central servers. In any case, inadequate support for a large number of small files poses serious problem for the existing distributed file systems as they are mainly targeted for larger files.

3.3 Volume Management

As briefly described in Section 3.2, BeanFS replicates files based on volumes. This volume-based management scheme is inspired by AFS [16] and Coda [12], but more fine-grained in BeanFS. Unlike a volume in AFS and Coda which includes all files of a partial subtree, files in BeanFS are grouped into volumes according to the directory they belong to in the global namespace. In other words, each directory in the namespace corresponds to a volume and all the files in the particular directory are contained in the same volume. Since the namespace is built and controlled not by the user but by the system in our environment, it suffices to use each directory as the granularity of file replication. For the target e-mail service system, we can simply form a flat directory structure where each user has its own directory and all the e-mail messages for the user are stored in the same directory. Similarly, for other types of immutable files such as images or video clips, each album or category can be located on different volumes.

A volume is replicated on a specified number of data servers. Each volume is identified by a 64-bit volume identifier (*vid*) and each file is accessed by the

file name and the associated *vid*. The mapping from a pathname to the *vid* is performed by the master.

The volume-based replication scheme alleviates the master's load significantly as most of metadata operations are actually performed on data servers. The master only needs to handle volume creation (`mkdir`) and deletion (`rmdir`) operations, which are not frequent operations. Meanwhile, data servers deal with the rest of metadata operations such as file creation (`creat`), deletion (`unlink`), and stat'ing (`stat`), as well as normal read and write operations.

3.4 BeanFS Components

3.4.1 Master Server

The master server manages the file system's global information including the namespace, mappings from directories to volumes, volume locations, and access control policies. By exchanging heartbeats with data servers, the master monitors the *health* of each data server along with the current utilizations of CPU, network, and storage. The master also coordinates system-wide activities such as volume synchronization, re-replication, and migration (cf. Section 5.2 and 5.3).

As mentioned earlier, the single master simplifies system-wide decisions and metadata management, but it can be a single point of failure or a performance bottleneck. For the former problem, we provide two stand-by servers and the fail-over is taken place automatically in case the master crashes (cf. Section 4.2). For the latter problem, we minimize the master's intervention during normal operations, and cache the volume location information shortly at the client side.

Contrary to the other systems in which the master reconstructs its information from data servers at startup time [11, 2], the master in BeanFS keeps all the information persistently in a MySQL relational database. This slightly increases the update cost, but eases data manipulation and simplifies database backup to stand-by servers. Since the volume information is much smaller in size and less frequently updated, keeping it in a relational database is not expensive in terms of both space and time. For 50 million volumes, the MySQL database occupies only 2.2 GB including the associated index structure.

3.4.2 Data Server

In data servers, each volume has its own directory in the local file system and all the files that belong to a volume are stored in the same directory. Thus, hot volumes as well as hot files can benefit from a local buffer cache. Data servers also perform actual tasks related to volume synchronization, re-replication, and migration under the master's control.

3.4.3 Client

The client module services file system requests from applications. To access files, each application is linked with the BeanFS client library. When a file system request arrives, the client module communicates with the master and identifies the file's *vid* and the volume location. For a read request, the client contacts one of data servers to retrieve the file contents. Other *update* requests such as `write`, `creat`, and `unlink`, are forwarded to all data servers which have the target volume *vid*.

3.4.4 RPC Layer

During the development of BeanFS, we find the existing RPC layers lacking in many important features required for implementing distributed file systems. This leads to the development of our own RPC layer called FlexRPC [14].

One of the most demanding features which other RPC layers do not provide properly is the support for full multithreaded environment in both the client and server sides. Specifically, FlexRPC implements dynamic thread pooling on both sides in order to minimize thread management overhead.

Another feature of FlexRPC is the support for various calling patterns at the RPC level. We have classified frequently-used data/control flows into three calling patterns: single, parallel multicasting, and serial multicasting. The *single* calling pattern is identical to the traditional calling pattern used in SunRPC. In *parallel multicasting*, a client invokes multiple RPC calls with identical arguments to many servers. In *serial multicasting*, a client transmits arguments only to one of data servers, and the server forwards the request to another server in a chained and pipelined manner. In FlexRPC, call handers in the server are capable of processing all the calling patterns simultaneously. BeanFS makes use of multicasting calling patterns extensively. The exact type of multicasting is determined at run time depending on the payload size; if the payload size is small, BeanFS uses parallel multicasting. For larger payload sizes, serial multicasting is used to save the network bandwidth.

FlexRPC supports both UDP and TCP protocols, and guarantees at-most-once semantics over UDP using *response cache*. In addition, FlexRPC internally caches used handles to speedup connection management and verifies the integrity of payload using CRC32. These functionalities have reduced the complexity and the development cost of BeanFS significantly.

4 Failure Management

BeanFS should be able to provide uninterrupted service, even in the presence of failures. Our goal is to maintain high availability and consistency among replicas, while minimizing performance degradation against component failures. BeanFS treats failures differently according to the location they occur: a client, a master, or a data server.

4.1 Client Failure

In BeanFS, each client has a responsibility to update all replicas. Let us assume that a client fails while creating a file. In this case, the create operation may be delivered to only a subset of replicas, resulting in inconsistency among replicas. A distributed commit protocol, such as two or three phase commit [24], can be employed to guarantee the consistency, but the use of such protocols will increase the latency of operations in a normal, failure-free condition.

BeanFS does not handle failures on clients immediately in favor of a simple and fast update mechanism. This is perfectly acceptable as the target environment does not enforce any consistency guarantee for failed update operations. Instead, another Mail server or Webmail client will retry the operation for the failed session with a new file name. The inconsistency resulted from client failures will be eventually fixed with a system management daemon running in background in data servers.

4.2 Master Failure

Since the master server is the most essential component for ensuring the continued operation of BeanFS, we use a reliable, relatively expensive hardware. This cost is negligible compared to the total hardware cost for data servers. To eliminate a single point of failure, an automatic fail-over mechanism is implemented for the master. Two stand-by servers are connected to the master in a daisy chain fashion, and mirror the database of the master in real time. When a failure occurs to the master or to the first stand-by server, the successor in the chain takes over the role of the failed node.

Two stand-by servers synchronize their databases with the predecessor's by the use of MySQL replication [3]. In fact, updates in the predecessor's database are propagated in an asynchronous manner. This asynchronous replication improves the update performance in the predecessor, but the following stand-by server is exposed to a potential danger that may lose some recent database updates when the predecessor fails suddenly. To solve this problem, the master and two stand-by servers share an external storage where MySQL binary logs are stored. When a fail-over happens, the successor replays the binary logs stored in the storage before string the master service. The required storage capacity for the binary logs is small (about several hundreds of MB) and RAID [18] ensures the reliability of the external shared storage.

Because the master keeps all the information in a database, a fail-over procedure is simple and fast, compared to GFS [11] in which a master needs to gather some information from numerous chunkservers during fail-over. In BeanFS, when a stand-by server detects a failure in the predecessor, it triggers a fail-over procedure. Clients and data servers retry RPC calls to the master until the fail-over completes. Requests to the master are redirected to a new master by refreshing ARP tables via the RARP protocol. If the failed node is recovered, it is attached to the end of the chain and becomes a new stand-by server.

4.3 Data Server Failure

Data servers run on inexpensive commodity hardware which has relatively high failure rate. For this reason, a volume is replicated on the specified number of data servers, and file system services are handled by remaining replicas in case of failures. Disk failures of data servers threaten the availability since they decrease the number of available replicas. Temporary network outages endanger the consistency among replicas as some update operations cannot be delivered to data servers. Thus, BeanFS should preserve the specified number of replicas against permanent data losses and ensure the consistency under temporary data server outages.

We categorize data server failures into transient and permanent, and treats them in different ways. A transient failure indicates the failure that can be resolved in a prompt and timely manner such as temporary network outages, momentary hardware malfunctioning, and machine reboots. Note that all volumes stored in the failed node are still available after the transient failure is resolved. A failure is called as permanent if the volumes are not available anymore due to disk failures, hardware breakdowns, and operating system crashes.

There are several challenges to handle data server failures. First, it is hard to distinguish transient and permanent failures. Second, a failed node misses update operations performed during the failure, hence the replica in the failed node becomes inconsistent with other valid replicas. Third, a permanent failure reduces the number of available replicas, jeopardizing the availability of the volume. Finally, another failure could occur again while the system is recovering from the previous failure.

The master periodically monitors the states of data servers with heartbeat messages. The master can also detect a failure in a data server if it does not respond to an RPC request. The master tells transient failures from permanent failures by timeout. By default, if a failure is not resolved in 30 minutes, it is regarded as a permanent failure. In addition, the administration's decision or disk errors detected by the monitoring system forces the failure to be treated as permanent immediately. The next section describes how BeanFS handles two types of failures in more detail.

5 Data Sever Recovery

5.1 Consistency Maintenance Protocol

BeanFS can guarantee volume consistency if it copies all the files that belong to the faulty server to another live data server as soon as a failure is detected. However, since the storage capacity provided by each data server easily goes beyond 1 TB, copying all replicas in our environment will generate a lot of small disk I/O, degrading the overall performance. Noting that volume inconsistency caused by transient failures can be resolved with relatively small overhead, we handle transient and permanent failures separately. BeanFS recovers volumes

Replica state	Description
$State(v, d) = N$	All files in volume v on data server d are normal
$State(v, d) = R$	Data server d is recovering files in volume v
$State(v, d) = F$	Files in volume v on data server d are inaccessible
$State(v, d) = M$	Files in volume v are being re-replicated or migrated to data server d
Volume state	
$State(v)$ $=[State(v, ds1), State(v, ds2), State(v, ds3)]$ where $ds1$, $ds2$, and $ds3$ denote data servers that have a replica for volume v.	

Table 1: Replica and volume state

from transient failures by volume synchronization (described in Section 5.2) and from permanent failures by re-replication (described in Section 5.3).

A part of replicas in BeanFS may not be available due to failures. Some of replicas may remain in an inconsistent state while the recovery procedure is in progress. Therefore, the system should control accesses to these unavailable or inconsistent replicas. For this reason, BeanFS introduces the notion of *replica state* and *volume state* as shown in Table 1. A normal replica is defined to be in N state. If a data server fails, all the replicas in the data server go to F state. As soon as the transient failure is resolved, i.e., the failed server is rebooted or network is reconnected, the state is changed to R and BeanFS starts volume synchronization to recover volumes from transient failures. Eventually, all replicas go back to N state. When a replica is copied to another data server (re-replication) due to permanent failure or load balancing, the state is changed to M. Note that a replica in R or M state can go into F state if the associated server fails in the middle of volume synchronization or re-replication. Figure 4 illustrates a state transition diagram for each replica in BeanFS. We define a volume state as a tuple consisting of each replica's state.

BeanFS maintains two types of logs for each volume: MO-LOG and DO-LOG. MO-LOG (Missed Operations Log) contains missed update operations for the volume during transient failures. MO-LOG is stored in the remaining N-state replicas of the same volume. Since all update operations are delivered to the remaining N-state replicas, the logging can be performed without any additional communication. MO-LOG is replayed on the failed replica after the failure is resolved. DO-LOG (Delayed Operations Log) is exploited by R or M-state replicas. Updates to replicas in R or M state cannot be executed

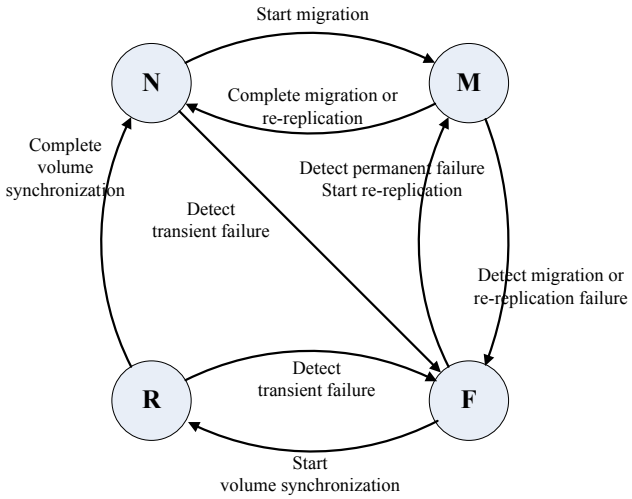


Figure 4: Replica state transition diagram

immediately because the target file may be unavailable or incomplete in the new data server. Thus, these updates are recorded in DO-LOG, and applied after all files are copied.

The volume state determines the action taken by clients and data servers. Upon receiving an update operation, a data server records the operation in MO-LOG if its volume state contains at least one F-state replica. On the other hand, if its replica is in R or M state, the data server records the operation in DO-LOG. A client sends read operations to one of N-state replicas, while it broadcasts update operations to all replicas except for F-state replicas.

Volume states are cached in each component: the master, data servers, and clients. One of the challenges in BeanFS is to make these distributed components see a consistent view of volume states. The most up-to-date replica states are maintained by the master, and a data server exactly knows the state of each replica it has. However, the cached replica states in other data servers can become stale at any time. This is particularly important in BeanFS since clients and data servers decide, based on volume states, whether to record MO-LOG or whether to send a read or an update request to a replica. For example, let us assume that the volume state for a volume v is initially given by $State(v) = [N, N, F]$, and a state transition $F \rightarrow R$ has taken place in the third replica, updating $State(v)' = [N, N, R]$. Since this state change is not automatically notified to clients, a client may send an update operation only to the first two N-state replicas, although it should be delivered to the third replica as well. One possible solution is that the client renews the volume state from the master before making any requests, but this is not only inefficient increasing the latency of normal operations, but also incorrect as the volume state can be changed again after the master responds to the client's state renewal request.

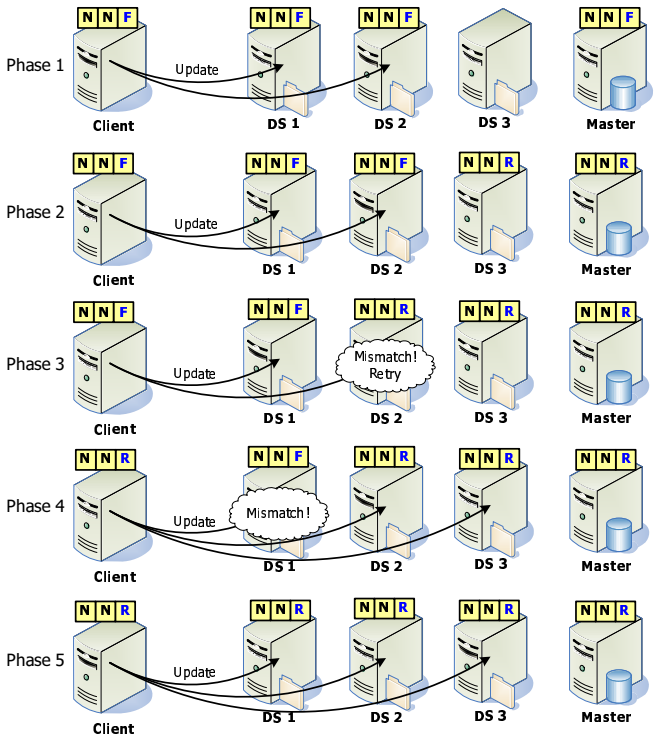


Figure 5: State matching example

BeanFS employs a novel mechanism called *state matching* to guarantee a consistent view of volume states among the master, clients, and data servers. State matching allows an operation to be executed in a data server if clients and data servers agree on the volume state. A client embed its cached volume state in every update operation to data servers. Each data server accepts the operation if the embedded volume state is identical to the volume state cached in the data server. Otherwise, the data server renews the volume state from the master and performs the comparison again. If they still do not match, the operation is rejected and the client retries the operation after renewing the volume state from the master.

Figure 5 demonstrates how state matching works. In the example, a volume is replicated on $DS1$, $DS2$, and $DS3$. In phase 1, a failure occurs in $DS3$ containing the third replica, and the client sends update operations to $DS1$ and $DS2$ only. $DS1$ and $DS2$ record the operations in MO-LOG for $DS3$. The failure is resolved in phase 2 and $DS3$ updates its replica state on the master from F to R. The client, $DS1$, and $DS2$ are not aware of this state change, but it does not matter because the client's update requests are still recorded in MO-LOG by $DS1$ and $DS2$. In phase 3, $DS3$ receives MO-LOG from $DS2$ during volume synchronization, letting $DS2$ know that the replica state in $DS3$ is changed

from F to R. At this point, $DS2$ stops logging the update operations in MO-LOG. An update operation delivered to $DS2$ in phase 3 is rejected due to state mismatch. Then, the client renews the volume state from the master and retries the operation (phase 4). $DS1$ still has a stale state in phase 4, but eventually $DS1$ has the correct state after consulting the master on state mismatch (phase 5).

5.2 Recovery from Transient Failure

When a data server D_{TF} is restored from a transient failure, all replicas in D_{TF} are recovered with volume synchronization. For every volume v stored in D_{TF}, the following steps are repeated during volume synchronization. First, the data server D_{TF} notifies its state transition $F \rightarrow R$ to the master. Second, D_{TF} requests an MO-LOG to D_N, one of data servers that has an N-state replica for the current volume v. At this point, D_N notices the state change of D and ceases to log incoming update operations in MO-LOG for v. On the other hand, clients start to send update operations to D_{TF} as a result of state matching. Third, D_{TF} replays update operations in MO-LOG, while recording the newly-arrived update operations in DO-LOG. Fourth, after processing all the entries in MO-LOG, D_{TF} replays update operations in DO-LOG. Finally, D_{TF} notifies the state transition $R \rightarrow N$ to the master. Clients and other data servers eventually renew the state of D_{TF} through state matching.

In our initial design, the master managed the state of each replica for every volume in the system. Recovering a volume from a transient failure requires three state changes: $N \rightarrow F$, $F \rightarrow R$, and $R \rightarrow N$. Since the master keeps all the information in MySQL, each state change requires a database update. This has resulted in a lot of database transactions during volume synchronization, increasing the recovery time significantly. To reduce the overhead, we have revised our algorithm so that the master only keeps track of the state of each *data server*, instead of the state of each replica. Under the revised algorithm, each data sever has one of three states: F, R, or N. All replicas in an F-state data server are regarded as in F-state. An R-state data server indicates that some of replicas are in R state while the rest may be already transitioned to N state. Similarly, an N-state data server represents that most of replicas are in N state, but some may be in M state. For the latter two cases, the master's information is somewhat ambiguous and the exact replica state is only known to the corresponding data server. This does not cause any correctness problem, however, since there is no difference among N, R, and M states for clients and other data servers.

As the number of volumes grows in a data server, volume synchronization overhead increases. For every volume, the data server needs to check whether MO-LOG for the volume exists in a remote data server. This problem could be overcome by employing a *volume group*. Volumes are grouped into a relatively smaller number of volume groups. Volumes in the same volume group are placed in the same set of data servers and share the state and MO-LOG. Since the number of volumes affected by a transient failure is usually small, all the

volumes in an unaffected volume group can be switched to N state simultaneously. Using volume groups also reduces memory requirement to cache the volume information.

5.3 Recovery from Permanent Failure

Permanent failures in BeanFS are recovered by re-replication. Once a failure is classified as permanent, the master starts re-replication for the failed data server D_{PF}. For every volume v in D_{PF}, the master allocates a new live data server D_M in place of D_{PF}. D_M copies all files in the volume v from D_N, one of data servers that has an N-state replica. Then, D_M notifies the completion of re-replication for the volume v to the master.

Figure 6 briefly outlines the re-replication procedure. (1) The system monitor detects a permanent failure and notifies this information to the re-replication manager. (2) The re-replication manager inserts a set of volumes to be re-replicated into the priority queue. The initial priority of a volume is determined by the number of the remaining replicas, but may be increased later if the volume stays in the queue too long or by the administrator's explicit command. (3) The re-replication scheduler determines the source and the destination data server for the target volume and sends a re-replication request to the destination server. (4) The destination server copies files for the target volume from the source server and replays DO-LOG accumulated during re-replication.

There are several considerations for re-replication. First, the master should control the *re-replication concurrency*, which indicates the number of concurrent re-replication instances in the whole system. This is because a large re-replication concurrency degrades normal file system operations due to a lot of small disk I/O. Second, the re-replicated replica in M state receives update operations, but they cannot be performed immediately since the target file may not be copied to the corresponding server yet. Instead, those update operations are logged in DO-LOG, and replayed later after re-replication finishes. Third, another failure can happen in both the source or the destination data server while re-replication is in progress. In this case, the master invalidates the failed re-replication, allocates a new data server for the replica, and triggers re-replication again. The remaining data in the failed node are cleaned during volume synchronization (if the failure is transient).

The same re-replication mechanism is also used for migration, which changes the location of a replica to another data server. Migration is mainly used for load balancing among data servers. There are two types of migration: manual and automatic. The manual migration is enabled by the administrator especially when a new data server is added to or removed from BeanFS. The automatic migration is triggered when the disk utilization in a data server exceeds the predefined high watermark, and it is continued until the utilization goes below the low watermark.

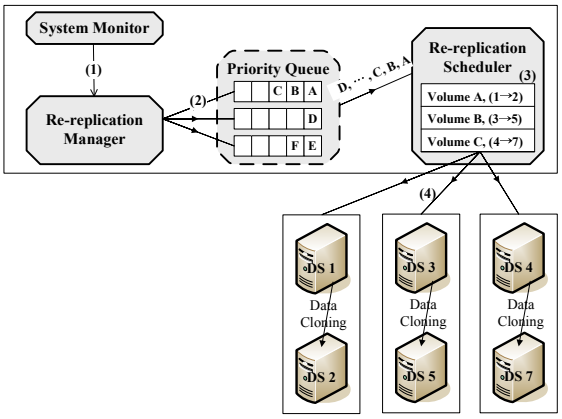


Figure 6: Re-replication procedure

Symbol	Description
λ_{tf}	Transient failure rate of a volume
λ_{pf}	Permanent failure rate of a volume
t_{tf}	Recovery time from transient failure
t_{pf}	Recovery time from permanent failure

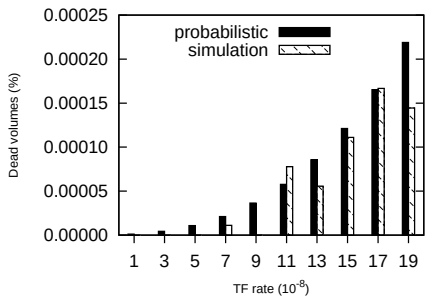
Table 2: Symbols used in the availability analysis

6 Volume Availability Analysis

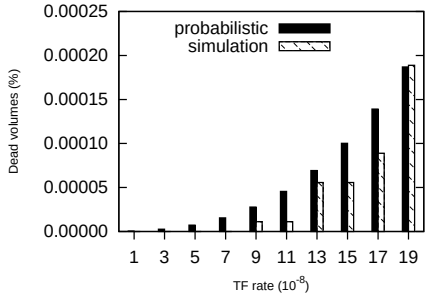
Recovery mechanisms in BeanFS rely on a valid (N-state) replica. A failed replica is recovered either by replaying logs stored in an N-state replica (volume synchronization) or by copying all files from an N-state replica (re-replication). This means that a volume cannot be recovered if there is no remaining N-state replica for the volume. We call this volume the *dead volume*. Although BeanFS has recovery mechanisms from transient and permanent failures, some volumes will become dead if a series of failures take place in a short period of time. In order to evaluate the volume availability in BeanFS, this section estimates the percentage of dead volumes generated in a given amount of time under various data server failure rates. For the sake of brevity, we assume that there is no

Case	Rate
3TF	$3\lambda_{tf}^3 t_{tf}^2$
2TF/1PF	$3\lambda_{pf}\lambda_{tf}^2(t_{pf}^2 + 2t_{tf}^2)$
1TF/2PF	$3\lambda_{pf}^2\lambda_{tf}(2t_{pf}^2 + t_{tf}^2)$
3PF	$3\lambda_{pf}^3 t_{pf}^2$

Table 3: A probabilistic model for dead volumes



(a) $\lambda_{pf} = 1.5 \times 10^{-8}$ (once in 772 days)



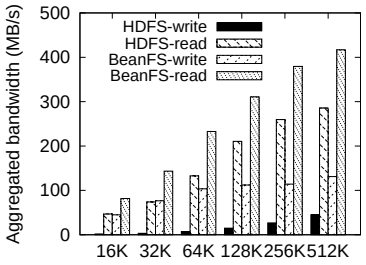
(b) $\lambda_{pf} = 1 \times 10^{-8}$ (once in 1157 days)

Figure 7: The estimated percentage of dead volumes during two years of operation

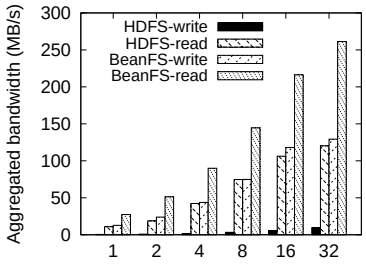
failure in the master and the replication degree is three.

The amount of dead volumes can be estimated in several ways. We first attempted to model BeanFS using a Markov chain with absorbing states [25] similar to [20]; each state represents the number of failed replicas for a volume, and failure/recovery events correspond to state transitions. In BeanFS, however, the recovery time does not follow exponential distribution and the states are not independent of each other. These do not satisfy the assumptions of the Markov chain.

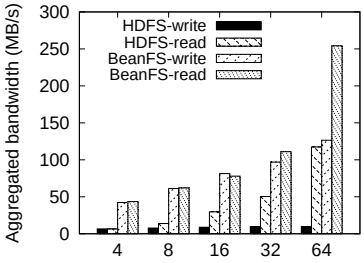
Instead, we have developed a probabilistic model in which the likelihood of dead volumes can be predicted for a given sequence of failures. Table 2 defines the symbols used in our model and Table 3 presents the possibility of becoming a dead volume after three successive failures. In Table 3, the case of *3TF* denotes that a chance of becoming a dead volume as three data servers containing the volume go into transient failures rather simultaneously. Similarly, the case of *2TF/1PF* represents the possibility that a volume becomes unavailable as a result of two transient failures and one permanent failure. Note that the cases listed in Table 3 only cover circumstances that a volume encounters two more failures while the first failure is being recovered. We ignore other more com-



(a) File size (8 clients, 64 data servers)



(b) The number of clients (32KB files, 64 data servers)



(c) The number of data servers (32 clients, 32KB files)

Figure 8: Microbenchmark results

plicated cases as they occur very rarely. Hence, the summation of each row in Table 3 approximates the rate of becoming a dead volume.

In addition, we have implemented an event-driven simulator based on CSIM [4], which models the behavior of BeanFS as closely as possible. All the functionalities of BeanFS, including volume synchronization, re-replication, and migration, are implemented on the simulator. Although it takes a considerable time to run, the simulation result is more accurate than the result obtained by the probabilistic model since the simulator considers all the complicated cases that were ignored in the probabilistic model. Furthermore, such parameters as volume sizes, failure rates, and the recovery time can be configured to match the characteristics of the real-world workload instead of using predefined mathematical distributions.

Figure 7 compares the estimated percentage of dead volumes obtained by the probabilistic model with that obtained by the simulator, during two years of operation of BeanFS. It is assumed that the BeanFS cluster consists of 200 data servers, containing the total 9 million volumes. The distributions of volume sizes and file sizes are based on the actual data sampled from the real e-mail service system described in Section 2. The time for volume synchronization and re-replication are also measured experimentally on the real BeanFS system.

In Figure 7, we consider two permanent failure rates, $\lambda_{pf} = 1.5 \times 10^{-8}$ (once

in 772 days $\approx$ 2 years) and $\lambda_{pf} = 1 \times 10^{-8}$ (once in 1157 days $\approx$ 3 years), which is fairly conservative when compared with the industry experience. In fact, if we assume that the disk drive is the sole cause of failure in a data server, $\lambda_{pf} = 1.5 \times 10^{-8}$ represents that every disk drive suffers from one permanent failure on average in two years. However, it is reported that the Annualized Failure Rate (AFR), the average percentage of disks failing per year, is far less than our settings; only 8% for 2-year-old disks and not more than 9% in most cases [19]. For transient failure rates, we varied them up to $\lambda_{tf} = 19 \times 10^{-8}$ (once in 61 days), which is higher than permanent failure rates by $13 \sim 19$ times.

On the whole, the probabilistic model and the simulation produce similar results. Even in the harsh environment where each volume experiences one permanent failure in two years ($\lambda_{pf} = 1.5 \times 10^{-8}$) and one transient failure in every two months ($\lambda_{tf} = 19 \times 10^{-8}$), the percentage of dead volumes is only 0.0002% during two years (18 volumes out of the total 9 million). This suggests that BeanFS ensures a high degree of volume availability.

It should be noted that the dead volume does not necessarily mean the loss of data. If at least one replica is in transient failure, the volume can be recovered manually even though some of latest updates may be lost. BeanFS loses the entire volume only if the volume falls into $3PF$, i.e., all the replicas are unavailable due to permanent failures. However, this is extremely rare to happen; there was only one such volume among all the simulated results shown in Figure 7.

7 Evaluation

The BeanFS cluster used in the evaluation consists of a single master, 64 data servers, and 32 client servers. All machines are configured with two dual-core Xeon LV processors (2.0 GHz), 2 GB memory, four 500 GB SATA disks, and a Gigabit Ethernet controller.

7.1 Microbenchmark

First, we compare the performance of BeanFS with that of Hadoop File System (HDFS) [2] using a microbenchmark. HDFS is an open source clone of Google File System (GFS) [11]. HDFS is composed of a central metadata sever called *namenode*, and numerous *datanodes* used to store file data. Note that HDFS is originally designed for large–typically larger than 64MB–files. In spite of this, we chose HDFS to contrast the metadata handling overhead between BeanFS and HDFS. Our microbenchmark is designed to measure the aggregated bandwidth of all clients when we vary the file size, the number of clients, and the number of data servers. During microbenchmark tests, the total 256 GB was initially written to the file system. In BeanFS, each volume holds about 512 MB regardless of the file size.

Figure 8(a) presents the changes in the aggregated bandwidth as the file size varies from 16 KB to 512 KB. We can see that BeanFS significantly out-

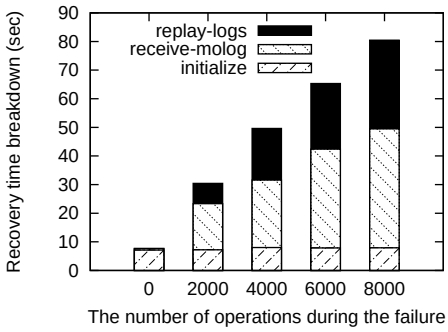


Figure 9: Volume synchronization time

performs HDFS in all cases. In particular, the aggregated write bandwidth of HDFS is much inferior to BeanFS. Although the overall file access path is similar, there are several differences between HDFS and BeanFS. First, HDFS is implemented in Java, while BeanFS in C. Second, HDFS tends to create system resources such as threads and sockets on demand without reusing them. In BeanFS, the FlexRPC layer manages system resources as a dynamic pool. Third, the namenode in HDFS allocates a block for each file and manages per-file metadata structure, while the master in BeanFS is only involved in volume location lookup. Due to these reasons, the namenode seems to be a performance bottleneck for a large number of files in HDFS, even though the namenode manages all the information in memory. This is confirmed by the fact that the aggregated write bandwidth grows from 4% to 34% of that of BeanFS as the file size is increased from 16 KB to 512 KB. The trend in the aggregated read bandwidth is similar, but it is closer to the BeanFS performance (14–46%). This is because the required processing in namenode for read operations is simpler than for write operations.

Figure 8(b) and 8(c) compares the performance of two file systems according to the number of clients and the number of data servers, respectively. The overall trend is similar to Figure 8(a) and we can observe that BeanFS is scalable with the number of clients and data servers.

7.2 E-mail workload

We have also developed a synthetic file system workload which reflects our target e-mail system. The *mailbox generator* creates a number of mailboxes. Each mailbox corresponds to a volume in BeanFS. The distributions of the file size and the mailbox size are based on the sampled data from the actual environment. Once the initial layout of the file system is created by the mailbox generator, the *operation generator* performs file system operations such as create, read, and delete, repeatedly. The ratio of these operations is also estimated from the real-world system and set to create : read : delete = 4 : 2 : 3.

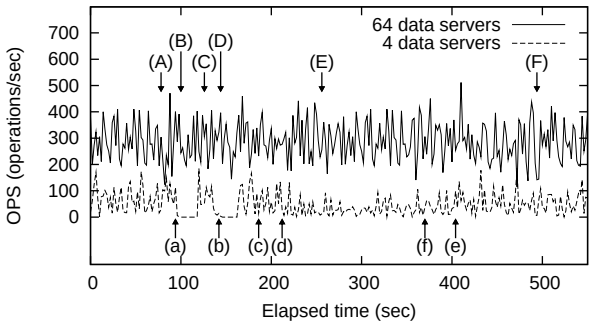


Figure 10: Impact of transient data server failures

For the e-mail workload, we generate 160,000 volumes in BeanFS. The total number of files reaches 125 million. File system operations are issued from 16 client machines concurrently. Under this environment, our measurement results show that BeanFS achieves the aggregated OPS (operations per second) of about 1,400. During the test, the CPU load was 40–60% for the master, and 20–30% for data servers. Due to small files, the performance is mainly limited by the disk bandwidth in data servers. If more data servers are added to BeanFS, we can get better aggregated performance.

7.3 Volume Synchronization

In this subsection, we evaluate the time for volume synchronization after a transient failure. Initially, we build 1,000 mailboxes (volumes) in three data servers. After injecting a transient failure in one of data servers, we generate the specified number of operations, and then restart the failed data server. Figure 9 illustrates the measured time for volume synchronization. Since the recovery time depends on the number of log entries, the time to receive and replay MO-LOG is proportional to the amount of update operations during the failure.

Figure 10 displays the impact of data server failures on the aggregated OPS in the e-mail workload, where two data servers $D1$ and $D2$ are temporarily shut down and restarted. We repeat the similar experiment twice, one for 64 data servers, and the other for 4 data servers. Each alphabet in Figure 10 represents the following events: (A) $D1$ fails (B) $D2$ fails (C) $D1$ restarts (D) $D2$ restarts (E) $D1$ finishes volume synchronization (F) $D2$ finishes volume synchronization. The lower-case alphabets denote the corresponding events when the number of data servers is four. In case of four data servers, the aggregated performance is slightly influenced by transient failures. However, if the number of data servers is 64, the performance impact of data server failures and volume synchronization is hardly noticeable.

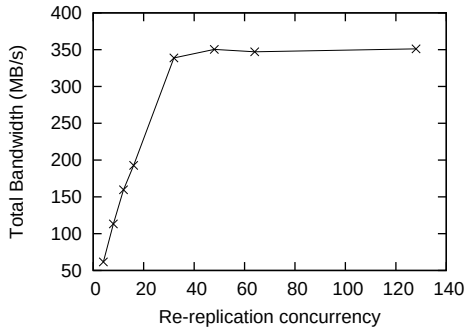


Figure 11: Re-replication performance

7.4 Re-replication

In order to investigate the recovery time from permanent failures, we have performed the following experiments, setting the re-replication concurrency to 32. First, we trigger a re-replication for a data server having 27,287 volumes occupying the total capacity of 205 GB. All the replicas in the data server has been moved to another data server in 10 minutes, with the re-replication bandwidth of 338.67 MB/sec. Second, to study the effect of the re-replication priority, we trigger permanent failures to two data servers, each of which contains 27,000 volumes. Due to these failures, 236 volumes has only a single valid replica and these volumes are re-replicated within 18 seconds. After 1,170 seconds, all the other replicas in the two failed servers are copied to another live data servers. We can see that our prioritized re-replication scheduling policy is effective in reducing the probability of becoming dead volumes.

Figure 11 shows how the re-replication concurrency affects the re-replication performance. The total re-replication bandwidth is increased, but is saturated after the re-replication concurrency reaches 32, which is the half of the number of data servers. This result suggests that a data server should be involved in at most one re-replication instance at a time either as a source or as a destination. Although each data server is equipped with four independent disks, the aggregated disk bandwidth is saturated under the heavy, small-sized disk I/O.

8 Related Work

NAS (Network-Attached Storage) [23] is the simplest storage solution. It is easy to use and its stability has been verified in the industry over years. However, NAS does not provide a location-independent namespace, thus requiring manual intervention during file reallocation for load balancing, failure recovery, or storage capacity upgrade. Moreover, the hardware cost of NAS is relatively high and a NAS head tends to be a performance bottleneck as the storage capacity increases.

The existing distributed file systems running on a cluster of commodity hardware can be classified into two types according to the way replica consistency is maintained. Harp [15], GFS [11], and Ceph [26] employ the primary replica technique [6] which serializes all operations at the primary. It enables the file system to guarantee strong consistency, but tends to have lower availability [7]. This type also requires a lot of communication and synchronous disk writes to maintain commit state among replicas permanently.

On the other hand, Coda [12], Porcupine [21], and Dynamo [10] are based on the optimistic replication technique which allows for more availability with weaker consistency. To guarantee consistency against write conflicts, they exploit application-specific conflict resolver [12, 10], loosely synchronized clock [21], or quorum protocol [10], which is not necessary for immutable files.

Distributed file systems can be also categorized by their metadata management scheme. The centralized metadata management scheme [11, 2] handles metadata operations in a central server. Basically, the centralized scheme makes the system simple and flexible, but the central server can be easily overloaded. Ceph [26] and Lustre [8] try to solve this problem with the metadata server clustering. However, this does not prevent one or more metadata servers from being overloaded under load imbalance or heavy metadata operations.

The decentralized metadata management scheme [12, 21, 10] is an alternative to solve the scalability problem in the metadata server. Dynamo [10] uses consistent hashing to find a node that corresponds to a file similar to peer-to-peer systems. Dynamo randomly distributes metadata operations over all nodes, but actually suffers from load imbalance. In Coda [12], the replication sites of a volume are stored in a volume location database (VLDB) which is mirrored in every server. As the number of volumes increases, the size of VLDB also grows and it becomes expensive to synchronize them among all nodes.

Farsite [5] and Pangaea [22] are proposed as distributed file systems for a wide-area environment. Their main concerns are peer-to-peer architecture, Byzantine failures, security, and narrow network bandwidth, thus they do not fit a cluster environment.

9 Conclusion

This paper presents the design and implementation of BeanFS, a distributed file system developed for a large number of immutable files. BeanFS is not intended to be general storage which offers full POSIX semantics. Instead, BeanFS aims at providing complementary storage just for immutable files, which occupies a significant portion of the total storage capacity required by the recent large-scale Internet services.

BeanFS has a number of features to provide higher level of scalability, availability, and aggregated performance. First, BeanFS uses the volume-based replication scheme in order to reduce the metadata management overhead in the central master. The master server only keeps track of the location information for each volume and the state of each data server. Second, BeanFS

implements a simple and lightweight consistency maintenance protocol for immutable files which does not require complicated primary replica or quorum protocol. All the distributed components see a consistent view of volume states via state matching. Finally, BeanFS recovers volumes from transient failures by volume synchronization and from permanent failures by re-replication. These mechanisms are proven effective in improving the availability of BeanFS.

A BeanFS cluster consisting of more than one hundred data servers has been operational in a production environment without any problem since December 2007.

References

[1] Amazon simple storage service (amazon s3). `http://aws.amazon.com/s3`.

[2] The hadoop opensource project. `http://lucene.apache.org/hadoop/`.

[3] Mysql reference manual. `http://dev.mysql.com/doc/`.

[4] CSIM 19: Development toolkit for simulation & modeling. `http://www.mesquite.com`.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 1–14, 2002.

[6] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 562–570, 1976.

[7] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database System*, 9(4):596–615, 1984.

[8] Cluster File Systems, Inc. *Lustre: A Scalable, High-Performance File System*. `http://www.clusterfs.com`.

[9] comScore, Inc. Worldwide Search Top 10, December 2007. `http://www.comscore.com/press/release.asp?press=2018`.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 205–220, 2007.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceeding of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, 2003.

[12] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer System (TOCS)*, 6(1):51–81, 1988.

[13] International Data Corporation (IDC). The diverse and exploding digital universe, 2007 & 2008.

[14] S.-H. Kim, Y. Lee, and J.-S. Kim. Flexrpc: A flexible remote procedure call facility for modern cluster file systems. In *Proceeding of 9th IEEE International Conference on Cluster Computing (CLUSTER'07)*, pages 257–284, 2007.

[15] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 226–238, 1991.

[16] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM (CACM)*, 29(3):184–201, 1986.

[17] T. Oreilly. What is web 2.0: Design patterns and business models for the next generation of software. (65):17–37, First Quarter 2007.

[18] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[19] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[20] K. Rao, J. L. Hafner, and R. A. Golding. Reliability for networked storage nodes. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 237–248, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 1–15, 1999.

[22] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 15–30, 2002.

[23] W. Singer. NAS and iSCSI Technology Overview. *Fall 2007 SNIA Technical Tutorials*.

[24] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. pages 295–317, 1987.

[25] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1982.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.