Distributed File Systems

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network.

The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.

Introduction

Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet.

A well designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks.

Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects.
One way to achieve this is to serialize objects and to store and retrieve the serialized objects using files.
But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed.
Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

Figure 12.1 Storage systems and their properties

	Sharing	*Persistence*	*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 (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

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

Figure 12.1 provides an overview of types of storage system. In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores.

DSM provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores aim to provide persistence for distributed shared objects.

Examples include the CORBA Persistent State Service and persistent extensions to Java. Peer-to-peer storage systems offer scalability to support client loads much larger than the systems , but they incur high performance costs in providing secure access control and consistency between updatable replicas.

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur.
Virtually all storage systems rely on the use of caching to optimize the performance of programs.
Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a '1', for one-copy consistency in Figure 12.1) – programs cannot observe any discrepancies between cached copies and stored data after an update.
When distributed replicas are used, strict consistency is more difficult to achieve.
Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick ( ) in the consistency column of Figure 12.1.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations.

The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions.

Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date.

This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard.

Persistent object systems vary considerably in their approach to caching and consistency.

The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients.

Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.

They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout.
Files are stored on disks or other non-volatile storage media.
Files contain both *data* and *attributes*.

The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence.

The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists.

A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

File attribute record structure

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

Figure 12.4 UNIX 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)	Puts the file attributes for file *name* into *buffer*.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files.
The naming of files is supported by the use of directories.
A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers.
Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems.
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 (reading, updating, executing and so on).
The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files.
It includes file attributes, directories and all the other persistent information used by the file system

Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems.

Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development.

Transparency • The file service is usually the most heavily loaded service in an intranet,so its functionality and performance are critical. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service 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.

modern UNIX standards in providing **advisory** or **mandatory file- or record-level locking**

In Unix record locking is the technique used to lock the portion of a file for certain amount of time to maintain consistency of the data from concurrent access to the file.

On this mechanism, Mandatory locking is the technique which is used to lock the portion of the file exclusively. If once mandatory locking is enabled to the file, no other process can read or write the data to the locked portion of the file.

Advisory Locking

•Advisory locking is a file-locking mechanism in Linux that **relies on the cooperation of processes.**

•It allows processes to voluntarily **request locks** on files, indicating that the process will be reading or modifying the files.

•Advisory locking is useful in situations where processes are designed to cooperate.

•In advisory locking, there are two types of locks,

- Shared locks
- Exclusive locks

•A process can get either one of these locks. Multiple processes can acquire shared locks simultaneously, allowing them to read the file concurrently.

•However, only one process can hold an exclusive lock at a given time, ensuring exclusive write access to the file.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations.
This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures.
Fortunately, a moderately fault-tolerant design is straightforward for simple servers.
To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics ; or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files.

At most once: Messages are delivered once, and if there is a system failure, messages may be lost and are not redelivered.

At least once: This means messages are delivered one or more times. If there is a system failure, messages are never lost, but they may be delivered more than once.

Consistency • Conventional file systems such as that provided in UNIX offer *one-copy update semantics*.

This refers to 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

A model for concurrent access to files in which the processes accessing the file see its contents as if only one copy of it existed.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists.

In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data.

RSA idea is also used for signing and verifying a message it is called RSA digital signature scheme.

Digital signature scheme changes the role of the private and public keys

Private and public keys of only the sender are used not the receiver

Sender uses her own private key to sign the document and the receiver uses the sender's public key to verify it.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

The NFS protocol defines a network file system, originally developed for local file sharing among Unix systems and released by Sun Microsystems in 1984.

The NFS protocol specification was first published by the Internet Engineering Task Force (IETF) as an internet protocol in RFC 1094 in 1989.

The current version of the NFS protocol is documented in RFC 7530, which documents the NFS version 4 (NFSv4) Protocol.
NFS enables system administrators to share all or a portion of a file system on a networked server to make it accessible to remote computer users.

Clients with Authorization to access the shared file system can mount NFS shares, also known as shared file systems.

NFS uses Remote Procedure Calls (RPCs) to route requests between clients and servers.
Cloud vendors also implement the NFS protocol for cloud storage, including Amazon Elastic File System, NFS file shares in Microsoft Azure and Google Cloud Filestore.

NFS is an application layer protocol, meaning that it can operate over any transport or network protocol stack.
However, in most cases NFS is implemented on systems running the TCP/IP protocol suite. The original intention for NFS was to create a simple and stateless protocol for distributed file system sharing.

Early versions of NFS used the User Datagram Protocol (UDP) for its transport layer. This eliminated the need to define a stateful storage protocol; however, NFS now supports both the Transmission Control Protocol (TCP) and UDP. Support for TCP as a transport layer protocol was added to NFS version 3 (NFSv3) in 1995.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system [Morris *et al.* 1986].

The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions.

A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]

What is the difference between NFS and AFS in distributed system?

In other words, AFS has stateful servers, whereas NFS has stateless servers. Another difference between the two file systems is that AFS provides location independence (the physical storage location of the file can be changed, without having to change the path of the file, etc.)

File Service Architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*.

The relevant modules and their relationships are shown in Figure 12.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files.

The client module provides a single programming interface with operations on files similar to those found in conventional file systems.

The design is *open* in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

Flat file service • The flat file service is concerned with implementing operations on the contents of files.

Unique file identifiers (UFIDs) are used to refer to files in all requests for flat file service operations.

The division of responsibilities between the file service and the directory service is based upon the use of UFIDs.

UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system.

When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

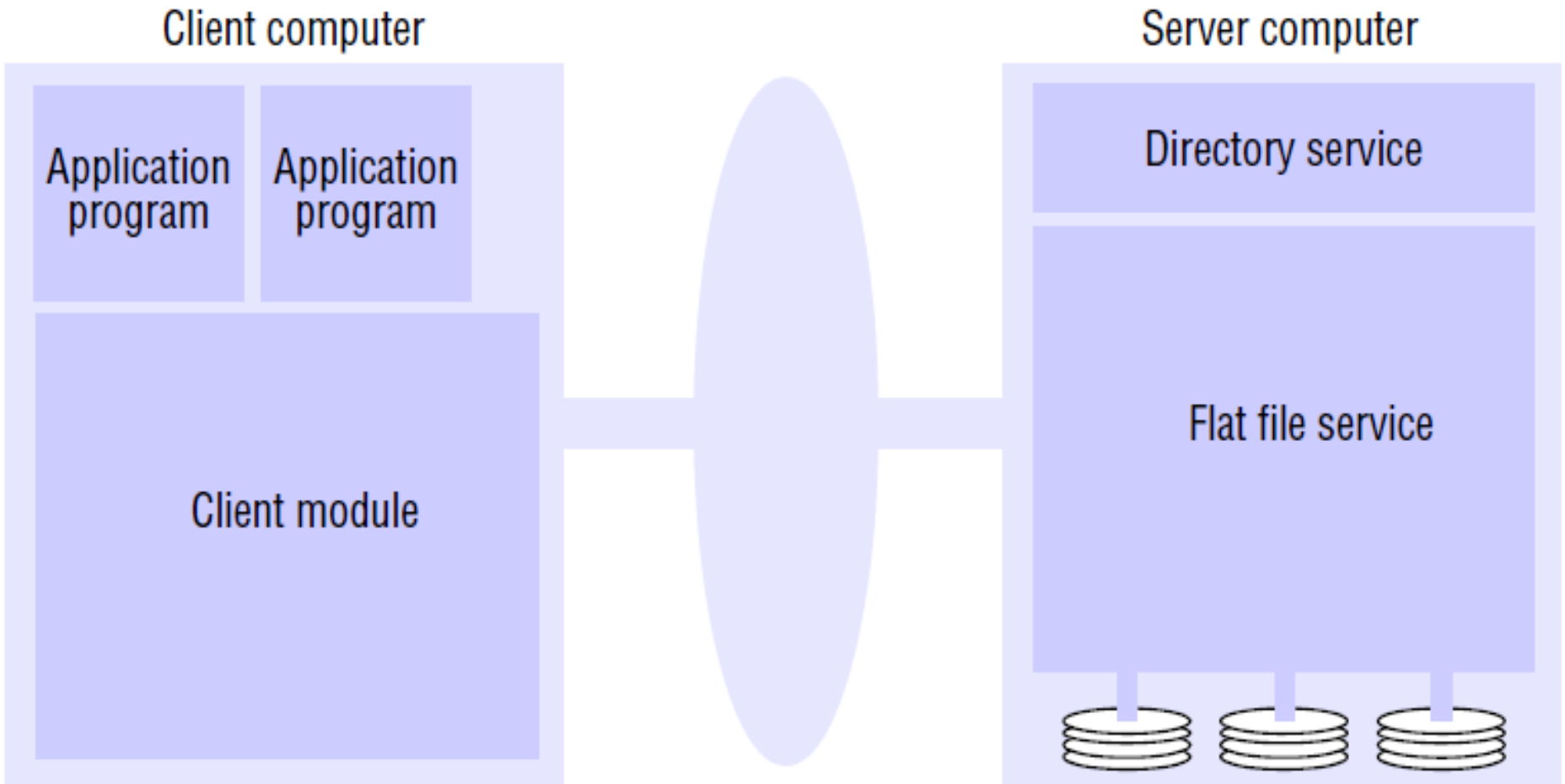
Directory service • The directory service provides a mapping between *text names* for files and their UFIDs.

Clients may obtain the UFID of a file by quoting its text name to the directory service.

The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories.

It is a client of the flat file service

- File service architecture



Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers.

For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service.

The client module also holds information about the network locations of the flat file server and directory server processes

In UNIX-like operating systems, **multi-part file names** refer to file names that contain multiple components separated by a delimiter, typically the forward slash ("/").

These components are often used to represent hierarchical directory structures, where each component represents a directory or subdirectory in the file system.

/home/user/documents/report.txt

In this example: / is the delimiter used to separate components.

home, user, and documents are directory names. report.txt is the file name.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules.

It is not normally used directly by user-level programs.

A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested.

Figure 12.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

Access control • In the UNIX file system, the user's access rights are checked against the access *mode* (read or write) requested in the *open* call (Figure 12.4 shows the UNIX file system API) and the file is opened only if the user has the necessary rights.

The user identity (UID) used in the access rights check is retrieved during the user's earlier authenticated login and cannot be tampered with in non-distributed implementations.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.

A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

• An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability, which is returned to the client for submission with subsequent requests.

• A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a directory service.

The primary purpose of the directory service is to provide a service for translating text names to UFIDs.

In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs.

Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter).

Figure 12.7 Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.

Each directory holds the names of the files and other directories that are accessible from it.

Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through the tree.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined.

A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree.

The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

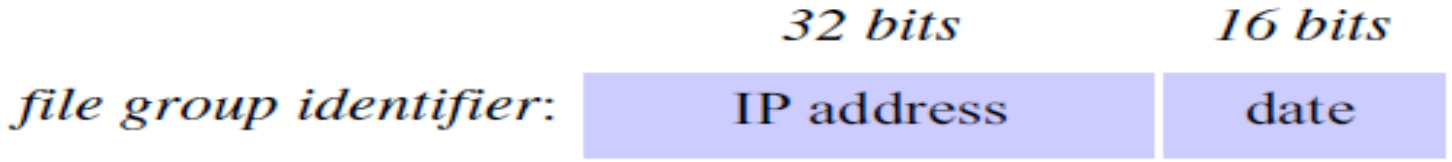
In the context of file systems, a reference count is a file attribute that tracks the number of references or pointers to a particular file or directory. Each time a file or directory is referenced, either by creating a hard link to it or by opening it through a file handle, the reference count for that file or directory is incremented. Similarly, when a reference to the file or directory is removed, such as by deleting a hard link or closing a file handle, the reference count is decremented.

A hard link is a directory entry that points directly to the data blocks of a file.
All hard links to the same file share the same inode (data structure representing the file), which means they refer to the same file data on disk.
Deleting any hard link to a file does not remove the file's contents from disk until all hard links to the file are removed.

File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs.
A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.)

In a distributed system, a filegroup typically refers to a logical grouping of files or data objects that are distributed across multiple nodes or storage devices.

In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers.
In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.
File group identifiers must be unique throughout a distributed system.
Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

Sun Network File System

All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems.

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls.

It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both.

A port mapper service is included to enable clients to bind to services in a given host by name.

The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon.

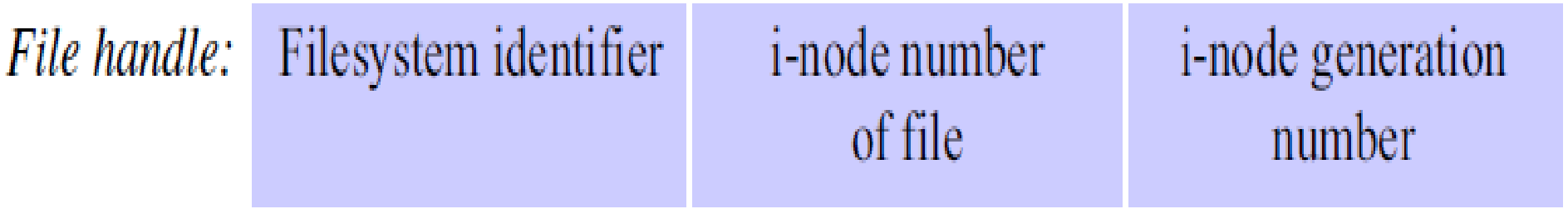
Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file system

The file identifiers used in NFS are called *file handles*.

A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file.

In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):



The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system).
The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed.
In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused.
The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file.
A VFS structure relates a remote file system to the local directory on which it is mounted.
The v-node contains an indicator to show whether a file is local or remote.
If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation).
If the file is remote, it contains the file handle of the remote file

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs.

It is integrated with the kernel and not supplied as a library for loading into client processes so that:

• user programs can access files via UNIX system calls without recompilation or reloading;

• a single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);

the encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

The NFS client module cooperates with the virtual file system in each client machine.

It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible.

It shares the same buffer cache that is used by the local input-output system.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients.
So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested.
The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes.

NFS Server Interface

The file and directory operations are integrated in a single service; the creation and insertion of file names in directories is performed by a single *create* operation, which takes the text name of the new file and the file handle for the target directory as arguments.

.

create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.

The other NFS operations on directories are *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*.
They resemble their UNIX counterparts with the exception of *readdir*, which provides a representation independent method for reading the contents of directories, and *statfs,* which gives the status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer.
On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting.
An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem.
Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted.

sudo mount -t nfs 192.168.1.100:/mnt/data /mnt/nfs_data
In this example:
192.168.1.100 is the IP address of the NFS server.
/mnt/data is the path to the shared directory on the NFS server.
/mnt/nfs_data is the directory where the NFS share will be mounted locally.
Make sure you have appropriate permissions and network connectivity to access the NFS server and its shared directory.

The /etc/exports file on the NFS server specifies which directories are exported (shared) to NFS clients and defines the access permissions for those directories. Each line in the /etc/exports file represents one export entry.

Here's an example of how the corresponding entry in the **/etc/exports** file might look for the NFS share we mounted in the previous example:
/mnt/data 192.168.1.0/24(rw,sync,no_subtree_check)

In this example:

/mnt/data is the path to the directory being exported.
192.168.1.0/24 is the network or IP range allowed to access the exported directory. Replace this with the appropriate network or IP range for your environment.
(rw,sync,no_subtree_check) are the export options:
rw: Allows clients to read from and write to the exported directory.
sync: Ensures that changes to files are immediately written to disk before the operation is considered complete. This ensures data consistency but may impact performance.
no_subtree_check: Disables subtree checking, which improves performance for nested directories but may reduce security in some cases.
After modifying the /etc/exports file, you'll need to reload the NFS server configuration to apply the changes. You can do this by running the following command:
sudo exportfs -ra

Securing NFS with Kerberos • The Kerberos authentication system developed at MIT, which has become an industry standard for securing intranet servers against unauthorized access and imposter attacks. The security of NFS implementations has been strengthened by the use of the Kerberos scheme to authenticate clients.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

A timestamp-based method is used to validate cached blocks before they are used. Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.

A cache entry is valid at time T if $T - Tc$ is less than a freshness interval t, or if the value for Tm recorded at the client matches the value of Tm at the server (that is, the data has not been modified at the server since the cache entry was made). Formally, the validity condition is:

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

The selection of a value for t involves a compromise between consistency and efficiency.

Server Caching

NFS servers use the cache at the server machine just as it is used for other file accesses. The use of the server's cache to hold recently read disk blocks does not raise any consistency problems; but when a server performs write operations, extra measures are needed to ensure that clients can be confident that the results of the write operations are persistent, even when server crashes occur.

In version 3 of the NFS protocol, the *write* operation offers two options for this (not shown in Figure 12.9):

1. Data in *write* operations received from clients is stored in the memory cache at the server *and* written to disk before a reply is sent to the client. This is called *writethrough* caching. The client can be sure that the data is stored persistently as soon as the reply has been received.
2. Data in *write* operations is stored only in the memory cache. It will be written to disk when a *commit* operation is received for the relevant file. The client can be sure that the data is persistently stored only when a reply to a *commit* operation for the relevant file has been received.