Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.

Viewing: These slides must be viewed in slide show mode.

Distributed Systems Course

Distributed File Systems

Chapter 2 Revision: Failure model

Chapter 8:

8.1 Introduction

8.2 File service architecture

8.3 Sun Network File System (NFS)

[8.4 Andrew File System (personal study)]

8.5 Recent advances

8.6 Summary

Learning objectives

- Understand the requirements that affect the design of distributed services
- NFS: understand how a relatively simple, widely-used service is designed
  - Obtain a knowledge of file systems, both local and networked
  - Caching as an essential design technique
  - Remote interfaces are not the same as APIs
  - Security requires special consideration
- Recent advances: appreciate the ongoing research that often leads to major advances

*

Chapter 2 Revision: Failure model

Figure 2.11

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.

*

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.

*

Storage systems and their properties

Figure 8.1

Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate consistency
X - no automatic consistency

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system

*

What is a file system? 1

- 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? ...

*

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? 3

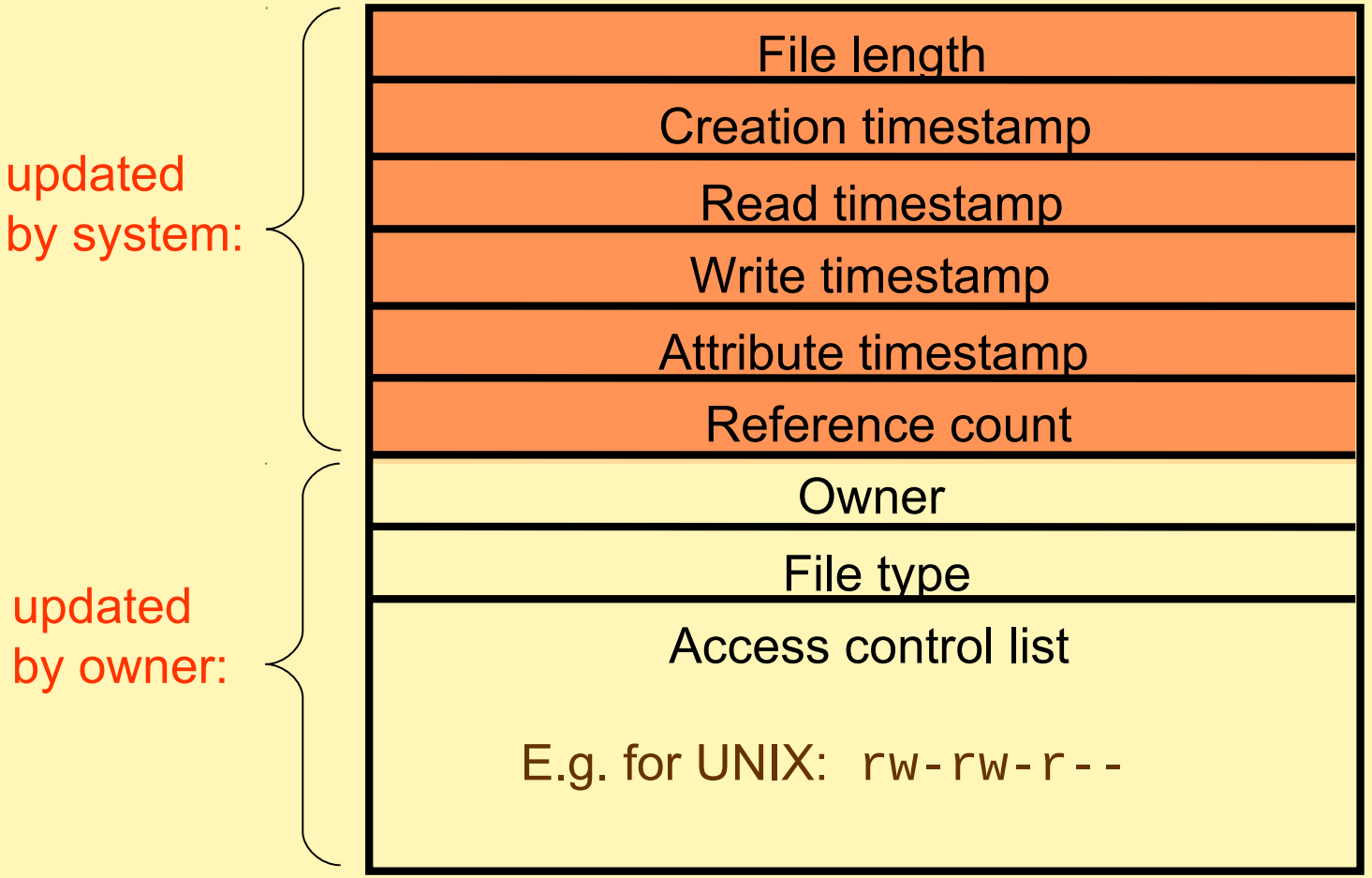
Figure 8.2 File system modules

Directory module:	relates file names to file IDs

*

What is a file system?

Figure 8.3 File attribute record structure



*

File service requirements

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

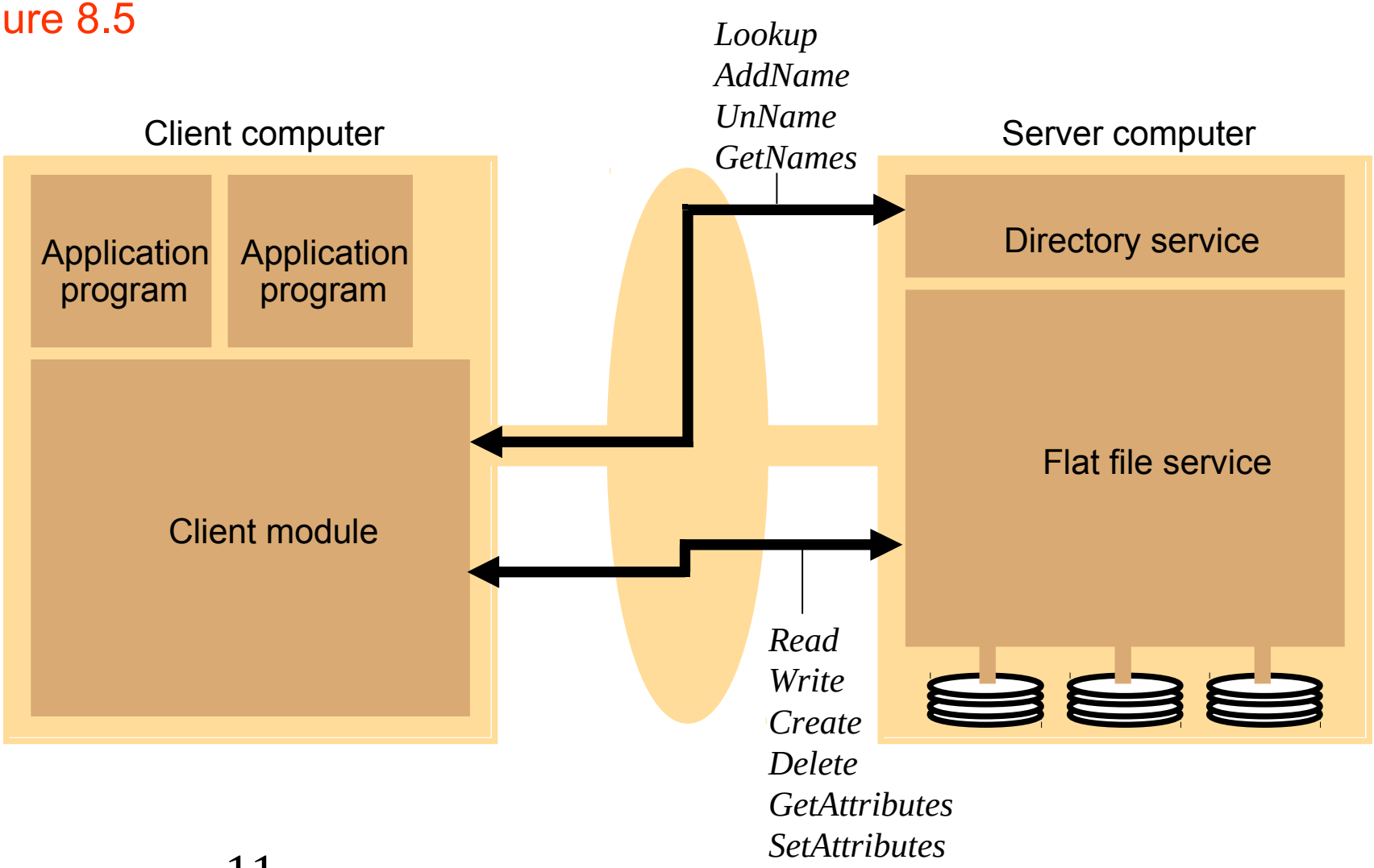
Efficiency

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

*

Model file service architecture

Figure 8.5



*

Server operations for the model file service

Figures 8.6 and 8.7

Flat file service

Read(FileId, i, n) -> Data

position of first byte

Write(FileId, i, Data)

position of first byte

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, File)

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.

*

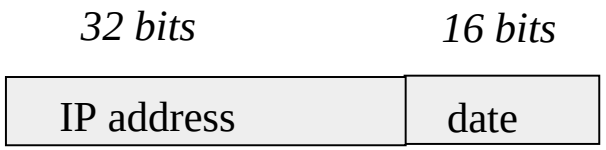
File Group

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*

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:



*

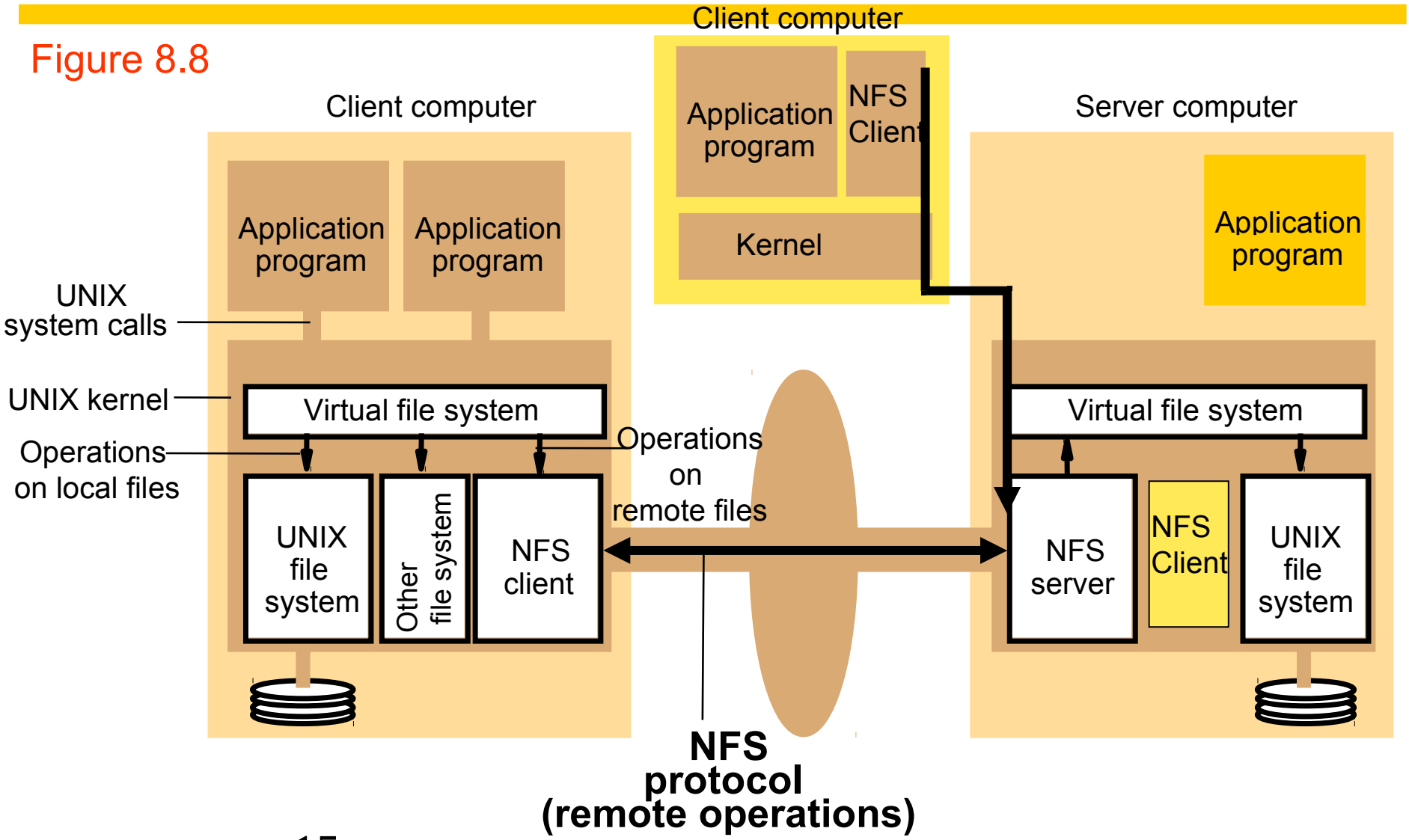
Case Study: Sun NFS

- 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

*

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:

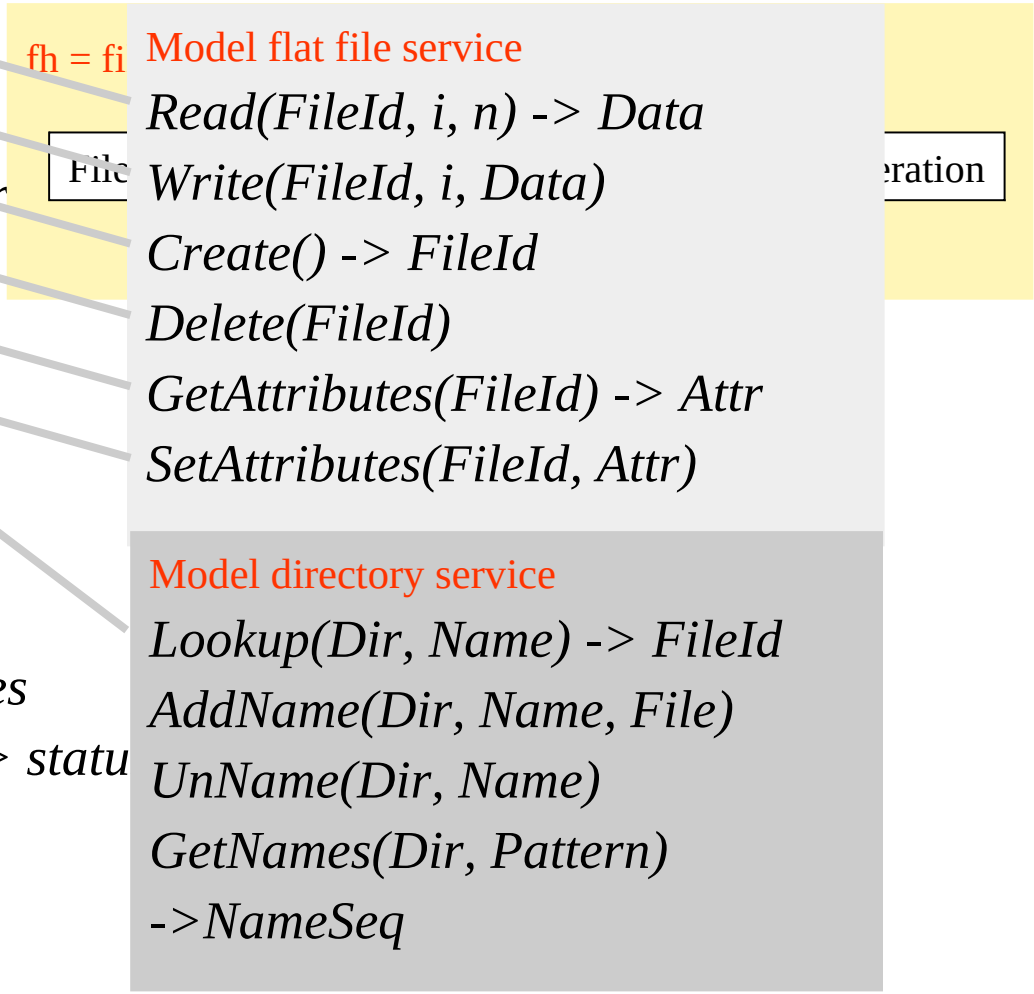
- Binary code compatible - no need to recompile applications
  - *Standard system calls that access remote files can be routed through the NFS client module by the kernel*
- Shared cache of recently-used blocks at client
- Kernel-level server can access i-nodes and file blocks directly
  - *but a privileged (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) -> statu*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*



*

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
  - Kerberos is described in Chapter 7. Integration of NFS with Kerberos is covered later in this chapter.

*

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:

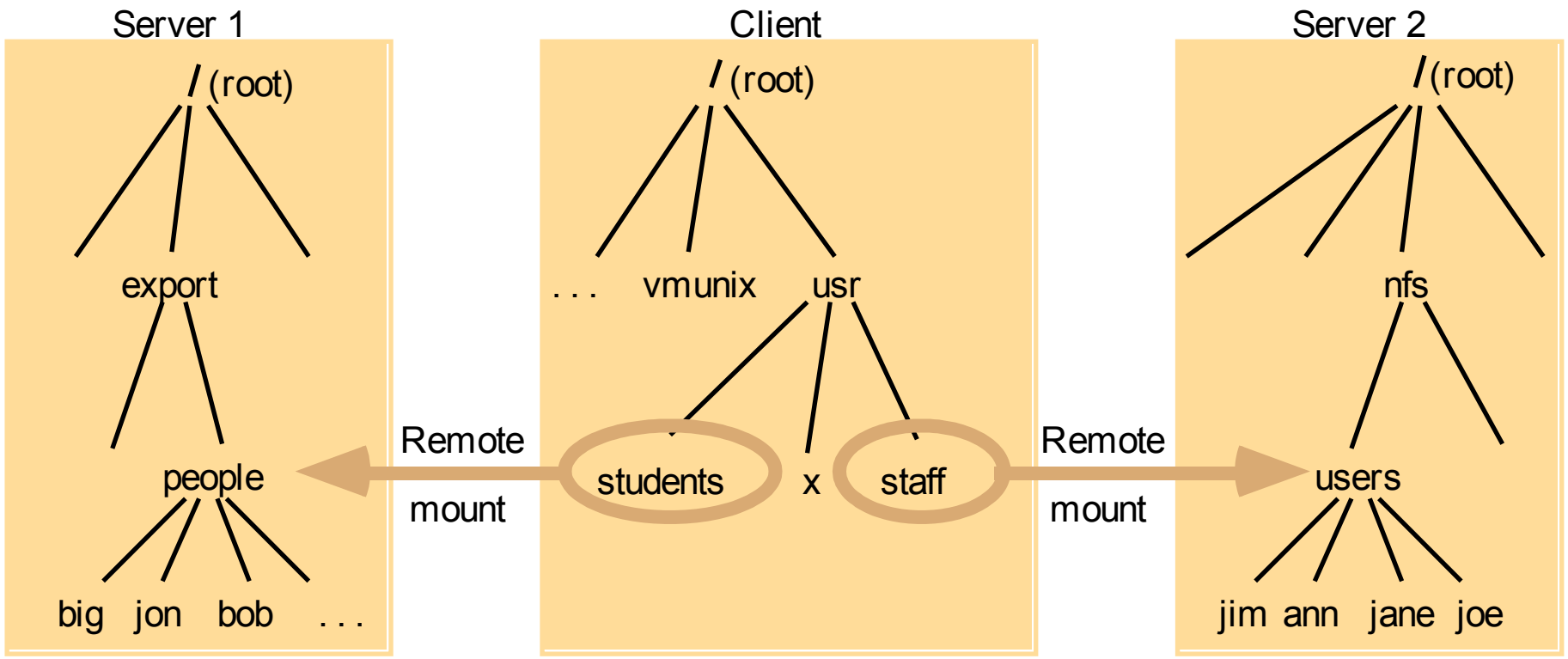
  < IP address, port number, file handle>

- *Hard* versus *soft* mounts

*

Local and remote file systems accessible on an NFS client

Figure 8.10



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.

*

Kerberized NFS

- Kerberos protocol is too costly to apply on each file access request
- Kerberos is used in the mount service:
  - to authenticate the user's identity
  - User's UserID and GroupID are stored at the server with the client's IP address
- For each file request:
  - The UserID and GroupID sent must match those stored at the server
  - IP addresses must also match
- This approach has some problems
  - can't accommodate multiple users sharing the same client computer
  - all remote filestores must be mounted each time a user logs in

*

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

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

*

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.
- More recent 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 (*See section 1.4.7*):

  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, see Chapter 14.

cont'd *

NFS summary 2

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.

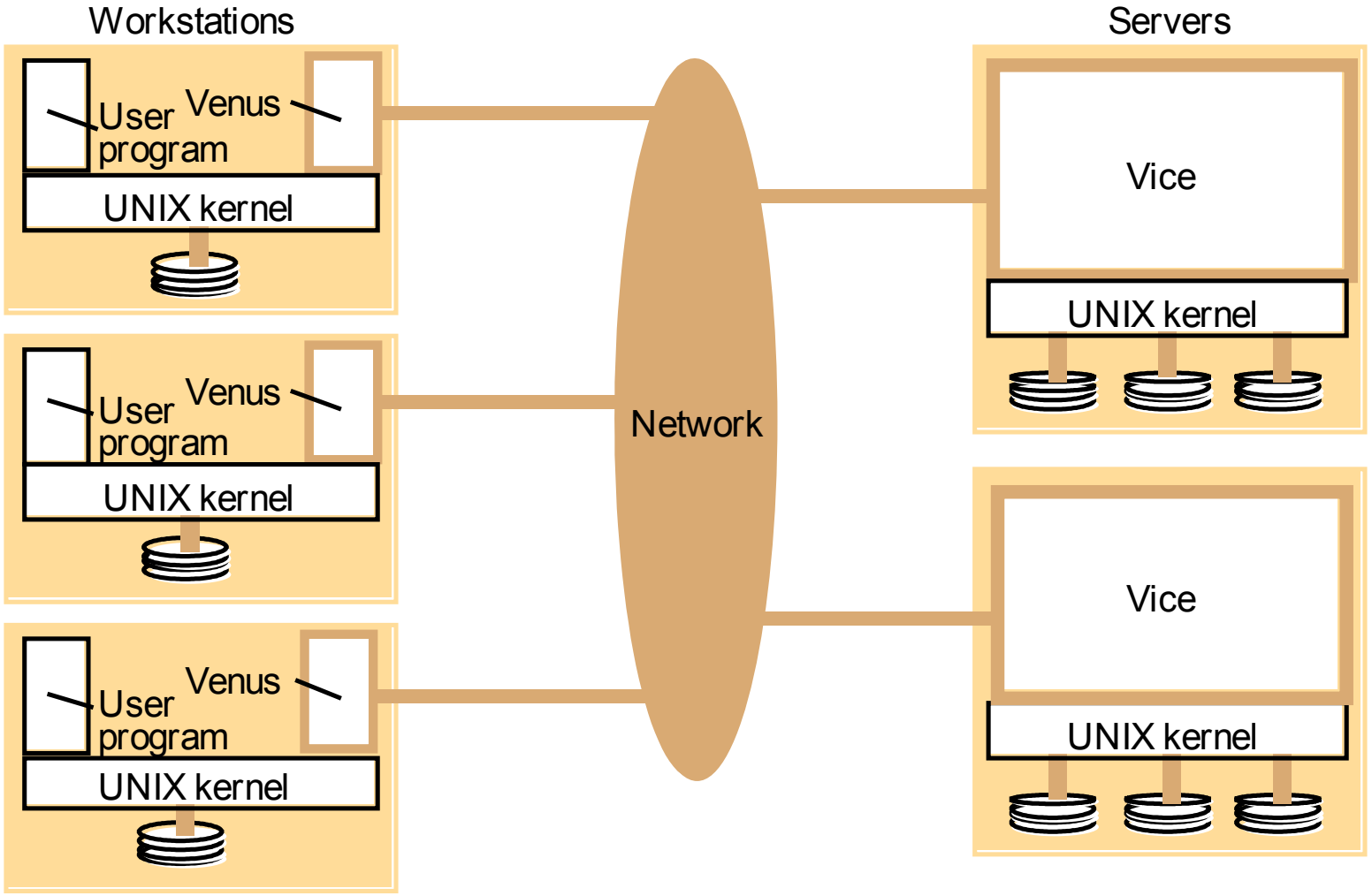
Performance: *Good*; multiprocessor servers achieve very high performance, but for a single filesystem it's not possible to go beyond the throughput of a multiprocessor server.

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).

*

Figure 8.11

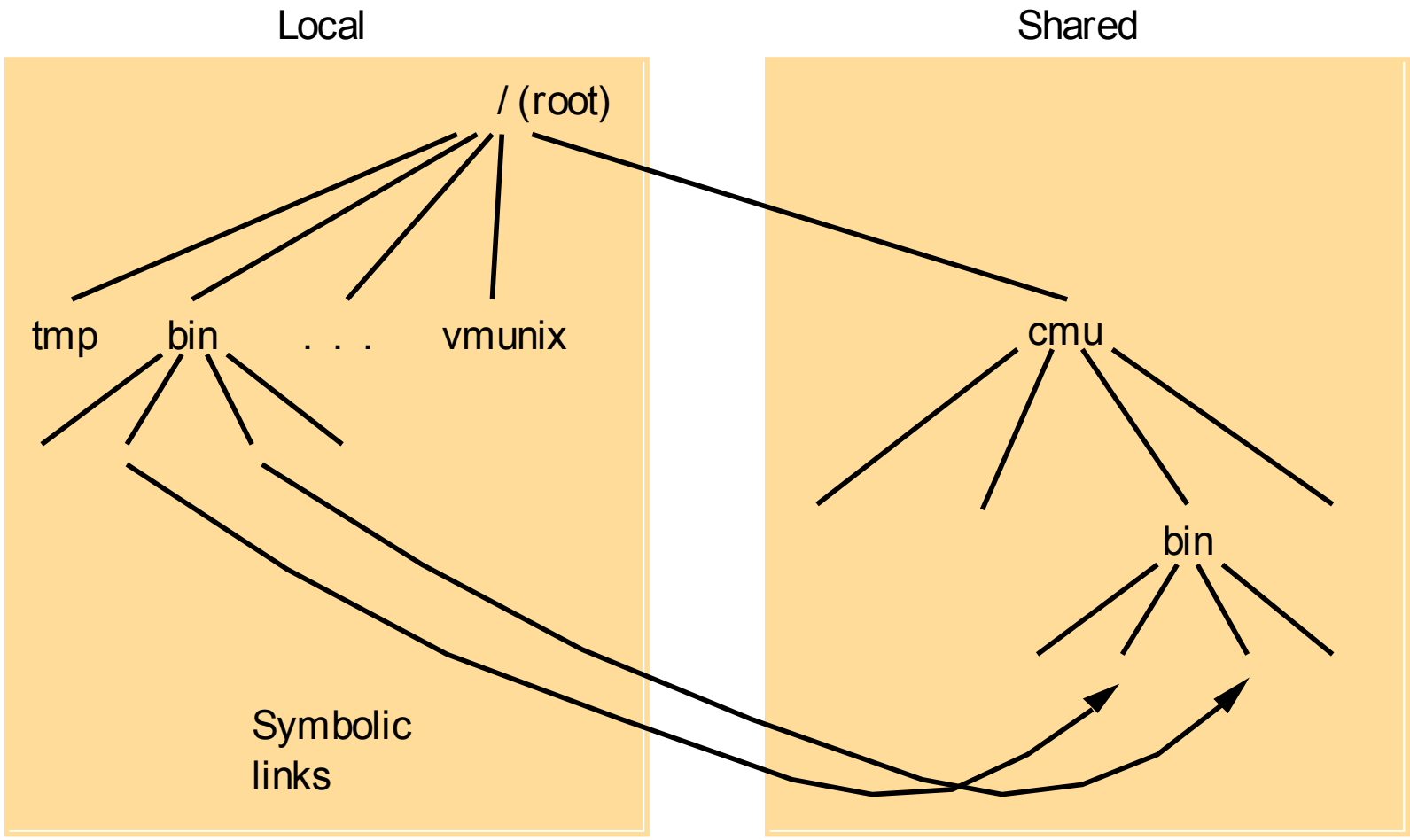
Distribution of processes in the Andrew File System



*

Figure 8.12

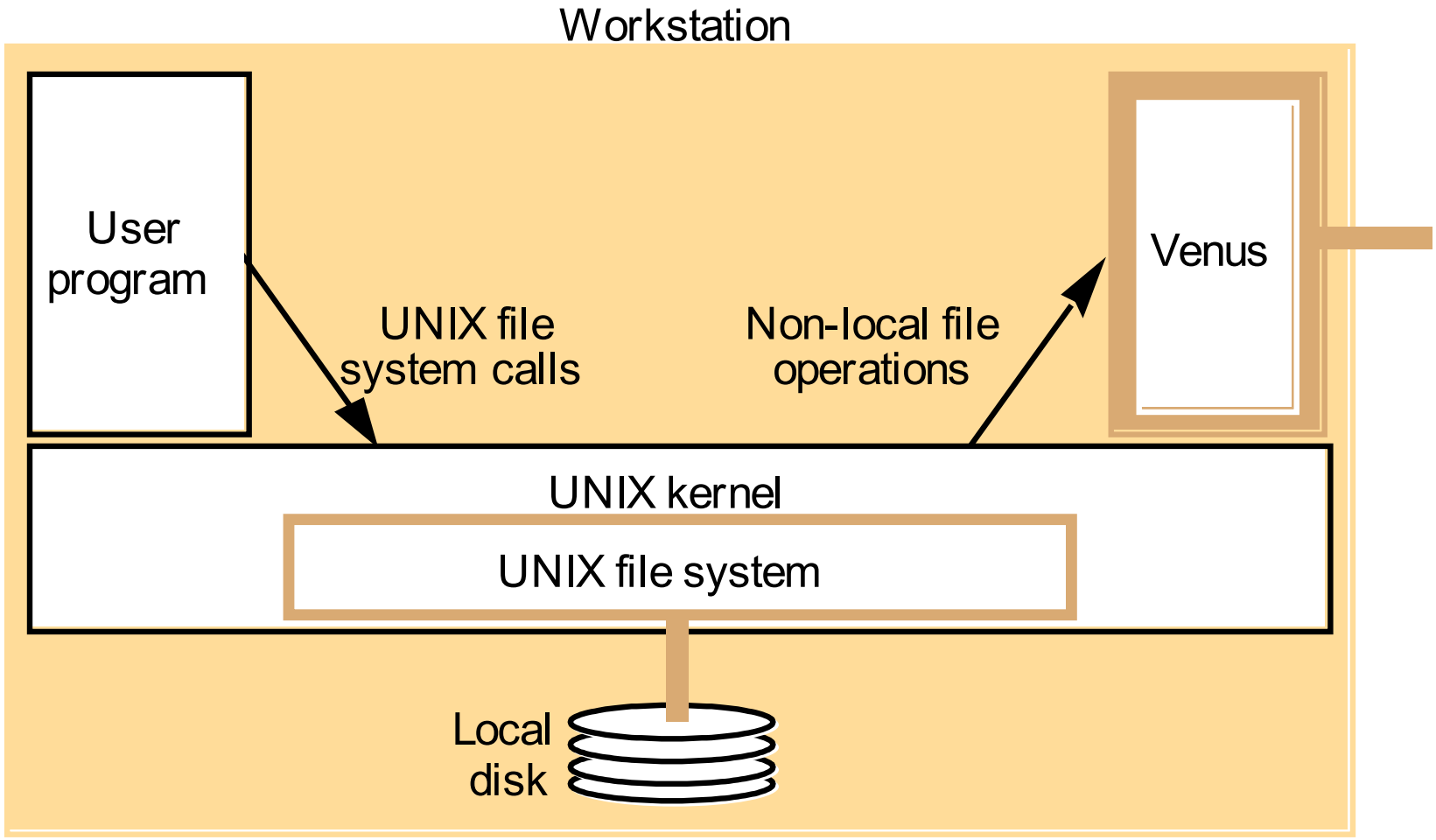
File name space seen by clients of AFS



*

Figure 8.13

System call interception in AFS



*

Figure 8.14

Implementation of file system calls in AFS

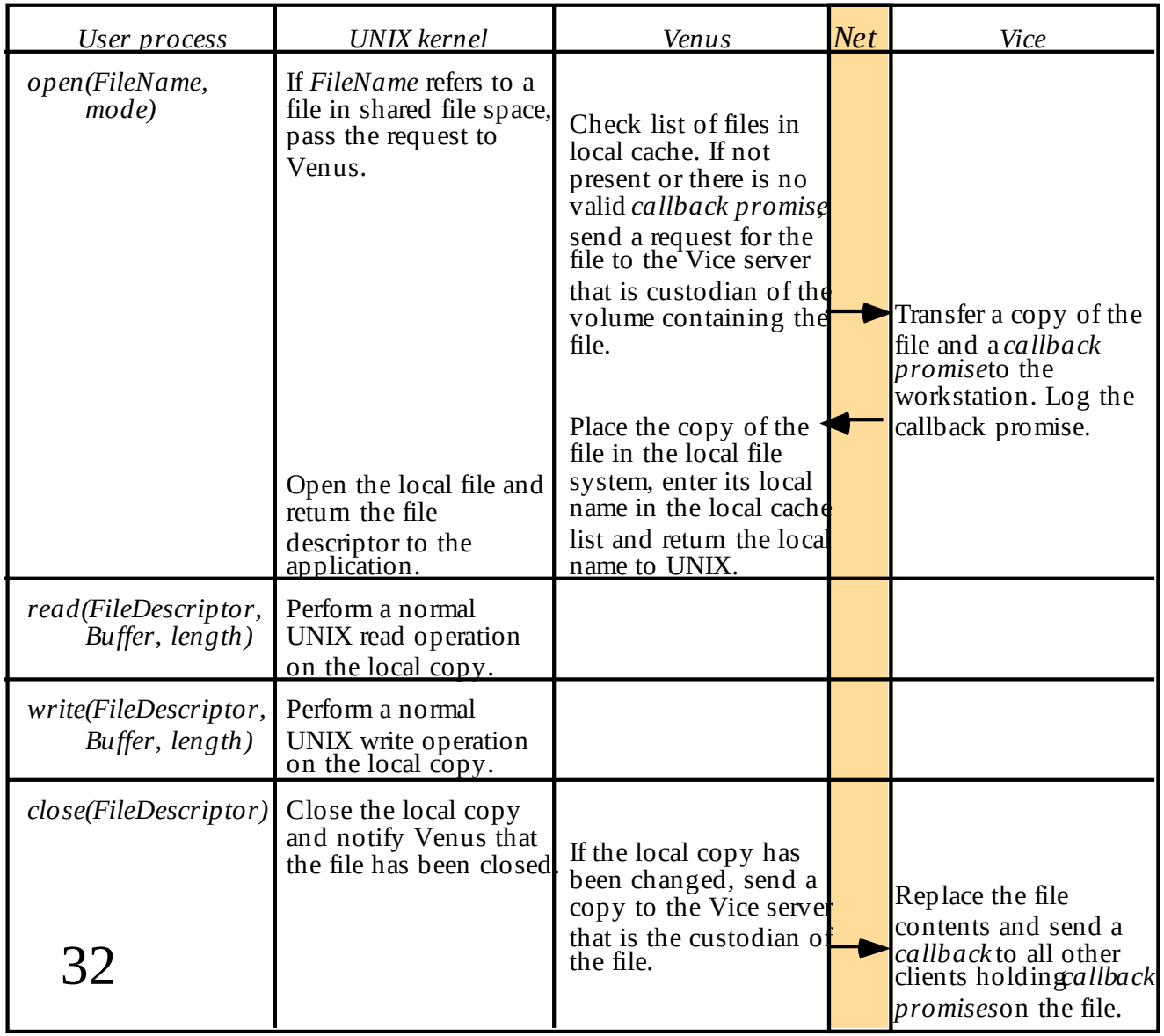


Figure 8.15

The main components of the 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.

*

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.

*

New design approaches

1

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)

Peer-to-peer systems: Napster, OceanStore (UCB), Farsite (MSR), Publius (AT&T research) - see web for documentation on these very recent systems

*

New design approaches 2

- Replicated read-write files
  - High availability
  - Disconnected working
    - *re-integration after disconnection is a major problem if conflicting updates have ocurred*
  - Examples:
    - *Bayou system (Section 14.4.2)*
    - *Coda system (Section 14.4.3)*

*

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
- Future requirements:
  - support for mobile users, disconnected operation, automatic re-integration (Cf. Coda file system, Chapter 14)
  - support for data streaming and quality of service (Cf. Tiger file system, Chapter 15)

*