The Expandable Network Disk

by

Athicha Muthitacharoen

S.B. Massachusetts Institute of Technology (1999)
M.Eng. Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author .....
Department of Electrical Engineering and Computer Science
August 29, 2008

Certified by .....
Robert T. Morris
Associate Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chair, Department Committee on Graduate Students

The Expandable Network Disk

by

Athicha Muthitacharoen

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis presents a virtual disk cluster called END, the Expandable Network Disk. END aggregates storage on a cluster of servers into a single virtual disk. END's main goals are to offer good performance during normal operation, and efficiently handle changes in the cluster membership.

END achieves these goals using a two-layer design, in which storage "bricks," servers that consist of CPU, memory, and hard disks, hold two kinds of information. The lower layer stores replicated immutable *chunks* of data, each indexed by a unique key. The upper layer maps each block address to the key of its current data chunk; each mapping is held on two bricks using primary-backup replication. This separation allows END flexibility in where it stores chunks and thus efficiency: it writes new chunks to bricks chosen for speed; it moves only address mappings (not data) when bricks fail and recover, which results in fast recovery; it fully replicates new writes during temporary brick failures; and it uses chunks on a recovered brick without risk of staleness.

The END prototype's write throughput on a cluster of 16 PC-based bricks is 150 megabytes/s with 2x replication. END continues after a single brick failure, re-incorporates a rebooting brick, and expands to include a new brick after a few seconds of reduced performance during each change.

The results show that END's two-layer design maintains good performance, resumes operation quickly after changes in the cluster, and maintains full replication of new writes even during a brick failure.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Acknowledgments

This thesis is the result of the collaboration with my advisor, Robert Morris, who was always available for brainstorming ideas, and gave extremely detailed and sharp feedback on every aspect of END. His remarkable dedication to research, his enthusiasm in teaching, and his high standards have left a strong impression with me. I can always count on him for honest advice, whether on technical ideas or career future.

My thesis committee members, Barbara Liskov and Sam Madden, provided valuable insights that improved the presentation of this document. Jeremy Stribling let me take over his machines while running countless hours of experiments. Frans Kaashoek, Sean Rhea, Bryan Ford, Jinyang Li, Russ Cox, Emil Sit, Alex Yip, Chris Laas, Micah Brodsky, and Frank Dabek provided numerous suggestions that improved the design, implementation, and presentation of END.

I am grateful to Frans Kaashoek for taking a chance in letting me join the PDOS research group. I have learned tremendously from him and from every group member, particularly David Mazières and Benjie Chen. Their intelligence, creativity, and most of all, passion have inspired me to pursue systems research.

My graduate studies were funded by an MIT Presidential fellowship, the IRIS project through the National Science Foundation under NSF Cooperative Agreement No. ANI-0225660, an NSF CAREER grant, and support from Google Inc.

Graduate school not only tests one's intellectual abilities, but also one's emotional capacities. I am indebted to many who have supported me along the way.

Jinyang Li endured countless hours of my complaints. Her unending optimism and ability to simplify complex situations is truly impressive. Thomer Gil and Rodrigo Rodrigues provided the European touch to systems hacking. Their friendship has carried me through many tough times. Kati Nybakken and Ramón de los Reyes, through the art of flamenco, taught me how to be expressive in ways that do not exist in scientific research. Hartmut Geyer continues to show me that the love of research and the love of life can coexist harmoniously.

My family has always been there for me. My little brother, whose discipline I admire, has always believed in me, even when I did not. My parents and grandparents had raised me with unconditional love and support. This thesis is dedicated to them, for without their encouragement, I would never have come this far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A storage cluster is a collection of servers connected via a network; each server stores a portion of the data. Storage clusters are attractive to many organizations [23]. A university research group, for instance, might want to use the spare disk space on its cluster of compute servers to store simulation trace files and large data files. Storage clusters that consist of non-dedicated servers also benefit small to medium business organizations [37]. Consolidating data in one place (the cluster) makes information convenient to access, secure, and back up. An ideal cluster storage system would present a disk-like interface compatible with existing file systems (perhaps via iSCSI), and offer the same performance as the underlying disk hardware combined. Such a system would keep running despite the failure of at least a single storage server, recover quickly after a rebooting server comes back on-line, and automatically offer more space to file systems as more physical storage is added.

Previous systems have brought us close to the ideal. Petal [19] distributes data over a set of servers and allows transparent addition and deletion of servers to an existing cluster. FAB [31] showed that quorum replication in a storage cluster can perform competitively with primary-backup replication. Ceph/RADOS [35, 36]

demonstrates how decentralized and self-configurable cluster storage systems can be made as fast as the underlying hardware and highly scalable.

These systems partition the block address space over the servers, so that each server is responsible for storing or replicating the blocks at a particular set of addresses. This scheme offers good performance given a fixed set of servers, because the system can read or write each data block by contacting just one server. However, the scheme makes these cluster storage systems inflexible during change. For instance, in order to maintain the server location mapping of the blocks, and guarantee that block contents are up to date, a significant portion of the servers' data content has to be moved when a new server is added, or when a rebooting server comes back on-line. In a large storage cluster, these changes happen frequently [7, 35, 36].

1.1 Designing Storage Clusters to Handle Change

A server in a storage cluster is composed of CPU, memory, and hard disks, and is referred to as a "brick." Storage clusters that partition data over the bricks by block address face tough challenges during a change to the system. They must handle the three major changes: brick additions, brick failures, and brick recoveries. The following section details how each previous system handles each case.

1.1.1 How previous systems handle change

To add a new brick to the cluster, Petal [19] reconfigures its block address assignment to include the new brick and copies to it data according to the new assignment. Because it can take hours for this process to complete, Petal divides the address range into "old", "new", and "fenced". Addresses in the old or new

region should be served from the set of bricks according to the old or new assignment, respectively. The fenced region is the part where data is being moved, and for which any client requests may require additional messages to get to the right brick. Petal stores each block on a primary and a backup brick. If either brick fails, Petal serves read requests from the brick that is up. A client sends writes first to the primary, and then to the backup if the primary is down. If the primary is the first one to receive a write, it sets the block's busy bit before sending the write both to its local store and the backup. Once the backup recovers, the busy bit tells the primary which blocks to eventually send to the backup. If the primary is down, the backup will receive the write request. Before writing the block to disk, the backup must set the block's stale bit on disk, so that when the primary recovers, the backup knows which blocks the primary is missing.

FAB [31] divides bricks into *seggroups*, partitions block addresses among them, and uses a quorum protocol to perform reads and writes. Thus, only a majority of bricks in a seggroup need to respond before a read or write operation is complete. Each FAB brick keeps, in NVRAM, timestamps of blocks that have been updated on its local disk until all bricks in the seggroup acknowledge the update of those blocks. A new brick in FAB must copy every block in each seggroup to which it belongs. During a single brick failure, FAB continues to serve requests since a majority of bricks in each seggroup is still alive. However, the remaining bricks in each seggroup with a failed brick cannot garbage collect the timestamps of block updates. To discard the timestamps, either the dead brick must restart and collect pending writes, or the remaining bricks must reconfigure to exclude the dead brick from the seggroup. If the dead brick recovers before reconfiguration, it needs to copy from each of its seggroup the recently updated blocks as indicated by the timestamps. If it recovers after reconfiguration and the discarding of timestamps, the recovered brick will have to copy every block in each seggroup to which it belongs, which is the same amount of data a new brick would copy.

Ceph/RADOS [35, 36] stores data objects, as supposed to data blocks, and allows clients to read/write byte extents of those objects. RADOS divides data objects into placement groups and assigns these groups to participating bricks according to the *cluster map*. Within a placement group, RADOS designates one brick to be the primary; all writes start at this brick. The cluster map changes with each addition, failure, and recovery. This prompts the primary of each placement group to figure out how much data needs to be copied to each brick to bring them up to date. The primary accomplishes this by consulting its local log of updates to the placement group and comparing it to the most recent updates on each replica brick.

Because of the assignment of blocks to bricks, these systems must make certain trade-offs when a brick becomes unavailable.

1. They cannot maintain full replication of new writes when a brick is down without copying large amounts of data due to reconfiguration. To maintain the same level of replication of new writes, the system must re-partition the block address assignment among the current bricks, and transfer data according to the new assignment. This transfer of data can result in the portion of the data being transferred becoming unavailable for a significant period. As a result, these systems invoke reconfiguration only when they are certain that the brick is permanently dead, which increases the risk of data loss because the longer they wait, the more likely another brick failure becomes.

2. They need to carefully keep track of updates in order to avoid transferring huge amounts of data to a failed brick when it recovers. There is a danger of running out of NVRAM (in FAB) or log space (in Petal and RADOS) to store recent updates. Because of this concern, these systems cannot wait too long before reconfiguring to exclude the dead brick.

3. Reconfiguration can result in unnecessary work. If a recovering brick joins after records of recent updates to its blocks have been discarded, it will have to copy over the entire portion of data for which it is responsible. A significant portion of the data might already exist on the brick's disk, resulting in unnecessary copying.

Furthermore, expansion of the system is expensive. Adding a new, empty brick to an existing cluster (in Petal), seggroup (in FAB), or placement group (in Ceph) involves copying portions of data to the new brick before it can serve any read/write request of the corresponding data.

This thesis investigates a new design for cluster disks that eliminates all the trade-offs outlined above.

1.2 END: The Expandable Network Disk

This thesis presents END (the Expandable Network Disk). Figure 1-1 shows END appearing to a file system as a giant disk, and accessible through a standard block-level I/O interface; as a result, END can store existing disk file systems (e.g. Linux's EXT2 or BSD's FFS) without modification. END only commits physical storage as blocks are written, so that its apparent address space can be much larger than the amount of physical storage. Though typical disk file systems must be initialized with a particular size, an administrator can create them on END with sizes large enough to accommodate future expansion. Then, as storage needs increase, the administrator can add new bricks and the file systems will automatically have access to more storage. END provides consistent storage to each file system; if a block read returns a value, subsequent reads will not return an older value of the block.

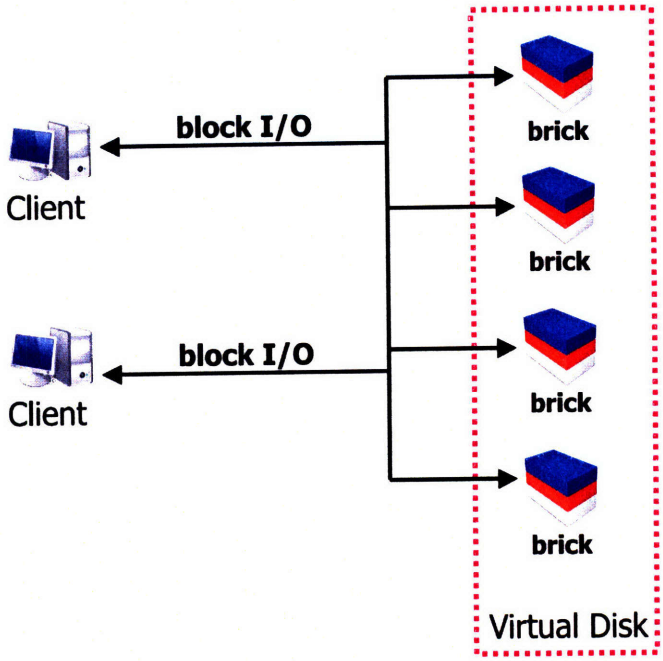

Figure 1-1: **Overview of END:** The data are partitioned among bricks. Many clients can use different parts of the END virtual disk.

END uses a two-layer design, in which each storage brick holds two kinds of information. The lower layer stores replicated immutable "chunks" of data, each indexed by a unique key. The upper layer maps each block address to the key of its current content; two bricks store each mapping with primary-backup replication. The choices of brick to hold a block's address and its data chunk are independent; if the same brick holds both, it is only by chance. This separation gives END the flexibility to store each new chunk on whatever brick is convenient; the immutability of chunks means that END can retrieve a chunk from any brick that has a copy and be sure its content is correct. Throughout this thesis, the upper layer is referred to as the *address map*, while the lower layer is called the *chunk store*.

1.2.1 Benefits of the two-layer design

END's level of indirection helps the system reconfigure quickly during changes. Because chunks are immutable, END can leave them where they are without worrying about consistency. END only needs to move the address map according to the new assignment. Because the address map is orders of magnitude smaller than the chunk store, reconfiguration in END is quick.

This design thus eliminates the problems faced by previous systems when a brick becomes unavailable:

1. END can maintain full replication of data written while a brick is unavailable because it can store the new data chunks on any brick that is alive, and because it can reconfigure quickly to redistribute the small address map and continue to accept new updates.

2. END does not need to maintain a list of updates destined for a failed brick, which simplifies the design and implementation.

3. Reconfiguration is fast. If a failed brick recovers and re-joins the system, the brick only needs to copy over parts of the address map, but not any data chunks. The data chunks on the brick's disks are immediately usable if they are still indexed by END's address map.

4. The two-layer design allows adding a new brick to occur without copying of any data chunks. END only needs to copy parts of the address map. The new brick can gradually pick up load in the chunk store as clients write new data.

1.2.2 Challenges posed by two-layer design

END's two-layer design faces several main challenges:

1. The extra level of indirection means an additional data structure that each read or write has to access, which could reduce performance.

2. Because it avoids the up-front cost of moving data chunks when a brick becomes unavailable, END must make sure that the level of chunk replication is maintained if a brick failure is deemed permanent.

3. Because each new chunk has a unique key, if a block is overwritten, its old data chunk is forever unreferenced. END must have a mechanism to garbage collect these chunks to free up disk space.

END solves the first challenge by laying out the chunk store and address map such that all virtual disk writes result in sequential physical writes to disk, and by batching writes to reduce the number of disk rotations per write. END divides each batch of updates over multiple bricks such that the speed at which the network delivers data to each brick matches the speed at which the brick can write data to disk. To solve the remaining two challenges, END runs a distributed garbage collection algorithm that copies referenced chunks with too few replicas and deletes unreferenced chunks.

An evaluation of END on Linux PCs with inexpensive IDE disks shows that END's performance for small synchronous updates is somewhat worse than a local disk file system because it requires twice as many seeks (to its address map and its chunk store) as an ordinary disk. END's bulk write throughput with 2x replication is 36 megabytes/s with four bricks and 150 megabytes/s with 16 bricks. In an END system that is half-full, a single brick addition, crash or recovery results in a few seconds of reduced throughput due to system reconfiguration. In comparison, adding a new brick to FAB [31] takes 25 minutes without any client traffic. The results demonstrate that END's design provide good performance during normal operation and is efficient during change.

1.3 Contributions

The contributions of this thesis are:

1. A two-layer design for cluster disks that decouples mutable address mappings from an immutable chunk store. This design offers many benefits over conventional designs that map a block address directly to its data. Specifically, it allows:

   - fast and efficient recovery from temporary failures
   - flexible placement of data chunks
   - quick addition of new bricks.

2. A technique that uses disks efficiently, resulting in good performance despite the overhead imposed by the two-layer design: specifically, an on-disk data layout for the chunk store that avoids disk seeks during a batch of writes, enabling the write protocol to write new data chunks at the speed close to that of the underlying disk.

3. A working implementation of END and an evaluation of the implementation in a cluster of bricks. The prototype recovers from a brick addition, crash, or rebooting in a few seconds during each change.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 describes END's basic design and how single reads and writes work, Chapter 3 describes how END efficiently handles changes to the cluster of bricks, Chapter 4 details the techniques use to improve bulk write performance, Chapter 5 sketches the prototype's implementation, Chap-

ter 6 presents an evaluation of the prototype, Chapter 7 compares END to related work, finally, Chapter 8 concludes and lay out future work.

Chapter 2

END Basic Design

The motivation for END arose from the need in our own research group for a reliable storage system that consolidates the spare disk space in compute servers and efficiently handles server down time that comes with using non-dedicated servers. The types of data that we would like to store include big simulation traces and video files.

For END to be useful for the purposes above, it should exhibit the following properties. To be adopted by many users, END should work with existing disk file systems. This means END should expose a block interface similar to normal disks and provide data consistency. To efficiently use disk space, and ease administrative burden, END should expand its capacity as more storage is added in a manner that is transparent to file systems. When there are no failures, it should perform well for large sequential reads and writes, to support the types of workload mentioned above. Finally, it should handle crashes and recovery of bricks efficiently to compensate for END's intended use with non-dedicated servers.

The last two properties, good performance during normal operation and efficiency with respect to change, often conflict and thus are the main challenges in END's design.

To handle change efficiently, END must move a small amount of data during cluster reconfiguration so that it can continue to serve reads and writes of all block addresses. END accomplishes efficient reconfiguration with a two-layer design that consists of an **Address Map**, which maps each block address to the key of its current data chunk (Section 2.2.3), and a **Chunk Store**, which maps chunk keys to immutable 8192-byte chunks of data (Section 2.2.2). When a brick crashes or recovers, END redistributes only the address map, not the chunk store, resulting in fast reconfiguration.

The rest of this chapter explains and justifies the basic design of END. This chapter covers enough material for the reader to understand how END replicates and serves data, and defers details on how END handle brick failures and optimize performance to later chapters. It begins with a description of how to use END, and the environment in which it is expected to operate.

2.1 Usage Scenario and Assumptions

Figure 2-1 shows the basic organization of END. An administrator runs the END server software on a set of bricks. These bricks collectively present a single virtual block address space of any desired size, for example 100 terabytes, regardless of the amount of physical storage currently available. The administrator then runs an END proxy on a client machine; the proxy causes a new pseudo-disk device to appear. The administrator formats a file system on the END pseudo-disk, specifying the maximum size the file system will ever need to be. Finally the administrator mounts the file system. Because END appears as an ordinary disk device, any standard disk file system implementation may be used. The client can make the file system available to applications directly or via a network file system such as NFS. END can support multiple independent clients; they should use different ranges of END's block address space.

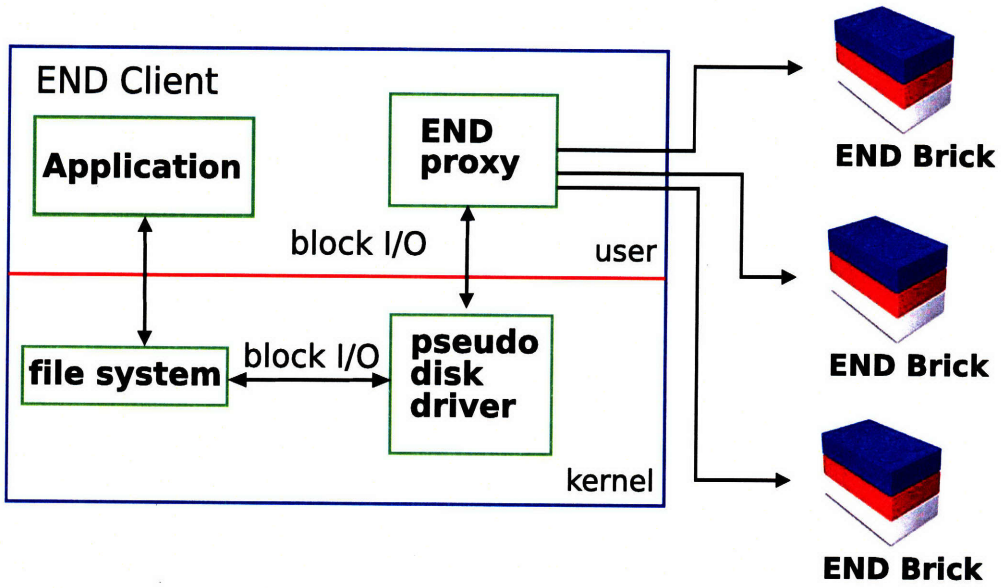

Figure 2-1: **A client's view of** END**:** A disk file system, such as FreeBSD FFS or Linux EXT3, sends block requests to a pseudo disk device, upon which it is mounted. The pseudo disk driver passes block requests to the END proxy, which translates them to END requests, and sends them to END bricks.

END makes a few assumptions about its environment. Its target users are small to medium organizations with non-dedicated servers. It assumes proxies and bricks trust each other and that they are on a LAN protected from attack. Finally, it assumes permanent brick failures are rare.

The END proxy maintains only soft state; all the configuration information and hard state of END are provided by the collection of bricks. Thus, if the client hardware fails permanently, the administrator can replace it and mount the file system stored in END from a new client machine.

2.2 END State

Each END brick stores two important pieces of state that correspond to the two layers of the design, the **Chunk Store** and the **Address Map**. Each brick also

stores a **View** and a **Bucket Map**, which help maintain consistency. The latter also allows END to spread the address map over the bricks, which helps improve performance of bulk reads and writes. This section describes each state in the order of increasing complexity.

2.2.1 View

The view state contains the list of bricks that are live and reachable among each other. A view contains a view number and a list of IP addresses. Each brick keeps a copy of the current view; the bricks agree on the view using the Paxos distributed consensus protocol [17]. The view module on each brick monitors the liveness of each other brick, and announcements of newly added bricks, to decide when a new view is required. Each brick has a unique 8-bit "brick ID" as well as an IP address. This size of the brick ID limits the number of bricks to 256, which should be adequate for a small to medium cluster. If the system administrator anticipates the cluster to grow larger, he can allocate more bits to the brick ID during the initial setup of END.

2.2.2 Chunk Store

The chunk store holds 8192 bytes of data corresponding to each chunk key. The details of chunk keys are an important part of the design because they provide END with its flexibility. Chunk keys have the following properties:

- A given chunk key always maps to the same chunk contents (chunks are immutable).
- In failure-free operation a client proxy can look at a key and guess correctly which brick stores the chunk. Under more adverse conditions the chunk with

Figure 2-2: **Details of a 64-bit chunk key.** The first byte contains the ID hint of the brick to which the writing proxy first sent the chunk. That brick assigns the remaining bits sequentially, as chunks are created.

a given key can be stored on any brick and found by client proxies after a search. In the common case, even with failures, a proxy needs to look up only one brick to find the key.

- Chunk keys are small (8 bytes) to help indices that contain them fit in memory; this is particularly important for workloads which use chunks in random order.

- Chunk keys can be freely chosen (modulo uniqueness) to allow flexible control over what bricks store a newly written chunk, and to allow flexible placement of the chunk and its meta-data on a brick's disk.

Figure 2-2 illustrates a chunk key, which is an 8-byte number. The high bits contain the "ID hint," the ID of the brick to which the writing proxy first sent the chunk (the ID hint is 8 bits in the prototype); a chunk key's ID hint indicates a good brick for later readers to ask for the block. That brick assigns the remaining chunk key bits. This arrangement lets the proxy decide which brick should store a newly written chunk, and lets the brick ensure that the new key is located in an efficient place in its index: the brick assigns keys sequentially so that sequentially created chunks have locality in the brick's B-tree index of keys (Section 4.1). Replicas of a chunk are initially stored on the bricks whose IDs follow that first brick, where reading proxies can easily find them. This assignment of chunk keys to bricks is similar to consistent hashing [15].

2.2.3 Address Map

The address map contains the current chunk key of the data for each block address. Each address is block-aligned with a block size of 8192 bytes. The address map is partitioned among the bricks into buckets according to the bucket map (see below). A "bucket" is a data structure that contains a portion of the address map. It is the unit of replication for address mappings. There are two copies of each bucket of the address map, on a primary brick and a backup brick. The primary serves all operations (lookups and updates), sending each update to the backup. The address map is sparse: blocks that have never been written have no mapping.

Each END brick uses the current view number to prevent delayed write RPCs from previous views from updating the current view's address map. Each END brick also keeps old buckets until they have been successfully copied in the next view by both the primary and backup.

2.2.4 Bucket Map

The bucket map partitions the block address space into buckets, and assigns each bucket to a primary and a backup brick which serve the address mappings for that bucket. The bucket map partitions the addresses according to their middle bits: all addresses with a particular value in their middle bits are in the same bucket. END uses the middle bits so that ranges of successive addresses are grouped together in one bucket, which helps throughput with single clients, and successive ranges are striped across buckets, which helps throughput when concurrent clients use different parts of the virtual disk.

There should be about an order of magnitude more buckets than bricks to allow the assignments to be balanced as bricks come and go, as in Porcupine's user map [30]. Every brick has a copy of the bucket map. There is a new bucket map

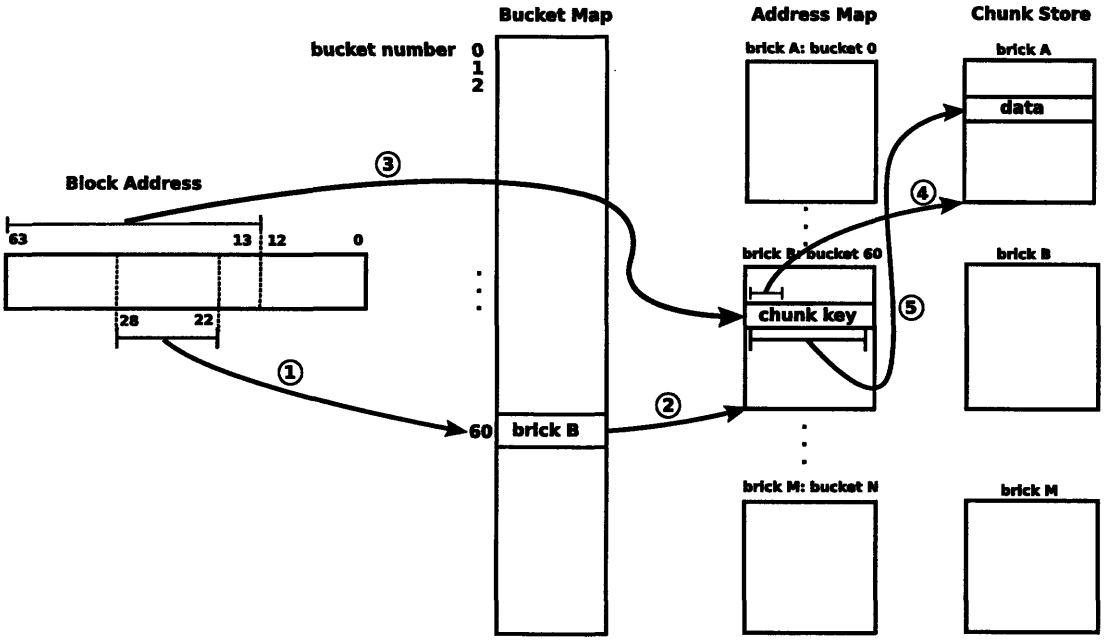

Figure 2-3: **Block Address Translation.** Given a block address, (1) the middle bits identify which bucket contains the address mapping. (2) The Bucket Map tells which brick is the primary for that bucket in a given view. (3) The upper 51 bits are used to look up the chunk key for this address from the corresponding bucket at the primary brick. (4) The high bits of the chunk key provide a hint of the brick at which the chunk is stored. (5) Finally, that brick uses the whole chunk key to locate the chunk on its disk.

associated with each new view. Each view's bucket map is derived from that of the previous view, with changes in response to bricks joining and leaving. Each new bucket assignment spreads the number of buckets roughly equally among bricks while minimizing the movement of buckets.

Figure 2-3 illustrates the translation of a block's address to its data.

The END proxy caches a copy of the view and the bucket map. These two tables are enough for the proxy to send address and chunk RPCs to the correct brick in most cases. If the proxy's information is out of date, any brick it talks to will reply with an error telling it to fetch new information.

2.3 Read Protocol

Figure 2-4 illustrates events that occur when a file system issues a block read request to the local END proxy. The proxy looks in its cached bucket map to find the primary brick that holds the bucket containing the block's address. The proxy asks that brick to look up the address in its address map. If there is no entry, the proxy returns a block full of zeroes, approximating the behavior of the first read of a block on an ordinary disk. Otherwise, the address map entry indicates the key of the block's current contents. The proxy uses its cached list of bricks in the view to find the chunk key's successor brick, and asks that brick for a copy of the chunk. If that brick doesn't have the chunk, the proxy tries the following bricks in ID order.

Even if the brick that originally stored the chunk is not available, the proxy will usually be able to retrieve the chunk with one RPC once it knows the chunk's key: this is because the chunk's replicas are stored at bricks with successive IDs. A proxy might have to issue multiple chunk retrieval RPCs if the original brick has died, and new bricks with IDs just after the original brick have joined. To

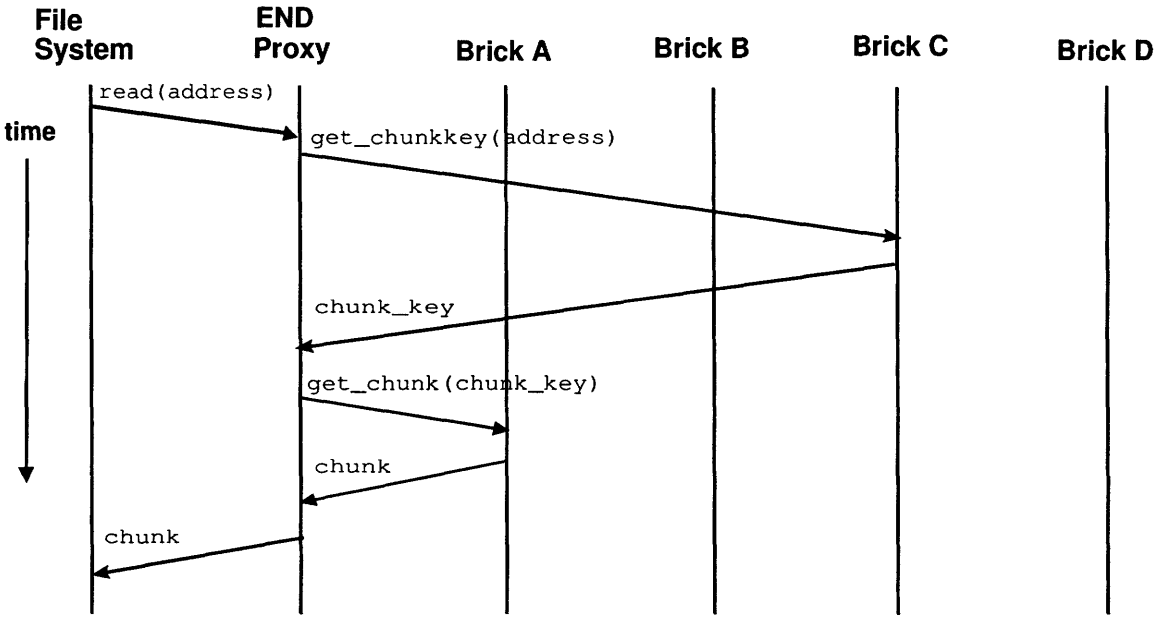

Figure 2-4: END **Read Protocol:** The proxy receives a read from the file system, and satisfies the request by looking in its local bucket map for the primary brick with the bucket that contains the address (brick C, in this case). The proxy asks brick C for the chunk key, which precedes brick A's ID. Finally, the proxy requests the data chunk from A.

reduce the number of retries a proxy has to issue, a system administrator can run the replica maintenance process (Section 3.4), which copies chunks to bricks with successive IDs.

2.4 Write Protocol

A single write to END involves updates of the replicated address map on two bricks, and insertion of replicas of the chunk data at two more bricks. In Figure 2-5, when an END proxy receives a block write request with a block address and data chunk, it first updates the chunk store by sending the chunk to an arbitrary brick. The proxy updates the chunk store first because the data chunk must be written to disk before an address map entry can point to it, to prevent dangling references.

In Figure 2-5, the proxy sends the chunk to brick A, which assigns the chunk key by picking a key that results in the most efficient update of its internal data structures, see Chapter 4. To replicate the data chunk, brick A sends the chunk, along with the newly assigned chunk key, to the brick whose ID follows A (brick B, in this case). This scheme helps reduce the bandwidth usage at the proxy, allowing it to send more data chunks to bricks during bulk writes.

A proxy sends the chunk to a brick in two phases: append and flush. The two phases help improve bulk write performance and are explained in Chapter 4.

After the data chunks are written to disk, the proxy updates the address map state by calculating the bucket number to which the block address belongs, and looking up the primary brick for that bucket from the Bucket Map. The proxy then updates the address map state using primary-backup replication. The process of updating the address map also has two phases, similar to those used to update the chunk store.

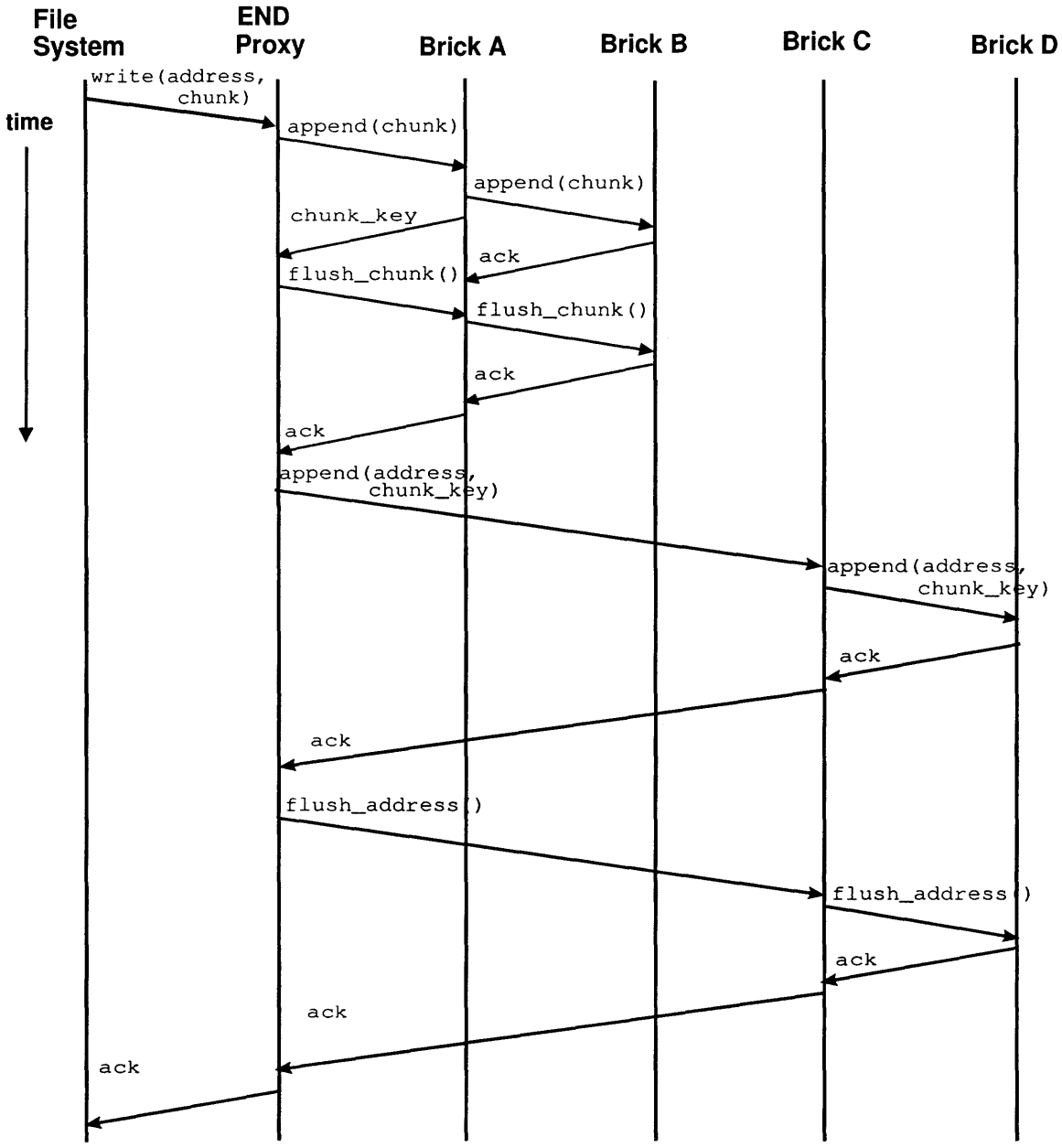

Figure 2-5: END **Write Protocol:** The proxy receives a write from the file system, and sends the data chunk to brick A, who picks a chunk key, and sends it along with the chunk to brick B. The proxy, after receiving the chunk key from brick A, tells A to flush its in-memory chunk buffer to disk. Brick A also tells brick B to flush its chunk buffer. The second half of the protocol concerns updating the address map entry. The proxy updates the address map entry using primary-backup at bricks C and D.

Chapter 3

Handling Change

There are three main goals for END's handling of brick failures. The first is to ensure that there are always two copies of every bucket (i.e., every part of the address map), and that those two copies are identical. END can afford to be less careful with the chunk data because, being immutable, a chunk cannot suffer from inconsistency. The second goal is for END to resume operation quickly after a brick failure. END must transfer just enough data to satisfy the first goal, and still be able respond to client requests without a long delay. The final goal is for END to reuse data that is still valid on a brick that has recovered from a crash.

3.1 Brick Reconfiguration

END bricks participate in a reconfiguration process whenever there is a brick addition, a crash, or a recovery in the cluster. The reconfiguration process makes sure that all the live bricks agree on the current view of the system, and that the address map buckets are replicated at the right primary and backup bricks. There are four END modules that interact during reconfiguration: Heartbeat, View, Bucket Map and Address Map. The **Heartbeat** module on each brick periodically sends

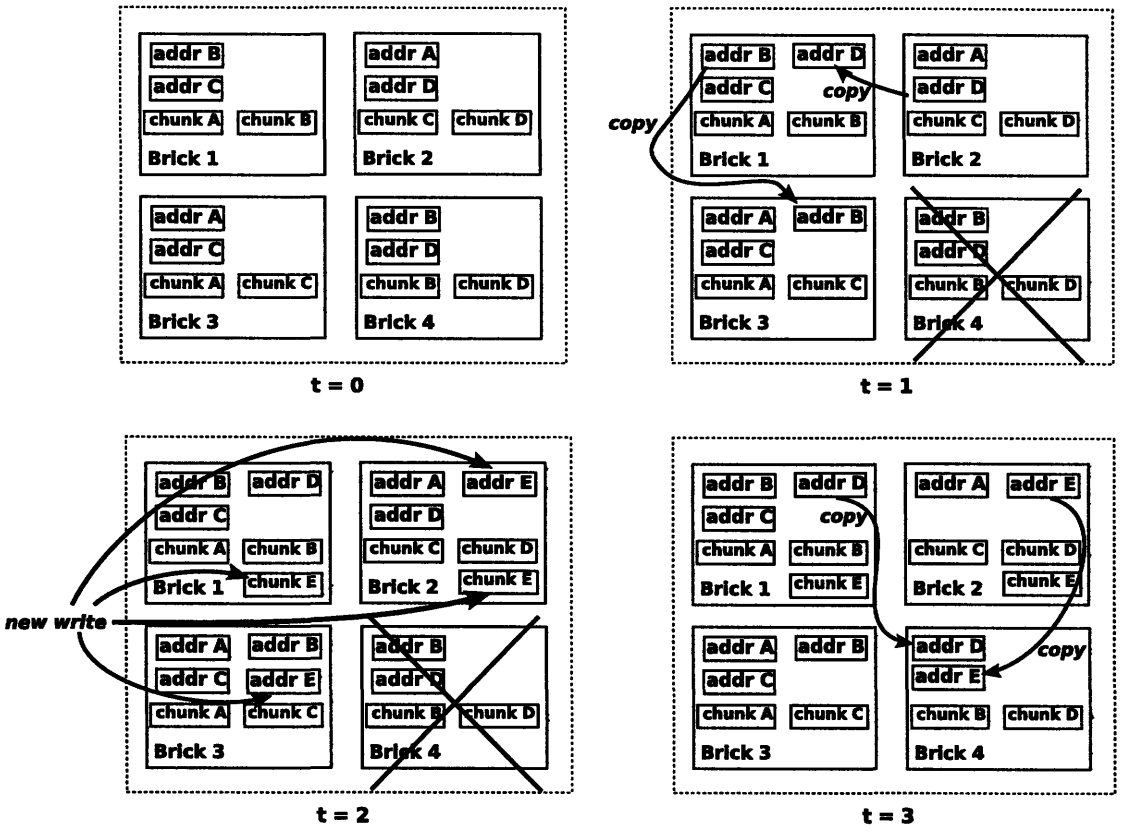

Figure 3-1: END **Reconfiguration.** Brick 4 becomes unavailable at t=1, at which point the system makes extra copies of parts of the address map that were assigned to Brick 4. At t=2, the proxy writes a new block, and the system stores both copies of the block's data chunks and address map entries on the remaining bricks. At t=3, Brick 4 comes back online; the system re-assigns a portion of the address map to it. The data chunks that are on Brick 4 are still referenced, making them fully-replicated without the need to re-copy.

heartbeat messages to all bricks and maintains a list of all responding bricks. If the **View** module on any brick sees that the list of responding bricks is different than the current view, it initiates a Paxos instance [17] among the bricks in the current view to make sure that the responding bricks agree on a new view.

After a view change, the **Bucket Map** module computes a new bucket map. The new bucket map contains a primary and a backup brick for each set of block addresses, chosen from the bricks in the new view. If a bucket's primary and backup from the old view are in the new view, they are chosen as the bucket's new primary and backup. If only the old primary is in the new view, a new backup brick is chosen. If only the old backup is in the new view, it is made the primary, and a new backup is chosen. In both cases, the brick with the fewest bucket assignments is chosen. Nevertheless, there is an exception to the previous rules; an old primary or backup might not become primary or backup in the new view if it is assigned more than the average number of buckets per brick. In this case, END picks the next brick that has fewer that average number of buckets assigned. If neither the old primary nor the old backup is in the new view, the bucket cannot be used unless one or both come back to life with disk contents intact.

The **Address Map** module must copy address mappings in response to changes in primary/backup assignments in the bucket map. If a brick sees that it is the new primary for a bucket, it tries to get a copy of the bucket from the primary from the previous view (which it is very likely to be itself); if the old primary is not in the current view, it gets the bucket from the old backup. A brick does not start serving address map **get** RPCs until it has a copy of the relevant bucket. If a brick is no longer responsible for a bucket, it must save a copy until both the primary and backup in the new view have a copy. After this condition is met, a brick can discard its old buckets, and respond to any request for them with a hint to fetch a new bucket map.

When a new brick joins END, it initiates a view change to a new view that includes it. The new view's bucket map will assign some buckets to the new brick to balance the load. If the new brick was previously part of END and its chunk database is intact, the chunks that it holds will be immediately usable if address map entries still refer to them.

If a new address map **append** RPC arrives at a brick while it is still fetching the corresponding bucket, the brick buffers the address map in-memory, instead of inserting it into the bucket, and returns to the proxy as in a normal append. Once a bucket fetch is complete, the brick goes through the append buffer and inserts any relevant address map entries to the bucket. If an address map flush RPC arrives and there are still pending address mappings in the buffer, the brick writes the buffer to disk. Section 5.2 describes how END uses that buffer should a brick crash and recover.

If an address map **get** RPC arrives and the corresponding bucket is still being fetched, the brick delays the reply until it completes the fetch.

Figure 3-1 illustrates the reconfiguration process. At t=1, Brick 4 becomes unavailable. END reconfigures as a result, making extra copies of Brick 4's address mappings, but not its data chunks. At t=2, new writes are fully replicated on the reconfigured system. At t=3, Brick 4 comes back online, and is re-incorporated. END moves some address map buckets to Brick 4, whose existing data chunks are immediately usable.

3.2 Proxy Behavior during Reconfiguration

When the proxy receives a timeout on one of its requests to a brick, or when a brick indicates in an RPC reply that it is undergoing view change, the proxy has to restart any failed operations. It does so by re-fetching the bucket map from a

random brick and restarting failed operations.

3.3 Consistency

END provides consistent storage to each proxy: if a block read returns a value to the proxy, subsequent reads of that block will not return older values. END provides consistency for each block by maintaining two invariants:

Invariant I Once the address map has returned a chunk key k for a `get` RPC of address a, future `get` RPCs of a will either yield k, or a newer chunk key, or that the entry is not found.

Invariant II Once the chunk store has responded to an `append` RPC of a chunk c with a chunk key k, future `get` RPCs of k will always yield chunk c, or that the chunk is not found.

Invariant II holds because there is only one data chunk associated with each chunk key. This is true because when a brick receives a new data chunk, the brick assigns the chunk a unique chunk key. Below is an explanation of how END maintains Invariant I:

1. The Paxos protocol ensures that there is only one view with view number v, because a majority of bricks in view $v-1$ has to agree on view v.

2. Within a view v, there is at most one primary and one backup for each address map bucket.

3. At the beginning of a view v, the primary p for an address map bucket A, copies A's content from view $v-1$. There are two cases:

   (a) If the primary for bucket A is the same in both views v and $v-1$, then it uses the same bucket A.

(b) Otherwise, the primary p checks to see if the primary p' from view $v-1$ is in view v. If so, p copies bucket A from p'; otherwise, p copies bucket A from A's backup b' in view $v-1$. If neither p' nor b' are in view v, the address map bucket A cannot be used.

4. At the beginning of a view v, before responding to any `get` RPC, a primary makes sure that the content of its and the backup's address map buckets are identical. There are two cases:

   (a) If the primary and backup bricks are the same as in view $v-1$, the primary sends all unacknowledged address mapping updates to the backup.

   (b) If the backup is different than the one in view $v-1$, the primary waits until it has finished sending the address map bucket to the backup.

5. During the lifetime of a view, the primary keeps a list of address mappings that have not been flushed to disk at *both* the primary and backup. The primary delays processing a `get` of an address in a pending update until both the primary and backup have flushed that update to disk.

Conditions 1. through 5. ensure that there is one view with number v, that at the beginning of each view, there are two identical copies of an address map bucket, and that the primary answers a `get` RPC only when both copies of the corresponding address mapping are identical. These guarantees ensure that Invariant 1 holds.

Invariants 1 and 2 together imply that once a proxy has read a block value, subsequent reads will never return older values.

3.4 Replica Maintenance and Garbage Collection

Periodically, END must check that every chunk has enough replicas, and free the storage consumed by chunks to which no address map entry refers. END integrates these two tasks because both need to know which chunks are still referenced and how many copies exist.

Each brick is responsible for garbage collecting and checking the replication of chunks of which it is the successor, and thus must learn of chunks in that range of which it has no copy, and of references from address mappings to chunks in that range. To accomplish this, each brick consults its indices to form a list of the chunk keys it stores and a list of the chunk keys mentioned in its portion of the address map. It sends, to each other brick, the keys from these lists for which the other brick is the successor.

Now each brick knows, for every chunk key for which it is the successor, which bricks have a copy of that chunk, and whether any address map entry refers to the chunk. It sorts these lists and compares them, forming a list of chunk replicas that can be deleted because there are too many replicas (or the chunk key is not referenced), and a list of chunk keys that have too few copies. The brick splits these lists according to the brick that should take action, and sends the sublists to those bricks.

If a brick needs to store a replica of a chunk, it finds a copy and fetches it. Chunk deletion is more involved, and is similar to LFS's segment cleaning [28]. A brick merges all the delete lists it receives. Recall that a brick stores chunks in multiple log files. It calculates the fraction of space that should be deleted in each log file, and ignores delete requests for log files that are more than half live. For each log file that is less than half live, the brick copies just the live chunks to a

new log file. It then updates the chunk index to reflect the live chunks' new offsets in the new log file, and finally deletes the old log file. The brick records the fact that it is cleaning the log file on disk so that, if a crash and reboot should occur, the recovering brick will know to fix its chunk index by re-scanning the log file.

The replica maintenance and garbage collection scheme is invoked manually, and involves all bricks in the cluster.

Chapter 4

Write Optimization

Because of the two-layer design, normal writes to END have to update two different data structures, and thus have the potential to be slow. END speeds up writes by laying out the chunk map in a log structure on disk (Section 4.1), batching and delaying writes (Section 4.2), and striping batches of chunk and address map updates among bricks (Sections 2.2.4 and 4.2.1).

4.1 Brick On-Disk Representation

File systems designed for disks expect that when the hard drive reports that a write has finished, the written data is stable on the hard drive and will survive a power failure. END must provide the same semantics in order to allow correct crash recovery of the file systems it stores. For short random synchronous writes this means that END risks having twice the delay of a conventional disk file system, since END might have to wait for a random I/O to the chunk store and then another to the address map for each write. However, if the file system sends writes in bulk, it does not care which order these writes become durable, and is usually expecting high throughput from the disk. For these reasons, it is important for

END to perform well with bulk write workloads.

The rest of this section describes how END lays out the chunk store and address map on disk to avoid disk seeks during each write, and perform close to the speed of the underlying disk during sequential bulk writes.

An END brick stores chunks in a log-structured database inspired by Venti [25]. The log structure lets the brick store writes that it receives sequentially in time at locations that are sequential on disk to avoid seeking. Since data that are written sequentially are often read sequentially, the log structure enables fast reads as well. In Figure 4-1, a chunk store consists of several append-only log files and an index file. Each log contains up to a million entries. END maintains multiple log files to reduce the cost of cleaning during garbage collection (Section 3.4). The logs are the permanent home of all chunks. The index contains, for each chunk key, which log contains the chunk data and its offset within the log. Each log entry contains a chunk key and an 8192-byte chunk. The index database can be regenerated from the log files, which is why the logs contain keys as well as values. Because the chunk index can be brought up to date from the logs, a brick need not update it on the disk for every write; it can write a snapshot of the index to disk periodically.

An END brick stores address map entries in a Berkeley DB4 database [33], because each entry is small enough that a working set of address maps fits in memory. Each brick stores each of its address map buckets in a separate database to help speed up the transfer of buckets during reconfiguration (Section 3.1). Berkeley DB4 includes logs to implement transactions and recovery. END wraps a transaction around as many address map inserts as possible before committing, and flushes only the DB4 log to disk after the commit. As a result, it can defer updates to DB4's B-tree file on disk.

Because of the on-disk data layout of the chunk store and address map, a sequence of file system writes (whether to random or sequential block addresses)

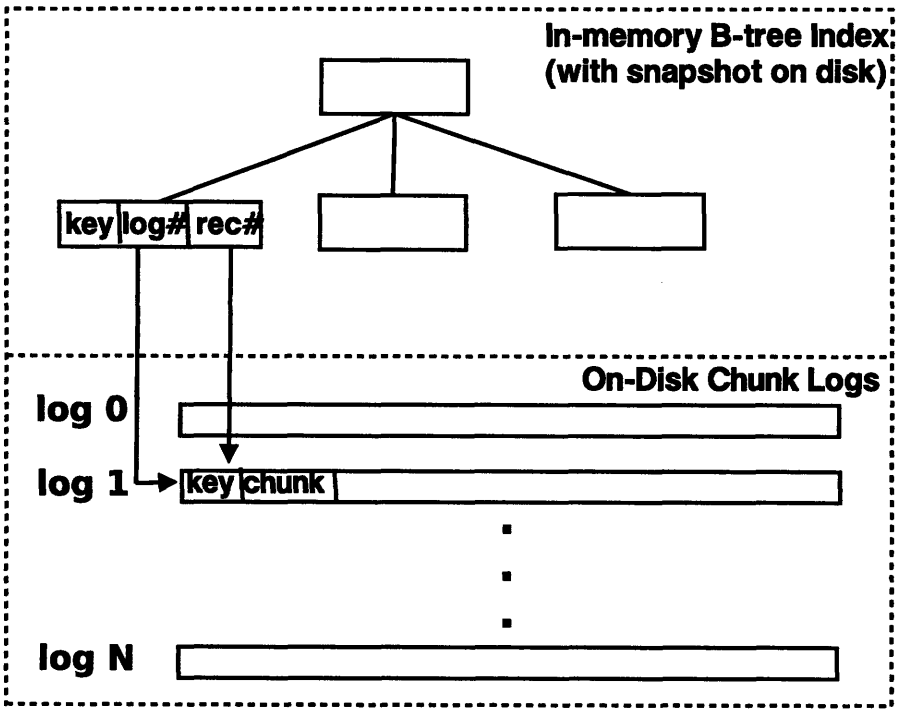

Figure 4-1: **On-disk data layout for the Chunk Store.** A Berkeley DB4 index maps each chunk key to a log file number and an entry within that log. A log file entry contains both the chunk key and the 8192 bytes of chunk data. END keeps the chunk key in the log to help rebuild the DB4 index when a brick recovers after a crash.

only needs to involve disk writes to append to the logs. In principle these appends need not involve seeks. The next section explains how END minimizes rotations, in addition to seeks, for multi-block writes.

Stepping back, END has many levels of indirection: the bucket map, the address map, the chunk store index, the chunk store logs, and the file system on the brick's disk that maps these files to the physical disk storage. A mapping affects performance if it either require synchronous disk writes to update, or is too big to have a high cache hit rate under a random workload. The design outlined above results in only the address map DB4 log and chunk store log requiring synchronous update (though both can use group commit), and all but the chunk store log are likely to fit in memory for caching.

4.2 Write Protocol

This section describes the algorithm END uses to ensure that large sequential writes have high throughput, an area of particular concern as explained in the previous section.

During a large sequential workload, the file system sends many concurrent writes to the END proxy, giving END the opportunity to manage the batch as a whole for efficiency. In order to allow bricks to append multiple entries to their logs per rotation, the proxy/brick protocol has separate RPCs to send data to be appended to a brick's in-memory log buffer and to flush the log buffer to disk. A brick replies to an append RPC immediately, but waits for the disk flush to complete before replying to a flush RPC. This behavior applies to both the chunk store and the address map. Hence, there are four phases in END's write protocol. The rest of this section describes how END divides up data chunks among bricks, and follows with an explanation of each write phase.

4.2.1 Placement of Data Chunks

Once the END proxy receives a batch of writes, it has the opportunity to choose where to store the chunks of a batch to maximize performance. One consideration is that the fewer disks a batch is spread across, the fewer disk seeks and rotations are required, which increases total capacity if there are many active file systems sharing the END cluster. On the other hand, the single-disk maximum throughput is about a quarter of the rate a single proxy can send data over the network (26 megabytes/s vs 117 megabytes/s in our setup). In order to both maximize single-proxy performance and maintain high total capacity for multiple proxies, a proxy spreads each batch of writes across four bricks. END partitions the bricks into groups of four, and a proxy chooses a random group for each batch; this increases the chances that multiple proxies will use different groups at different times and thus obtains good parallelism from the bricks. The level of indirection provided by the chunk keys allows END the required flexibility in where to place the chunks.

4.2.2 Write Phases

The following paragraphs explain each of the four phases in END's write protocol. The figures that accompany each phase shows details on one of the bricks involved in the write. A more complete diagram involving all bricks is presented at the end of this section.

Phase I: Sending Data Chunks In Figure 4-2, when the file system sends a write to the proxy, the proxy sends the data chunk in an append RPC to a brick A, chosen randomly from the current group it is using. Brick A assigns the chunk's key, including its own ID as the hint ID, and increments low bits by one higher than the previous key that it has assigned. After appending the chunk and its key to A's in-memory log buffer, and updating

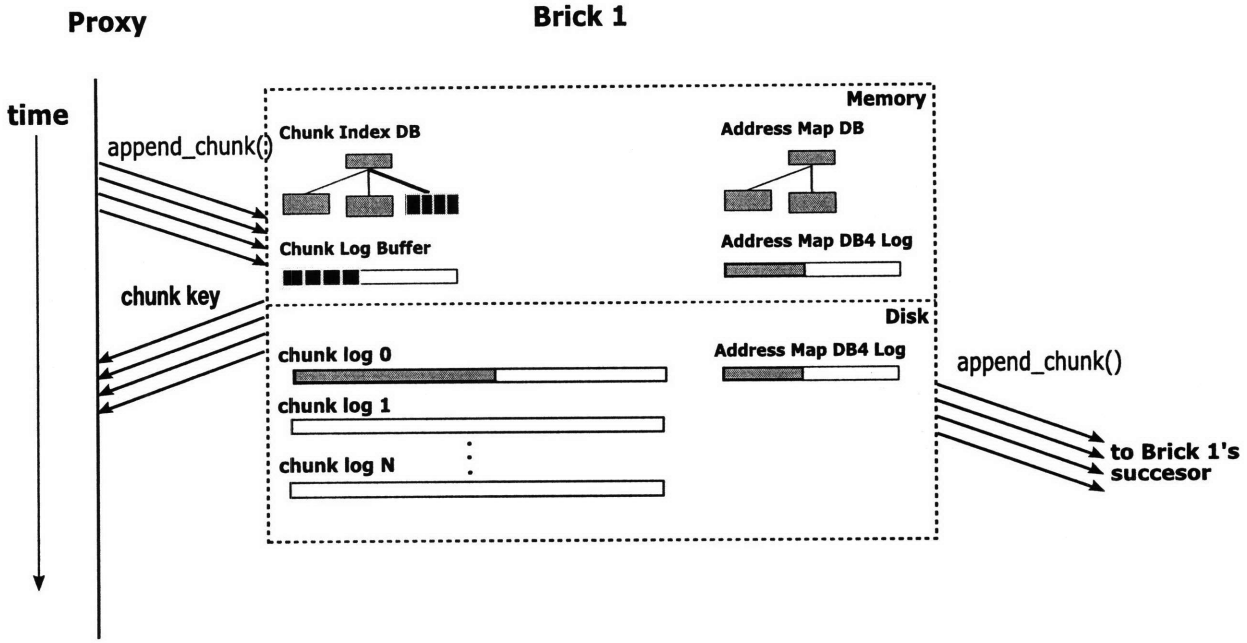

Figure 4-2: END **Write Phase I: Sending Data Chunks** The proxy sends a batch of chunks to a group of bricks, only Brick 1 is shown for simplicity. The diagram shows the internal content of Brick 1's memory and disk when it receives the chunks. Brick 1 appends the data chunks (in black) to its log buffer (which was initially empty), updates its chunk index, sends the chunk keys to the proxy, and finally sends each data chunk to a replica brick.

its in-memory chunk index, A includes the key in its reply to the proxy, and sends a copy of both the key and chunk to the brick's successor brick. This scheme spreads the network load of sending the second copy of each chunk over multiple bricks, avoiding a bottleneck at the proxy.

If the application is performing large sequential writes, the proxy will typically get many writes in a row from the file system, and will thus send many chunks before proceeding to Phase II.

The proxy picks a new group of four bricks each time it is idle (has no outstanding RPCs), or after it finished sending a threshold number of chunks to a group (1,024 chunks, in the prototype).

Phase II: Flushing Data Chunks Once the proxy has no outstanding data append RPCs to a particular brick because the brick has replied to all Phase I RPCs, sends the brick a data flush RPC. Figure 4-3 shows the steps to write data chunks to disk. The brick writes its in-memory chunk log buffer to disk, waits until the data is on the disk, and replies to the RPC. The proxy does not explicitly cause the second copy of each chunk to be written to disk; these writes occur as a side-effect of first-copy flushes, and the bricks' periodic flushing of their chunk log buffers. END assumes that the latter flushing happens often enough such that the period where there is only one persistent copy of a chunk is small.

Phase III: Sending Address Mappings When a brick has responded indicating that it has flushed the data chunks to disk, the proxy sends append RPCs for the block addresses of the chunks the brick has flushed. The destinations of these RPCs may include multiple bricks depending on the block address range. If the number of RPCs is fewer than the system's default batch size (1,024 in our prototype), and the addresses are consecutive, then there is usually just one destination brick. END enforces the batch size by using a

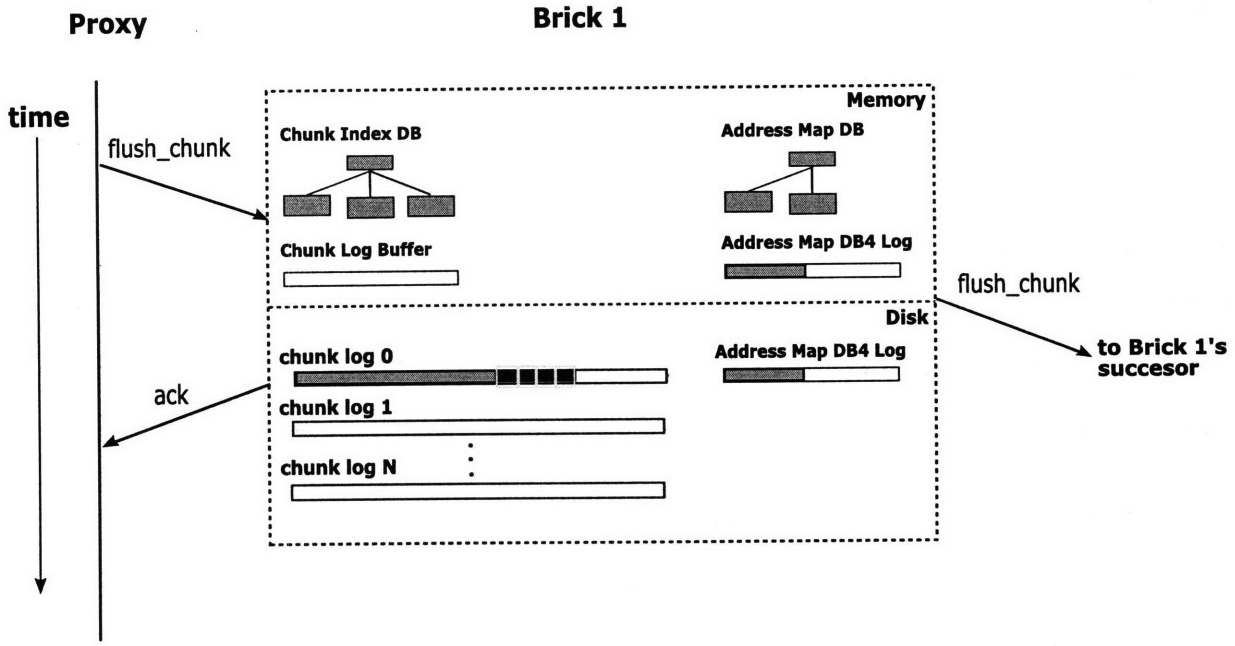

Figure 4-3: END **Write Phase II: Flushing Data Chunks** Once the proxy receives acknowledgments for all data chunks that it sent to Brick 1, it tells it to flush the chunks to disk. Upon receiving the RPC, Brick 1 sends a flush RPC to its replica, writes the content of its chunk log buffer to disk (in black), and replies to the proxy.

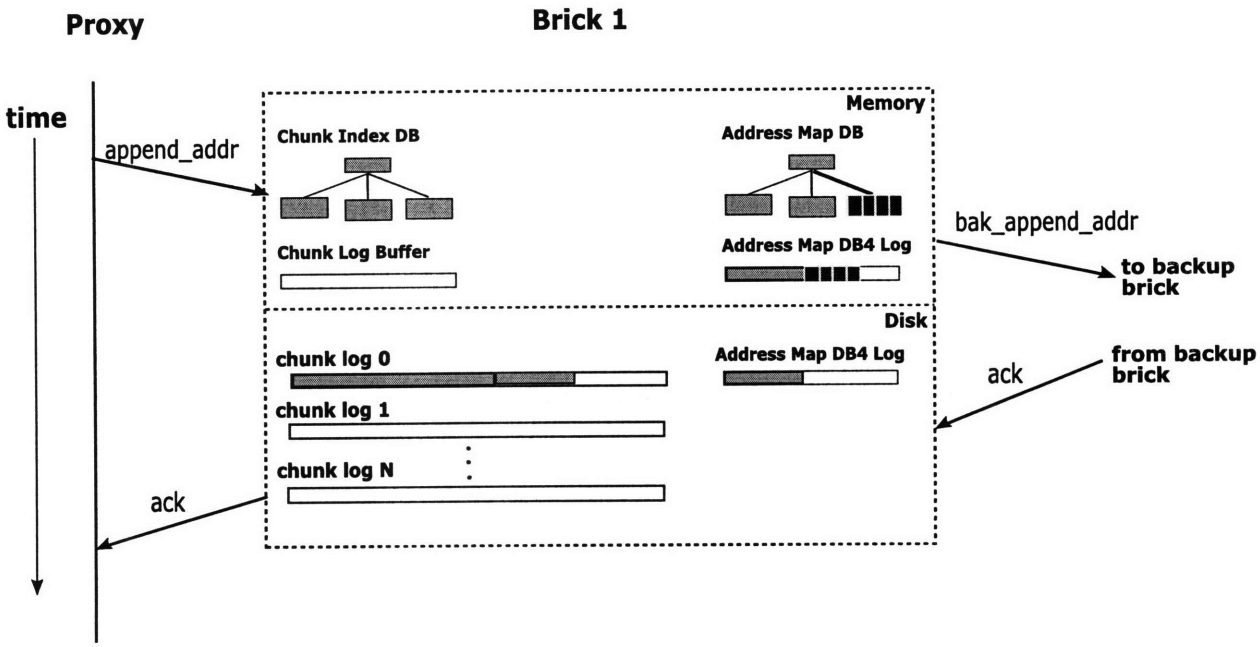

Figure 4-4: END **Write Phase III: Sending Address Mappings** The proxy knows that at least one copy of each data chunk in the batch is persistent on disk, so it asks the primary (Brick 1, in this case) to update the mapping between the address and the chunk keys. The diagram shows Brick 1 updating its in-memory address index database (shaded in black), and then sending an RPC to the backup brick to update its address map. Once Brick 1 receives a successful reply from the backup, it replies to the proxy.

certain number of bits in the middle of a block address to determine to which bucket an address belongs (10 bits, in this case). Figure 4-4 shows that the proxy sends append map RPCs only to the primary of the relevant buckets. For each append map RPC, the target primary starts a DB4 transaction (if there is none on-going), inserts the address mappings to its bucket database, and sends an RPC to the relevant backup. The backup inserts to its database and replies to the primary, which replies to the proxy.

To avoid inconsistency from reordered or lost address map append RPCs from the primary to the backup, the primary keeps a list of mappings that have not been flushed to disk at *both* the primary and backup. The primary delays processing a read request for an address in a pending update until both the primary and backup have flushed that update to disk. This policy prevents the primary from replying to a read request for a block whose latest write has not been acknowledged by the backup, and which might be lost.

Phase IV: Flushing Address Mappings In Figure 4-5, once the proxy has no outstanding append map RPCs to a particular brick, it sends it an address flush RPC. The target brick may have uncommitted address map inserts that lie in more than one bucket. The brick sends an address flush RPC to the secondary for each such bucket (usually just one); each secondary commits any on-going transaction and flushes the DB4 log to disk, waits for the flush to finish, and replies to the primary. The primary commits its transactions, flushes the log to disk, waits for the secondaries, and replies to the proxy. At that point the proxy responds to the file system that the corresponding writes have completed.

Figures 4-6 and 4-7 present a higher level view of END's write protocol. It includes all the bricks involved in a batch of writes.

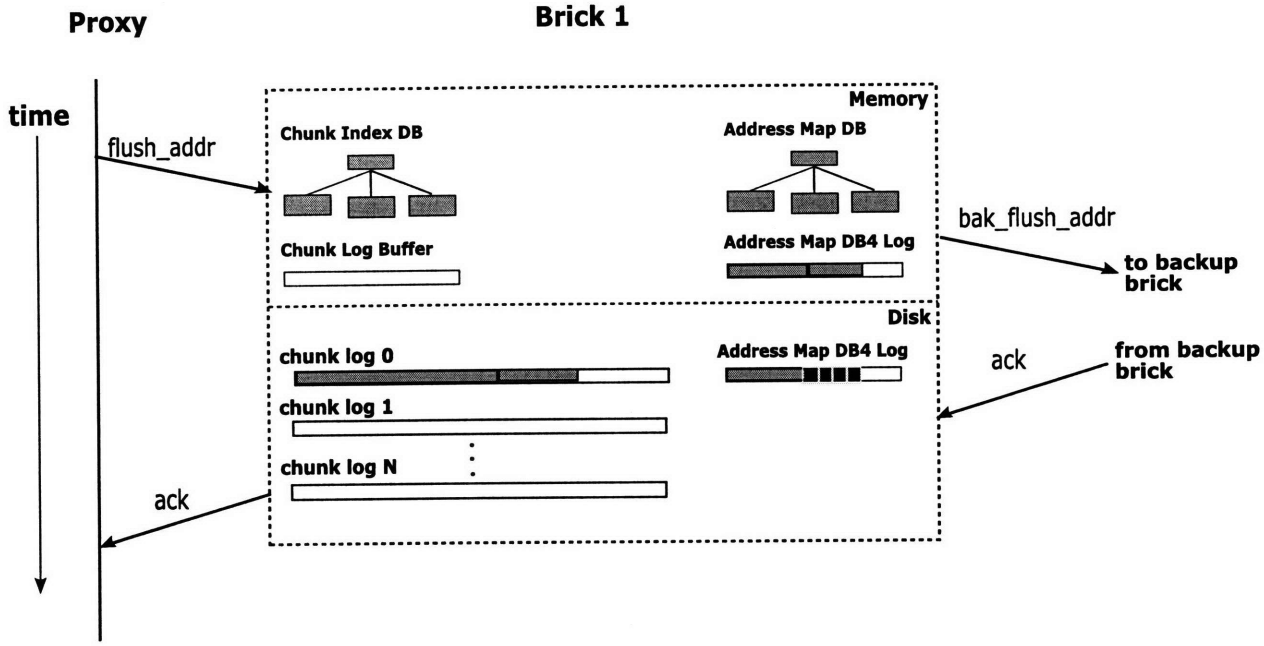

Figure 4-5: END **Write Phase IV: Flushing Address Mappings** The proxy tells Brick 1 to write the changes to its address map to disk. Brick 1 sends an RPC to a backup brick to flush its address map changes to disk, and tells its local address DB4 to flush its log file to disk (in black). Once Brick 1 receives a successful reply from the backup, it can reply successfully to the proxy.

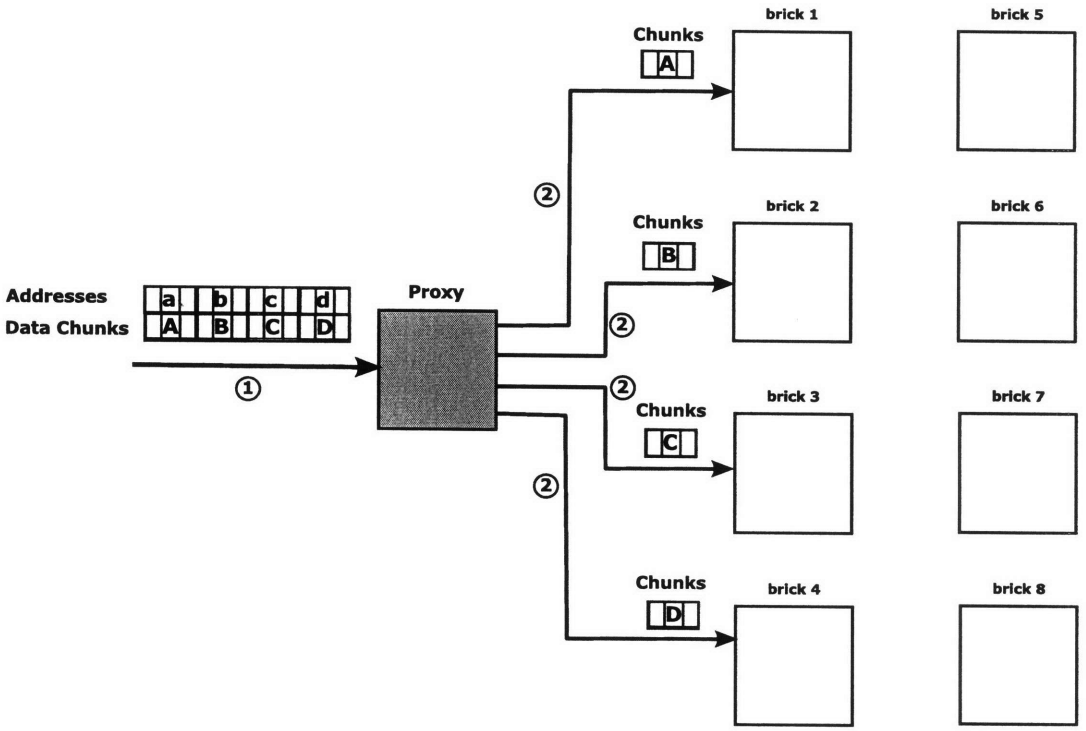

Figure 4-6: END **Write Phases I and II:** (1) When the proxy receives a batch of writes from the file system, it picks a group of bricks (four bricks per group, in this case), and (2) sends an equal number of data chunks to each brick.

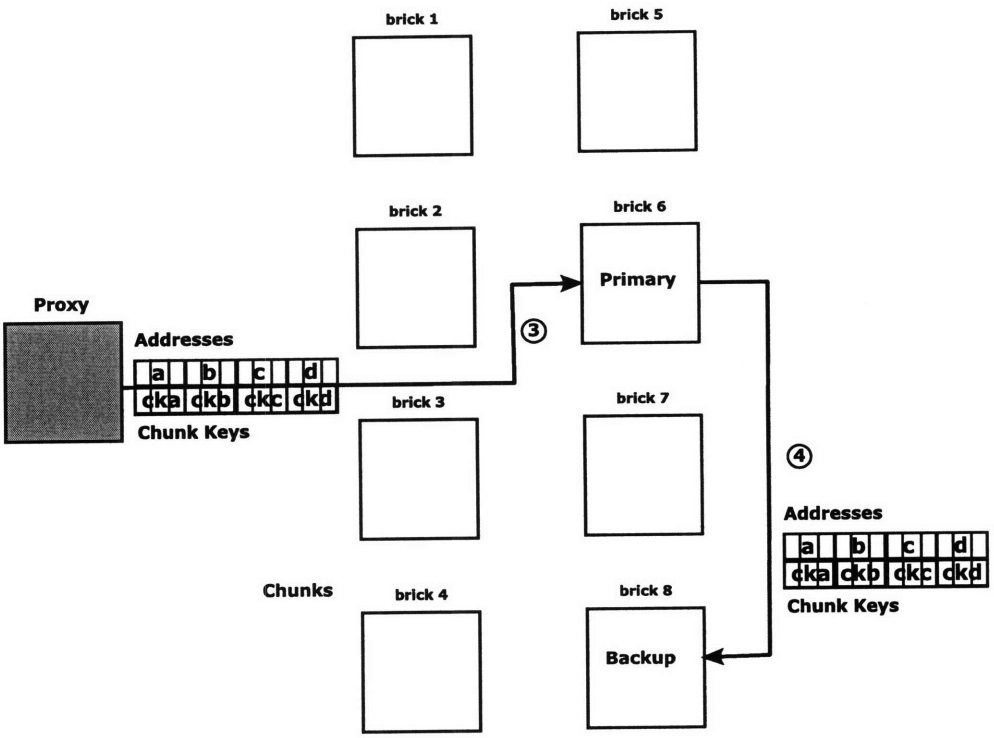

Figure 4-7: END **Write Phases III and IV:** After the data chunks are written to disk, (3) the proxy sends the addresses and their corresponding chunk keys to the primary (Brick 6, for this batch of addresses). (4) Brick 6 sends these address mappings to the backup (Brick 8) before replying to the proxy.

Chapter 5

Implementation

We have implemented a prototype of END in C++. The END brick software runs on Linux, while the END proxy software runs on FreeBSD or Linux. END uses the libasync [22] tool-kit for event-driven programming, and Berkeley DB4 [33] to store the address map entries and the indexes into END's chunk logs. END sends chunks and address map buckets via TCP, and other traffic via RPC over UDP. END proxy receives file system block requests via the FreeBSD GEOM Gate [6] network block device.

5.1 Write Optimization

The brick software uses the following techniques to avoid seeks during writes:

- At system initialization time END pre-writes each chunk log file with zeroes to force the file system to pre-allocate of blocks; this allows the brick to later write a chunk to the log file without incurring a seek to add a new block pointer to the file's i-node.
- If a brick has more than one disk, END places the chunk and address databases

on separate disks to avoid having to seek between files.

- Each brick flushes its in-memory chunk log to disk in the background while it is processing the writes in a batch, so that when a flush RPC arrives much of the disk I/O may already be complete.

- END directs DB4 to not write DB4's B-tree and internal log to disk after each update; instead, END has DB4 write only periodically for chunks and after receiving flush RPCs for addresses. During crash recovery the chunk DB4 index may be missing recent chunk log records, so the brick scans the tail of the chunk log to bring the index file up to date.

5.2 Brick Crash Recovery

When a brick re-starts, it must repair its chunk indices. It first asks DB4 to repair its B-trees by replaying any DB4 logs that might be on disk. Since END writes the chunk index DB4 log to disk infrequently, the DB4 indices may be missing the chunks at the end of END's log files. For this reason, the brick then scans its most recent log file from the last recorded point at which it flushed the DB4 log (written to a file on disk after each DB4 log flush), and inserts those chunk indices into the DB4 index. The brick then invokes DB4's recovery for its address map index. Once this process completes, END reads the buffer address mappings on disk (Section 3.1), and inserts them to the appropriate database.

The brick then invokes the View Module to join the current configuration. The View Module initiates a Paxos instance, which will form a view if that is possible under Paxos' majority rules. The brick then fetches copies of the address buckets assigned to it as described in Section 3.1.

5.3 Reconfiguration Implementation

END makes sure reconfiguration is efficient by using the following techniques:

- END's Paxos implementation lets the brick with the lowest ID initiate the next round of Paxos to lower the chance of competing proposals. Since the heartbeat module sends a ping to every brick in the system every second, it is very likely that every brick has the same knowledge of live bricks within a few seconds. Competing Paxos proposals can result in long delays before the bricks reach an agreement on the next view, thereby increasing reconfiguration time.

- END transfers address map buckets between bricks during reconfiguration by sending the underlying raw DB4 file, rather than reading/writing each entry via DB4.

5.4 Proxy Implementation

The proxy receives block I/O requests from the FreeBSD GEOM Gate pseudo disk device, upon which FreeBSD's FFS file system is mounted. END creates two processes to manage this communication. The first process receives block requests from the file system via the pseudo disk device, and sends these requests via a UNIX pipe to the second process which converts the file system block requests into END messages. The second process converts responses from END and sends them to the file system via the pseudo disk drive.

Chapter 6

Evaluation

This chapter explores END's basic performance, its throughput scaling as the number of bricks increases, and its ability to incorporate new bricks and handle temporary brick failures efficiently. In addition, the chapter presents the cost of maintaining the desired level of chunk replication and of garbage collecting unreferenced data chunks.

6.1 Experimental Setup

Our experimental setup consists of 16 bricks and 11 proxy machines. Each machine has a 2.8 GHz Pentium 4 CPU, 1GB of memory, two 160-gigabyte SATA disks (7200 rpm, 8 milliseconds average seek time), and a Gigabit Ethernet interface. We turn off IDE write caching so that data is durably on the disk when `fsync()` returns. Each brick uses one disk to store its address map DB4 files, the other to store its chunk logs and index files. The bricks run Linux 2.6.16 with `libata` SATA support, which helps improve write throughput when write caching is off. Each disk has a measured average sequential write throughput of 26 megabytes/s through the EXT2 file system. All machines are connected to the same Gigabit

Ethernet switch.

The four proxies for the file system experiments in Section 6.2 run on FreeBSD 5.4 so they can use the GEOM Gate pseudo-disk. For the rest of the experiments, the proxies run on a set of seven Linux 2.6.16 clients. The seven clients do not use a file system or pseudo-disk on top of END; instead, they directly generate disk operations for the proxy to perform. We use this arrangement because the FreeBSD/FFS/GEOM configuration does not generate enough work to explore the limits of END's performance.

6.2 File System Performance

This section compares the performance of a file system stored on END to that of a local disk file system. In each of the file system experiments, END runs on four bricks, and there are four client proxy machines. One proxy is used in Section 6.2.1, while up to four proxies are used in Section 6.2.2. The local disk measurements are taken on a machine with the same hardware as the bricks, but running FreeBSD and the FFS file system. Both the local disk and the END disk have a BSD FFS file system mounted on them. Soft updates are off in Section 6.2.1 and on in Section 6.2.2.

6.2.1 Latency of File System Operations

This section evaluates the latency of small file operations. To measure write latency, we ran a benchmark that issues 1,000 write requests of 8192 random bytes at random offsets in a 10 Gigabyte file. The benchmark issues one request at a time, calling `fsync()` after each one, and reports the average. To measure read latency, the benchmark issues read requests at random offsets one at a time and reports the average latency over 1,000 reads. Every read and write offset is 8192-

byte-aligned. The benchmark measures file create latency by creating 1,000 files in a single directory and taking the average. We disabled FFS's soft updates so that creates would map predictably to synchronous disk writes.

Experimental Setup	Write (*milliseconds*)	Read (*milliseconds*)	Create (*milliseconds*)
Local disk+FFS	25	8	17
END+FFS	30	11	34

Table 6.1: Latency of Individual FS Operations.

Table 6.1 shows the average latency of each operation. For each application-level write, FFS issues two block writes: one to write the file content, the other to update the file's inode. Two disk writes to the local disk take a total of two seeks and two half-rotations on average, for a total of 24 milliseconds, close to the measured 25 milliseconds. END requires two disk rotations to do each of the two writes (two parallel appends to the chunk log and two synchronous appends to the address database), or three disk rotations in total. This should take 24 milliseconds, which is close to the measured 30 milliseconds.

For each read, the local disk waits for one disk seek plus half a rotation to read the requested block, which should take 12 milliseconds on average. The measured 8 millisecond result is lower than predicted because parts of the file also fit in the file system's buffer cache, and because the file blocks are laid out consecutively on disk thus taking less time on average to seek among them. For each read, END looks up the block's address map on one brick and then reads the block content on perhaps another brick. The first operation does not access the disk because the address map fits in memory. The second operation causes a read to disk, which takes 1 seek and half a rotation on average. At 8 milliseconds per seek or rotation, it should take END 12 milliseconds on average to read a block. The measured 11 milliseconds is close to the predicted time.

For each create, FFS issues two block writes: one to write the new file's inode, the other to update the directory content. The local disk FFS issues 2 disk writes per create, which should make a create take as long as a write (24 milliseconds). The actual result is lower than expected because the file inodes and the directory inode are close to each other since all the files are created in the same directory. Each FFS create results in END issuing four sequential disk writes (as explained above in the case of writes), resulting in an estimated 32 milliseconds which is close to the measured 34 milliseconds.

6.2.2 File System Throughput of Large Writes

This section compares END's write throughput to that of a local disk file system. The benchmark sequentially writes a 10-Gigabyte file, calls `fsync()`, unmounts the file system, and calculates the throughput. We enabled FFS's soft updates so that the file system can optimize its block requests.

Experimental Setup	Throughput (megabytes/s)
Local disk+FFS	6.5
END+1 FFS client	6.8
END+2 FFS clients	11.9
END+4 FFS clients	18.8

Table 6.2: Sequential Write Throughput

Table 6.2 shows the throughput of storing a large file on END compared with a local disk. END consists of four bricks in this experiment. The local disk throughput with FreeBSD FFS is low because the FreeBSD IDE driver issues one 64-kilobyte write at a time, and is thus limited to a throughput of 64 kilobytes per rotation. END's limited throughput is the result of FFS sending batches of only 70 blocks, which is too small to hide the per-batch costs of flushing chunks and address map entries (the next section details these costs). With increasing of FFS

clients, END's throughput increases because the clients together send more blocks per batch, thus aggregating the cost to process a batch over more data.

6.3 Single-Client Large Writes

In order to evaluate the limit of END's write throughput with a single client, this section presents the performance of bulk writes sent directly from a load generator to the proxy. The generator sends 10,000 successive batches of 1,024 block writes, each write to a unique block address. We ran the generator on a Linux proxy talking to a 16-brick END cluster. In this experiment, we measured an average write throughput of 31.5 megabytes/s.

Phase	Time (milliseconds)	Predicted Time (milliseconds)
1. appending chunks	121	94
2. flushing chunks	23	20
3. appending addresses	73	78
4. flushing addresses	49	> 24

Table 6.3: Bulk write performance breakdown for a batch of 1,024 8192-byte writes on an END system of 16 bricks and 1 proxy.

Table 6.3 shows the average measured time for each phase of a write batch along with the time a simple model would predict. The time for the first phase is the time the proxy takes to send 8.4 megabytes of data to the four bricks in the current group and the time each brick takes to send 2.1 megabytes to its successor and receive, on average, 0.5 megabytes from its predecessor. Because there are four groups of four bricks in the cluster, there is a one in four chance that a given brick will be in the same group as its predecessor. Thus, on average in a given batch, a brick will receive 0.5 megabytes from its predecessor. The predicted time to send 8.4 megabytes over the proxy's gigabit link and to send/receive 2.6 megabytes over each brick's gigabit link is 94 milliseconds, somewhat lower than

the measured time. The result is that each brick of the four-brick group receives 2.6 megabytes of chunk replicas.

The second phase time is dominated by the flush to disk of an average of 524 kilobytes of data from each disk's chunk log. This is less than 2.6 megabytes because each brick was able to flush about 2.1 megabytes in the background during the first phase. The predicted time to write 524 kilobytes of data to disk at 26 megabytes/s is 20 milliseconds, which is a little lower than the measured time.

The time for the third phase is dominated by the time to insert the address maps into DB4. END performs all the address map inserts for a batch as part of a single DB4 transaction and takes 38 microseconds on average per entry. A batch of addresses usually goes to one primary brick. Since each batch of addresses contains 1,024 entries, the predicted time to append the addresses is 39 milliseconds each at the primary and backup brick for a total of 78 milliseconds. The time to append do not overlap because the primary inserts each entry into its DB4 before assigning that entry a corresponding backup. The primary does not know in advance if all the address map entries in the current batch belong to same bucket and thus have to sort them before sending each group of addresses to the corresponding backup.

Finally, the primary and backup bricks commit the address map DB4 transactions and flush the DB4 logs to disk, which takes at least 24 milliseconds per each DB4 log (disk seek + rotational latency to update the log file, and to update the log file's inode). The measured time is higher because the backup is often busy completing other phases that it does not immediately respond to the RPCs.

The average total measured time per batch is 266 milliseconds to process 8.4 megabytes of data, yielding a write throughput of 31.5 megabytes/s.

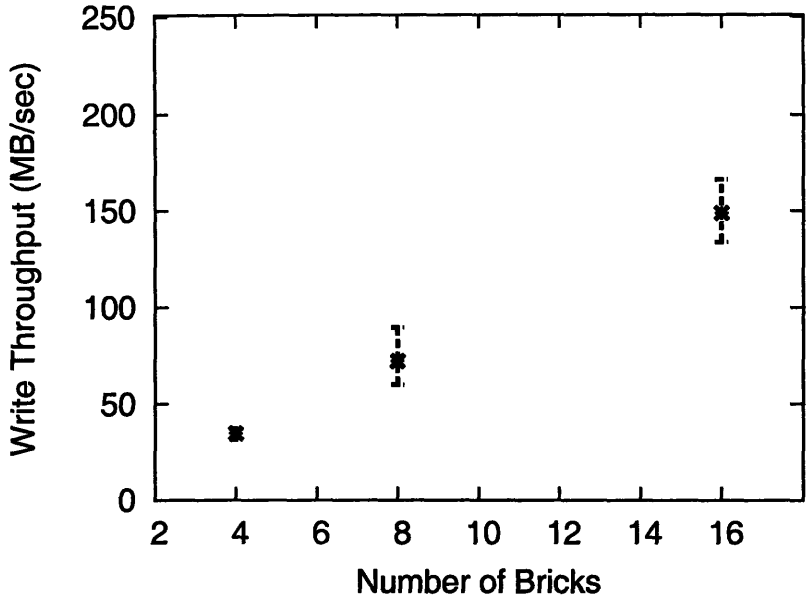

Figure 6-1: Write throughput with 7 proxies and different numbers of bricks. Each point is the average of 5 trials. The error bars indicate the minimum and maximum throughput among the trials.

6.4 Scalability with Bricks

Figure 6-1 shows how write throughput increases with the number of bricks. The experiments involve load from seven proxy machines each running 10 instances of the synthetic load generator (70 instances total). Multiple proxy machines increases the aggregate link capacity from the clients to the bricks, allowing clients to send data at $117 \times 7 = 819$ megabytes/s, a rate that far exceeds $26 \times 16 = 416$ megabytes/s, the combined rate at which the bricks can write to disk. Multiple instances of the load generator per machine ensures that the client machine is rarely idle. Each generator writes to a different range of block addresses. We took five measurements for each number of bricks, and report the minimum, average, and maximum.

Figure 6-1 shows the write throughput increasing from 36 megabytes/s with four bricks to 150 megabytes/s with 16 bricks. The reason END achieves this

speedup is that the four-brick group mechanism and middle-bits mapping of addresses to address map buckets causes the proxies often to spread load evenly across the bricks, which leads to good parallel speedup. Each brick achieves roughly 9.4 megabytes/s of application write throughput with 2x replication. The reason for this performance can be explained the same way as in Section 6.3. With 16 bricks, each four-brick group serves $70 \div 4 \approx 17$ load generators during a batch of writes. During phase 1, each brick receives 71.3 megabytes and sends 35.7 megabytes of data chunks, which takes 916 milliseconds over gigabit links. Phase 2 takes 2.8 seconds to write the 71.3 megabytes of data chunks received to disk. During Phases 3 and 4, the generators can send address maps to any of the 16 bricks, depending on the block address range. Each of the 17 load generators sends writes to different block address ranges, and so, on average, will send address maps to a different brick from the other 16 generators. Since there are 17 generators and 16 bricks, the batch of address maps from each load generator goes to roughly one brick. Thus, the time for Phases 3 and 4 is the same as in Section 6.3, 122 milliseconds. The total time for four bricks to process 142.6 megabytes of data is 3.8 seconds, which is a throughput of 37.2 megabytes/s for four bricks or 9.3 megabytes/s for one.

Figure 6-2 shows how the read throughput increases with the number of bricks. In this case, each of the 70 load generators sequentially reads the data written in the previous experiment; each generator reads from a different range of blocks. The total amount of chunk data that proxies read is 10N gigabytes, where N is the number of bricks, too large to fit in the bricks' caches.

Figure 6-2 shows increasing throughput for two reasons. First, different proxies read different address ranges and thus tend to spread the look up of address map entries evenly across different bricks. Second, each proxy reads chunk data in the order it was originally written, so that the bricks end up issuing efficient sequential reads from their chunk log files. Each brick offers a chunk read throughput of

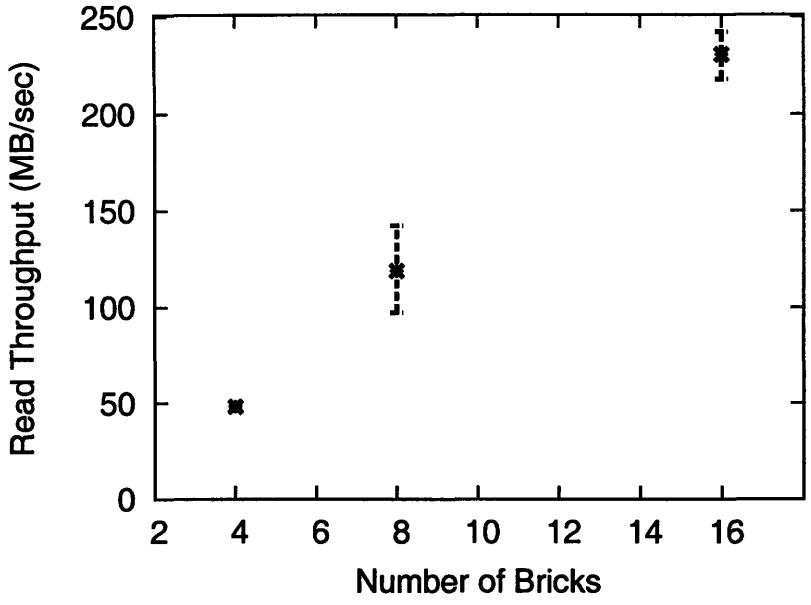

Figure 6-2: Read throughput with 7 proxies and different numbers of bricks. Each point is the average of 5 trials. The error bars indicate the minimum and maximum throughput among the trials.

$\sim$ 13.75 megabytes/s. The reason for this performance stems from how the data were laid out during the previous write experiments. In the 16-brick case, each brick received writes from five load generators on average during a batch. Hence, each batch of writes from a generator is interleaved with writes from three other generators. So each brick can expect $256 \div 5 = 51$ consecutive writes on average to come from the same generator. During a bulk sequential read, a brick can read 51 blocks sequentially before having to seek to the next group of blocks. The time to read 51 8KB blocks of data is 16 milliseconds plus 12 milliseconds to seek (and wait for half a disk rotation) to read the next group of blocks, resulting in an estimated read throughput of 14.9 megabytes/s.

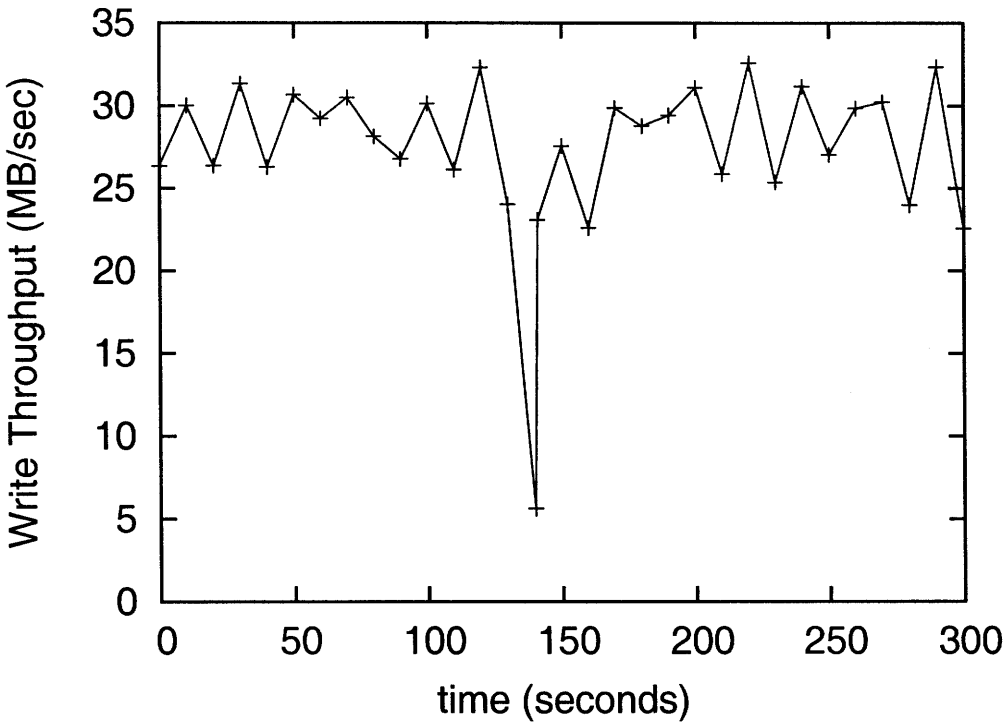

Figure 6-3: The effect on continuous write throughput when a new brick joins at time 130.

6.5 Effect of Incorporating a New Brick

Figure 6-3 shows the effect on write throughput when a new brick joins the system. Before time 130, one proxy is generating writes to 15 bricks. At time 130, a 16th brick joins the system. Figure 6-3 shows the throughput decreases to 5 megabytes/s during reconfiguration, and increases a second later to the same level as before adding the new brick (the single proxy cannot saturate the system's capacity).

The following events happen during the second of decreased throughput. END reconfigures to include the new brick, and redistributes the address map buckets. The proxy's append address request fails because the bucket at which it intends to write has been moved to the new brick. The proxy re-fetches the bucket map and retries the request at the new brick, which is still busy copying its portion of buckets from other bricks, hence the delay in responding to the proxy's writes. The new brick needs to fetch 10 buckets total, each bucket's DB4 file is $\sim$ 20 megabytes in size and so takes almost a second to write to disk. The proxy sees reduced throughput for only one second because the new brick buffers writes to incomplete buckets, reapplying those writes to the database later once it has finished fetching.

6.6 Effect of Temporary Failure

Next, we show END's behavior during a temporary failure. We pre-load an END system of 16 bricks with one terabyte of data, and then run the bulk write load generator on one proxy. Figure 6-4 shows the throughput as we kill the END process on one brick three minutes into the experiment, and then restart it seven minutes later with disk contents intact.

A few seconds after the failure, throughput drops to 8 megabyte/s and then to zero for one second. The following events happened during this pause. The

dead brick stops responding to requests from the proxy. Detecting failure, END reconfigures to exclude the dead brick. The proxy re-fetches the bucket map and retries the fetch request until END finishes reconfiguration and sends it the new bucket map. The proxy then resends the writes. END spreads second copies of the dead brick's 10 buckets over the remaining live bricks. A brick can write a 20-megabyte bucket in 700 milliseconds, which adds to the total two-second delay in responding to the proxy. After the proxy's retry is complete, throughput reaches the same level as before the failure. At this point, chunks that were stored on the dead brick have only one live replica. All writes issued while the brick is down will have two replicas.

Seven minutes after the failure, the dead brick rejoins the system. Now the situation is very similar to that in Section 6.5, where the address buckets need to be redistributed. If there were reads, the rejoining brick can serve requests with data already in its chunk store.

6.7 Chunk Maintenance and Garbage Collection

This section presents an evaluation of the maintenance algorithms. We pre-load a 16-brick END system with a terabyte of data, rewrite 20% of the addresses, and run chunk replication and garbage collection. END finishes maintenance and garbage collection in 8 minutes. The following paragraph details the cost of this process.

Each brick finishes passing the referenced and stored chunk keys and marking the chunks in about 3 minutes. There are 80 primary buckets, each with approximately 700,000 address entries. Each brick holds 5 primary buckets and takes about 10 seconds to iterate through a bucket's addresses and assign each to the appropriate chunk replica bricks for each entry. So each takes 50 seconds to assemble and send the chunk keys in all of its primary buckets to the appropriate

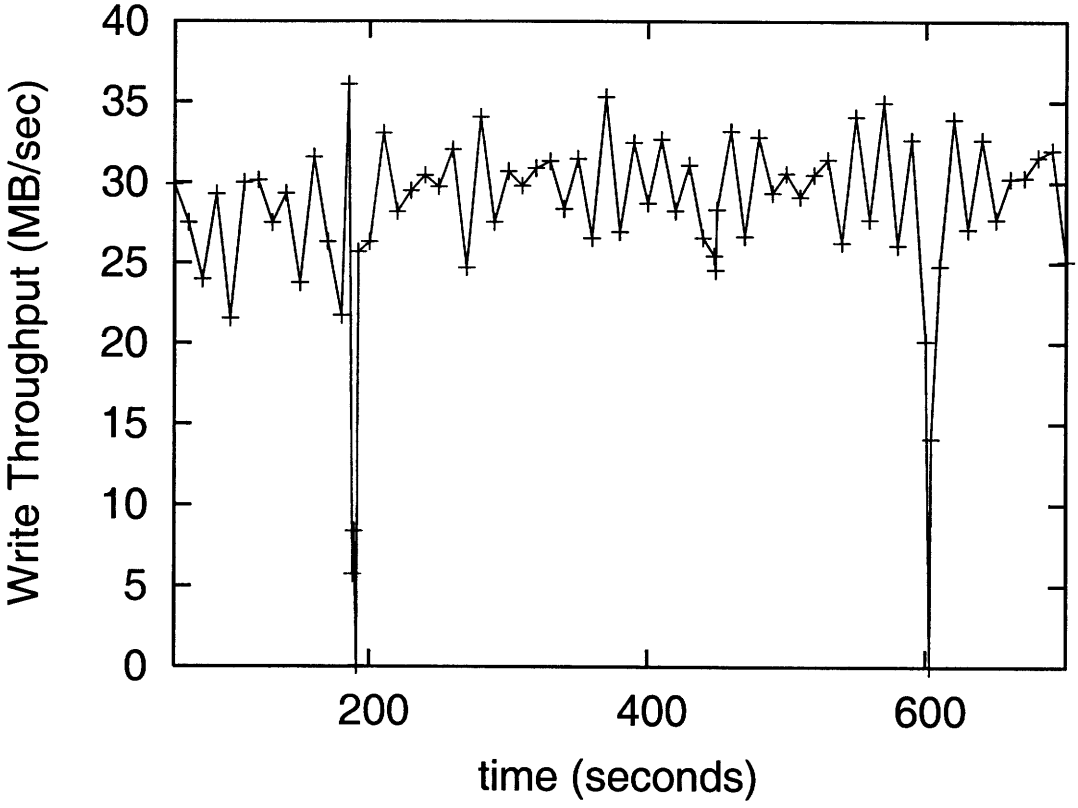

Figure 6-4: A brick fails at time 180 and recovers seven minutes later.

bricks. Most of this time is in traversing the DB4 index. Each brick then goes through the chunk index and assembles the keys of all the data chunks that it stores. There are at least double the number of data chunks on each brick to the primary address map entries because END has to keep track of all copies of the data chunks and because we have overwritten some portion of the address with new data. As a result, each brick finishes assembling and sending keys for stored chunks in roughly 100 seconds.

Because there are no failures, there is no need for any brick to fetch copies of data chunks. There is more than half a log worth of unreferenced chunks per brick; each brick finishes cleaning in 5 minutes. The time to clean a log is roughly as long as it takes to write the log sequentially (at 26 megabytes/s for a 8.2 gigabyte log file). This is because END reads the entire log, copying live records to another file, and swapping the two files when finished.

In general, the cost of maintenance and cleaning depends on the amount of data in the system and the amount of overwritten data. Assuming that each brick stores the same amount of data, it can be expressed as,

$$T_A + T_C + N_L T_L$$

where $T_A =$ the time for a brick to assemble and send referenced chunk keys

$T_C =$ the time for a brick to assemble and send keys for stored chunks

$N_L =$ the maximum number of chunk logs per brick where more than half of the entries are unreferenced

$T_L =$ the time to sequentially write an entire log file.

Chapter 7

Related Work

END builds on a long history of cluster storage research, particularly Petal [19], FAB [31], Ceph/RADOS [35, 36], and Boxwood [21]. END is motivated by the desire to use spare disk space on non-dedicated servers in a small university cluster to store data so that they can be conveniently accessed with a traditional disk interface, and automatically replicated. Because these servers are not dedicated storage servers, END needs to remain efficient, and the data must be available, when servers become unavailable. A design that handles failures and recoveries efficiently, while providing good performance during normal operation, is END's main contribution relative to previous systems.

END's most important difference from Petal, FAB [31], Ceph's RADOS [36], and Boxwood's Chunk Manager [21] is its separation of chunks from address mappings. These systems have meta-data similar in function to END's address mappings, but a block's meta-data must be on the same brick as the data. Thus Petal, FAB, RADOS, and Boxwood must treat the temporary failure of a brick specially, by recording all changes that the brick should have seen and playing back those changes when it recovers. While that brick is down, writes will not be replicated as much as desired. If that behavior is not acceptable, these systems must view the

failure as permanent, create a complete new replica of the data, and cannot make use of the failed brick if it does eventually recover. END, because it can move its address mappings without moving the data, is not forced to make these trade-offs.

Like Petal [19], END uses a level of indirection to allow expansion by adding disks and servers. END shares with GFS [7] and BigTable [4] the goals of scalable storage and of harnessing commodity components, but END's emphasis is on preserving the existing disk interface for use with existing file system layouts. GFS also relies on a single master to keep track of data chunks location and replication level. END cannot rely on a single server to store persistent information.

END's log-structured chunk store is borrowed from LFS [28] and Venti [25]. END's bucket map is inspired by the user map in Porcupine [30].

7.1 Cluster Block/Object Storage Systems

Petal [19] is a distributed virtual disk that incorporates storage from every brick in the system, and presents a single disk image. To add a new brick to the cluster, Petal reconfigures its block address assignment to include the new brick and copies to it data according to the new assignment. Because it can take hours for this process to complete, Petal divides the address range into "old", "new", and "fenced". Addresses in the old or new region should be served from the set of bricks according to the old or new assignment, respectively. The fenced region is the part where data is being moved, for which any client requests may require additional messages to get to the right brick. Petal stores each block on a primary and a backup brick. When either brick is unavailable, Petal serves read requests from the brick that is up. A client sends writes first to the primary, and then to the backup if the primary is down. If the primary is the first one to receive a write, it sets the block's busy bit before sending the write both to its local store and the

backup. The busy bit tells the primary which blocks to send to the backup that has just become available. If the primary is down, the backup will receive the write request. Before writing the block to disk, the backup must set the block's stale bit on disk so that when the primary becomes available, the backup knows which blocks the primary is missing.

Petal assigns mutable blocks to physical machines, and so has to copy whole blocks when the assignment changes. END assigns mutable address mappings that are much smaller than whole blocks. Thus, END moves only a small amount of data during change in the cluster.

FAB [31] divides bricks into *seggroups*, partitions block addresses among them, and uses a quorum protocol to perform reads and writes. Thus, only a majority of bricks in a seggroup need to respond before a read or write operation is complete. Each FAB brick keeps, in NVRAM, timestamps of blocks that have been updated on its local disk until all bricks in the seggroup acknowledge the update of those blocks. During a brick failure, FAB continues to serve requests since a majority of bricks in each seggroup is still alive. However, the remaining bricks in each seggroup with a failed brick cannot garbage collect the timestamps of block updates. To discard the timestamps, the remaining bricks must reconfigure to exclude the dead brick from the seggroup. If the dead brick recovers before reconfiguration, it needs to copy from each of its seggroup the recently updated blocks as indicated by the timestamps. Otherwise, the recovered brick will have to participate in reconfiguration and copy every block in each seggroup to which it belongs. Adding a new brick follows the same reconfiguration procedure.

Unlike FAB, END uses primary-backup replication. If a failed brick recovers before FAB decides to reconfigure (perhaps due to its NVRAM filling up), then FAB is efficient in that it only needs to copy new blocks to the recovering brick. Otherwise, FAB may have to copy over data that already exist on the recovering

brick. In contrast, END only copies the small address map. Any newly written data chunks are already replicated on the remaining live bricks due to END's flexible data chunk allocation. Adding a new brick in FAB always result in more data transfer up front than in END, because FAB needs to copy whole blocks to the new brick while END only copies the address map.

Ceph/RADOS [35, 36] stores data objects, as supposed to data blocks, and allows clients to read/write byte extents of those objects. RADOS divides data objects into placement groups and assigns these groups to participating bricks according to the *cluster map*. Within a placement group, RADOS designates one brick to be the primary; all writes start at this brick. When the cluster map changes, the primary of each placement group figures out how much data needs to be copied to each brick to bring them up to date. The primary accomplishes this by consulting its local log of updates to the placement group and comparing it to the most recent updates on each replica brick.

The placement groups in RADOS are similar to the buckets in END, except that END stores blocks instead of objects. Furthermore, END splits the blocks into addresses and chunks, so it can reconfigure quickly and thus does not need to keep track of updates that are destined for bricks that are undergoing change. RADOS faces the tension of delaying reconfiguration at the expense of reduced replication of new updates (in case the dead brick comes back online), or reconfigure and copy the entire content of a placement group to a new replica.

DDS [9] provides scalable storage with a cluster of storage bricks, with an even more general interface than END, since DDS allows arbitrary keys. DDS has slower recovery after temporary failure than END, since DDS must generally entirely rebuild the recovered brick's contents. DDS also cannot tolerate network partitions, which may limit its applicability in large clusters with complex switched LAN topologies.

NASD [8] is an object-based network disk whose goal is to provide efficient and secure access to its data. END shares the goal of efficient access with NASD, but trusts all the requests it receives over the LAN. Another crucial goal for END is to handle changes in the cluster efficiently. This goal is not the emphasis in NASD.

7.2 Cluster File Systems

Lustre [13] is similar to Ceph [35] in that it keeps a separate layer of file system meta-data and stores file data as objects. The object store for Lustre keeps a mutable mapping of object ID to data, which ties them to the same brick. Thus, during temporary failures, Lustre faces the same tension as Ceph/RADOS, while END does not.

ZFS [24] is a file system that allocates blocks from pooled storage, which could include anything from a single disk partition to hundreds of disks in a server farm. The block allocation strategy it uses is similar to END's. ZFS uses copy-on-write whenever it writes to disk, similar to WAFL [11]. The latter is an on-disk file system that handles NFS operations efficiently. WAFL keeps files' data together with their data; which allows it to write data blocks anywhere on disk; which in turn, allows it to create snapshots quickly. In contrast, END appends a new block to its chunk store when the file system writes to it, and keeps the chunk index information in a separate database.

The Google File System [7] is a cluster file system whose goal is to be scalable to data intensive workload while using commodity components. A GFS cluster consists of one master and hundreds of chunk servers. Clients send all meta-data operations to the master, but sends all reads and writes of data directly to chunk servers. The latter design decision allows GFS to scale with the number of clients. GFS master grants leases on the primary chunk server of a particular

GFS chunk and clients caches the lease. The primary chunk server can also renew the lease indefinitely, so if there are no failures at the primary or master, the client will never have to contact the master for chunk operations. Like GFS, END uses commodity components to provide scalable storage in a cluster environment. However, END has no central component such as the GFS master. END provides a block storage and does not need to manage file system meta-data. Each END node has information of where each block address range is stored in the system.

Harp [20] is a replicated file system that uses a distributed consensus protocol to reconfigure after failures. Like Harp, END uses primary and backup replication to guarantee consistency, but Harp applies updates to in-memory logs, and writes to disk in the background to increase performance. To prevent data loss, each Harp node has an Uninterruptible Power Supply (UPS) that enables it to flush its logs to disk before shutting down during a power outage. END commits updates to disk before returning to the client, and does not assume a UPS. A recovering Harp node fetches any information from the remaining live nodes in order to bring itself up to date before initiating a view change. This scheme reduces the time to complete a view change. In contrast, an END brick stores a portion of the address map and may not store the same portion in each view, so END cannot fetch the address map before initiating a view change. Nevertheless, END's view change is fast because the amount of address map that a new or recovering brick has to copy is small.

Zebra [10], PVFS [3], GPFS [32], and Panasas [34] are parallel file systems that scale to extremely high throughput with parallel data access workloads. They obtain high throughput via data striping and data consistency with a central meta-data manager (Zebra, PVFS, and Panasas) and distributed locking techniques that minimize contention (GPFS). END shares the goal of scalable data access with these systems, but END is a distributed block store and thus it offers block-level consistency as supposed to file-level. GPFS handles node failures by replaying

the operations of the failed node on the remaining nodes. Panasas invokes RAID rebuild when it detects a storage node failure. Zebra uses both log replay and RAID rebuild to recover from failure. END handles temporary failures by re-assigning address maps among remaining bricks and copying over the address maps according to the new assignment. For permanent failures, END uses a decentralized algorithm to check the number of chunk copies and make extra ones as necessary.

7.3 Log Structured Storage

The Log-structured File System [28] stores all file system data in a log on disk. LFS can write data to disk almost without seeking; its log segments are the inspiration for chunk logs in END. Consequently, END faces the same challenges of reading data and cleaning the logs as LFS. Unlike LFS, END does not keeping indexing information in the log, but borrows Venti's [25] use of a database to keep the index. This approach is simpler, because the index is separate from the data. It provides fast look up, because the index is usually small enough to fit in memory. It is easy to maintain, because the index is soft state and can be completely re-built from the logs. Although for fast recovery, END periodically checkpoints the index to disk.

Similar to LFS with regard to cleaning the logs, END does not keep a separate free chunk list or bitmap. During cleaning each END brick checks the liveness of chunks whose keys are within its range. Once an END brick has identified the live chunks stored locally in its logs, it cleans the logs with number of dead chunks more than half the total number of records. The reason for this heuristic is such that cleaning a log would result in more free space than the amount of live data copied. This thesis has not performed a cleaning cost-benefit analysis in END as is done in LFS.

7.4 Immutable Storage

Distributed hash tables [5, 26, 29] focus on providing data integrity and good performance for wide-area data storage. END's initial design for the chunk store was inspired by these systems, but END also provides mutable block-addressed data access, as well as assuming a high-performance cluster environment.

END shares the use of immutable storage with EMC's Centera [14], but END's main aim is to support ordinary read/write file systems. Centera stores content-addressable data and is designed to serve read-only content. END is designed to serve mutable data, and thus needs a mutable layer with references to immutable content. END also has to take care of garbage collection of unreferenced immutable chunks.

7.5 Wide-area File Systems

Some wide-area file systems, such as OceanStore/Pond [16, 27] and FarSite [1], provide a filesystem interface built on DHT storage. These systems include their own filesystem tailored to the performance and security requirements of wide-area DHT storage, for example processing file system updates with Byzantine-fault-tolerant clusters. END, in contrast, provides a disk interface so that it is usable with existing disk filesystems, trusts all the bricks, and assumes a high-performance LAN.

7.6 Distributed Databases

Distributed database systems replicate data among bricks, and thus face challenges similar to END's. HARBOR [18] is a distributed database that re-incorporates a

recovering brick by querying other live bricks for updates and uses timestamps to determine which tuples need to be copied. HARBOR shares the same goal of fast recovery with END, but HARBOR is a database system, which has significantly different data structures from a virtual disk.

Chain declustering [12] is a technique that replicates portions of the database such that a brick failure will not affect the load balancing property of the system. IBM DB2 Parallel Edition [2] uses *hash partitioning functions* to determine to which brick to assign a tuple, adjusting the hash functions results in adjusting the load at each brick. END could benefit from using these techniques to replicate its Address Map and Chunk Store to keep the load balanced when bricks fail.

Chapter 8

Conclusions and Future Work

This thesis describes the END cluster storage system, which presents a fault-tolerant virtual disk interface usable by existing file system implementations. The key challenge END faces is to lay out its data over the bricks and their disks in a way that is both efficient to access in ordinary operation, and efficient to maintain as bricks fail and re-join. END solves this challenge with a two-level design: the mapping from block addresses to chunk keys lets END store data in efficient locations, write new data with full replication despite failed bricks, and re-incorporate recovered bricks efficiently. The experimental results show that END's two-layer design maintains good performance and efficiently handles changes in the cluster.

8.1 Future Directions

Areas for future design and implementation include:

1. A load balancing mechanism in case different bricks have significantly different disk capacities.

2. A mechanism to gradually move existing chunks to a newly joined brick.

3. An algorithm to decide how often to invoke replica maintenance and garbage collection. The algorithm should minimize the probability that a data chunk is lost, yet maximize the available disk and network bandwidth for serving data requests.

4. A log cleaning policy that uses a cost-benefit analysis [28] to maximize disk space utilization and minimize write cost during cleaning.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer and Roger Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[2] Chaitanya Baru, Gilles Fecteau, Ambuj Goyal, Hui-i Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland and W. G. Wilson. DB2 Parallel Edition. *IBM Systems Journal* 43(2), 1995.

[3] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, USENIX Association, Berkeley, CA, USA, 2000.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Symposium on Operating System Design and Implementation*, Seattle, Washington, November 2006.

[5] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[6] Pawel Jakub Dawidek. FreeBSD Geom Gate. `http://www.freebsd.org/doc/en/books/handbook/geom-ggate.html`.

[7] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. The Google File System. In *Symposium on Operating Systems Principles*, Bolton Landing, New York, October 2003.

[8] Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[9] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein and David E. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

[10] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *Symposium on Operating Systems Principles*, Asheville, North Carolina, October 1993.

[11] Dave Hitz, James Lau and Michael Malcolm. File System Design for an NFS File Server Appliance. Technical Report, Network Appliance, Inc, 2005.

[12] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *IEEE International Conference on Data Engineering*, February 1990.

[13] Cluster File Systems Inc. Lustre: A Scalable, High-Performance File System. `http://www.lustre.org/docs/whitepaper.pdf`.

[14] EMC Inc. EMC Centera Content Addressed Storage System. `http://emc.com/products/systems/centera.jsp?openfolder=platform`.

[15] David R. Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, El Paso, Texas, May 1997.

[16] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells and Ben Y. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2003.

[17] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16(2):133-169, 1998.

[18] Edmond Lau and Samuel Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *International Conference on Very Large Databases*, September 2006.

[19] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, October 1996.

[20] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira and Michael Williams. Replication in the Harp File System. In *Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991.

[21] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Symposium on Operating System Design and Implementation*, San Francisco, California, December 2004.

[22] David Mazières. A Toolkit for User-Level File Systems. In *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[23] Jennifer McAdams. 27 Billion Gigabytes to Be Archived By 2010. `http://www.computerworld.com/action/article.do?command=viewArticleBasic&taxonomyId=19&articleId=307657&intsrc=hm_topic`, *Computer World*, December 2007.

[24] Sun Microsystems. Zettabyte File System. `http://www.sun.com/2004-0914/feature/`.

[25] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *USENIX Conference on File and Storage Technologies*, 2002.

[26] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp and Scott Shenker. A Scalable Content-Addressable Network. In *ACM Special Interest Group on Data Communications*, San Diego, California, August 2001.

[27] Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao and John Kubiatowicz. Pond: The OceanStore Prototype. In *USENIX Conference on File and Storage Technologies*, 2003.

[28] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991.

[29] Antony I. T. Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[30] Yasushi Saito, Brian N. Bershad and Henry M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. In *Symposium on Operating Systems Principles*, Kiawah Island, South Carolina, December 1999.

[31] Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant and Susan Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, October 2004.

[32] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *USENIX Conference on File and Storage Technologies*, 2002.

[33] Sleepy Cat Software. BerkeleyDB Version 4. `http://www.sleepycat.com`.

[34] Brent Welch andMarc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller Jason Small Jim Zelenka and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *USENIX Conference on File and Storage Technologies*, 2008.

[35] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Dyarrell D. E. Long and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Symposium on Operating System Design and Implementation*, Seattle, Washington, November 2006.

[36] Sage A. Weil, Andrew W. Leung, Scott A. Brandt and Carlos Maltzahn. RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-Scale

Storage Clusters. In *Petascale Data Storage Workshop, Supercomputing '07*, 2007.

[37] Making Use of Terabytes of Unused Storage. `http://ask.slashdot.org/article.pl?sid=08/02/09/1319258`.