CLOUD COMPUTING

UNIT-5

# UNIT-5

# CLOUD COMPUTING

# 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.
    - 1993:15.8EB;equivalenttofourCD-ROMsperuser.
    - 2000:54.5EB;equivalentto12CD-ROMsperuser.
    - 2007:295EB;equivalenttoalmost61CD-ROMsperuser.

Though it pales in comparison with processor technology, the evolution of storage technology is astounding. A 2003 studyshows that during the 1980–2003 period the storagedensityofharddiskdrives(HDD)increasedbyfourordersofmagnitude,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.

ThedensityofDynamic Random AccessMemory(DRAM )increased from about 1 Gb/in2in1990to100Gb/in2in2003.ThecostofDRAMtumbledfromabout$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 imagingdevicesgeneratedataatahighrateanddumpitontostoragesystems

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 theCPU cycles along with thevast storagecapacitynecessary 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 withoff-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 theNetworkFileSystem (NFS)andtheAndrewFileSystem (AFS),totoday’sGoogle File System (GFS) and the *Megastore*.

# 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.

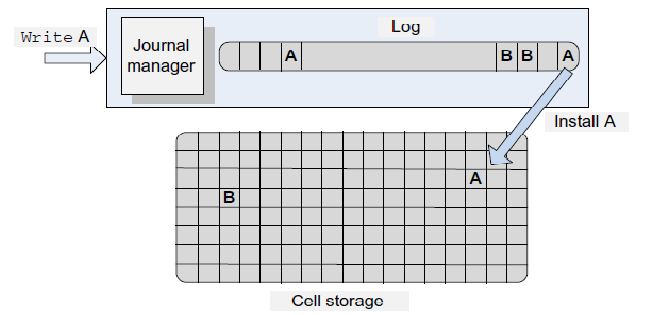
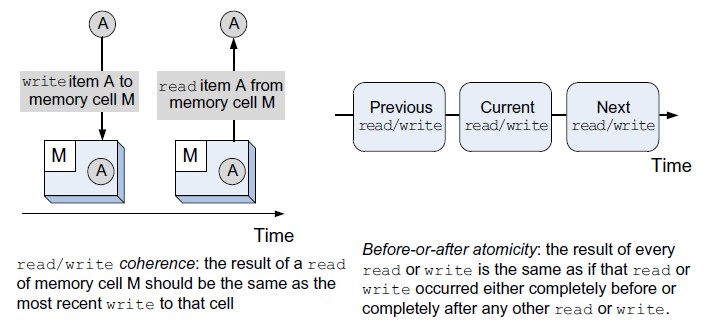
Two abstract models of storage are commonly used: *cell storage* and *journal storage*. Cell storage assumes that the storage consists of cells of the same size andthat each object fits exactly in one cell. This model reflects the physical organizationof several storage media; the primary memory of a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

*Journalstorage*isafairlyelaborateorganizationforstoringcompositeobjectssuch 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 thecurrent value. The user does not have direct access to the *cell storage*; instead the user canrequestthe*journalmanager*to(i)startanewaction;(ii)readthevalueofacell;

(iii)writethevalueofa cell; (iv)commit an action; or(v)abort anaction.The*journal*

*manager*translatesuserrequeststocommandssenttothecellstorage:(i)readacell;

(ii)writeacell;(iii)allocateacell;or(iv)deallocateacell.



**FIGURE8.1:**Illustrationcapturingthesemanticsofread/write*coherence*and*before-or-afteratomicity*.

**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 newvalue 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**,inthelogandcellstorageareshown. Anewvalueof**A**is writtenfirst tothe*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 logprovides 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(seeFigure8.2).The*log*isalwayskeptonnonvolatilestorage(e.g.,disk)and the

considerably larger *cell storage* resides typically on nonvolatile memory, but can beheld 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 fundsfrom oneaccount andcreditingit toanother. Thesystem mayfail duringorafter eachoneoftheactions,andstepstoensurecorrectnessmustbetaken. Correctnessofa 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 timeconsistency*.Toguaranteecorrectness,atransaction-processingsystemsupports*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 numberofnodes, andprovidea global namingspace. Inaparallel datasystem,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 recoveryafter a failure.

A DBMS supports a *query language,* a dedicated programming language used to develop database applications. Several database models, including the navigational model ofthe1960s,therelational model ofthe1970s,theobject-oriented model ofthe 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, cloudapplications require low latency, scalability, and high availability and demand a consistent view of the data.

These requirements cannot be satisfied simultaneouslyby 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 aquerylanguage 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 fewyears. 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.

# Distributed file systems:

In this section we discuss the first distributed file systems, developed in the 1980sby software companies and universities. The systems covered are the Network File System developed by Sun Microsystems in 1984, the Andrew File System developedat 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 fortheir operations.Diskless workstationswere appealingduetoreduced 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 promotersofdistributedsystemsbasedonworkstations,proceededtodeveloptheNFS in the early 1980s.

**Network File System(NFS).** NFS was the first widelyused distributed file system; thedevelopment ofthis applicationbased onthe client-server model wasmotivatedby 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, manydesign 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:

* ProvidethesamesemanticsasalocalUFStoensurecompatibilitywithexisting applications.
  + FacilitateeasyintegrationintoexistingUFS.
  + Ensure that the system would be widely used and thus support clients running on different operating systems.
  + Acceptamodestperformancedegradationduetoremoteaccessoveranetwork with a bandwidth of several Mbps.

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:

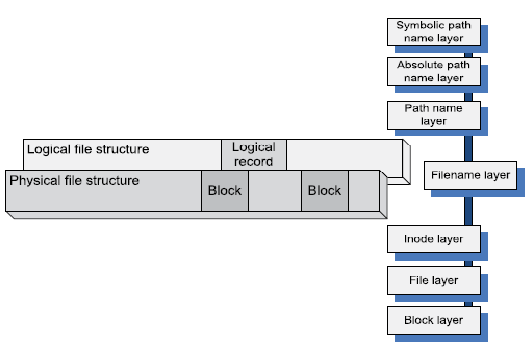
* + The layered design provides the necessary *flexibility* for the file system; layering allows separation of concerns and minimization of the interaction among the modules necessary to implement the system. The addition of the *vnode* layer allowed the *Unix* File System to treat local and remote file 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 thedata from theperspectiveoftheapplication.The*physicalorganization*reflectsthestoragemodel 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.

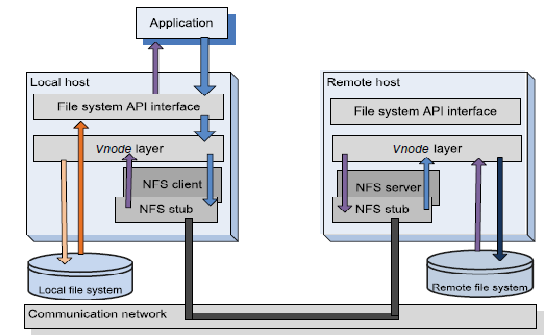


Several *control structures* maintained by the kernel of the operating system support file handling by a running process. These structures are maintained in the user area of the process address space and can only be accessed in kernel mode. To accessa file, a process must first establish a connection 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 runsonthelocalhostwhiletheserverisatthesiteoftheremotefilesystem,andthey

interact by means of remote procedure calls (RPCs) (see Figure 8.4). The APIinterface 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 NSF 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 theremote 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 toa 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 theuserifthecallwassuccessfulthefirsttime.Iftheserverfailstoexecutethefirstcall,the

secondcallreturnsnormally.NotealsothatthereisnocloseRPCbecausethisactiononly makes changes in the process open file structure and does not affect the remote file.

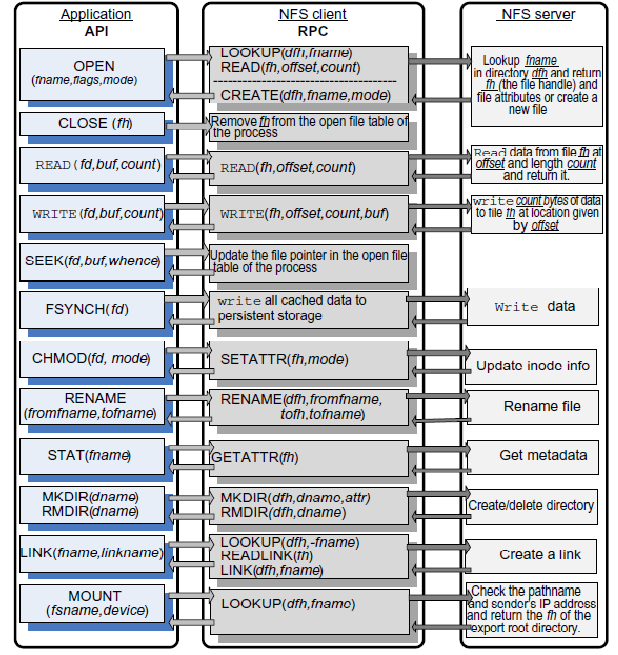
TheNFS hasundergone significanttransformations over the years. It has evolvedfromVersion 2 , discussed in this section, to Version 3in 1994 and then to Version 4 in 2000.

**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 Andrewworkstation, 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 modifiedcopiesof filesbackontheserverstheycame from.Readingandwritingfrom/toafileare 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, thelocal 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 everytime 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 andcan 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 directoryname, *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 andobviouslyimprovesperformancebyreducingresponsetime.Afileaccessrequest 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 issent 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 ofphysicalmemory. The main-memorycaches allowed disklessworkstations 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 reportshow that SFS was 30–35% faster than either NFS or AFS.

Thefilecacheisorganizedasacollectionof4KBblocks; acacheblock hasa virtual address consisting ofauniquefileidentifiersupplied bytheserverand ablock 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 memorymanage separate sets of physical memorypages 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 knowsthetimeofeachreadand writeoperation.Wheneverthefilesystem orthevirtual memory management experiences a file cache miss or a page fault, it compares the age ofitsoldestcacheblock orpage, respectively, withtheageoftheoldestoneoftheother system; the oldest cache block or page is forced to release the real memory frame.

AnimportantdesigndecisionrelatedtotheSFSwasto*delaywrite-backs;*this means that ablock is first written to cache, and thewriting to thedisk is delayed for a timeoftheorderoftensofseconds.Thisstrategyspeedsupwritingandavoids

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 concurrencyis concerned, we distinguish *sequential write sharing*, when a file cannot be opened simultaneouslyfor 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 writesharing, the client caching for the file is disabled; all reads and writes are carried out through the server.

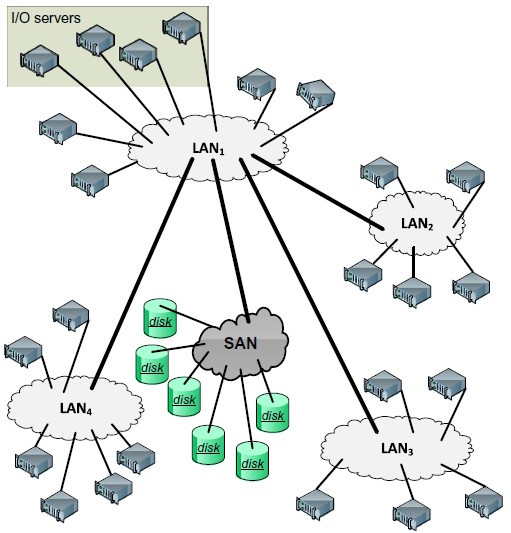
# General Parallel File System:

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 early2000s as a successor to the TigerShark multimedia file system. GPFS is a parallel filesystem that emulates closely the behavior of a general-purpose POSIX systemrunning 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 of1 TB each .

**FIGURE 8.6:** A GPFS configuration. The disks are interconnected by a SAN and compute servers are distributed in four LANs, *LAN*1–*LAN*4. The I/O nodes/servers are connected to *LAN*1.



The maximum file size is (263 −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 storedin inodes and indirect blocks.

Reliability is a major concern in a system with many physical components. To recoverfrom system failures,GPFS recordsall metadataupdatesin a*write-ahead*log file. *Write-ahead* means that updates are written to persistent storage only after thelog records have been written. For example, when a new file is created, a directory blockmustbeupdatedandaninodeforthefilemustbecreated.Theserecordsare

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, thenthe 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, anyI/O nodeis ableto initiaterecoveryon behalfofafailed 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 lockmanagers* runningin 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 attemptingto writeto afileacquires atoken coveringtheentirefile, [0*,* ∞].This node is allowed to carryout all reads and writes to the file without anyneed 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 writessequentially 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*,* ∞], respectively, and the two nodes canoperateconcurrently, withouttheneedforfurthernegotiations.Byte-rangetokens are rounded to block boundaries.

Byte-range tokennegotiationsamongnodes usethe*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 datasharing.InthismodethefileblocksarecontrolledbytheI/Onodesinaround-

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,collectstokensonceafileisclosed,anddowngradesorupgradestokenswhen 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.

Accesstometadataissynchronized.Forexample,whenmultiplenodeswritetothe 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 updatesarechanneledthroughit.Thefilesizeand thelastupdatetimearedetermined 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 indicatewhetherasub-blockisfreeorused.Thesystemdiskmapispartitionedinto*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.

# 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.
    - ThevastmajorityoffilesrangeinsizefromafewGBtohundredsofTB.
    - Themostcommonoperationistoappendtoanexistingfile;randomwriteoperations to a file are extremely infrequent.
    - Sequentialreadoperationsarethenorm.
    - Theusersprocessthedatainbulkandarelessconcernedwiththeresponsetime.
    - 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 ofthis analysis:

1. Segmentafileinlargechunks.
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 formaintaining 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. Supportefficientcheckpointingandfastrecoverymechanisms.
8. Supportanefficientgarbage-collectionmechanism.

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 choiceis motivated bythedesireto optimizeperformanceforlarge 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 thesame chunk; thus it reduces thenumberof requests to locatethe 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 eachfile, 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*’smemoryand are updated at system startuporwhena new chunkserver joins the cluster. This strategy allows the *master* to have up-to-date information about the location of the chunks.

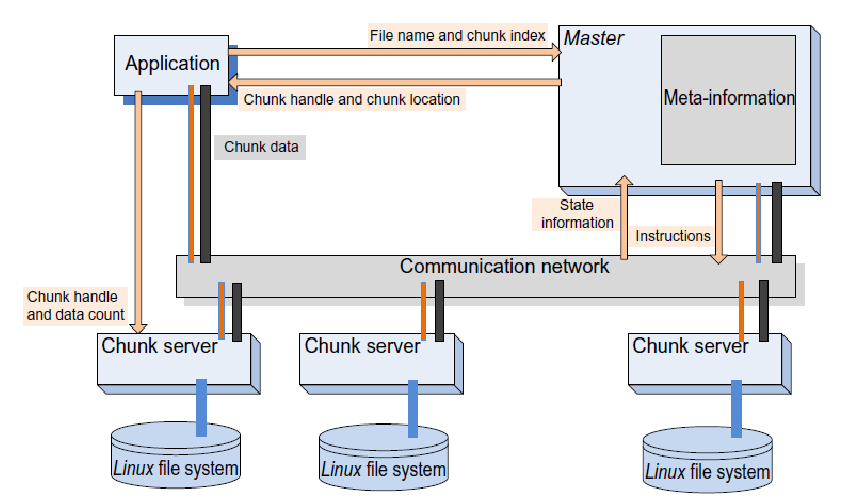
System reliability is a major concern, and the operation log maintains a historical record of metadata changes, enabling the *master* to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the *master* replays the operation log. To minimize the recovery time, the *master* periodicallycheckpoints its state and at recoverytime replays onlythe 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 applicationsendstothe*master*thefilenameandthechunkindex,theoffsetinthefilefortheread or write operation; the *master* responds with the chunk handle and the location of the chunk. Then the application communicates directlywith the chunk server to carryout 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 appendthat 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.

Whendatafor awritestraddlesthechunkboundary,twooperationsare carriedout, 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 replicasof the chunk. The client caches this information.
  2. Theclient sends thedata to all chunk servers holdingreplicas ofthe chunk; each oneofthechunkserversstoresthedatainaninternal LRUbufferandthensends 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. Theprimarysendsthewriterequeststoallsecondaries.
  5. Each secondary applies the mutations in the order of the sequence numbers and then sends an acknowledgment to the primary.



* 1. 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 decoupledfromthecontrolflow.Thedataandthecontrolpathsareshownseparately,datapathswith thick lines and control paths with thin lines. Arrows show the flowof 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 clientaccess not only from C++ but also from Java and Python.