SYLLABUS: STORAGE SYSTEMS

Evolution of storage technology, storage models, file systems and database, distributed file systems, general parallel file systems. Google file system.

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.

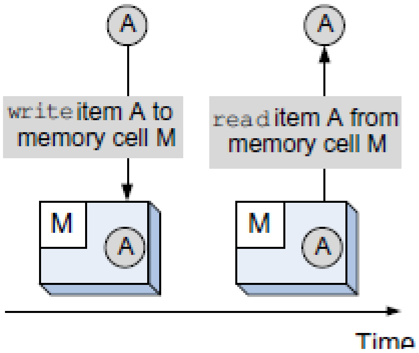
- These rapid technological advancements have changed the balance between initial investment instorage devices and system management costs.
- Now the cost of storage management is the dominant element of the total cost of a storage system.
- This effect favors the centralized storage strategy supported by a cloud a centralized approach can automate some of the storage management functions, such as replication and backup, and thus reduce substantially the storage management cost.
- The storage systems face substantial pressure because the volume of data generated has increased exponentially during the past few decades.
- 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.
- As the volume of data increases, new methods and algorithms for data mining that require powerful computing systems have been developed.
- 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.
- Nowadays large-scale systems are built with offthe-shelf components. The emphasis of the design philosophy has shifted **from *performance at any cost*** to ***reliability at the lowest possible cost*.**

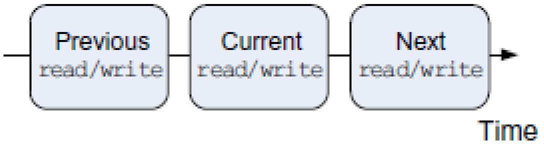
STORAGE MODELS, FILE SYSTEMS AND DATABASE

1.Storage Model: 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.
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.





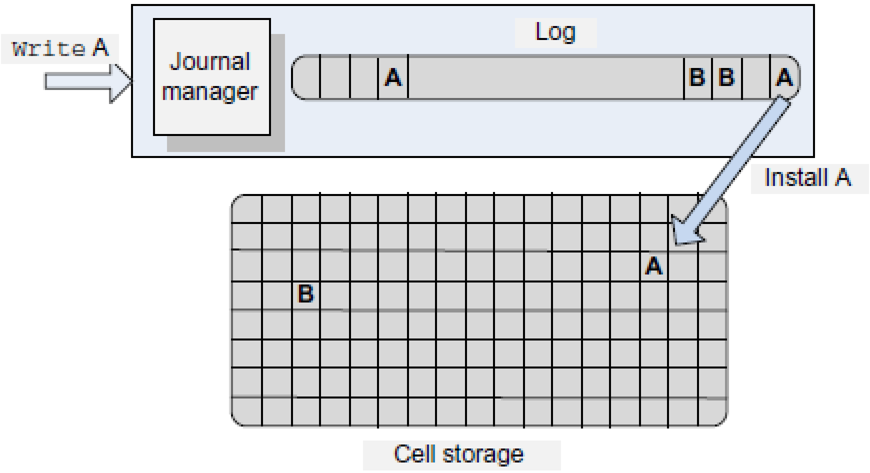
- ✓ **Journal storage** is a fairly elaborate organization for storing composite objects such as records consisting of multiple fields. Journal storage consists of a *manager* and *cell storage*, where the entire history of a variable is maintained, rather than just the current value.

The user does not have direct access to the *cell storage*; instead the user can request the *journal manager* to

- start a new action;
- read the value of a cell;
- write the value of a cell;
- commit an action; or abort an action.

The *journal manager* translates user requests to commands sent to the cell storage:

- read a cell;
- write a cell;
- allocate a cell; or deallocate a cell.



- ❖ 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.
- ❖ 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.

2.File system: 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:

1. **Network File Systems (NFSs)** : 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
2. **Storage Area Networks (SANs)** : 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. 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
3. **Parallel File Systems (PFSs) :** 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.

3. 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.
- ❖ Most cloud applications are data intensive and test the limitations of the existing infrastructure, 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.
- ❖ 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.

DISTRIBUTED FILE SYSTEMS

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 onworkstations, proceeded to develop the NFS in the early 1980s.

1. **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.

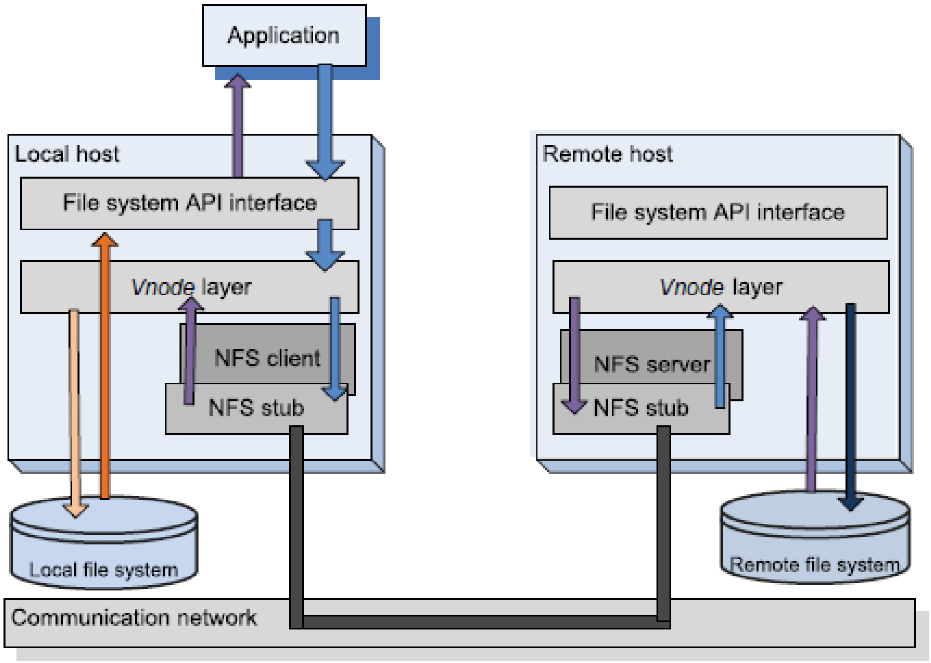
• Ensure that the system would be widely used and thus support clients running on different operating systems.

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

Three important characteristics of the *Unix* File System:

- 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 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 procedurecalls (RPCs).



NFS client-server interaction

❖ 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.

❖ NFS uses a *vnode* layer to distinguish between operations on local and remote files. 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.

Disadvantage: communication failures could sometimes lead to unexpected behavior

2. **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.

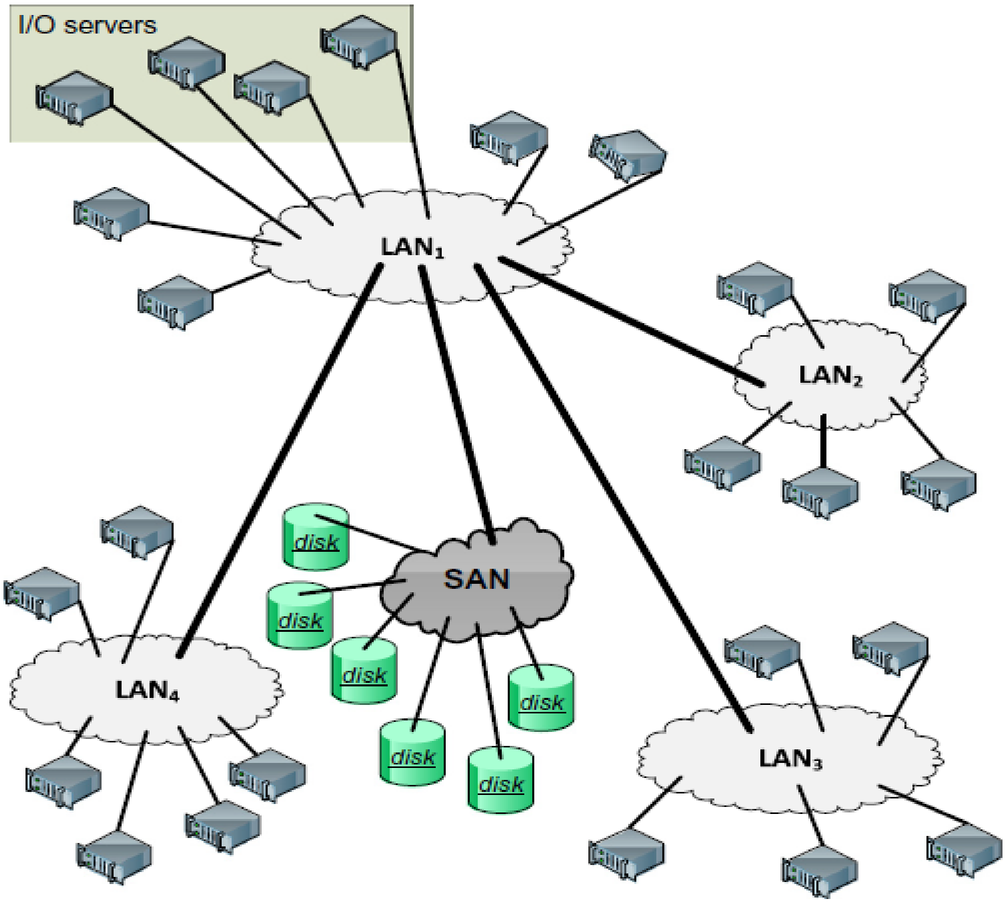
3.Sprite Network File System (SFS). SFS is a component of the Sprite network operating system. SFS supports non-write-through caching of files on the client as well as the server systems.

- 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 satisfiedat 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 timewhen a typicalworkstation 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 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.
- 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 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.

GENERAL PARALLEL FILE SYSTEMS

- Parallel I/O implies execution of multiple input/output operations concurrently. Support for parallel I/O is essential to the performance of many applications.
- Distributed file systems became ubiquitous, the natural next step in the evolution of the file systemwas 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.
- 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. GPFSwas 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.Afile consists of blocks of equal size, ranging from 16 KBto 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.
- 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.
- 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.
- Data striping allows concurrent access and improves performance but can have unpleasant side-effects. The system uses RAID devices with the stripes equal to the block size and dual-attached RAID controllers.



GPFS CONFIGURATION

- ❖ Consistency and performance, critical to any distributed file system, are difficult to balance. 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 $f\,p1$ and the second one at offset $f\,p2 > f\,p1$, the range of the tokens for the two tokens are $[0, f\,p2]$ and $[f\,p2,\infty]$, respectively, and the two

nodes can operate concurrently, without the need for further negotiations. Byte-range tokens are rounded to block boundaries.

- 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. Access to metadata is also synchronized.
- 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 subblocks to reduce disk fragmentation for small files; thus, the block map has 32 bits to indicate whether a subblock is free or used.

GOOGLE FILE SYSTEM(GFS)

- 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.
- The 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, butwithout placing an additional burden on the application developers

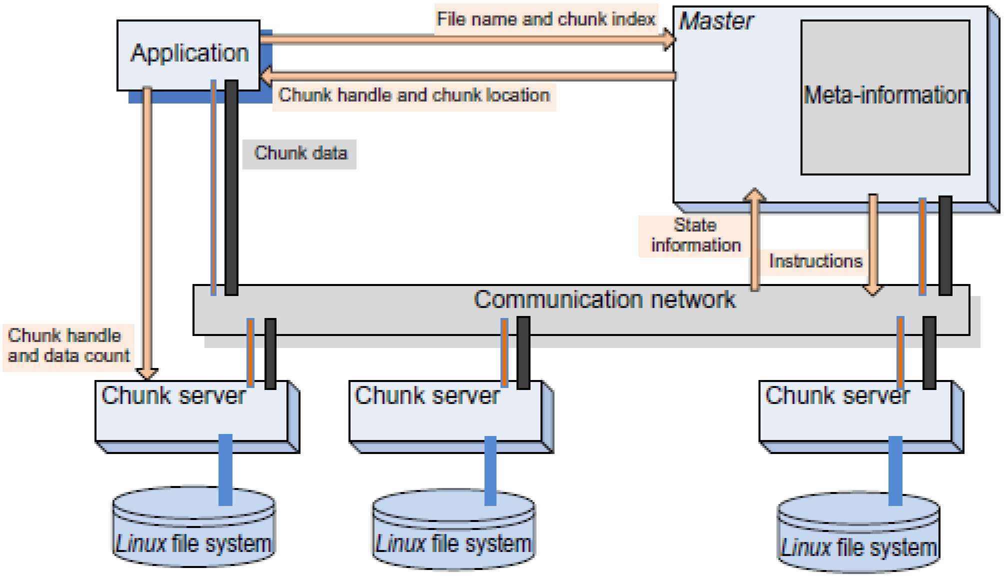
Several design decisions were made as a result of this analysis:

- ✓ Segment a file in large chunks.
- ✓ Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file.
- ✓ 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.

- ✓ 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.
- ✓ Ensure consistency by channeling critical file operations through a *master*, a component of the cluster that controls the entire system.
- ✓ Minimize the involvement of the *master* in file access operations to avoid hot-spot contention and to ensure scalability.
- ✓ Support efficient checkpointing and fast recovery mechanisms.
- ✓ 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.
- ❖ The architecture of a GFS cluster is illustrated in the below figure. 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



Architecture of a GFS cluster

- 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. 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.

- The consistencymodel 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.
- 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.