**The Google File System**

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung Google∗

# ABSTRACT

We have designed and implemented the Google File Sys- tem, a scalable distributed ﬁle system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous dis- tributed ﬁle systems, our design has been driven by obser- vations of our application workloads and technological envi- ronment, both current and anticipated, that reﬂect a marked departure from some earlier ﬁle system assumptions. This has led us to reexamine traditional choices and explore rad- ically diﬀerent design points.

The ﬁle system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our ser- vice as well as research and development eﬀorts that require large data sets. The largest cluster to date provides hun- dreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present ﬁle system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

# Categories and Subject Descriptors

D [**4**]: 3—*Distributed ﬁle systems*

# General Terms

Design, reliability, performance, measurement

# Keywords

Fault tolerance, scalability, data storage, clustered storage



∗The authors can be reached at the following addresses:

*{sanjay,hgobioﬀ,shuntak}@google.com*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SOSP’03,* October 19–22, 2003, Bolton Landing, New York, USA. Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

# INTRODUCTION

We have designed and implemented the Google File Sys- tem (GFS) to meet the rapidly growing demands of Google’s data processing needs. GFS shares many of the same goals as previous distributed ﬁle systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application work- loads and technological environment, both current and an- ticipated, that reﬂect a marked departure from some earlier ﬁle system design assumptions. We have reexamined tradi- tional choices and explored radically diﬀerent points in the design space.

First, component failures are the norm rather than the exception. The ﬁle system consists of hundreds or even thousands of storage machines built from inexpensive com- modity parts and is accessed by a comparable number of client machines. The quantity and quality of the compo- nents virtually guarantee that some are not functional at any given time and some will not recover from their cur- rent failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power sup- plies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, ﬁles are huge by traditional standards. Multi-GB ﬁles are common. Each ﬁle typically contains many applica- tion objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of ap- proximately KB-sized ﬁles even when the ﬁle system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited. Third, most ﬁles are mutated by appending new data rather than overwriting existing data. Random writes within a ﬁle are practically non-existent. Once written, the ﬁles are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running ap- plications. Some may be archival data. Some may be in- termediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge ﬁles, appending becomes the fo- cus of performance optimization and atomicity guarantees,

while caching data blocks in the client loses its appeal. Fourth, co-designing the applications and the ﬁle system

API beneﬁts the overall system by increasing our ﬂexibility.

For example, we have relaxed GFS’s consistency model to vastly simplify the ﬁle system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a ﬁle without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for diﬀerent purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

# DESIGN OVERVIEW

* 1. **Assumptions**

In designing a ﬁle system for our needs, we have been guided by assumptions that oﬀer both challenges and op- portunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.

*•*

The system stores a modest number of large ﬁles. We expect a few million ﬁles, each typically 100 MB or larger in size. Multi-GB ﬁles are the common case and should be managed eﬃciently. Small ﬁles must be supported, but we need not optimize for them.

*•*

The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a ﬁle. A small ran- dom read typically reads a few KBs at some arbitrary oﬀset. Performance-conscious applications often batch and sort their small reads to advance steadily through the ﬁle rather than go back and forth.

*•*

The workloads also have many large, sequential writes that append data to ﬁles. Typical operation sizes are similar to those for reads. Once written, ﬁles are sel- dom modiﬁed again. Small writes at arbitrary posi- tions in a ﬁle are supported but do not have to be eﬃcient.

*•*

The system must eﬃciently implement well-deﬁned se- mantics for multiple clients that concurrently append to the same ﬁle. Our ﬁles are often used as producer- consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concur- rently append to a ﬁle. Atomicity with minimal syn- chronization overhead is essential. The ﬁle may be read later, or a consumer may be reading through the ﬁle simultaneously.

*•*

High sustained bandwidth is more important than low latency. Most of our target applications place a pre- mium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

*•*

# Interface

GFS provides a familiar ﬁle system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identiﬁed by path- names. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* ﬁles.

Moreover, GFS has *snapshot* and *record append* opera- tions. Snapshot creates a copy of a ﬁle or a directory tree at low cost. Record append allows multiple clients to ap- pend data to the same ﬁle concurrently while guaranteeing the atomicity of each individual client’s append. It is use- ful for implementing multi-way merge results and producer- consumer queues that many clients can simultaneously ap- pend to without additional locking. We have found these types of ﬁles to be invaluable in building large distributed applications. Snapshot and record append are discussed fur- ther in Sections 3.4 and 3.3 respectively.

# Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly ﬂaky application code is acceptable.

Files are divided into ﬁxed-size *chunks*. Each chunk is identiﬁed by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux ﬁles and read or write chunk data speciﬁed by a chunk handle and byte range. For reliability, each chunkis replicated on multi- ple chunkservers. By default, we store three replicas, though users can designate diﬀerent replication levels for diﬀerent regions of the ﬁle namespace.

The master maintains all ﬁle system metadata. This in- cludes the namespace, access control information, the map- ping from ﬁles to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master peri- odically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the ﬁle system API and communicates with the master and chunkservers to read or write data on behalf of the applica- tion. Clients interact with the master for metadata opera- tions, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches ﬁle data. Client caches oﬀer little beneﬁt because most applications stream through huge ﬁles or have working sets too large to be cached. Not having them simpliﬁes the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache ﬁle data because chunks are stored as local ﬁles and so Linux’s buﬀer cache already keeps frequently accessed data in memory.

# Single Master

Having a single master vastly simpliﬁes our design and enables the master to make sophisticated chunk placement

(file name, chunk index)



/foo/bar

**GFS master**

File namespace

Application

|  |
| --- |
| chunk 2ef0 |
|  |
|  |
|  |

**GFS client**

(chunk handle, chunk locations)

(chunk handle, byte range) chunk data

Linux file system

**GFS chunkserver**

Legend:

Instructions to chunkserver

Chunkserver state

**GFS chunkserver**

Data messages Control messages

### Figure 1: GFS Architecture

Linux file system

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write ﬁle data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with refer- ence to Figure 1. First, using the ﬁxed chunk size, the client translates the ﬁle name and byte oﬀset speciﬁed by the ap- plication into a chunk index within the ﬁle. Then, it sends the master a request containing the ﬁle name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the ﬁle name and chunk index as the key. The client then sends a request to one of the replicas, most likely the closest one. The request speciﬁes the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the ﬁle is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the informa- tion for chunks immediately following those requested. This extra information sidesteps several future client-master in-

teractions at practically no extra cost.

# Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical ﬁle sys- tem block sizes. Each chunk replica is stored as a plain Linux ﬁle on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size oﬀers several important advantages. First, it reduces clients’ need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location informa- tion. The reduction is especially signiﬁcant for our work- loads because applications mostly read and write large ﬁles sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persis-

tent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunksize, even with lazy space allocation, has its disadvantages. A small ﬁle consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same ﬁle. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk ﬁles sequentially.

However, hot spots did develop when GFS was ﬁrst used by a batch-queue system: an executable was written to GFS as a single-chunk ﬁle and then started on hundreds of ma- chines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous re- quests. We ﬁxed this problem by storing such executables with a higher replication factor and by making the batch- queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

# Metadata

The master stores three major types of metadata: the ﬁle and chunk namespaces, the mapping from ﬁles to chunks, and the locations of each chunk’s replicas. All metadata is kept in the master’s memory. The ﬁrst two types (names- paces and ﬁle-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the mas- ter’s local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location informa- tion persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

## *In-Memory Data Structures*

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and eﬃcient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver fail- ures, and chunk migration to balance load and disk space

usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The mas- ter maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most ﬁles contain many chunks, only the last of which may be partially ﬁlled. Sim- ilarly, the ﬁle namespace data typically requires less then 64 bytes per ﬁle because it stores ﬁle names compactly us- ing preﬁx compression.

If necessary to support even larger ﬁle systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and ﬂexibility we gain by storing the metadata in memory.

## *Chunk Locations*

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with reg- ular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to real- ize that a chunkserver has the ﬁnal word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

## *Operation Log*

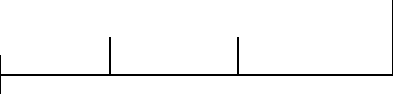
The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that deﬁnes the order of concurrent op- erations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identiﬁed by the logical times at which they were created.

Since the operation log is critical, we must store it reli- ably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we eﬀectively lose the whole ﬁle system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client opera- tion only after ﬂushing the corresponding log record to disk both locally and remotely. The master batches several log records together before ﬂushing thereby reducing the impact of ﬂushing and replication on overall system throughput.

The master recovers its ﬁle system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the

|  |  |  |
| --- | --- | --- |
|  | Write | Record Append |
| Serial success  Concurrent | *defined*  *consistent* | *defined*  interspersed with  *inconsistent* |
| successes | but *undefined* |  |

### Table 1: File Region State After Mutation



Failure

*inconsistent*

limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without ex- tra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the mas- ter’s internal state is structured in such a way that a new checkpoint can be created without delaying incoming muta- tions. The master switches to a new log ﬁle and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million ﬁles. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log ﬁles. Older checkpoints and log ﬁles can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not aﬀect correctness because the recovery code detects and skips incomplete checkpoints.

# Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and eﬃcient to implement. We now discuss GFS’s guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

## *Guarantees by GFS*

File namespace mutations (e.g., ﬁle creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master’s operation log deﬁnes a global total order of these operations (Section 2.6.3).

The state of a ﬁle region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summa- rizes the result. A ﬁle region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *deﬁned* after a ﬁle data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the aﬀected region is deﬁned (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undeﬁned but consistent: all clients see the same data, but it may not reﬂect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region in- consistent (hence also undeﬁned): diﬀerent clients may see diﬀerent data at diﬀerent times. We describe below how our applications can distinguish deﬁned regions from undeﬁned

regions. The applications do not need to further distinguish between diﬀerent kinds of undeﬁned regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-speciﬁed ﬁle oﬀset. A record append causes data (the “record”) to be appended *atomically at least once* even in the presence of concurrent mutations, but at an oﬀset of GFS’s choosing (Section 3.3). (In contrast, a “regular” append is merely a write at an oﬀset that the client believes to be the current end of ﬁle.) The oﬀset is returned to the client and marks the beginning of a deﬁned region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated ﬁle region is guaranteed to be deﬁned and contain the data writ- ten by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunkversion numbers to detect any replica that has become stale because it has missed mu- tations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This win- dow is limited by the cache entry’s timeout and the next open of the ﬁle, which purges from the cache all chunk in- formation for that ﬁle. Moreover, as most of our ﬁles are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get cur- rent chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identiﬁes failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it be- comes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

## *Implications for Applications*

GFS applications can accommodate the relaxed consis- tency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate ﬁles by appending rather than overwriting. In one typical use, a writer gener- ates a ﬁle from beginning to end. It atomically renames the ﬁle to a permanent name after writing all the data, or pe- riodically checkpoints how much has been successfully writ- ten. Checkpoints may also include application-level check- sums. Readers verify and process only the ﬁle region up to the last checkpoint, which is known to be in the deﬁned state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more eﬃ- cient and more resilient to application failures than random writes. Checkpointing allows writers to restart incremen- tally and keeps readers from processing successfully written

ﬁle data that is still incomplete from the application’s per- spective.

In the other typical use, many writers concurrently ap- pend to a ﬁle for merged results or as a producer-consumer queue. Record append’s append-at-least-once semantics pre- serves each writer’s output. Readers deal with the occa- sional padding and duplicates as follows. Each record pre- pared by the writer contains extra information like check- sums so that its validity can be veriﬁed. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent op- erations), it can ﬁlter them out using unique identiﬁers in the records, which are often needed anyway to name corre- sponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other ﬁle interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

# SYSTEM INTERACTIONS

We designed the system to minimize the master’s involve- ment in all operations. With that background, we now de- scribe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snap- shot.

# Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append opera- tion. Each mutation is performed at all the chunk’s replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunklease to one of the repli- cas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is deﬁned ﬁrst by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize manage- ment overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mu- tated, the primary can request and typically receive exten- sions from the master indeﬁnitely. These extension requests and grants are piggybacked on the *HeartBeat* messages reg- ularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a ﬁle that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control ﬂow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

4 step 1

3

2

6

7

5

6

Secondary Replica B

Primary Replica

Secondary Replica A

Master

Client

Legend:

Control Data

ﬁle region may end up containing fragments from diﬀerent clients, although the replicas will be identical because the in- dividual operations are completed successfully in the same order on all replicas. This leaves the ﬁle region in consistent but undeﬁned state as noted in Section 2.7.

# Data Flow

We decouple the ﬂow of data from the ﬂow of control to use the network eﬃciently. While control ﬂows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine’s network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine’s network bandwidth, the

### Figure 2: Write Control and Data Flow

becomes unreachable or replies that it no longer holds a lease.

1. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buﬀer cache until the data is used or aged out. By decoupling the data ﬂow from the control ﬂow, we can improve performance by scheduling the expensive data ﬂow based on the net- work topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
2. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identiﬁes the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serial- ization. It applies the mutation to its own local state in serial number order.
3. The primary forwards the write request to all sec- ondary replicas. Each secondary replica applies mu- tations in the same serial number order assigned by the primary.
4. The secondaries all reply to the primary indicating that they have completed the operation.
5. The primary replies to the client. Any errors encoun- tered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary repli- cas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modiﬁed region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the be- ginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control ﬂow described above but may be interleaved with and overwritten by con- current operations from other clients. Therefore, the shared

data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine’s full outbound bandwidth is used to trans- fer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the “closest” machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 for- wards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that “distances” can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data trans- fer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is espe- cially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring *B* bytes to *R* replicas is *B/T* + *RL* where *T* is the network throughput and *L* is la- tency to transfer bytes between two machines. Our network links are typically 100 Mbps (*T* ), and *L* is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

# Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client speciﬁes the oﬀ- set at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client speciﬁes only the data. GFS appends it to the ﬁle at least once atomically (i.e., as one continuous sequence of bytes) at an oﬀset of GFS’s choosing and returns that oﬀset to the client. This is similar to writ- ing to a ﬁle opened in O APPEND mode in Unix without the

race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applica- tions in which many clients on diﬀerent machines append to the same ﬁle concurrently. Clients would need addi- tional complicated and expensive synchronization, for ex- ample through a distributed lock manager, if they do so with traditional writes. In our workloads, such ﬁles often

serve as multiple-producer/single-consumer queues or con- tain merged results from many diﬀerent clients.

Record append is a kind of mutation and follows the con- trol ﬂow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the ﬁle Then, it sends its request to the pri- mary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the max- imum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst- case fragmentation at an acceptable level.) If the record ﬁts within the maximum size, which is the common case, the primary appends the data to its replica, tells the secon- daries to write the data at the exact oﬀset where it has, and ﬁnally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may con- tain diﬀerent data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This prop- erty follows readily from the simple observation that for the operation to report success, the data must have been written at the same oﬀset on all replicas of some chunk. Further- more, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher oﬀset or a diﬀerent chunk even if a diﬀerent replica later becomes the primary. In terms of our consistency guar- antees, the regions in which successful record append opera- tions have written their data are deﬁned (hence consistent), whereas intervening regions are inconsistent (hence unde- ﬁned). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

# Snapshot

The snapshot operation makes a copy of a ﬁle or a direc- tory tree (the “source”) almost instantaneously, while min- imizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it ﬁrst revokes any outstanding leases on the chunks in the ﬁles it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to ﬁnd the lease holder. This will give the master an opportunity to create a new copy of the chunk ﬁrst.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source ﬁle or directory tree. The newly created snap- shot ﬁles point to the same chunks as the source ﬁles.

The ﬁrst time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to ﬁnd the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk

handle C’. It then asks each chunkserver that has a current replica of C to create a new chunk called C’. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the net- work (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no dif- ferent from that for any chunk: the master grants one of the replicas a lease on the new chunkC’ and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

# MASTER OPERATION

The master executes all namespace operations. In addi- tion, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now dis- cuss each of these topics.

# Namespace Management and Locking

Many master operations can take a long time: for exam- ple, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional ﬁle systems, GFS does not have a per-directory data structure that lists all the ﬁles in that directory. Nor does it support aliases for the same ﬁle or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With preﬁx compression, this table can be eﬃciently represented in memory. Each node in the namespace tree (either an absolute ﬁle name or an absolute directory name) has an associated read-write lock. Each master operation acquires a set of locks before it runs. Typically, if it involves /d1/d2/.../dn/leaf, it will

acquire read-locks on the directory names /d1, /d1/d2, ...,

/d1/d2/.../dn, and either a read lock or a write lock on the full pathname /d1/d2/.../dn/leaf. Note that leaf may be a ﬁle or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a ﬁle /home/user/foo from being created while /home/user is being snapshotted to /save/user. The snapshot oper- ation acquires read locks on /home and /save, and write locks on /home/user and /save/user. The ﬁle creation ac- quires read locks on /home and /home/user, and a write lock on /home/user/foo. The two operations will be seri- alized properly because they try to obtain conﬂicting locks on /home/user. File creation does not require a write lock on the parent directory because there is no “directory”, or

*inode*-like, data structure to be protected from modiﬁcation. The read lock on the name is suﬃcient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple ﬁle creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the ﬁle name. The read lock on the directory name suﬃces to prevent the directory from being deleted, renamed, or snapshotted. The write locks on

ﬁle names serialize attempts to create a ﬁle with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are ﬁrst ordered by level in the namespace tree and lexicographically within the same level.

# Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or diﬀerent racks. Communication between two machines on diﬀerent racks may cross one or more network switches. Addition- ally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to dis- tribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize net- work bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against diskor machine failures and fully utilizes each machine’s net- work bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will sur- vive and remain available even if an entire rack is damaged or oﬄine (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traﬃc, especially reads, for a chunk can exploit the aggre- gate bandwidth of multiple racks. On the other hand, write traﬃc has to ﬂow through multiple racks, a tradeoﬀ we make willingly.

# Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk cre- ation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several fac- tors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of “recent” creations on each chunkserver. Although creation itself is cheap, it reliably predicts immi- nent heavy write traﬃc because chunks are created when de- manded by writes, and in our append-once-read-many work- load they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-speciﬁed goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher prior- ity to a chunkthat has lost two replicas than to a chunkthat has lost only one. In addition, we prefer to ﬁrst re-replicate chunks for live ﬁles as opposed to chunks that belong to re- cently deleted ﬁles (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and “clones” it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traﬃc from overwhelming client traﬃc, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it ex- amines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually ﬁlls up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traﬃc that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which ex- isting replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

# Garbage Collection

After a ﬁle is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the ﬁle and chunk levels. We ﬁnd that this approach makes the system much simpler and more reliable.

## *Mechanism*

When a ﬁle is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the ﬁle is just renamed to a hidden name that includes the deletion times- tamp. During the master’s regular scan of the ﬁle system namespace, it removes any such hidden ﬁles if they have ex- isted for more than three days (the interval is conﬁgurable). Until then, the ﬁle can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden ﬁle is removed from the namespace, its in- memory metadata is erased. This eﬀectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identiﬁes orphaned chunks (i.e., those not reachable from any ﬁle) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master’s metadata. The chunkserver is free to delete its replicas of such chunks.

## *Discussion*

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of pro- gramming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the ﬁle- to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux ﬁles under designated directories on each chunkserver. Any such replica not known to the master is “garbage.”

The garbage collection approach to storage reclamation oﬀers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may suc- ceed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion mes- sages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver’s. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of names- paces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can re- spond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user eﬀort to ﬁne tune usage when stor- age is tight. Applications that repeatedly create and delete temporary ﬁles may not be able to reuse the storage right away. We address these issues by expediting storage recla- mation if a deleted ﬁle is explicitly deleted again. We also allow users to apply diﬀerent replication and reclamation policies to diﬀerent parts of the namespace. For example, users can specify that all the chunks in the ﬁles within some directory tree are to be stored without replication, and any deleted ﬁles are immediately and irrevocably removed from the ﬁle system state.

# Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to- date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notiﬁed and therefore before it can start writing to the chunk. If another replica is currently unavail- able, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the mas- ter assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it eﬀectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver veriﬁes the version number when it performs the operation so that it is always accessing up-to-date data.

# FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and

quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Com- ponent failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose prob- lems when they inevitably occur.

# High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet eﬀective strategies: fast recovery and replication.

## *Fast Recovery*

Both the master and the chunkserver are designed to re- store their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experi- ence a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Sec- tion 6.2.2 reports observed startup times.

## *Chunk Replication*

As discussed earlier, each chunk is replicated on multiple chunkservers on diﬀerent racks. Users can specify diﬀerent replication levels for diﬀerent parts of the ﬁle namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go oﬄine or detect corrupted replicas through checksum ver- iﬁcation (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redun- dancy such as parity or erasure codes for our increasing read- only storage requirements. We expect that it is challenging but manageable to implement these more complicated re- dundancy schemes in our very loosely coupled system be- cause our traﬃc is dominated by appends and reads rather than small random writes.

## *Master Replication*

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been ﬂushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, “shadow” masters provide read-only access to the ﬁle system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for ﬁles that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since ﬁle content is read from chunkservers, appli- cations do not observe stale ﬁle content. What could be

stale within short windows is ﬁle metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infre- quently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary’s decisions to create and delete replicas.

# Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guar- antee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by main- taining checksums.

A chunkis broken up into 64 KB blocks. Each has a corre- sponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver veriﬁes the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little eﬀect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for veriﬁcation. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the check- sum for the last partial checksum block, and compute new checksums for any brand new checksum blocks ﬁlled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the ﬁrst and last blocks of the range being overwritten, then perform the write, and

ﬁnally compute and record the new checksums. If we do not verify the ﬁrst and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corrup- tion in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

# Diagnostic Tools

Extensive and detailed diagnostic logging has helped im- measurably in problem isolation, debugging, and perfor- mance analysis, while incurring only a minimal cost. With- out logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate di- agnostic logs that record many signiﬁcant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without aﬀecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the ﬁle data being read or writ- ten. By matching requests with replies and collating RPC records on diﬀerent machines, we can reconstruct the en- tire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the beneﬁts) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

# MEASUREMENTS

In this section we present a few micro-benchmarks to illus- trate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

# Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this conﬁguration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are conﬁgured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

## *Reads*

*N* clients read simultaneously from the ﬁle system. Each client reads a randomly selected 4 MB region from a 320 GB ﬁle set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buﬀer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for *N* clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The eﬃciency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

## *Writes*

*N* clients write simultaneously to *N* distinct ﬁles. Each client writes 1 GB of data to a new ﬁle in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s be- cause we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or

2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three diﬀerent replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not sig- niﬁcantly aﬀect the aggregate write bandwidth delivered by the system to a large number of clients.

## *Record Appends*

Figure 3(c) shows record append performance. *N* clients append simultaneously to a single ﬁle. Performance is lim- ited by the network bandwidth of the chunkservers that store the last chunk of the ﬁle, independent of the num- ber of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by diﬀerent clients. Our applications tend to produce multiple such ﬁles con- currently. In other words, *N* clients append to *M* shared ﬁles simultaneously where both *N* and *M* are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a signiﬁcant issue in practice be- cause a client can make progress on writing one ﬁle while

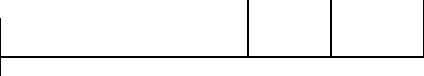
the chunkservers for another ﬁle are busy.

# Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hun- dred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much

|  |  |  |
| --- | --- | --- |
| Chunkservers  Available disk space | 342  72 TB | 227  180 TB |
| Used disk space | 55 TB | 155 TB |
| Number of Files | 735 k | 737 k |
| Number of Dead files | 22 k | 232 k |
| Number of Chunks | 992 k | 1550 k |
| Metadata at chunkservers | 13 GB | 21 GB |
| Metadata at master | 48 MB | 60 MB |

### Table 2: Characteristics of two GFS clusters



Cluster

A

B

longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single “task” consists of many processes on many machines reading and writing many ﬁles simultaneously.

* + 1. *Storage*

As shown by the ﬁrst ﬁve entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. “Used space” includes all chunk replicas. Virtually all ﬁles are replicated three times. Therefore, the clusters store 18 TB and 52 TB of ﬁle data respectively.

The two clusters have similar numbers of ﬁles, though B has a larger proportion of dead ﬁles, namely ﬁles which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its ﬁles tend to be larger.

## *Metadata*

The chunkservers in aggregate store tens of GBs of meta- data, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per ﬁle on average. This agrees with our assumption that the size of the master’s memory does not limit the system’s capacity in practice. Most of the per-ﬁle metadata is the ﬁle names stored in a preﬁx-compressed form. Other metadata includes ﬁle own- ership and permissions, mapping from ﬁles to chunks, and each chunk’s current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from diskbefore the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

## *Read and Write Rates*

Table 3 shows read and write rates for various time pe- riods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.

60

Network limit

Aggregate read rate

Network limit

Aggregate write rate

Network limit

Aggregate append rate

100 10

Read rate (MB/s)

Write rate (MB/s)

Append rate (MB/s)

40

50

0

0 5 10 15

Number of clients N

* + - 1. Reads

20

0

0 5 10 15

Number of clients N

* + - 1. Writes

5

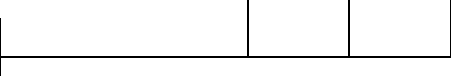
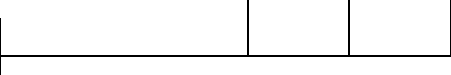
0

0 5 10 15

Number of clients N

* + - 1. Record appends

**Figure 3: Aggregate Throughputs.** Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% conﬁdence intervals, which are illegible in some cases because of low variance in measurements.



Write rate (last minute) Write rate (last hour) Write rate (since restart)

Master ops (last minute) Master ops (last hour) Master ops (since restart)

1. MB/s 101 MB/s
2. MB/s 117 MB/s

25 MB/s 13 MB/s

325 Ops/s 533 Ops/s

381 Ops/s 518 Ops/s

202 Ops/s 347 Ops/s

|  |  |  |
| --- | --- | --- |
| Cluster | A | B |
| Read rate (last minute) | 583 MB/s | 380 MB/s |
| Read rate (last hour) | 562 MB/s | 384 MB/s |
| Read rate (since restart) | 589 MB/s | 49 MB/s |

### Table 3: Performance Metrics for Two GFS Clusters

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network con- ﬁguration can support 750 MB/s, so it was using its re- sources eﬃciently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

## *Master Load*

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which con- tained hundreds of thousands of ﬁles) looking for particular ﬁles. We have since changed the master data structures to allow eﬃcient binary searches through the namespace. It can now easily support many thousands of ﬁle accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data struc- tures.

## *Recovery Time*

After a chunkserver fails, some chunks will become under- replicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about

15,000 chunks containing 600 GB of data. To limit the im- pact on running applications and provide leeway for schedul- ing decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most

6.25 MB/s (50 Mbps). All chunks were restored in 23.2 min- utes, at an eﬀective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

# Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identi- cal to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data process- ing.

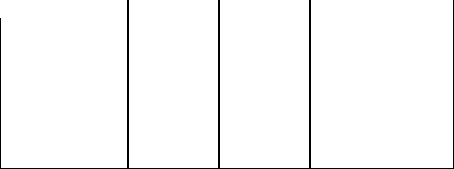
## *Methodology and Caveats*

These results include only client originated requests so that they reﬂect the workload generated by our applications for the ﬁle system as a whole. They do not include inter- server requests to carry out client requests or internal back- ground activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Ex- plicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to re- compile and restart thousands of running clients to do so and cumbersome to collect the results from as many ma- chines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual inﬂuence may also exist between general applications

**Table 4: Operations Breakdown by Size (%).** For reads, the size is the amount of data actually read and trans- ferred, rather than the amount requested.



Operation

Cluster

Read

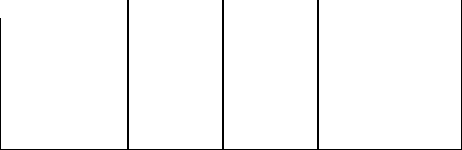
X Y

Write Record Append

X Y

X

Y



Operation

Cluster

1B..1K

1K..8K

Read

X Y

Write

X

Record Append

Y

X

*<* .1 *<* .1 *<* .1 *<* .1 *<* .1

13.8 3.9 *<* .1 *<* .1 *<* .1

Y

*<* .1

0.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0K | 0.4 | 2.6 | 0 | 0 | 0 | 0 |
| 1B..1K | 0.1 | 4.1 | 6.6 | 4.9 | 0.2 | 9.2 |
| 1K..8K | 65.2 38.5 | | 0.4 1.0 | | 18.9 | 15.2 |
| 8K..64K | 29.9 45.1 | | 17.8 43.0 | | 78.0 | 2.8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 8K..64K | 11.4 | 9.3 | 2.4 | 5.9 | 2.3 | 0.3 |
| 64K..128K | 0.3 | 0.7 | 0.3 | 0.3 | 22.7 | 1.2 |
| 128K..256K | 0.8 | 0.6 | 16.5 | 0.2 | *<* .1 | 5.8 |
| 256K..512K | 1.4 | 0.3 | 3.4 | 7.7 | *<* .1 | 38.4 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 64K..128K | 0.1 | 0.7 | 2.3 | 1.9 | *<* .1 | 4.3 |
| 128K..256K | 0.2 | 0.3 | 31.6 | 0.4 | *<* .1 | 10.6 |
| 256K..512K | 0.1 | 0.1 | 4.2 | 7.7 | *<* .1 | 31.2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 512K..1M | 65.9 55.1 | 74.1 58.0 | .1 | 46.8 |
| 1M..inf | 6.4 30.1 | 3.3 28.0 | 53.9 | 7.4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 512K..1M | 3.9 | 6.9 | 35.5 28.7 | 2.2 | 25.5 |
| 1M..inf | 0.1 | 1.8 | 1.5 12.3 | 0.7 | 2.2 |

and ﬁle systems, but the eﬀect is likely more pronounced in our case.

## *Chunkserver Workload*

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (un- der 64 KB) come from seek-intensive clients that look up small pieces of data within huge ﬁles. The large reads (over

512 KB) come from long sequential reads through entire ﬁles.

A signiﬁcant number of reads return no data at all in clus- ter Y. Our applications, especially those in the production systems, often use ﬁles as producer-consumer queues. Pro- ducers append concurrently to a ﬁle while a consumer reads the end of ﬁle. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from signiﬁcant buﬀer- ing within the writers. Writers that buﬀer less data, check- point or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher per- centage of large record appends than cluster X does because our production systems, which use cluster Y, are more ag- gressively tuned for GFS.

Table 5 shows the total amount of data transferred in op- erations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but signiﬁcant portion of the read data be- cause of the random seek workload.

## *Appends versus Writes*

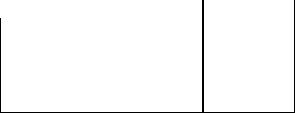
Record appends are heavily used especially in our pro- duction systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ra- tios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two appli- cations with particular buﬀer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This ap-

**Table 5: Bytes Transferred Breakdown by Opera- tion Size (%).** For reads, the size is the amount of data actually read and transferred, rather than the amount re- quested. The two may diﬀer if the read attempts to read beyond end of ﬁle, which by design is not uncommon in our workloads.

|  |  |
| --- | --- |
| Cluster | X Y |
| Open | 26.1 16.3 |
| Delete | 0.7 1.5 |
| FindLocation | 64.3 65.8 |
| FindLeaseHolder | 7.8 13.4 |
| FindMatchingFiles | 0.6 2.2 |
| All other combined | 0.5 0.8 |

### Table 6: Master Requests Breakdown by Type (%)



proximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

## *Master Workload*

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLo- cation*) for reads and lease holder information (*FindLease- Locker*) for data mutations.

Clusters X and Y see signiﬁcantly diﬀerent numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this diﬀerence is further hidden in the diﬀerence in *Open* requests because an old version of a ﬁle may be implicitly deleted by being opened for write from scratch (mode “w” in Unix open terminology).

*FindMatchingFiles* is a pattern matching request that sup- ports “ls” and similar ﬁle system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the ﬁle system to understand global appli- cation state. In contrast, cluster X’s applications are under more explicit user control and usually know the names of all needed ﬁles in advance.

# EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend ﬁle system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production sys- tems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact re- sponded reliably only to the more recent ones. Since the pro- tocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive’s state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data cor- ruption, while concurrently we modiﬁed the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of fsync(). Its cost is proportional to the size of the ﬁle rather than the size of the modiﬁed portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated

to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modiﬁes the address space in an mmap() call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Even-

tually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by

replacing mmap() with pread() at the cost of an extra copy. Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel

and share the changes with the open source community.

# RELATED WORK

Like other large distributed ﬁle systems such as AFS [5], GFS provides a location independent namespace which en- ables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a ﬁle’s data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS cur- rently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the ﬁle system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed ﬁle systems like Frangipani, xFS, Min- nesota’s GFS[11] and GPFS [10] remove the centralized server

and rely on distributed algorithms for consistency and man- agement. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain ﬂexibil- ity. In particular, a centralized master makes it much easier to implement sophisticated chunkplacement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully repli- cated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persis- tent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to pro- vide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simpliﬁed the problem signiﬁ- cantly by focusing on the needs of our applications rather than building a POSIX-compliant ﬁle system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated ﬁxed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota’s GFS and NASD, we do not seek to alter the model of the storage device. We focus on ad- dressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues dis- tributed across machines and careful data ﬂow control, GFS uses a persistent ﬁle that can be appended to concurrently by many producers. The River model supports m-to-n dis- tributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues eﬃciently. Multiple consumers can read the same ﬁle, but they must coordinate to partition the incoming load.

# CONCLUSIONS

The Google File System demonstrates the qualities es- sential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are speciﬁc to our unique setting, many may apply to data pro- cessing tasks of a similar magnitude and cost consciousness. We started by reexamining traditional ﬁle system assump- tions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically diﬀerent points in the design space. We treat component failures as the norm rather than the exception, optimize for huge ﬁles that are mostly appended to (perhaps concurrently) and then read (usually sequen- tially), and both extend and relax the standard ﬁle system

interface to improve the overall system.

Our system provides fault tolerance by constant moni- toring, replicating crucial data, and fast and automatic re- covery. Chunk replication allows us to tolerate chunkserver

failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently re- pairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating ﬁle system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involve- ment in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to pri- mary replicas in data mutations. This makes possible a sim- ple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

# ACKNOWLEDGMENTS

We wish to thankthe following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeﬀ Dean, and David des- Jardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Ed- jlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new ﬁle system and gave us useful feedback. Yoshka helped with early testing.

# REFERENCES

1. Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network ﬁle systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.
2. Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS ’99)*, pages 10–22, Atlanta, Georgia, May 1999.
3. Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.
4. Garth A. Gibson, David F. 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 *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

1. John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed ﬁle system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
2. InterMezzo. [http://www.inter-mezzo.org,](http://www.inter-mezzo.org/) 2003.
3. Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp ﬁle system. In *13th Symposium on Operating System Principles*, pages 226–238, Paciﬁc Grove, CA, October 1991.
4. Lustre. http://www.lustreorg, 2003.
5. David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.
6. Frank Schmuck and Roger Haskin. GPFS: A

shared-disk ﬁle system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.

1. Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O’Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.
2. Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed ﬁle system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237,

Saint-Malo, France, October 1997.