Distributed File System

Marco Aiello

(based on http://www.cdk4.net)

Distributed File Systems

- A distributed file system enables the applications of a operating system to access files transparently from their being stored locally or remotely.
- A well-designed distributed file system provides access to files at the same speed as to local file system or even faster
- Advantages:
  - Remote access
  - Economy
  - Maintenance
  - Robustness
  - Sharing of resources

Storage systems and their properties

- In first generation of distributed systems (1974-95), file systems (e.g. NFS) were the only networked storage systems.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Dropbox is an example of a distributed file system

What is a file system?

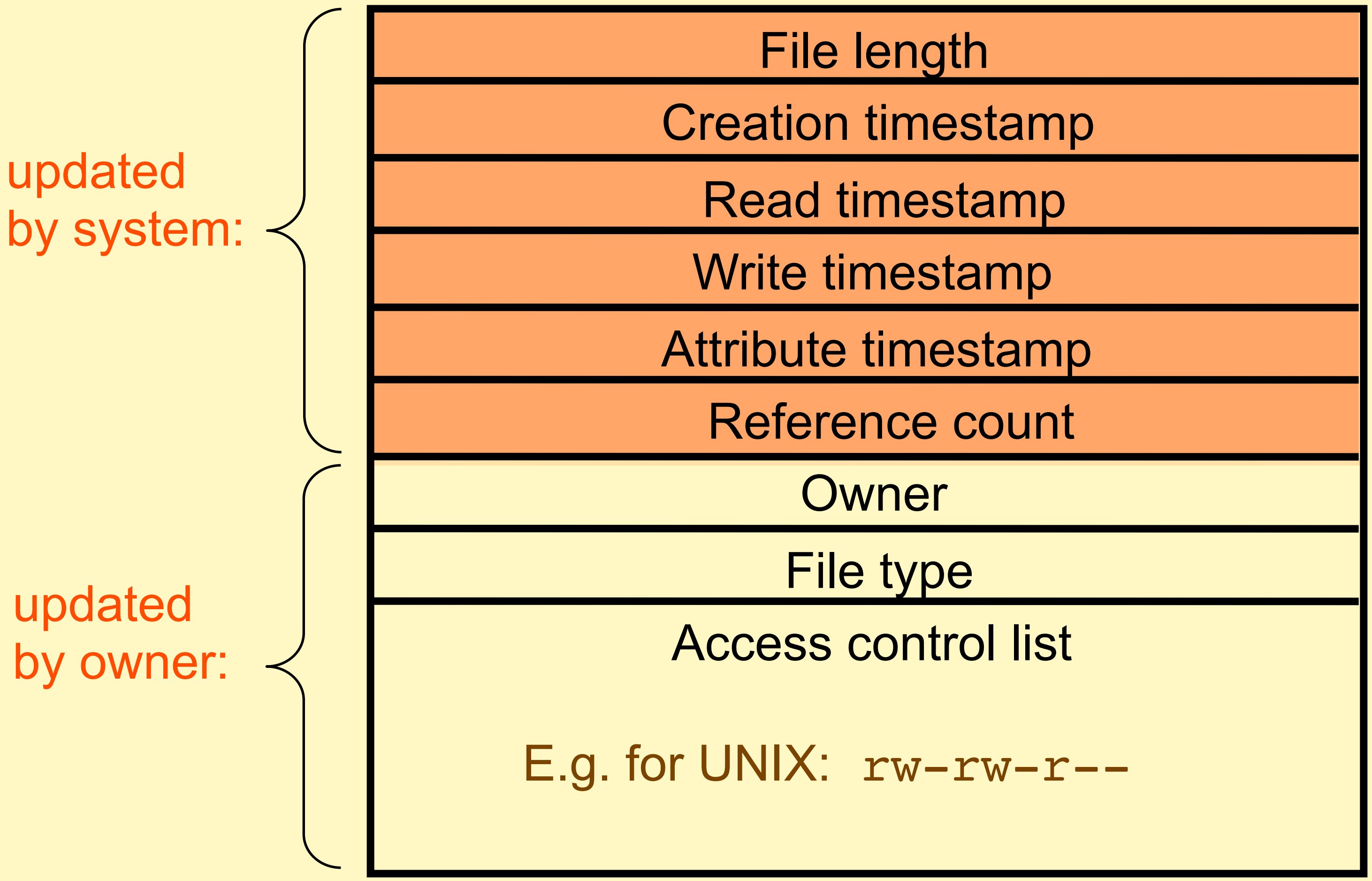
- Persistent stored data sets
- Hierarchic name space visible to all processes
- API with the following characteristics:
  - access and update operations on persistently stored data sets
  - sequential access model (with additional random facilities)
- Sharing of data between users, with access control
- Concurrent access:
  - certainly for read-only access
  - what about updates?
- Other features:
  - mountable file stores
  - more? ...

Figure 12.2
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

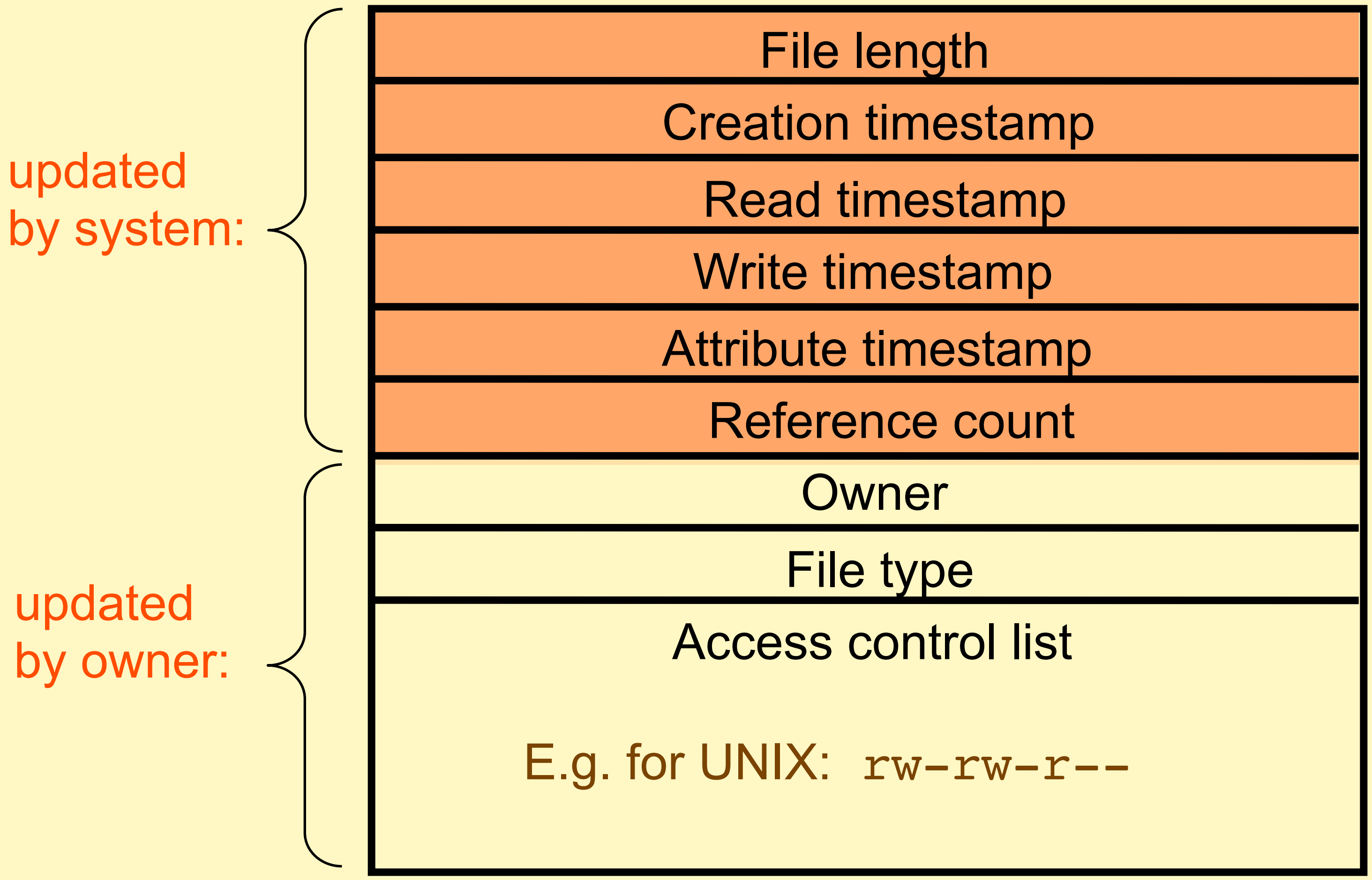
What is a file system?

Figure 8.3 File attribute record structure



What is a file system?

Figure 8.3 File attribute record structure



What is a file system?

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*.

What is a file system?

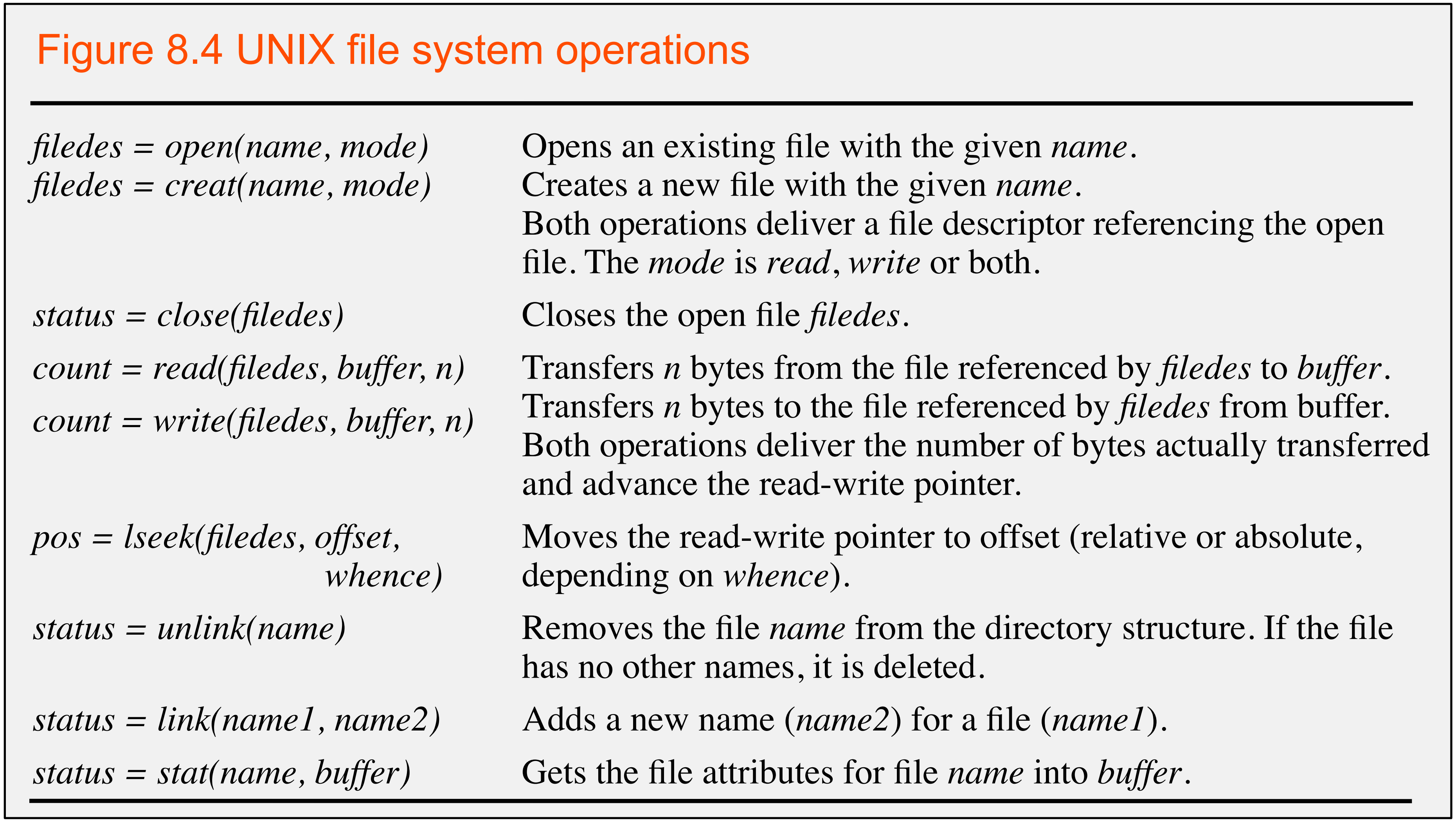


Figure 8.4 UNIX file system operations

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*.

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

Tranparencies

Access:	Same operations
Location:	Same name space after relocation of files or processes
Mobility:	Automatic relocation of files is possible
Performance:	Satisfactory performance across a specified range of system loads
Scaling:	Service can be expanded to meet additional loads

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

<u>Concurrency properties</u>

Isolation

File-level or record-level locking

Other forms of concurrency control to minimise contention

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

<u>Replication properties</u>

File service maintains multiple identical copies of files

- Load-sharing between servers makes service more scalable
- Local access has better response (lower latency)
- Fault tolerance

Full replication is difficult to implement.

Caching (of all or part of a file) gives most of the benefits (except fault tolerance)

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

Heterogeneity properties

Service can be accessed by clients running on (almost) any OS or hardware platform.

Design must be compatible with the file systems of different OSes

Service interfaces must be *open* - precise specifications of APIs are published.

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

Fault tolerance

Service must continue to operate even when clients make errors or crash.

- at-most-once semantics
- at-least-once semantics
  - requires idempotent operations

Service must resume after a server machine crashes.

If the service is replicated, it can continue to operate even during a server crash.

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

Consistency

Unix offers one-copy update semantics for operations on local files - caching is completely transparent.

Difficult to achieve the same for distributed file systems while maintaining good performance and scalability.

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

<u>Security</u>

Must maintain access control and privacy as for local files.

- based on identity of user making request
- identities of remote users must be authenticated
- privacy requires secure communication

Service interfaces are open to all processes not excluded by a firewall.

- vulnerable to impersonation and other attacks

File service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency

Efficiency

Goal for distributed file systems is usually performance comparable to local file system.

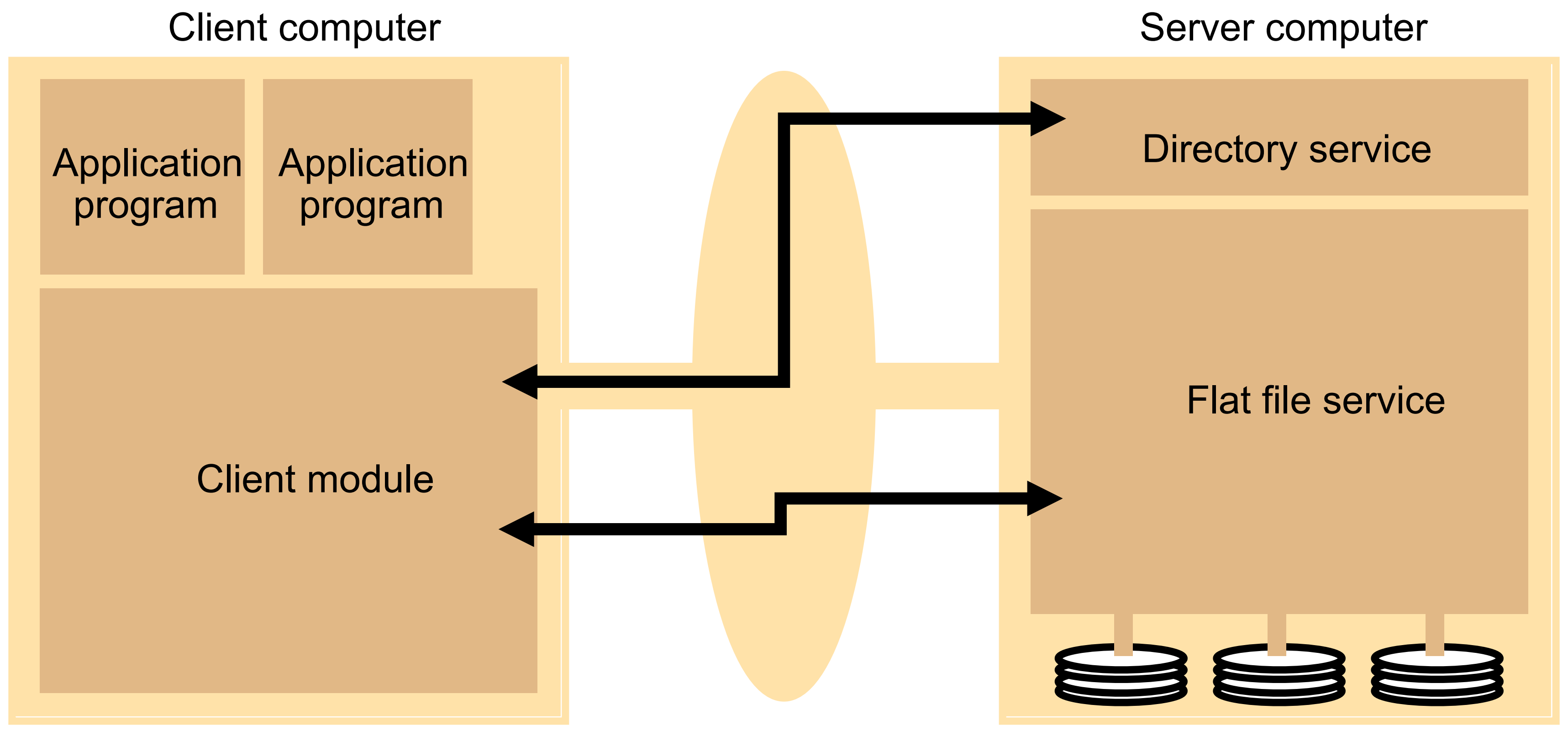
*

File service architecture

- Flat services based on UFID (unique file identifiers): Client module, server module (directory and flat file services)
- Fault tolerance:
  - idempotent operations (i.e., repeatable)
  - stateless servers
- Unix is not suitable for stateless implementation (read, write are not idempotent)
- Access control (for stateless server implementation):
  - Access check whenever a file name is converted in UFID
  - User identity check with every client request

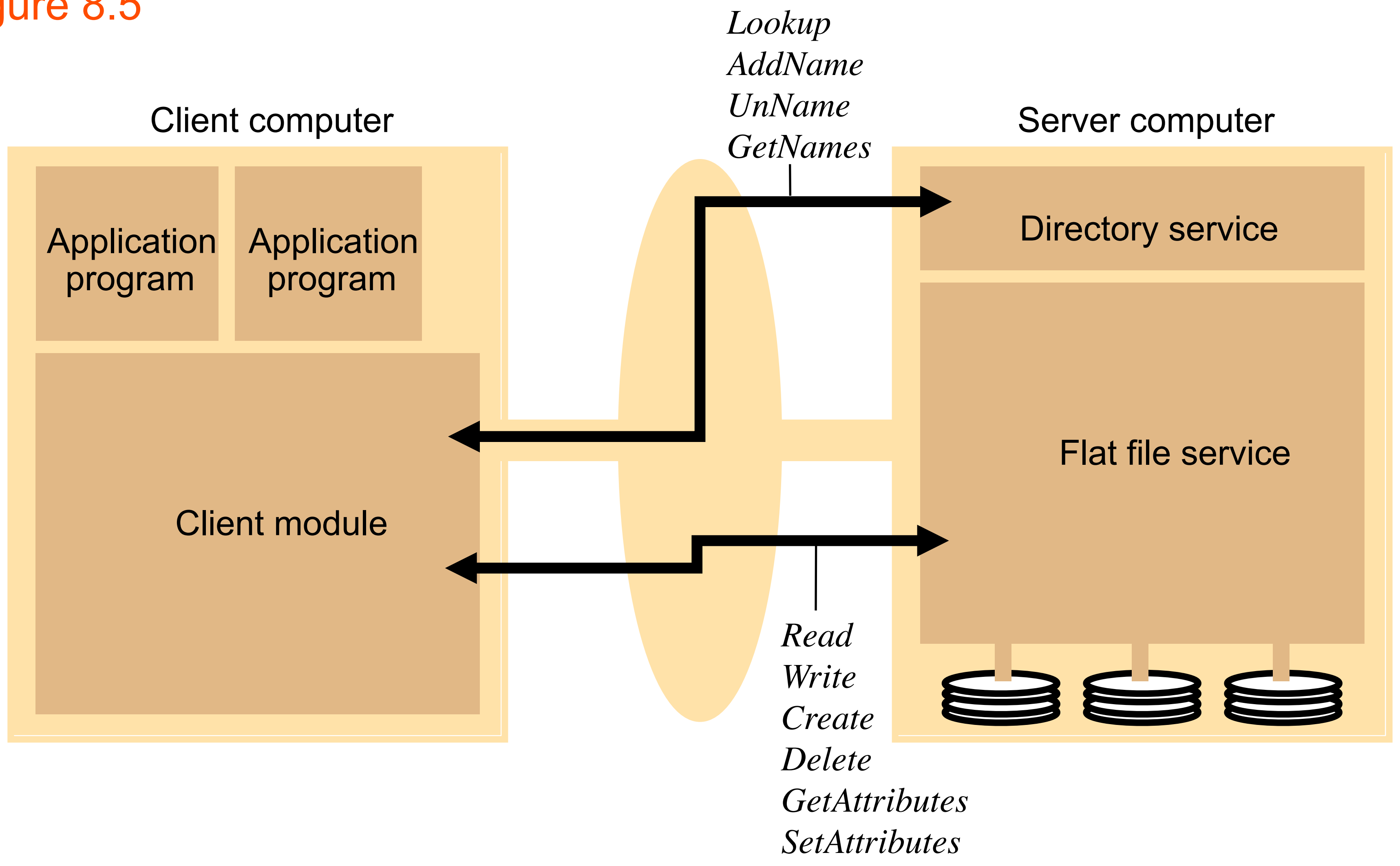
Model file service architecture

Figure 8.5



Model file service architecture

Figure 8.5



*

Server operations (RPC) for the model file service

Figures 8.6 and 8.7

Flat file service

Read(FileId, i, n) -> Data

Write(FileId, i, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, FileId)

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

Server operations (RPC) for the model file service

Figures 8.6 and 8.7

Flat file service

Read(FileId, $\overset{\text{position of first byte}}{i}$, n) -> Data

Write(FileId, $\overset{\text{position of first byte}}{i}$, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, FileId)

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

Server operations (RPC) for the model file service

Figures 8.6 and 8.7

Flat file service

Read(FileId, $\overset{\text{position of first byte}}{i}$, n) -> Data

Write(FileId, $\overset{\text{position of first byte}}{i}$, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, FileId)

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

Pathname lookup

Pathnames such as '/usr/bin/tar' are resolved by iterative calls to *lookup()*, one call for each component of the path, starting with the ID of the root directory '/' which is known in every client.

*

File Group

A collection of files that can be located on any server or moved between servers while maintaining the same names.

- Similar to a UNIX *filesystem*
- Helps with distributing the load of file serving between several servers.
- File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
  - *Used to refer to file groups and files*

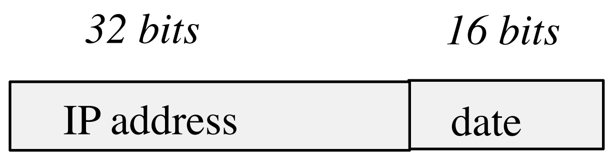
File Group

A collection of files that can be located on any server or moved between servers while maintaining the same names.

- Similar to a UNIX *filesystem*
- Helps with distributing the load of file serving between several servers.
- File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
  - *Used to refer to file groups and files*

To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently.

File Group ID:



Case Study: Sun NFS

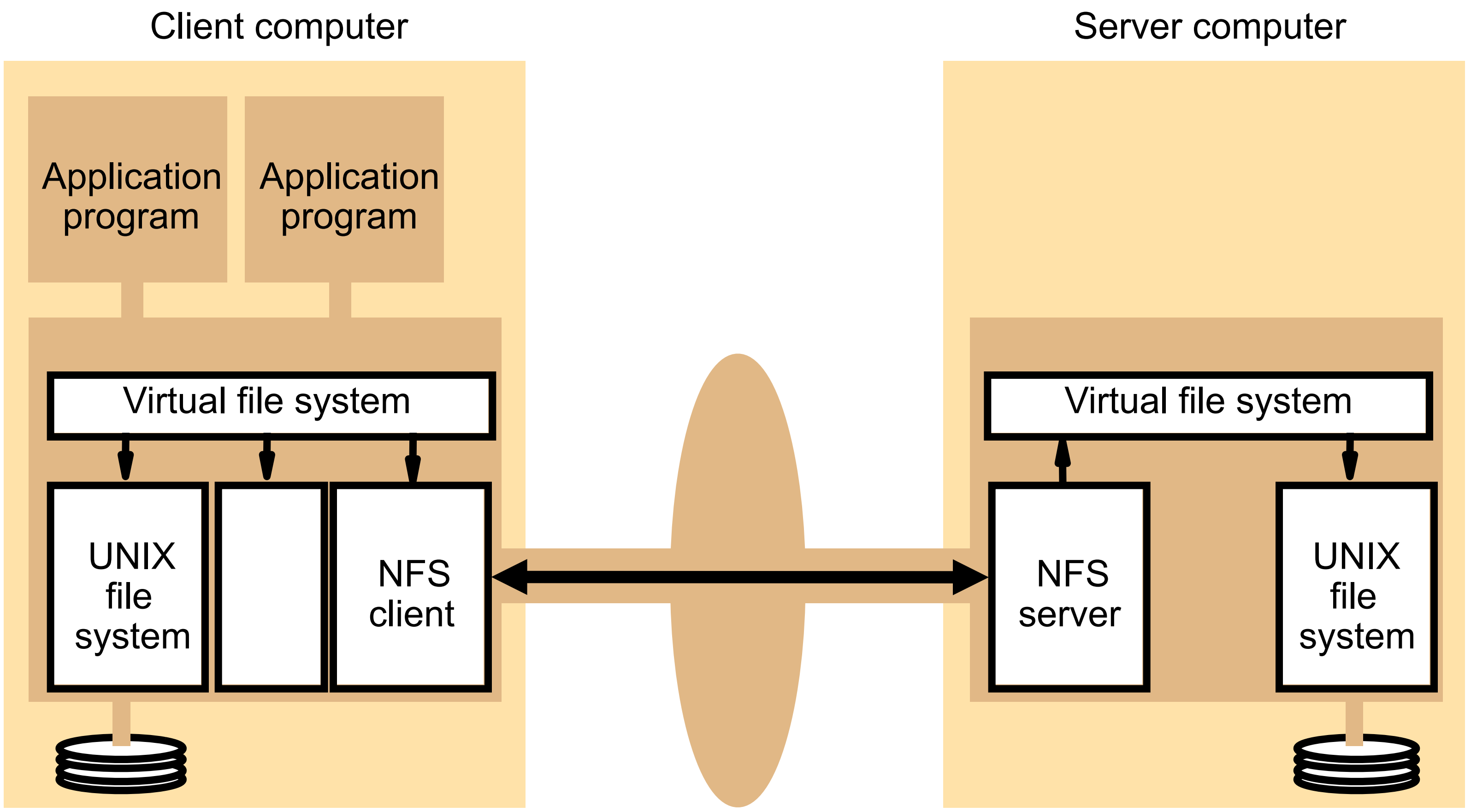
- An industry standard for file sharing on local networks since the 1980s
- An open standard with clear and simple interfaces
- Closely follows the abstract file service model defined above
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance
- Limited achievement of:
  - concurrency
  - replication
  - consistency
  - security

Case Study: Sun NFS

- An industry standard for file sharing on local networks since the 1980s
- An open standard with clear and simple interfaces
- Closely follows the abstract file service model defined above
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance
- Limited achievement of:
  - concurrency
  - replication
  - consistency
  - security

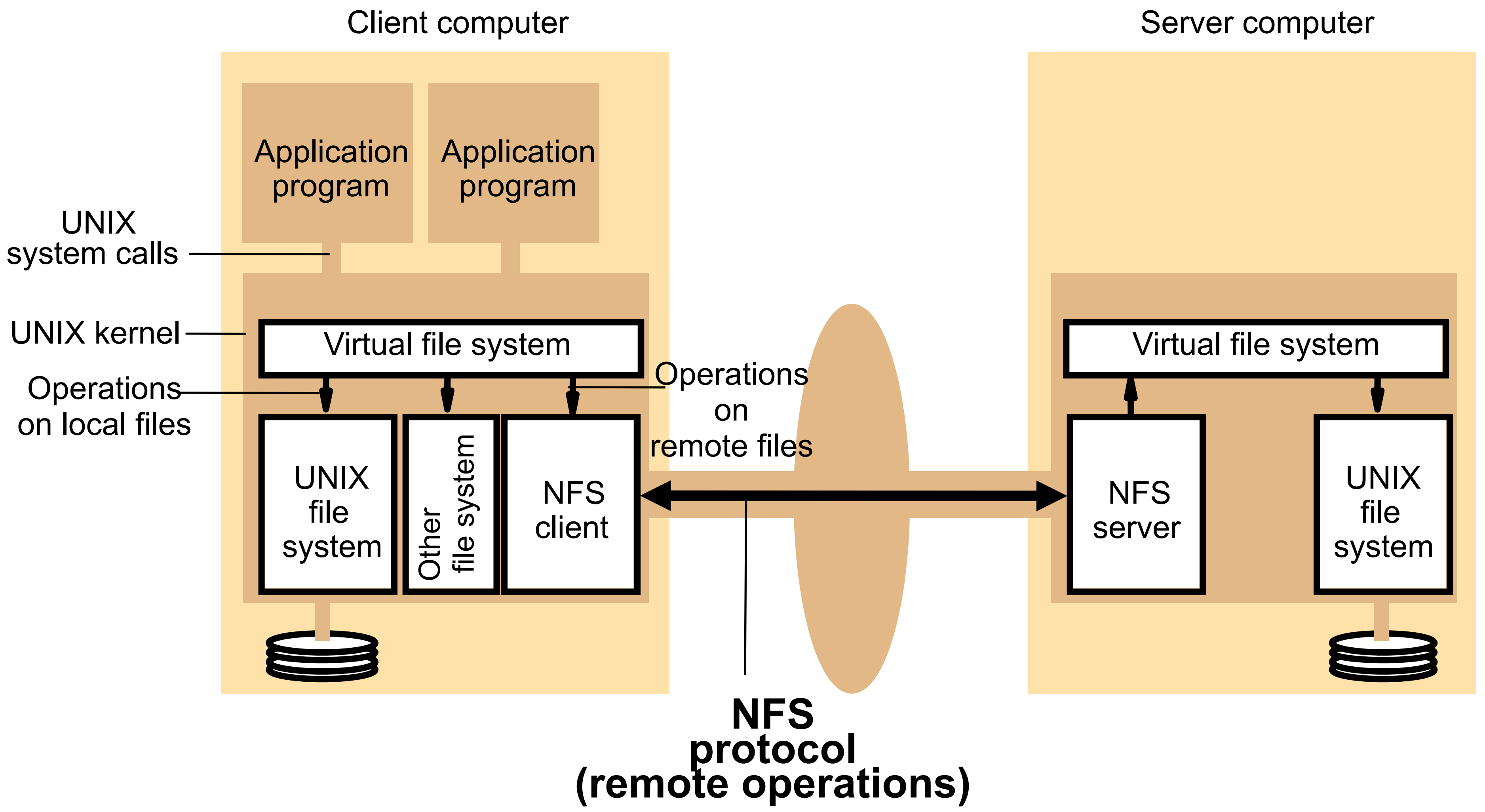
NFS architecture

Figure 8.8



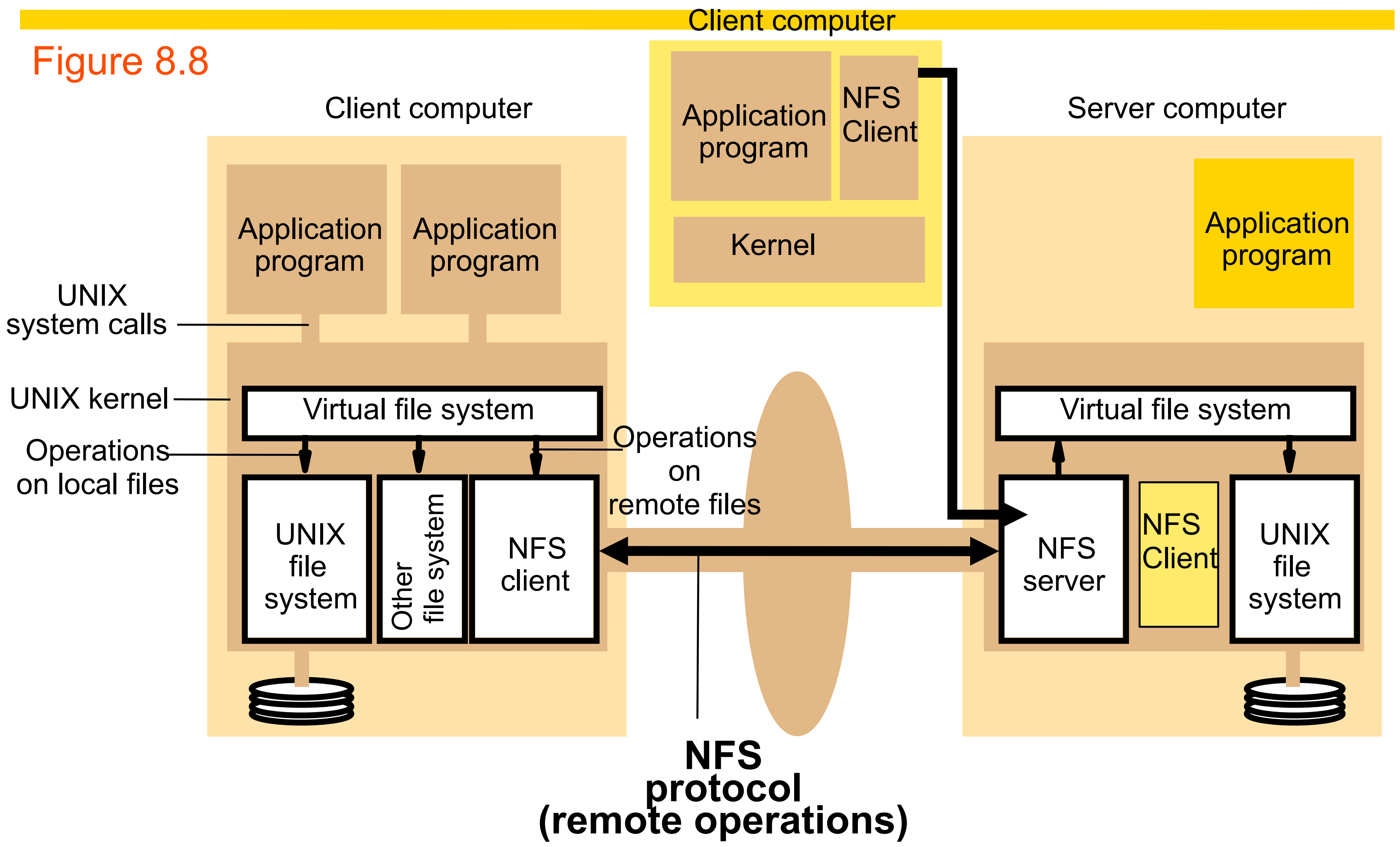
NFS architecture

Figure 8.8



NFS architecture

Figure 8.8



NFS architecture: does the implementation have to be in the system kernel?

No:

- there are examples of NFS clients and servers that run at application-level as libraries or processes (e.g. early Windows and MacOS implementations, current PocketPC, etc.)

But, for a Unix implementation there are advantages:

- Binary code compatible - no need to recompile applications
  - *Standard system calls that access remote files can be routed through the NFS client module by the kernel*
- Shared cache of recently-used blocks at client
- Kernel-level server can access i-nodes and file blocks directly
  - *but a privileged (root) application program could do almost the same.*
- Security of the encryption key used for authentication.

NFS server operations (simplified)

Figure 8.9

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

fh = file handle:

Filesystem identifier	i-node number	i-node generation

NFS server operations (simplified)

Figure 8.9

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> statu*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

Model flat file service

Read(FileId, i, n) -> Data
Write(FileId, i, Data)
Create() -> FileId
Delete(FileId)
GetAttributes(FileId) -> Attr
SetAttributes(FileId, Attr)

Model directory service

Lookup(Dir, Name) -> FileId
AddName(Dir, Name, File)
UnName(Dir, Name)
GetNames(Dir, Pattern)
->NameSeq

*

NFS access control and authentication

- Stateless server, so the user's identity and access rights must be checked by the server on each request.
  - In the local file system they are checked only on *open()*
- Every client request is accompanied by the userID and groupID
  - not shown in the Figure 8.9 because they are inserted by the RPC system
- Server is exposed to imposter attacks unless the userID and groupID are protected by encryption
- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution

Mount service

- Mount operation:

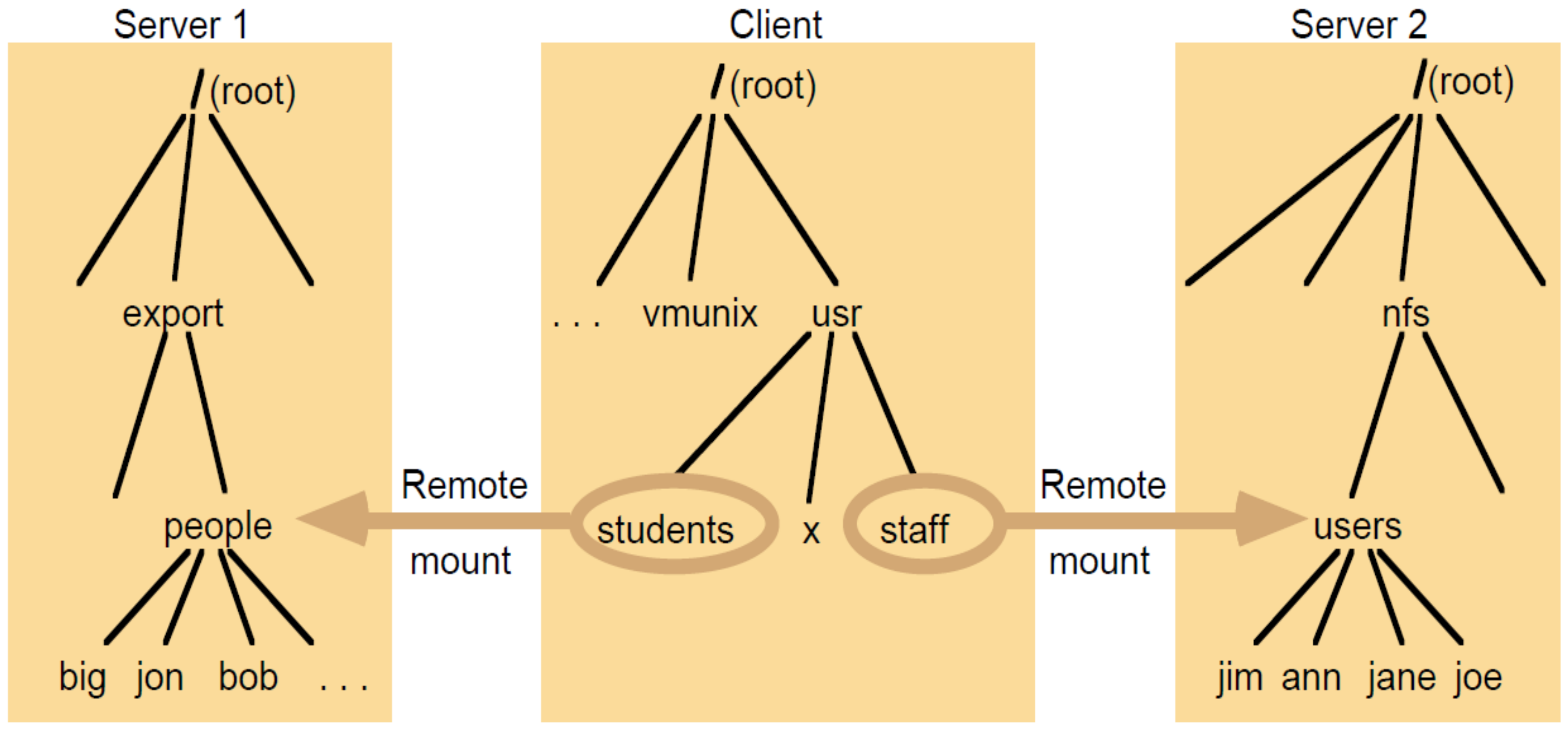
  mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server
- Each client maintains a table of mounted file systems holding:

  < IP address, port number, file handle>

- *Hard* (client blocks until success) versus *soft* mounts

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.

Automounter

NFS client catches attempts to access 'empty' mount points and routes them to the Automounter

- Automounter has a table of mount points and multiple candidate serves for each
- it sends a probe message to each candidate server and then uses the mount service to mount the filesystem at the first server to respond

- Keeps the mount table small
- Provides a simple form of replication for read-only filesystems
  - E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.

NFS optimization - server caching

- Similar to UNIX file caching for local files:
  - pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.
  - For local files, writes are deferred to next sync event (30 second intervals)
  - Works well in local context, where files are always accessed through the local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.
- NFS v3 servers offers two strategies for updating the disk:
  - *write-through* - altered pages are written to disk as soon as they are received at the server. When a *write()* RPC returns, the NFS client knows that the page is on the disk.
  - *delayed commit* - pages are held only in the cache until a *commit()* call is received for the relevant file. This is the default mode used by NFS v3 clients. A *commit()* is issued by the client whenever a file is closed.

NFS optimization - client caching

- Server caching does nothing to reduce RPC traffic between client and server
  - further optimization is essential to reduce server load in large networks
  - NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations
  - synchronization of file contents (*one-copy semantics*) is not guaranteed when two or more clients are sharing the same file.
- Timestamp-based validity check
  - reduces inconsistency, but doesn't eliminate it
  - validity condition for cache entries at the client:

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

  - t is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories
  - it remains difficult to write distributed applications that share files with NFS

*

NFS optimization - client caching

- Server caching does nothing to reduce RPC traffic between client and server
  - further optimization is essential to reduce server load in large networks
  - NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations
  - synchronization of file contents (*one-copy semantics*) is not guaranteed when two or more clients are sharing the same file.
- Timestamp-based validity check
  - reduces inconsistency, but doesn't eliminate it
  - validity condition for cache entries at the client:

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

  - t is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories
  - it remains difficult to write distributed applications that share files with NFS

t	freshness guarantee
Tc	time when cache entry was last validated
Tm	time when block was last updated at server
T	current time

*

Other NFS optimizations

- Sun RPC runs over UDP by default (can use TCP if required)
- Uses UNIX BSD Fast File System with 8-kbyte blocks
- *reads()* and *writes()* can be of any size (negotiated between client and server)
- the guaranteed freshness interval t is set adaptively for individual files to reduce *gettattr()* calls needed to update Tm
- file attribute information (including Tm) is piggybacked in replies to all file requests

Other NFS optimizations

- Sun RPC runs over UDP by default (can use TCP if required)
- Uses UNIX BSD Fast File System with 8-kbyte blocks
- *reads()* and *writes()* can be of any size (negotiated between client and server)
- the guaranteed freshness interval *t* is set adaptively for individual files to reduce *gettattr()* calls needed to update *Tm*
- file attribute information (including *Tm*) is piggybacked in replies to all file requests

NFS performance

- Early measurements (1987) established that:
  - *write()* operations are responsible for only 5% of server calls in typical UNIX environments
    - *hence write-through at server is acceptable*
  - *lookup()* accounts for 50% of operations -due to step-by-step pathname resolution necessitated by the naming and mounting semantics.
- Latert measurements (1993) show high performance:

  1 x 450 MHz Pentium III: > 5000 server ops/sec, < 4 millisec. average latency

  24 x 450 MHz IBM RS64: > 29,000 server ops/sec, < 4 millisec. average latency

  see www.spec.org for more recent measurements
- Provides a good solution for many environments including:
  - large networks of UNIX and PC clients
  - multiple web server installations sharing a single file store

NFS summary *1*

- An excellent example of a simple, robust, high-performance distributed service.
- Achievement of transparencies:

  Access: *Excellent*; the API is the UNIX system call interface for both local and remote files.

  Location: *Not guaranteed but normally achieved*; naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.

  Concurrency: *Limited but adequate for most purposes*; when read-write files are shared concurrently between clients, consistency is not perfect.

  Replication: *Limited to read-only file systems*; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.

NFS summary

Achievement of transparencies (continued):

Failure: *Limited but effective*; service is suspended if a server fails. Recovery from failures is aided by the simple stateless design.

Mobility: *Hardly achieved*; relocation of files is not possible, relocation of filesystems is possible, but requires updates to client configurations.

Performance: *Good*; multiprocessor servers achieve very high performance, but for a single filesystem it's not possible to go beyond the throughput of a multiprocessor server.

Scaling: *Good*; filesystems (file groups) may be subdivided and allocated to separate servers. Ultimately, the performance limit is determined by the load on the server holding the most heavily-used filesystem (file group).

Recent advances in file services

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.

Summary

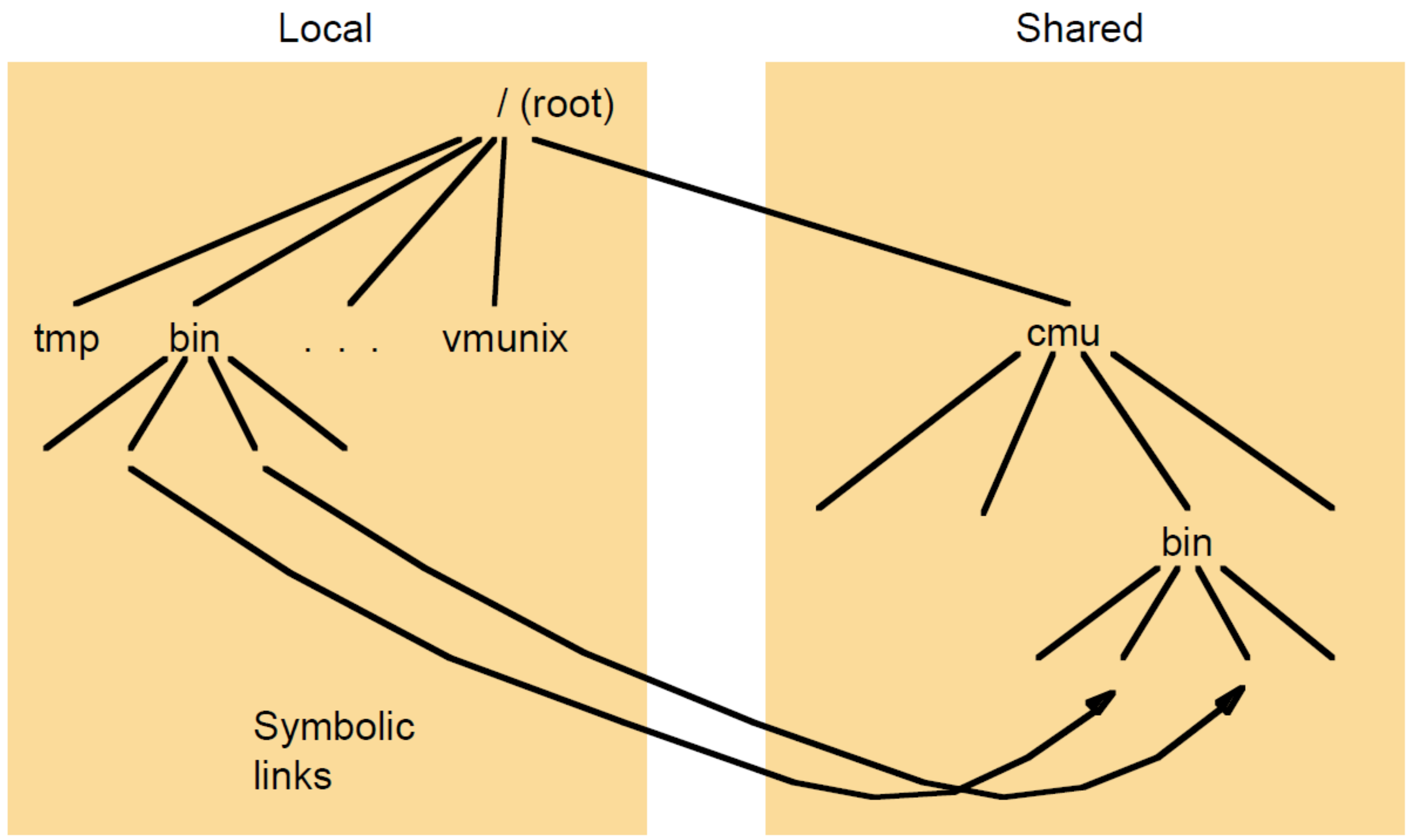
- Sun NFS is an excellent example of a distributed service designed to meet many important design requirements
- Effective client caching can produce file service performance equal to or better than local file systems
- Consistency *versus* update semantics *versus* fault tolerance remains an issue
- Most client and server failures can be masked
- Superior scalability can be achieved with whole-file serving (Andrew FS) or the distributed virtual disk approach

The Andrew File System

- Originated at CMU in 1983 (Andrew Carnegie and Andrew Mellon)
- It addresses scalability and security
- Based on kerberos for authentication
- It distinguishes among local and shared files

Figure 8.12

File name space seen by clients of AFS



AFS cont’d

- **Entire** files are cached at clients
- When a file system call is issued, it is intercepted
- If it is on a shared file, the Venus module intervenes
- Venus manages the files as a write-through cache on closes

Figure 8.13

System call interception in AFS

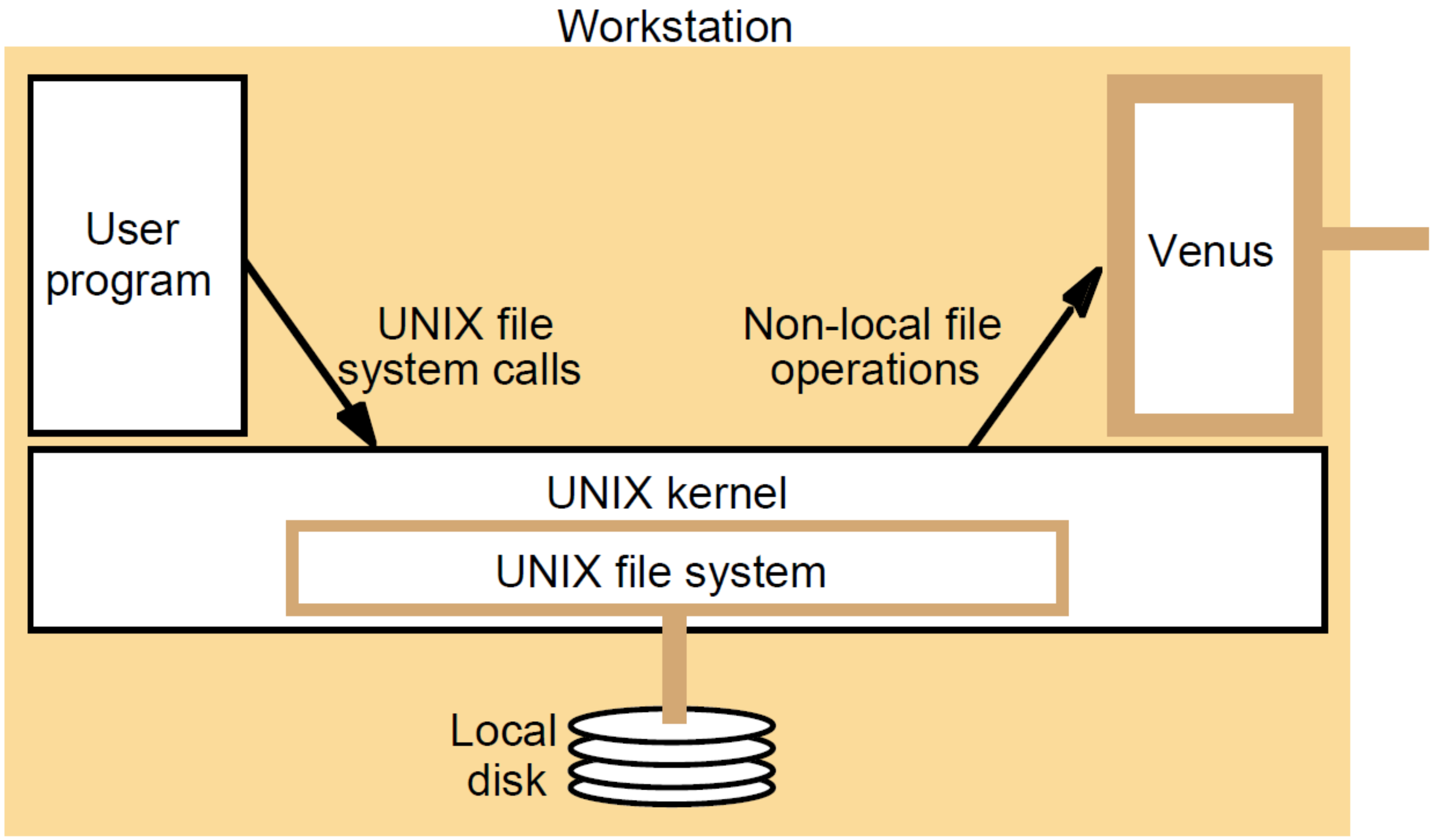
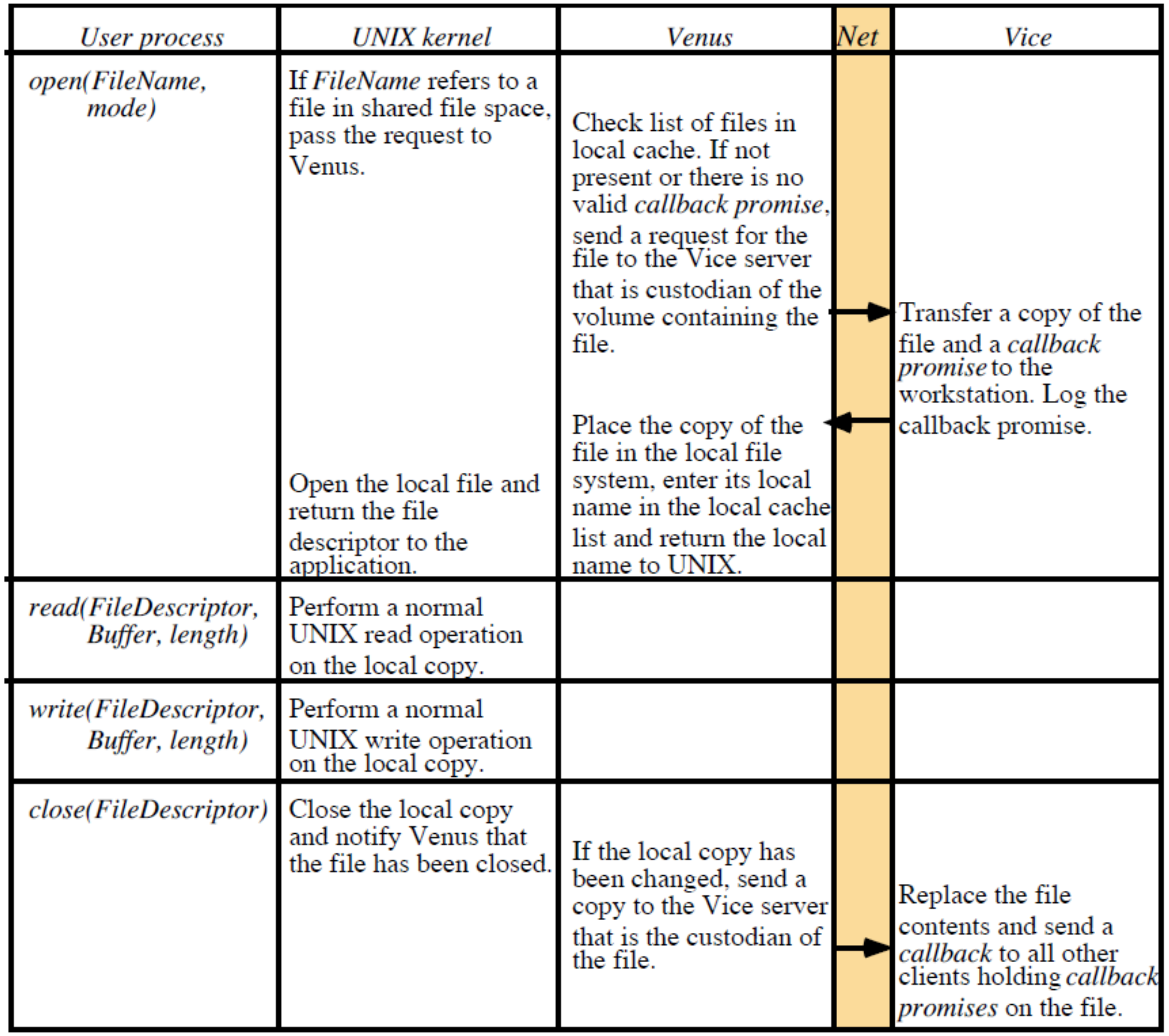


Figure 8.14
Implementation of file system calls in AFS



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 server 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.

Summary

- When designing a distributed file system, ask yourself:
  - What is the size distribution of files?
  - What are the relative and absolute frequencies of different file operations?
  - How often does the data in the file change?
  - How often do users share files for reading and for writing?
  - Does the type of a file substantially influence these properties?
- If you have needs different from standard, build your own. E.g., Google File System:
  - Big files, rarely deleted or shrinked
  - Files are appenended or read, rearly created or modified
  - Files are divided in chunks with unique IDs and replicated (nodes are not reliable)
  - *Masterservers* store metadata and grant leases (with timeouts) to access chunks on *Chunkservers*
  - Atomicity guaranteed via coordination among all chunkservers (one acts as primary)