

UNIT-6 Storage Systems

8.1	The Evolution of Storage Technology
8.2	Storage Models, File Systems, and Databases
8.3	Distributed File Systems: The Precursors
8.4	General Parallel File system.....
8.5	Google File System
8.6	<i>ApacheHadoop</i>
8.7	Locks and <i>Chubby</i> : A Locking Service
8.8	Transaction Processing and <i>NoSQL</i> Databases
8.9	<i>BigTable</i>
8.10	<i>Megastore</i>
8.11	<i>Amazon Simple Storage Service(S3)</i>

Storage Systems

A cloud provides the vast amounts of storage and computing cycles demanded by many applications. The network-centric content model allows a user to access data stored on a cloud from any device connected to the Internet. Mobile devices with limited power reserves and local storage take advantage of cloud environments to store audio and video files. Clouds provide an ideal environment for multimedia content delivery.

A variety of sources feed a continuous stream of data to cloud applications. An ever-increasing number of cloud-based services collect detailed data about their services and information about the users of these services. Then the service providers use the clouds to analyze that data.

Storage and processing on the cloud are intimately tied to one another; indeed, sophisticated strategies to reduce the access time and to support real-time multimedia access are necessary to satisfy the requirements of content delivery. On the other hand, most cloud applications process very large amounts of data; effective data replication and storage management strategies are critical to the computations performed on the cloud.

A new concept, “*big data*,” reflects the fact that many applications use data sets so large that they cannot be stored and processed using local resources. The consensus is that “big data” growth can be viewed as a three-dimensional phenomenon; it implies an increased volume of data, requires an increased processing speed to process more data and produce more results, and at the same time it involves a diversity of data sources and data types.

Applications in many areas of science, including genomics, structural biology, high-energy physics, astronomy, meteorology, and the study of the environment, carry out complex analysis of data sets, often of the order of terabytes.¹ In 2010, the four main detectors at the Large Hadron Collider (LHC) produced 13 PB of data; the Sloan Digital Sky Survey (SDSS) collects about 200 GB of data per night. As a result of this increasing appetite for data, file systems such as Btrfs, XFS, ZFS, exFAT, NTFS, HFS Plus, and ReFS support disk formats with theoretical volume sizes of several exabytes.

8.1 The evolution of storage technology

The technological capacity to store information has grown over time at an accelerated pace:

- 1986: 2.6 EB; equivalent to less than one 730 MB CD-ROM of data per computer user.
- 1993: 15.8 EB; equivalent to four CD-ROMs per user.
- 2000: 54.5 EB; equivalent to 12 CD-ROMs per user.
- 2007: 295 EB; equivalent to almost 61 CD-ROMs per user.

Though it pales in comparison with processor technology, the evolution of storage technology is astounding. A 2003 study shows that during the 1980–2003 period the storage density of hard disk drives (HDD) increased by four orders of magnitude, from about 0.01 Gb/in² to about 100 Gb/in². During the same period the prices fell by five orders of magnitude, to about 1 cent/Mbyte. HDD densities are projected to climb to 1,800 Gb/in² by 2016, up from 744 Gb/in² in 2011.

The density of Dynamic Random Access Memory (DRAM) increased from about 1 Gb/in² in 1990 to 100 Gb/in² in 2003. The cost of DRAM tumbled from about \$80/MB to less than \$1/MB during the same period. In 2010 Samsung announced the first monolithic, 4 gigabit, low-power, double-data-rate (LPDDR2) DRAM using a 30 nm process.

These rapid technological advancements have changed the balance between initial investment in storage devices and system management costs. Now the cost of storage management is the dominant element of the total cost of a storage system. This effect favors the centralized storage strategy supported by a cloud; indeed, a centralized approach can automate some of the storage management functions, such as replication and backup, and thus reduce substantially the storage management cost.

While the density of storage devices has increased and the cost has decreased dramatically, the access time has improved only slightly. The performance of I/O subsystems has not kept pace with the performance of computing engines, and that affects multimedia, scientific and engineering, and other modern applications that process increasingly large volumes of data.

The storage systems face substantial pressure because the volume of data generated has increased exponentially during the past few decades; whereas in the 1980s and 1990s data was primarily generated by humans, nowadays machines generate data at an unprecedented rate. Mobile devices, such as smart-phones and tablets, record static images, as well as movies and have limited local storage capacity, so they transfer the data to cloud storage systems. Sensors, surveillance cameras, and digital medical imaging devices generate data at a high rate and dump it onto storage systems

accessible via the Internet. Online digital libraries, ebooks, and digital media, along with reference data, add to the demand for massive amounts of storage.

As the volume of data increases, new methods and algorithms for data mining that require powerful computing systems have been developed. Only a concentration of resources could provide the CPU cycles along with the vast storage capacity necessary to perform such intensive computations and access the very large volume of data.

Although we emphasize the advantages of a concentration of resources, we have to be acutely aware that a cloud is a large-scale distributed system with a very large number of components that must work in concert. The management of such a large collection of systems poses significant challenges and requires novel approaches to systems design. Case in point: Although the early distributed file systems used custom-designed reliable components, nowadays large-scale systems are built with off-the-shelf components. The emphasis of the design philosophy has shifted from *performance at any cost* to *reliability at the lowest possible cost*. This shift is evident in the evolution of ideas, from the early distributed file systems of the 1980s, such as the Network File System (NFS) and the Andrew File System (AFS), to today's Google File System (GFS) and the *Megastore*.

8.2 Storage models, file systems, and databases

A *storage model* describes the layout of a data structure in physical storage; a *data model* captures the most important logical aspects of a data structure in a database. The physical storage can be a local disk, a removable media, or storage accessible via a network.

Two abstract models of storage are commonly used: *cell storage* and *journal storage*. Cell storage assumes that the storage consists of cells of the same size and that each object fits exactly in one cell. This model reflects the physical organization of several storage media; the primary memory of a computer is organized as an array of memory cells, and a secondary storage device (e.g., a disk) is organized in sectors or blocks read and written as a unit. *read/write coherence* and *before-or-after atomicity* are two highly desirable properties of any storage model and in particular of cell storage (see Figure 8.1).

Journal storage is a fairly elaborate organization for storing composite objects such as records consisting of multiple fields. Journal storage consists of a *manager* and *cell storage*, where the entire history of a variable is maintained, rather than just the current value. The user does not have direct access to the *cell storage*; instead the user can request the *journal manager* to (i) start a new action; (ii) read the value of a cell; (iii) write the value of a cell; (iv) commit an action; or (v) abort an action. The *journal*

manager translates user requests to commands sent to the cell storage: (i) read a cell; (ii) write a cell; (iii) allocate a cell; or (iv) deallocate a cell.

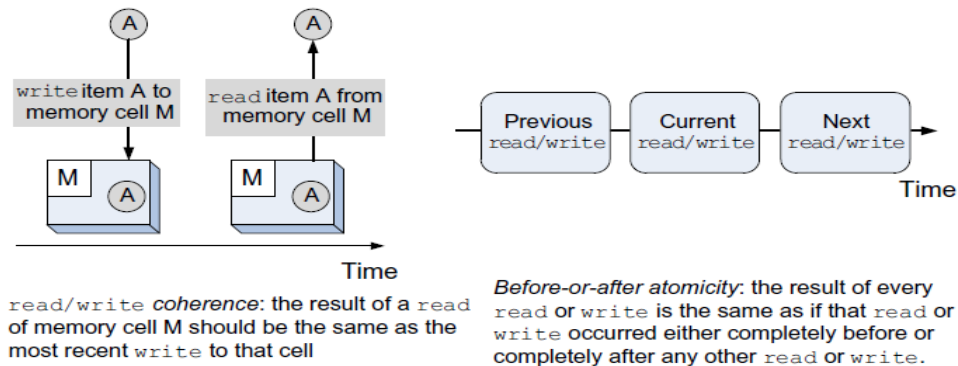


FIGURE 8.1: Illustration capturing the semantics of read/write coherence and before-or-after atomicity.

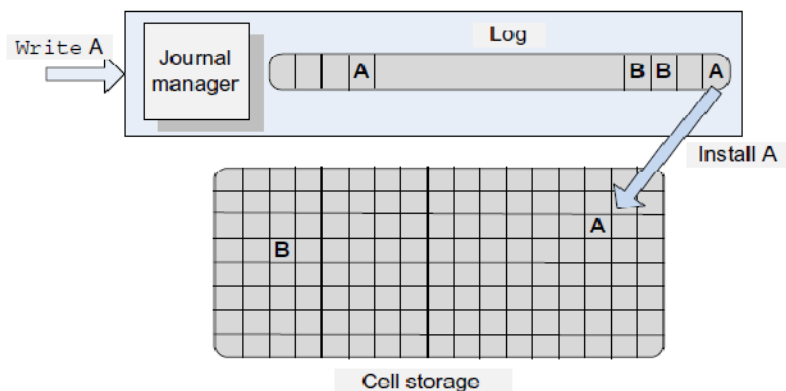


FIGURE 8.2: A log contains the entire history of all variables. The log is stored on nonvolatile media of *journal storage*. If the system fails after the new value of a variable is stored in the log but before the value is stored in cell memory, then the value can be recovered from the log. If the system fails while writing the log, the cell memory is not updated. This guarantees that all actions are *all-or-nothing*. Two variables, A and B, in the log and cell storage are shown. A new value of A is written first to the log and then *installed* on cell memory at the unique address assigned to A.

In the context of storage systems, a *log* contains a history of all variables in *cell storage*. The information about the updates of each data item forms a record appended at the end of the log. A log provides authoritative information about the outcome of an action involving *cell storage*; the cell storage can be reconstructed using the log, which can be easily accessed – we only need a pointer to the last record.

An *all-or-nothing* action first records the action in a *log* in *journal storage* and then *installs* the change in the *cell storage* by overwriting the previous version of a data item (see Figure 8.2). The *log* is always kept on nonvolatile storage (e.g., disk) and the

considerably larger *cell storage* resides typically on nonvolatile memory, but can be held in memory for real-time access or using a write-through cache.

Many cloud applications must support online transaction processing and have to guarantee the correctness of the transactions. Transactions consist of multiple actions; for example, the transfer of funds from one account to another requires withdrawing funds from one account and crediting it to another. The system may fail during or after each one of the actions, and steps to ensure correctness must be taken. Correctness of a transaction means that the result should be guaranteed to be the same as though the actions were applied one after another, regardless of the order. More stringent conditions must sometimes be observed; for example, banking transactions must be processed in the order in which they are issued, the so-called *external time consistency*. To guarantee correctness, a transaction-processing system supports *all-or-nothing atomicity*, discussed in Section 2.10.

A *file system* consists of a collection of *directories*. Each directory provides information about a set of files. Today high-performance systems can choose among three classes of file system: networks file systems (NFSs), storage area networks (SANs), and parallel file systems (PFSs). The NFS is very popular and has been used for some time, but it does not scale well and has reliability problems; an NFS server could be a single point of failure.

Advances in networking technology allow the separation of storage systems from computational servers; the two can be connected by a SAN. SANs offer additional flexibility and allow cloud servers to deal with nondisruptive changes in the storage configuration. Moreover, the storage in a SAN can be *pooled* and then allocated based on the needs of the servers; pooling requires additional software and hardware support and represents another advantage of a centralized storage system. A SAN-based implementation of a file system can be expensive, since each node must have a Fibre Channel adapter to connect to the network.

Parallel file systems are scalable, are capable of distributing files across a large number of nodes, and provide a global naming space. In a parallel data system, several I/O nodes serve data to all computational nodes and include a metadata server that contains information about the data stored in the I/O nodes. The interconnection network of a parallel file system could be a SAN.

Most cloud applications do not interact directly with file systems but rather through an application layer that manages a database. A *database* is a collection of logically related records. The software that controls the access to the database is called a *database management system (DBMS)*. The main functions of a DBMS are to enforce data integrity, manage data access and concurrency control, and support recovery after a failure.

A DBMS supports a *query language*, a dedicated programming language used to develop database applications. Several database models, including the navigational model of the 1960s, the relational model of the 1970s, the object-oriented model of the 1980s, and the *NoSQL* model of the first decade of the 2000s, reflect the limitations of the hardware available at the time and the requirements of the most popular applications of each period.

Most cloud applications are data intensive and test the limitations of the existing infrastructure. For example, they demand DBMSs capable of supporting rapid application development and short time to market. At the same time, cloud applications require low latency, scalability, and high availability and demand a consistent view of the data.

These requirements cannot be satisfied simultaneously by existing database models; for example, relational databases are easy to use for application development but do not scale well. As its name implies, the *NoSQL* model does not support SQL as a query language and may not guarantee the *atomicity, consistency, isolation, durability* (ACID) properties of traditional databases. *NoSQL* usually guarantees the eventual consistency for transactions limited to a single data item. The *NoSQL* model is useful when the structure of the data does not require a relational model and the amount of data is very large. Several types of *NoSQL* database have emerged in the last few years. Based on the way the *NoSQL* databases store data, we recognize several types, such as key-value stores, *BigTable* implementations, document store databases, and graph databases.

Replication, used to ensure fault tolerance of large-scale systems built with commodity components, requires mechanisms to guarantee that all replicas are consistent with one another. This is another example of increased complexity of modern computing and communication systems due to physical characteristics of components, a topic discussed in Chapter 10. Section 8.7 contains an in-depth analysis of a service implementing a consensus algorithm to guarantee that replicated objects are consistent.

8.3 Distributed file systems: The precursors

In this section we discuss the first distributed file systems, developed in the 1980s by software companies and universities. The systems covered are the Network File System developed by Sun Microsystems in 1984, the Andrew File System developed at Carnegie Mellon University as part of the Andrew project, and the Sprite Network File System developed by John Osterhout's group at UC Berkeley as a component of the *Unix*-like distributed operating system called Sprite. Other systems developed at about the same time are Locus, Apollo, and the Remote File System (RFS). The main concerns in the design of these systems were scalability, performance, and security (see Table 8.1.)

In the 1980s many organizations, including research centers, universities, financial institutions, and design centers, considered networks of workstations to be an ideal environment for their operations. Diskless workstations were appealing due to reduced hardware costs and because of lower maintenance and system administration costs. Soon it became obvious that a distributed file system could be very useful for the management of a large number of workstations. Sun Microsystems, one of the main promoters of distributed systems based on workstations, proceeded to develop the NFS in the early 1980s.

Network File System (NFS). NFS was the first widely used distributed file system; the development of this application based on the client-server model was motivated by the need to share a file system among a number of clients interconnected by a local area network.

A majority of workstations were running under *Unix*; thus, many design decisions for the NFS were influenced by the design philosophy of the *Unix File System* (UFS). It is not surprising that the NFS designers aimed to:

- Provide the same semantics as a local UFS to ensure compatibility with existing applications.
 - Facilitate easy integration into existing UFS.
 - Ensure that the system would be widely used and thus support clients running on different operating systems.
 - Accept a modest performance degradation due to remote access over a network with a bandwidth of several Mbps.

Table 8.1 A comparison of several network file systems [250].

File System	Cache Size and Location	Writing Policy	Consistency Guarantees	Cache Validation
NFS	Fixed, memory	On close or 30 s delay	Sequential	On open, with server consent
AFS	Fixed, disk	On close	Sequential	When modified server asks client
Sprite	Variable, memory	30 s delay	Sequential, concurrent	On open, with server consent
Locus	Fixed, memory	On close	Sequential, concurrent	On open, with server consent
Apollo	Variable, memory	Delayed or on unlock	Sequential	On open, with server consent
RFS	Fixed, memory	Write-through	Sequential, concurrent	On open, with server consent

Before we examine NFS in more detail, we have to analyze three important characteristics of the *Unix* File System that enabled the extension from local to remote file management:

- The layered design provides the necessary *flexibility* for the file system; layering allows separation of concerns and minimization of the interaction among the modules necessary to implement the system. The addition of the *vnode* layer allowed the *Unix* File System to treat local and remote file access uniformly.
- The hierarchical design supports *scalability* of the file system; indeed, it allows grouping of files into special files called *directories* and supports multiple levels of directories and collections of directories and files, the so-called *file systems*. The hierarchical file structure is reflected by the file-naming convention.
- The metadata supports a systematic rather than an ad hoc design philosophy of the file system. The so called *inodes* contain information about individual files and directories. The inodes are kept on persistent media, together with the data. Metadata includes the file owner, the access rights, the creation time or the time of the last modification of the file, the file size, and information about the structure of the file and the persistent storage device cells where data is stored. Metadata also supports device independence, a very important objective due to the very rapid pace of storage technology development.

The *logical organization* of a file reflects the data model – the view of the data from the perspective of the application. The *physical organization* reflects the storage model and describes the manner in which the file is stored on a given storage medium. The layered design allows UFS to separate concerns for the physical file structure from the logical one.

Recall that a *file* is a linear array of cells stored on a persistent storage device. The *file pointer* identifies a cell used as a starting point for a read or write operation. This linear array is viewed by an application as a collection of logical records; the file is stored on a physical device as a set of physical records, or blocks, of a size dictated by the physical media.

The lower three layers of the UFS hierarchy – the block, the file, and the inode layer – reflect the physical organization. The block layer allows the system to locate individual blocks on the physical device; the file layer reflects the organization of blocks into files; and the inode layer provides the metadata for the objects (files and directories). The upper three layers – the path name, the absolute path name, and the symbolic path name layer – reflect the logical organization. The file-name layer mediates between the machine-oriented and the user-oriented views of the file system (see Figure 8.3).

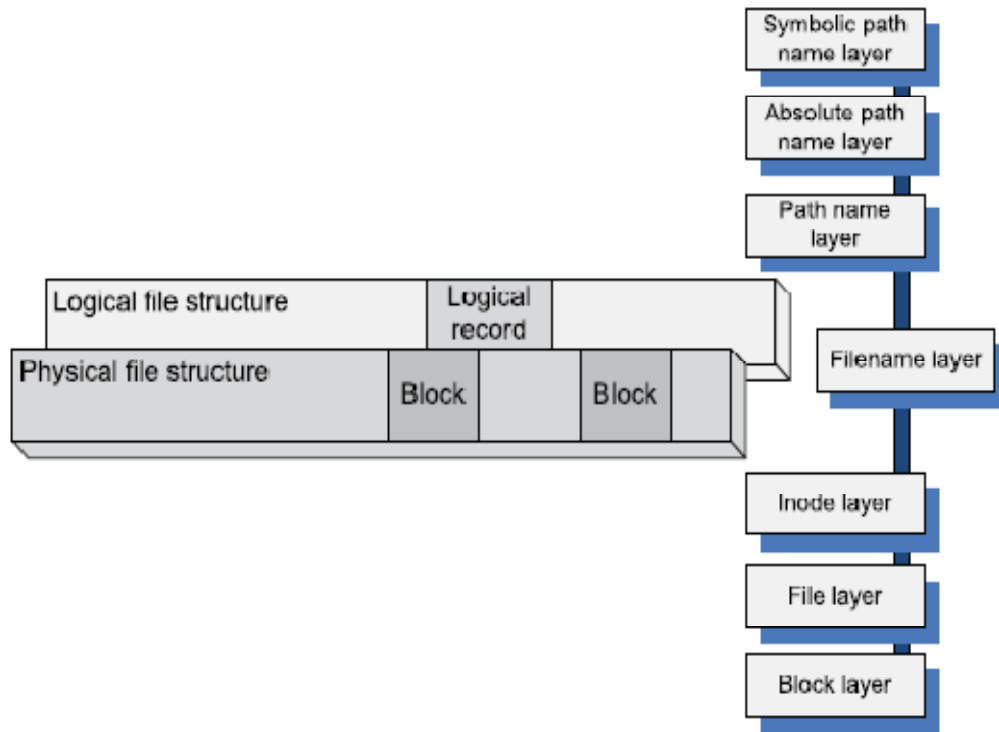


FIGURE 8.3: The layered design of the *Unix* File System separates the physical file structure from the logical one. The lower three layers – block, file, and inode – are related to the physical file structure, while the upper three layers – path name, absolute path name, and symbolic path name – reflect the logical organization. The filename layer mediates between the two.

Several *control structures* maintained by the kernel of the operating system support file handling by a running process. These structures are maintained in the user area of the process address space and can only be accessed in kernel mode. To access a file, a process must first establish a connection with the file system by opening the file. At that time a new entry is added to the *file description table*, and the meta-information is brought into another control structure, the *open file table*.

A *path* specifies the location of a file or directory in a file system; a *relative path* specifies this location relative to the current/working directory of the process, whereas a *full path*, also called an *absolute path*, specifies the location of the file independently of the current directory, typically relative to the root directory. A local file is uniquely identified by a *file descriptor (fd)*, generally an index in the open file table.

The Network File System is based on the client-server paradigm. The client runs on the local host while the server is at the site of the remote file system, and they

interact by means of remote procedure calls (RPCs) (see Figure 8.4). The API interface of the local file system distinguishes file operations on a local file from the ones on a remote file and, in the latter case, invokes the RPC client. Figure 8.5 shows the API for a *Unix File System*, with the calls made by the RPC client in response to API calls issued by a user program for a remote file system as well as some of the actions carried out by the NFS server in response to an RPC call. NFS uses a *vnode* layer to distinguish between operations on local and remote files, as shown in Figure 8.4.

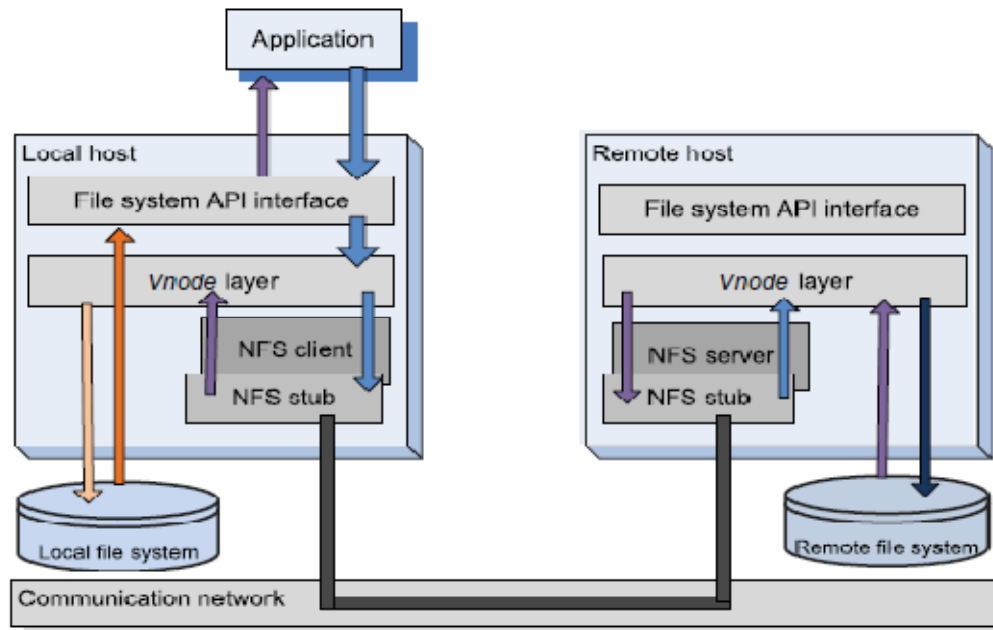


FIGURE 8.4: The NFS client-server interaction. The *vnode* layer implements file operation in a uniform manner, regardless of whether the file is local or remote. An operation targeting a local file is directed to the local file system, whereas one for a remote file involves NFS. An NFS client packages the relevant information about the target and the NFS server passes it to the *vnode* layer on the remote host, which, in turn, directs it to the remote file system.

A remote file is uniquely identified by a *file handle (fh)* rather than a file descriptor. The file handle is a 32-byte internal name, a combination of the file system identification, an inode number, and a generation number. The file handle allows the system to locate the remote file system and the file on that system; the generation number allows the system to reuse the inode numbers and ensures correct semantics when multiple clients operate on the same remote file.

Although many RPC calls, such as read, are idempotent,³ communication failures could sometimes lead to unexpected behavior. Indeed, if the network fails to deliver the response to a read RPC, then the call can be repeated without any side effects. By contrast, when the network fails to deliver the response to the *rmdir* RPC, the second call returns an error code to the user if the call was successful the first time. If the server fails to execute the first call, the

second call returns normally. Note also that there is no close RPC because this action only makes changes in the process open file structure and does not affect the remote file.

The NFS has undergone significant transformations over the years. It has evolved from Version 2 , discussed in this section, to Version 3 in 1994 and then to Version 4 in 2000.

Andrew File System (AFS). AFS is a distributed file system developed in the late 1980s at Carnegie Mellon University (CMU) in collaboration with IBM . The designers of the system envisioned a very large number of workstations interconnected with a relatively small number of servers; it was anticipated that each individual at CMU would have an Andrew workstation, so the system would connect up to 10,000 workstations. The set of trusted servers in AFS forms a structure called Vice. The OS on a workstation, 4.2 BSD *Unix*, intercepts file system calls and forwards them to a user-level process called Venus, which caches files from Vice and stores modified copies of files back on the servers they came from. Reading and writing from/to a file are performed directly on the cached copy and bypass Venus; only when a file is opened or closed does Venus communicate with Vice.

The emphasis of the AFS design was on performance, security, and simple management of the file system. To ensure scalability and to reduce response time, the local disks of the workstations are used as persistent cache. The master copy of a file residing on one of the servers is updated only when the file is modified. This strategy reduces the load placed on the servers and contributes to better system performance.

Another major objective of the AFS design was improved security. The communications between clients and servers are encrypted, and all file operations require secure network connections. When a user signs into a workstation, the password is used to obtain security tokens from an authentication server. These tokens are then used every time a file operation requires a secure network connection.

The AFS uses *access control lists* (ACLs) to allow control sharing of the data. An ACL specifies the access rights of an individual user or group of users. A set of tools supports ACL management. Another facet of the effort to reduce user involvement in file management is *location transparency*. The files can be accessed from any location and can be moved automatically or at the request of system administrators without user involvement and/or inconvenience. The relatively small number of servers drastically reduces the efforts related to system administration because operations, such as backups, affect only the servers, whereas workstations can be added, removed, or moved from one location to another without administrative intervention.

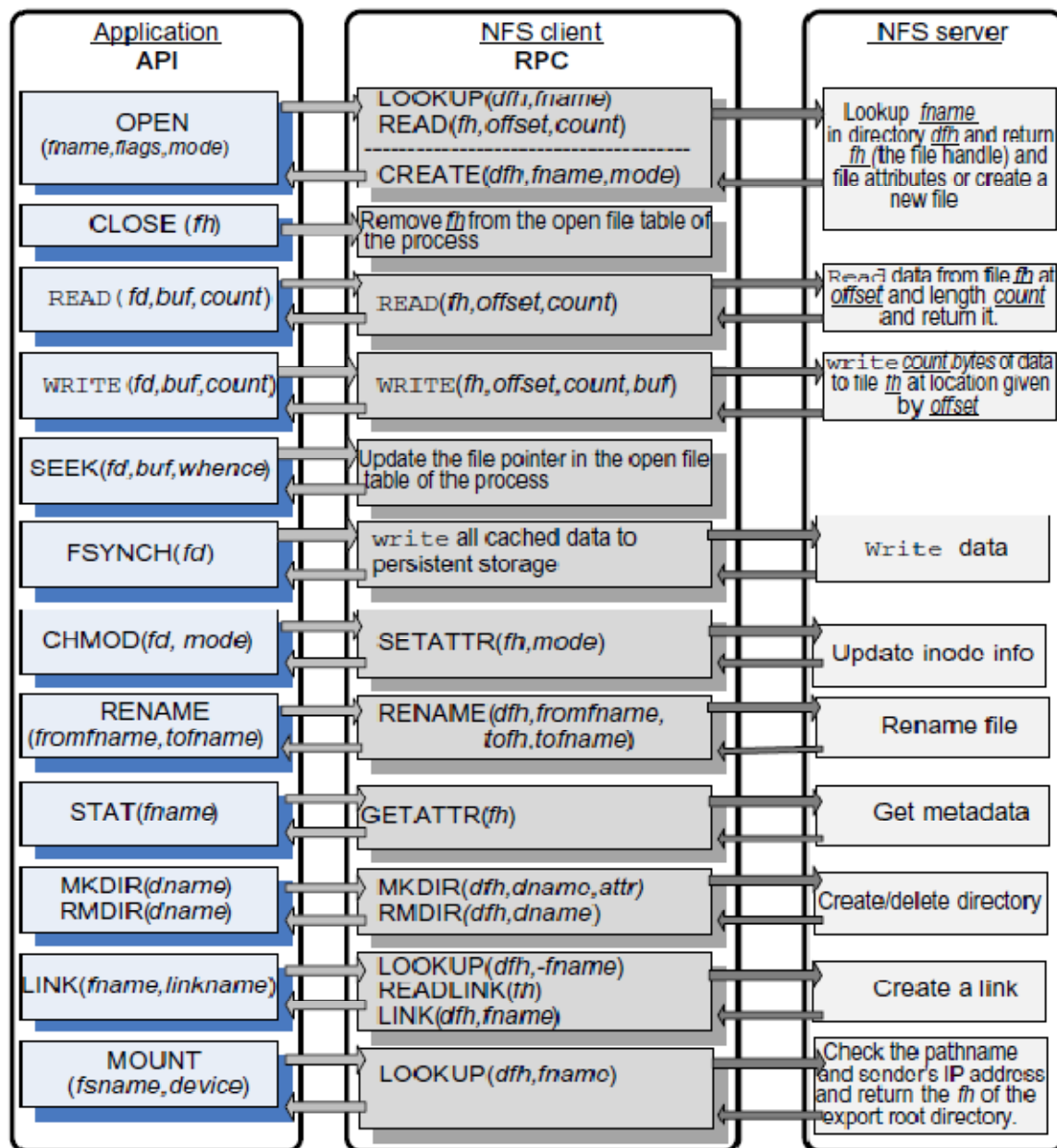


FIGURE 8.5: The API of the *Unix* File System and the corresponding RPC issued by an NFS client to the NFS server. The actions of the server in response to an RPC issued by the NFS client are too complex to be fully described. *fd* stands for file descriptor, *fh* for file handle, *fname* for filename, *dname* for directory name, *dfh* for the directory where the file handle can be found, *count* for the number of bytes to be transferred, *buf* for the buffer to transfer the data to/from, and *device* for the device on which the file system is located *fsname* (stands for files system name).

Sprite Network File System (SFS). SFS is a component of the Sprite network operating system. SFS supports non-write-through caching of files on the client as well as the server systems. Processes running on all workstations enjoy the same semantics for file access as they would if they were run on a single system. This is possible due to a cache consistency mechanism that flushes portions of the cache and disables caching for shared files opened for read/write operations.

Caching not only hides the network latency, it also reduces server utilization and obviously improves performance by reducing response time. A file access request made by a client process could be satisfied at different levels. First, the request is directed to the local cache; if it's not satisfied there, it is passed to the local file system of the client. If it cannot be satisfied locally then the request is sent to the remote server. If the request cannot be satisfied by the remote server's cache, it is sent to the file system running on the server.

The design decisions for the Sprite system were influenced by the resources available at a time when a typical workstation had a 1–2 MIPS processor and 4–14 Mbytes of physical memory. The main-memory caches allowed diskless workstations to be integrated into the system and enabled the development of unique caching mechanisms and policies for both clients and servers. The results of a file-intensive benchmark report show that SFS was 30–35% faster than either NFS or AFS.

The file cache is organized as a collection of 4 KB blocks; a cache block has a virtual address consisting of a unique file identifier supplied by the server and a block number in the file. Virtual addressing allows the clients to create new blocks without the need to communicate with the server. File servers map virtual to physical disk addresses. Note that the page size of the virtual memory in Sprite is also 4K.

The size of the cache available to an SFS client or a server system changes dynamically as a function of the needs. This is possible because the Sprite operating system ensures optimal sharing of the physical memory between file caching by SFS and virtual memory management.

The file system and the virtual memory manage separate sets of physical memory pages and maintain a time of last access for each block or page, respectively. Virtual memory uses a version of the clock algorithm to implement a least recently used (LRU) page replacement algorithm, and the file system implements a strict LRU order, since it knows the time of each read and write operation. Whenever the file system or the virtual memory management experiences a file cache miss or a page fault, it compares the age of its oldest cache block or page, respectively, with the age of the oldest one of the other system; the oldest cache block or page is forced to release the real memory frame.

An important design decision related to the SFS was to *delay write-backs*; this means that a block is first written to cache, and the writing to the disk is delayed for a time of the order of tens of seconds. This strategy speeds up writing and avoids

writing when the data is discarded before the time to write it to the disk. The obvious drawback of this policy is that data can be lost in case of a system failure. *Write-through* is the alternative to the delayed write-back; it guarantees reliability because the block is written to the disk as soon as it is available on the cache, but it increases the time for a write operation.

Most network file systems guarantee that once a file is closed, the server will have the newest version on persistent storage. As far as concurrency is concerned, we distinguish *sequential write sharing*, when a file cannot be opened simultaneously for reading and writing by several clients, from *concurrent write sharing*, when multiple clients can modify the file at the same time. Sprite allows both modes of concurrency and delegates the cache consistency to the servers. In case of concurrent write sharing, the client caching for the file is disabled; all reads and writes are carried out through the server.

8.4 General Parallel File System

Parallel I/O implies execution of multiple input/output operations concurrently. Support for parallel I/O is essential to the performance of many applications. Therefore, once distributed file systems became ubiquitous, the natural next step in the evolution of the file system was to support parallel access. Parallel file systems allow multiple clients to read and write concurrently from the same file.

Concurrency control is a critical issue for parallel file systems. Several semantics for handling the shared access are possible. For example, when the clients share the file pointer, successive reads issued by multiple clients advance the file pointer; another semantic is to allow each client to have its own file pointer. Early supercomputers such as the Intel Paragon⁴ took advantage of parallel file systems to support applications based on the same program, multiple data (SPMD) paradigm.

The General Parallel File System (GPFS) was developed at IBM in the early 2000s as a successor to the TigerShark multimedia file system. GPFS is a parallel file system that emulates closely the behavior of a general-purpose POSIX system running on a single system. GPFS was designed for optimal performance of large clusters; it can support a file system of up to 4 PB consisting of up to 4, 096 disks of 1 TB each .

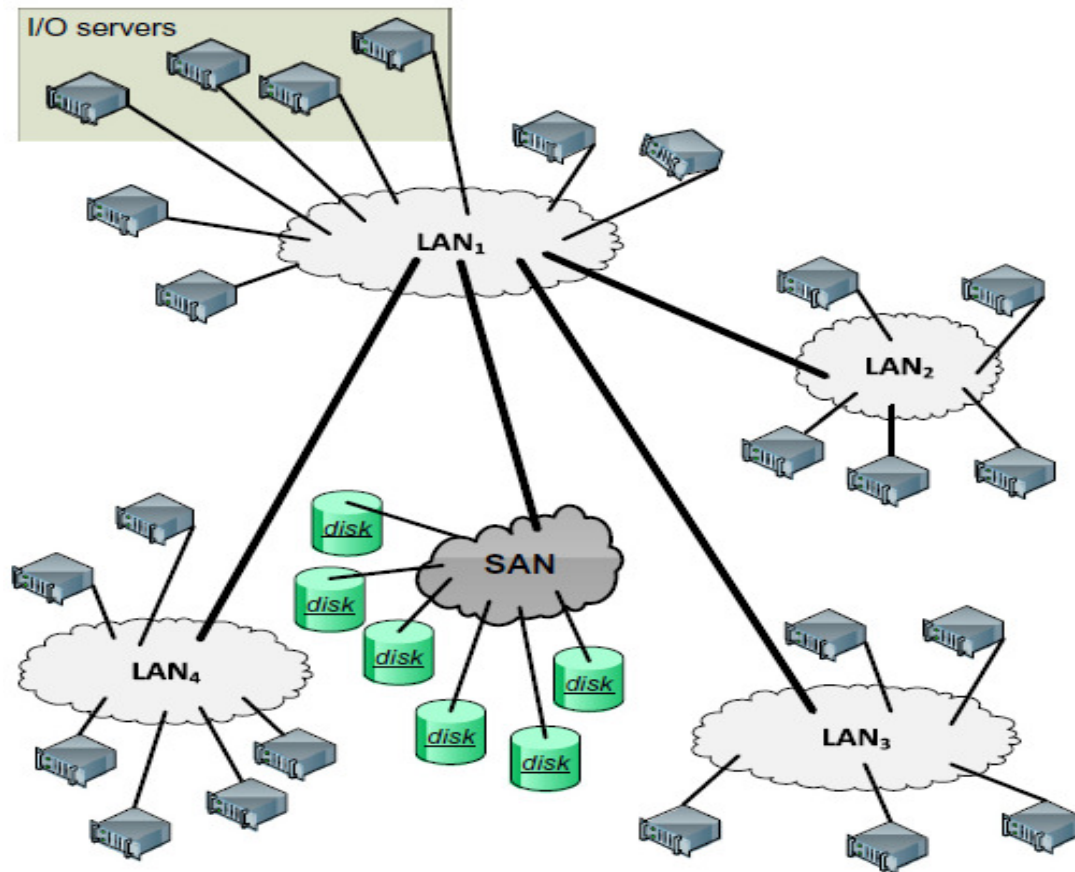


FIGURE 8.6: A GPFS configuration. The disks are interconnected by a SAN and compute servers are distributed in four LANs, LAN₁–LAN₄. The I/O nodes/servers are connected to LAN₁.

The maximum file size is $(2^{63} - 1)$ bytes. A file consists of blocks of equal size, ranging from 16 KB to 1 MB striped across several disks. The system could support not only very large files but also a very large number of files. The directories use *extensible hashing* techniques⁵ to access a file. The system maintains user data, file metadata such as the time when last modified, and file system metadata such as allocation maps. Metadata, such as file attributes and data block addresses, is stored in inodes and indirect blocks.

Reliability is a major concern in a system with many physical components. To recover from system failures, GPFS records all metadata updates in a *write-ahead* log file. *Write-ahead* means that updates are written to persistent storage only after the log records have been written. For example, when a new file is created, a directory block must be updated and an inode for the file must be created. These records are

transferred from cache to disk after the log records have been written. When the directory block is written and then the I/O node fails before writing the inode, then the system ends up in an inconsistent state and the log file allows the system to recreate the inode record.

The log files are maintained by each I/O node for each file system it mounts; thus, any I/O node is able to initiate recovery on behalf of a failed node. Disk parallelism is used to reduce access time. Multiple I/O read requests are issued in parallel and data is prefetched in a buffer pool.

Data striping allows concurrent access and improves performance but can have unpleasant side-effects. Indeed, when a single disk fails, a large number of files are affected. To reduce the impact of such undesirable events, the system attempts to mask a single disk failure or the failure of the access path to a disk. The system uses RAID devices with the stripes equal to the block size and dual-attached RAID controllers. To further improve the fault tolerance of the system, GPFS data files as well as metadata are replicated on two different physical disks.

Consistency and performance, critical to any distributed file system, are difficult to balance. Support for concurrent access improves performance but faces serious challenges in maintaining consistency. In GPFS, consistency and synchronization are ensured by a distributed locking mechanism; a *central lock manager* grants *lock tokens* to *local lock managers* running in each I/O node. Lock tokens are also used by the cache management system.

Lock granularity has important implications in the performance of a file system, and GPFS uses a variety of techniques for various types of data. *Byte-range tokens* are used for read and write operations to data files as follows: The first node attempting to write to a file acquires a token covering the entire file, $[0, \infty]$. This node is allowed to carry out all reads and writes to the file without any need for permission until a second node attempts to write to the same file. Then the range of the token given to the first node is restricted. More precisely, if the first node writes sequentially at offset $f p_1$ and the second one at offset $f p_2 > f p_1$, the range of the tokens for the two tokens are $[0, f p_2]$ and $[f p_2, \infty]$, respectively, and the two nodes can operate concurrently, without the need for further negotiations. Byte-range tokens are rounded to block boundaries.

Byte-range token negotiations among nodes use the *required range* and the *desired range* for the offset and for the length of the current and future operations, respectively. *Data shipping*, an alternative to byte-range locking, allows fine-grained data sharing. In this mode the file blocks are controlled by the I/O nodes in a round-

robin manner. A node forwards a read or writes operation to the node controlling the target block, the only one allowed to access the file.

A *token manager* maintains the state of all tokens; it creates and distributes tokens, collects tokens once a file is closed, and downgrades or upgrades tokens when additional nodes request access to a file. Token management protocols attempt to reduce the load placed on the token manager; for example, when a node wants to revoke a token, it sends messages to all the other nodes holding the token and forwards the reply to the token manager.

Access to metadata is synchronized. For example, when multiple nodes write to the same file, the file size and the modification dates are updated using a *shared write lock* to access an inode. One of the nodes assumes the role of a *metanode*, and all updates are channeled through it. The file size and the last update time are determined by the metanode after merging the individual requests. The same strategy is used for updates of the indirect blocks. GPFS global data such as access control lists (ACLs), quotas, and configuration data are updated using the distributed locking mechanism.

GPFS uses *disk maps* to manage the disk space. The GPFS block size can be as large as 1 MB, and a typical block size is 256 KB. A block is divided into 32 sub-blocks to reduce disk fragmentation for small files; thus, the block map has 32 bits to indicate whether a sub-block is free or used. The system disk map is partitioned into n regions, and each disk map region is stored on a different I/O node. This strategy reduces conflicts and allows multiple nodes to allocate disk space at the same time. An *allocation manager* running on one of the I/O nodes is responsible for actions involving multiple disk map regions. For example, it updates free space statistics and helps with deallocation by sending periodic hints of the regions used by individual nodes.

8.5 Google File System

The Google File System (GFS) was developed in the late 1990s. It uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs. It is not surprising that a main concern of the GFS designers was to ensure the reliability of a system exposed to hardware failures, system software errors, application errors, and last but not least, human errors.

The system was designed after a careful analysis of the file characteristics and of the access models. Some of the most important aspects of this analysis reflected in the GFS design are:

- Scalability and reliability are critical features of the system; they must be considered from the beginning rather than at some stage of the design.

- The vast majority of files range in size from a few GB to hundreds of TB.
- The most common operation is to append to an existing file; random write operations to a file are extremely infrequent.
- Sequential read operations are the norm.
- The users process the data in bulk and are less concerned with the response time.
- The consistency model should be relaxed to simplify the system implementation, but without placing an additional burden on the application developers.

Several design decisions were made as a result of this analysis:

1. Segment a file in large chunks.
2. Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file.
3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow; schedule the high-bandwidth data flow by pipelining the data transfer over TCP connections to reduce the response time. Exploit network topology by sending data to the closest node in the network.
4. Eliminate caching at the client site. Caching increases the overhead for maintaining consistency among cached copies at multiple client sites and it is not likely to improve performance.
5. Ensure consistency by channeling critical file operations through a *master*, a component of the cluster that controls the entire system.
6. Minimize the involvement of the *master* in file access operations to avoid hot-spot contention and to ensure scalability.
7. Support efficient check pointing and fast recovery mechanisms.
8. Support an efficient garbage-collection mechanism.

GFS files are collections of fixed-size segments called *chunks*; at the time of file creation each chunk is assigned a unique *chunk handle*. A chunk consists of 64 KB blocks and each block has a 32-bit checksum. Chunks are stored on *Linux* files systems and are replicated on multiple sites; a user may change the number of the replicas from the standard value of three to any desired value. The chunk size is 64 MB; this choice is motivated by the desire to optimize performance for large files and to reduce the amount of metadata maintained by the system.

A large chunk size increases the likelihood that multiple operations will be directed to the same chunk; thus it reduces the number of requests to locate the chunk and, at the same time, it allows the application to maintain a persistent network connection with the server where the chunk is located. Space fragmentation occurs infrequently because the chunk for a small file and the last chunk of a large file are only partially filled.

The architecture of a GFS cluster is illustrated in Figure 8.7. A *master* controls a large number of *chunk servers*; it maintains metadata such as filenames, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers. Some of the metadata is stored in persistent storage (e.g., the *operation log* records the file namespace as well as the file-to-chunk mapping).

The locations of the chunks are stored only in the control structure of the *master*'s memory and are updated at system startup or when a new chunk server joins the cluster. This strategy allows the *master* to have up-to-date information about the location of the chunks.

System reliability is a major concern, and the operation log maintains a historical record of metadata changes, enabling the *master* to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the *master* replays the operation log. To minimize the recovery time, the *master* periodically checkpoints its state and at recovery time replays only the log records after the last checkpoint.

Each chunk server is a commodity *Linux* system; it receives instructions from the *master* and responds with status information. To access a file, an application sends to the *master* the filename and the chunk index, the offset in the file for the read or write operation; the *master* responds with the chunk handle and the location of the chunk. Then the application communicates directly with the chunk server to carry out the desired file operation.

The consistency model is very effective and scalable. Operations, such as file creation, are atomic and are handled by the *master*. To ensure scalability, the *master* has minimal involvement in file mutations and operations such as write or append that occur frequently. In such cases the *master* grants a lease for a particular chunk to one of the chunk servers, called the *primary*; then, the primary creates a serial order for the updates of that chunk.

When data for a write straddles the chunk boundary, two operations are carried out, one for each chunk. The steps for a write request illustrate a process that buffers data and decouples the control flow from the data flow for efficiency:

1. The client contacts the *master*, which assigns a lease to one of the chunk servers for a particular chunk if no lease for that chunk exists; then the *master* replies with the ID of the primary as well as secondary chunk servers holding replicas of the chunk. The client caches this information.

2. The client sends the data to all chunk servers holding replicas of the chunk; each one of the chunk servers stores the data in an internal LRU buffer and then sends an acknowledgment to the client.
3. The client sends a write request to the primary once it has received the acknowledgments from all chunk servers holding replicas of the chunk. The primary identifies mutations by consecutive sequence numbers.
4. The primary sends the write requests to all secondaries.
5. Each secondary applies the mutations in the order of the sequence numbers and then sends an acknowledgment to the primary.
6. Finally, after receiving the acknowledgments from all secondaries, the primary informs the client.

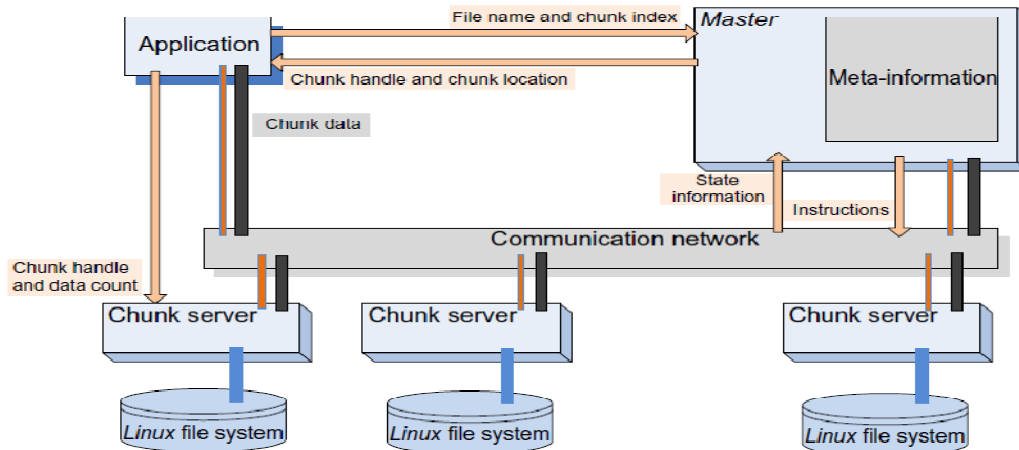


FIGURE 8.7: The architecture of a GFS cluster. The *master* maintains state information about all system components; it controls a number of *chunk servers*. A chunk server runs under *Linux*; it uses metadata provided by the *master* to communicate directly with the application. The data flow is decoupled from the control flow. The data and the control paths are shown separately, data paths with thick lines and control paths with thin lines. Arrows show the flow of control among the application, the *master*, and the chunk servers.

The system supports an efficient checkpointing procedure based on *copy-on-write* to construct system snapshots. A lazy garbage collection strategy is used to reclaim the space after a file deletion. In the first step the filename is changed to a hidden name and this operation is time stamped. The *master* periodically scans the namespace and removes the metadata for the files with a hidden name older than a few days; this mechanism gives a window of opportunity to a user who deleted files by mistake to recover the files with little effort.

Periodically, chunk servers exchange with the *master* the list of chunks stored on each one of them; the *master* supplies them with the identity of orphaned chunks whose metadata has been deleted, and such chunks are then deleted. Even when control messages are lost, a chunk server will carry out the housecleaning at the next *heartbeat* exchange with the *master*. Each chunk server maintains in core the checksums for the locally stored chunks to guarantee data integrity.

CloudStore is an open-source C++ implementation of GFS that allows client access not only from C++ but also from Java and Python.

8.6 *Apache Hadoop*

A wide range of data-intensive applications such as marketing analytics, image processing, machine learning, and Web crawling use *Apache Hadoop*, an open-source, Java-based software system.⁶ *Hadoop* supports distributed applications handling extremely large volumes of data. Many members of the community contributed to the development and optimization of *Hadoop* and several related Apache projects such as *Hive* and *HBase*.

Hadoop is used by many organizations from industry, government, and research; the long list of *Hadoop* users includes major IT companies such as Apple, IBM, HP, Microsoft, Yahoo!, and Amazon; media companies such as The New York Times and Fox; social networks, including Twitter, Facebook, and LinkedIn; and government agencies, such as the U.S. Federal Reserve. In 2011, the Facebook *Hadoop* cluster had a capacity of 30 PB.

A *Hadoop* system has two components, a *MapReduce* engine and a database (see Figure 8.8). The database could be the *Hadoop File System (HDFS)*, Amazon *S3*, or *CloudStore*, an implementation of the Google File System discussed in Section 8.5. *HDFS* is a distributed file system written in Java; it is portable, but it cannot be directly mounted on an existing operating system. *HDFS* is not fully POSIX compliant, but it is highly performant.

The *Hadoop* engine on the master of a multinode cluster consists of a *job tracker* and a *task tracker*, whereas the engine on a slave has only a *task tracker*. The *job tracker* receives a *MapReduce* job from a client and dispatches the work to the *task trackers* running on the nodes of a cluster. To increase efficiency, the *job tracker* attempts to dispatch the tasks to available slaves closest to the place where it stored the task data. The *task tracker* supervises the execution of the work allocated to the node. Several scheduling algorithms have been implemented in *Hadoop* engines, including Facebook's fair scheduler and Yahoo!'s capacity scheduler; see Section 6.8 for a discussion of cloud scheduling algorithms.

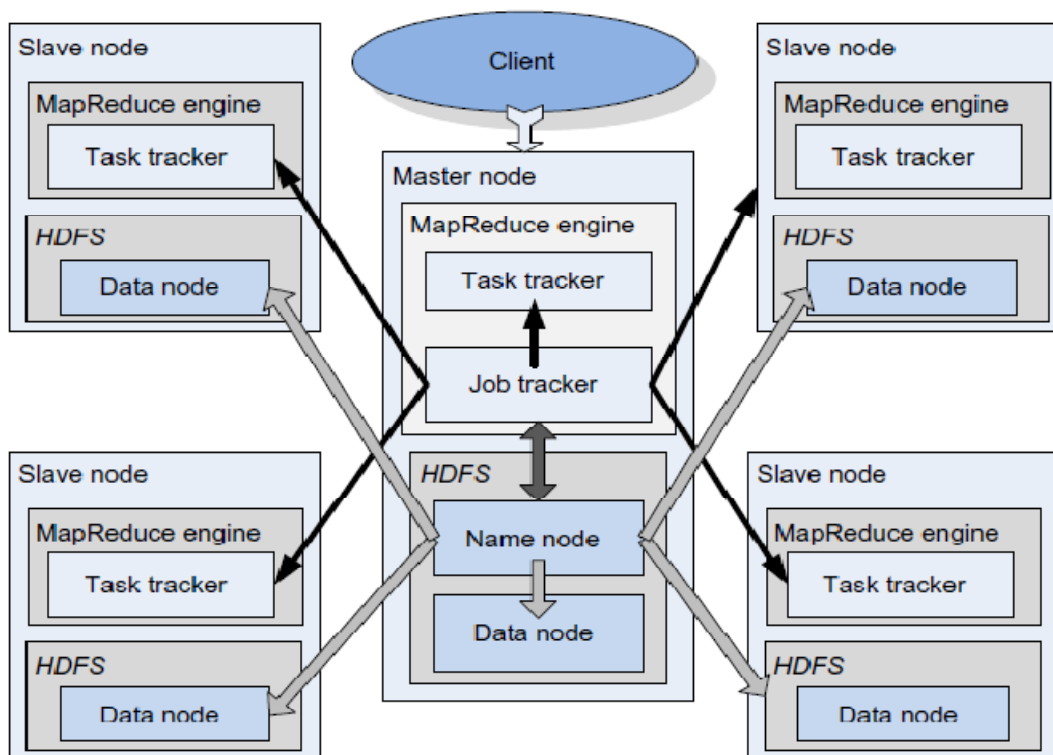


FIGURE 8.8: A *Hadoop* cluster using *HDFS*. The cluster includes a master and four slave nodes. Each node runs a *MapReduce* engine and a database engine, often *HDFS*. The *job tracker* of the master's engine communicates with the *task trackers* on all the nodes and with the *name node* of *HDFS*. The *name node* of *HDFS* shares information about data placement with the *job tracker* to minimize communication between the nodes on which data is located and the ones where it is needed.

HDFS replicates data on multiple nodes. The default is three replicas; a large dataset is distributed over many nodes. The *name node* running on the master manages the data distribution and data replication and communicates with *data nodes* running on all cluster nodes; it shares with the *job tracker* information about data placement to minimize communication between the nodes on which data is located and the ones where it is needed. Although *HDFS* can be used for applications other than those based on the *MapReduce* model, its performance for such applications is not at par with the ones for which it was originally designed.

8.7 Locks and *Chubby*: A locking service

Locks support the implementation of reliable storage for loosely coupled distributed systems; they enable controlled access to shared storage and ensure atomicity of read and write operations. Furthermore, critically important to the design of reliable distributed storage systems are distributed consensus problems, such as the election of a master from a group of data servers. A master has an important role in system management; for example, in GFS the master maintains state information about all system components.

Locking and the election of a master can be done using a version of the Paxos algorithm for asynchronous consensus, discussed in Section 2.11. The algorithm guarantees safety without any timing assumptions, a necessary condition in a large-scale system when communication delays are unpredictable. Nevertheless, the algorithm must use clocks to ensure liveness and to overcome the impossibility of reaching consensus with a single faulty process.

Distributed systems experience communication problems such as lost messages, messages out of sequence, or corrupted messages. There are solutions for handling these undesirable phenomena; for example, one can use virtual time, that is, sequence numbers, to ensure that messages are processed in an order consistent with the time they were sent by all processes involved, but this complicates the algorithms and increases the processing time.

Advisory locks are based on the assumption that all processes play by the rules. Advisory locks do not have any effect on processes that circumvent the locking mechanisms and access the shared objects directly. *Mandatory locks* block access to the locked objects to all processes that do not hold the locks, regardless of whether they use locking primitives.

Locks that can be held for only a very short time are called *fine-grained*, whereas *coarse-grained* locks are held for a longer time. Some operations require meta-information about a lock, such as the name of the lock, whether the lock is shared or held in exclusivity, and the generation number of the lock. This meta-information is sometimes aggregated into an opaque byte string called a *sequencer*.

The question of how to most effectively support a locking and consensus component of a large-scale distributed system demands several design decisions. A first design decision is whether the locks should be mandatory or advisory. *Mandatory locks* have the obvious advantage of enforcing access control; a traffic analogy is that a mandatory lock is like a drawbridge. Once it is up, all traffic is forced to stop.

An *advisory lock* is like a stop sign; those who obey the traffic laws will stop, but some might not. The disadvantages of mandatory locks are added overhead and less flexibility. Once a data item is locked, even a high-priority task related to maintenance or recovery cannot access the data unless it forces the application holding the lock to terminate. This is a very significant problem in large-scale systems where partial system failures are likely.

A second design decision is whether the system should be based on fine-grained or coarse-grained locking. *Fine-grained locks* allow more application threads to access shared data in any time interval, but they generate a larger workload for the lock server. Moreover, when the lock server fails for a period of time, a larger number of applications are affected. Advisory locks and *coarse-grained locks* seem to be a better choice for a system expected to scale to a very large number of nodes distributed in data centers that are interconnected via wide area networks.

A third design decision is how to support a systematic approach to locking. Two alternatives come to mind: (i) delegate to the clients the implementation of the consensus algorithm and provide a library of functions needed for this task, or (ii) create a locking service that implements a version of the asynchronous Paxos algorithm and provide a library to be linked with an application client to support service calls. Forcing application developers to invoke calls to a Paxos library is more cumbersome and more prone to errors than the service alternative. Of course, the lock service itself has to be scalable to support a potentially heavy load.

Another design consideration is flexibility, the ability of the system to support a variety of applications. A name service comes immediately to mind because many cloud applications read and write small files. The names of small files can be included in the namespace of the service to allow atomic file operations. The choice should also consider the performance; a service can be optimized and clients can cache control information. Finally, we should consider the overhead and resources for reaching consensus. Again, the service seems to be more advantageous because it needs fewer replicas for high availability.

In the early 2000s, when Google started to develop a lock service called *Chubby*, it was decided to use advisory and coarse-grained locks. The service is used by several Google systems, including the GFS discussed in Section 8.5 and *BigTable* (see Section 8.9).

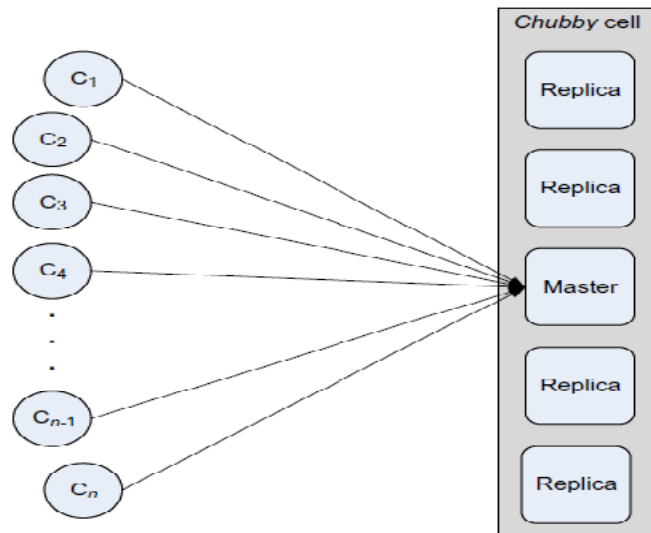


FIGURE 8.9: A *Chubby* cell consisting of five replicas, one of which is elected as a master; n clients use RPCs to communicate with the master.

The basic organization of the system is shown in Figure 8.9. A *Chubby* cell typically serves one data center. The cell server includes several *replicas*, the standard number of which is five. To reduce the probability of correlated failures, the servers hosting replicas are distributed across the campus of a data center.

The replicas use a distributed consensus protocol to elect a new *master* when the current one fails. The master is elected by a majority, as required by the asynchronous Paxos algorithm, accompanied by the commitment that a new master will not be elected for a period called a *master lease*. A *session* is a connection between a client and the cell server maintained over a period of time; the data cached by the client, the locks acquired, and the handles of all files locked by the client are valid for only the duration of the session.

Clients use RPCs to request services from the master. When it receives a write request, the master propagates the request to all replicas and waits for a reply from a majority of replicas before responding. When it receives a read request the master responds without consulting the replicas. The client interface of the system is similar to, yet simpler than, the one supported by the *Unix* File System. In addition, it includes notification of events related to file or system status. A client can subscribe to events such as file content modification, change or addition of a child node, master failure, lock acquired, conflicting lock requests, and invalid file handle.

The files and directories of the *Chubby* service are organized in a tree structure and use a naming scheme similar to that of *Unix*. Each file has a *file handle* similar to the

file descriptor. The master of a cell periodically writes a snapshot of its database to a GFS file server.

Each file or directory can act as a lock. To write to a file the client must be the only one holding the file handle, whereas multiple clients may hold the file handle to read from the file. Handles are created by a call to *open ()* and destroyed by a call to *close ()*. Other calls supported by the service are *GetContentsAndStat ()*, to get the file data and meta-information, *SetContents*, and *Delete ()* and several calls allow the client to acquire and release locks. Some applications may decide to create and manipulate a sequencer with calls to *SetSequencer ()*, which associates a sequencer with a handle, *Get-Sequencer ()* to obtain the sequencer associated with a handle, or check the validity of a sequencer with *CheckSequencer ()*.

The sequence of calls *SetContents()*, *SetSequencer()*, *GetContentsAndStat()*, and *CheckSequencer()* can be used by an application for the election of a master as follows: all candidate threads attempt to open a lock file, call it *lfile*, in exclusive mode; the one that succeeds in acquiring the lock for *lfile* becomes the master, writes its identity in *lfile*, creates a sequencer for the lock of *lfile*, call it *lfseq*, and passes it to the server. The other threads read the *lfile* and discover that they are replicas; periodically they check the sequencer *lfseq* to determine whether the lock is still valid. The example illustrates the use of *Chubby* as a name server. In fact, this is one of the most frequent uses of the system.

We now take a closer look at the actual implementation of the service. As pointed out earlier, *Chubby* locks and *Chubby* files are stored in a database, and this database is replicated. The architecture of these replicas shows that the stack consists of the *Chubby* component, which implements the *Chubby* protocol for communication with the clients, and the active components, which write log entries and files to the local storage of the replica see (Figure 8.10).

Recall that an *atomicity log* for a transaction-processing system allows a crash recovery procedure to undo all-or-nothing actions that did not complete or to finish all-or-nothing actions that committed but did not record all of their effects. Each replica maintains its own copy of the log; a new log entry is appended to the existing log and the Paxos algorithm is executed repeatedly to ensure that all replicas have the same sequence of log entries.

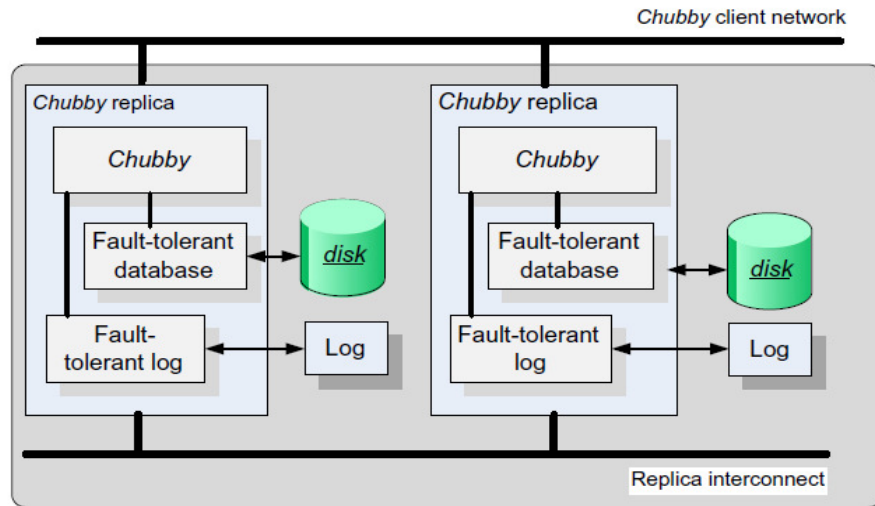


FIGURE 8.10: Chubby replica architecture. The Chubby component implements the communication protocol with the clients. The system includes a component to transfer files to a fault-tolerant database and a fault-tolerant log component to write log entries. The fault-tolerant log uses the Paxos protocol to achieve consensus. Each replica has its own local file system; replicas communicate with one another using a dedicated interconnect and communicate with clients through a client network.

The next element of the stack is responsible for the maintenance of a fault-tolerant database – in other words, making sure that all local copies are consistent. The database consists of the actual data, or the *local snapshot* in Chubby speak, and a *replay log* to allow recovery in case of failure. The state of the system is also recorded in the database.

The Paxos algorithm is used to reach consensus on sets of values (e.g., the sequence of entries in a replicated log). To ensure that the Paxos algorithm succeeds in spite of the occasional failure of a replica, the following three phases of the algorithm are executed repeatedly:

1. Elect a replica to be the master/coordinator. When a master fails, several replicas may decide to assume the role of a master. To ensure that the result of the election is unique, each replica generates a sequence number larger than any sequence number it has seen, in the range $(1, r)$, where r is the number of replicas, and broadcasts it in a *propose* message. The replicas that have not seen a higher sequence number broadcast a *promise* reply and declare that they will reject proposals from other candidate masters. If the number of respondents represents a majority of replicas, the one that sent the *propose* message is elected master.
2. The master broadcasts to all replicas an *accept* message, including the value it has selected, and waits for replies, either *acknowledge* or *reject*.

3. Consensus is reached when the majority of the replicas send an *acknowledge* message; then the master broadcasts the *commit* message.

Implementation of the Paxos algorithm is far from trivial. Although the algorithm can be expressed in as few as ten lines of pseudocode, its actual implementation could be several thousand lines of C++ code. Moreover, practical use of the algorithm cannot ignore the wide variety of failure modes, including algorithm errors and bugs in its implementation, and testing a software system of a few thousands lines of codes is challenging.

8.8 Transaction processing and *NoSQL* databases

Many cloud services are based on *online transaction processing* (OLTP) and operate under tight latency constraints. Moreover, these applications have to deal with extremely high data volumes and are expected to provide reliable services for very large communities of users. It did not take very long for companies heavily involved in cloud computing, such as Google and Amazon, e-commerce companies such as eBay, and social media networks such as Facebook, Twitter, or LinkedIn, to discover that traditional relational databases are not able to handle the massive amount of data and the real-time demands of online applications that are critical for their business models.

The search for alternate models with which to store the data on a cloud is motivated by the need to decrease the latency by caching frequently used data in memory on dedicated servers, rather than fetching it repeatedly. In addition, distributing the data on a large number of servers allows multiple transactions to occur at the same time and decreases the response time. The relational schema are of little use for such applications in which conversion to key-value databases seems a much better approach. Of course, such systems do not store meaningful metadata information, unless they use extensions that cannot be exported easily.

A major concern for the designers of OLTP systems is to reduce the response time. The term *memcaching* refers to a general-purpose distributed memory system that caches objects in main memory (RAM); the system is based on a very large hash table distributed across many servers. The *memcached* system is based on a client-server architecture and runs under several operating systems, including *Linux*, *Unix*, *Mac OS X*, and *Windows*. The servers maintain a key-value associative array. The API allows the clients to add entries to the array and to query it. A key can be up to 250 bytes long, and a value can be no larger than 1 MB. The *memcached* system uses an LRU cache-replacement strategy.

Scalability is the other major concern for cloud OLTP applications and implicitly for datastores. There is a distinction between *vertical scaling*, where the data and the workload are distributed to systems that share resources such as cores and processors, disks, and possibly RAM, and *horizontal scaling*, where the systems do not share either primary or secondary storage.

Cloud stores such as *document stores* and *NoSQL* databases are designed to scale well, do not exhibit a single point of failure, have built-in support for consensus-based decisions, and support partitioning and replication as basic primitives. Systems such as Amazon's *SimpleDB*, discussed in Section 3.1;

CouchDB or *Oracle NoSQL database* are very popular, though they provide less functionality than traditional databases. The *key-value* data model is very popular. Several such systems, including Voldemort, Redis, Scalaris, and Tokyo cabinet, are discussed in.

The “soft-state” approach in the design of *NoSQL* allows data to be inconsistent and transfers the task of implementing only the subset of the ACID properties required by a specific application to the application developer. The *NoSQL* systems ensure that data will be “eventually consistent” at some future point in time instead of enforcing consistency at the time when a transaction is “committed.” Data partitioning among multiple storage servers and data replication are also tenets of the *NoSQL* philosophy⁷; they increase availability, reduce response time, and enhance scalability.

The name *NoSQL* given to this storage model is misleading. Michael Stonebreaker notes that “blinding performance depends on removing overhead. Such overhead has nothing to do with SQL, but instead revolves around traditional implementations of ACID transactions, multi-threading, and disk management.”

The overhead of OLTP systems is due to four sources with equal contribution: logging, locking, latching, and buffer management. Logging is expensive because traditional databases require transaction durability; thus, every write to the database can be completed only after the log has been updated. To guarantee atomicity, transactions lock every record, and this requires access to a lock table. Many operations require multithreading, and the access to shared data structures, such as lock tables, demands short-term latches⁸ for coordination. The breakdown of the instruction count for these operations in existing DBMSs is as follows: 34.6% for buffer management, 14.2% for latching, 16.3% for locking, 11.9% for logging, and 16.2% for hand-coded optimization.

Today OLTP databases could exploit the vast amounts of resources of modern computing and communication systems to store the data in main memory rather than rely on disk-resident B-trees and heap files, locking-based concurrency control, and support for multithreading optimized for the computer technology of past decades. Logless, single-threaded, and transaction-less databases could replace the traditional ones for some cloud applications.

Data replication is critical not only for system reliability and availability, but also for its performance. In an attempt to avoid catastrophic failures due to power blackouts, natural disasters, or other causes (see also Section 1.6), many companies have established multiple data centers located in different geographic regions. Thus, data replication must be done over a *wide area network* (WAN). This could be quite challenging, especially for log data, metadata, and system configuration information, due to increased probability of communication failure and larger communication delays. Several strategies are possible, some based on master/slave configurations, others based on homogeneous replica groups.

Master/slave replication can be asynchronous or synchronous. In the first case the master replicates write-ahead log entries to at least one slave, and each slave acknowledges appending the log record as soon as the operation is done. In the second case the master must wait for the acknowledgments from all slaves before proceeding. Homogeneous replica groups enjoy shorter latency and higher availability than master/slave configurations. Any member of the group can initiate mutations that propagate asynchronously.

8.9 *BigTable*

BigTable is a distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of storage servers. The system uses the Google File System discussed in Section 8.5 to store user data as well as system information. To guarantee atomic read and write operations, it uses the *Chubby* distributed lock service (see Section 8.7); the directories and the files in the namespace of *Chubby* are used as locks.

The system is based on a simple and flexible data model. It allows an application developer to exercise control over the data format and layout and reveals data locality information to the application clients. Any read or write row operation is atomic, even when it affects more than one column. The column keys identify *column families*, which are units of access control. The data in a column family is of the same type. Client applications written in C++ can add or delete values, search for a subset of data, and look up data in a row.

A row key is an arbitrary string of up to 64 KB, and a row range is partitioned into *tablets* serving as units for load balancing. The time stamps used to index various versions of the data in a cell are 64-bit integers; their interpretation can be defined by the application, whereas the default is the time of an event in microseconds. A column key consists of a string defining the family name, a set of printable characters, and an arbitrary string as qualifier.

The organization of a *BigTable* (see Figure 8.11) shows a sparse, distributed, multidimensional map for an email application. The system consists of three major components: a library linked to application clients to access the system, a master server, and a large number of tablet servers. The master server controls the entire system, assigns tablets to tablet servers and balances the load among them, manages garbage collection, and handles table and column family creation and deletion.

Internally, the space management is ensured by a three-level hierarchy: the *root tablet*, the location of which is stored in a *Chubby* file, points to entries in the second element, the *metadata* tablet, which, in turn, points to *user* tablets, collections of locations of users' tablets. An application client searches through this hierarchy to identify the location of its tablets and then caches the addresses for further use.

The performance of the system reported in is summarized in Table 8.2. The table shows the number of random and sequential read and write and scan operations for 1,000 bytes, when the number of servers increases from 1 to 50, then to 250, and finally to 500. Locking prevents the system from achieving a linear speed-up, but the performance of the system is still remarkable due to a fair number of optimizations. For example, the number of scans on 500 tablet servers is $7,843/2 \times 10^3$ instead of $15,385/2 \times 10^3$. It is reported that only 12 clusters use more than 500 tablet servers, whereas some 259 clusters use between 1 and 19 tablet servers.

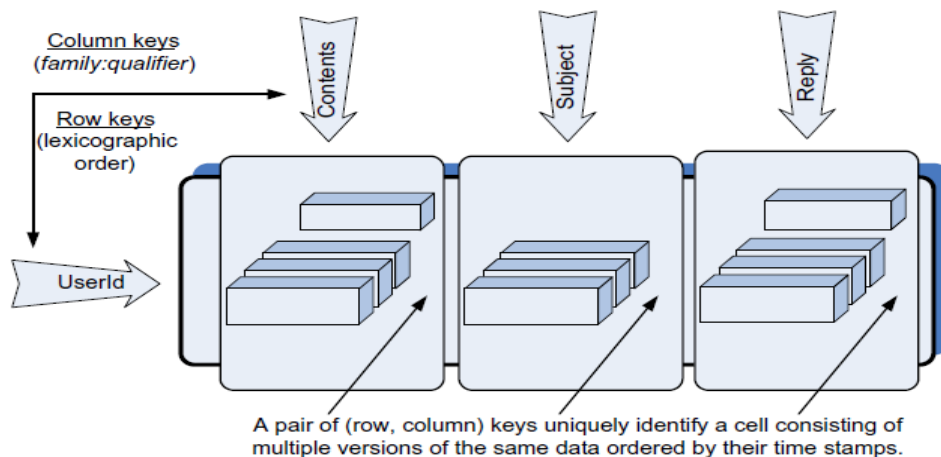


FIGURE 8.11: A *BigTable* example. The organization of an email application as a sparse, distributed, multidimensional map. The slice of the *BigTable* shown consists of a row with the key “UserId” and three family columns. The “Contents” key identifies the cell holding the contents of emails received, the cell with key “Subject” identifies the subject of emails, and the cell with the key “Reply” identifies the cell holding the replies. The versions of records in each cell are ordered according to their time stamps. The row keys of this *BigTable* are ordered lexicographically. A column key is obtained by concatenating the *family* and the *qualifier* fields. Each value is an uninterpreted array of bytes.

Table 8.2 *BigTable* performance: the number of operations per tablet server.

Number of Tablet Servers	Random Read	Sequential Read	Random Write	Sequential Write	Scan
1	1,212	4,425	8,850	8,547	15,385
50	593	2,463	3,745	3,623	10,526
250	479	2,625	3,425	2,451	9,524
500	241	2,469	2,000	1,905	7,843

BigTable is used by a variety of applications, including *Google Earth*, *Google Analytics*, *Google Finance*, and Web crawlers. For example, *Google Earth* uses two tables, one for preprocessing and one for serving client data. The preprocessing table stores raw images; the table is stored on disk because it contains some 70 TB of data.

Each row of data consists of a single image; adjacent geographic segments are stored in rows in close proximity to one another. The column family is very sparse; it contains a column for every raw image. The preprocessing stage relies heavily on *MapReduce* to clean and consolidate the data for the serving phase. The serving table stored on GFS is “only” 500 GB, and it is distributed across several hundred tablet servers, which maintain in-memory column families. This organization enables the serving phase of *Google Earth* to provide a fast response time to tens of thousands of queries per second.

Google Analytics provides aggregate statistics such as the number of visitors to a Web page per day. To use this service, Web servers embed a *JavaScript* code into their Web pages to record information every time a page is visited. The data is collected in a *raw-click BigTable* of some 200 TB, with a row for each end-user session. A *summary* table of some 20 TB contains predefined summaries for a Website.

8.10 Megastore

Megastore is scalable storage for online services. The system, distributed over several data centers, has a very large capacity, 1 PB in 2011, and it is highly available. *Megastore* is widely used internally at Google; it handles some 23 billion transactions daily: 3 billion write and 20 billion read transactions.

The basic design philosophy of the system is to partition the data into *entity groups* and replicate each partition independently in data centers located in different geographic areas. The system supports full ACID semantics within each partition and provides limited consistency guarantees across partitions (see Figure 8.12). *Megastore* supports only those traditional database features that allow the system to scale well and that do not drastically affect the response time.

Another distinctive feature of the system is the use of the Paxos consensus algorithm, discussed in Section 2.11, to replicate primary user data, metadata, and system configuration information across data centers and for locking. The version of the Paxos algorithm used by *Megastore* does not require a single master. Instead, any node can initiate read and write operations to a write-ahead log replicated to a group of symmetric peers.

The entity groups are application-specific and store together logically related data. For example, an email account could be an entity group for an email application. Data should be carefully partitioned to avoid excessive communication between entity groups. Sometimes it is desirable to form multiple entity groups, as in the case of blogs.

The middle ground between traditional and *NoSQL* databases taken by the *Megastore* designers is also reflected in the data model. The data model is declared in a *schema* consisting of a set of *tables* composed of *entries*, each entry being a collection of named and typed *properties*. The unique primary key of an entity in a table is created as a composition of entry properties. A *Megastore* table can be a *root* or a *child* table. Each *child entity* must reference a special entity, called a *root entity* in its root table. An entity group consists of the primary entity and all entities that reference it.

The system makes extensive use of *BigTable*. Entities from different *Megastore* tables can be mapped to the same *BigTable* row without collisions. This is possible because the *BigTable* column name is a concatenation of the *Megastore* table name and the name of a property. A *BigTable* row for the root entity stores the transaction and all metadata for the entity group. As we saw in Section 8.9, multiple versions of the data

with different time stamps can be stored in a cell. *Megastore* takes advantage of this feature to implement *multi-version concurrency control* (MVCC); when a mutation of a transaction occurs, this mutation is recorded along with its time stamp, rather than marking the old data as obsolete and adding the new version. This strategy has several advantages: read and write operations can proceed concurrently, and a read always returns the last fully updated version.

A write transaction involves the following steps: (1) Get the timestamp and the log position of the last committed transaction. (2) Gather the write operations in a log entry. (3) Use the consensus algorithm to append the log entry and then commit. (4) Update the *BigTable* entries. (5) Clean up.

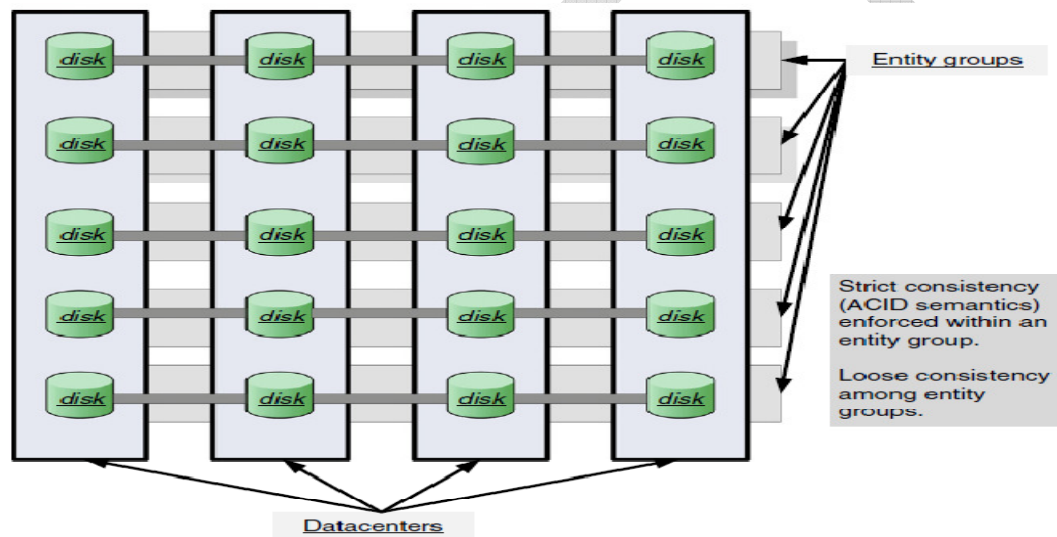


FIGURE 8.12: *Megastore* organization. The data is partitioned into *entity groups*; full ACID semantics within each partition and limited consistency guarantees across partitions are supported. A partition is replicated across data centers in different geographic areas.

8.11 Amazon Simple Storage Service(S3)

Amazon S3 *SimpleStorageService* is a scalable, high-speed, low-cost web-based service designed for online backup and archiving of data and application programs. It allows uploading, store, and downloading any type of files up to 5 GB in size. This service allows the subscribers to access the same systems that Amazon uses to run its own web sites. The subscriber has control over the accessibility of data, i.e. privately/publicly accessible.

How to Configure S3?

Following are the steps to configure a S3 account.

Step 1 – Open the Amazon S3 console using this link – <https://console.aws.amazon.com/s3/home>

Step 2 – Create a Bucket using the following steps.

- A prompt window will open. Click the Create Bucket button at the bottom of the page.



- Create a Bucket dialog box will open. Fill the required details and click the Create button.

Create a Bucket - Select a Bucket Name and Region

Cancel

A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the [Amazon S3 documentation](#).

Bucket Name:

Region:

Set Up Logging >

Create

Cancel

- The bucket is created successfully in Amazon S3. The console displays the list of buckets and its properties.

Create Bucket

Actions

None

Properties

Transfers

All Buckets

Name
<input type="text" value="example.com"/>

Bucket: example.com

Bucket: example.com

Region: US Standard

Creation Date: Tue Mar 04 16:58:27 GMT-500 2014

Owner: Me

Permissions

Static Website Hosting

Logging

Notifications

Lifecycle

Tags

Requester Pays

Versioning

- Select the Static Website Hosting option. Click the radio button Enable website hosting and fill the required details.

☐ Do not enable website hosting

☒ Enable website hosting

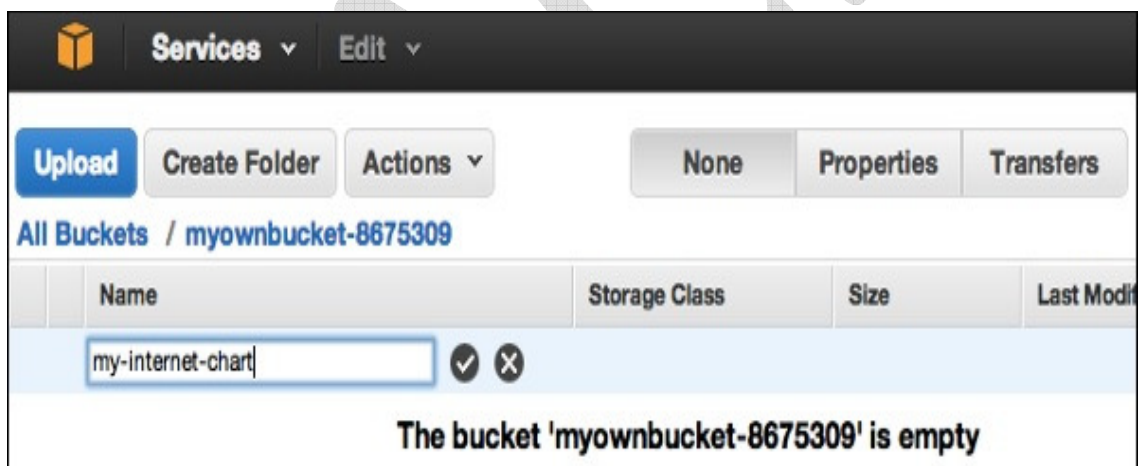
Index Document:

Error Document:

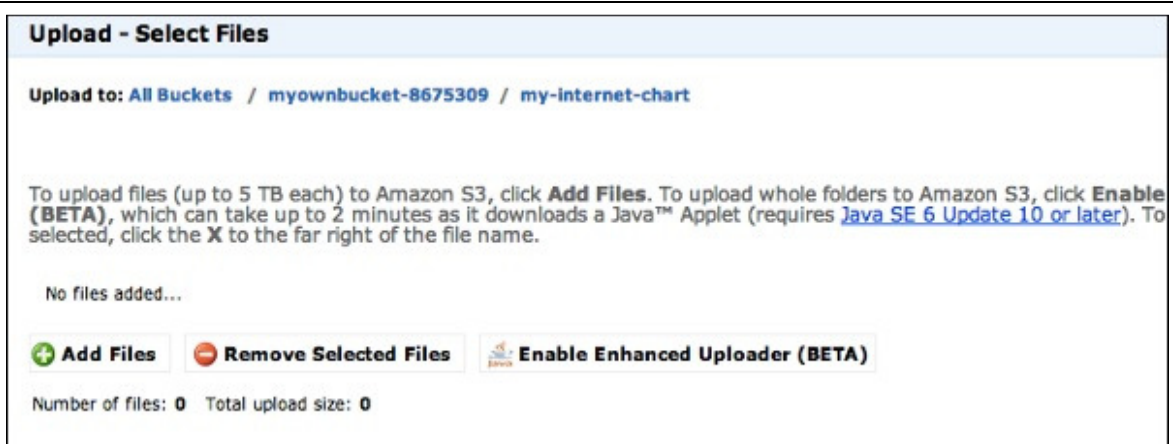
► **Edit Redirection Rules:** You can set custom rules to automatically redirect web page requests for specific content.

Step 3 – Add an Object to a bucket using the following steps.

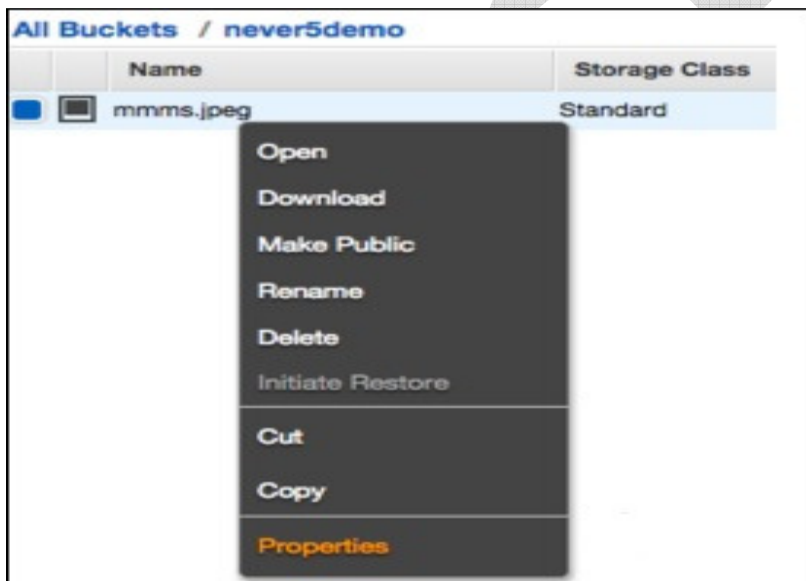
- Open the Amazon S3 console using the following link – <https://console.aws.amazon.com/s3/home>
- Click the Upload button.
-



- Click the Add files option. Select those files which are to be uploaded from the system and then click the Open button.



- Click the start upload button. The files will get uploaded into the bucket.
- To open/download an object** – In the Amazon S3 console, in the Objects & Folders list, right-click on the object to be opened/downloaded. Then, select the required object.

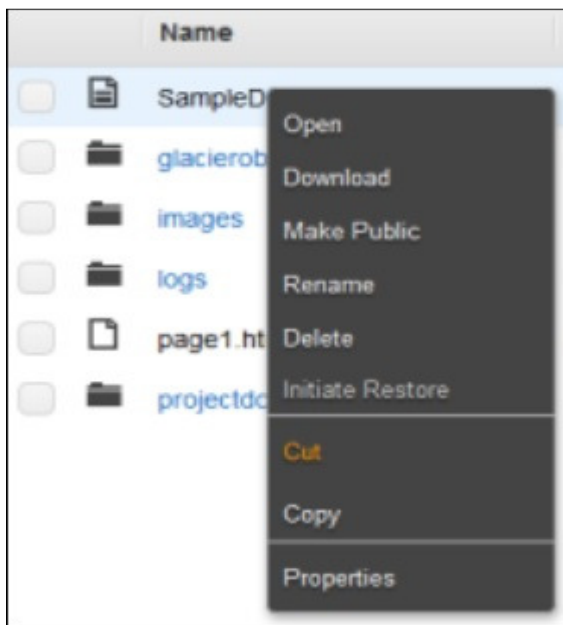


How to Move S3 Objects?

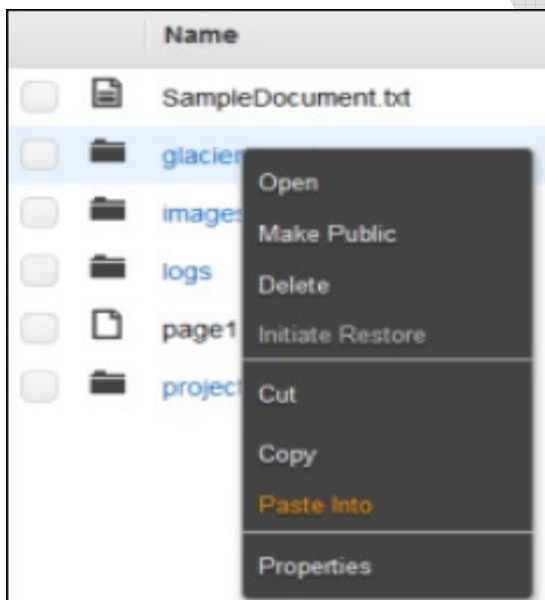
Following are the steps to move S3 objects.

step 1 – Open Amazon S3 console.

step 2 – Select the files & folders option in the panel. Right-click on the object that is to be moved and click the Cut option.



step 3 – Open the location where we want this object. Right-click on the folder/bucket where the object is to be moved and click the Paste into option.

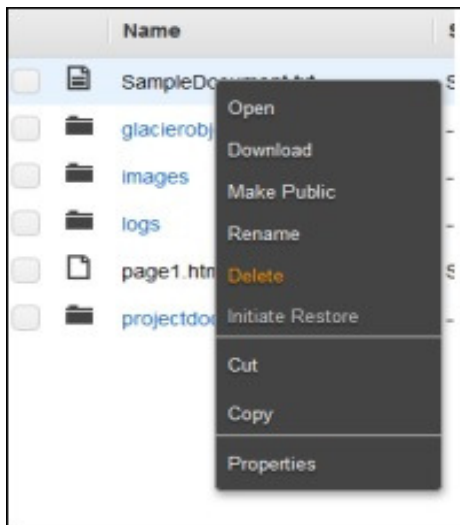


How to Delete an Object?

Step 1 – Open Amazon S3.

Step 2 – Select the files & folders option in the panel. Right-click on the object that is to be deleted. Select the delete option.

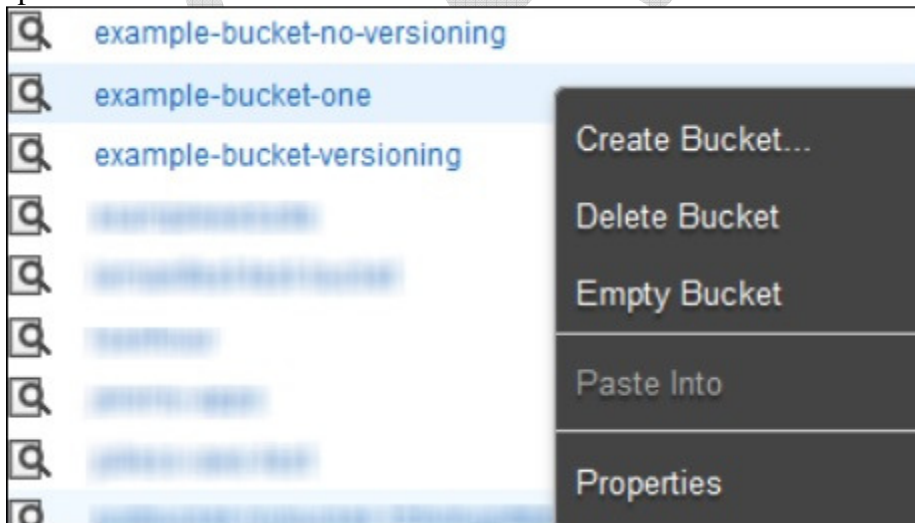
Step 3 – A pop-up window will open for confirmation. Click Ok.



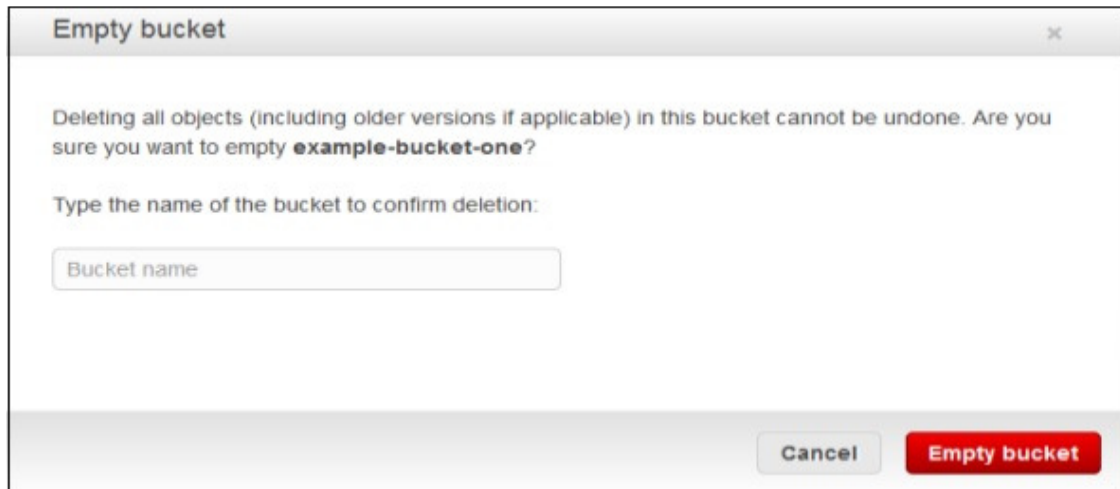
How to Empty a Bucket?

Step 1 – Open Amazon S3 console.

Step 2 – Right-click on the bucket that is to be emptied and click the empty bucket option.



Step 3 – A confirmation message will appear on the pop-up window. Read it carefully and click the **Empty bucket** button to confirm.



The screenshot shows a modal dialog box titled "Empty bucket" with a close button (X) in the top right corner. The main text inside the dialog reads: "Deleting all objects (including older versions if applicable) in this bucket cannot be undone. Are you sure you want to empty **example-bucket-one**?" Below this text, there is a label "Type the name of the bucket to confirm deletion:" followed by a text input field containing the placeholder text "Bucket name". At the bottom right of the dialog, there are two buttons: a grey "Cancel" button and a red "Empty bucket" button.

Amazon S3 Features

- **Low cost and Easy to Use** – Using Amazon S3, the user can store a large amount of data at very low charges.
- **Secure** – Amazon S3 supports data transfer over SSL and the data gets encrypted automatically once it is uploaded. The user has complete control over their data by configuring bucket policies using AWS IAM.
- **Scalable** – Using Amazon S3, there need not be any worry about storage concerns. We can store as much data as we have and access it anytime.
- **Higher performance** – Amazon S3 is integrated with Amazon CloudFront, that distributes content to the end users with low latency and provides high data transfer speeds without any minimum usage commitments.
- **Integrated with AWS services** – Amazon S3 integrated with AWS services include Amazon CloudFront, Amazon CloudWatch, Amazon Kinesis, Amazon RDS, Amazon Route 53, Amazon VPC, AWS Lambda, Amazon EBS, Amazon Dynamo DB, etc.