Module 5 - Distributed File Systems

File Systems

- File system
  - Operating System interface to disk storage
- File system attributes (Metadata)

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

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

Operations on Unix File System

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*.

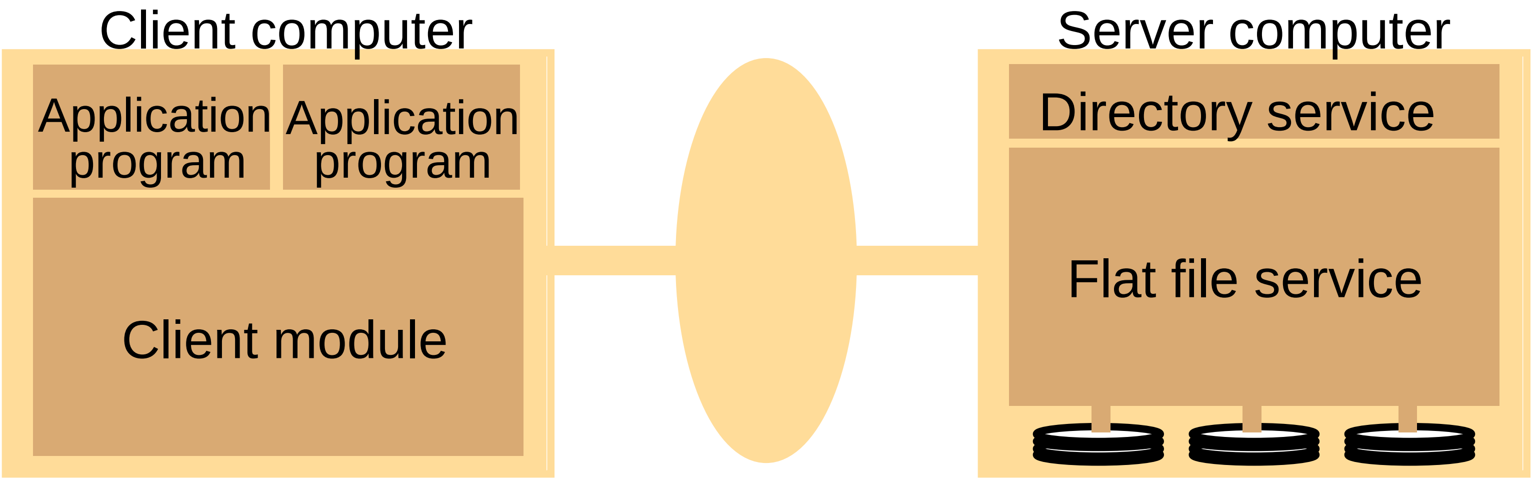
Distributed File System

- File system emulating non-distributed file system behaviour on a physically distributed set of files, usually within an intranet.
- Requirements
  - Transparency
    - Access transparency
    - Location transparency
    - Mobility transparency
    - Performance transparency
    - Scaling transparency
  - Allow concurrent access
  - Allow file replication
  - Tolerate hardware and operating system heterogeneity
  - Security
    - Access control
    - User authentication

Requirements (2)

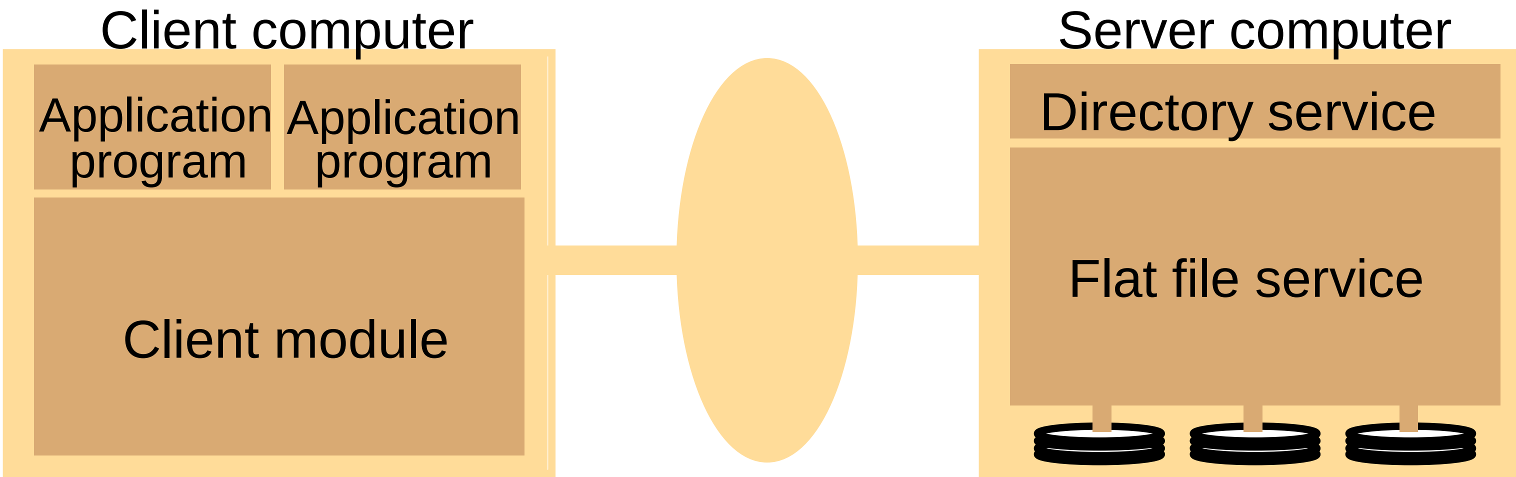
- Fault tolerance: continue to provide correct service in the presence of communication or server faults
  - → At-most-once semantics for file operations
  - → At-least-once semantics with a server protocol designed in terms of idempotent file operations
  - → Replication (stateless, so that servers can be restarted after failure)
- Consistency
  - → One-copy update semantics
    - – all clients see contents of file identically as if only one copy of file existed
    - – if caching is used: after an update operation, no program can observe a discrepancy between data in cache and stored data
- Efficiency
  - → Latency of file accesses
  - → Scalability (e.g., with increase of number of concurrent users)

Architecture



- Flat File Service
  - Performs file operations
  - Uses "unique file identifiers" (UFIDs) to refer to files
  - Flat file service interface
    - → RPC-based interface for performing file operations
    - → Not normally used by application level programs

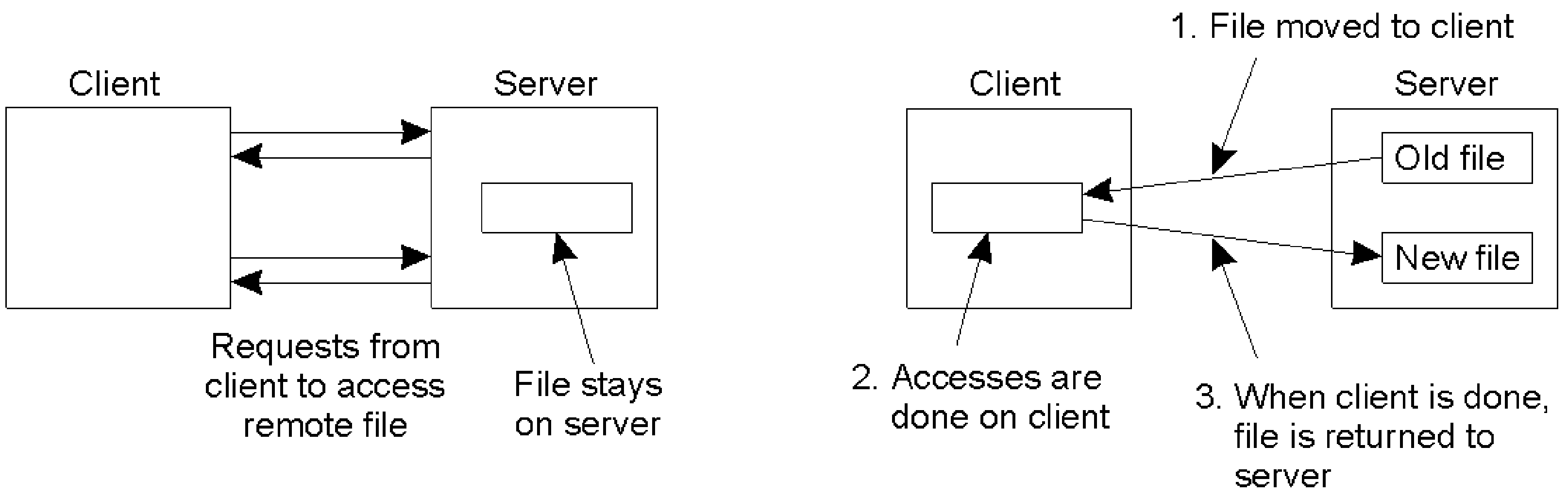
Architecture (2)



- Directory Service
  - Mapping of UFIDs to “text” file names, and vice versa
- Client Module
  - Provides API for file operations available to application program

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

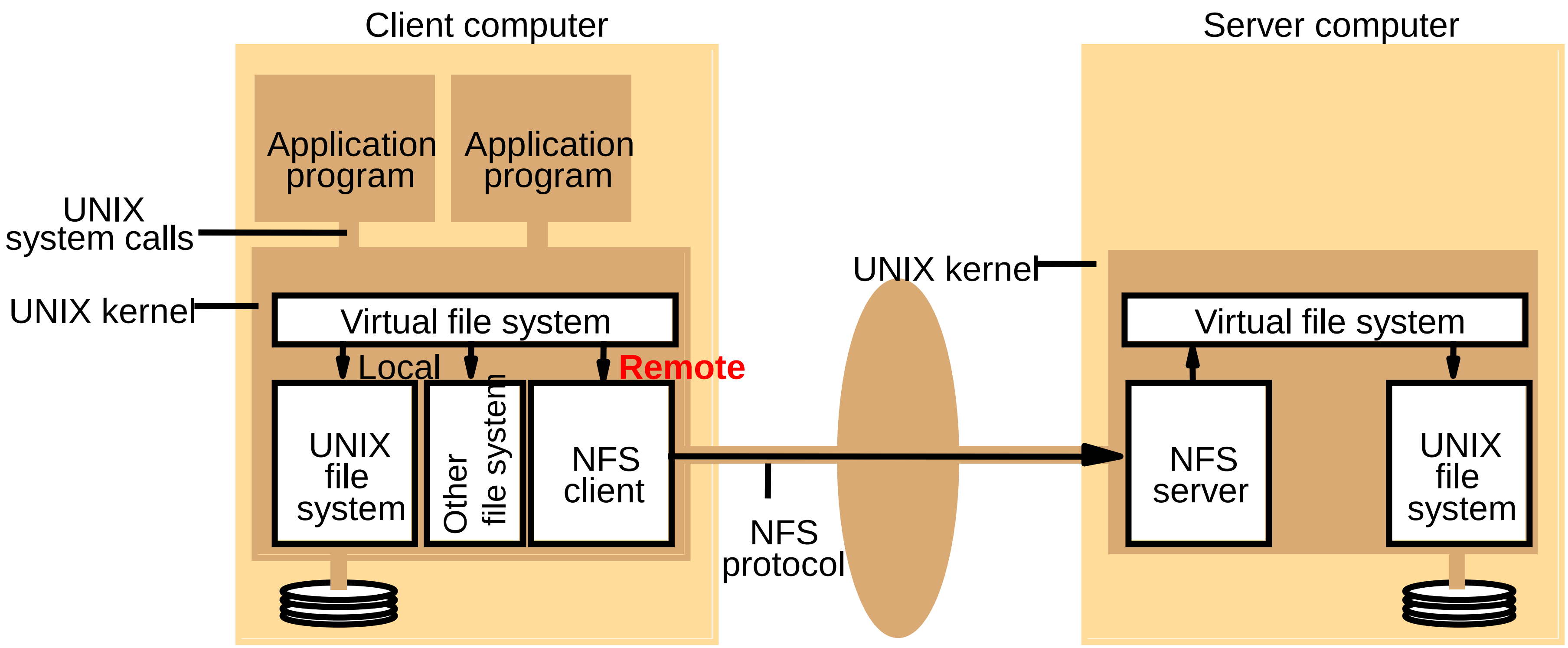
Distributed File Access Alternatives



Remote access model

Download/upload model

Sun Network File System



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

Architecture of NFS V.3

- Access transparency
  - No distinction between local and remote files
  - Virtual file system keeps track of locally and remotely available file systems
  - File identifiers: file handles
    - → File system identifier (unique number allocated at creation time)
    - → i-node number
    - → i-node generation number (because i- node- numbers are reused)

Selected NFS Operations

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*.

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

Selected NFS operations (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*.

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

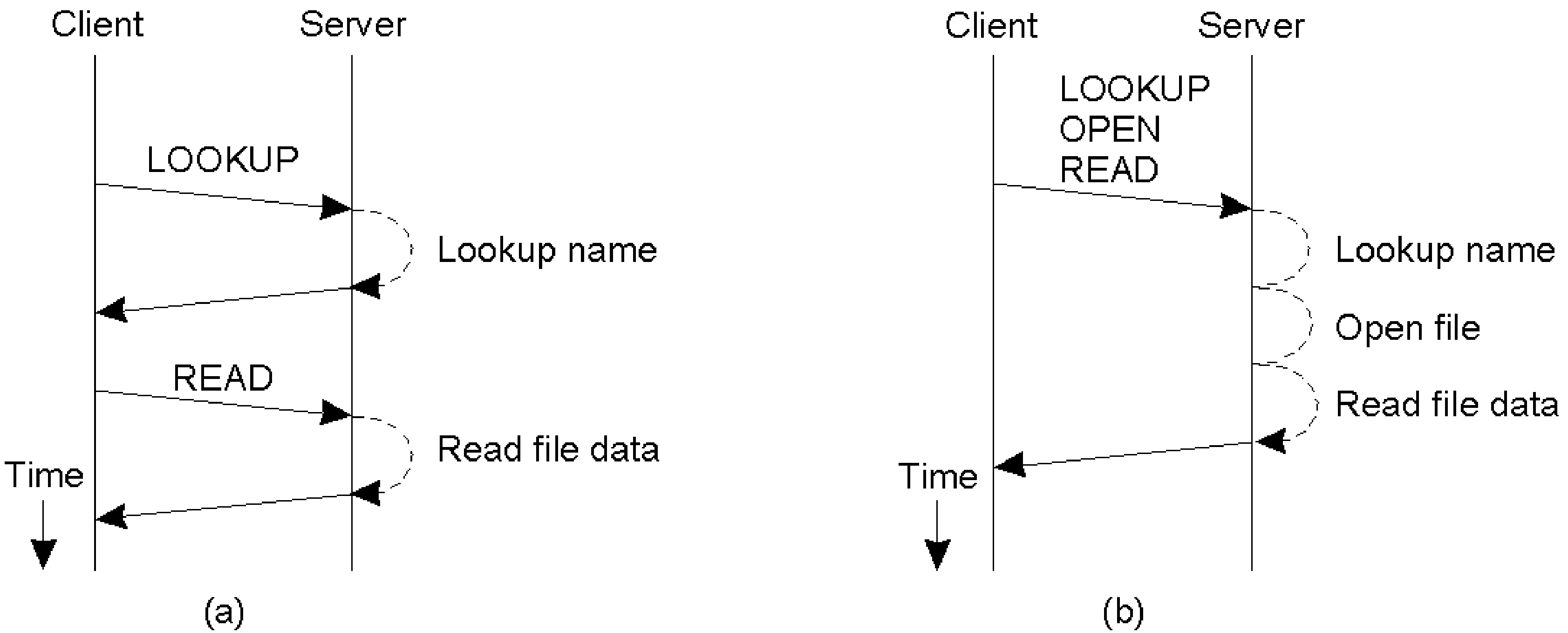
Changes in File System V3 to V4

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Read the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Access Control/Authentication

- NFS requests transmitted via Remote Procedure Calls (RPCs)
  - Clients send authentication information (user/group IDs)
  - Checked against access permissions in file attributes
- Potential security loophole
  - Any client may address RPC requests to server providing another client's identification information
  - Introduction of security mechanisms in NFS
    - → DES encryption of user identification information
    - → Kerberos authentication

Communication



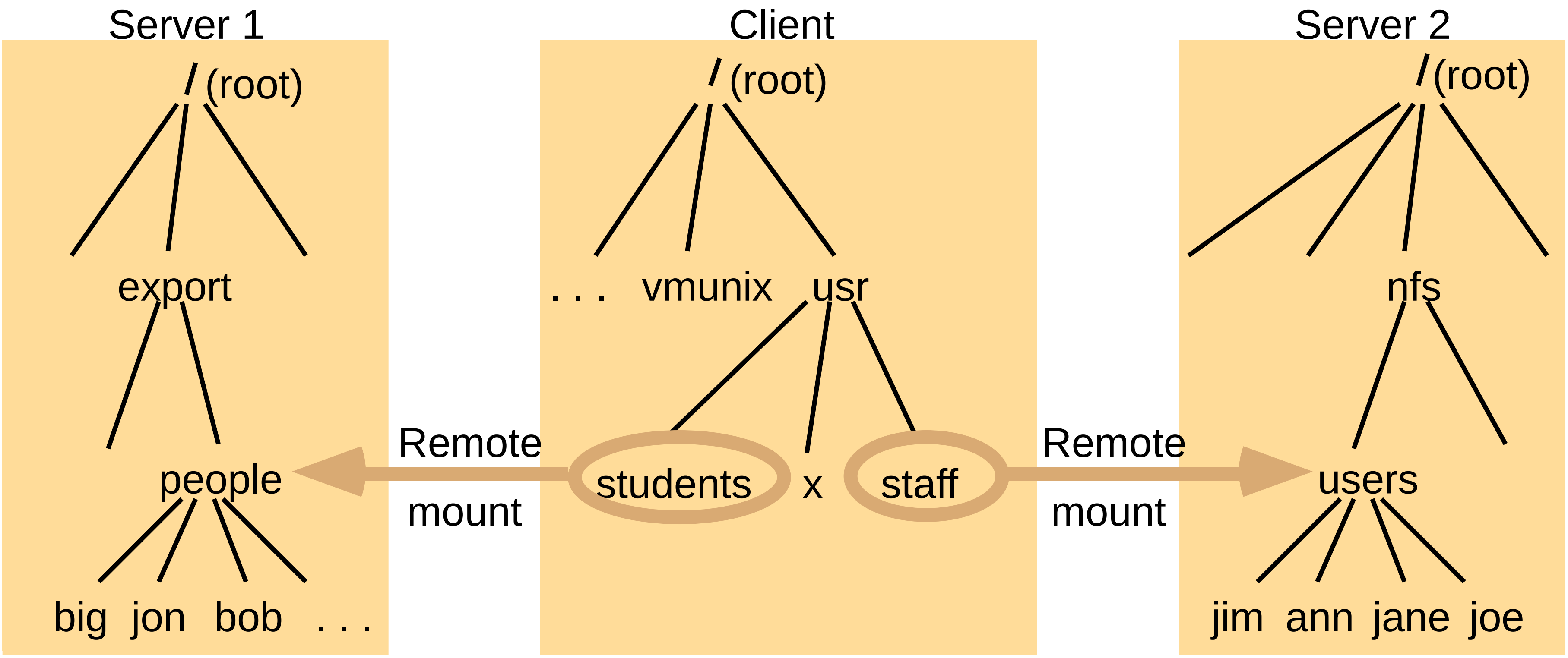
a) Reading data from a file in NFS version 3.

b) Reading data using a compound procedure in version 4.

Mounting of File Systems

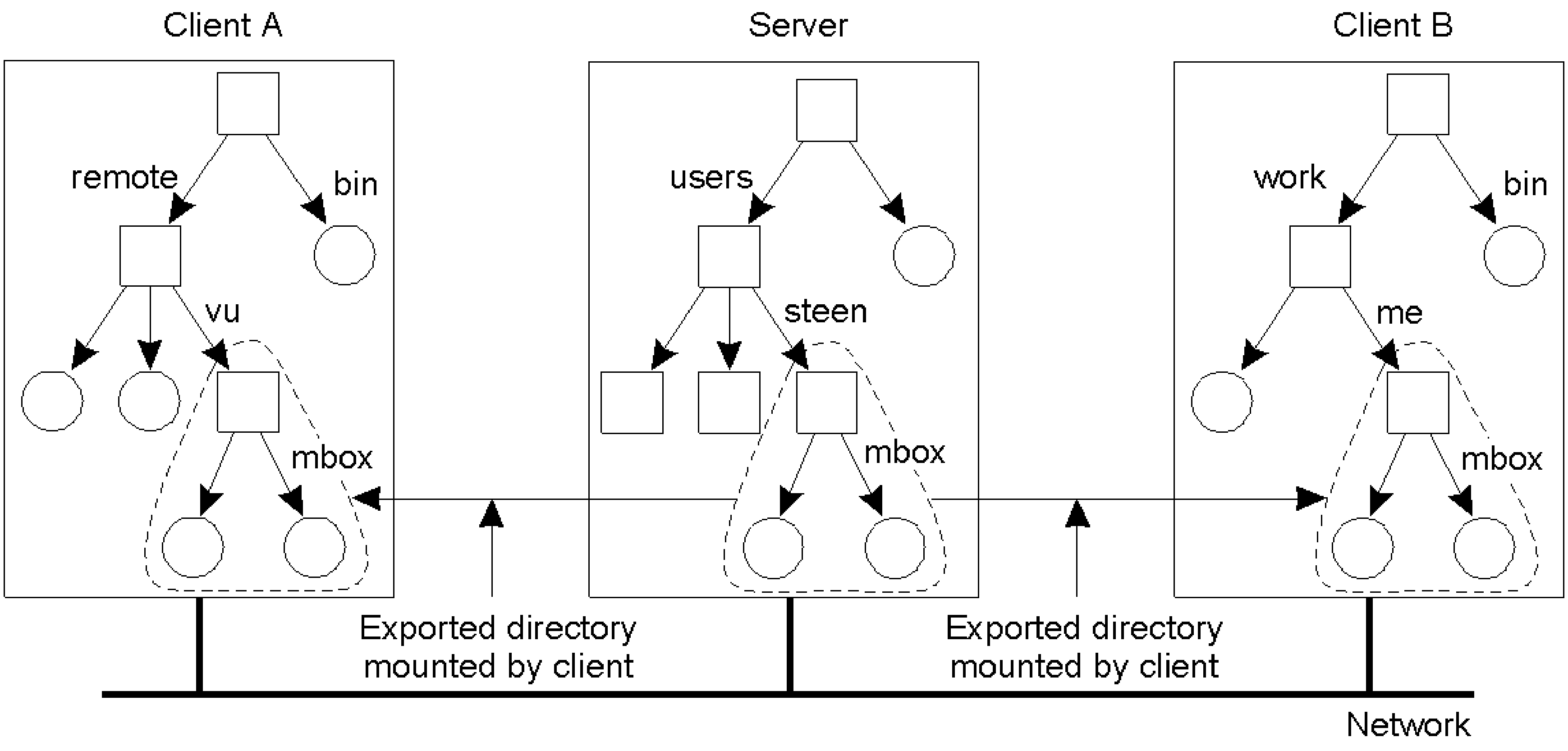
- Making remote file systems available to a local client, specifying remote host name and pathname
- Mount protocol (RPC-based)
  - Returns file handle for directory name given in request
  - Location (IP address and port number) and file handle are passed to Virtual File System and NFS client
- Hard-mounted (mostly used in practice)
  - User-level process suspended until operation completed
  - Application may not terminate gracefully in failure situations
- Soft-mounted
  - Error message returned by NFS client module to user-level process after small number of retries

Mounting Example



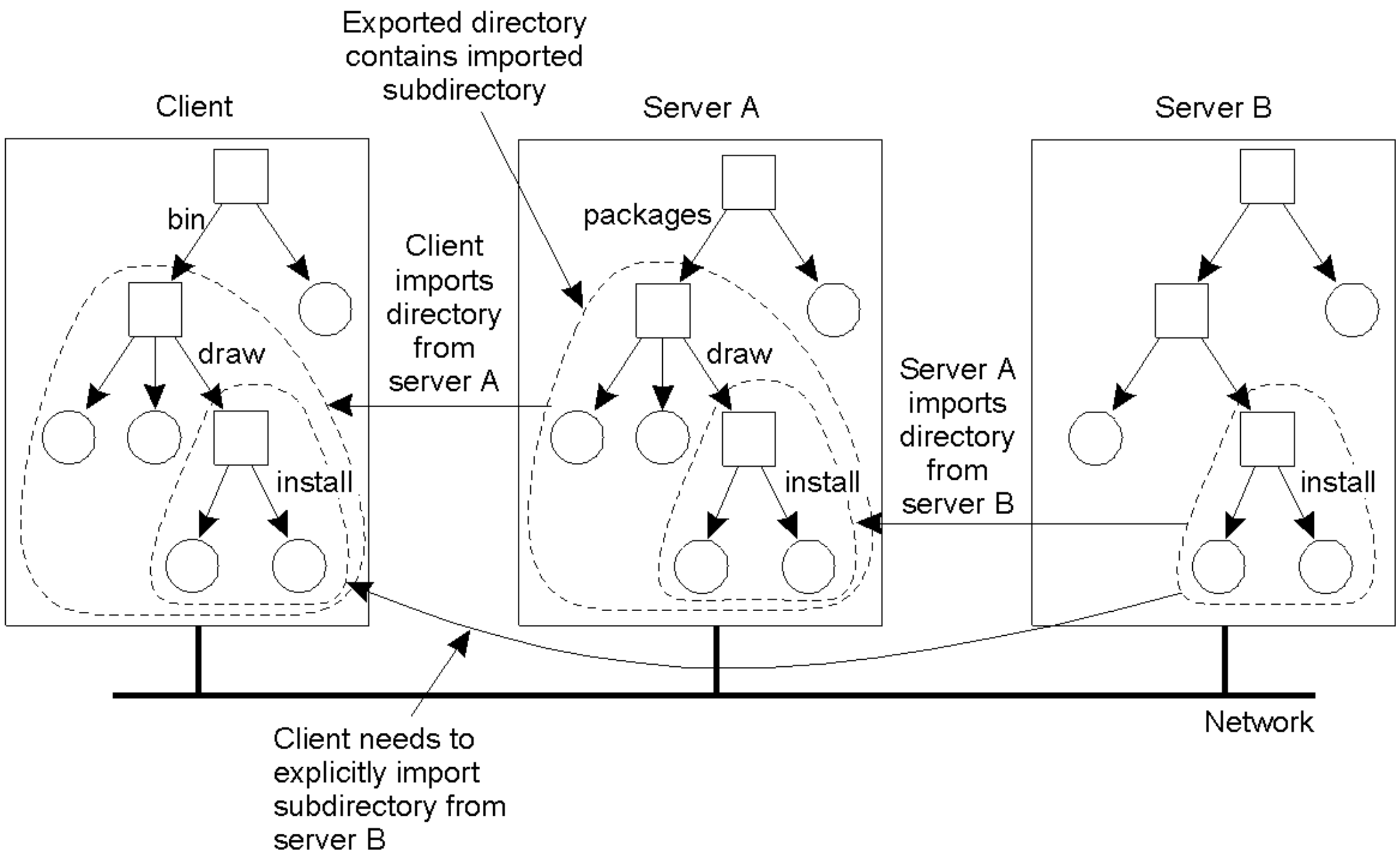
The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Naming (1)

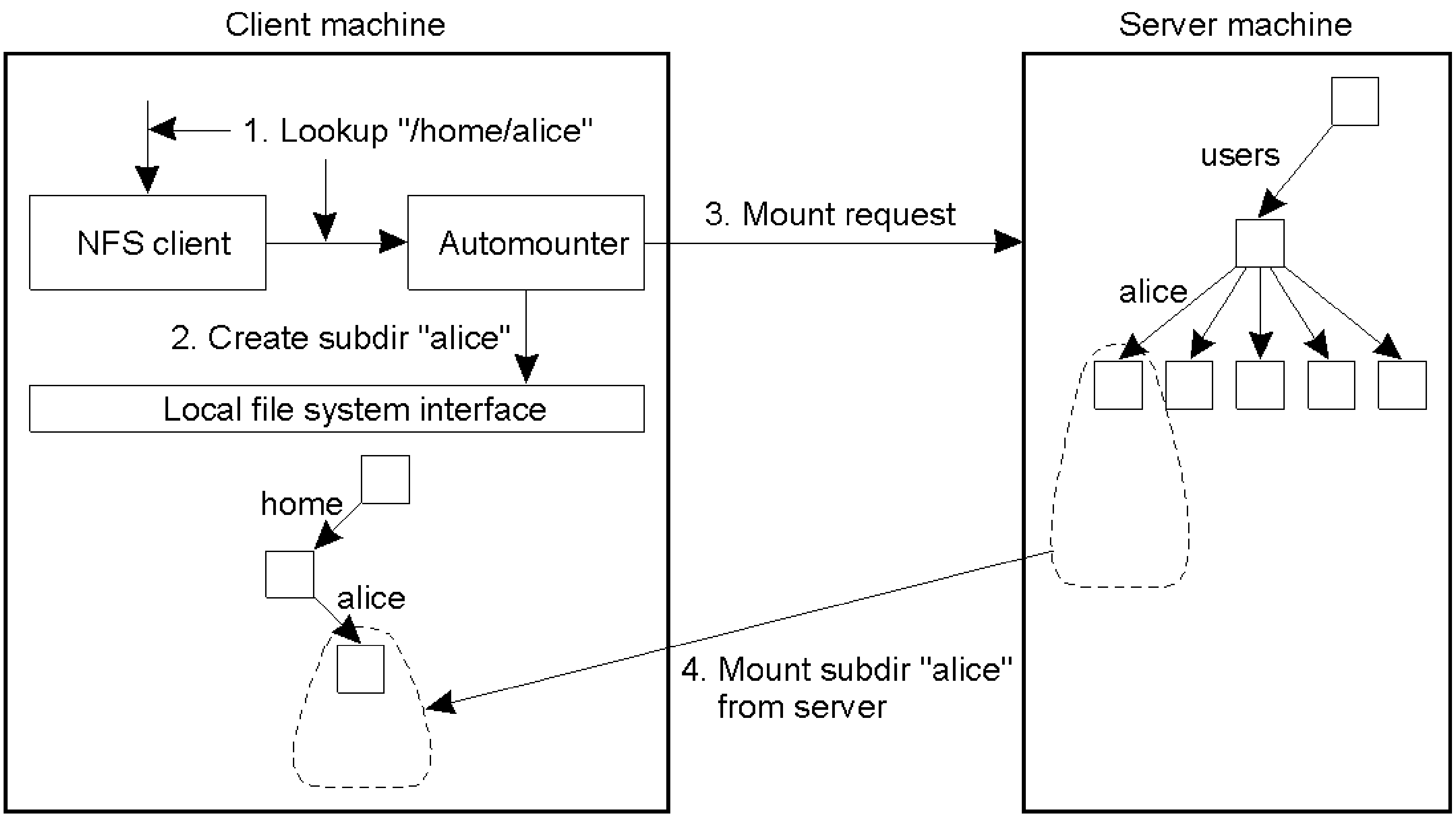

Client A
Server
Client B
remote
bin
vu
mbox
users
steen
mbox
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client
Network

Naming (2)

- Mounting nested directories from multiple servers in NFS.

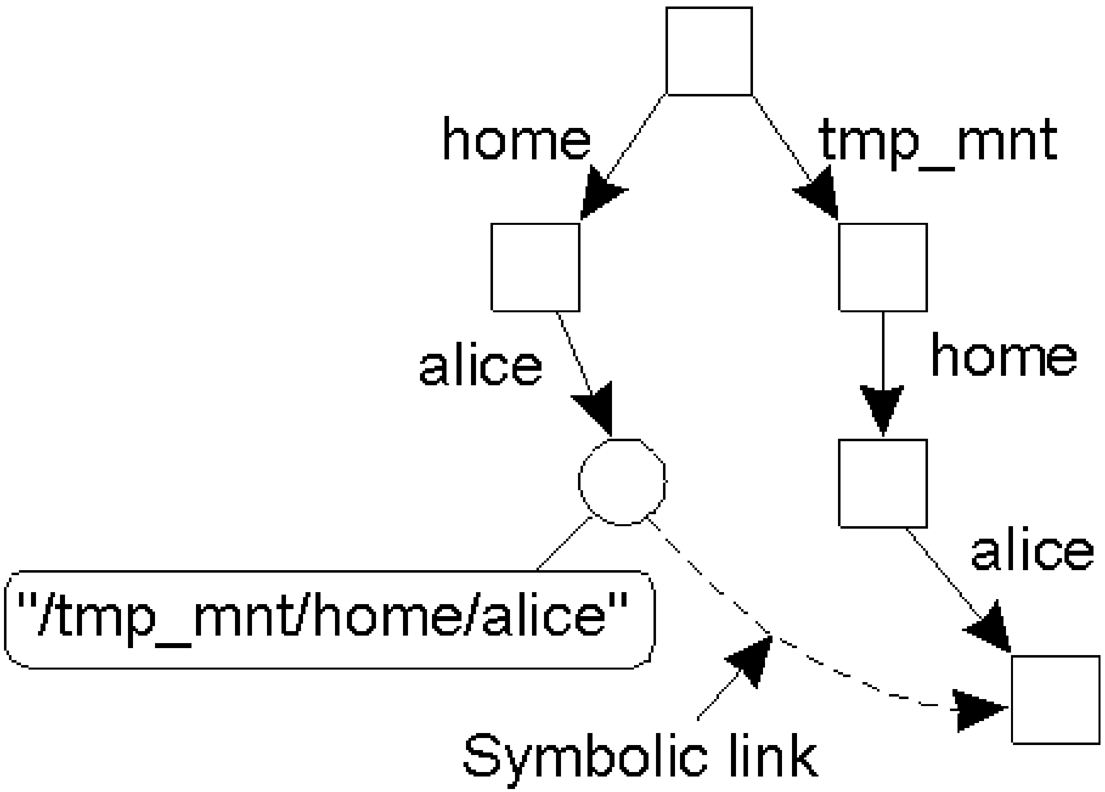


Automounting (1)



A simple automounter for NFS.

Automounting (2)



Using symbolic links with automounting.

Caching in Sun NFS

- Caching in server and client indispensable to achieve necessary performance
- Server caching
  - Disk caching as in non-networked file systems
  - Read operations: unproblematic
  - Write operations: consistency problems
  - Write- through caching
    - → Store updated data in cache and written on disk before sending reply to client
    - → Relatively inefficient if frequent write operations occur
  - Commit operation
    - → Stored only in cache memory
    - → Write back to disk only when commit operation for file received

Caching in Sun NFS (2)

- Client caching
  - Caching of read , write, getattr, lookup and readdir operations
  - Potential inconsistency: the data cached in client may not be identical to the same data stored on the server
  - Timestamp-based scheme used in polling server about freshness of a data object (presumption of synchronized global time, e.g., through NTP)
    - → *Tc*: time cache entry was last validated
    - → $Tm_{client/server}$: time when block was last modified at the server as recorded by client/server
    - → *t*: freshness interval
    - → Freshness condition: at time *T*
      - $[(T - Tc) < t] \vee [Tm_{client} = Tm_{server}]$
      - if $(T - Tc) < t$ (can be determined without server access), then entry presumed to be valid
      - if not $(T - Tc) < t$, then Tm_{server} needs to be obtained by a getattr call
      - if $Tm_{client} = Tm_{server}$, then entry presumed valid; update its *Tc* to current time, else obtain data from server and update Tm_{client}

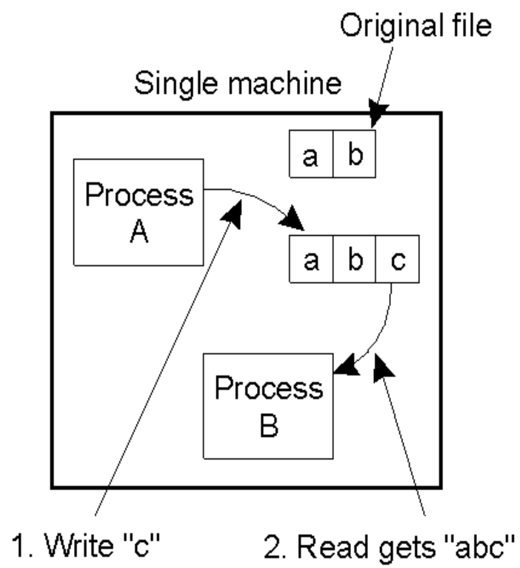
Caching in Sun NFS (3)

- Client caching - write operations
  - Mark modified cache page as "dirty" and schedule page to be flushed to server (asynchronously)
    - → Flush happens with closing of file, or when sync is issued,
    - → When asynchronous block input/output (bio) daemon is used and active
      - – when read, then read-ahead: when read occurs, bio daemon sends next file block
      - – when write, then bio daemon will send block asynchronously to server
    - → Bio daemons: performance improvement reducing probability that client blocks waiting for
      - – read operations to return, or
      - – write operations to be committed at the server

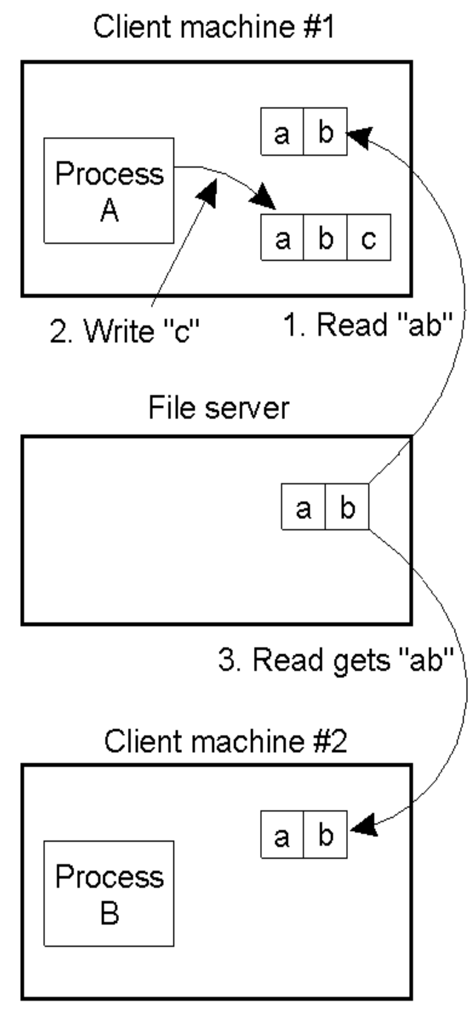
Semantics of File Sharing (1)

a) On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.

b) In a distributed system with caching, obsolete values may be returned.



(a)



(b)

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms

Semantics of File Sharing (2)

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

Four ways of dealing with the shared files in a distributed system.

File Locking in NFS (1)

Operation	Description
Lock	Creates a lock for a range of bytes
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the leas on a specified lock

NFS version 4 operations related to file locking.

File Locking in NFS (2)

	Current file denial state			
Request access	**NONE**	**READ**	**WRITE**	**BOTH**
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(a)

	Requested file denial state			
Current access state	**NONE**	**READ**	**WRITE**	**BOTH**
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(b)

The result of an *open* operation with share reservations in NFS.

a) When the client requests shared access given the current denial state.

b) When the client requests a denial state given the current file access state.

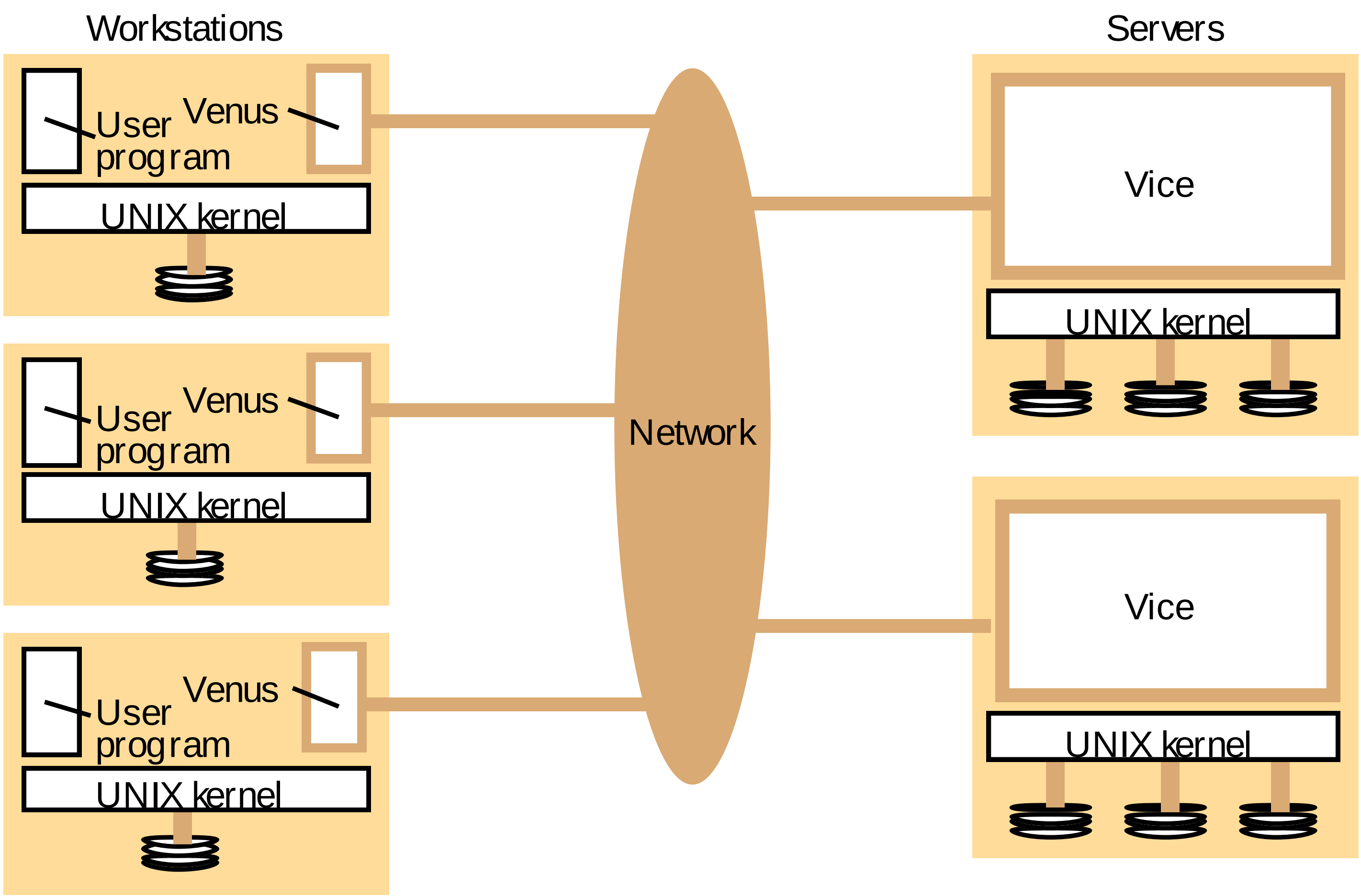
Andrew File System (AFS)

- Started as a joint effort of Carnegie Mellon University and IBM
- Today basis for DCE/DFS: the distributed file system included in the Open Software Foundations’ Distributed Computing Environment
- Some UNIX file system usage observations, as pertaining to caching
  - Infrequently updated shared files and local user files will remain valid for long periods of time (the latter because they are being updated on owners workstations)
  - Allocate large local disk cache, e. g., 100 Mbyte. This is sufficient for the establishment of a working set of the files used by one user and will ensure that files are still in this cache when needed next time

Andrew File System (AFS)

- Some UNIX file system usage observations, as pertaining to caching (continued)
  - Assumptions about typical file accesses (based on empirical evidence)
    - → usually small files, less than 10 Kbytes
    - → reads much more common than writes (appr. 6: 1)
    - → usually sequential access, random access not frequently found
    - → user-locality: most files are used by only one user
    - → burstiness of file references: once file has been used, it will be used in the nearer future with high probability
- Design decisions for AFS
  - Whole-file serving: entire contents of directories and files transferred from server to client (AFS-3: in chunks of 64 Kbytes)
  - Whole file caching: when file transferred to client it will be stored on that client's local disk

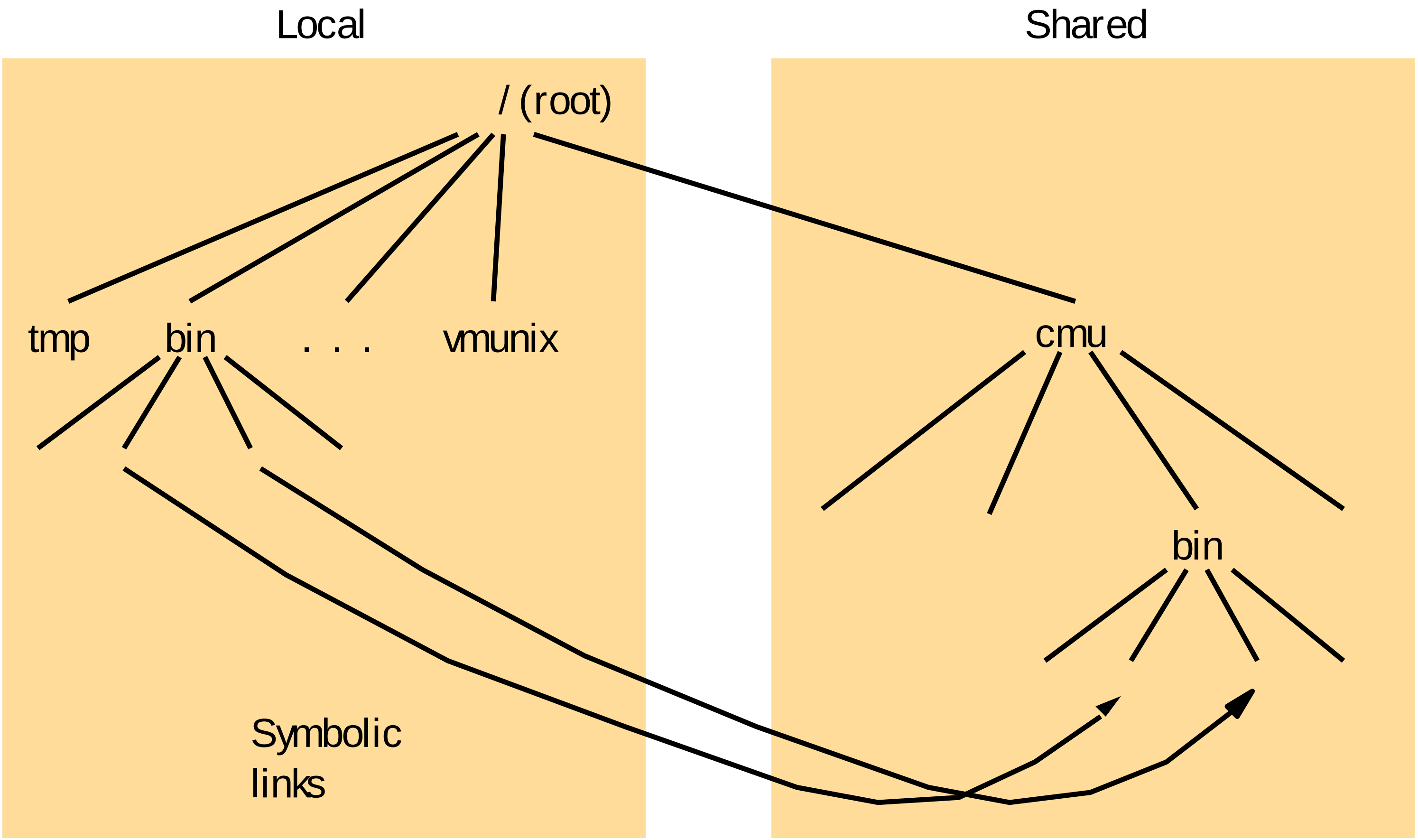
AFS Architecture



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

File Name Space

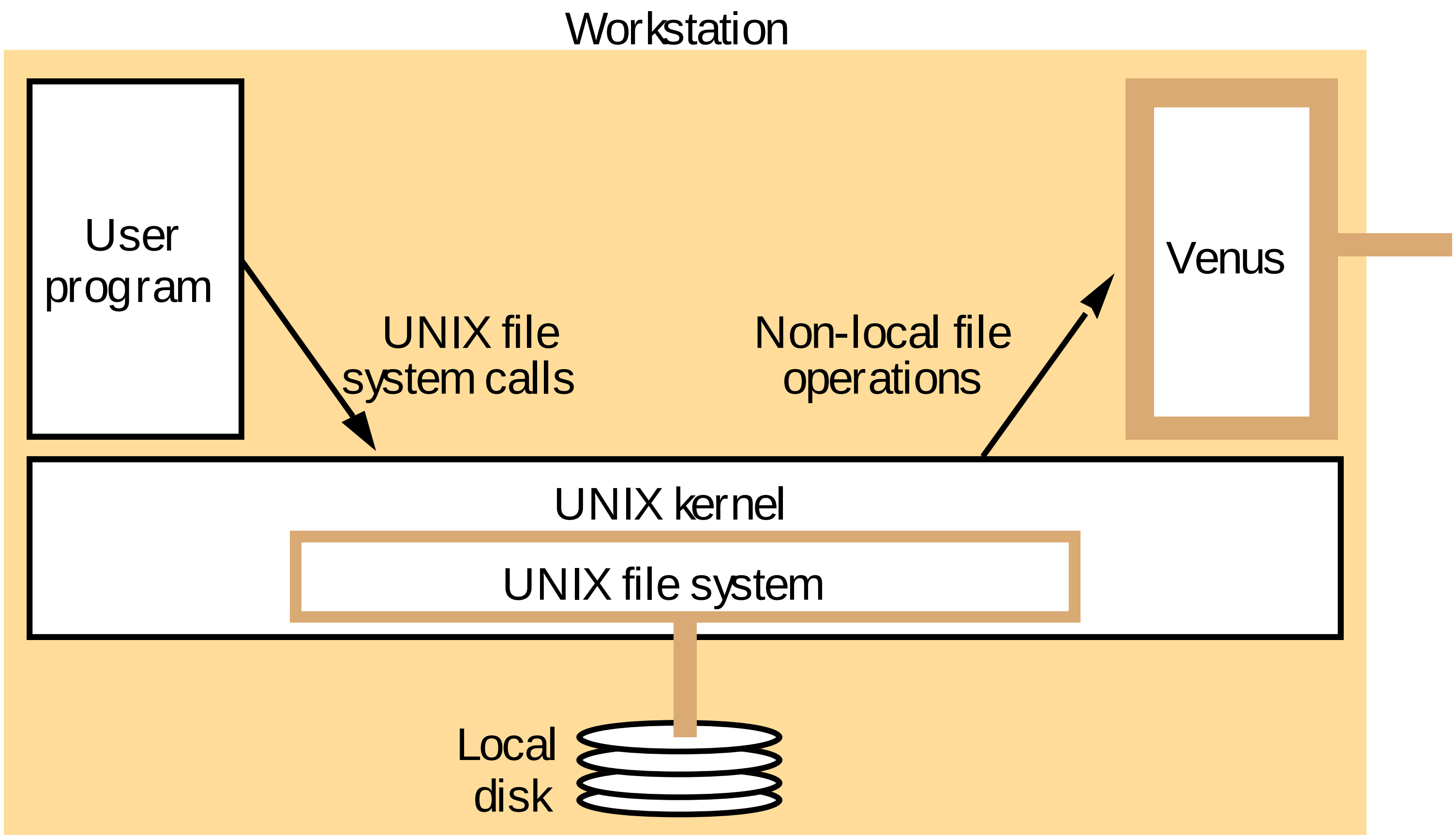
Seen by clients of AFS



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

AFS System Call Intercept

Handling by Venus



Implementation of System Calls

Callbacks and Callback promises

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.	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.	→	
				Transfer a copy of the file and a ***callback promise*** to the workstation. Log the callback promise.
		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.	←	
	Open the local file and return the file descriptor to the application.			

Implementation of System Calls

Callbacks and Callback promises

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
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.

Callback Mechanism

- Ensures that cached copies of files are updated when another client performs a close operation on that file
- Callback promise
  - Token stored with cached file
  - Status: valid or cancelled
- When server performs request to update file (e. g., following a close), then it sends callback to all Venus processes to which it has sent callback promise
  - RPC from server to Venus process
  - Venus process sets callback promise for local copy to cancelled
- Venus handling an open
  - Check whether local copy of file has valid callback promise
  - If canceled, fresh copy must be fetched from Vice server

Callback Mechanism (2)

- Restart of workstation after failure
  - Retain as many locally cached files as possible, but callbacks may have been missed
  - Venus sends cache validation request to the Vice server
    - → Contains file modification timestamp
    - → If timestamp is current, server sends valid and callback promise is reinstated with valid
    - → If timestamp not current, server sends cancelled
- Problem: communication link failures
    - → Callback must be renewed with above protocol before new open if a time T has lapsed since file was cached or callback promise was last validated
- Scalability
  - AFS callback mechanism scales well with increasing number of users
    - → Communication only when file has been updated
    - → In (AFS-1) timestamp approach: for each open
  - Since majority of files not accessed concurrently, and reads more frequent than writes, callback mechanism performs better

File Update Semantics

- To ensure strict one-copy update semantics: modification of cached file must be propagated to any other client caching this file before any client can access this file
  - Rather inefficient
- Callback mechanism is an approximation of one-copy semantics
- Guarantees of currency for files in AFS (version 1)
  - After successful open: latest(*F*, *S*)
    - → Current value of file *F* at client *C* is the same as the value at server *S*
  - After a failed open/close: failure(*S*)
    - → Open/close not performed at server
  - After successful close: updated(*F*, *S*)
    - → Client's value of *F* has been successfully propagated to *S*

File Update Semantics in AFSv2

- Vice keeps callback state information about Venus clients: which clients have received callback promises for which files
- Lists retained over server failures
- When callback message is lost due to communication link failure, an old version of a file may be opened after it has been updated by another client
- Limited by time T after which client validates callback promise (typically, T=10 minutes)
- Currency guarantees
  - After successful open:
    - → latest(F, S, 0)
      - – copy of F as seen by client is no more than 0 seconds out of date
    - → or (lostCallback(S, T) and inCache(F) and latest(F, S, T))
      - – callback message has been lost in the last T time units
      - – the file F was in the cache before open was attempted
      - – and copy is no more than T time units out of date

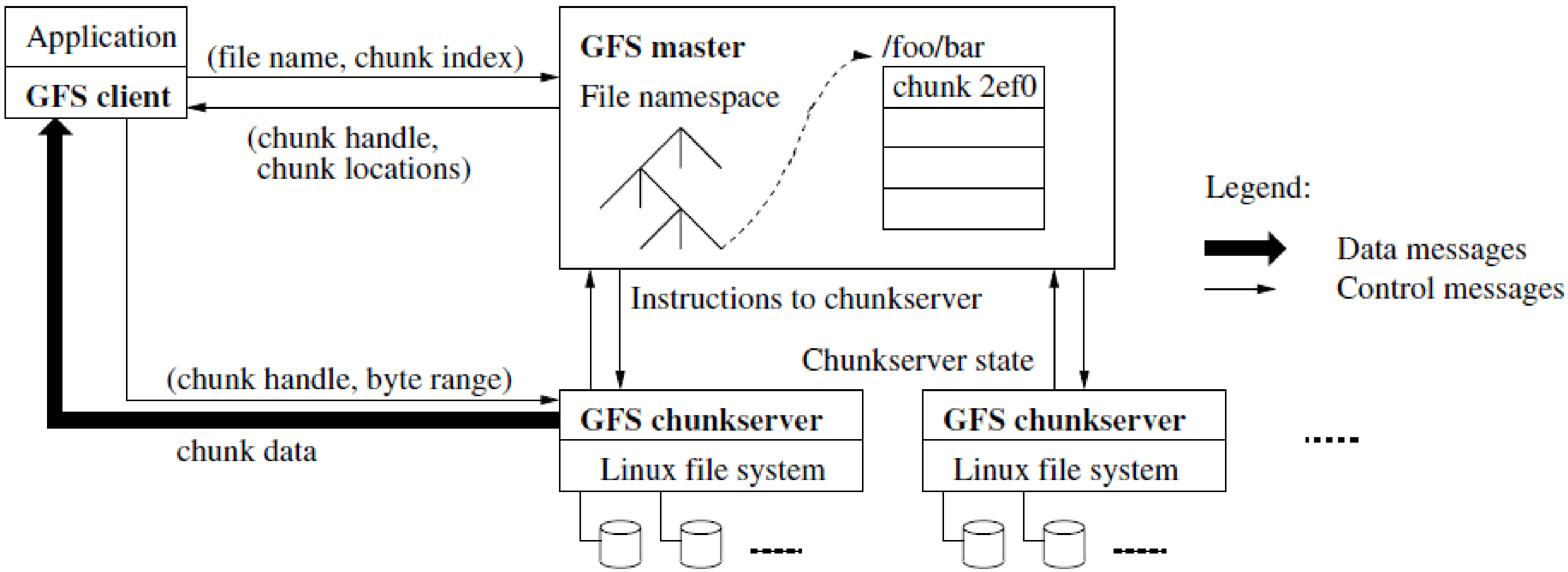
Cache Consistency & Concurrency Control

- AFS does not control concurrent updates of files, this is left up to the application
  - Deliberate decision, not to support distributed database system techniques, due to overhead this causes
- Cache consistency only on open and close operations
  - Once file is opened, modifications of file are possible without knowledge of other processes' operations on the file
  - Any close replaces current version on server
    - → All but the update resulting from last close operation processed at server will be lost, without warning
  - Application programs on same workstation share same cached copy of file, hence using standard UNIX block-by-block update semantics
- Although update semantics not identical to local UNIX file system, sufficiently close so that it works well in practice

Google File System

- The previous distributed file systems we've looked at provide clients remote access to a single server's file system
- GFS:
  - Files are distributed across thousands of servers
    - → Some of these servers are expected to fail at any given time
  - Often used to hold log that are huge by traditional standards
    - → Multi-GB to TB in size
  - Non-standard semantics
    - → For most Google workloads, random writes are non-existant
    - → GFS only provides atomic appends to files
  - Bandwidth more important than latency
    - → Main client is MapReduce, a batch computational framework that is not latency sensitive.

Google File System



- File names are mapped to a sequence of chunks
- Chunks are relatively large (64MB) and are managed by the chunkservers.
- Master keeps track of the file namespace, the mappings from file to chunk, and the chunk replica locations.