Module 7
File Systems & Replication

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

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*.

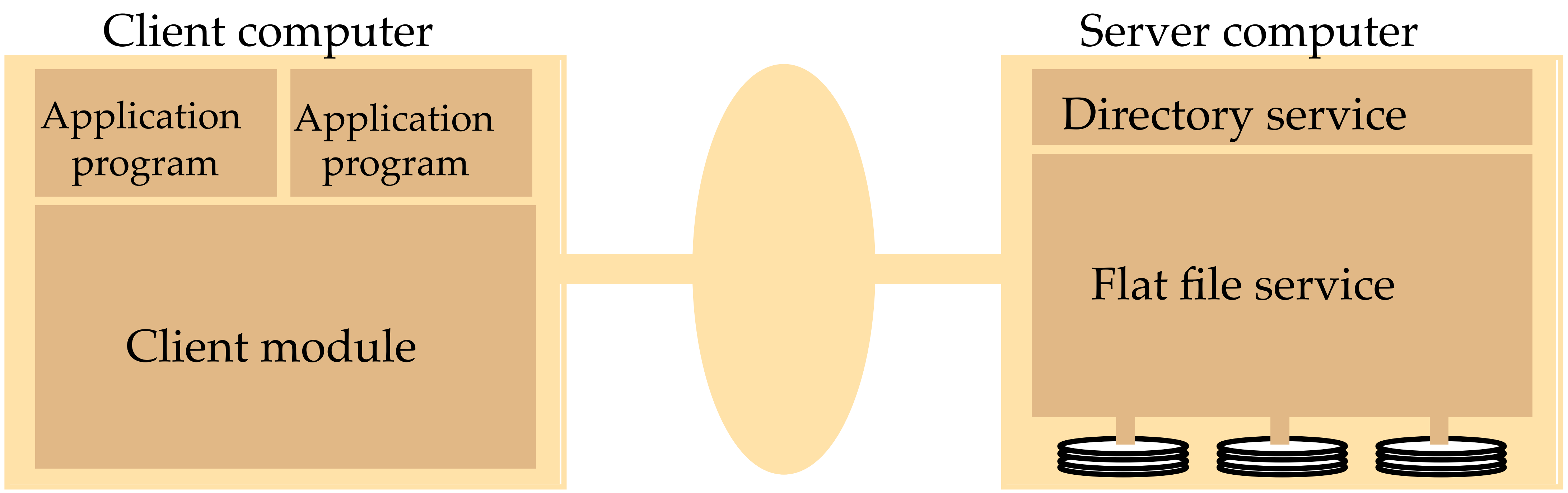
Distributed File System

- 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)

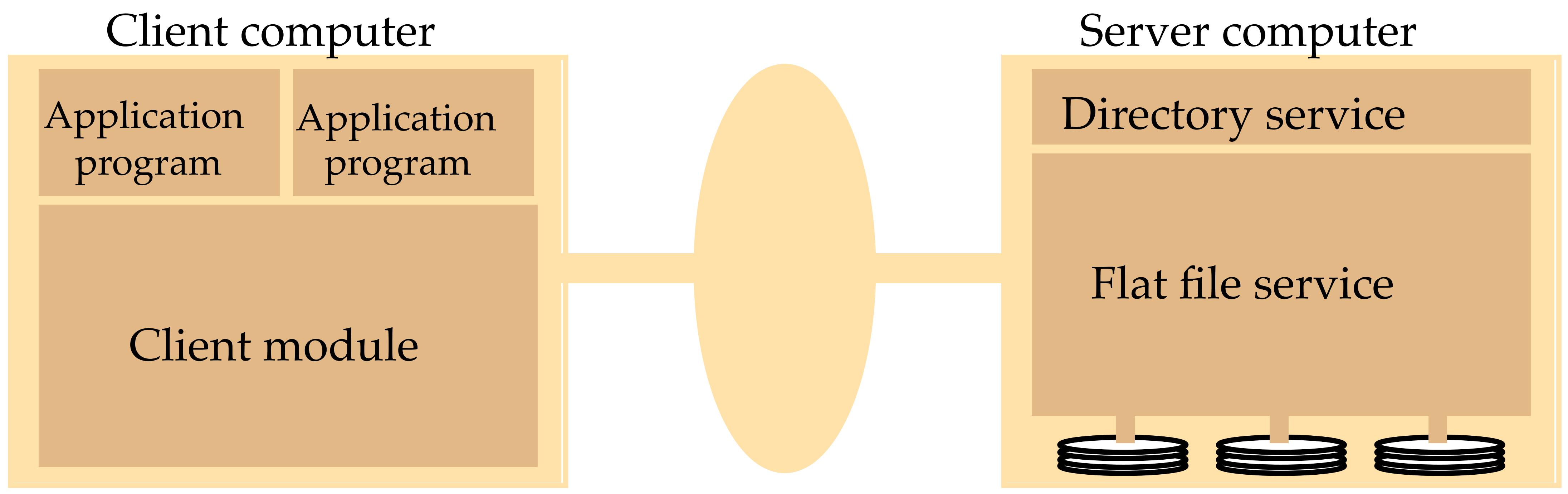
- Fault tolerance: continue to provide correct service in the presence of communication or server faults
  - At-most-once semantics for file operations
  - At-least-once semantics with a server protocol designed in terms of idempotent file operations
  - Replication (stateless, so that servers can be 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



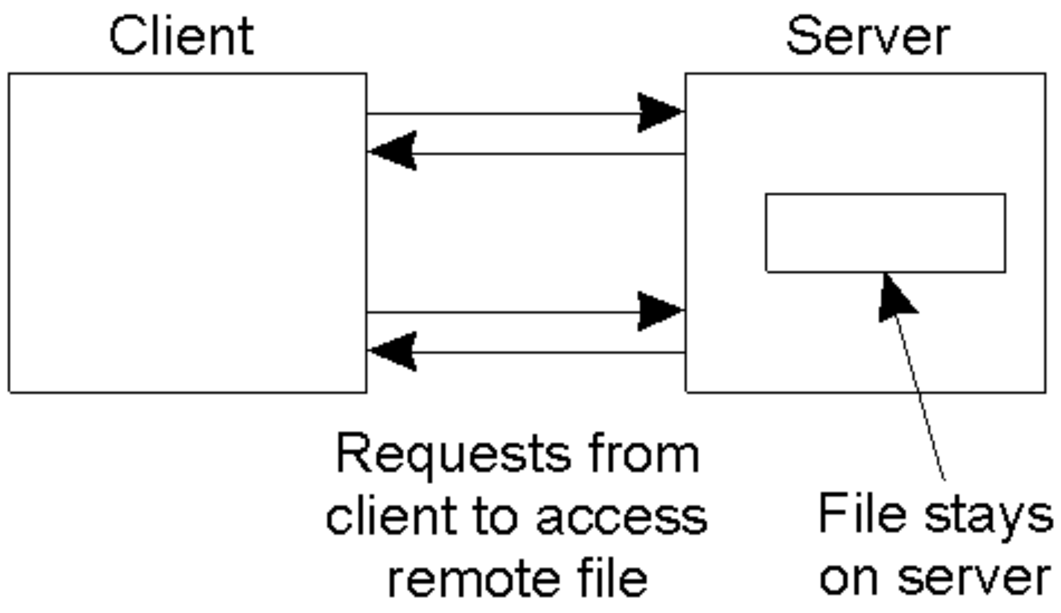
- Flat File Service
  - ➡ Performs file operations
  - ➡ Uses “unique file identifiers” (UFIDs) to refer to files
  - ➡ Flat file service interface
    - ✦ RPC-based interface for performing file operations
    - ✦ Not normally used by application level programs

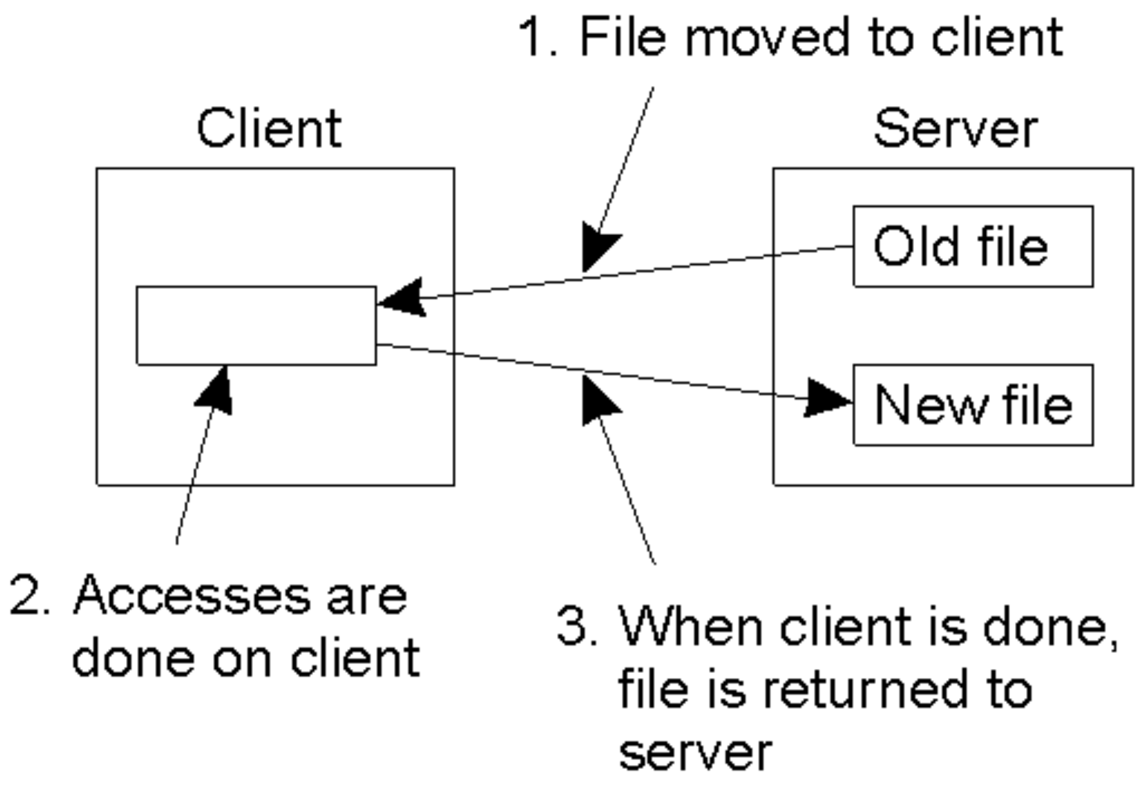
Architecture (2)



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

Distributed File Access Alternatives

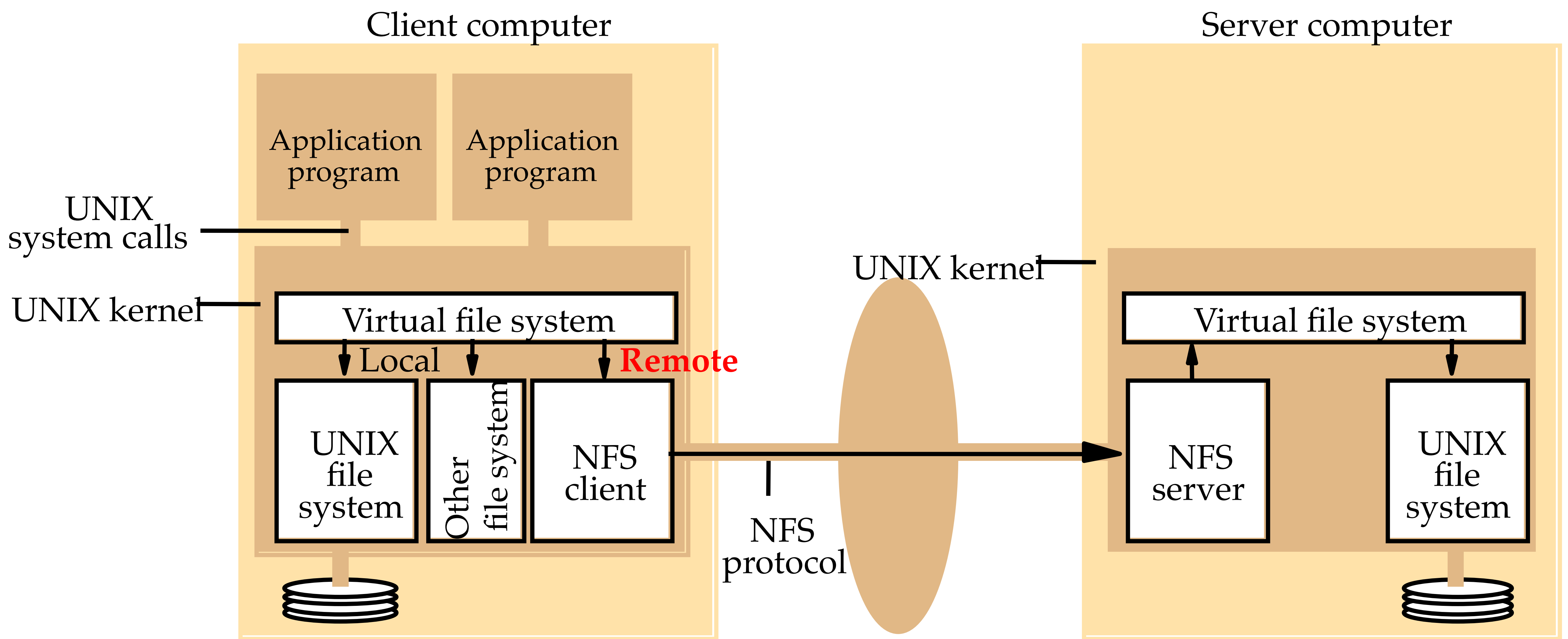



1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server

Remote access model

Download/upload model

Sun Network File System



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

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)

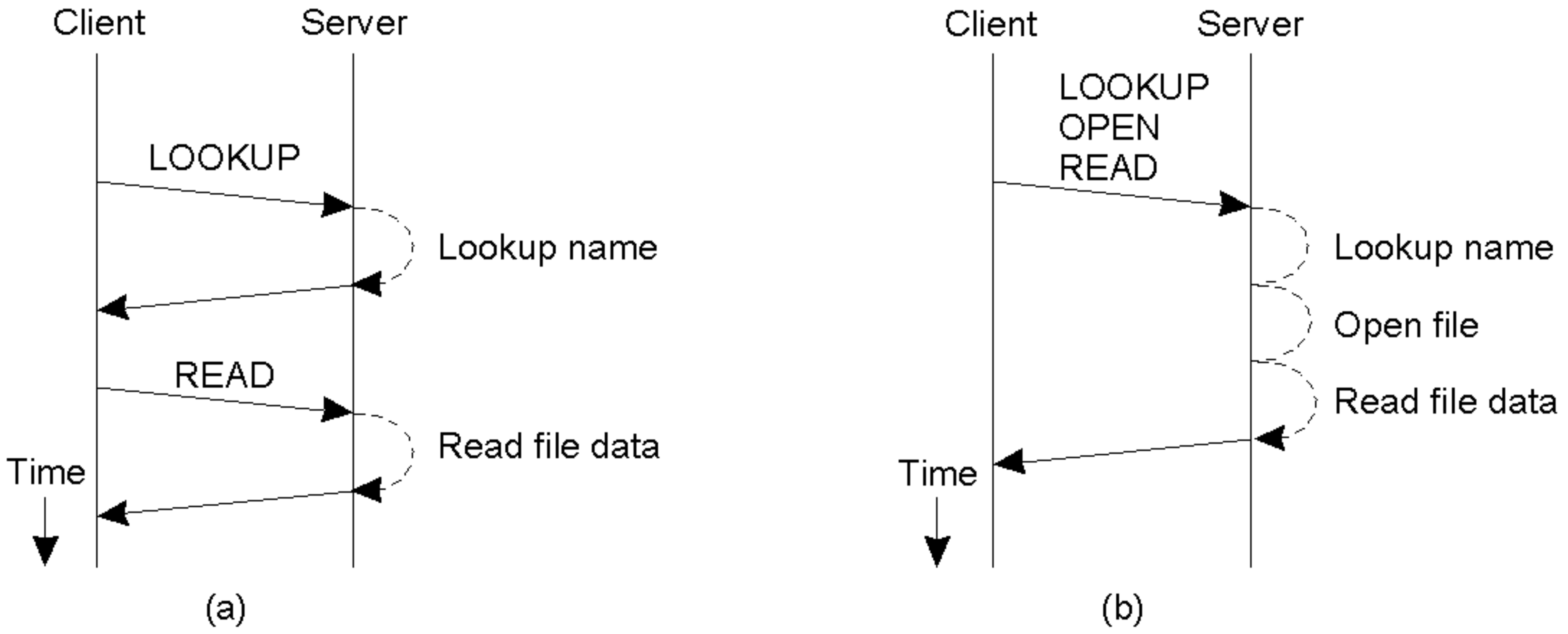
Selected NFS Operations

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*.

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*.

Communication

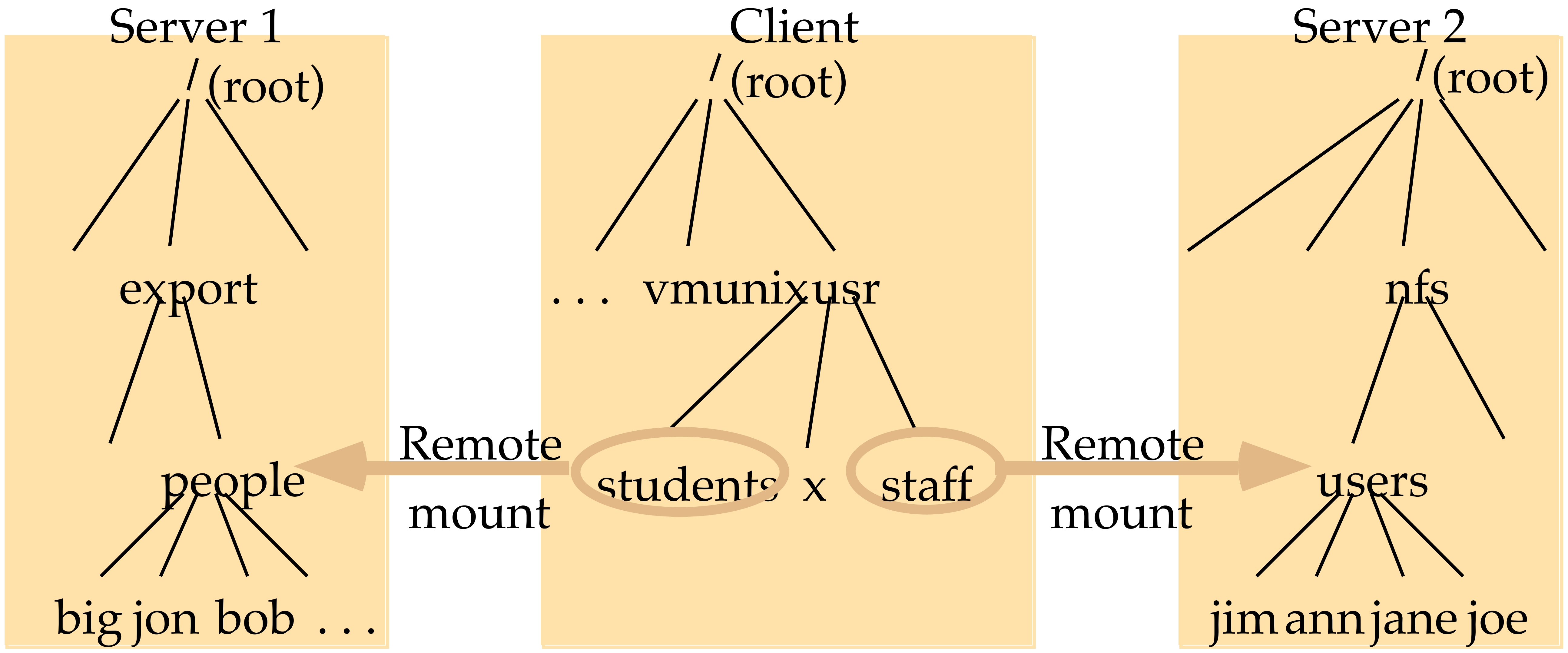


a) Reading data from a file in NFS version 3.
b) Reading data using a compound procedure in version 4.

Mounting of File Systems

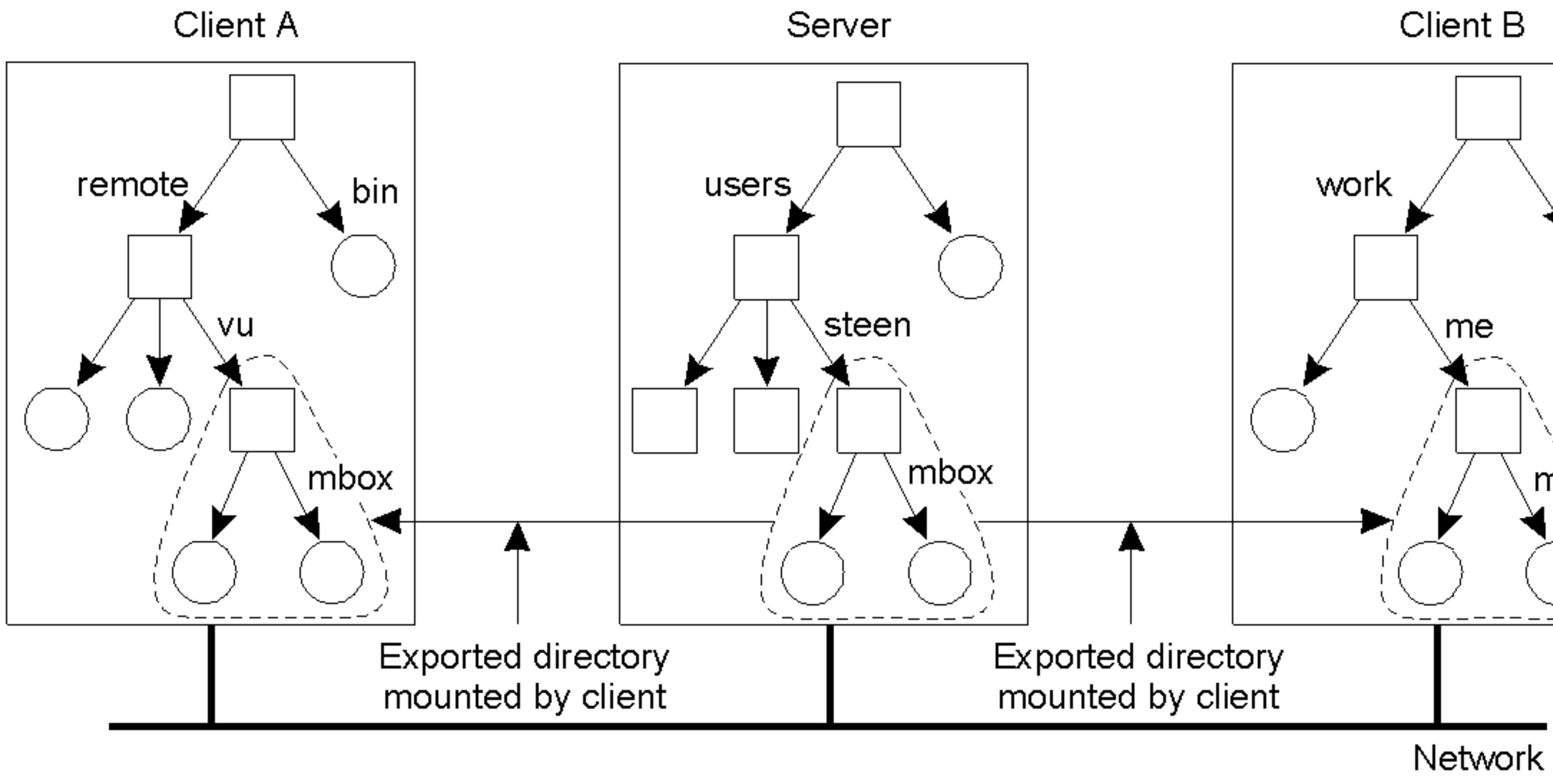
- Making remote file systems available to a local client, specifying remote host name and pathname
- Mount protocol (RPC-based)
  - ➡ Returns file handle for directory name given in request
  - ➡ Location (IP address and port number) and file handle are passed to Virtual File System and NFS client
- 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.

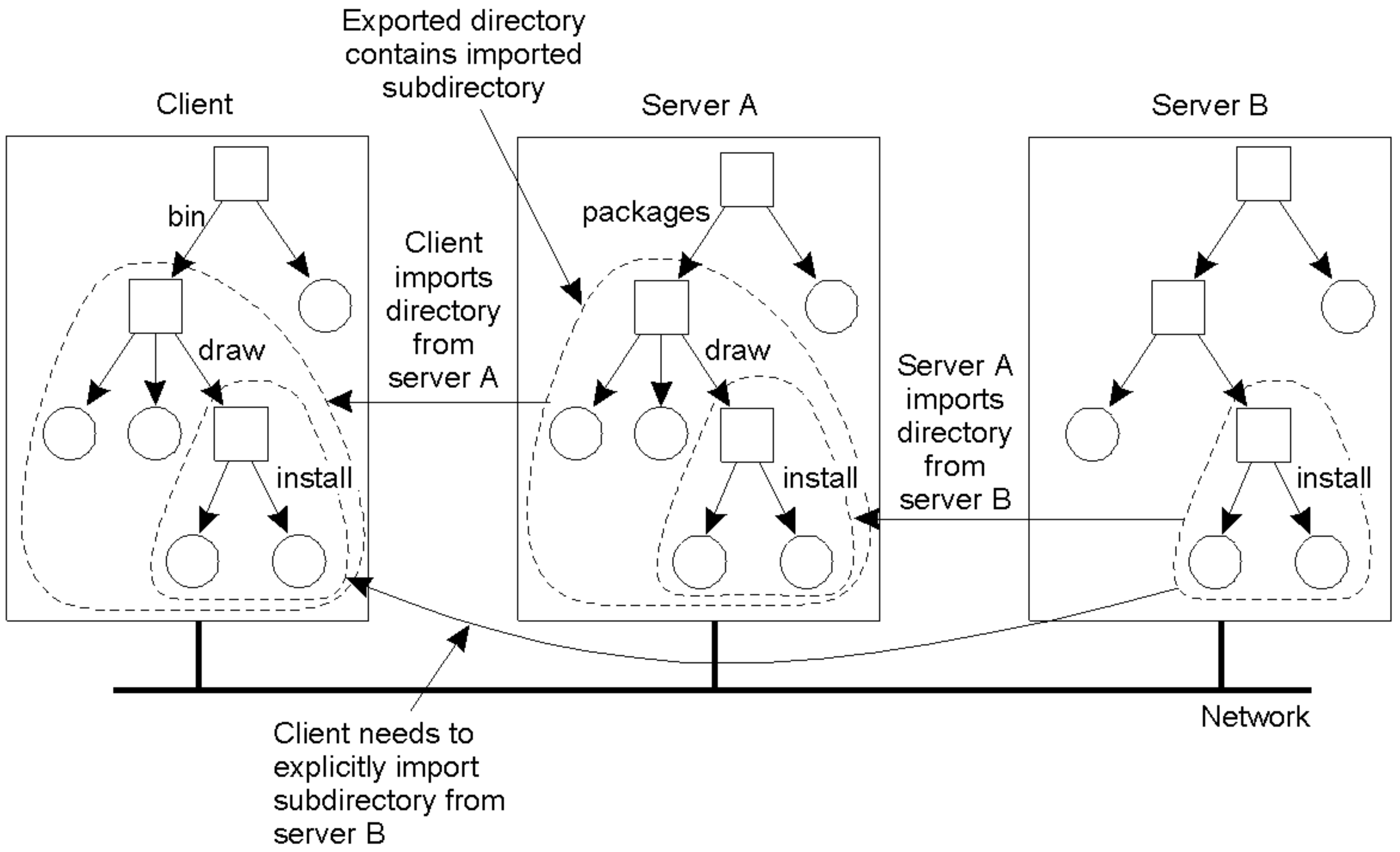
Naming (1)


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

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2007

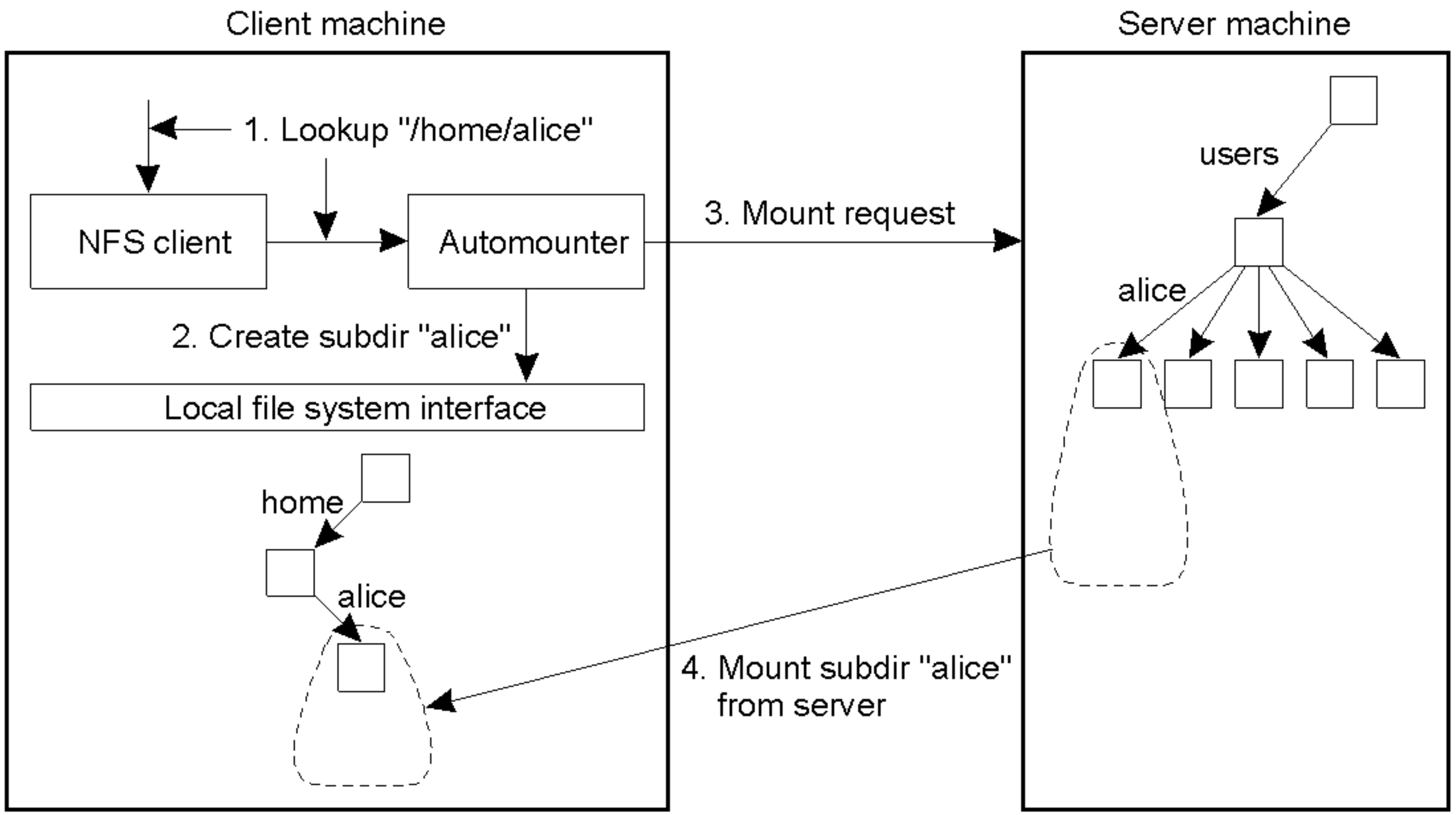
Naming (2)

- Mounting nested directories from multiple servers in NFS.


Exported directory contains imported subdirectory
Client
Server A
Server B
bin
draw
install
packages
draw
install
install
Client imports directory from server A
Server A imports directory from server B
Network
Client needs to explicitly import subdirectory from server B

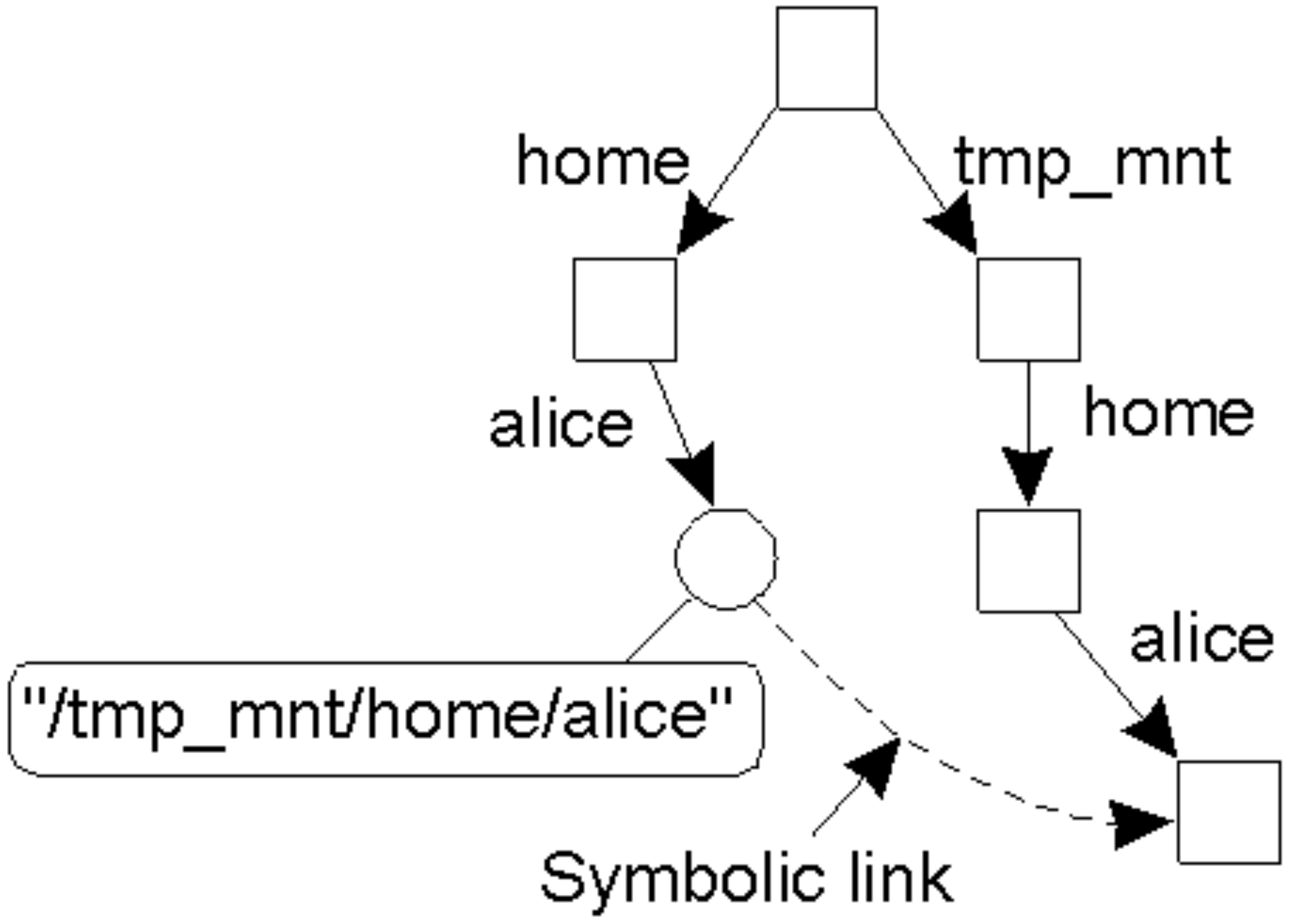
From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2007

Automounting (1)



A simple automounter for NFS.

Automounting (2)

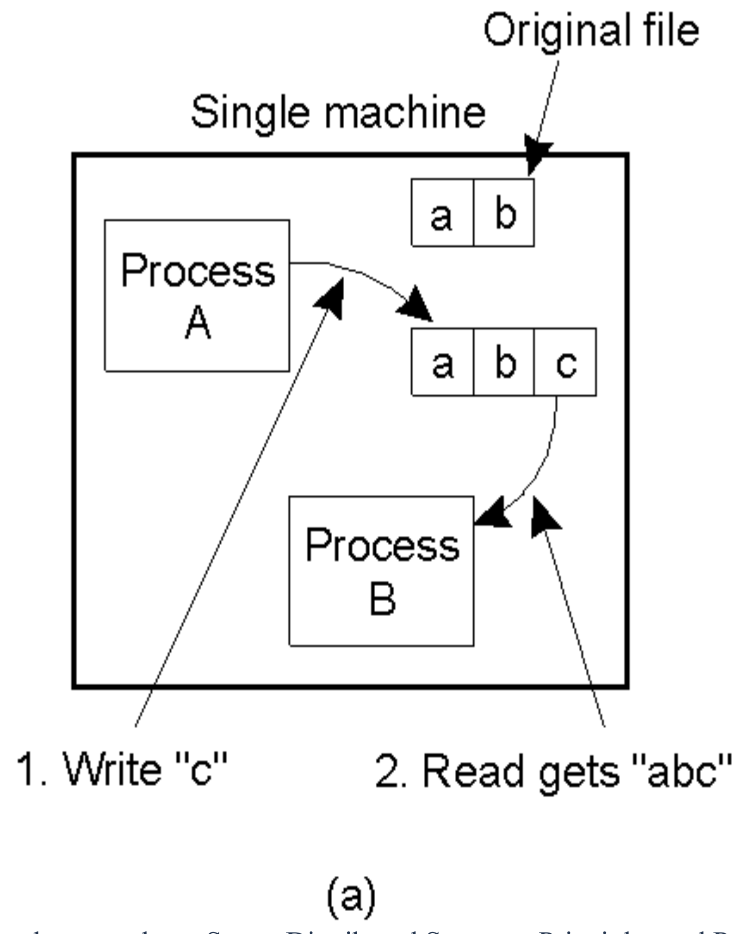


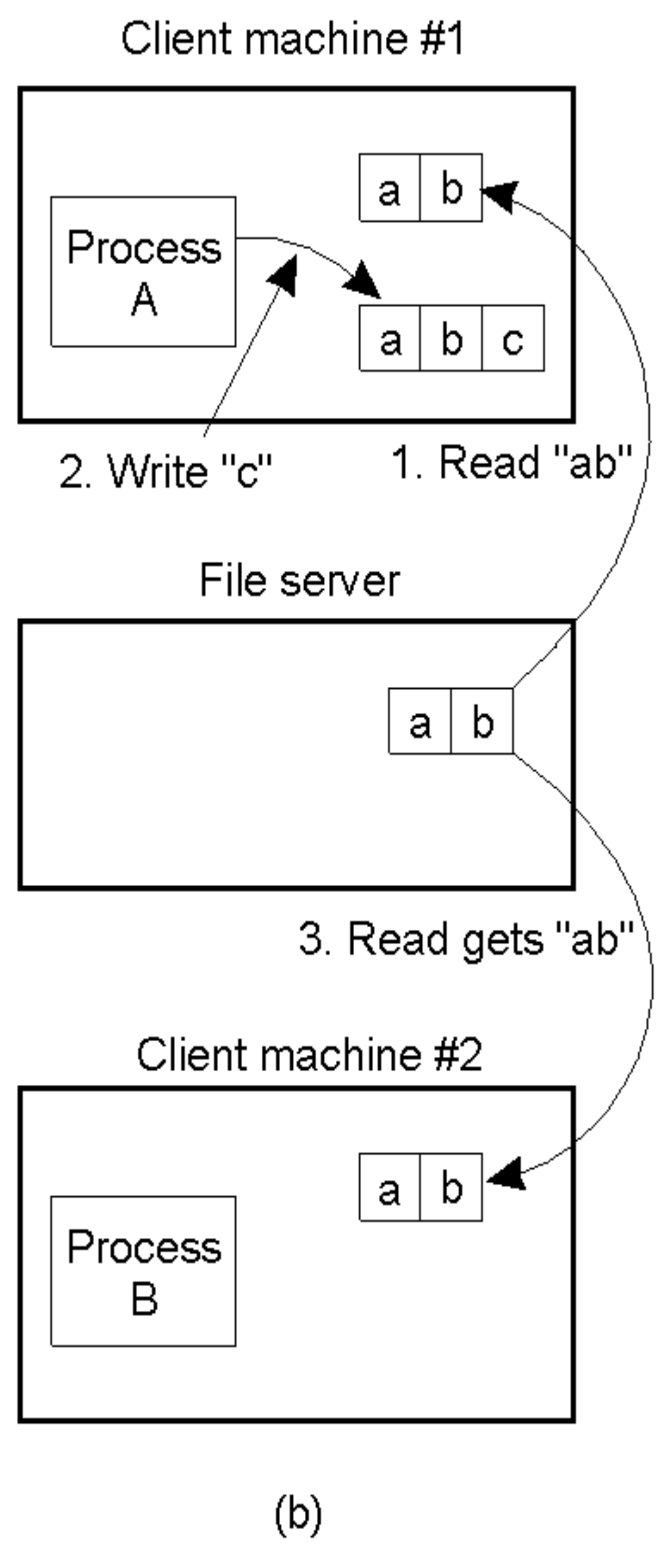
Using symbolic links with automounting.

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.





From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2007

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)

Request access / **Current file denial state**

	NONE	**READ**	**WRITE**	**BOTH**
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(a)

Current access state / **Requested file denial 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.

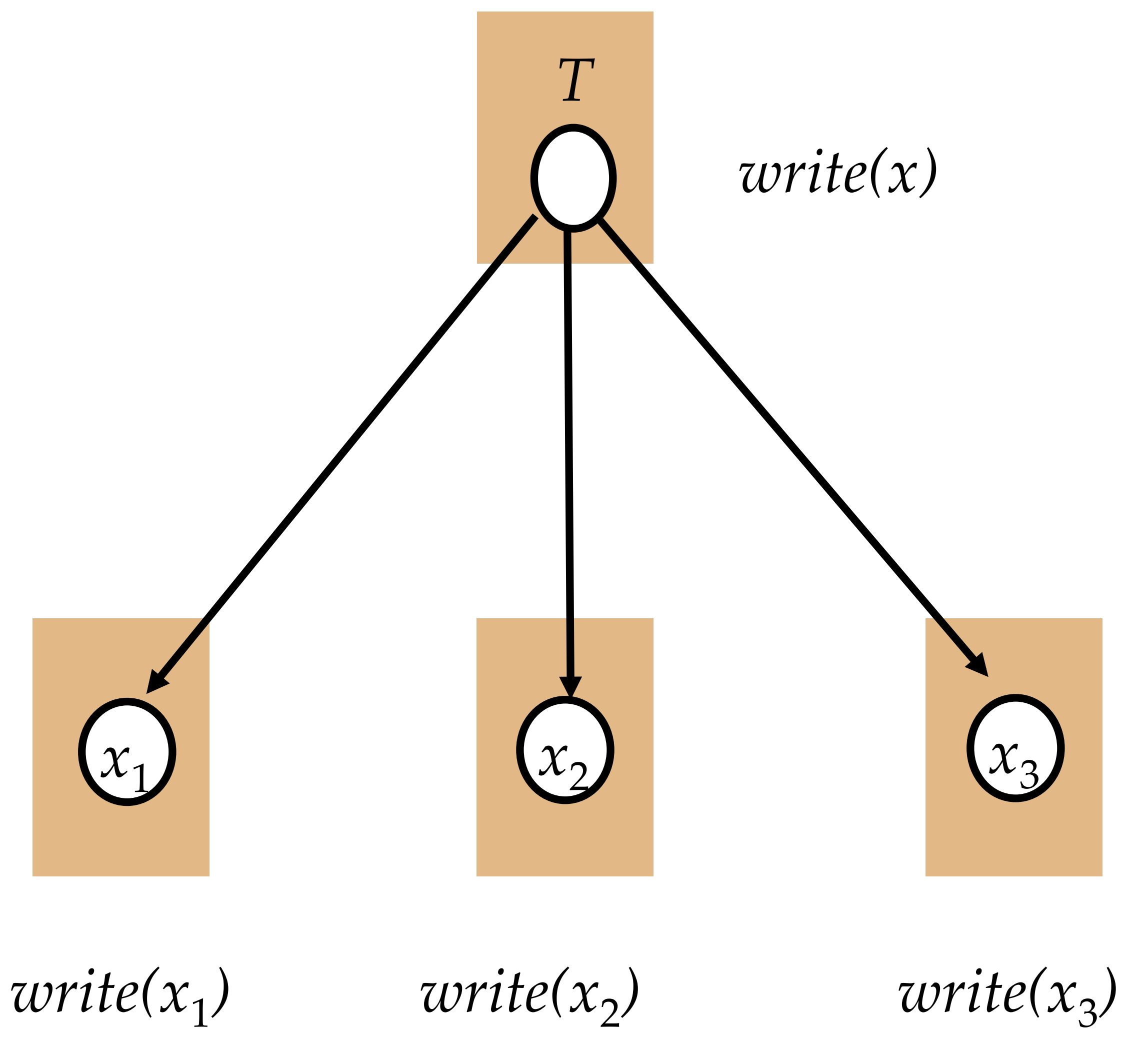
Replication

Replication

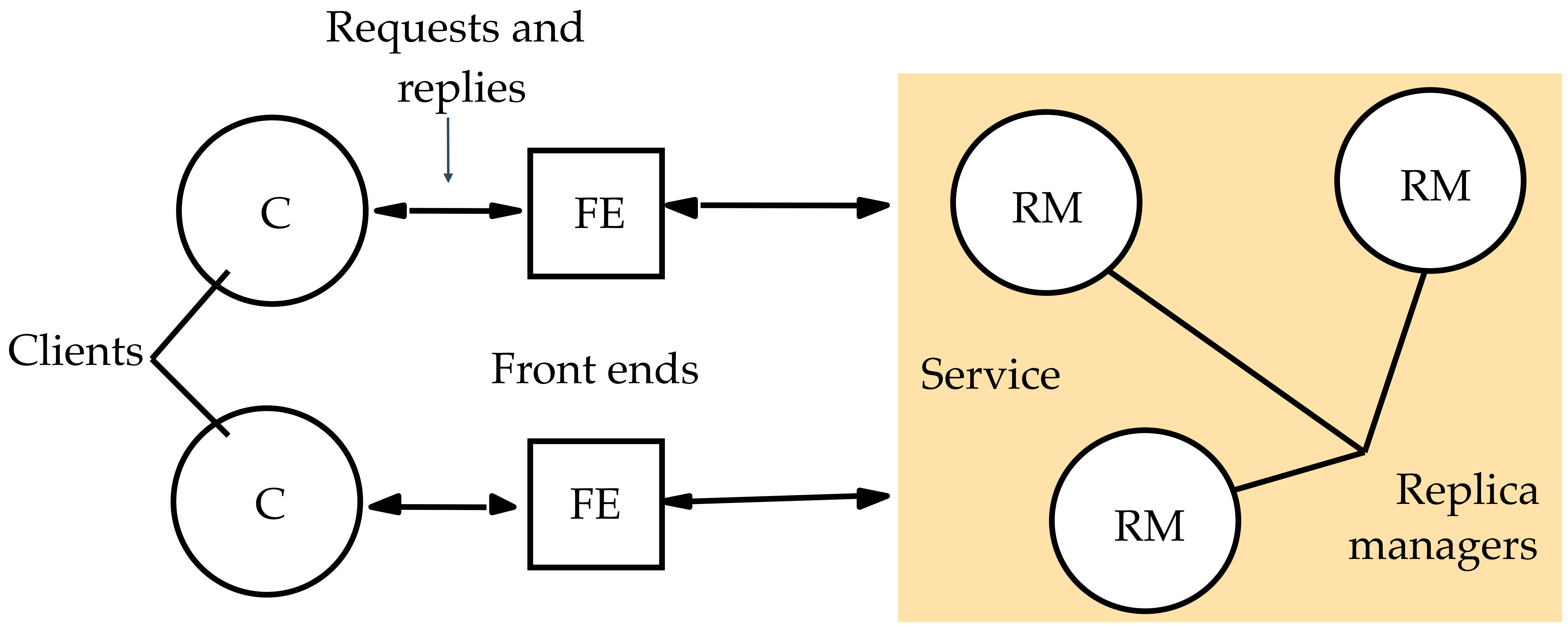
- Why replicate?
  - Reliability
    - Avoid single points of failure
  - Performance
    - Scalability in numbers and geographic area
- Why not replicate?
  - Replication transparency
  - Consistency issues
    - Updates are costly
    - Availability *may* suffer if not careful

Logical vs Physical Objects

- There are physical copies of logical objects in the system.
- Operations are specified on logical objects, but translated to operate on physical objects.



Replication Architecture



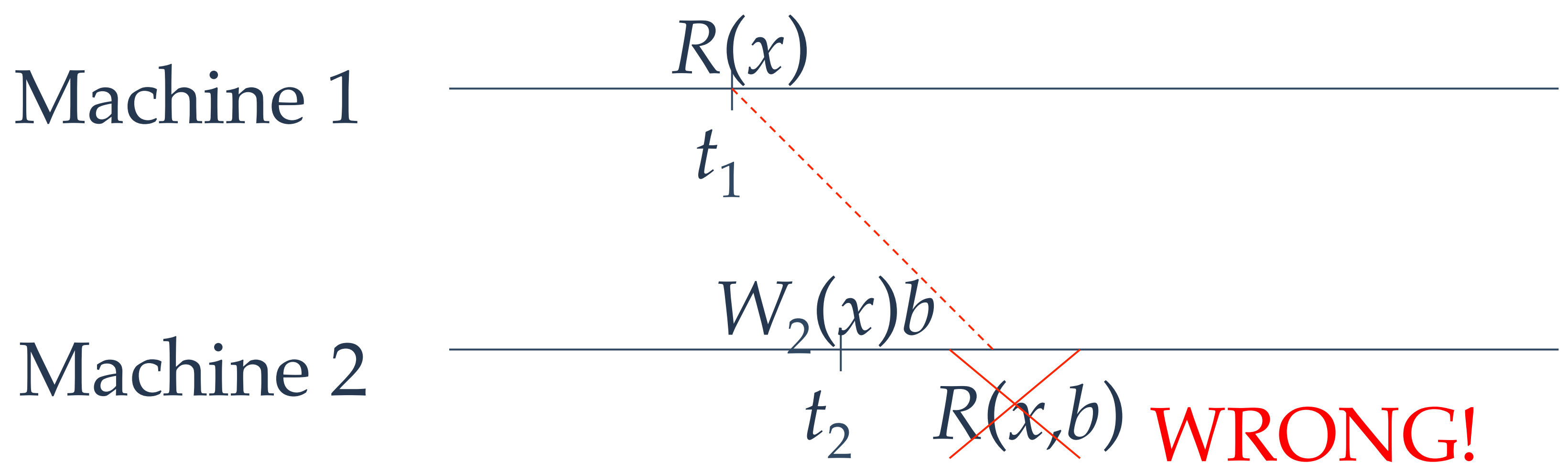
From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

Issues

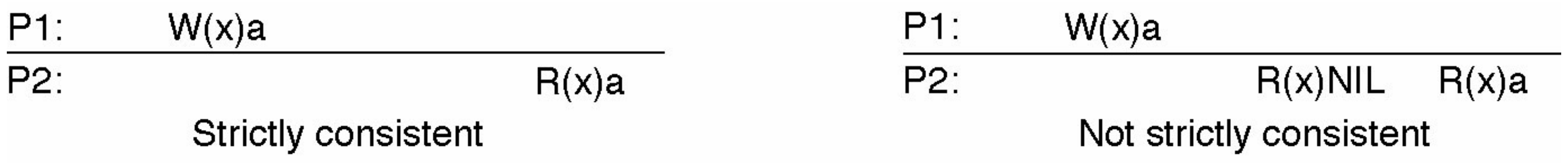
- Consistency models - How do we reason about the consistency of the "global state"?
  - ➡ Data-centric consistency
    - ✦ Strict consistency
    - ✦ Linearizability
    - ✦ Sequential consistency
  - ➡ Client-centric consistency
    - ✦ Eventual consistency
- Update propagation - How does an update to one copy of an item get propagated to other copies?
- Replication protocols - What is the algorithm that takes one update propagation method and enforces a given consistency model?

Strict Consistency

- Any *read*(*x*) returns a value corresponding to the result of the most recent *write*(*x*).

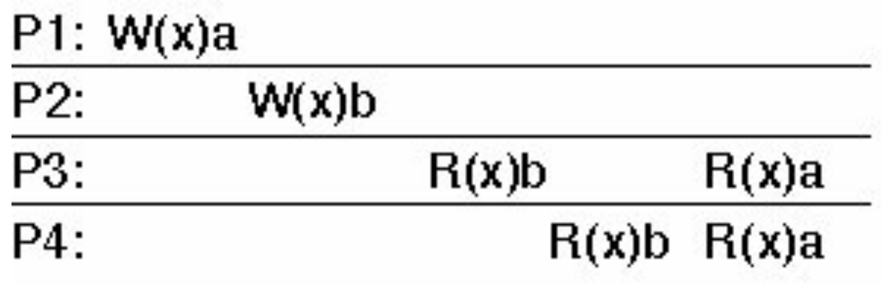


- Relies on absolute global time; all writes are instantaneously visible to all processes and an absolute global time order is maintained.
- Cannot be implemented in a distributed system

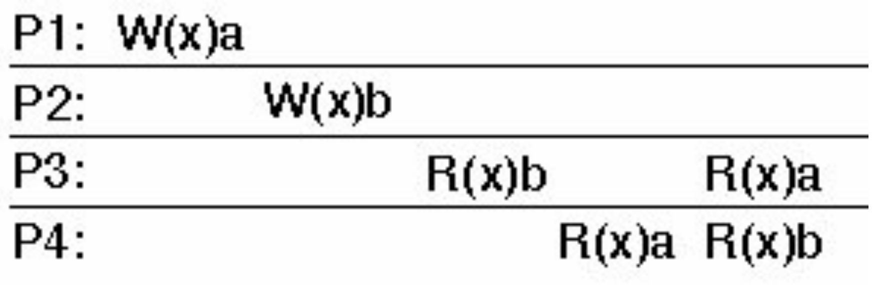


Sequential Consistency

- Similar to linearizability, but no requirement on timestamp order.
- The result of execution should satisfy the following criteria:
  - ➡ Read and write operations by all processes on the data store were executed in some sequential order;
  - ➡ Operations of each individual process appear in this sequence in the order specified by its program.
- These mean that all processes see the same interleaving of operations ⇨ similar to serializability.

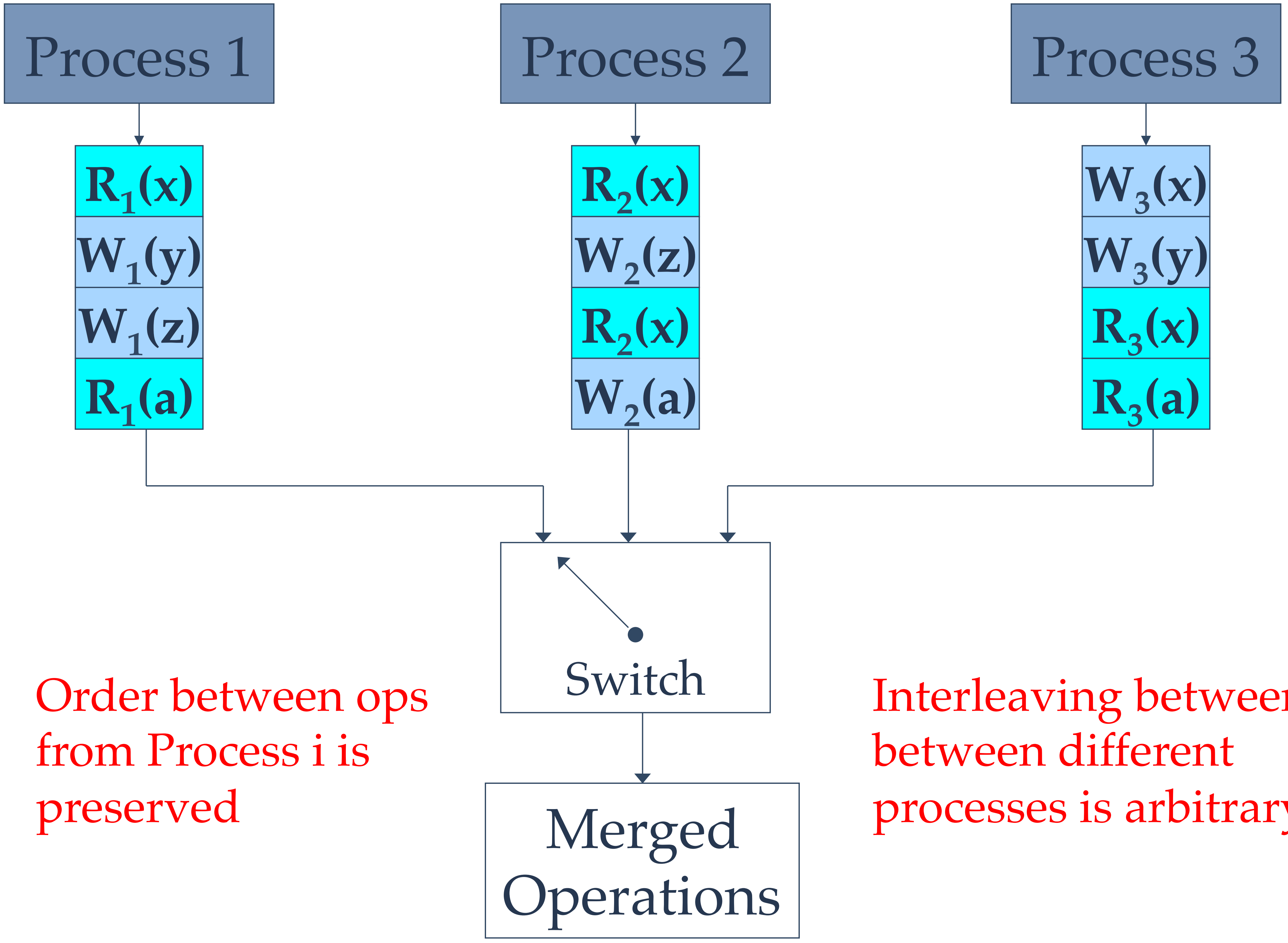


Sequentially consistent



Not sequentially consistent

Sequential Consistency



Order between ops from Process i is preserved

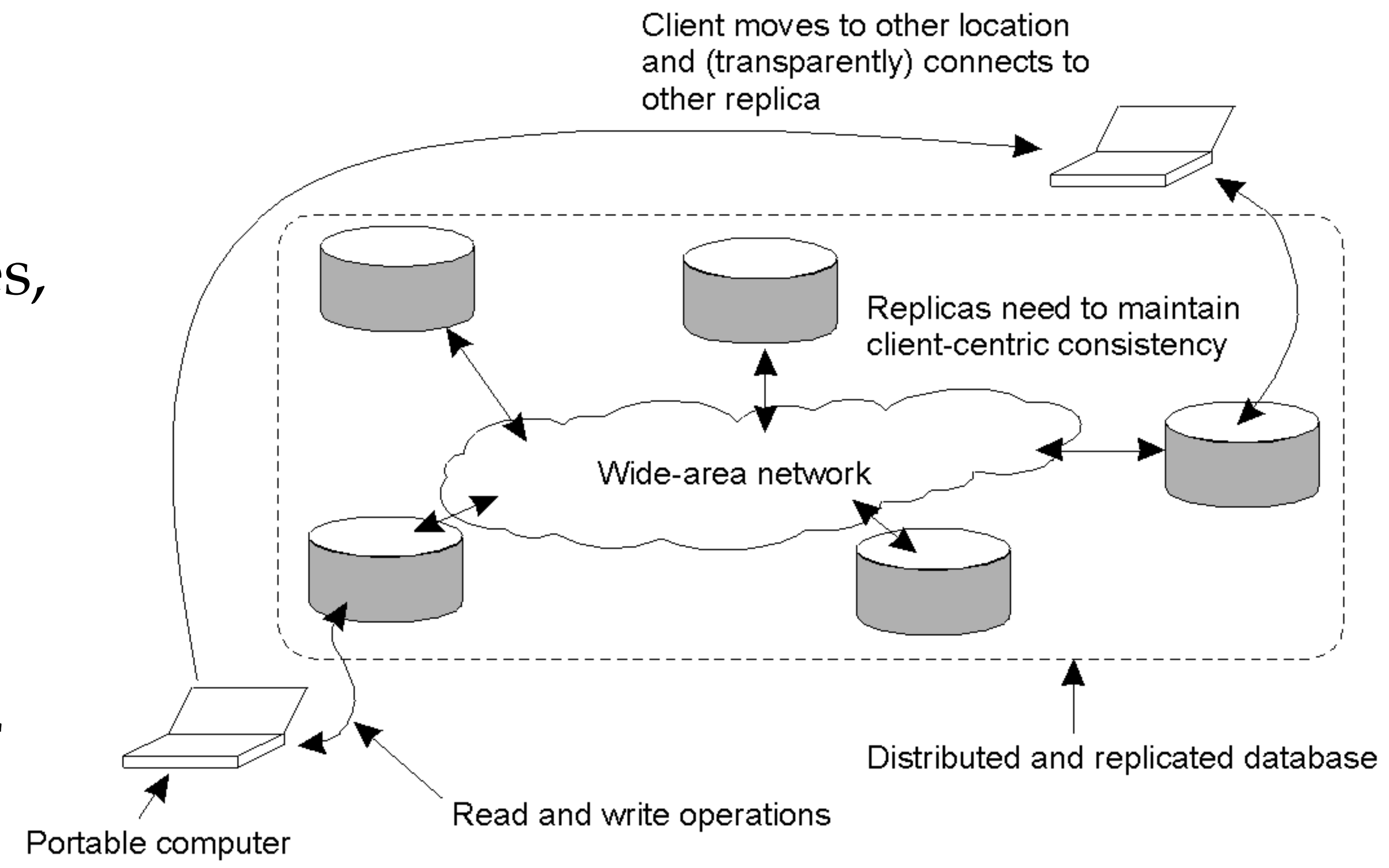
Interleaving between ops between different processes is arbitrary

Transactional Replica Consistency

- Efficient implementation of sequential consistency requires transactions.
- One-copy equivalence
  - ➡ The effect of transactions performed by clients on replicated objects should be the same as if they had been performed on a single set of objects.
- One-copy serializability
  - ➡ The effect of transactions performed by clients on replicated objects should be the same as if they had been performed *one at-a-time* on a single set of objects.
  - ➡ This is done within transactional boundaries.

Client-Centric Consistency

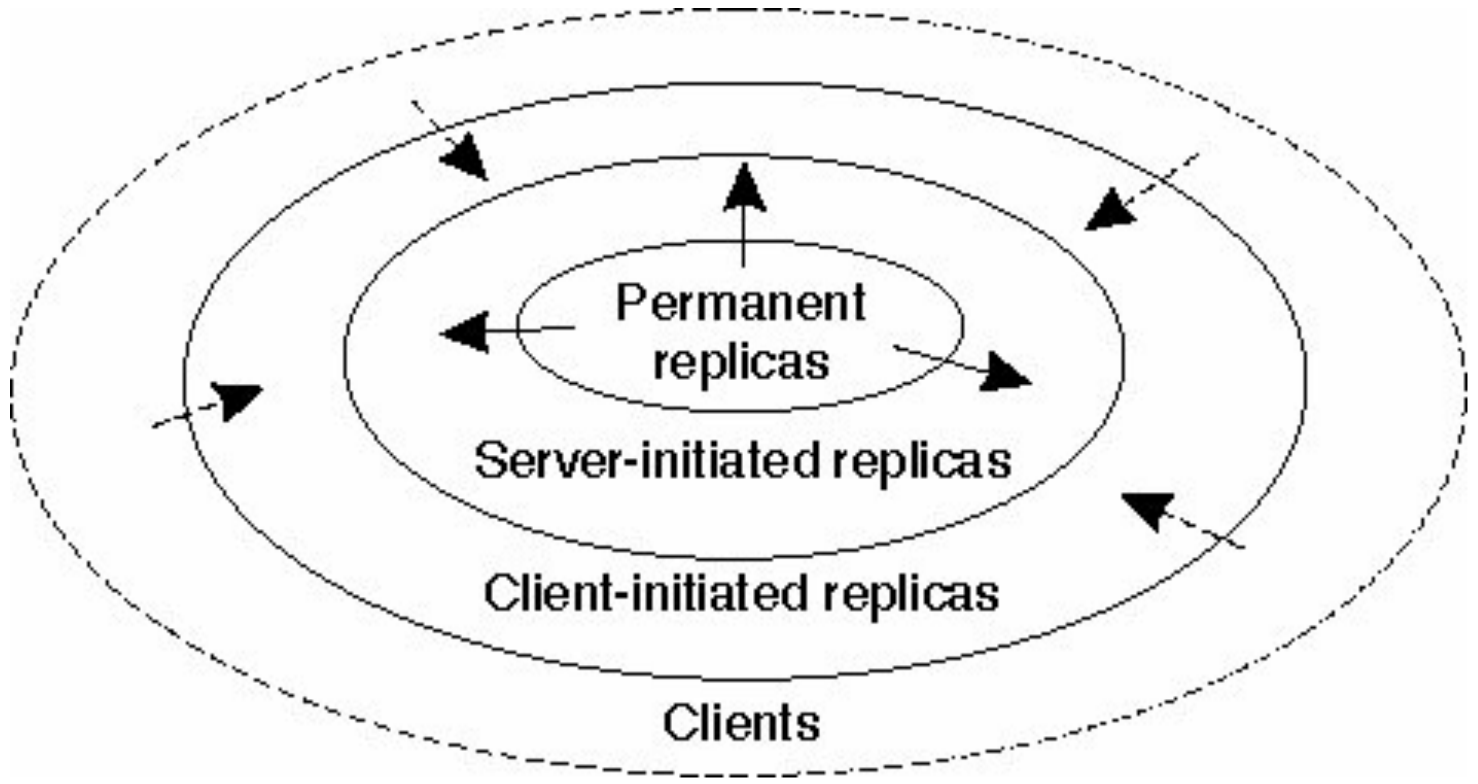
- More relaxed form of consistency ➡ only concerned with replicas being eventually consistent (eventual consistency).
- In the absence of any further updates, all replicas converge to identical copies of each other ➡ only requires guarantees that updates will be propagated.
- Easy if a user always accesses the same replica; problematic if the user accesses different replicas.
  - ➡ Client-centric consistency: guarantees for a single client the consistency of access to a data store.


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Client-Centric Consistency (2)

- Monotonic reads
  - ➡ If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.
- Monotonic writes
  - ➡ A write operation by a process on a data item x is completed before any successive write operation on x by the same process.
- Read your writes
  - ➡ The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.
- Writes follow reads
  - ➡ A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read.

Replica Placement Alternatives



- Permanent replicas
  - Put a number of replicas at specific locations
  - Mirroring
- Server-initiated replicas
  - Server decides where and when to place replicas
  - Push caches
- Client-initiated replicas
  - Client caches

Update Propagation

- What to propagate?
  - Propagate only a notification
    - Invalidation
  - Propagate updated data
    - Possibly only logs
  - Propagate the update operation
    - Active replication
- Who propagates?
  - Server: push approach
  - Client: pull approach
- Epidemic protocols
  - Update propagation in eventual-consistency data stores.

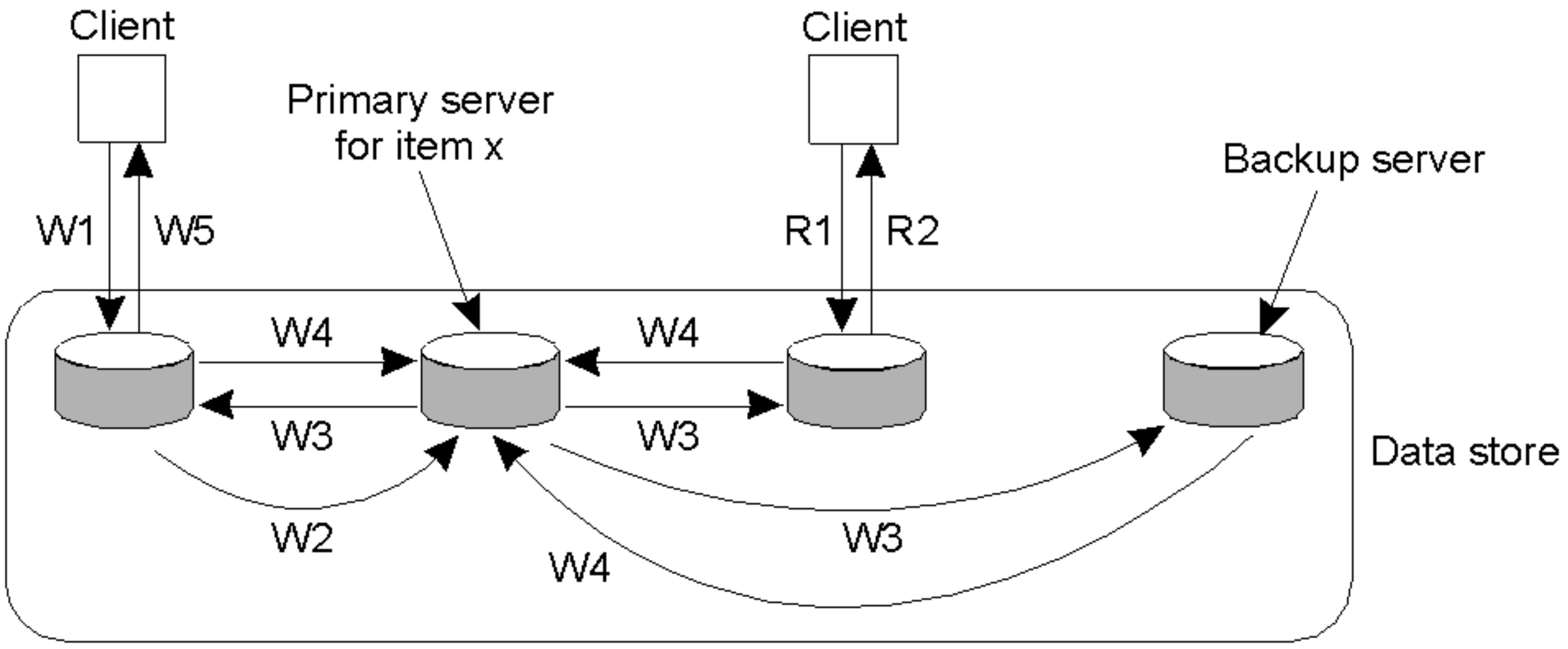
Pull versus Push Protocols

Issue	**Push-based**	**Pull-based**
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Replication Protocols

- We focus on those that enforce sequential consistency.
- Primary-based protocols
  - ➡ Remote-Write protocols
  - ➡ Local-Write protocols
- Replicated Write protocols
  - ➡ Active replication
  - ➡ Quorum-based protocols
- Read-one-Write-All (ROWA)

Primary Copy Remote-Write Protocol

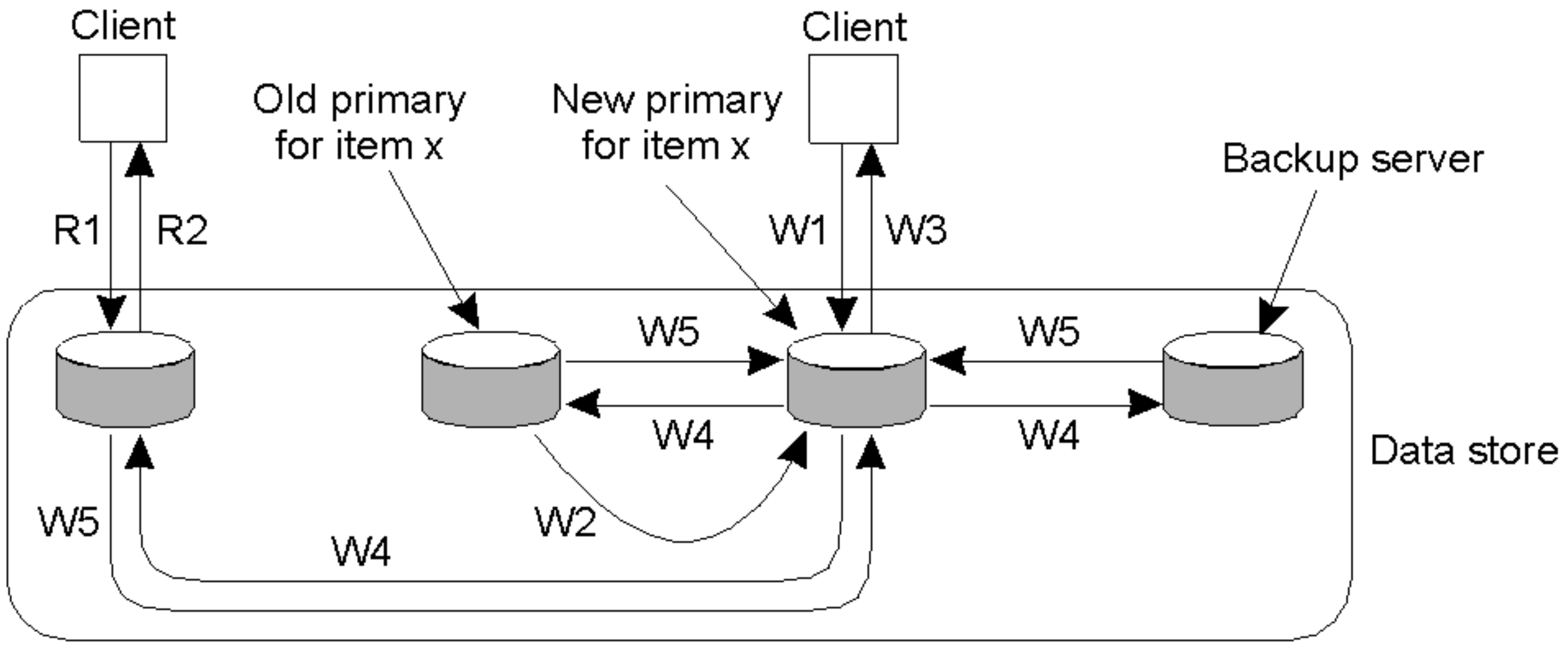


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2n edition

Primary Copy Local-Write Protocol



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

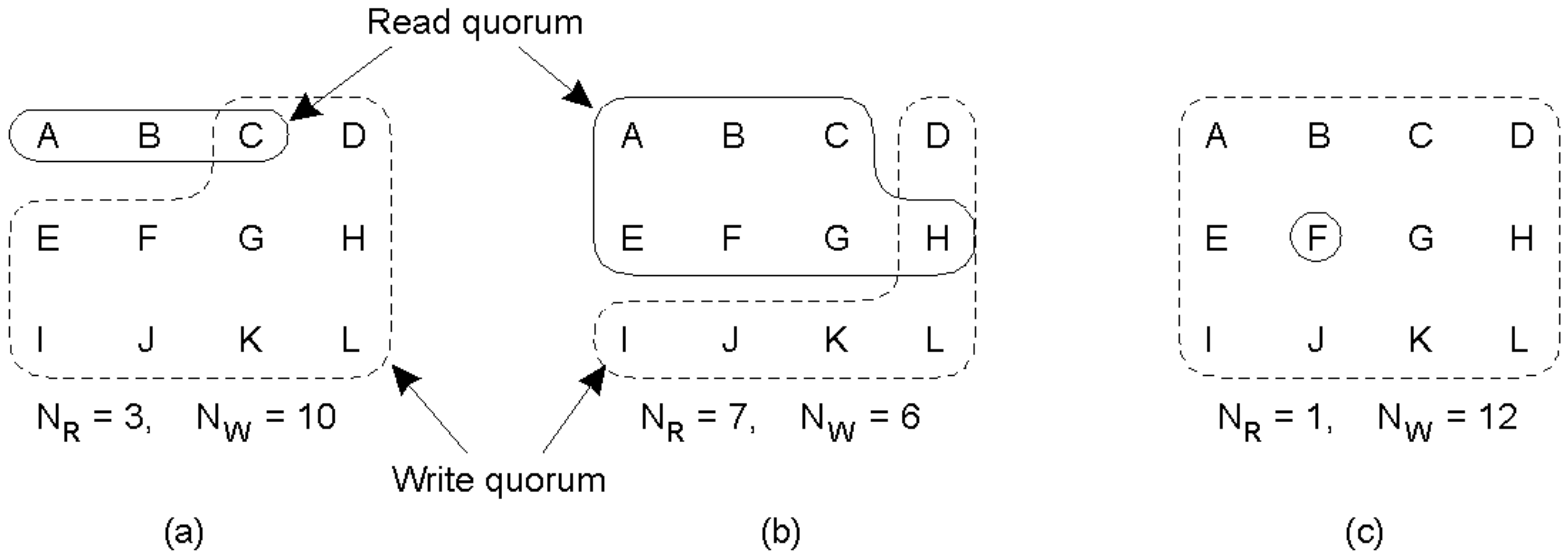
R1. Read request
R2. Response to read

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

Quorum-Based Protocol

- Assign a vote to each *copy* of a replicated object (say V_i) such that $\sum_i V_i = V$
- Each operation has to obtain a *read quorum* (V_r) to read and a *write quorum* (V_w) to write an object
- Then the following rules have to be obeyed in determining the quorums:
  - ➡ $V_r + V_w > V$ an object is not read and written by two transactions concurrently
  - ➡ $V_w > V/2$ two write operations from two transactions cannot occur concurrently on the same object

Quorum Example



Three examples of the voting algorithm:

a) A correct choice of read and write set

b) A choice that may lead to write-write conflicts

c) ROWA