CSD511 – Distributed Systems
分散式系統

Chapter 8
Distributed File Systems

吳俊興
國立高雄大學 資訊工程學系

Chapter 8 Distributed File Systems

8.1 Introduction

8.2 File service architecture

8.3 Case study: Sun Network File System

8.4 Case study: The Andrew File System

8.5 Enhancements and further developments

8.6 Summary

1.1 Introduction

- Sharing of resources is a key goal for distributed systems
  - printers, storages, network bandwidths, memories, …
- Mechanisms for data sharing
  - Web servers
  - P2P file sharing
  - Distributed storage systems
    - Distributed file systems
    - Distributed object systems
- Goal of distributed file service
  - Enable programs to store and access remote files exactly as they do local ones

Figure 8.1 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. 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..✓. slightly weaker guarantees. 2: considerably weaker guarantees.

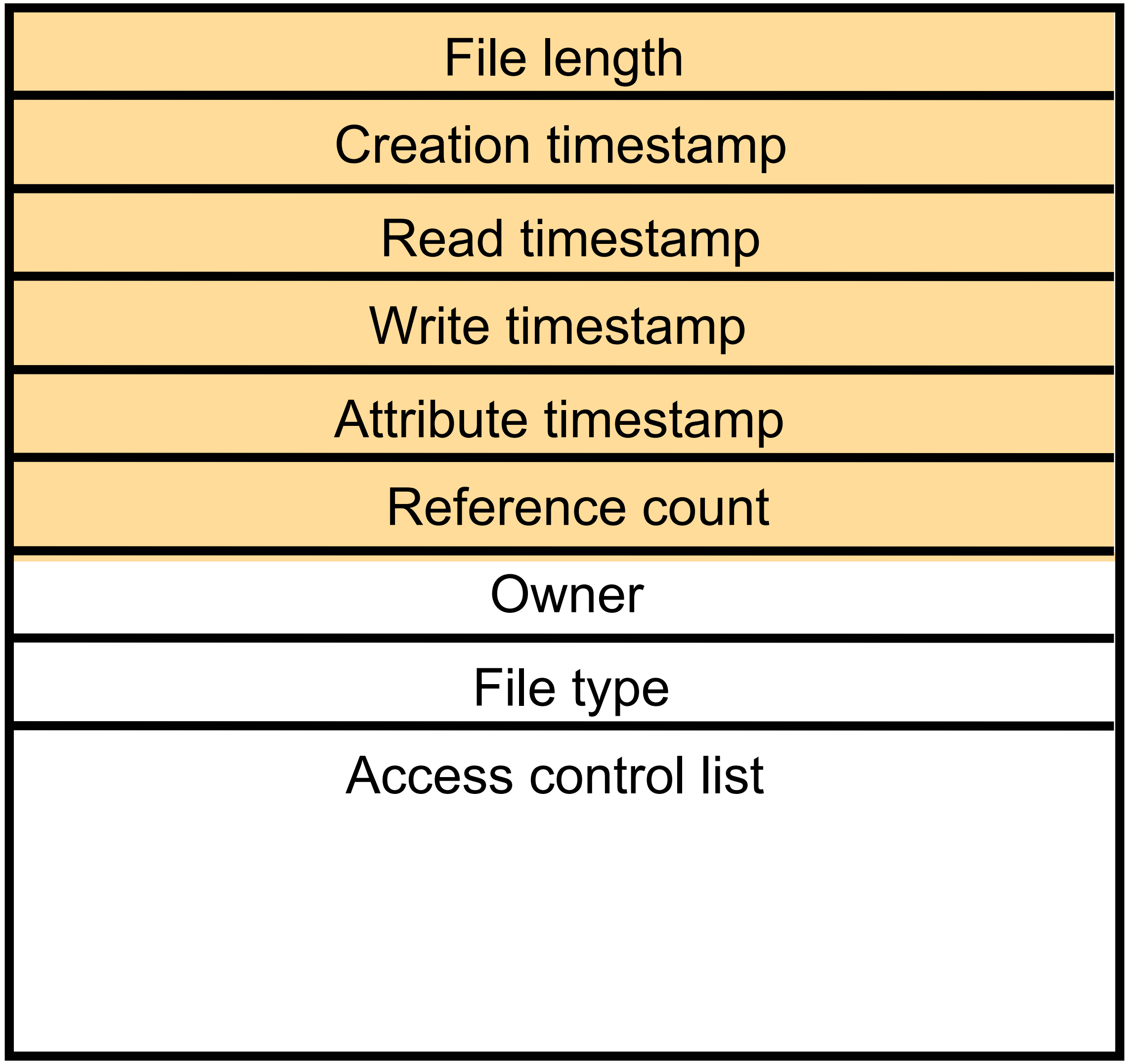
8.1.1 Characteristics of file systems

- File system: responsible for organization, storage, retrieval, naming, sharing and protection of files
  - file: containing data and attributes (Fig 8.3)
  - directory: mapping from text names to internal file identifiers
  - file operation: system calls in UNIX (Fig. 8.4)

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

Figure 8.2 File system modules

Figure 8.3 File attribute record structure



File length

Creation timestamp

Read timestamp

Write timestamp

Attribute timestamp

Reference count

Owner

File type

Access control list

Figure 8.4 UNIX file system operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	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)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	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*.

8.1.2 Distributed file system requirements

- Transparency:
  - access, location, mobility, performance, scaling
- 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
  - common services: advisory or mandatory file- or record-level locking
- File replication
  - A file may be represented by several copies of its contents at different locations
  - advantages: load sharing and fault tolerance
- Hardware and operating system heterogeneity
- Fault tolerance
  - The service continue to operate in the face of client and server failures
- Consistency
  - An inevitable delay in the propagation of modifications to all sites
- Security
- Efficiency

8.1.3 Case studies

- Sun NFS (Network File System)
  - introduced in 1985
  - the first file service designed as a product
  - RFC1813: NFS protocol version 3
  - Each computer can act as both a client and a server
- Andrew File System (AFS)
  - developed at CMU for use as a campus computing and information system
  - 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

8.2 File service architecture

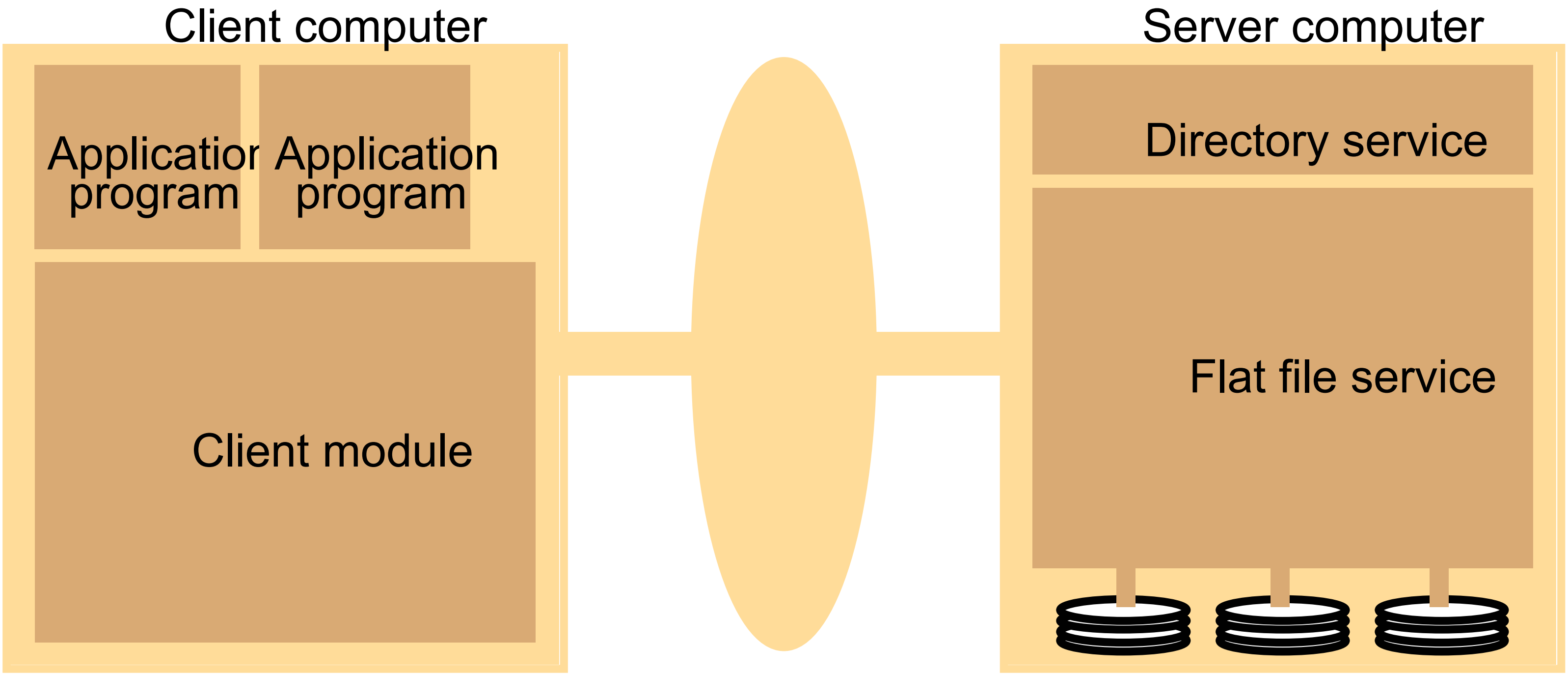


Figure 8.5 File service architecture (Author's abstract model)

- Stateless file service architecture
  - Flat file service: unique file identifiers (UFID)
  - Directory service: map names to UFIDs
  - Client module
    - integrate/extend flat file and directory services
    - provide a common application programming interface (can emulate different file interfaces)
    - stores location of flat file and directory services

Flat file service interface

- RPC used by client modules, not by user-level programs
- Compared to UNIX
  - no open/close
    - Create is not idempotent
    - at-least-once semantics
    - reexecution gets a new file
  - specify starting location in Read/Write
    - stateless server

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).

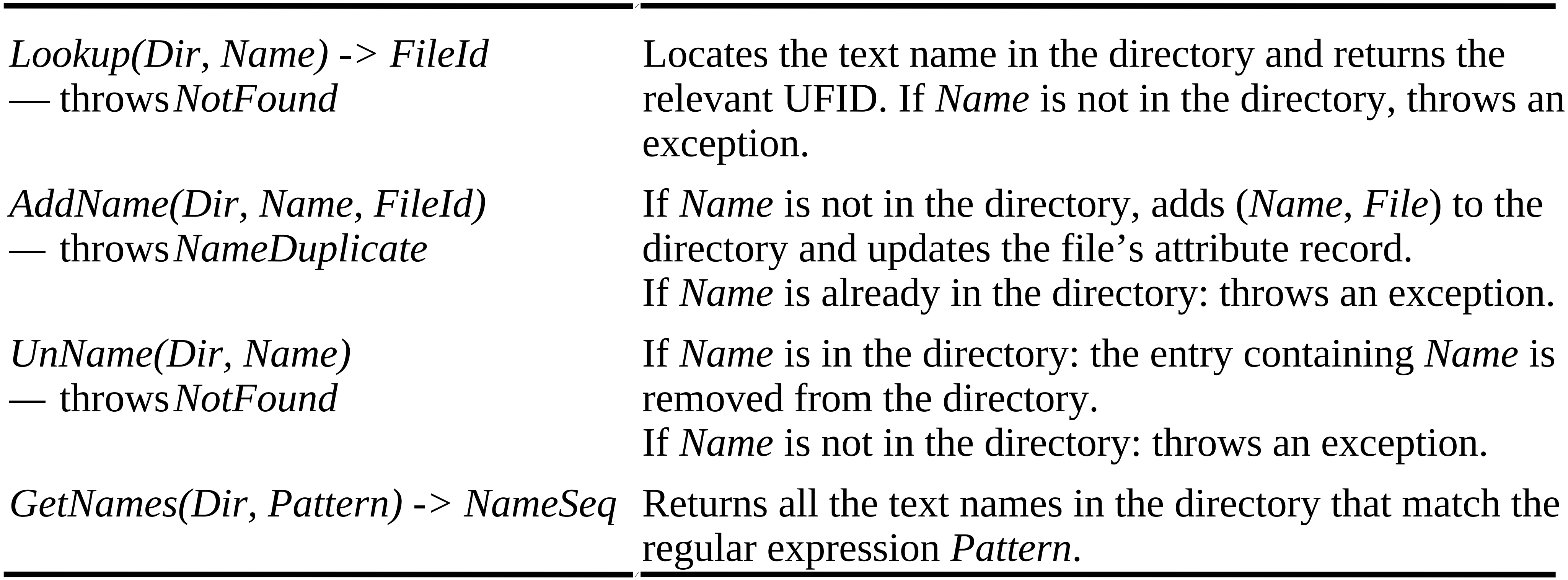
Figure 8.6 Flat file service operations

Access control

- UNIX checks access rights when a file is opened
  - subsequent checks during read/write are not necessary
- distributed environment
  - server has to check
  - stateless approaches
    1. access check once when UFID is issued
       - client gets an encoded "capability" (who can access and how)
       - capability is submitted with each subsequent request
    2. access check for each request.
  - second is more common

Directory service operations

- A directory service interface translates text names to file identifiers and performs a number of other services such as those listed among the sample commands in figure 8.7. This is the remote procedure call interface to extend the local directory services to a distributed model.



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*.

Figure 8.7 Directory service operations

File collections

- Hierarchical file system
  - Directories containing other directories and files
  - Each file can have more than one name (pathnames)
    - how in UNIX, Windows?

- File groups
  - a logical collection of files on one server
    - a server can have more than one group
    - a group can change server
    - a file can't change to a new group (copying doesn't count)
    - filesystems in unix
    - different devices for non-distributed
    - different hosts for distributed

	32 bits	16 bits
file group identifier	IP address	date

8.3 Case Study: Sun NFS

- Sun NFS
  - Industry standard for local networks since the 1980's
  - OS independent (originally unix implementation)
  - rpc over udp or tcp

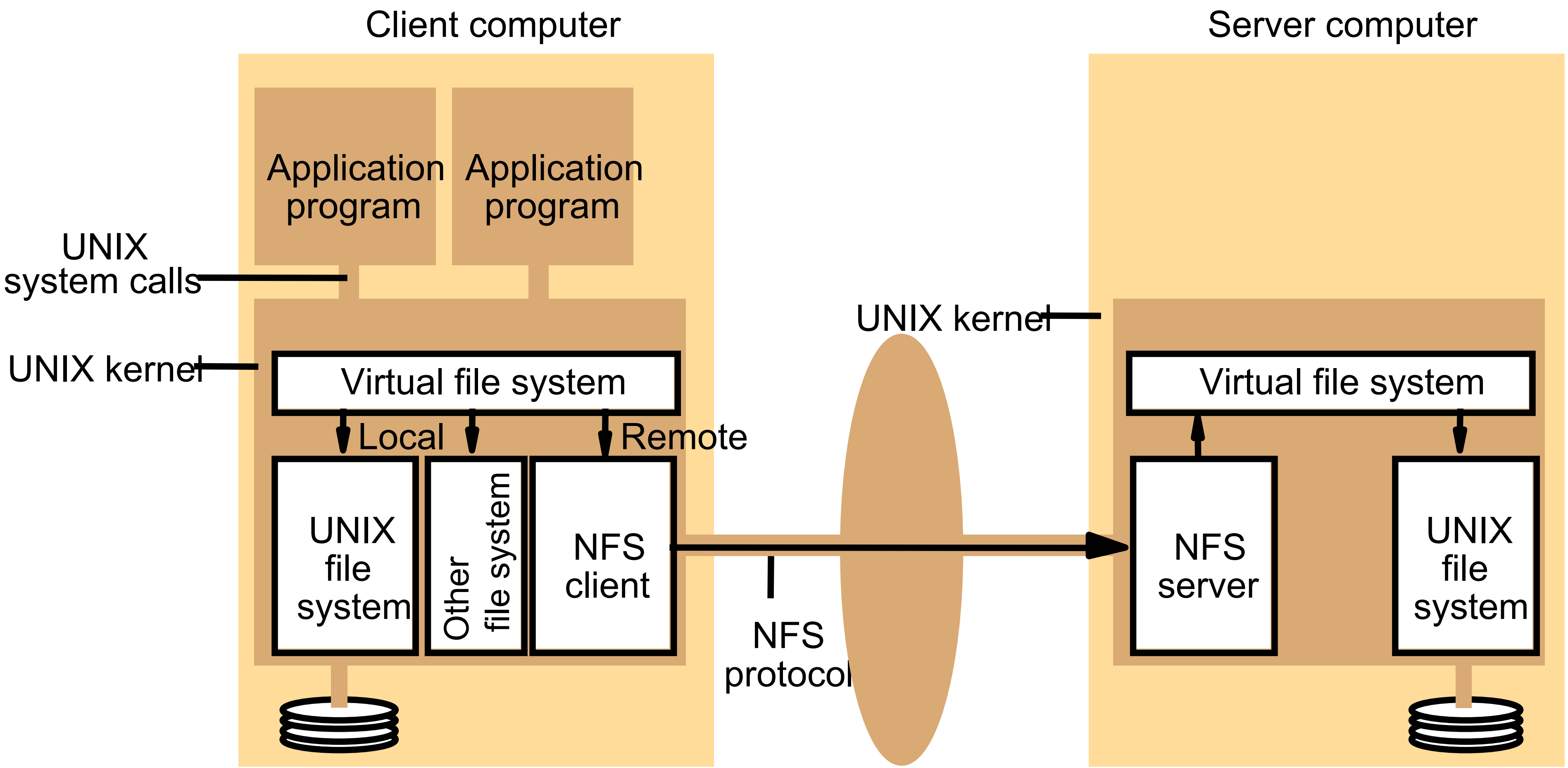
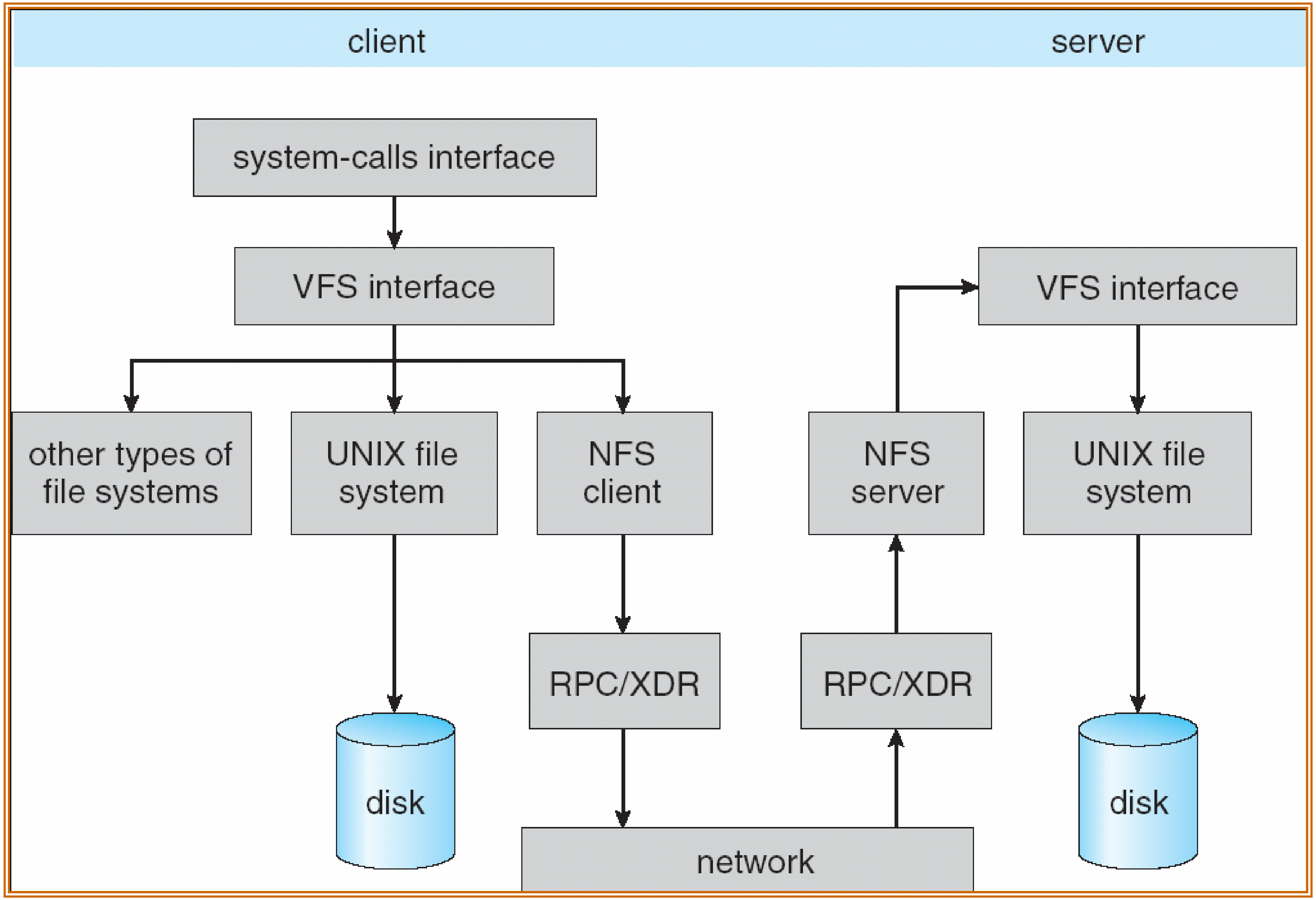


Figure 8.8 NFS architecture

Schematic view of NFS architecture



Refer to Section 11.9 of the OS Textbook

Virtual file system

- Part of unix kernel to support access transparency
- NFS file handles, 3 components:
  - i-node (index node)
    - structure for finding the file
  - filesystem identifier
    - different groups of files
  - i-node generation number
    - i-nodes are reused
    - incremented when reused
- VFS
  - struct for each file system
  - v-node for each open file
    - file handle for remote file
    - i-node number for local file

Access control, client integration, pathname translation

- Access control
  - nfs server is stateless, doesn't keep open files for clients
  - server check identity each time (uid and gid)
- Client integration
  - nfs client emulates Unix file semantics
  - in the kernel, not in a library, because:
    - access files via system calls
    - single client module for multiple user processes
    - encryption can be done in the kernel
- Pathname translation
  - pathname: /users/students/dc/abc
  - server doesn't receive the entire pathname for translation, why?
  - client breaks down the pathnames into parts
  - iteratively translate each part
  - translation is cached

Figure 8.9 server operations (simplified)

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.*

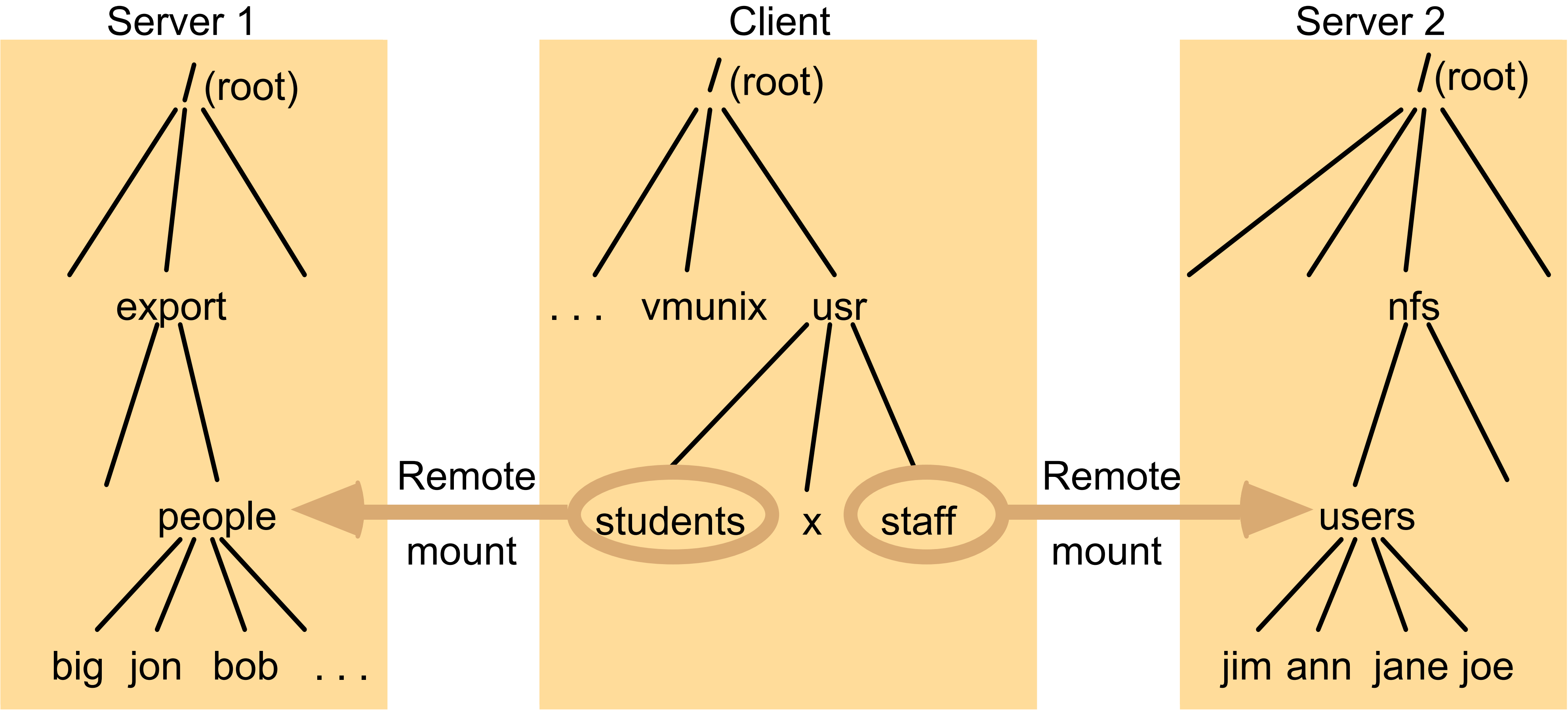
Figure 8.9 NFS server operations (simplified) *cont.*

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*.

Mount

- Mount service
  - mounting: the process of including a new filesystem
  - /etc/exports has filesystems that can be mounted by others
  - clients use a modified mount command for remote filesystems
  - communicates with the mount process on the server in a mount protocol
  - hard-mounted
    - user process is suspended until request is successful
    - when server is not responding
    - request is retried until it's satisfied
  - soft-mounted
    - if server fails, client returns failure after a small # of retries
    - user process handles the failure
- Automounter
  - what if a user process reference a file on a remote filesystem that is not mounted
  - table of mount points (pathname) and servers

Accessible file systems 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
the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

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

Caching

- Server caching
  - caching file pages, directory/file attributes
  - read-ahead: prefetch pages following the most-recently read file pages
  - delayed-write: write to disk when the page in memory is needed for other purposes
  - "sync" flushes "dirty" pages to disk every 30 seconds
  - two write option
    1. write-through: write to disk before replying to the client
    2. cache and commit:
       - stored in memory cache
       - write to disk before replying to a "commit" request from the client
- Client caching
  - caches results of read, write, getattr, lookup, readdir
  - clients responsibility to poll the server for consistency

Client caching: reading

- timestamp-based methods for consistency validation
  - *Tc*: time when the cache entry was last validated
  - *Tm*: time when the block was last modified at the server
- cache entry is valid if:
  1. $T - Tc < t$, where t is the freshness interval
     - t is adaptively adjusted:
       - files: 3 to 30 seconds depending on freq of updates
       - directories: 30 to 60 seconds
  2. $Tm_{client} = Tm_{server}$
- cache validation
  - need validation for all cache accesses due to no share check
  - condition “1" can be determined by the client alone--performed first
  - Reducing getattr() to the server [for getting Tm_{server}]
    1. new value of Tm_{server} is received, apply to all cache entries from the same file
    2. piggyback getattr() on file operations
    3. adaptive alg for update t
  - validation doesn't guarantee the same level of consistency as one-copy

Client caching: writing

- dirty: modified page in cache
- flush to disk: file is closed or sync from client
- bio-daemon (block input-output)
  - read-ahead: after each read request, request the next file block from the server as well
  - delayed write: after a block is filled, it's sent to the server
  - reduce the time to wait for read/write

Other optimization

- UDP packet is extended to 9KB to containing entire file block (8KB for UNIX BSD FFS) and RPC message in a single packet
  - Clients and servers of NFSv3 can negotiate sizes larger than 8 KB
- Piggybacked
  - File status information cached at clients must be updated at least every 3 seconds for active files
  - All operations that refer to files or directories are taken as implicit *getattr* requests, and the current attribute values are piggybacked along with the other results of the operation

Security, Performance

- Security
  - stateless nfs server
  - user's identity in each request
  - Kerberos authentication during the mount process, which includes uid and host address
  - server maintain authentication info for the mount
  - on each file request, nfs checks the uid and address
  - one user per client
- Performance
  - overhead/penalty is low
  - main problems
    - frequent getattr() for cache validation (piggybacking)
    - relatively poor performance is write-through is used on the server (delay-write/commit in current versions)
  - write < 5%
  - lookup is almost 50% (step by step pathname translation)

Summary for NFS

An excellent example of a simple, robust, high-performance distributed service

- access transparency: same system calls for local or remote files
- location transparency: could have a single name space for all files (depending on all the clients to agree the same name space)
- mobility transparency: mount table need to be updated on each client (not transparent)
- scalability: can usually support large loads, add processors, disks, servers...
- file replication: read-only replication, no support for replication of files with updates
- hardware and OS: many ports
- fault tolerance: stateless and idempotent
- consistency: not quite one-copy for efficiency
- security: added encryption--Kerberos
- efficiency: pretty efficient, wide-spread use

8.4 Case Study: the Andrew File System

- Goal: provide transparent access to remote shared files
  - like NFS and compatible with NFS
- Differing from NFS
  - Primarily attributable to the scalability
    - Caching of whole files in client nodes
- Two unusual design characteristics
  - Whole file serving: the entire contents of directories and files are transmitted to client computers
    - 64KB chunks in AFS3
  - Whole-file caching: clients permanently cache a copy of a file or a chunk on its local disk

Scenario of AFS

- Open a new shared remote file
  - A user process issues open() for a file not in the local cache and then sends a request to the server
  - The server returns the requested file
  - The copy is stored in the client's local UNIX file system and the resulting UNIX file descriptor is returned to the client
- Subsequent read, write and other operations on the file are applied to the local copy
- When the process in the client issues close()
  - if the local copy has been updated, its contents are sent back to the server
    - server updates the contents and the timestamps on the file
    - the copy on the client's local disk is retained

Characteristics

- Good for shared files likely to remain valid for long periods
  - infrequently updated
  - normally accessed by only a single user

  Overwhelming majority of file accesses
- Local cache can be allocated a substantial proportion of the disk space
  - should be enough for a working set of files used by one user
- Assumptions about average and maximum file size and reference locality
  - Files are small; most are less than 10KB in size
  - Read operations are much more common than writes
  - Sequential access is much more common than random access
  - Most files are written by only one user. When a file is shared, it is usually only one user who modified it
  - Files are referenced in bursts. A file referenced recently is very probably referenced soon.
- Maybe good for distributed database applications

Venus and Vice: software components in AFS

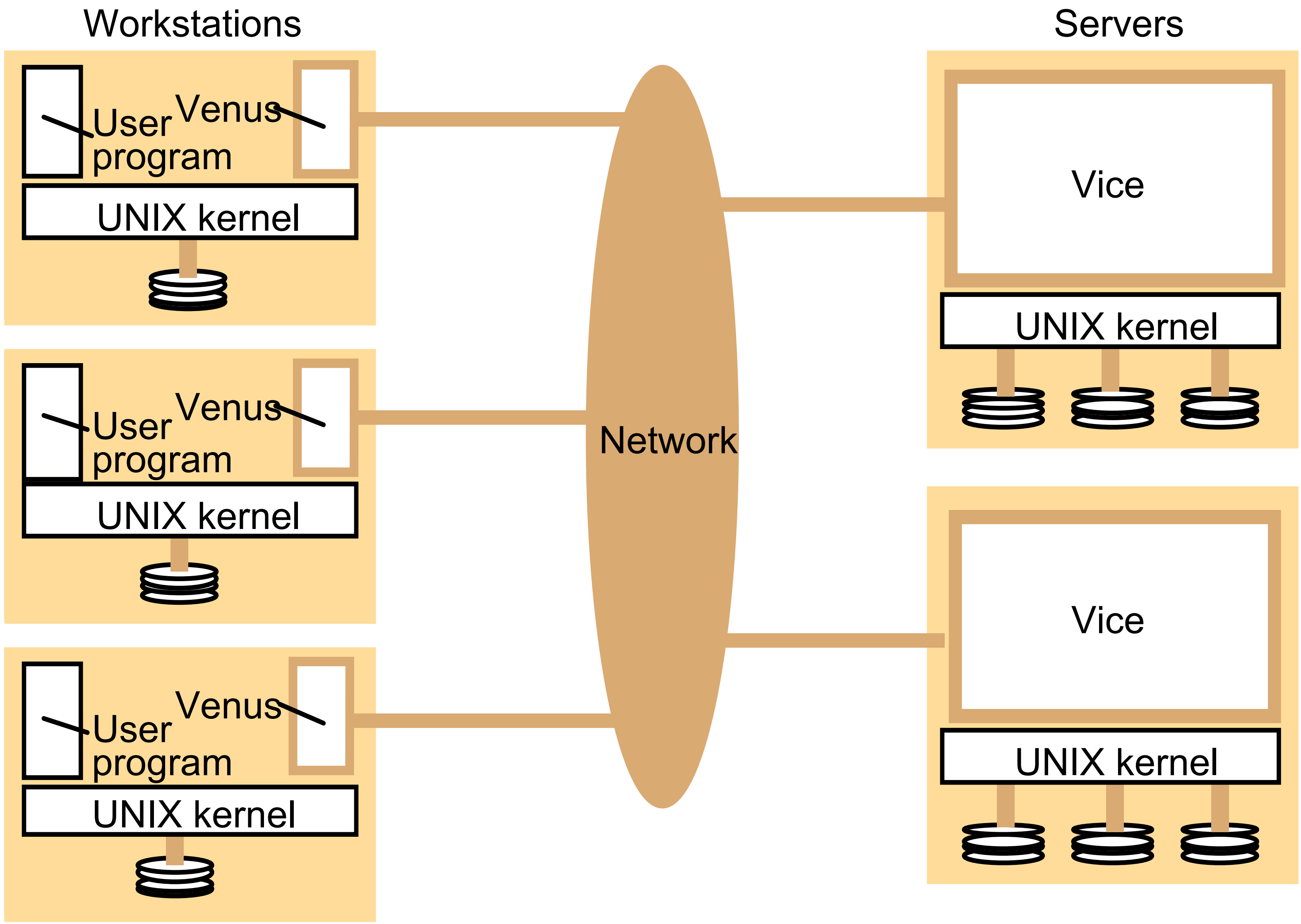
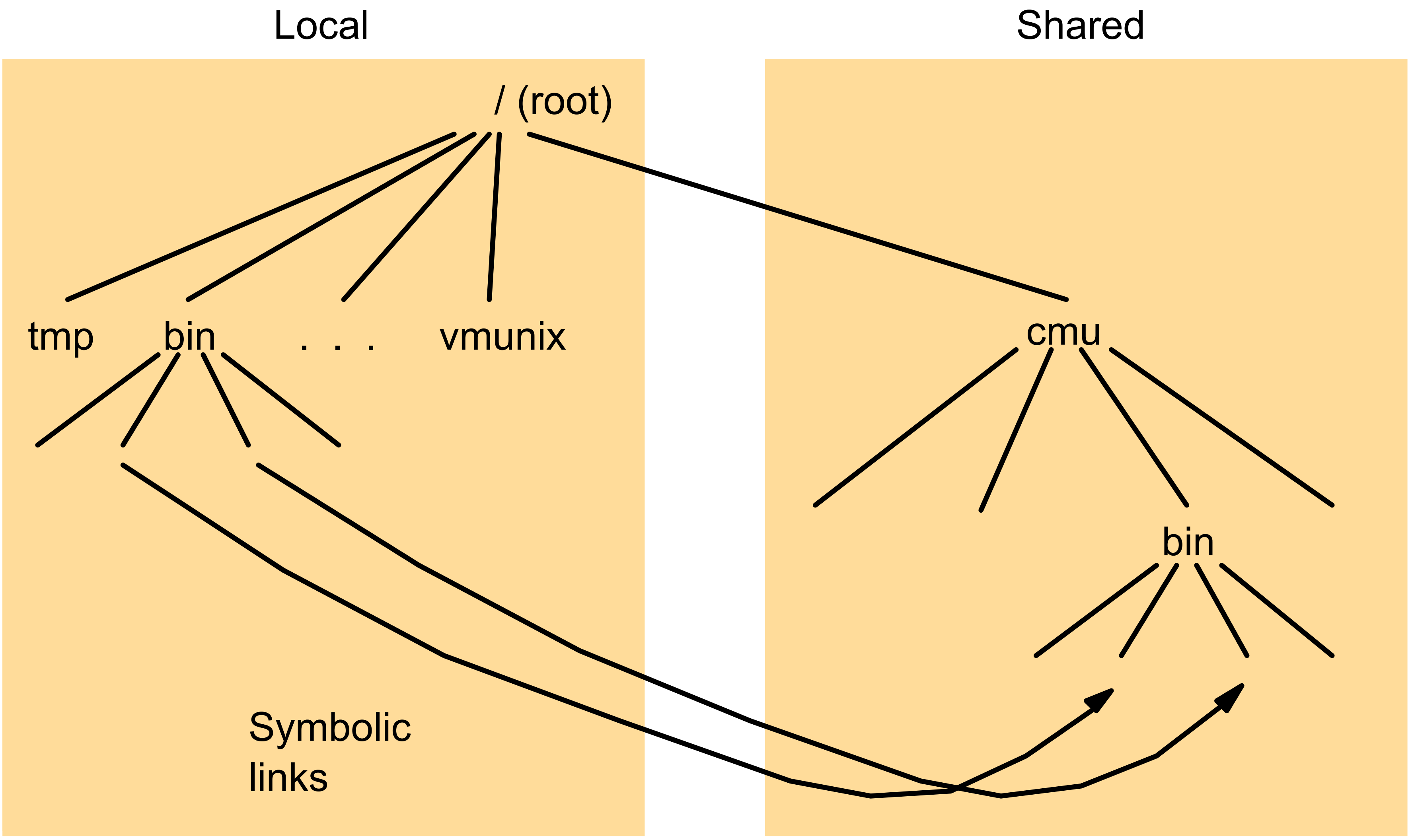


Figure 8.11 Distribution of processes in the Andrew File System

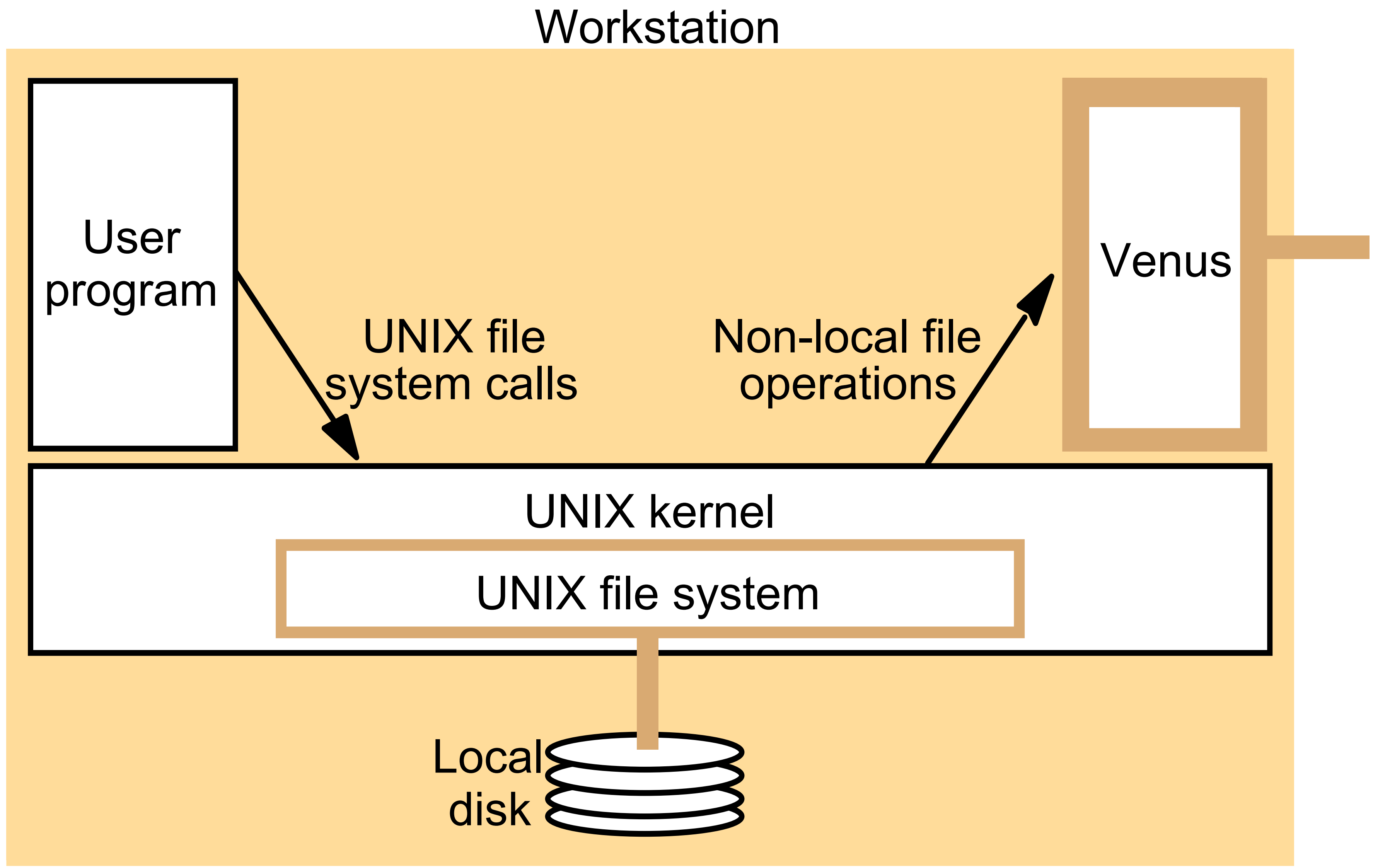
File name space seen by clients of AFS



A special sub-tree (called /cmu) containing all of the shared files

Figure 8.12 File name space seen by clients of AFS

System call interception in AFS



Open, close and some other file system calls are intercepted when refer to shared files

Figure 8.13 System call interception in AFS

Figure 8.14
Implementation of file system calls in AFS

User process	Unix Kernel	Venus	Net	Vice
open(FileName, mode)	If FileName refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid ***callback promise***, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and a ***callback promise*** to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice serve r that is the custodian of the file.	→	Replace the file contents and send a ***callback*** to all other clients holding callback promises on the file.

Main components of Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Figure 8.15

8.5 Enhancements and further developments

NFS enhancements

WebNFS - NFS server implements a web-like service on a well-known port. Requests use a 'public file handle' and a pathname-capable variant of *lookup().* Enables applications to access NFS servers directly, e.g. to read a portion of a large file.

One-copy update semantics (Spritely NFS, NQNFS) - Include an *open()* operation and maintain tables of open files at servers, which are used to prevent multiple writers and to generate callbacks to clients notifying them of updates. Performance was improved by reduction in *gettattr()* traffic.

Improvements in disk storage organisation

RAID - improves performance and reliability by striping data redundantly across several disk drives

Log-structured file storage - updated pages are stored contiguously in memory and committed to disk in large contiguous blocks (~ 1 Mbyte). File maps are modified whenever an update occurs. Garbage collection to recover disk space.

New design approaches

- Distribute file data across several servers
  - Exploits high-speed networks (ATM, Gigabit Ethernet)
  - Layered approach, lowest level is like a 'distributed virtual disk'
  - Achieves scalability even for a single heavily-used file
- 'Serverless' architecture
  - Exploits processing and disk resources in all available network nodes
    - Service is distributed at the level of individual files
  - Examples:
    - xFS (section 8.5): Experimental implementation demonstrated a substantial performance gain over NFS and AFS
    - Frangipani (section 8.5): Performance similar to local UNIX file access
    - Tiger Video File System (see Chapter 15)
    - P2P systems: Napster, OceanStore (UCB), Farsite (MSR), Publius (AT&T research) - see web for documentation on these very recent systems
- Replicated read-write files
  - High availability
  - Disconnected working
    - re-integration after disconnection is a major problem if conflicting updates have occurred