UNIT VI

STORAGE SYSTEMS

Evolution of storage technology, storage models, file systems and database distributed file systems, General parallel file systems. Google file system, Apache Hadoop, BigTable, Megastore, Amazon Simple Storage Service (S3).

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.

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.

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.

A 2003 study shows that during

- 1980–2003 period the storage density of hard disk drives (HDD) increased by four orders of magnitude, from about 0.01 Gb/in^2 to about 100 Gb/in^2.

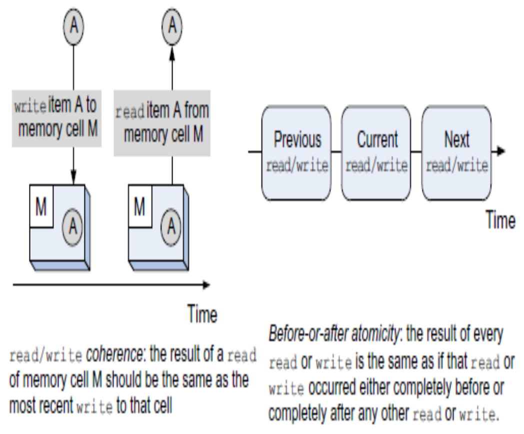
- During the same period the prices fell by five orders of magnitude, to about 1 cent /Mbyte. HDD densities are projected to climb to 1,800 Gb/in^2 by 2016, up from 744 Gb/in^2 in 2011.
- The density of Dynamic Random Access Memory (DRAM) increased from about 1 Gb/in^2 in 1990 to 100 Gb/in^2 in 2003.
- 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.

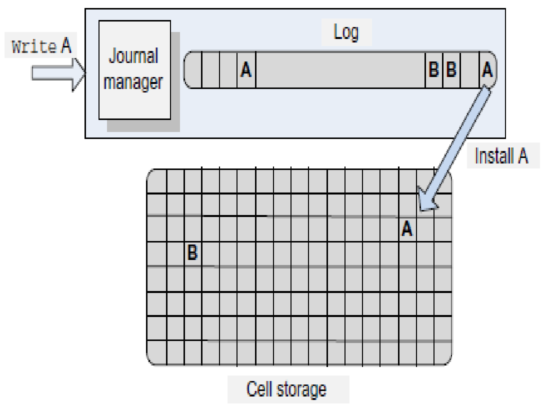
These rapid technological advancements have changed the balance between initial investment in storage devices and system management costs. Now the cost of storage management is the dominant element of the total cost of a storage system. This effect favors the centralized storage strategy supported by 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.

STORAGE MODELS, FILE SYSTEMS AND DATABASE

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 which 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 below fig) *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.


A
A
write item A to memory cell M
read item A from memory cell M
M
A
M
A
Time
Previous read/write
Current read/write
Next read/write
Time
read/write coherence: the result of a read of memory cell M should be the same as the most recent write to that cell
Before-or-after atomicity: the result of every read or write is the same as if that read or write occurred either completely before or completely after any other read or write.



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) de-allocate a cell.

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: network files 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. 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.

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, 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.
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.

DISTRIBUTED FILE SYSTEMS

The first distributed file systems, developed in the 1980s by software companies and universities. The systems covered are

- **Network File System** developed by Sun Microsystems in 1984.
- **Andrew File System** developed at Carnegie Mellon University as part of the Andrew project.
- **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)**.

File System	**Writing policy**	**Consistency guarantees**	**Cache validation**	**Cache Size and Location**
Sprite	30s delay	Concurrent , sequential	On open with server consent	Variable, memory
NFS	On close or 30 s delay	Sequential	On open with server consent	Fixed memory
AFS	On close	Sequential	When modified server asks clients	Fixed disk
locus	On close	Sequential	On open with server consent	Fixed disk
RFS	Write through	sequential	On open with server consent	Fixed memory
Apollo	Delayed or on unlock	Sequential	On open with server consent	Variable memory

Fig: comparison of several network file systems

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.*

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 modest performance degradation due to remote access over a network with a bandwidth of several Mbps.

- 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.
- The hierarchical design supports *scalability* of the file system.
- 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 lower three layers of the **UFS** (Unix File System) 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.

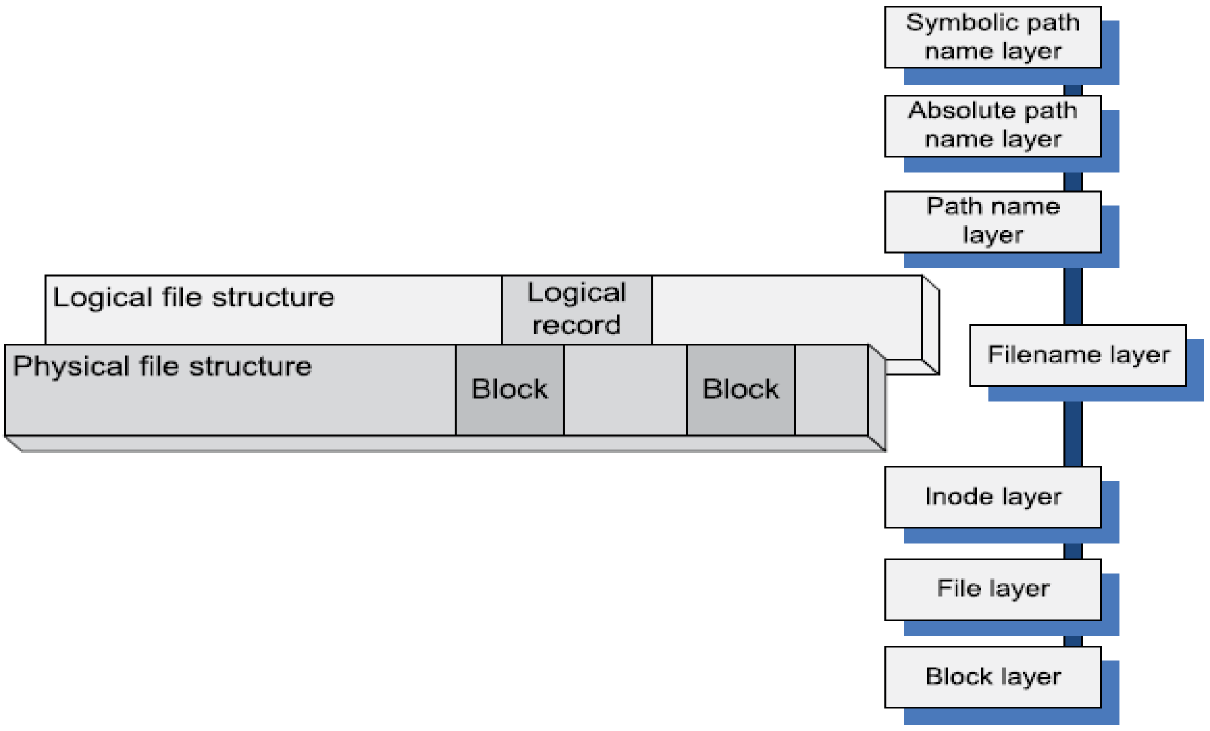


Fig: The layered design of the *Unix* File System

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. 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*,** 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.

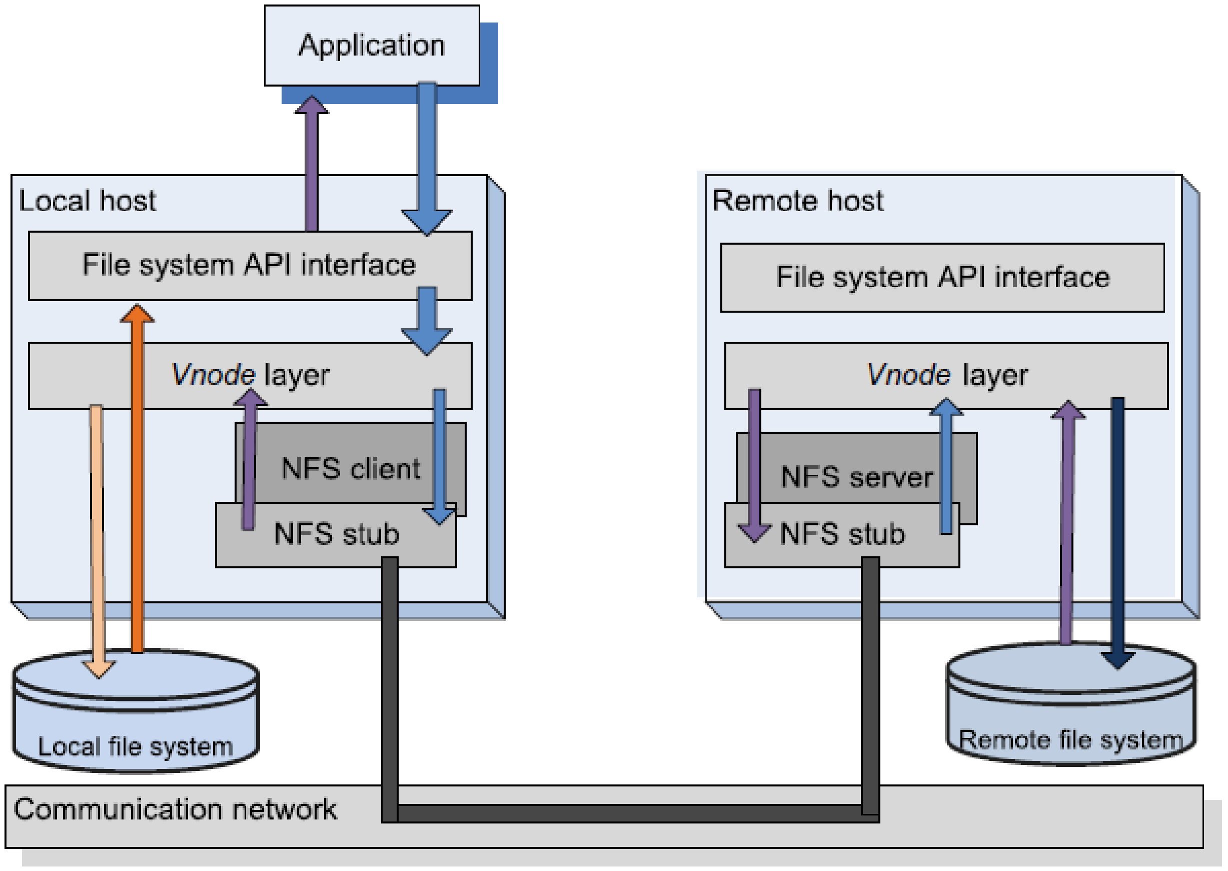
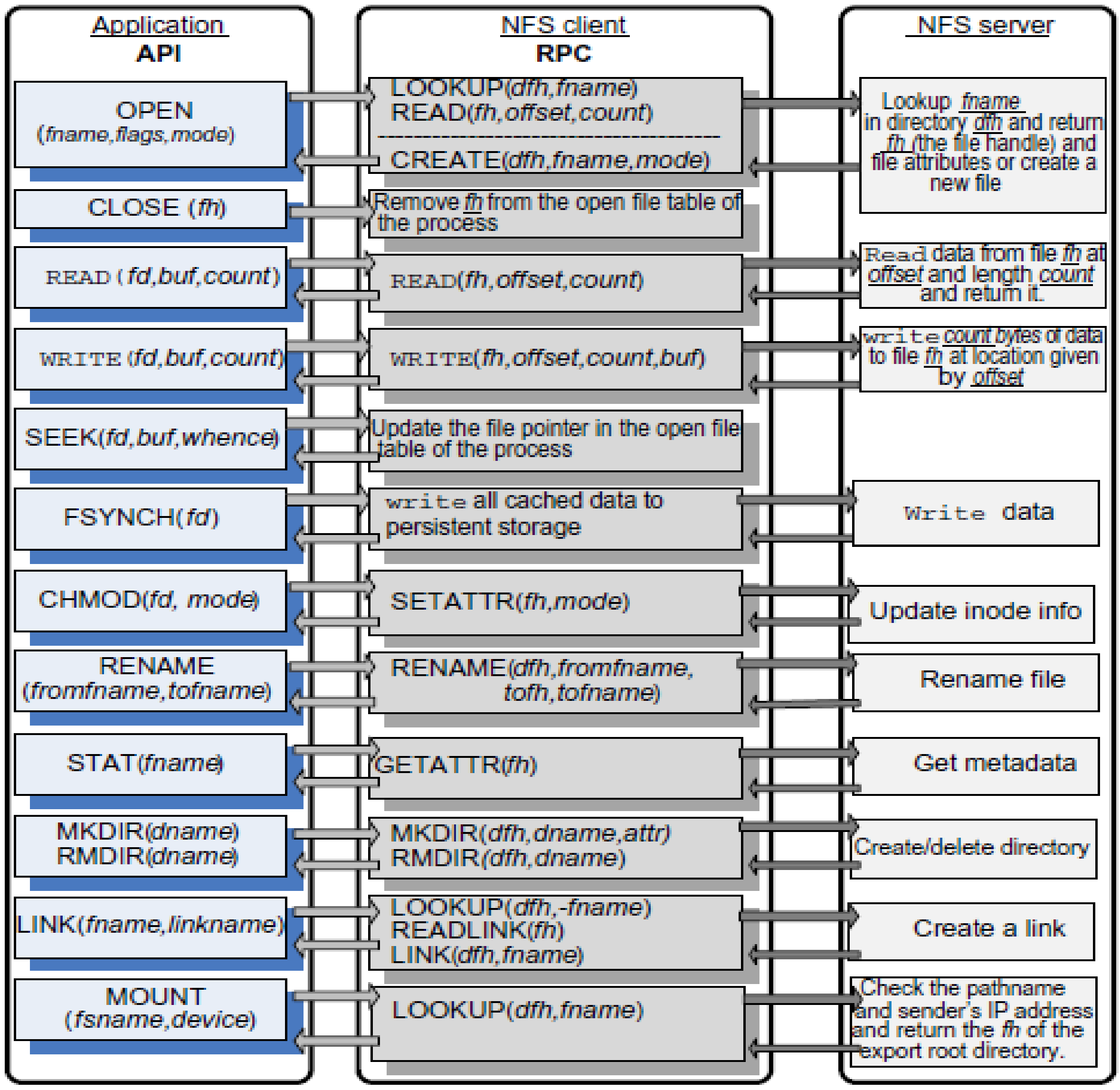


FIG: The NFS client-server interaction

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). 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.



Application API
NFS client RPC
NFS server
OPEN (fname,flags,mode)
LOOKUP(dfh,fname)
READ(fh,offset,count)
CREATE(dfh,fname,mode)
Lookup fname in directory dfh and return fh (the file handle) and file attributes or create a new file
CLOSE (fh)
Remove fh from the open file table of the process
READ (fd,buf,count)
READ(fh,offset,count)
Read data from file fh at offset and length count and return it.
WRITE (fd,buf,count)
WRITE(fh,offset,count,buf)
write count bytes of data to file fh at location given by offset
SEEK(fd,buf,whence)
Update the file pointer in the open file table of the process
FSYNCH(fd)
write all cached data to persistent storage
Write data
CHMOD(fd, mode)
SETATTR(fh,mode)
Update inode info
RENAME (fromfname,tofname)
RENAME(dfh,fromfname, tofh,tofname)
Rename file
STAT(fname)
GETATTR(fh)
Get metadata
MKDIR(dname)
RMDIR(dname)
MKDIR(dfh,dname,attr)
RMDIR(dfh,dname)
Create/delete directory
LINK(fname,linkname)
LOOKUP(dfh,-fname)
READLINK(fh)
LINK(dfh,fname)
Create a link
MOUNT (fsname,device)
LOOKUP(dfh,fname)
Check the pathname and sender's IP address and return the fh of the export root directory.

Fig: The API of the *Unix* File System and the corresponding RPC issued by an NFS client to the NFS server

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 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. 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.
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.

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.

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, 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.

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.

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. 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.

The General Parallel File System (GPFS) was developed at IBM in the early 2000s as a successor to the Tiger Shark multimedia file system.

- GPFS is a parallel file system 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.
- The maximum file size is (2^{63}–1) bytes. A file consists of blocks of equal size, ranging from 16 KB to 1MB 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.
- 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.

- 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

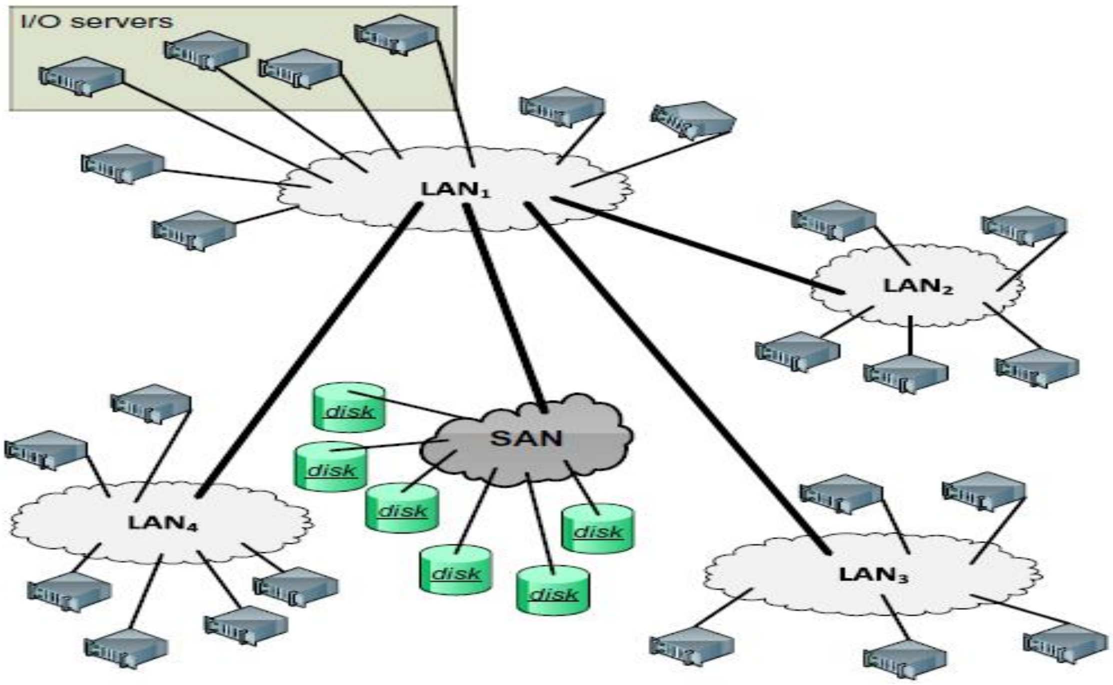


Fig: A GPFS configuration.

- 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. Access to metadata is synchronized.
- 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.
- 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.

\

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 peta bytes of storage to a large user community with diverse needs.

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.
4. Eliminate caching at the client site.
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

- The architecture consists of ***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.
- 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.

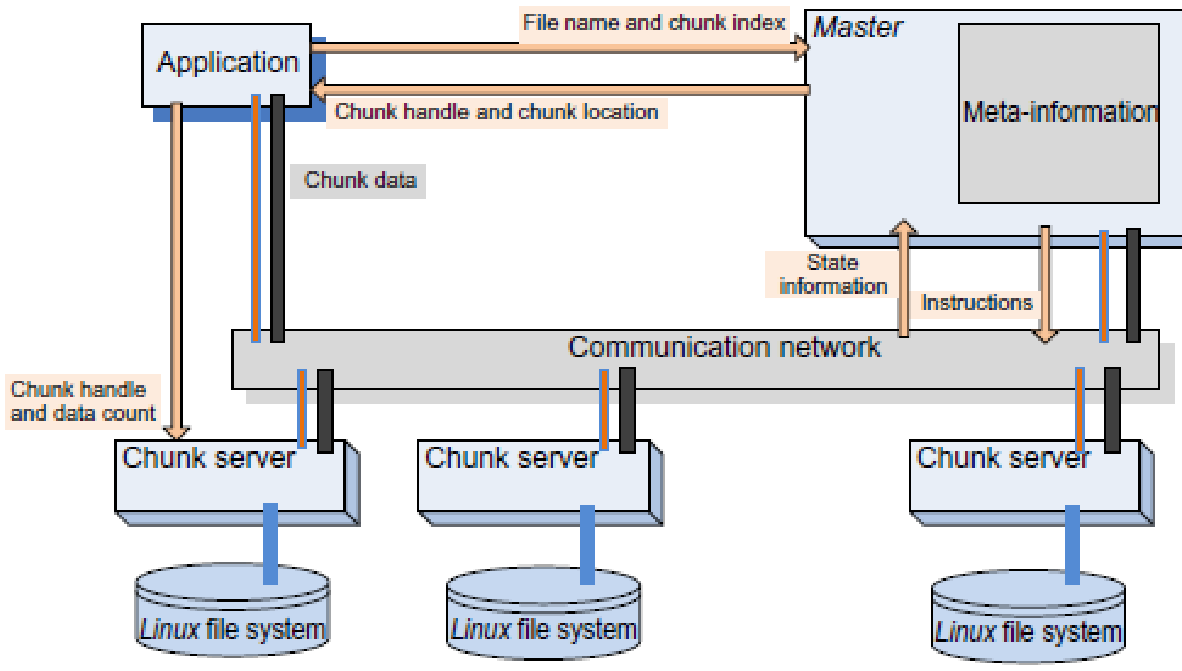


Fig: The architecture of a GFS cluster.

- 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.
- The system supports an efficient check pointing procedure based on *copy-on-write* to construct system snapshots.
- 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.

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.

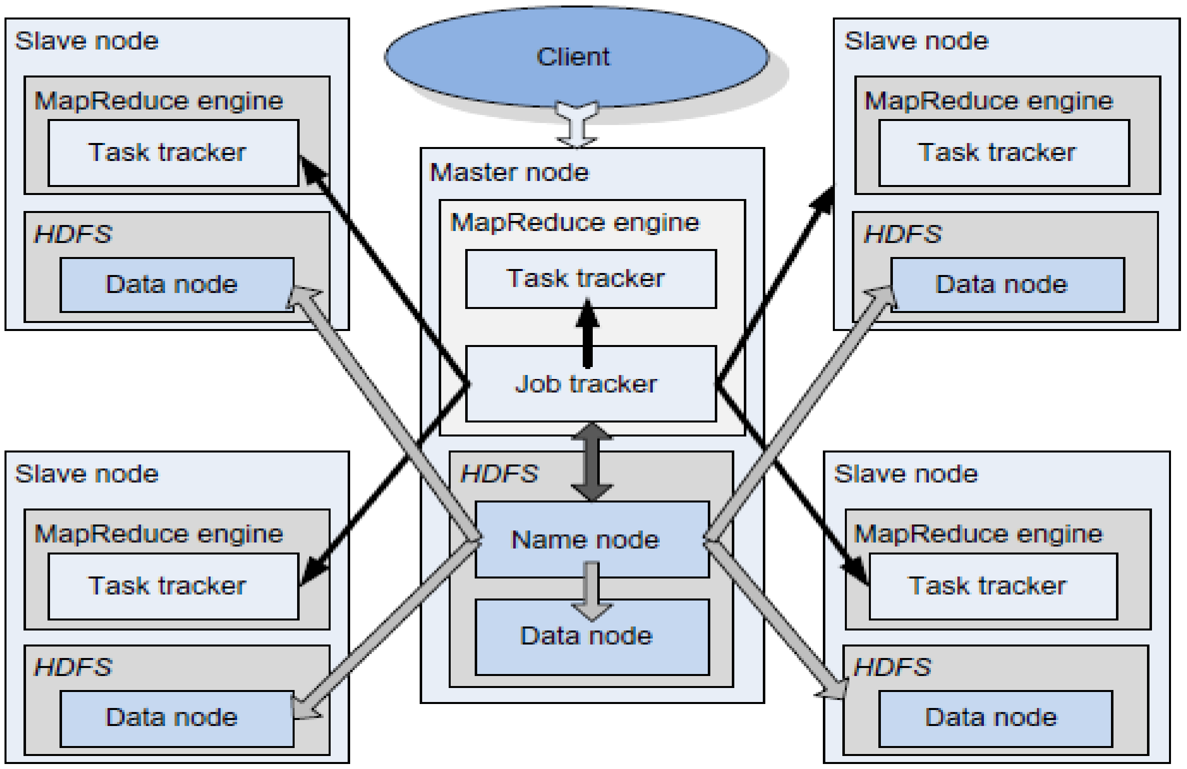
3. The client sends a write request to the primary once it has received the acknowledgments from all chunk servers holding replicas of the chunk. The primary identifies mutations by consecutive sequence numbers.
4. The primary sends the write requests to all secondaries.
5. Each secondary applies the 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 .

APACHE HADOOP

Apache Hadoop, an open-source, Java-based software system. ***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. A ***Hadoop*** system has two components, a *MapReduce* engine and a database (see below Figure). The database could be the ***Hadoop File System (HDFS),*** Amazon *S3*, or *CloudStore*. *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 per formant. The ***Hadoop*** engine on the master of a multinode cluster consists of a ***job tracker*** and a ***task tracker***, where as 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.
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.

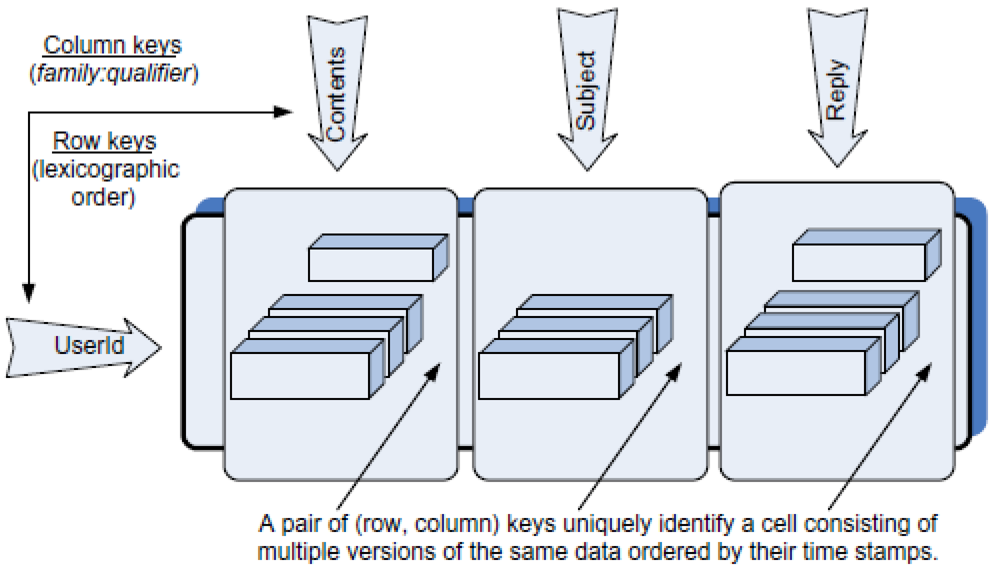


A *Hadoop* cluster using *HDFS*.

BIG TABLE

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 used to store user data as well as system information. To guarantee atomic read and write operations, it uses the ***Chubby*** distributed lock service; 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 organization of a *BigTable* (see above Figure) 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.


Column keys
(family:qualifier)
Row keys
(lexicographic order)
Contents
Subject
Reply
UserId
A pair of (row, column) keys uniquely identify a cell consisting of multiple versions of the same data ordered by their time stamps.

A *BigTable* example

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

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.

MEGASTORE

Megastore is scalable storage for online services. The system, distributed over several data centers, has a very large capacity, 1PBin 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.

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 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 peer.

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.

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.

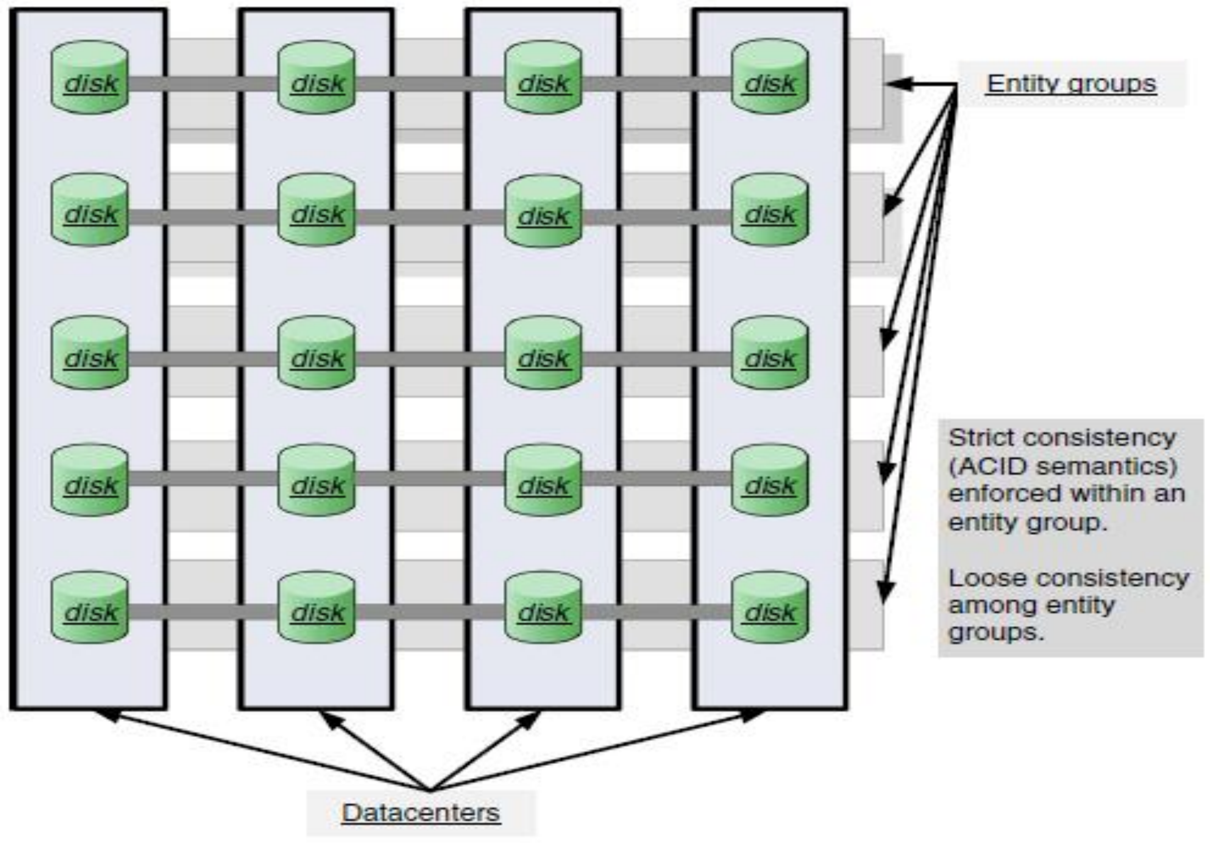

disk
Entity groups
Strict consistency (ACID semantics) enforced within an entity group.
Loose consistency among entity groups.
Datacenters

Fig: megastore organization

Amazon Web Services: Amazon has been a leader in providing public cloud services (http://aws.amazon.com/). Amazon applies the IaaS model in providing its services. The below fig. shows the AWS architecture. EC2 provides the virtualized platforms to the host VMs where the cloud application can run. S3 (Simple Storage Service) provides the object-oriented storage service for users. EBS (Elastic Block Service) provides the block storage interface which can be used to support traditional applications. SQS stands for Simple Queue Service, and its job is to ensure a reliable message service between two processes. Different from Google, Amazon provides a more flexible cloud computing platform for developers to build cloud applications. Small and medium-size companies can put their business on the Amazon cloud platform. Using the AWS platform, they can service large numbers of Internet users and make profits through those paid services.

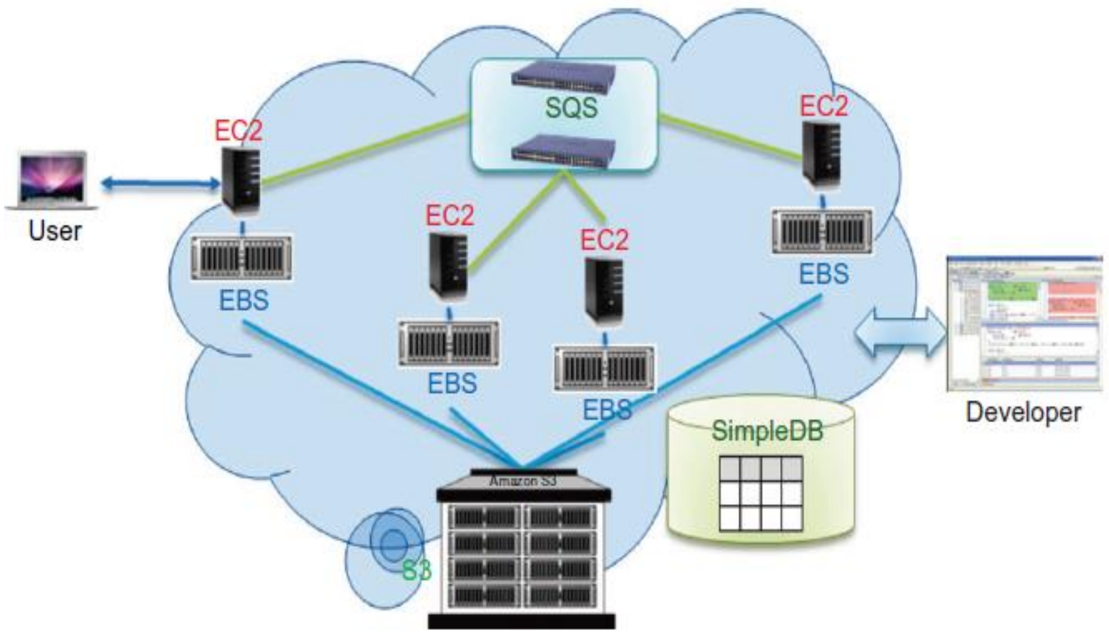


Fig: Amazon cloud computing infrastructure

Amazon offers the Simple Queue Service (SQS) and Simple Notification Service (SNS), Elastic load balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances and allows you to avoid non operating nodes and to equalize load on functioning images. Both auto-scaling and elastic load balancing are enabled by Cloud Watch which monitors running instances.

Programming on Amazon EC2

Amazon was the first company to introduce VMs in application hosting. Customers can rent VMs instead of physical machines to run their own applications. By using VMs, customers can load any software of their choice. The elastic feature of such a service is that a customer can create, launch, and terminate server instances as needed, paying by the hour for active servers. Amazon provides

Several types of preinstalled VMs. Instances are often called Amazon Machine Images (AMIs) which are preconfigured with operating systems based on Linux or Windows, and additional software.

The workflow to create an AMI is

Create an AMI→Create Key Pair→Configure Firewall→Launch

There exist three types of AMI

Private AMI : images are created by us and which are private by default.

Public AMI: images are created by the users and released to the AWS community.

Paid QAMI: one can create image providing specify functions.

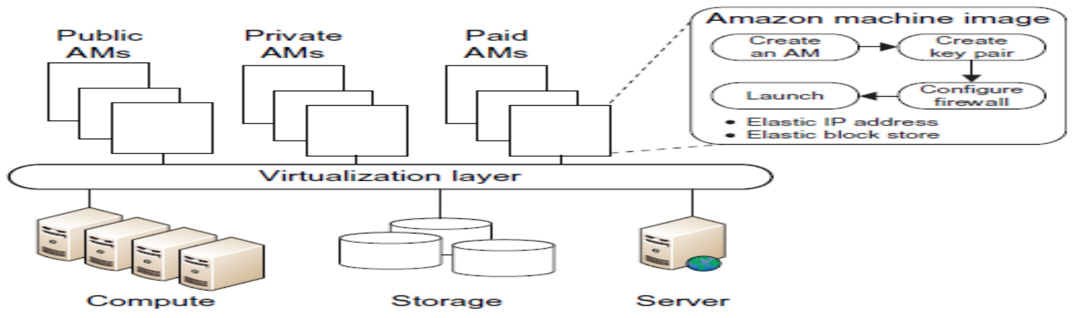


Fig: Amazon EC2 execution environment.

Amazon Simple Storage Service (S3) : Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. S3 provides the object-oriented storage service for users. Users can access their objects through Simple Object Access Protocol (SOAP) with either browsers or other client programs which support SOAP.

The fundamental operation unit of S3 is called an object. Each object is stored in a bucket and retrieved via a unique, developer-assigned key. In other words, the bucket is the container of the object. Besides unique key attributes, the object has other attributes such as values, metadata, and access control information. From the programmer's perspective, the storage provided by S3 can be viewed as a very coarse-grained key-value pair. There are two types of web service interface for the user to access the data stored in Amazon clouds. One is a REST (web 2.0) interface, and the other is a SOAP interface.

Some key features of S3:

- Redundant through geographic dispersion.
- Designed to provide 99.999999999 percent durability and 99.99 percent availability of objects
- Authentication mechanisms to ensure that data is kept secure from unauthorized access. • Per-object URLs and ACLs (access control lists).
- Default downloads protocol of HTTP. A BitTorrent protocol interface is provided to lower costs for high-scale distribution.
- $0.055 (more than 5,000 TB) to 0.15 per GB per month storage (depending on total amount).
- First 1 GB per month input or output free and then $.08 to $0.15 per GB for transfers outside an S3 region.
- There is no data transfer charge for data transferred between Amazon EC2 and Amazon S3.

Amazon Elastic Block Store (EBS) and SimpleDB

The Elastic Block Store (EBS) provides the volume block interface for saving and restoring the virtual images of EC2 instances. Traditional EC2 instances will be destroyed after use. The status of EC2 can now be saved in the EBS system after the machine is shut down. Users can use EBS to save persistent data and mount to the running instances of EC2.

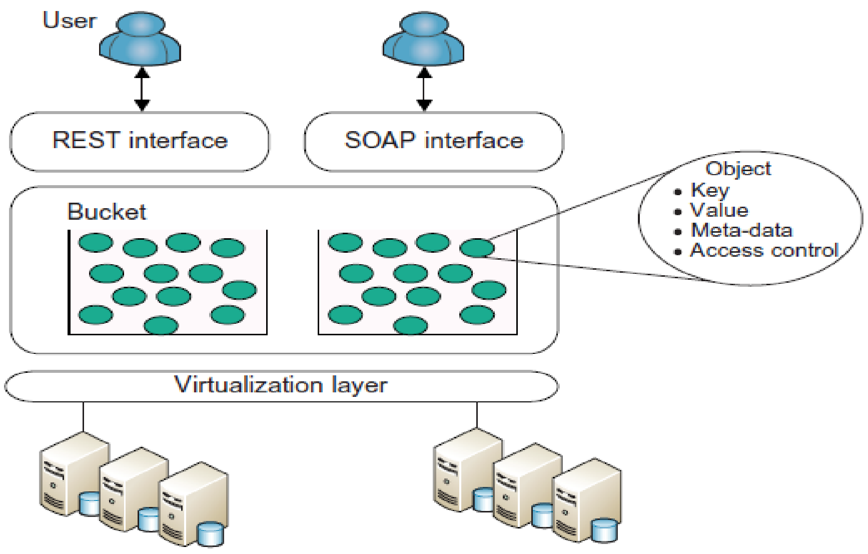


Fig: Amazon S3 execution environment

Multiple volumes can be mounted to the same instance. These storage volumes behave like raw, unformatted block devices, with user-supplied device names and a block device interface.

Amazon SimpleDB Service: SimpleDB provides a simplified data model based on the relational database data model. Structured data from users must be organized into domains. Each domain can be considered a table. The items are the rows in the table. A cell in the table is recognized as the value for a specific attribute (column name) of the corresponding row. This is similar to a table in a relational database .