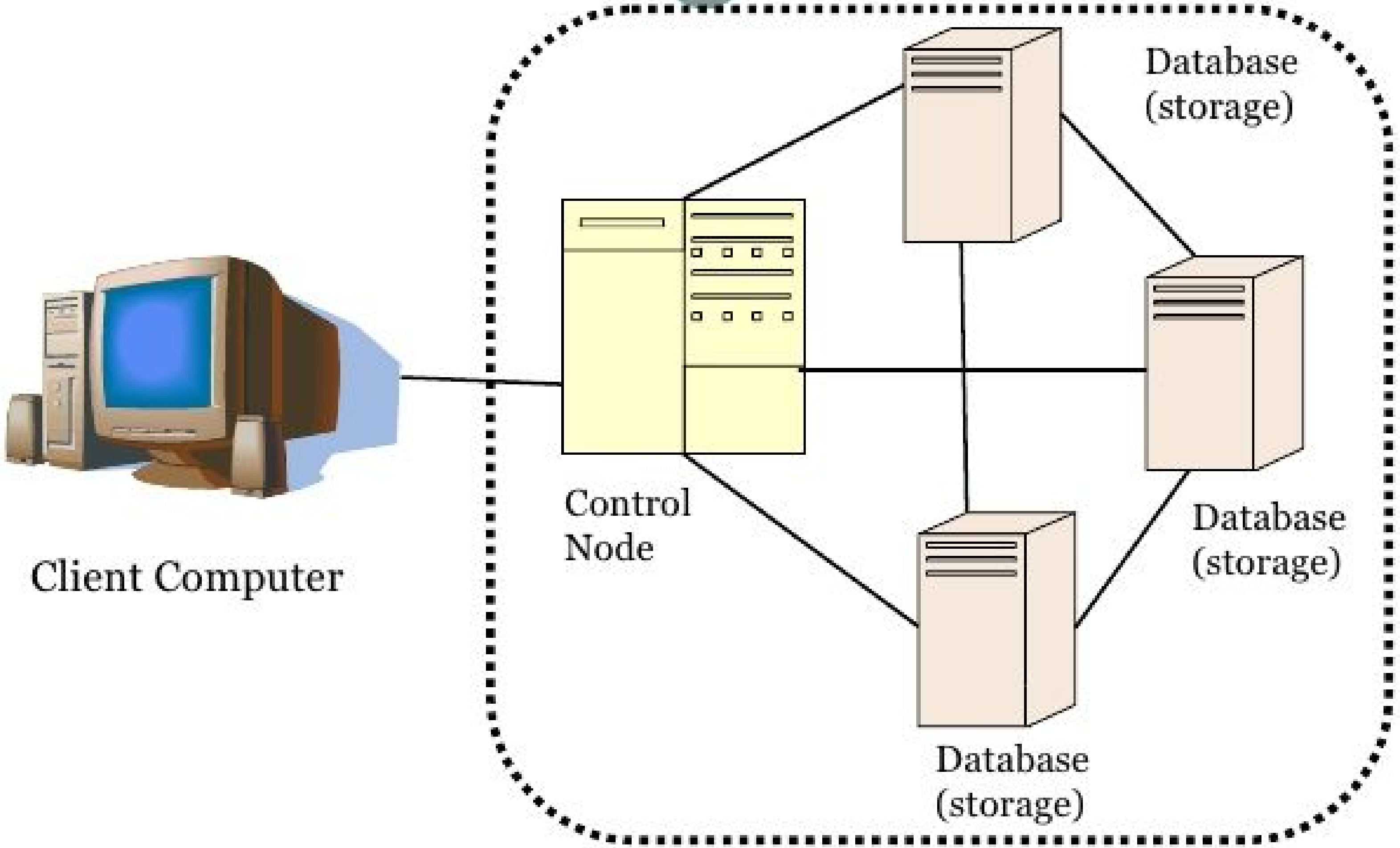
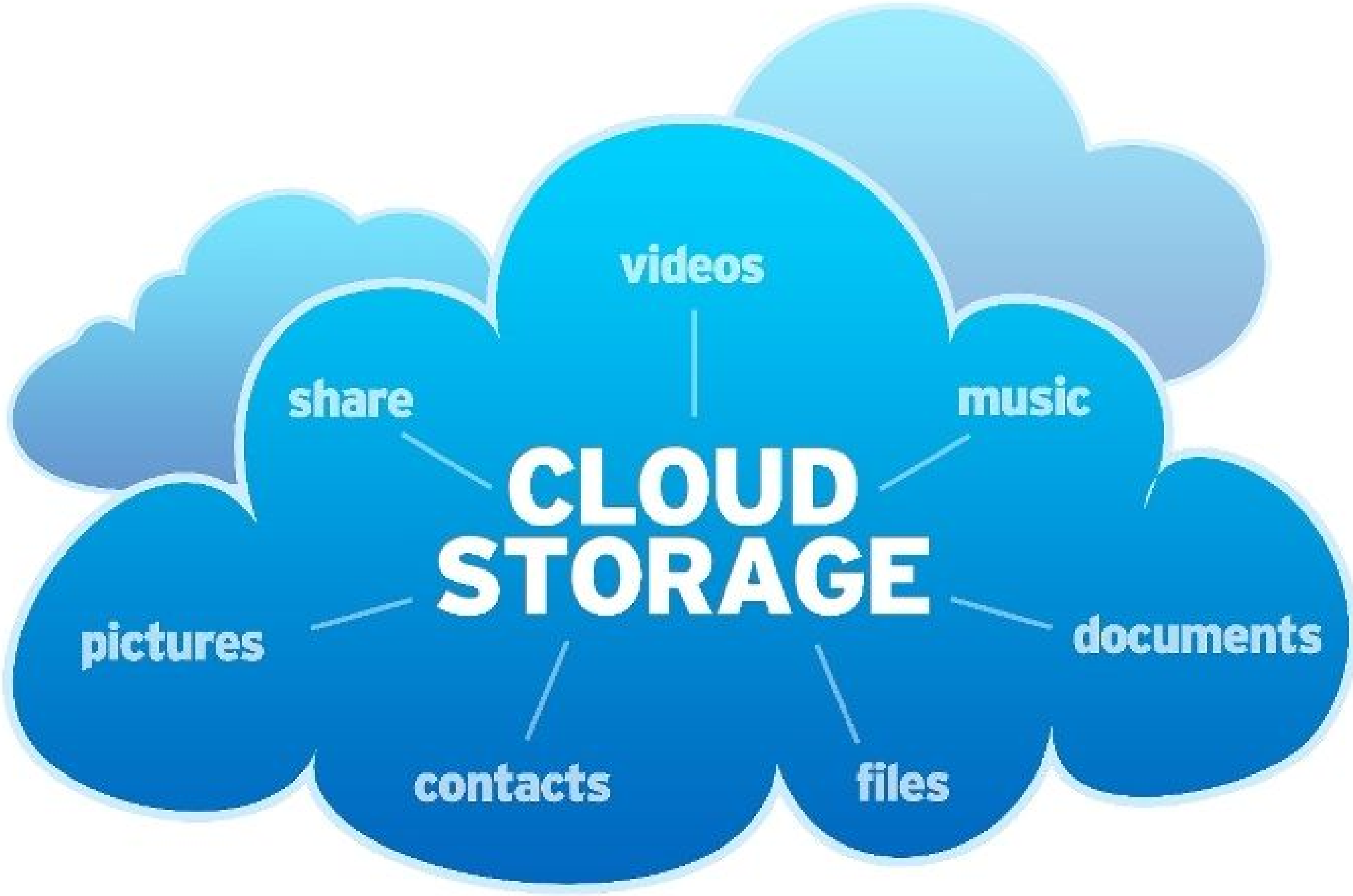
Storage Systems

CHAPTER 07

Cloud Storage System Architecture




videos
share
music
CLOUD STORAGE
pictures
documents
contacts
files

Data storage on a cloud

- 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 and local storage take advantage of cloud environments to store audio and video files.
- Clouds provide an ideal environment for multimedia content delivery.

Data storage on a cloud

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

Data storage on a cloud

- Storage and processing on the cloud are intimately tied to one another.
  - 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.
  - Strategies to reduce the access time and to support real-time multimedia access are necessary to satisfy the requirements of content delivery.

Big data

- New concept ➔ reflects the fact that many applications use data sets that cannot be stored and processed using local resources.
- Three-dimensional phenomena.
  - Increased volume of data.
  - Requires increased processing speed to process more data and produce more results.
  - Involves a diversity of data sources and data types.

Big data

- Applications in many areas of science, including genomics, structural biology, high-energy physics, astronomy, meteorology, and the study of the environment, carry out complex analysis of data sets, often of the order of terabytes (TBs).
  - In 2010, the four main detectors at the Large Hadron Collider (LHC) produced 13 PB of data.
  - The Sloan Digital Sky Survey (SDSS) collects about 200 GB of data per night.

Big data

- Massive amounts of data - in 2013
  - The Internet video will generate over 18 EB/month.
  - Global mobile data traffic will reach 2 EB/month.

  (1 EB = 10^{18} bytes, 1 PB = 10^{15} bytes, 1 TB = 10^{12} bytes, 1 GB = 10^{9} bytes)

The evolution of storage technology

- The technological capacity to store information has grown over time at an accelerated pace.
  - 1986 - 2.6 EB ➔ 1, CD-ROM /person.
  - 1993 - 15.8 EB ➔ 4 CD-ROM/person.
  - 2000 - 54.5 EB ➔ 12 CD-ROM/person.
  - 2007 -295.0 EB➔ 61 CD-ROM/person.

The evolution of storage technology

- Hard disk drives (HDD) - during the 1980-2003 period:
  - Storage density of has increased by four orders of magnitude from about 0.01 Gb/in^2 to about 100 Gb/in^2
  - Prices have fallen by five orders of magnitude to about 1 cent/MB.
  - HDD densities are projected to climb to 1,800 Gb/in^2 by 2016, from 744 Gb/in^2 in 2011.

The evolution of storage technology

- Dynamic Random Access Memory (DRAM) - during the period 1990-2003:
  - The density increased from about 1 Gb/in^2 in 1990 to 100 Gb/in^2 .
  - The cost has fallen from about $80/MB to less than $1/MB.

Storage system challenge

- 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/extraordinary rate.

Storage system challenge

- Mobile devices, such as smart-phones and tablets, record static images, as well as movies and have limited local storage capacity, so they transfer the data to cloud storage systems.
- Sensors, surveillance cameras, and digital medical imaging devices generate data at a high rate and dump it onto storage systems accessible via the Internet.
- Online digital libraries, ebooks, and digital media, along with reference data, add to the demand for massive amounts of storage.

Design philosophy

- The emphasis of the design philosophy has shifted from ***performance at any cost*** to ***reliability at the lowest possible cost***. This shift is evident in the evolution of ideas, from the early distributed file systems of the 1980s, such as the Network File System (NFS) and the Andrew File System (AFS), to today's Google File System (GFS) and the *Megastore.*

Megastore

- **Megastore** is a storage system developed to meet the requirements of today's interactive online services.
- **Megastore** blends the scalability of a NoSQL data store with the convenience of a traditional RDBMS.
- Megastore handles 3 billion writes and 20 billion reads daily on 8 PB of data across the globe supporting millions of users of Google Apps.

Storage models

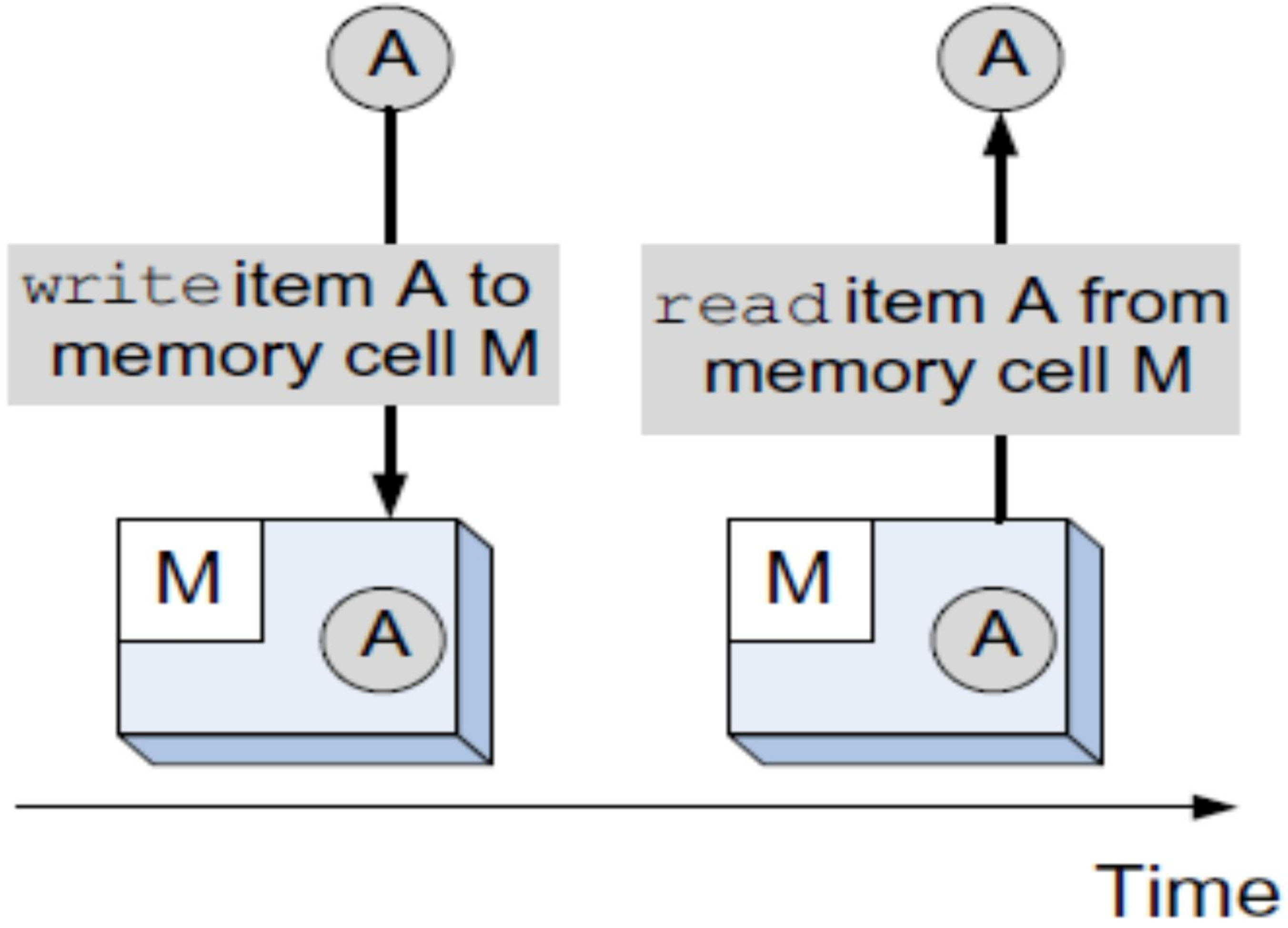
- A *storage model* describes the layout of a data structure in physical storage; The physical storage can be a local disk, a removable media, or storage accessible via a network.
- A data model ➔ captures the most important logical aspects of a data structure in a database.

Two abstract models of storage

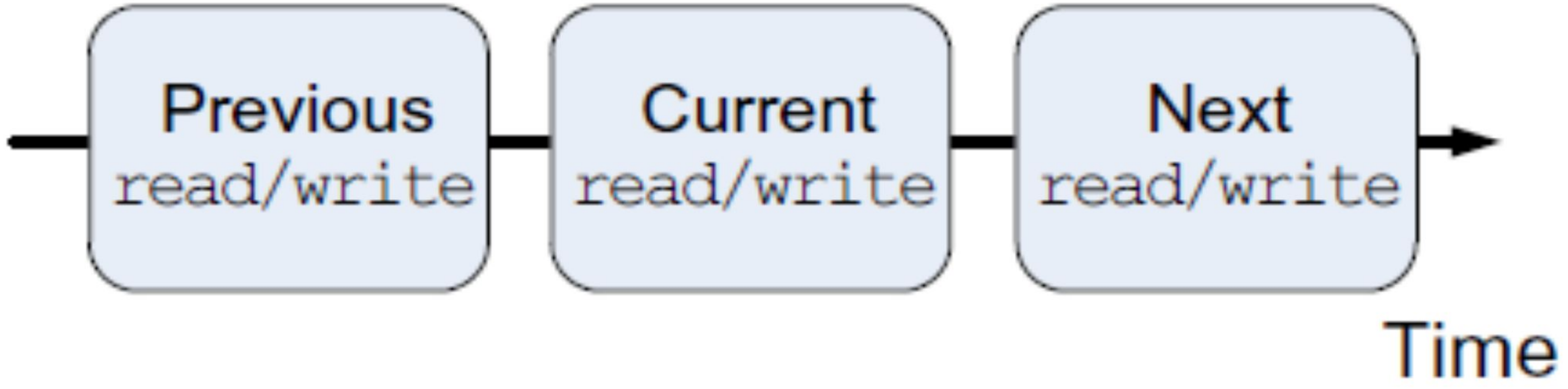
- Cell storage
- Journal storage

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.
- The read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model and in particular of cell storage



`read/write` *coherence*: the result of a `read` of memory cell M should be the same as the most recent `write` to that cell



ad
e

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`.

Journal storage

- *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.

Journal storage

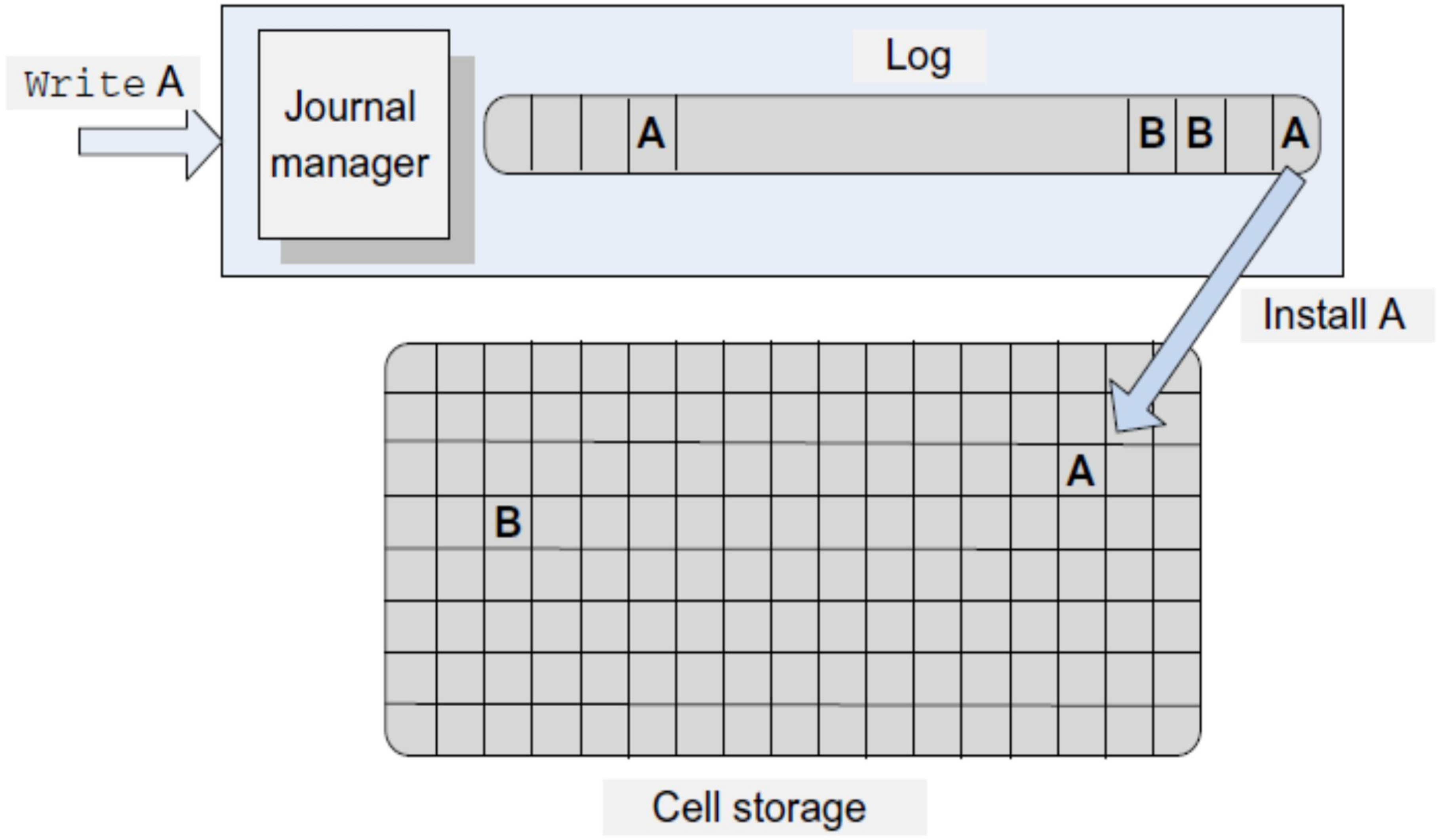
- The user does not have direct access to the *cell storage*; instead the user can request the *journal manager* to (i) start a new action; (ii) read the value of a cell; (iii) write the value of a cell; (iv) commit an action; or (v) abort an action.
- The *journal manager* translates user requests to commands sent to the cell storage: (i) read a cell; (ii) write a cell; (iii) allocate a cell; or (iv) deallocate a cell.

Journal storage

- A *log* contains the entire history of all variables. The log is stored on nonvolatile media (disk) of *journal storage*.
- 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.

Journal storage

- All-or-nothing system keeps track of the changes made to the journal storage in circular log in a dedicated area of the file system before committing them to the main file system.
- In the event of a system crash or power failure, such file systems are quicker to bring back online and less likely to become corrupted.



Two variables, A and B, in the log and cell storage are shown. A new value of A is written first to the log and then installed on cell memory at the unique address assigned to A.

Logical and physical organization of a file

- File ➔ a linear array of cells stored on a persistent storage device. 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 size dictated by the physical media.
- File pointer➔ identifies a cell used as a starting point for a **read** or **write** operation.
- The logical organization of a file ➔ reflects the data model, the view of the data from the perspective of the application.
- The physical organization of a file ➔ reflects the storage model and describes the manner the file is stored on a given storage media

File system

- A *file system* consists of a collection of *directories and files*. Each directory provides information about a set of files.
  - Traditional – Unix File System.
  - Distributed File system
  - Storage Area Networks (SAN)
  - Parallel File Systems (PFS)

Unix File System (UFS)

- The layered design provides <u>flexibility</u>.
  - The layered design allows UFS to separate the concerns for the physical file structure from the logical one.
  - The *vnode* layer allowed UFS to treat local and remote file access uniformly.
- The hierarchical design supports <u>scalability</u> of the file system. It allows grouping of files into special files called *directories* and supports multiple levels of directories.
- File system is the collections of directories and files.
- A local file is uniquely identified by a *file descriptor (fd)*, generally an index in the open file table.

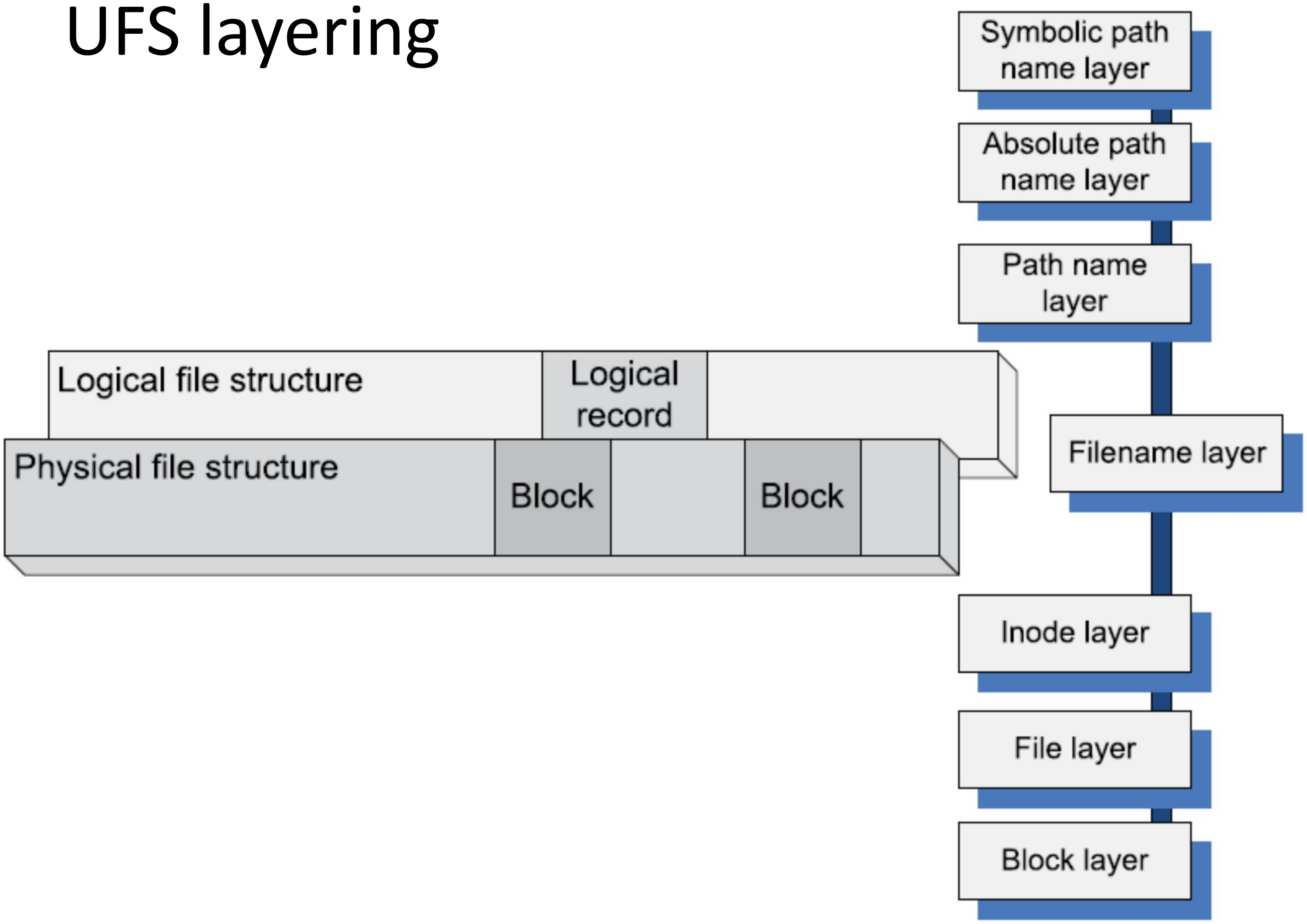
Unix File System (UFS)

- The metadata supports a systematic design philosophy of the file system and device-independence.
  - Metadata includes: file owner, access rights, creation time, time of the last modification, file size, the structure of the file and the persistent storage device cells where data is stored.
  - the inode layer provides the metadata for the objects (files and directories). The inodes are kept on persistent media together with the data.

Unix File System (UFS)

- 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 filename layer mediates between the machine-oriented and the user-oriented views of the file system

UFS layering



Distributed File System

- A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer.
- A distributed file system (DFS) is a file system with data stored on a server. The data is accessed and processed as if it was stored on the local client machine.
- The servers have full control over the data and give access control to the clients.
- A DFS manages files and folders across multiple computers.

Distributed File System

- When the client retrieves a file from the server, the file appears as a normal file on the client machine, and the user is able to work with the file in the same ways as if it were stored locally on the workstation.
- When the user finishes working with the file, it is returned over the network to the server, which stores the now-altered file for retrieval at a later time.

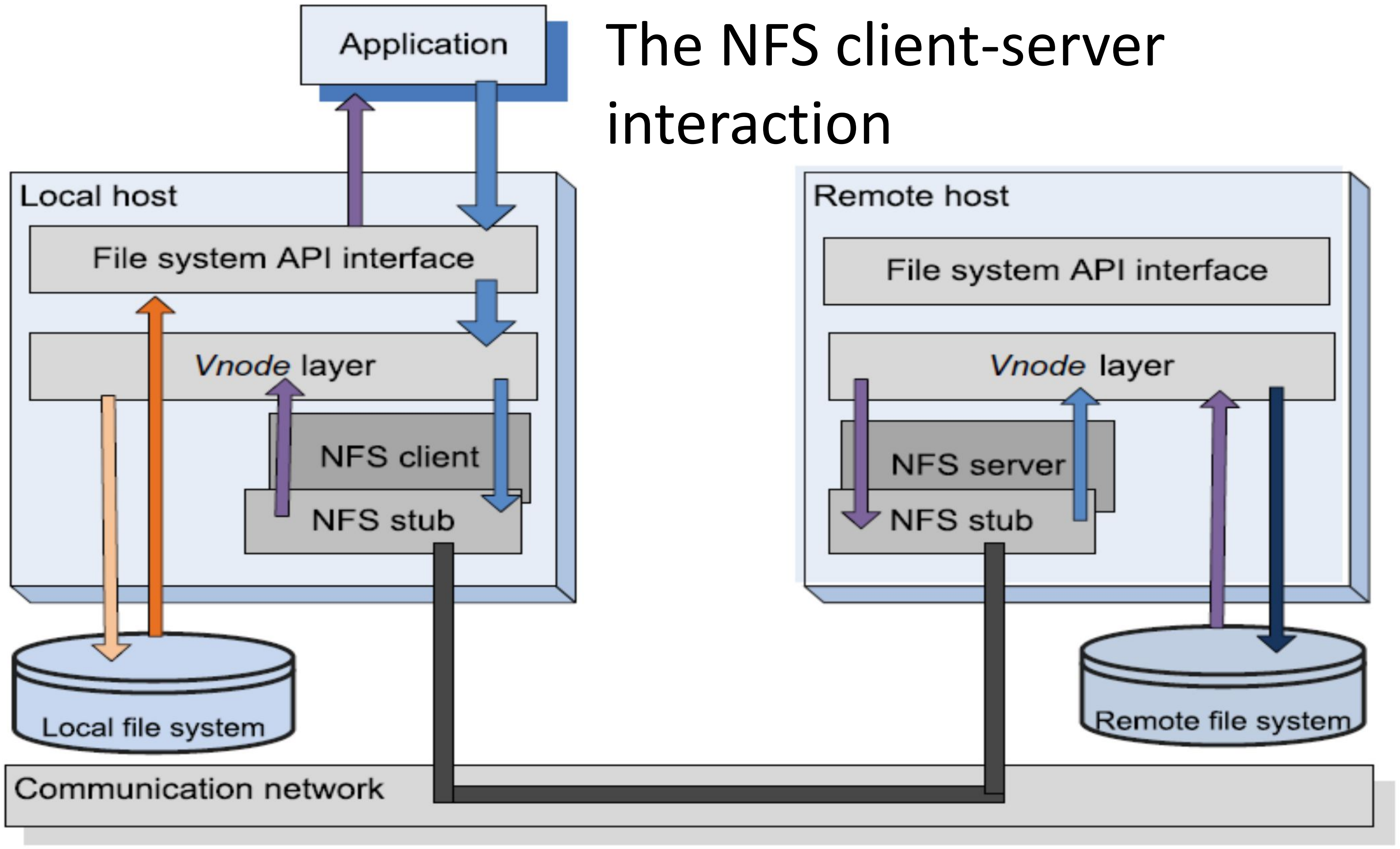
Examples of distributed file systems.

- Sun Microsystems' Network File System (NFS)
- Novell NetWare,
- Microsoft's Distributed File System,
- IBMs DFS
- **Andrew File System (AFS).**

NFS

- The Network File System is based on the client-server paradigm.
- A Network File System (NFS) allows remote hosts to mount file systems over a network and interact with those file systems as though they are mounted locally.
- 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 allows the system to locate the remote file system and the file on that system.

The NFS client-server interaction



The vnode layer implements file operation in a uniform manner, regardless of whether the file is local or remote.

NFS

- The NFS protocol is one of several distributed file system standards for network-attached storage (NAS).
- NFS the most widely used network-based file system.
- The NFS has undergone significant transformations over the years
- An NFS server could be a single point of failure.

Storage Area Networks (SAN)

- SAN is a **network** which provides access to consolidated, block level data **storage**.
- SAN connects storage systems and computational servers
- Allow cloud servers to deal with changes in the storage configuration. The storage in a SAN can be pooled and then allocated based on the needs of the servers. A SAN-based implementation of a file system can be expensive, as each node must have a Fibre Channel adapter to connect to the network.

Parallel File Systems (PFS)

- Is a type of distributed file system that distributes file data across multiple servers and provides for concurrent access by multiple tasks of a parallel application.
- It is scalable, capable of distributing files across a large number of nodes, with a global naming space.
- Several I/O nodes serve data to all computational nodes.
- It includes also a metadata server which contains information about the data stored in the I/O nodes.
- The interconnection network of a PFS could be a SAN.

Parallel File Systems (PFS)

- Parallel I/O implies execution of multiple input/output operations concurrently.
- 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.

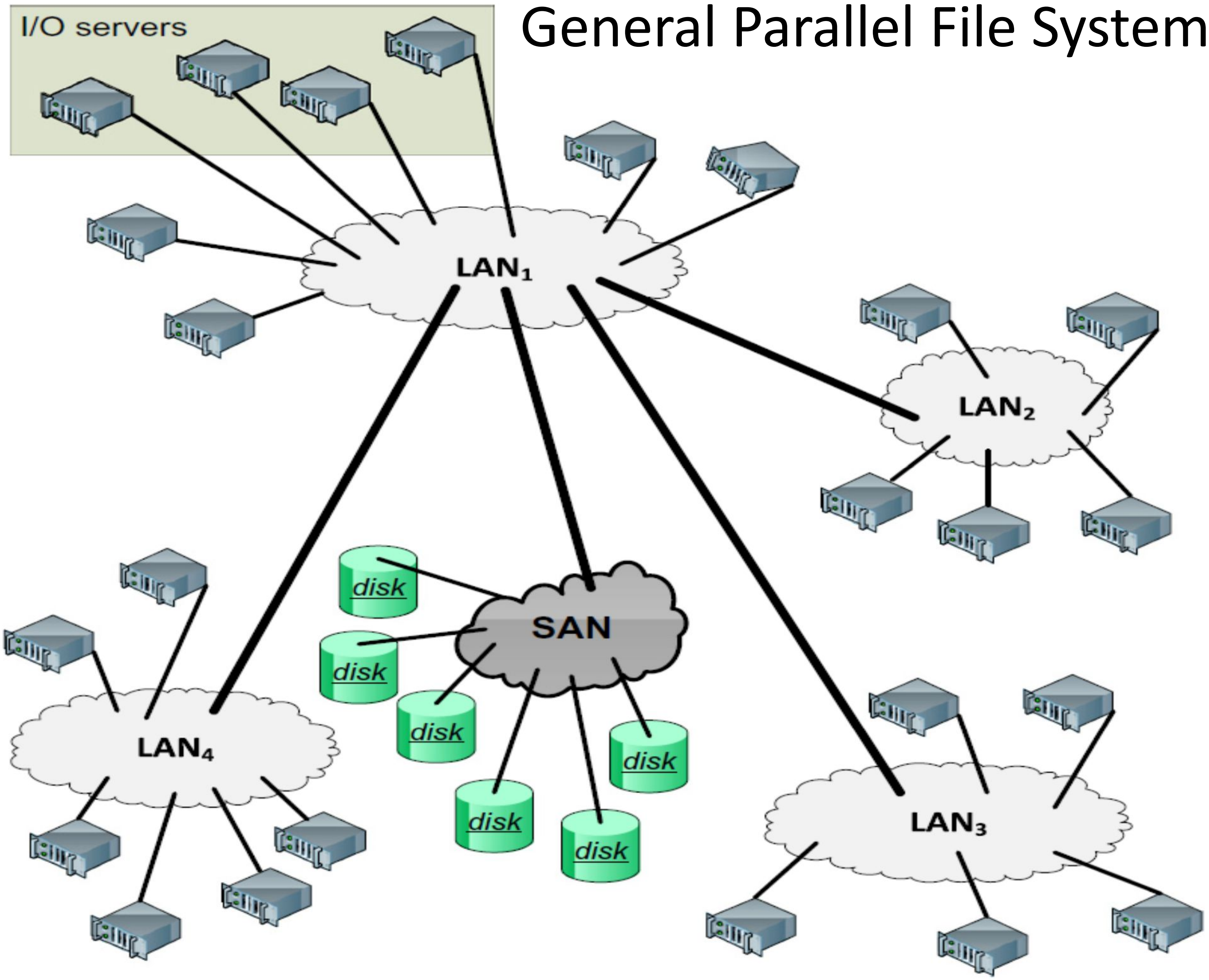
General Parallel File Systems (GPFS)

- The General Parallel File System (GPFS) was developed at IBM in the early 2000s as a successor to the TigerShark multimedia file 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 4096 disks of 1 TB each.

General Parallel File Systems (GPFS)

- Maximum file size is (2^{63} -1) bytes.
- A file consists of blocks of equal size, ranging from 16 KB to 1 MB, stripped across several disks.

General Parallel File System



General Parallel File Systems (GPFS)

- 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.

GPFS reliability

- 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. When a single disk fails, a large number of files are affected. (RAID concept and replication of meta data is adopted)

GPFS distributed locking

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

Lock granularity

- Has important implications on the performance. GPFS uses a variety of techniques for different types of data.
- Byte-range tokens ➔ 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; 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.
- Data-shipping ➔ allows fine-grain data sharing. In this mode the file blocks are controlled by the I/O nodes in a round-robin manner. A node forwards a read or write operation to the node controlling the target block, the only one allowed to access the file.

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.
- **Google File System** (**GFS** or **GoogleFS**) is a proprietary distributed file system developed by Google to provide efficient, reliable access to data using large clusters of commodity hardware.
- A new version of Google File System code named Colossus was released in 2010.

Google File System (GFS)

- The 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.
- GFS provides fault tolerance, reliability, scalability, availability and performance to large networks and connected nodes.
- GFS is made up of several storage systems built from low-cost commodity hardware components.
- **Google file system** is the name for Google's search data storage system.

Design considerations

- 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.

Design considerations

- 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.

Design decisions

1. Segment a file in large chunks.
2. Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file.
3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow. Schedule the high-bandwidth data flow by pipelining the data transfer over TCP connections to reduce the response time

Design decisions

4. Eliminate caching at the client site. Caching increases the overhead for maintaining consistency among cashed copies.
5. Ensure consistency by channeling critical file operations through a master, a component of the cluster which controls the entire system.
6. Minimize the involvement of the master in file access operations to avoid hot-spot contention and to ensure scalability.
7. Support efficient check-pointing and fast recovery mechanisms.
8. Support an efficient garbage collection mechanism.

GFS

- Large-scale distributed “filesystem”
- Master: responsible for metadata
- Chunk servers: responsible for reading and writing large chunks of data
- Chunks replicated on 3 machines, master responsible for ensuring replicas exist

GFS chunks

- GFS files are collections of fixed-size segments called chunks.
- At the time of file creation each chunk is assigned a unique *chunk handle*.
- The chunk size is 64 MB; this choice is motivated by the desire to optimize the performance.
- 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. The default RF (Replication Factor) is 3, user may change to any desired value.

The architecture of a GFS cluster

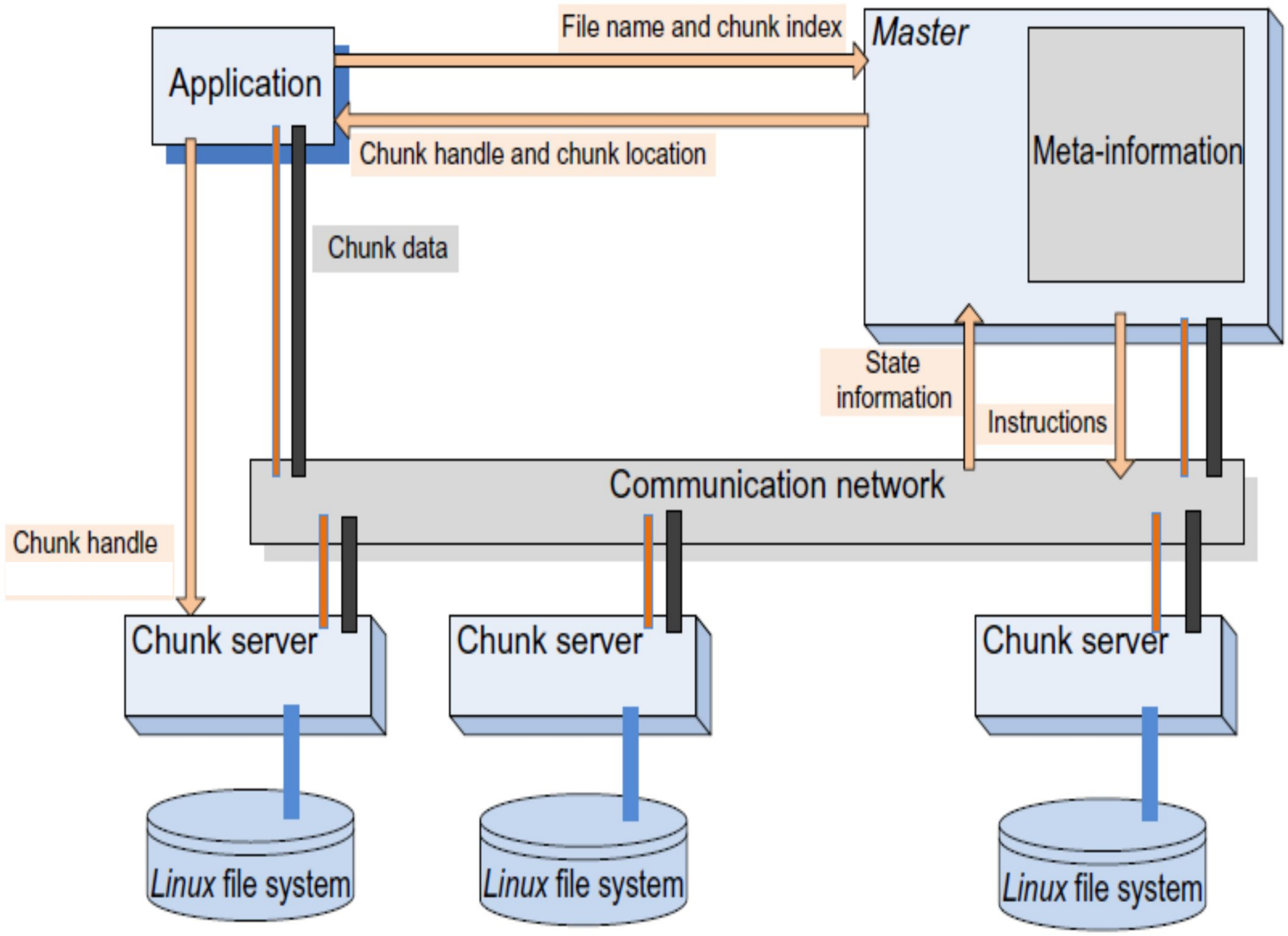
- A *master* controls a large number of *chunk servers*; it maintains metadata such as filenames, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers.
- The locations of the chunks are stored in the control structure of 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.

The architecture of a GFS cluster

- 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, changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage.
- To recover from a failure, the *master* replays the operation log.
- To minimize the recovery time, the *master* periodically checkpoints its state and at recovery time replays only the log records after the last checkpoint.

The architecture of a GFS cluster

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


File name and chunk index
Master
Application
Meta-information
Chunk handle and chunk location
Chunk data
State information
Instructions
Communication network
Chunk handle
Chunk server
Chunk server
Chunk server
Linux file system
Linux file system
Linux file system

The architecture of a GFS cluster

- Each chunk server is a commodity *Linux* system; it receives instructions from the *master* and responds with status information.
- To access a file, an application sends to the *master* the filename and the chunk index, the offset in the file for the read or write operation; the *master* responds with the chunk handle and the location of the chunk.
- Then the application communicates directly with the chunk server to carry out the desired file operation.
- *CloudStore* is an open-source C++ implementation of GFS that allows client access from C++ , Java and Python.

Apache Hadoop

- A wide range of data-intensive applications such as marketing analytics, image processing, machine learning, and Web crawling use *Apache Hadoop.*
- *Apache Hadoop*, an open-source, Java-based software system, supports distributed applications handling extremely large volumes of data.
- Hadoop is a complete eco-system of open source projects/technologies/ and tools that provide us the framework to deal with big data.

Apache Hadoop

- Hadoop is an open source framework from Apache and is used to store process and analyze data which are very huge in volume.
- Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware.
- It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

Apache Hadoop

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

Apache Hadoop

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

Apache Hadoop

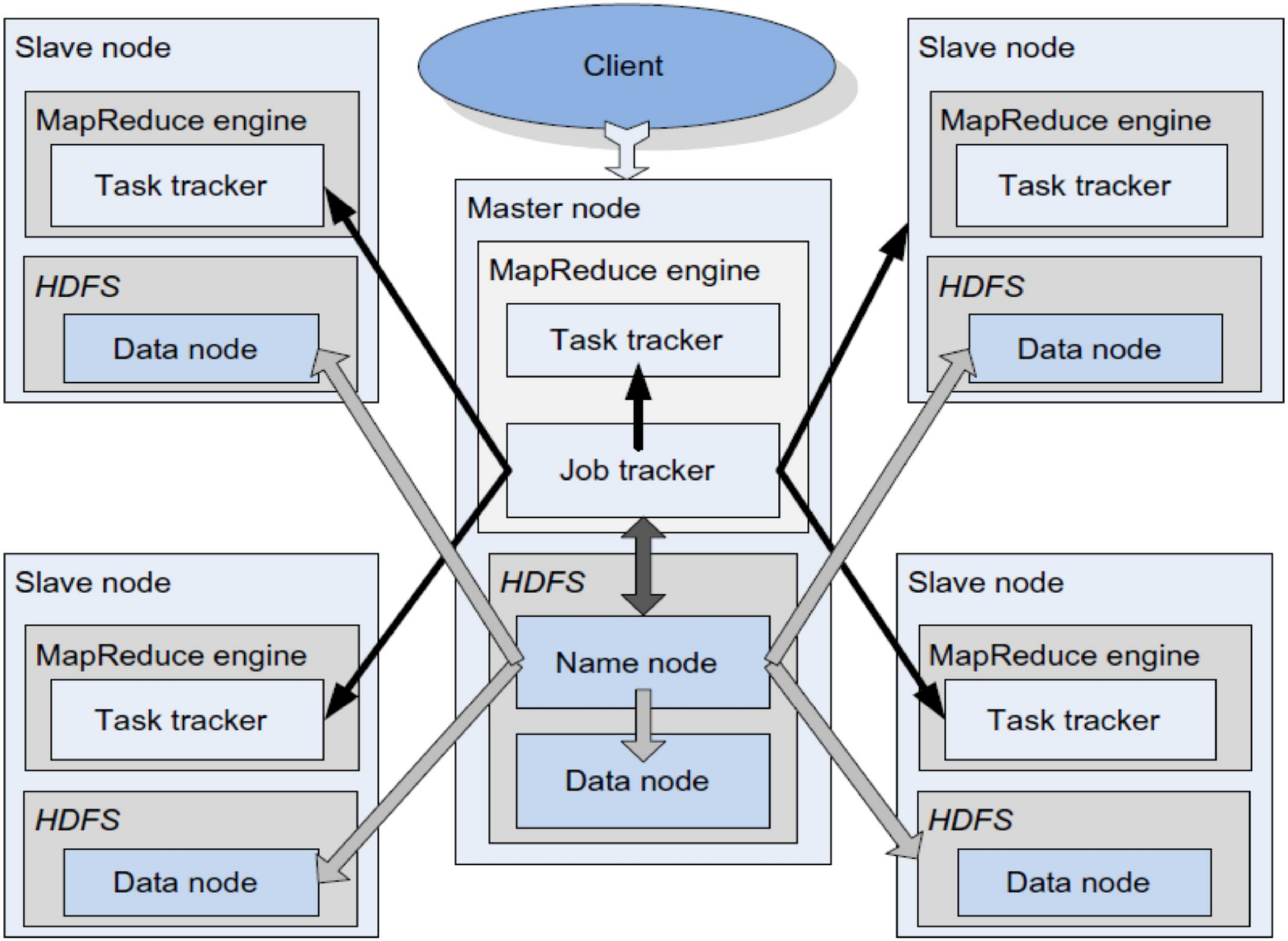
- The *Hadoop* engine on the master of a cluster consists of a *job tracker* and a *task tracker*, whereas the engine on a slave has only a *task tracker*.
- The job tracker receives a MapReduce job from a client and dispatches the work to the task trackers running on the nodes of a cluster.
- The *name node* running on the master manages the data distribution and data replication (default RF=3) and communicates with *data nodes* running on all cluster nodes.

Apache Hadoop

- The *job tracker* attempts to dispatch the tasks to available slaves which has the task data.
- The *task tracker* supervises the execution of the work allocated to the node.

A *Hadoop* cluster using *HDFS*

- The cluster includes a master and slave nodes. Each node runs a *MapReduce* engine and a database engine (*HDFS)*.
- The *job tracker* of the master’s engine communicates with the *task trackers* on all the nodes and with the *name node* of *HDFS*.
- A name node coordinates all the data nodes. It governs the distribution of data going to each machine.


Slave node
MapReduce engine
Task tracker
HDFS
Data node
Client
Master node
MapReduce engine
Task tracker
Job tracker
HDFS
Name node
Data node
Slave node
MapReduce engine
Task tracker
HDFS
Data node
Slave node
MapReduce engine
Task tracker
HDFS
Data node
Slave node
MapReduce engine
Task tracker
HDFS
Data node

Locks

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

Locks

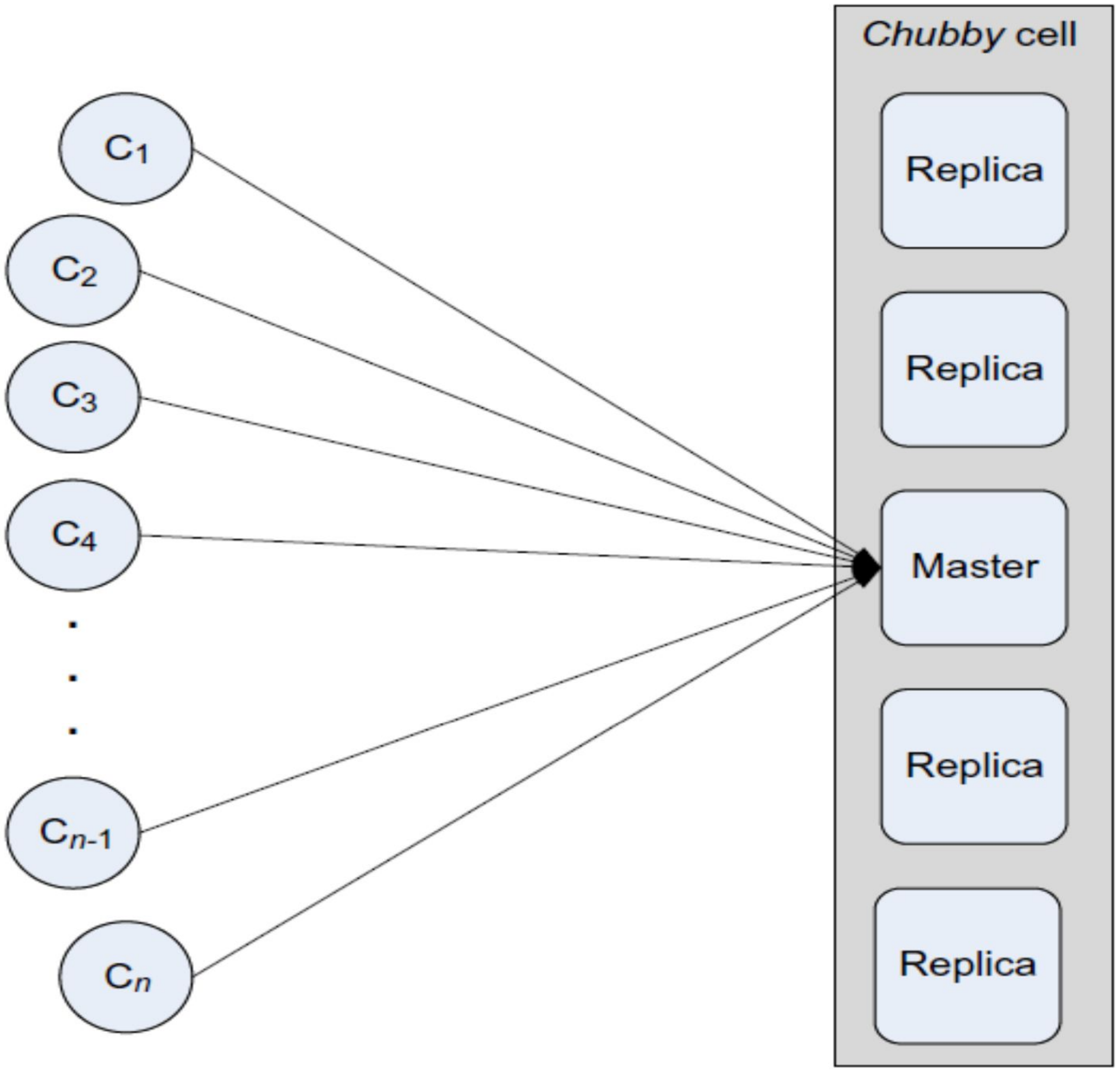
- Advisory locks → based on the assumption that all processes play by the rules; An *advisory lock* is like a stop sign; those who obey the traffic laws will stop.
- Mandatory locks→ block access to the locked objects to all processes that do not hold the locks.
- Fine-grained locks → locks that can be held for only a very short time
- Coarse-grained locks → locks held for a longer time.

Chubby lock service

- It is Google locking service.
- Chubby lock service is intended to provide coarse-grained locking as well as reliable storage for a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network.
- Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance

Chubby lock service

- The Google File System and Bigtable uses a Chubby lock service.



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

Chubby lock service

- A *Chubby* cell typically serves one data center. The cell server includes several *replicas*, the standard number of which is five.
- To reduce the probability of failures, the servers hosting replicas are distributed across the campus of a data center.
- The replicas use a distributed consensus protocol to elect a new *master* when the current one fails.

Chubby operation

- Clients use RPCs to request services from the master.
  - When it receives a write request, the master propagates the request to all replicas and waits for a reply from a majority of replicas before responding.
  - When it receives a read request, the master responds without consulting the replicas.
- The client interface of the system includes notification for events related to file or system status.

Chubby operation

- A client can subscribe to events such as: file contents modification, change or addition of a child node, master failure, lock acquired, conflicting lock requests, invalid file handle.
- Each file or directory can act as a lock. To write to a file the client must be the only one holding the file handle, while multiple clients may hold the file handle to read from the file.

Transaction processing

- Many cloud services are based on *online transaction processing* (OLTP) and operate under tight latency constraints.
- Major requirements:
  - Short response time: *memcaching* refers to a general-purpose distributed memory system that caches objects in main memory (RAM)
  - Scalability.
    - Vertical scaling
    - Horizontal scaling

Transaction processing

- The search for alternate models to store the data on a cloud is motivated by the needs of OLTP applications:
  - To decrease the latency by caching frequently used data in memory on dedicated servers.
  - To distribute the data on a large number of servers allows multiple transactions to occur at the same time and decreases the response time

NoSQL databases

- Companies heavily involved in cloud computing, such as Google and Amazon, e-commerce companies such as eBay, and social media networks such as Facebook, Twitter, or LinkedIn, discover that traditional relational databases are not able to handle the massive amount of data and the real-time demands of online applications that are critical for their business models.
- Solution is NoSQL.

NoSQL databases (not only relational)

- A **NoSQL** is a non-relational (No tables/schema less), open source distributed database used for dealing with big data (SSD, USD and SD) and real-time web applications.
- A **NoSQL** provides a mechanism for storage and retrieval of data that is not modeled in the tabular relations (RDBMS).
- NoSQL databases can be scaled horizontally across hundreds or thousands of servers.

NoSQL databases

- NoSQL databases are sometimes referred to as cloud databases, non-relational databases, or Big Data databases.
- NoSQL databases have become the first alternative to relational databases, with high performance, scalability (Horizontal), availability, and fault tolerance are being the key deciding factors.

key-value

Amazon DynamoDB (Beta)

ORACLE BERKELEY DB 11g

redis

graph

Neo4j the graph database

InfiniteGraph

sones

column

HBASE

riak

Cassandra

document

CouchDB relax

mongoDB

terrastore

BigTable

- BigTable is a distributed storage system that is structured as a large table: that may be petabytes in size and distributed among tens of thousands of machines.
- It is designed for storing items such as billions of URLs, with many versions per page; over 100 TB of satellite image data; hundreds of millions of users; and performing thousands of queries/second.

BigTable

- Bigtable is Google's NoSQL Big Data database service. It's the database that powers many core Google services, including Search, Google Analytics, Maps, Google Earth, Google finance, web crawlers and Gmail.
- *BigTable* is a distributed storage system to store massive amounts of data and to scale up to thousands of storage servers.
- BigTable is built on top of GFS (Google File System).

BigTable

- To guarantee atomic read and write operations, it uses the *Chubby* distributed lock service. the directories and the files of *Chubby* are used as locks.
- Bigtable is designed to handle massive workloads at **consistent low latency and high throughput**, so it is a great choice for both operational and analytical applications, including **IoT, user analytics, and financial data analysis**.

BigTable

- BigTable is designed with semi-structured data storage in mind. It is a large map that is indexed by a row key, column key, and a timestamp.
- Each value within the map is an array of bytes that is interpreted by the application. Every read or write of data to a row is atomic, regardless of how many different columns are read or written within that row.
- It is described as "a sparse, distributed, persistent multidimensional map."
- Bigtable is a sparsely populated table that can scale to billions of rows and thousands of columns, allowing to store terabytes or even petabytes of data.

BigTable Organization

- The system consists of three major components: client application 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.
- BigTable is a collection of (key, value) pairs where the key identifies a row and the value is the set of columns.

BigTable Organization

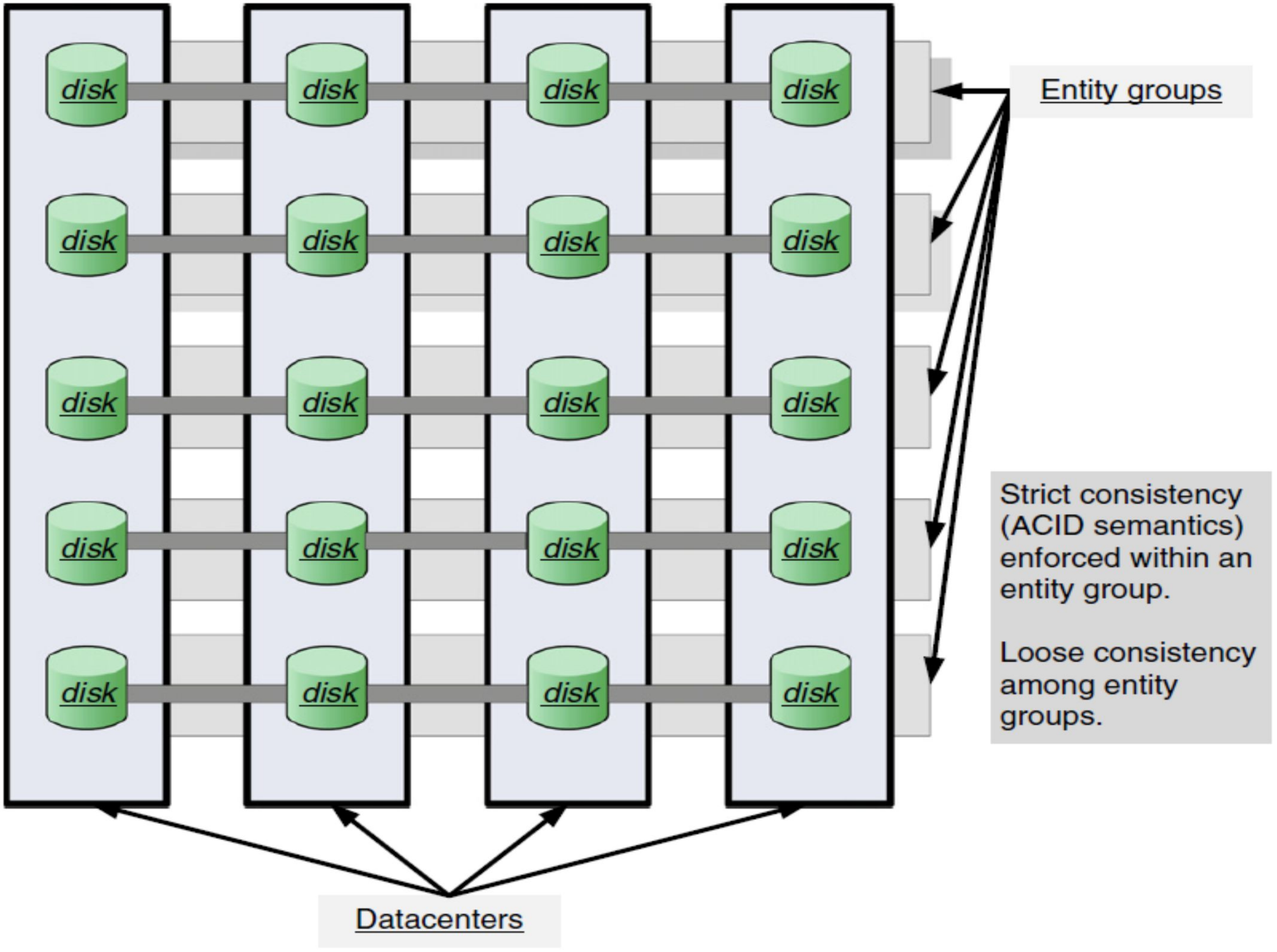
- The Apache open source version of BigTable, HBase, is built on top of HDFS (Hadoop Distributed File System) or Amazon S3.

Megastore

- *Megastore* is scalable storage for online services. The system, distributed over several data centers, has a very large capacity, 1 PB in 2011, and it is highly available.
- *Megastore* is widely used internally at Google; it handles some 23 billion transactions daily: 3 billion write and 20 billion read transactions.
- The basic design philosophy of the system is to partition the data into *entity groups* and replicate each partition independently in data centers located in different geographic areas.

Megastore data model

- Reflects a middle ground between traditional and NoSQL databases.
- The data model is declared in a schema consisting of a set of *tables*, composed of *entries*.


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

Megastore

- The system supports full ACID semantics within each partition and provides limited consistency guarantees across partitions.
- The system replicates primary user data, metadata, and system configuration information across data centers and for locking.
- The entity groups are application-specific and store together logically related data. For example, an email account could be an entity group for an email application

Megastore

- Data should be carefully partitioned to avoid excessive communication between entity groups. Sometimes it is desirable to form multiple entity groups, as in the case of blogs.
- In Megastore, read always returns the last fully updated version.
- The system makes extensive use of *BigTable*. Entities from different *Megastore* tables can be mapped to the same *BigTable* row without collisions.

A write transaction in ***Megastore***

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.

All the best for

your Exams