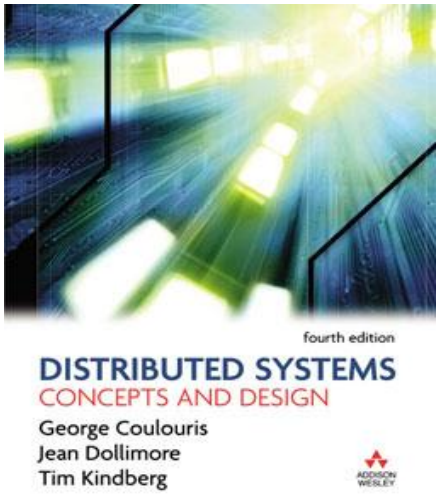
Slides for Chapter 8: Distributed File Systems



From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

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.

Introduction

A key goal of distributed systems – sharing of resources

The requirements for sharing within local networks and intranets lead to a need for a different type of service

This chapter describes the architecture and implementation of basic distributed file systems.

File system – an OS facility which provides a convenient programming interface to disk storage

Introduction

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects.

Figure 8.1
Storage systems and their properties

	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 (DSM, Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

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

Introduction

Distributed Shared Memory (DSM) provides an emulation of a shared memory by the replication of memory pages or segments at each host.

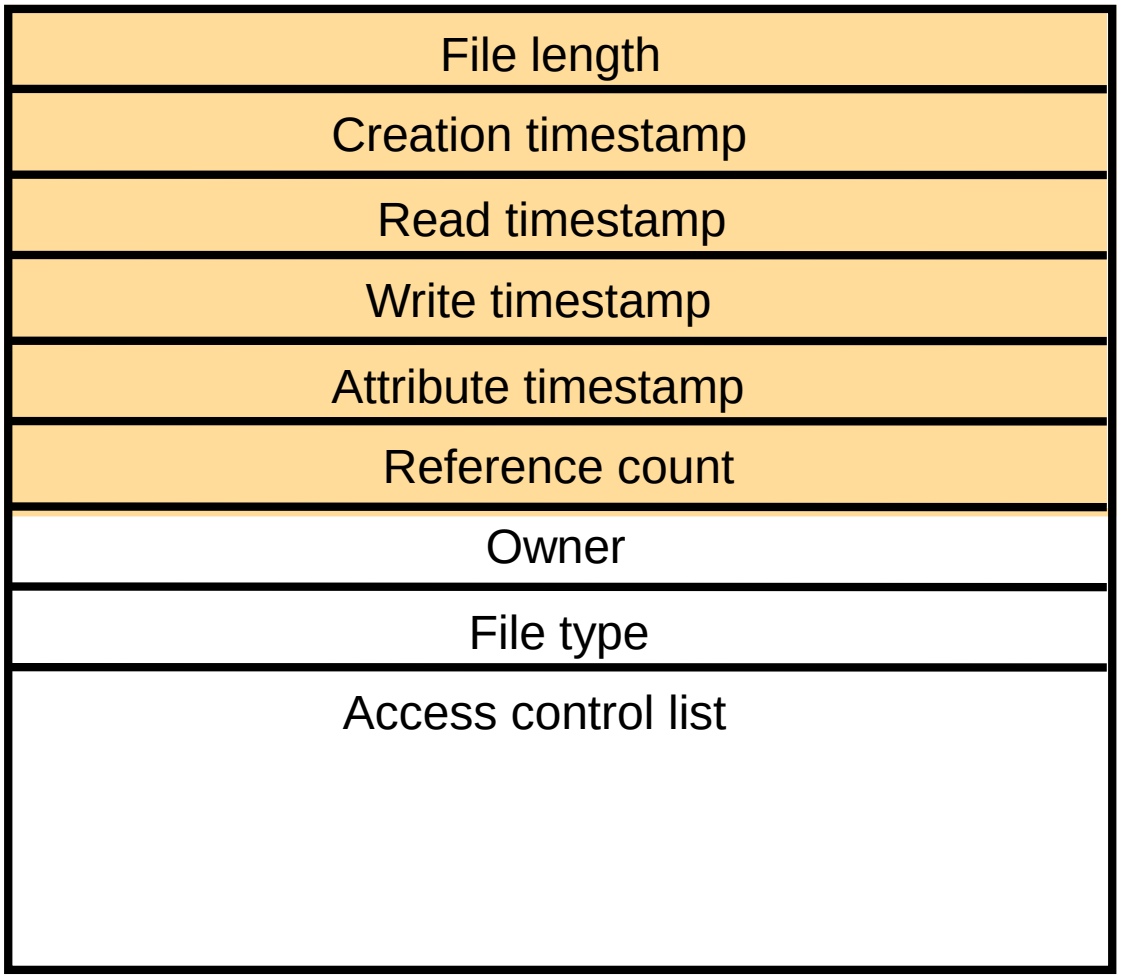
Peer-to-peer storage systems offer scalability to support client loads but they incur high performance costs in providing secure access control and consistency between updatable replicas.

Characteristics of file systems

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

Files contain both data and attributes.

Figure 8.3
File attribute record structure



Characteristics of file systems

File systems are designed to store and manage large numbers of files with facilities for creating, naming and deleting files.
- directory , . . .

Metadata - refers to all of the extra information stored by a file system that is needed for the management of files.

Figure 8.2
File system 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 system operations

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program.

Tile file system is responsible for applying access control for files.

Figure 8.4
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)	Gets the file attributes for file *name* into *buffer*.

Distributed file system requirements

Transparency
- Access transparency
- Location transparency
- Mobility transparency
- Performance transparency
- Scaling transparency

Concurrent file updates

File replication

Hardware and operating system heterogeneity

Distributed file system requirements

Fault tolerance

Consistency - one-copy update semantics

- When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications

Security

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

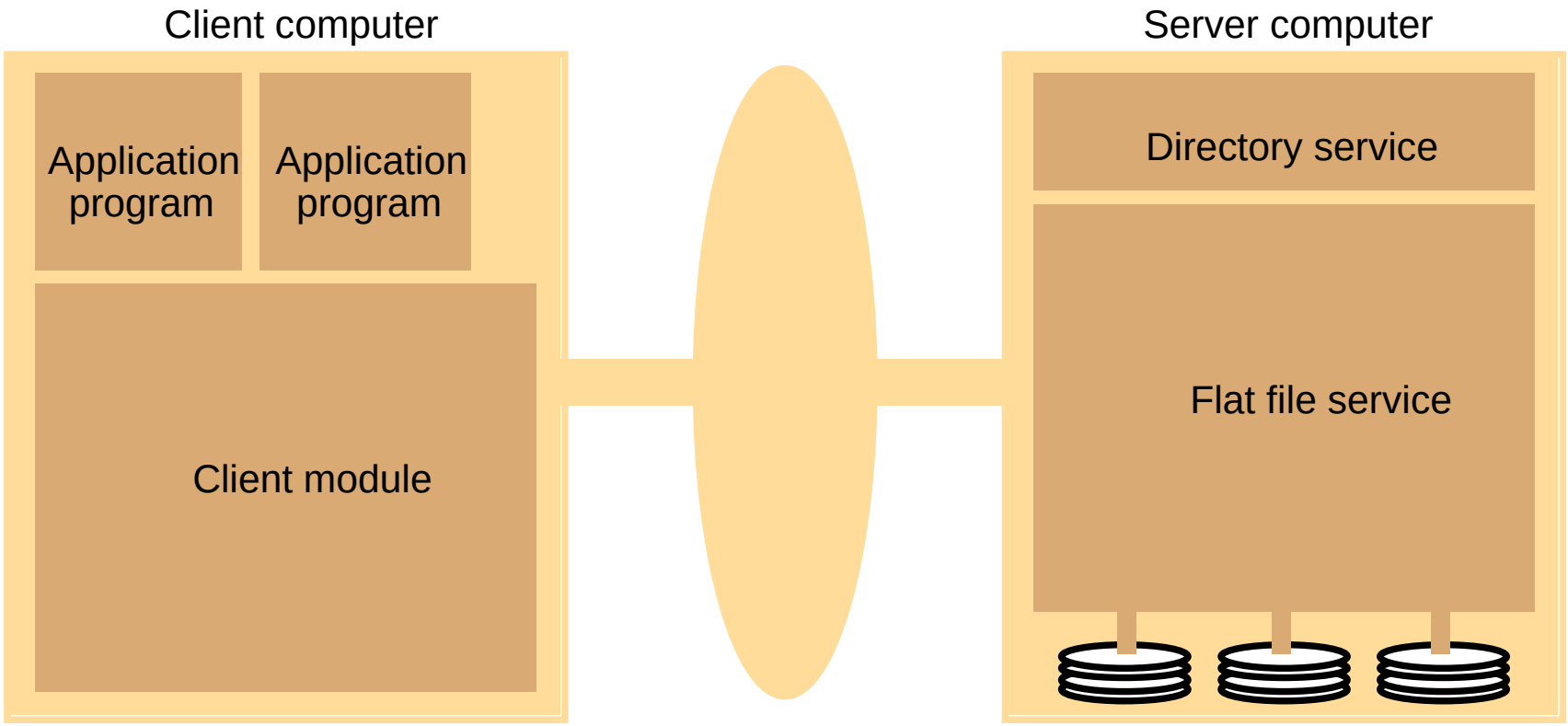
- should achieve a comparable level of performance.

File service architecture

The file service can be structured as 3 components

- a flat file service
- a directory service
- a client module.

Figure 8.5
File service architecture



File service architecture

Flat file service

- concerned with implementing operations on the contents of files
- Unique file identifiers (UFIDs)

Directory service

- provides a mapping between text names for files and their UFIDs
- provides functions needed to generate directories, 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

- runs in each client computer integrating
- extends the operations of the flat file service and the directory service under a single application programming interface
- holds information about the network locations of the flat file server and directory server processes
- caching

File service architecture

Flat file service interface

- the RPC interface used by client modules

Figure 8.6
Flat file service operations

Read(FileId, i, n) -> Data — throws *BadPosition*	If $1 \leq i \leq Length(File)$: 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 $1 \leq i \leq Length(File)+1$: 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 8.3).

File service architecture

Flat file service interface

- comparison with the UNIX interface

- differs for reasons of fault tolerance
  - Repeatable operations
  - Stateless servers

File service architecture

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
- In distributed implementations, access rights checks have to be performed at the server
- 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 ...

File service architecture

Directory service interface

- Its primary purpose - a service for translating text names to UFIDs

Figure 8.7
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: the entry containing *Name* is removed 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*.

File service architecture

Hierarchic file system

- Unix, pathname
- A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services
- A function can be provided in the client module that gets the UFID of a file given its pathname

File service architecture

File groups

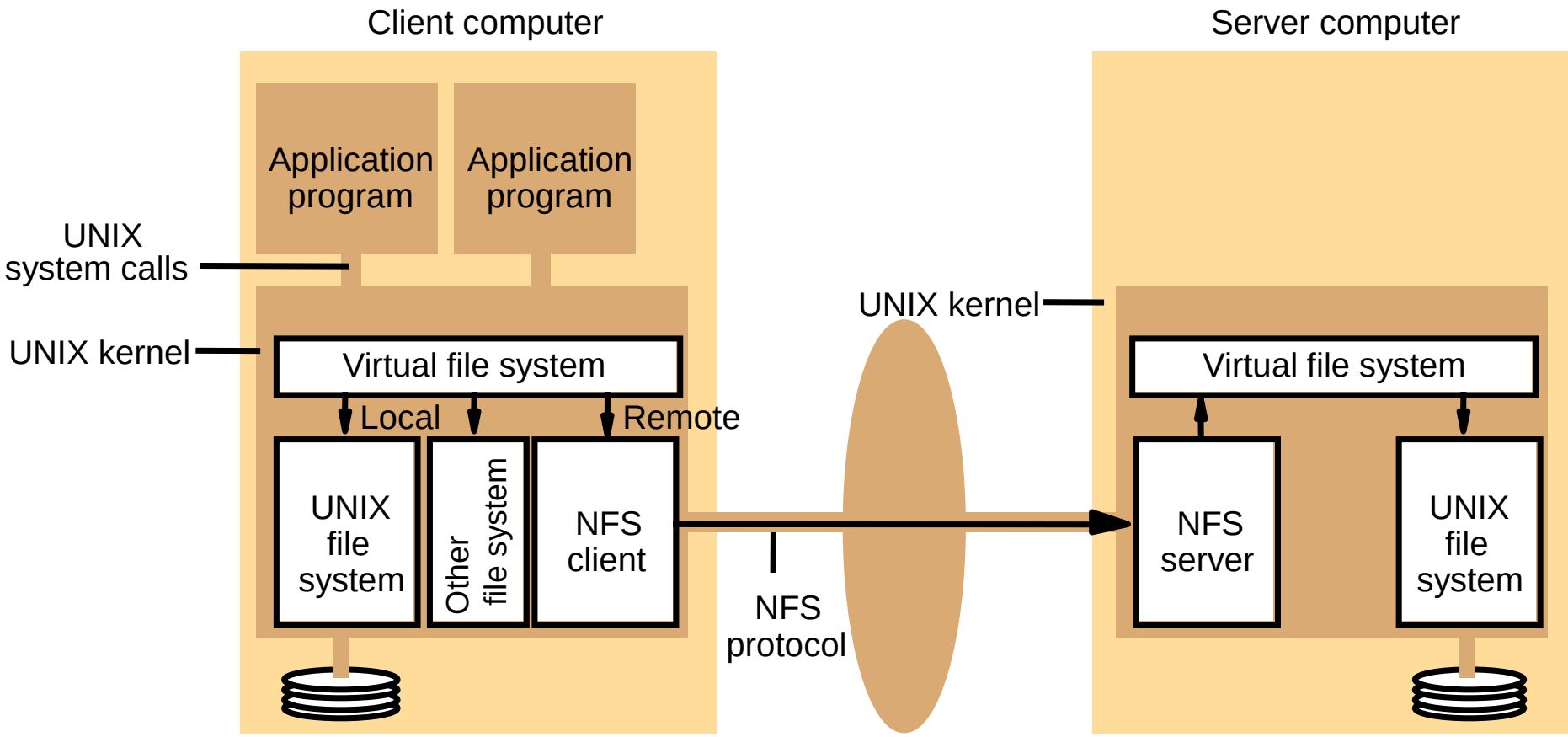
- a collection of files located on a given server
- a server may hold several file groups
- groups can be moved between servers
- In a distributed file service, file groups support the allocation of files to file servers in larger logical units
- the representation of UFIDs includes a file group identifier component
- File group identifiers must be unique throughout a distributed system

Case study: Sun Network File System

All implementations of NFS support the NFS protocol

The NFS server module resides in the kernel on each computer that acts as an NFS server

Figure 8.8
NFS architecture



Case study: Sun Network File System

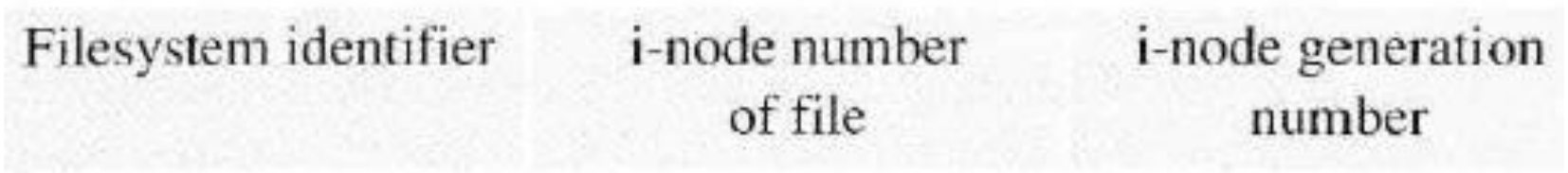
NFS provides access transparency

Virtual file system

- distinguish between local and remote files
- to translate between file identifiers
- The file identifiers used in NFS are called file handles

Case study: Sun Network File System

Virtual file system: File handle



The virtual file system layer has

- one VFS structure for each mounted file system
- one v-node per open file.

Case study: Sun Network File System

Client integration

- The NFS client module is integrated with the kernel and not supplied as a library for loading into client processes

Case study: Sun Network File System

Access control and authentication

- Unlike the UNIX file system, the NFS server is stateless
- does not keep files open on behalf of its clients

Case study: Sun Network File System

NFS server interface

- A simplified representation of the RPC interface provided by NFS Version 3 servers
- The file and directory operations are integrated in a single service

Figure 8.9
NFS server operations (simplified) – 1

lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
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.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

Continues on next slide ...

Figure 8.9
NFS server operations (simplified) – 2

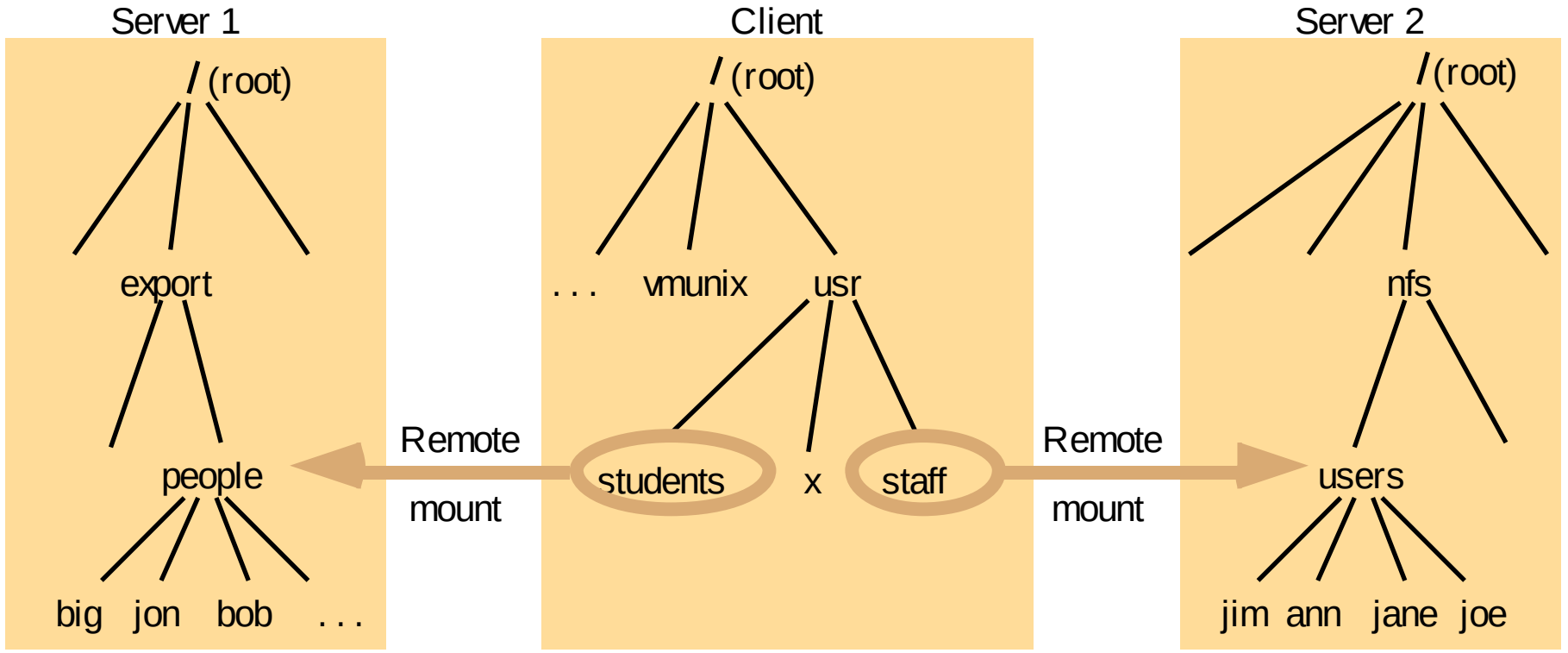
symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Case study: Sun Network File System

Mount service

- The mounting of sub-trees of remote filesystems by clients
- a separate mount service process that runs on each NFS server computer
- Clients use a mount command to request mounting of a remote filesystem
- mount protocol

Figure 8.10
Local and remote file systems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Case study: Sun Network File System

Mount service

- Remote filesystems may be hard-mounted or soft-mounted in a client computer

Case study: Sun Network File System

Path name translation

- UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process
- In NFS, pathnames cannot be translated at a server

Case study: Sun Network File System

Automounter

- mounts a remote directory dynamically whenever an ‘empty’ mount point is referenced by a client
- maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each

Case study: Sun Network File System

Server caching - In UNIX systems

- file pages, directories and file attributes that have been read from disk are retained in a main memory buffer cache until the buffer space is required for other pages
- Read-ahead
- Delayed-write
- the UNIX *sync* operation

Case study: Sun Network File System

Server caching - NFS servers

- use the cache at the server machine just as it is used for other file accesses
- recently read disk blocks
- when a server performs write operations, extra measures are needed
  - write-through caching
  - commit operation

Case study: Sun Network File System

Client caching

- The NFS client module caches the results of read, write, getattr, lookup and readdir operations
- A timestamp-based method is used to validate cached blocks before they are used

Tc - time when the cache entry was last validated
Tm - time when the block was last modified at the server
t - freshness interval

Case study: Sun Network File System

Client caching

- A cache entry is valid if
$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$

- The selection of a value for t is a compromise between consistency and efficiency
- NFS clients cannot determine whether a file is being shared or not

Case study: Sun Network File System

Client caching - Several measures are used to reduce the traffic of getattr calls to the server:

- Whenever a new value of Tm_{server} is received at the client
- The current attribute values are sent 'piggybacked' with the results of every operation on a file
- The adaptive algorithm for setting freshness interval

Case study: Sun Network File System

Securing NFS with Kerberos

- In the original implementation, the user's identity is included in each request in the form of an unencrypted numeric identifier
- NFS server is supplied with full Kerberos authentication data for the user when their home and root filesystems are mounted

Case study: Sun Network File System

Performance - Early performance figures showed two remaining problem areas

- frequent use of the *getattr* call
- relatively poor performance of the write operation