The Distributed File System (DFS)

The sharing of stored information is perhaps the most important aspect of distributed resource sharing

Introducing DFSs

- The design of large-scale wide-area read-write file storage systems poses problems of *load balancing*, *reliability*, *availability* and *security*
- DFSs emulate the functionality of a non-distributed file system for client programs running on multiple remote computers
- DFSs provide an essential underpinning for organizational computing based on intranets

Origins of File Systems

- Abstract away the details of physical disk storage
- Originally developed to provide a convenient programming interface to disk storage
- Access control and file-locking mechanisms are important features
- First file servers developed in 1970s
- First distributed file systems developed in the early 1980s (e.g., Sun's NFS)

Rationale

The concentration of persistent storage at a few servers reduces the need for local storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization

Example Storage Systems Overview

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore

Types of consistency:
1: strict one-copy. 3: slightly weaker guarantees. 2: considerably weaker guarantees.

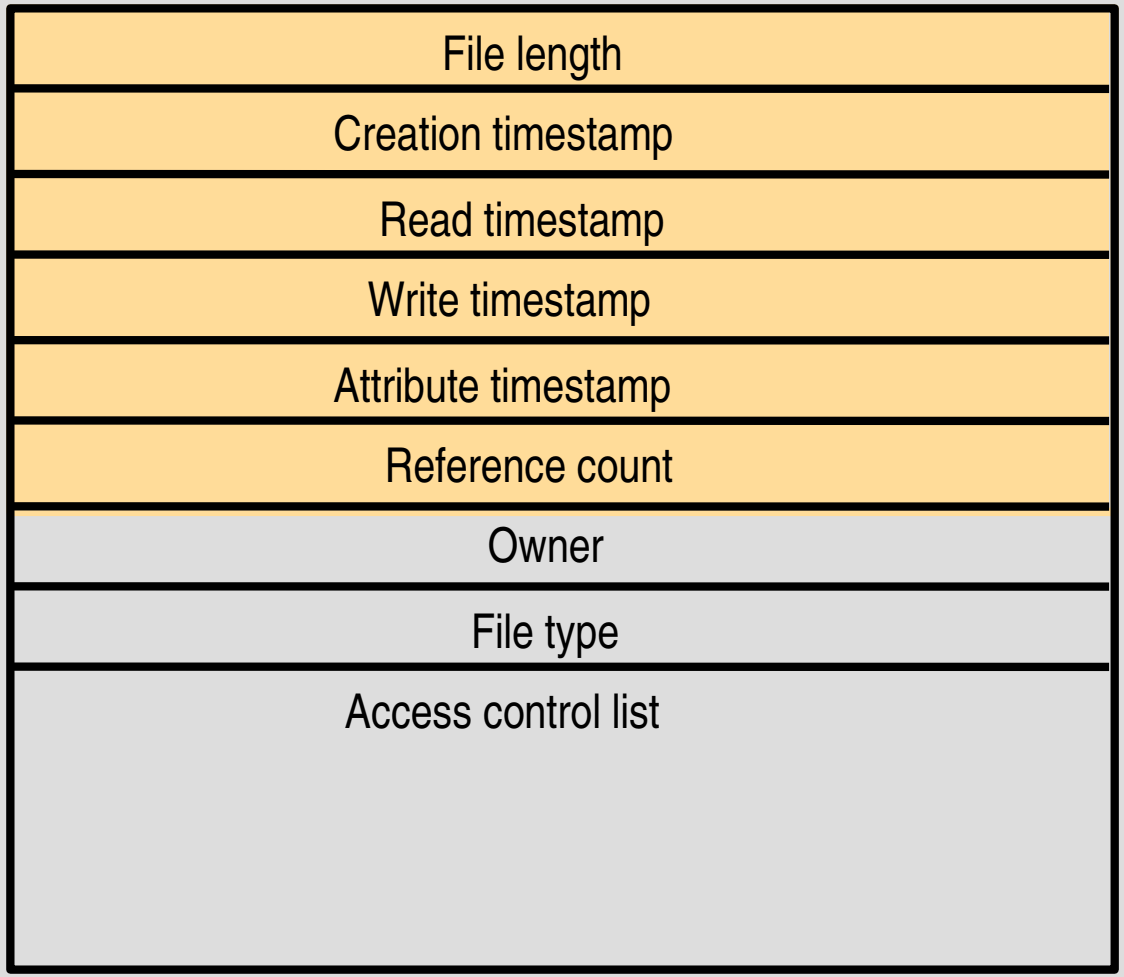
File System Layered Modules

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Characteristics of File Systems

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files (as far as the file system is concerned) contains both data and attributes
- File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files.

File Attribute Record Structure



More Characteristics

- File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested
- The term "meta-data" is often used to refer to all of the extra information stored by a file system that is needed for the management of files
- The file system is responsible for applying access control for files

File System Operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Key Point

The implementation of a distributed file service requires all of the components of a traditional file system, with additional components to deal with client-server communication and with the distributed naming and location of files

DFS Design Requirements

- Transparency
- Concurrent File Updates
- File Replication
- Hardware and OS Heterogeneity
- Fault Tolerance
- Consistency
- Security
- Efficiency

Transparency

- The design must balance the flexibility and scalability that derive from transparency against software complexity and performance

- **Access transparency** - clients need to be unaware of the distribution of files, so a program designed to work on a local file can work on a remote file without modification
- **Location transparency** - files or groups of files can be relocated without changing their actual path (or file) names

Transparency, continued

- **Mobility transparency** - a file may be moved, either by systems administrators or automatically
- **Performance transparency** - programs should continue to perform satisfactorily, even under heavy load
- **Scaling transparency** - file systems can be expanded by incremental growth to deal with a wide range of loads and network sizes

Concurrent File Updates

- Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file
- Most current file services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking

File Replication

- Several copies of a file's contents at different locations enable multiple servers to share the load of providing the service
- It enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed
- Few DFSs support replication fully, most support a limited form of replication and some form of caching

Hardware and OS Heterogeneity

- The service interfaces should be defined so that client and server software can be implemented for different OSes and computers
- This is an important aspect of *openness*

Fault Tolerance

- It is essential that the file service continue to operate in the face of client and server failures
- Some designs are based on *at-most-once invocation semantics*
- Others support *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files
- Another important design goal is *statelessness* - the file server can be restarted and the service restored after a failure without any need to recover "previous state"

Consistency

- Traditionally, file systems have offered *one-copy update semantics*
- **One-copy update semantics** - a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed
- When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics

Security

- In addition to access control lists, there is a need to authenticate client requests so that access control at the server is based on correct user identities
- There is also a need to protect the contents of request and reply messages with digital signatures and encryption of secret data

Efficiency

- A goal is to provide a comparable level of performance of at least the same power and generality as conventional file systems
- It must also be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently