DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 11
DISTRIBUTED FILE SYSTEMS

*** Modified by Prof. Rivalino Matias, Jr.**

Outline

- Concepts
- Architectures
- Implementations

Introduction

- Distributed file systems (DFS) allow multiple processes to share data from multiple locations.
  - They have been used as the basic layer for distributed systems.
- File systems in DS have the same purpose than in local system (non-distributed systems).
  - They are built on top of the local file systems.
  - It requires several additional mechanisms related to communication, synchronization, replication, consistency, etc.
- Most used approaches are based on client-server architectures.
  - But fully decentralized solutions exist as well.
- Examples:
  - NFS – Network File System
  - AFS – Andrew File System

Architectures

- **Client-server architecture**
  - Many DFS are organized along the lines of client-server architectures.
  - The basic idea behind this model is that each distributed file server provides a standardized view of its local file system.
    - In other words, it should not matter how that local file system is implemented.
  - To do so, the DSFS comes with a communication protocol that allows clients to access the files stored on a server, thus allowing a heterogeneous collection of processes, possibly running on different operating systems and machines, to share a common file system.
  - The **Network File System** (NFS) is one example of client-server-based file system. It is widely-deployed in UNIX-based systems.

Client-Server Architectures

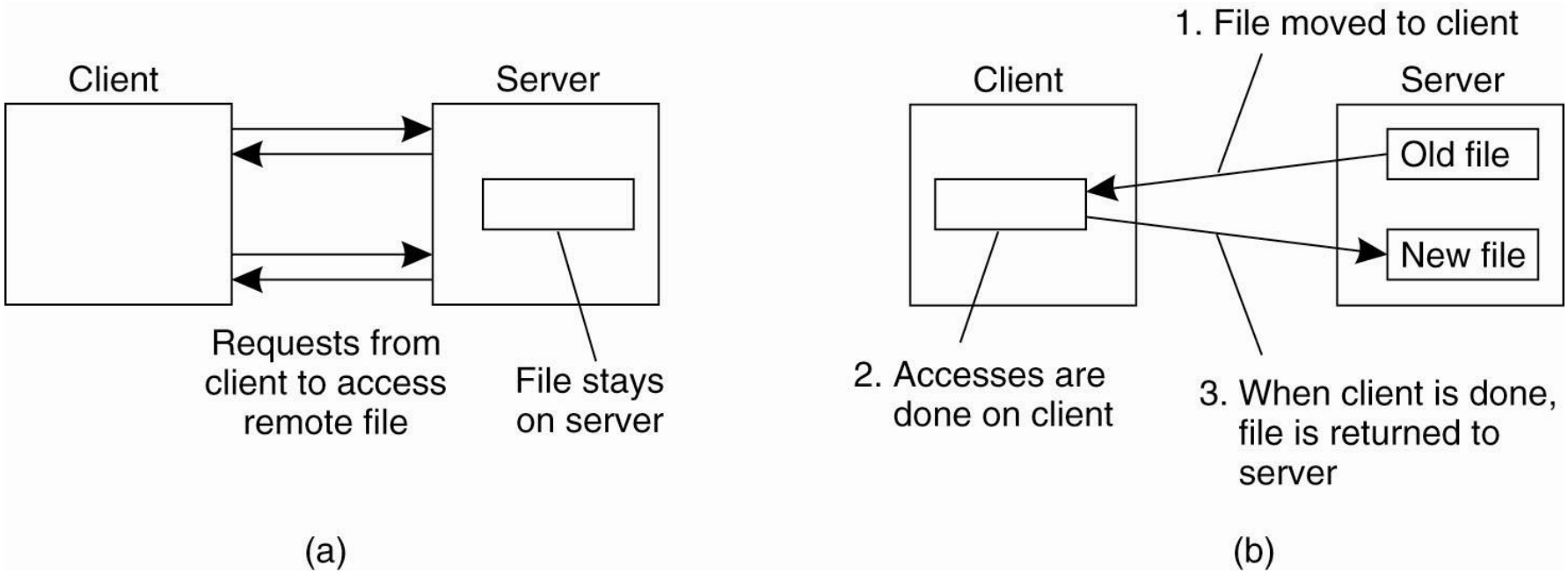

Figure 11-1. (a) The remote access model. (b) The upload/download model.

Client-Server Architectures

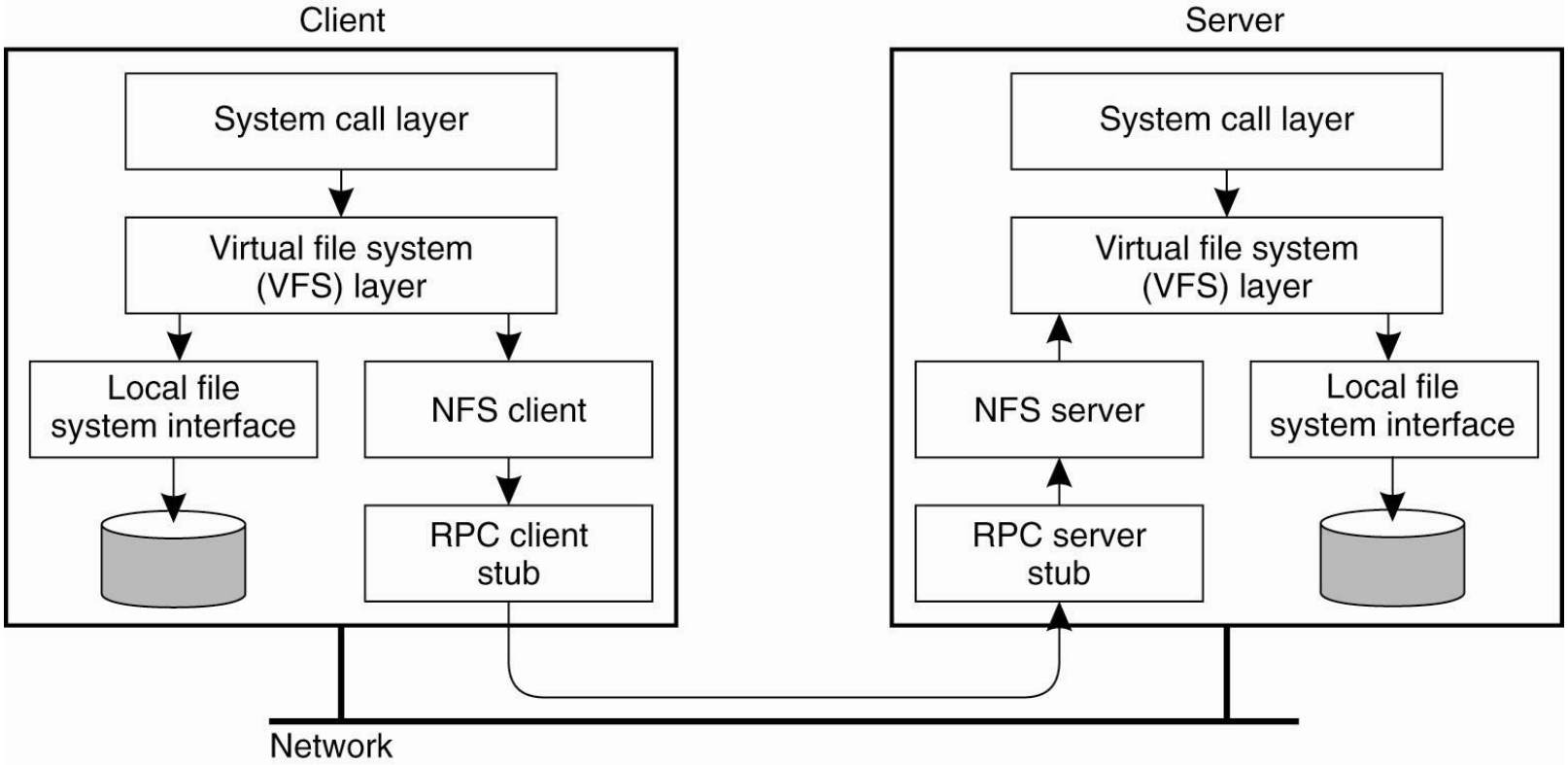

Figure 11-2. The basic NFS architecture for UNIX systems.

File System Model

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
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory

Figure 11-3. An incomplete list of file system operations supported by NFS.

File System Model

Operation	v3	v4	Description
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	Get 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

Figure 11-3. An incomplete list of file system operations supported by NFS.

Architectures

- **Cluster-Based Distributed File Systems**
  - Server clusters are often used for parallel applications, and their associated file systems are adjusted accordingly.
  - One well-known technique is to deploy file-striping techniques, by which a single file is distributed across multiple servers.
    - **The basic idea is simple**: by distributing a large file across multiple servers, it becomes possible to fetch different parts in parallel.
    - Such an organization works well only if the application is organized in such a way that parallel data access makes sense.
  - This requires that the data as stored in the file have a very regular structure, for example, a (dense) matrix.
  - For general-purpose applications, or those with irregular or many different types of data structures, file striping may not be an effective approach.
    - It is more convenient to partition the file system as a whole and store different files on different servers.

Cluster-Based Distributed File Systems

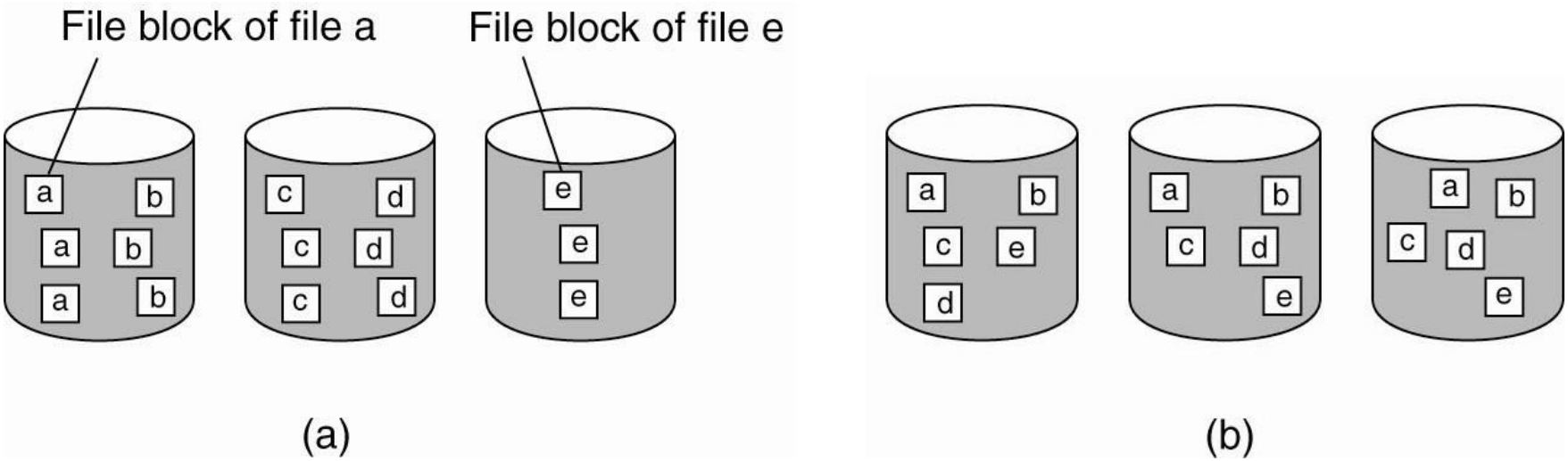

Figure 11-4. The difference between (a) distributing whole files across several servers and (b) striping files for parallel access.

Architectures

- **Cluster-Based Distributed File Systems**
  - This type of FS can be applied to very large data centers such as those used by companies like Amazon and Google.
    - These companies offer services to Web clients resulting in reads and updates to a massive number of files distributed across literally tens of thousands of computers.
  - In such environments, at any single moment there will be a computer malfunctioning.
  - To address these problems, Google has developed its own Google file system (GFS).
    - Google files tend to be very large, commonly ranging up to multiple gigabytes, where each one contains lots of smaller objects.
    - Moreover, updates to files usually take place by appending data rather than overwriting parts of a file.
    - These observations, along with the fact that server failures are the norm, lead to constructing clusters of servers.

Cluster-Based Distributed File Systems

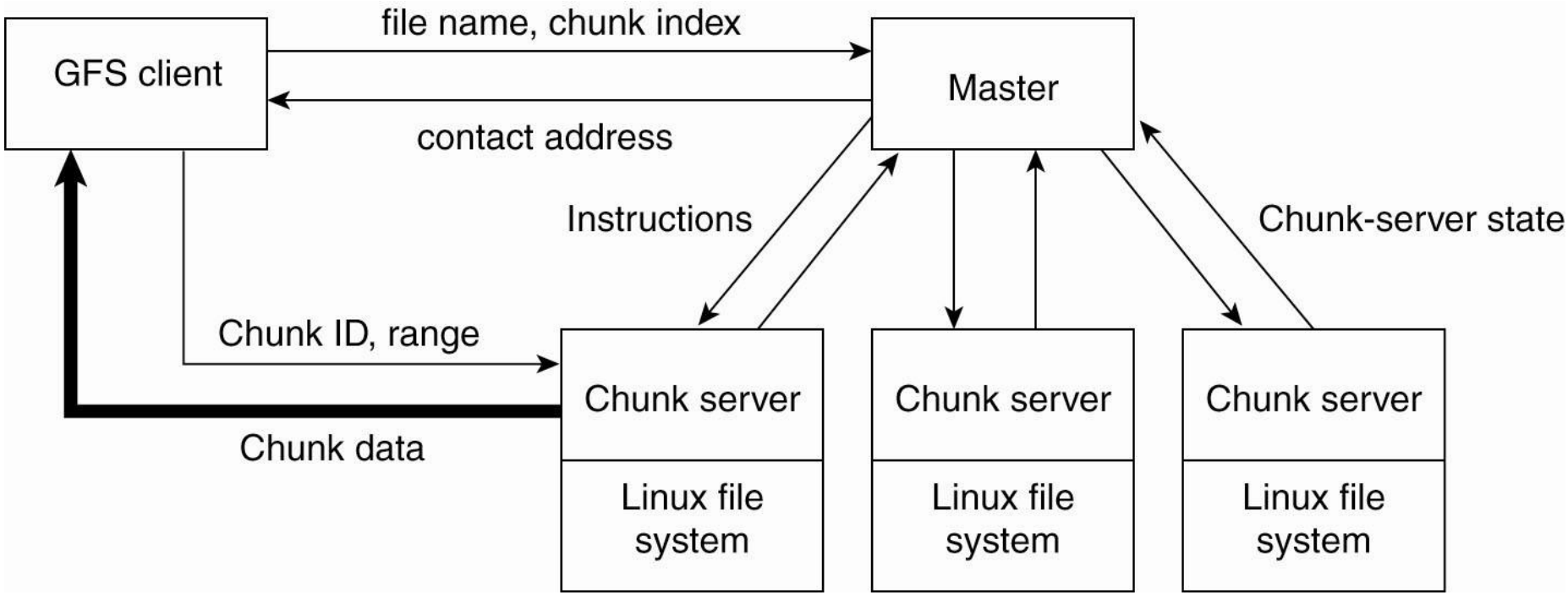

Figure 11-5. The organization of a Google cluster of servers.

Processes

- When it comes to processes, distributed file systems have no unusual properties.
  - In many cases, there will be different types of cooperating processes: **storage servers** and **file managers**.
- The most interesting aspect concerning file system processes is whether or not they should be stateless.
  - NFS is a good example illustrating the trade-offs.
    - One of its long-lasting distinguishing features (compared to other distributed file systems), was the fact that servers were stateless.
    - In other words, the NFS protocol did not require that servers maintained any client state.
    - This approach was followed in versions 2 and 3, but has been abandoned for version 4.

Processes

- NFS: Stateless vs. Stateful
  - The primary advantage of the stateless approach is simplicity.
  - When a stateless server crashes, there is essentially no need to enter a recovery phase to bring the server to a previous state.
  - However, we still need to consider that the client cannot be given any guarantees whether a request has been carried out.
  - The stateless approach in the NFS could not always be fully followed in practical implementations.
    - E.g, locking a file cannot easily be done by a stateless server.
    - In the case of NFS, a separate lock manager is used to handle this situation.
    - Likewise, certain authentication protocols require that the server maintains state on its clients.
    - Nevertheless, NFS servers could generally be designed in such a way that only very little information on clients needed to be maintained.
    - For the most part, the scheme worked adequately.

Processes

- NFS: Stateless vs. Stateful
  - Starting with version 4, the stateless approach was abandoned, although the new protocol is designed in such a way that a server does not need to maintain much information about its clients.
  - Besides those just mentioned, there are other reasons to choose for a **stateful approach**.
    - An important reason is that NFS4 is expected to also work across wide-area networks.
    - This requires that clients can make effective use of caches, in tum requiring an efficient cache consistency protocol.
    - Such protocols often work best in collaboration with a server that maintains some information on files as used by its clients.
    - For example, a server may associate a lease with each file it hands out to a client, promising to give the client exclusive read and write access until the lease expires or is refreshed.

Processes

- NFS4 (other features)
  - This version supports the **open** operation.
  - It also supports callback procedures by which a server can do an RPC to a client.
    - Clearly, callbacks also require a server to keep track of its clients.

Communication

- As with processes, there is nothing particularly special or unusual about communication in distributed file systems.
  - Many of them are based on remote procedure calls (RPCs).
  - The main reason for choosing an RPC mechanism is to make the system independent from underlying operating systems, networks, and transport protocols.

Remote Procedure Calls in NFS

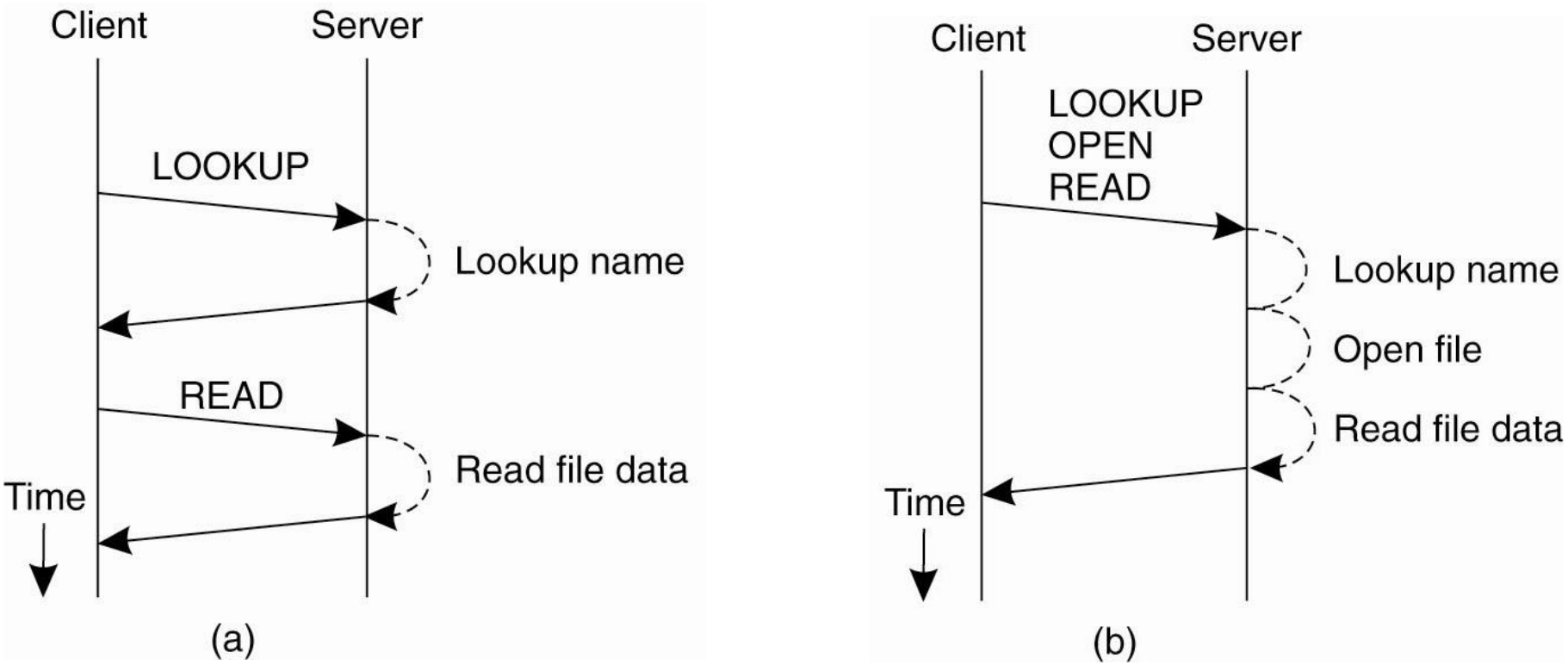

Figure 11-7. (a) Reading data from a file in NFS version 3. (b) Reading data using a compound procedure in version 4.

Communication

- The RPC2 subsystem.
  - RPC2 is a package that offers reliable RPCs on top of the (unreliable) UDP protocol.
  - Each time a remote procedure is called, the RPC2 client code starts a new thread that sends an invocation request to the server and subsequently blocks until it receives an answer.
  - As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request.
  - If the server dies, sooner or later this thread will notice that the messages have ceased and report back failure to the calling application.

Communication

- The RPC2 subsystem.
  - An interesting aspect of RPC2 is its support for side effects.
  - A side effect is a mechanism by which the client and server can communicate using an application-specific protocol.
    - Consider a client opening a file at a video server.
    - What is needed in this case is that the client and server set up a continuous data stream with an isochronous transmission mode.
    - In other words, data transfer from the server to the client is guaranteed to be within a minimum and maximum end-to-end delay.
    - It allows the client and the server to set up a separate connection for transferring the video data to the client on time. Connection setup is done as a side effect of an RPC call to the server.

The RPC2 Subsystem

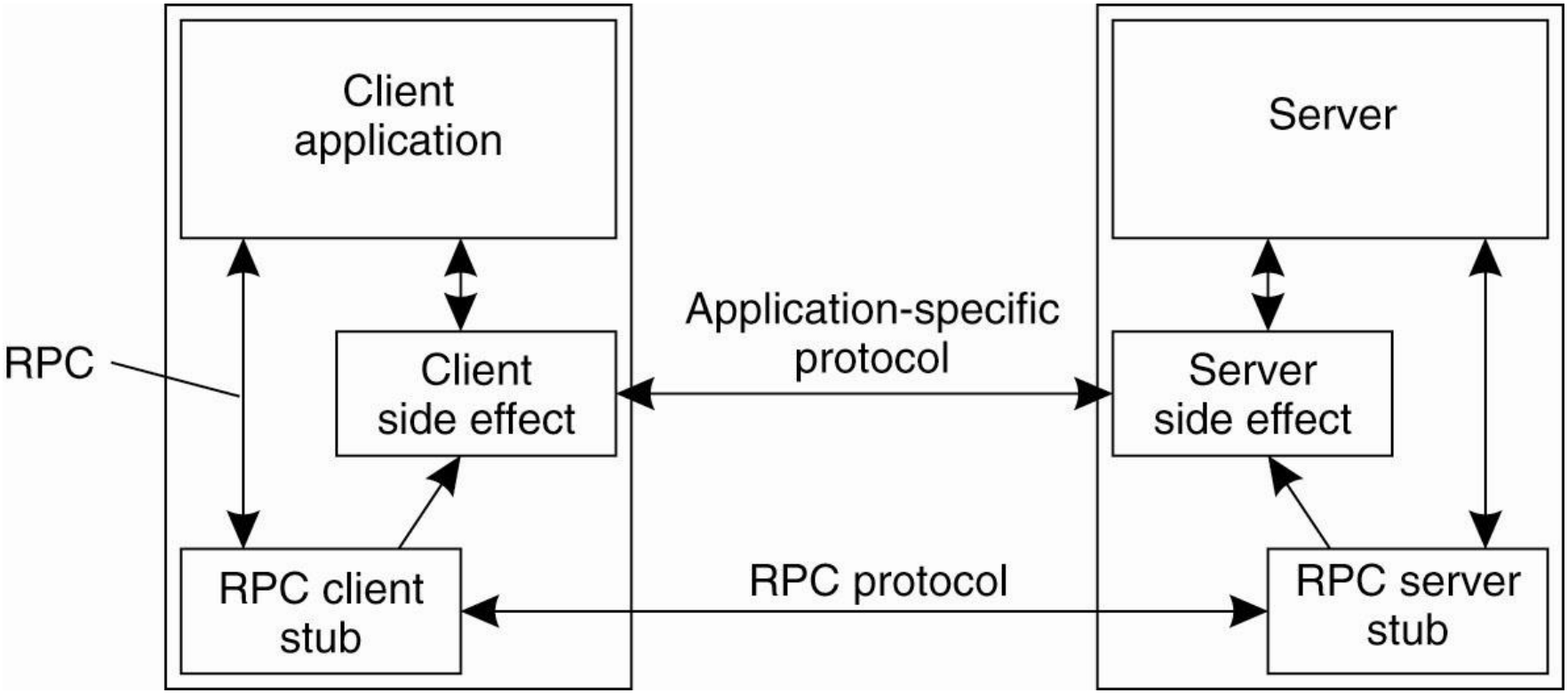

Figure 11-8. Side effects in Coda's RPC2 system.

Communication

- The RPC2 subsystem.
  - Another feature of RPC2 that makes it different from other RPC systems is its support for multicasting.
  - This feature allows that servers keep track of which clients have a local copy of a file.
  - When a file is modified, a server invalidates local copies by notifying the appropriate clients through an RPC.
  - Clearly, if a server can notify only one client at a time, invalidating all clients may take some time, so this feature enables servers invalidate all multiple clients at once.

The RPC2 Subsystem

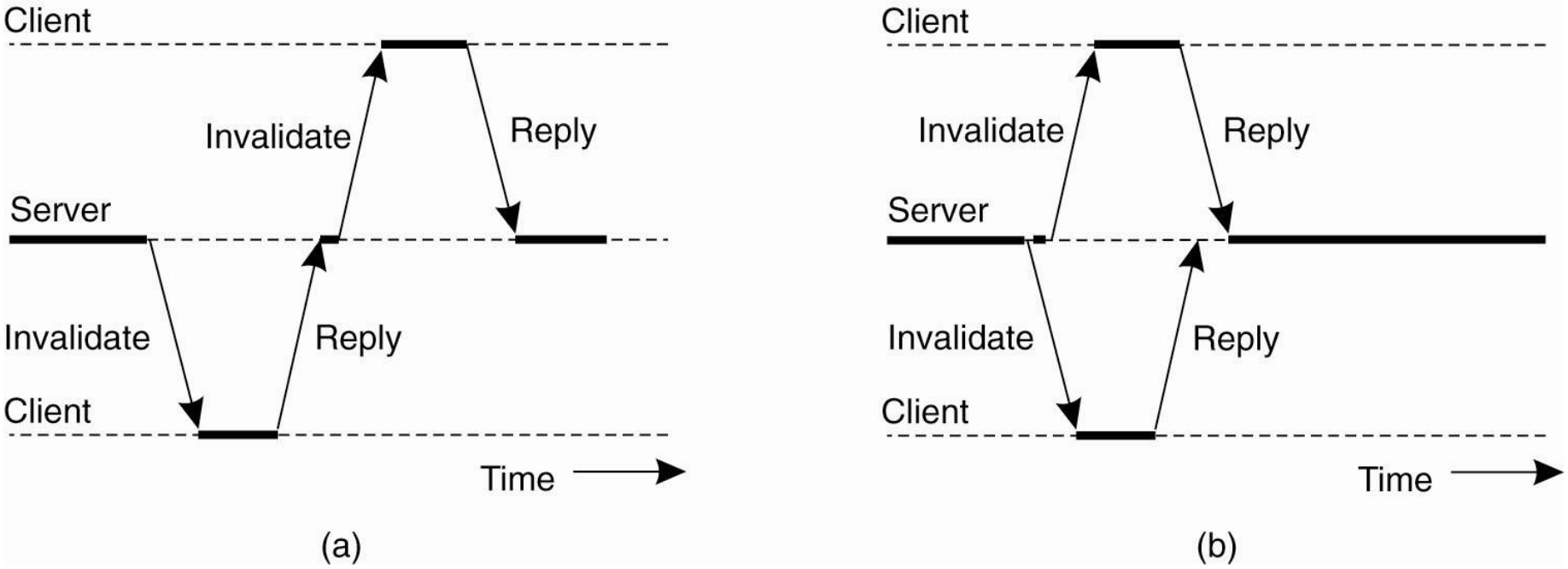

Figure 11-9. (a) Sending an invalidation message one at a time. (b) Sending invalidation messages in parallel.

Naming

- Naming arguably plays an important role in distributed file systems.
- In virtually all cases, names are organized in a hierarchical name space (see Chap. 5.)
- In the following we will again consider NFS as a representative for how naming is often handled in distributed file systems.

Naming

- Naming in NFS
  - The fundamental idea underlying the NFS naming model is to provide clients complete transparent access to a remote file system as maintained by a server.
  - This transparency is achieved by letting a client be able to mount a remote file system into its own local file system.

Naming in NFS

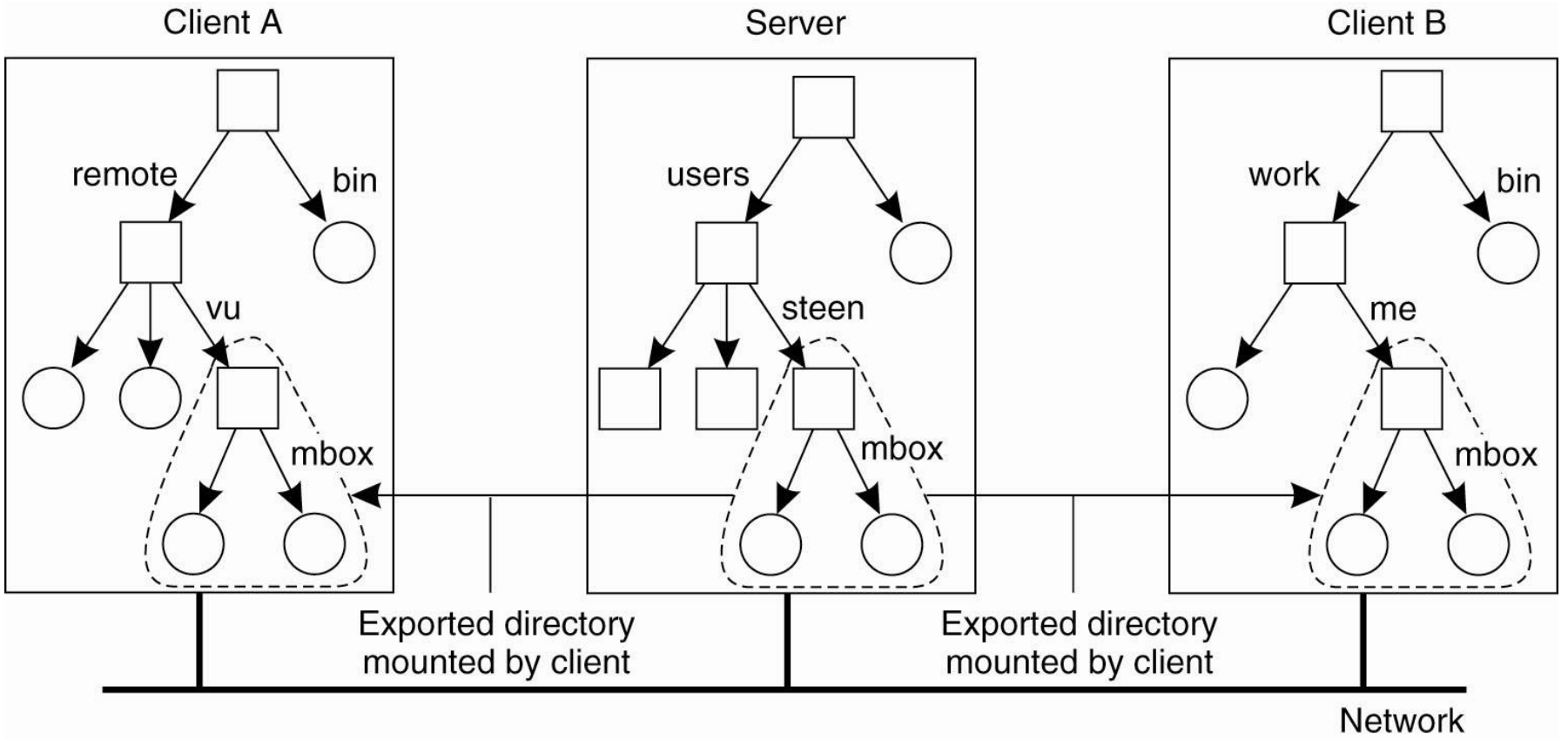

Figure 11-11. Mounting (part of) a remote file system in NFS.

Naming

- Naming in NFS
  - There is a serious implication related to how namespace is seen in different clients.
    - In principle, users do not share name spaces.
    - As can be seen (previous slides), the file named **/remote/vu/mbox** at client A is named **/work/me/mbox** at client B.
    - A file's name therefore depends on how clients organize their own local name space, and where exported directories are mounted.
    - The drawback of this approach in a distributed file system is that sharing files becomes much harder.
      - For example, Alice cannot tell Bob about a file using the name she assigned to that file, for that name may have a completely different meaning in Bob's name space of files.

Naming in NFS

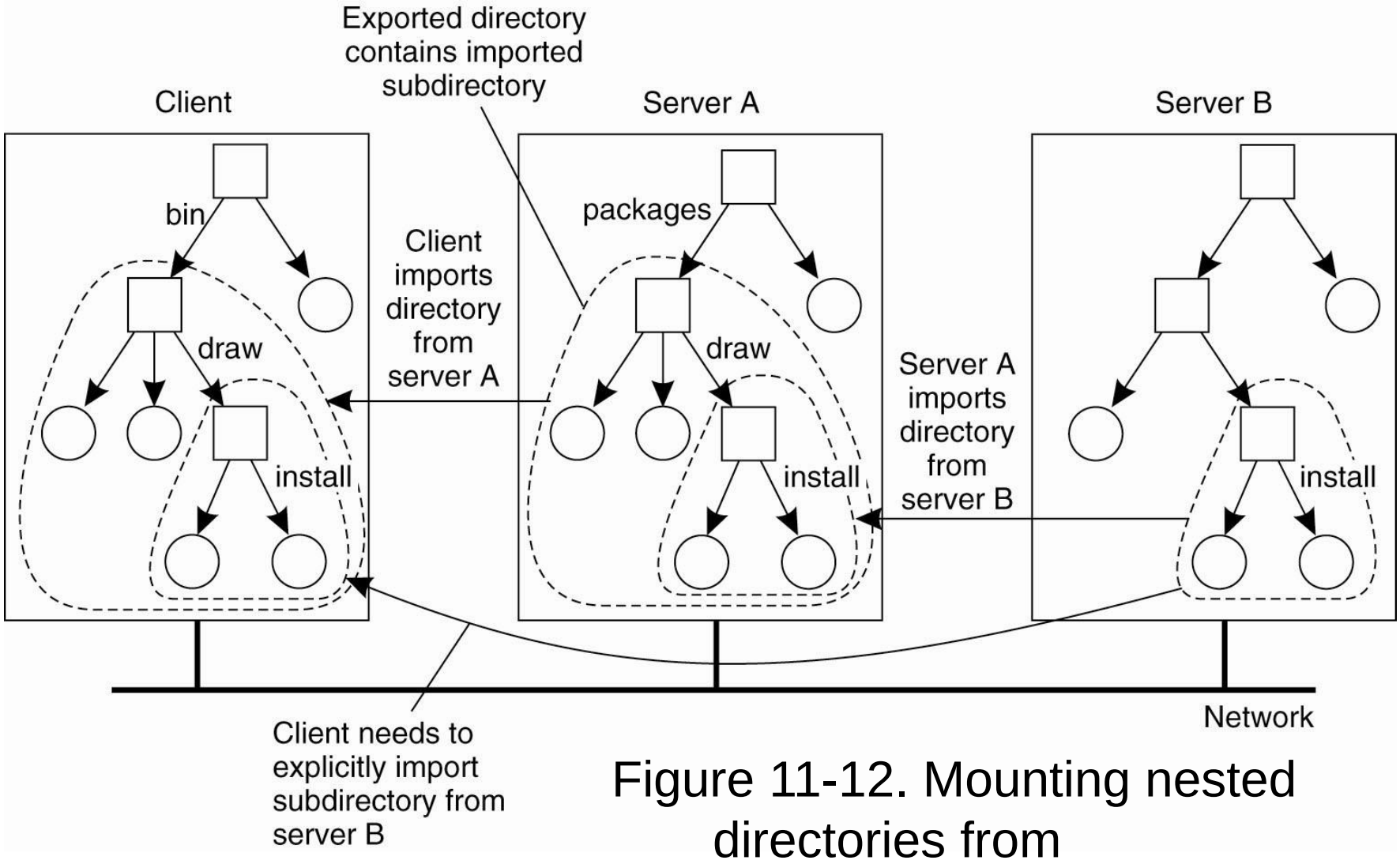

Figure 11-12. Mounting nested directories from multiple servers in NFS.

Naming

- Naming in NFS
  - The NFS naming model essentially provides users with their own name space. Sharing in this model may become difficult if users name the same file differently.
  - One solution to this problem is to provide each user with a local name space that is partly standardized, and subsequently mounting remote file systems the same for each user.

Naming

- NFS: Automounting
  - Another problem with the NFS naming model has to do with deciding when a remote file system should be mounted.
  - Consider a large system with thousands of users.
    - Assume that each user has a local directory **/home** that is used to mount the *home* directories of other users.
    - For example, Alice's home directory may be locally available to her as **/home/alice**, although the actual files are stored on a remote
    - server.
    - This directory can be automatically mounted when Alice logs into her workstation.
    - In addition, she may have access to Bob's public files by accessing Bob's directory through **/home/bob**.
    - The question is whether Bob's home directory should also be mounted automatically when Alice logs in.

Automounting

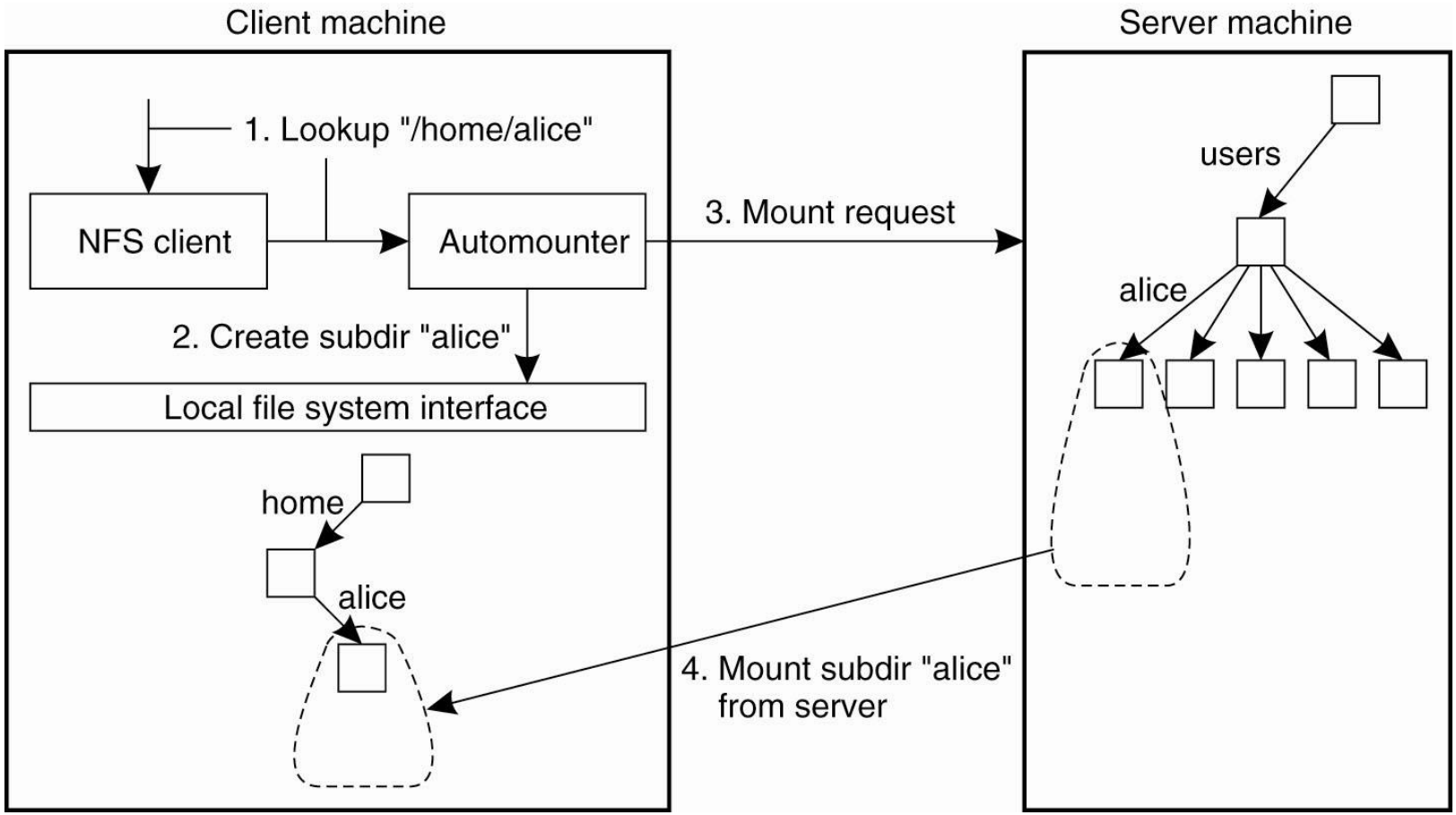

Figure 11-13. A simple automounter for NFS.

Synchronization

- Synchronization for file systems would not be an issue if files were not shared.
- However, in a distributed system, the semantics of file sharing becomes a bit tricky when performance issues are at stake.
- To this end, different solutions have been proposed.

Synchronization

- Semantics of File Sharing
  - When two or more users share the same file at the same time, it is necessary to define the semantics of reading and writing precisely to avoid problems.
  - In single-processor systems that permit processes to share files, such as UNIX, the semantics normally state that when a read operation follows a **write** operation, the **read** returns the value just written.
  - Similarly, when two **writes** happen in quick succession, followed by a **read**, the value read is the value stored by the last **write**.

Semantics of File Sharing

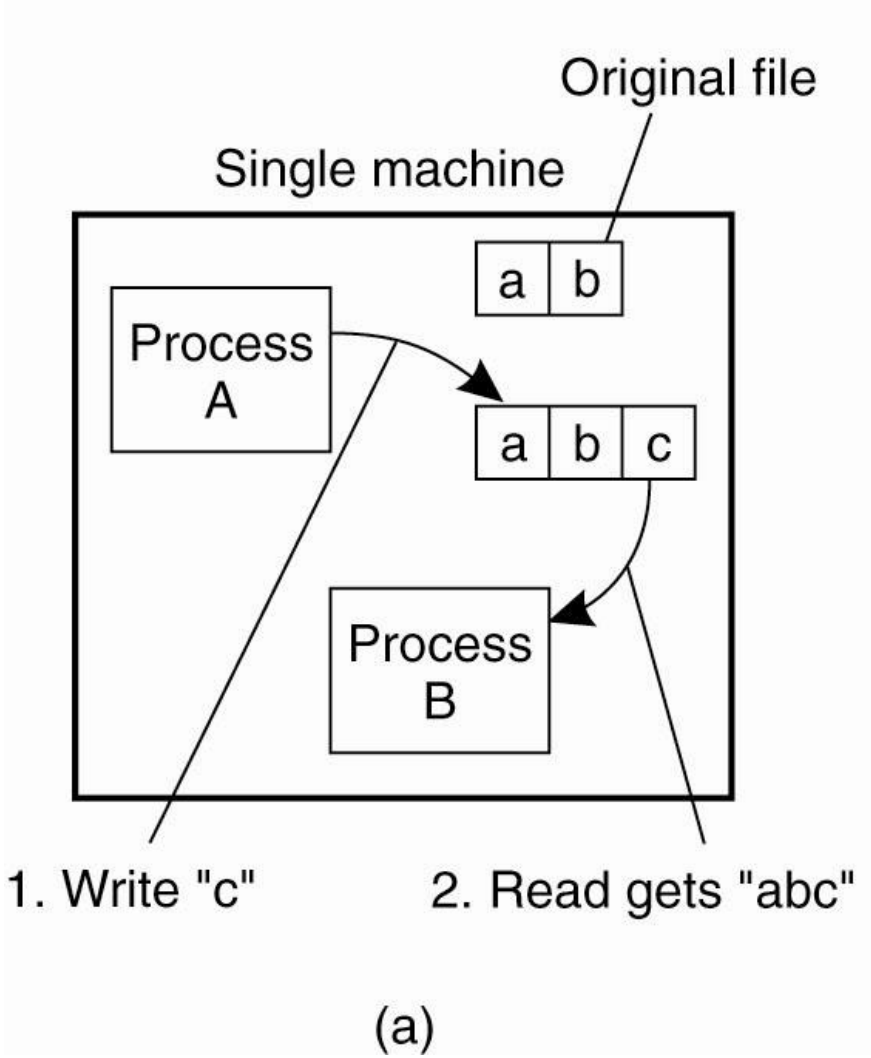

Figure 11-16. (a) On a single processor, when a read follows a write, the value returned by the read is the value just written.

Semantics of File Sharing

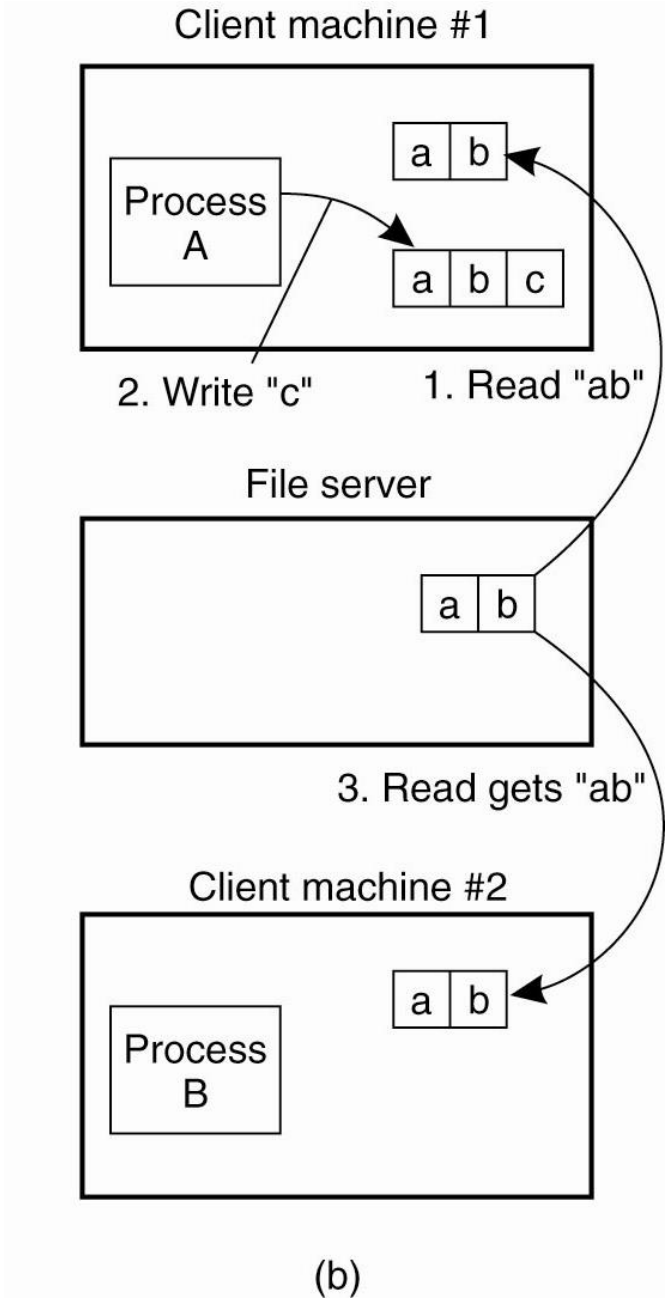

Figure 11-16. (b) In a distributed system with caching, obsolete values may be returned.

Semantics of File Sharing

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
Transactions	All changes occur atomically

Figure 11-17. Four ways of dealing with the shared files in a distributed system.

File Locking

- In client-server architectures with stateless servers, we need additional facilities for synchronizing access to shared files.
- The traditional way of doing this is to make use of a lock manager.

File Locking

- NFS4
  - Conceptually, file locking in NFSv4 is simple.
  - There are essentially only four operations related to locking (see next slide).
  - NFS4 distinguishes **read** locks from **write** locks.
  - Multiple clients can simultaneously access the same part of a file provided they only **read** data.
  - A **write** lock is needed to obtain exclusive access to modify part of a file.
  - Locks are granted for a specific time (determined by the server).
    - In other words, they have an associated **lease**.
    - Unless a client renews the lease on a lock it has been granted, the server will automatically remove it.
    - This approach is followed for other server-provided resources as well and helps in recovery after failures.
    - Using the **renew** operation, a client requests the server to renew the lease on its lock (and, in fact, other resources as well).

File Locking

Operation	Description
Lock	Create 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 lease on a specified lock

Figure 11-18. NFSv4 operations related to file locking.

Consistency & Replication

- Caching and replication play an important role in distributed file systems, most notably when they are designed to operate over wide-area networks.

Consistency & Replication

- Client-side Caching (in NFS)
  - Caching in NFSv3 has been mainly left outside of the protocol.
    - This approach has led to the implementation of different caching policies, most of which never guaranteed consistency.
    - At best, cached data could be stale for a few seconds compared to the data stored at a server.
    - However, implementations also exist that allowed cached data to be stale for 30 seconds without the client knowing.

Consistency & Replication

- Client-side Caching (in NFS)
  - NFSv4 solves some of caching consistency problems, but essentially still leaves cache consistency to be handled in an implementation-dependent way.
    - Each client can have a memory cache that contains data previously read from the server.
    - In addition, there may also be a disk cache that is added as an extension to the memory cache, using the same consistency parameters.
    - Typically, clients cache file data, attributes, file handles, and directories.
    - Different strategies exist to handle consistency of the cached data, cached attributes, and so on.

Client-Side Caching

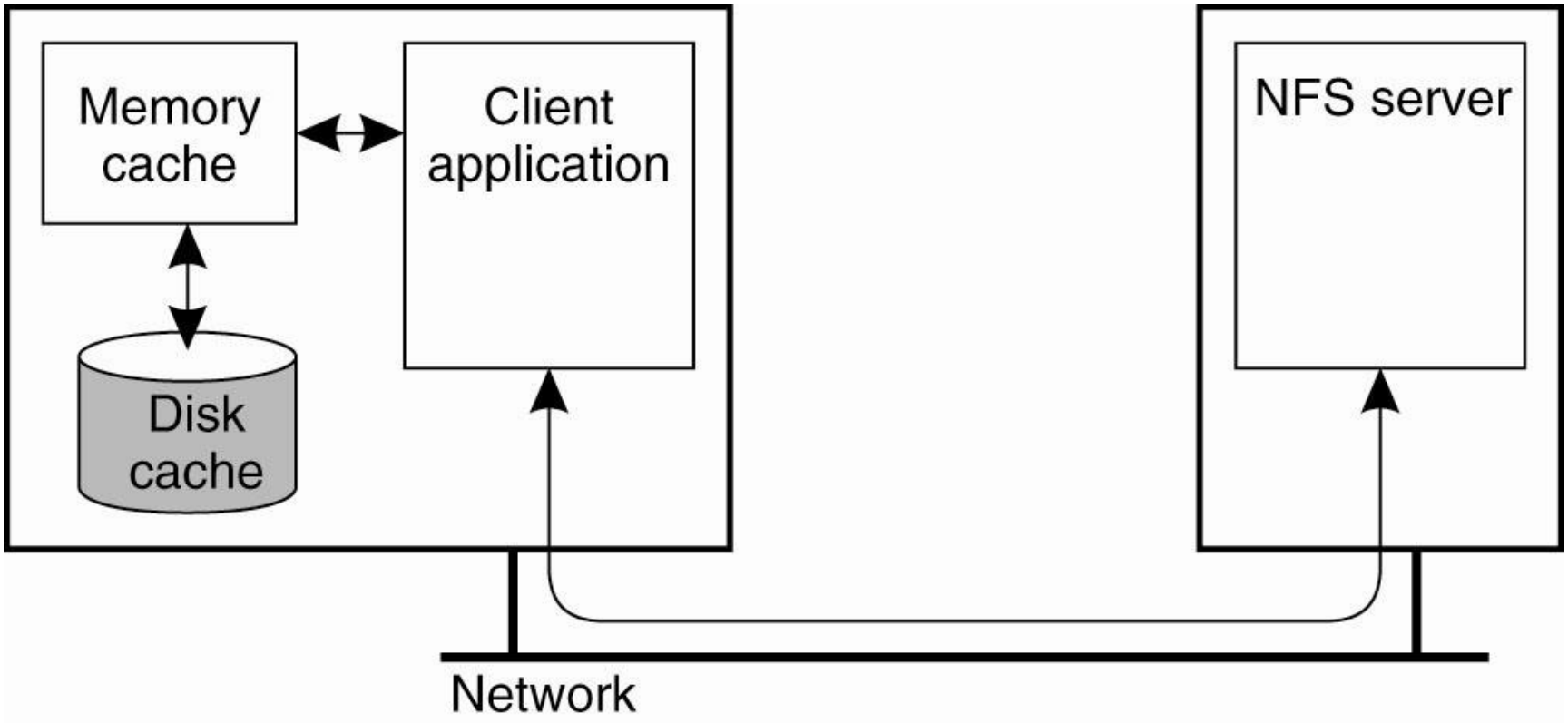

Figure 11-21. Client-side caching in NFS.

Client-Side Caching

- Client-side Caching (in NFS4)
  - NFSv4 solves some of caching consistency problems, but essentially still leaves cache consistency to be handled in an implementation-dependent way.
    - It supports two different approaches for caching file data.
    - The simplest approach is when a client opens a file and caches the data it obtains from the server as the result of various read operations.
    - In addition, write operations can be carried out in the cache as well. When the client closes the file, NFS requires that any modifications on the local cached file must be flushed to the server.
    - Several clients on the same machine can share a single cache.
    - Once (part of) a file has been cached, a client can keep its data in the cache even after closing the file.
      - ✓ NFS requires that whenever a client opens a previously closed file that has been (partly) cached, the client must immediately revalidate the cached data.
      - ✓ Revalidation takes place by checking when the file was last modified and invalidating the cache in case it contains stale data.

Client-Side Caching

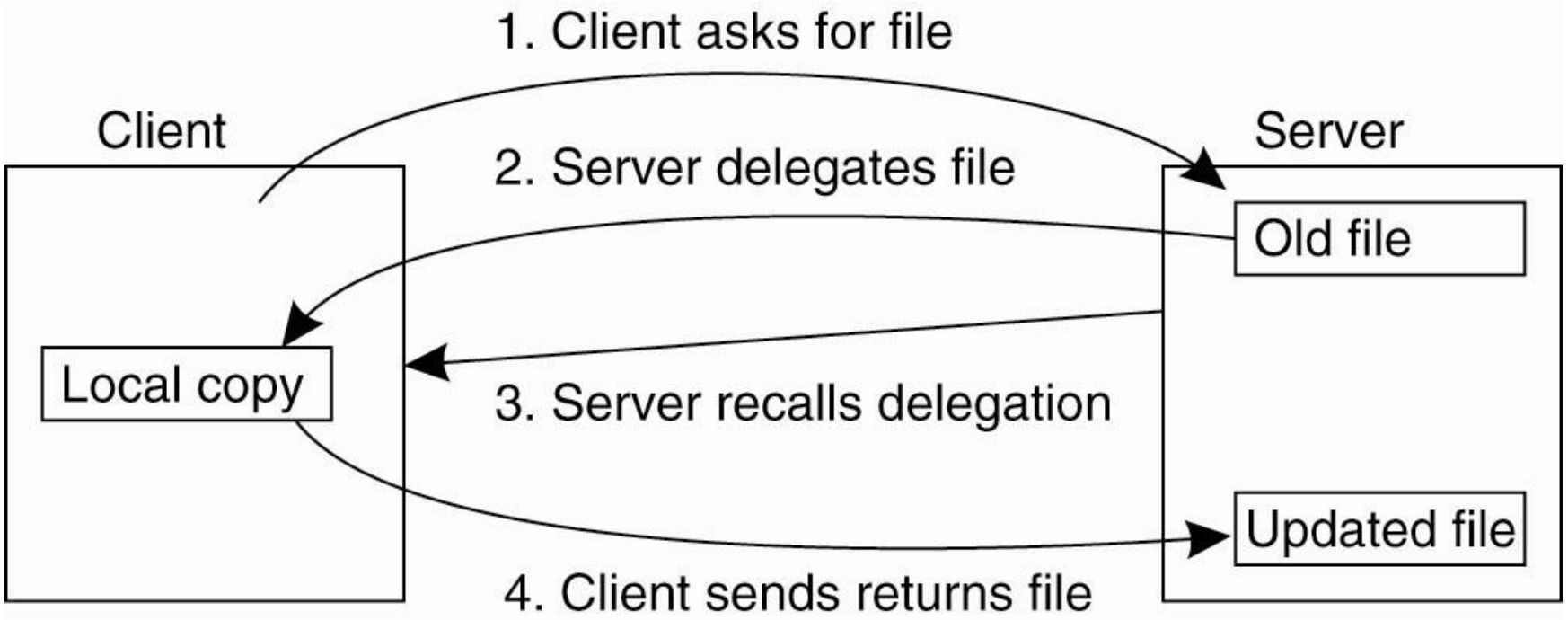

Figure 11-22. Using the NFSv4 callback mechanism to recall file delegation.

Security

- Security in DFS's organized along a client-server architecture is to have the servers handle authentication and access control.
- This is a straightforward way of dealing with security, an approach that has been adopted, for example, in systems such as NFS.

Security

- Secure in NFS
  - The basic idea behind a DFS is that a remote file system should be presented to clients as if it were a local file system.
  - Hence, it should come as no surprise that security in NFS mainly focuses on the communication between a client and a server.
  - Secure communication means that a secure channel between the two should be set up.
  - In addition to secure RPCs, it is necessary to control file accesses, which are handled by means of access control file attributes in NFS.
  - A file server is in charge of verifying the access rights of its clients.

Security in NFS

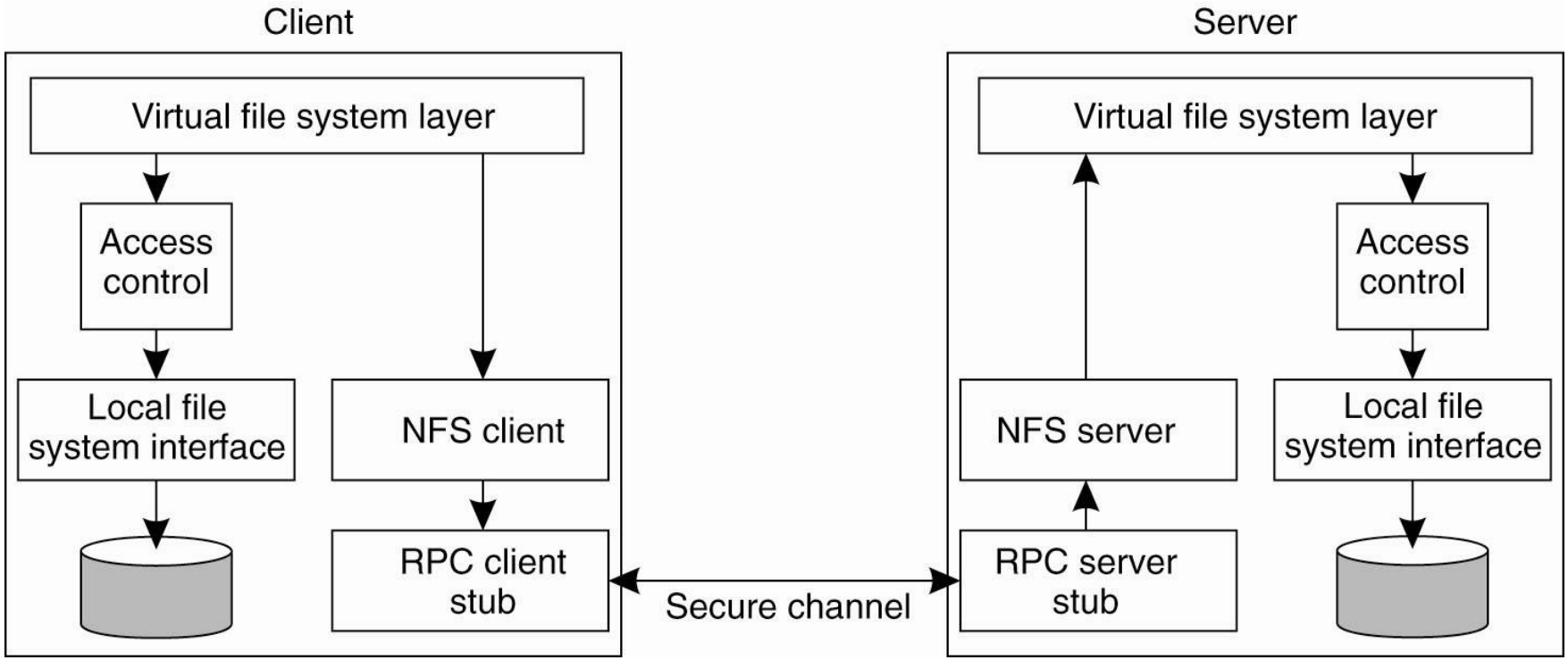

Figure 11-28. The NFS security architecture.

Secure RPCs

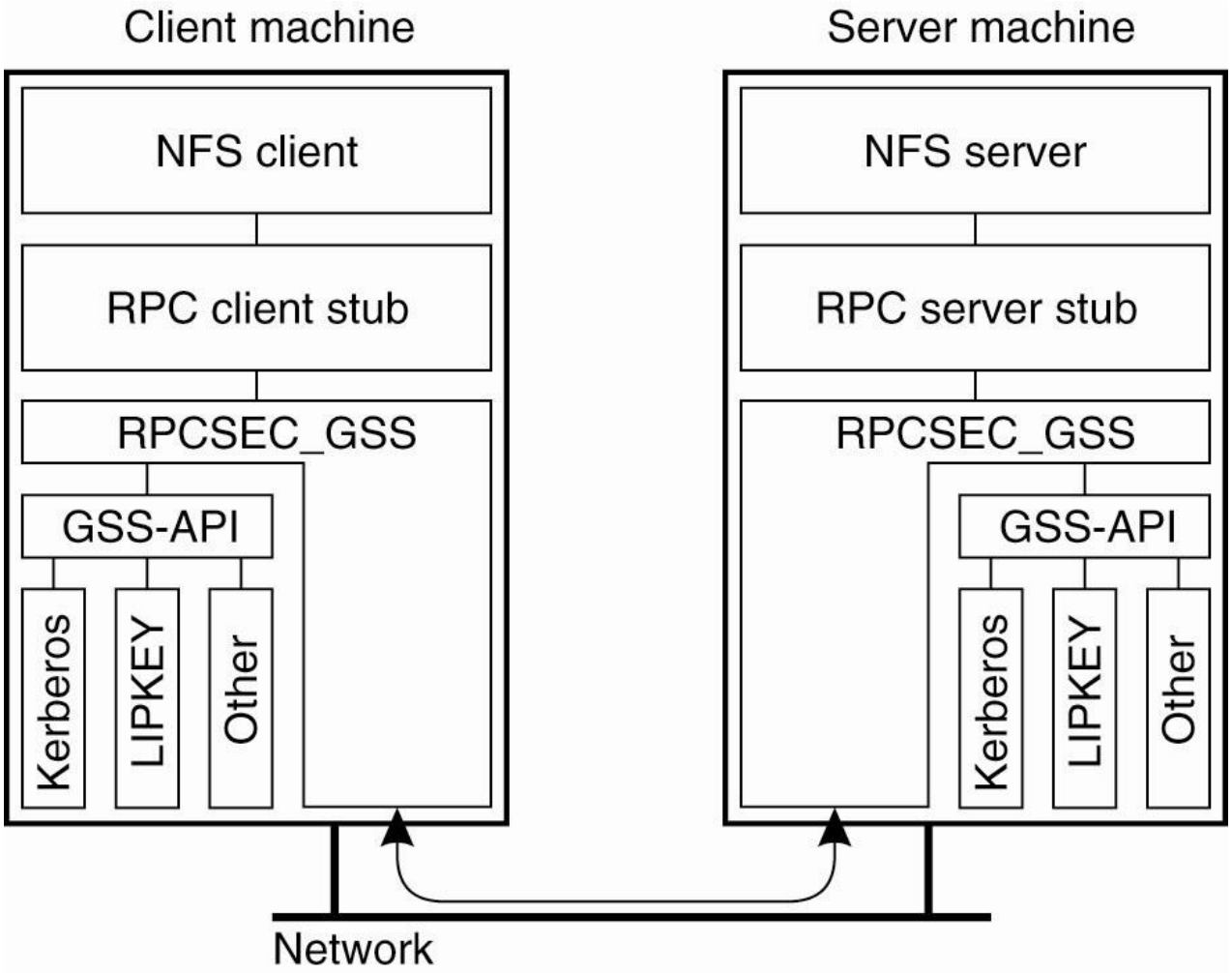

Figure 11-29. Secure RPC in NFSv4.

Security

- Access Control
  - Authorization in NFS is analogous to secure RPC: it provides the mechanisms but does not specify any particular policy.
  - Access control is supported by means of the ACL file attribute.
    - This attribute is a list of access control entries, where each entry specifies the access rights for a specific user or group.
    - Many of the operations that NFS distinguishes with respect to access control are relatively straightforward and include those for reading, writing, and executing files, manipulating
    - file attributes, listing directories, and so on.
  - The NFS model for access control has much richer semantics than most UNIX models.

Access Control

Type of user	Description
Owner	The owner of a file
Group	The group of users associated with a file
Everyone	Any user or process
Interactive	Any process accessing the file from an interactive terminal
Network	Any process accessing the file via the network
Dialup	Any process accessing the file through a dialup connection to the server
Batch	Any process accessing the file as part of batch job
Anonymous	Anyone accessing the file without authentication
Authenticated	Any authenticated user or process
Service	Any system-defined service process

Figure 11-30. The various kinds of users and processes distinguished by NFS with respect to access control.

Security

- Decentralized Authentication
  - One of the main problems with systems such as NFS is that in order to properly handle authentication, it is necessary that users are registered through a central system administration.
  - A solution to this problem is provided by using the Secure File Systems (SFS) in combination with decentralized authentication servers.
  - The basic idea is quite simple:
    - What other systems lack is the possibility for a user to specify that a remote user has certain privileges on his files.
    - In virtually all cases, users must be globally known to all authentication servers.
    - A simpler approach would be to let Alice specify that "Bob, whose details can be found at X," has certain privileges.
    - The authentication server that handles Alice's credentials could then contact server X to get information on Bob.

Decentralized Authentication

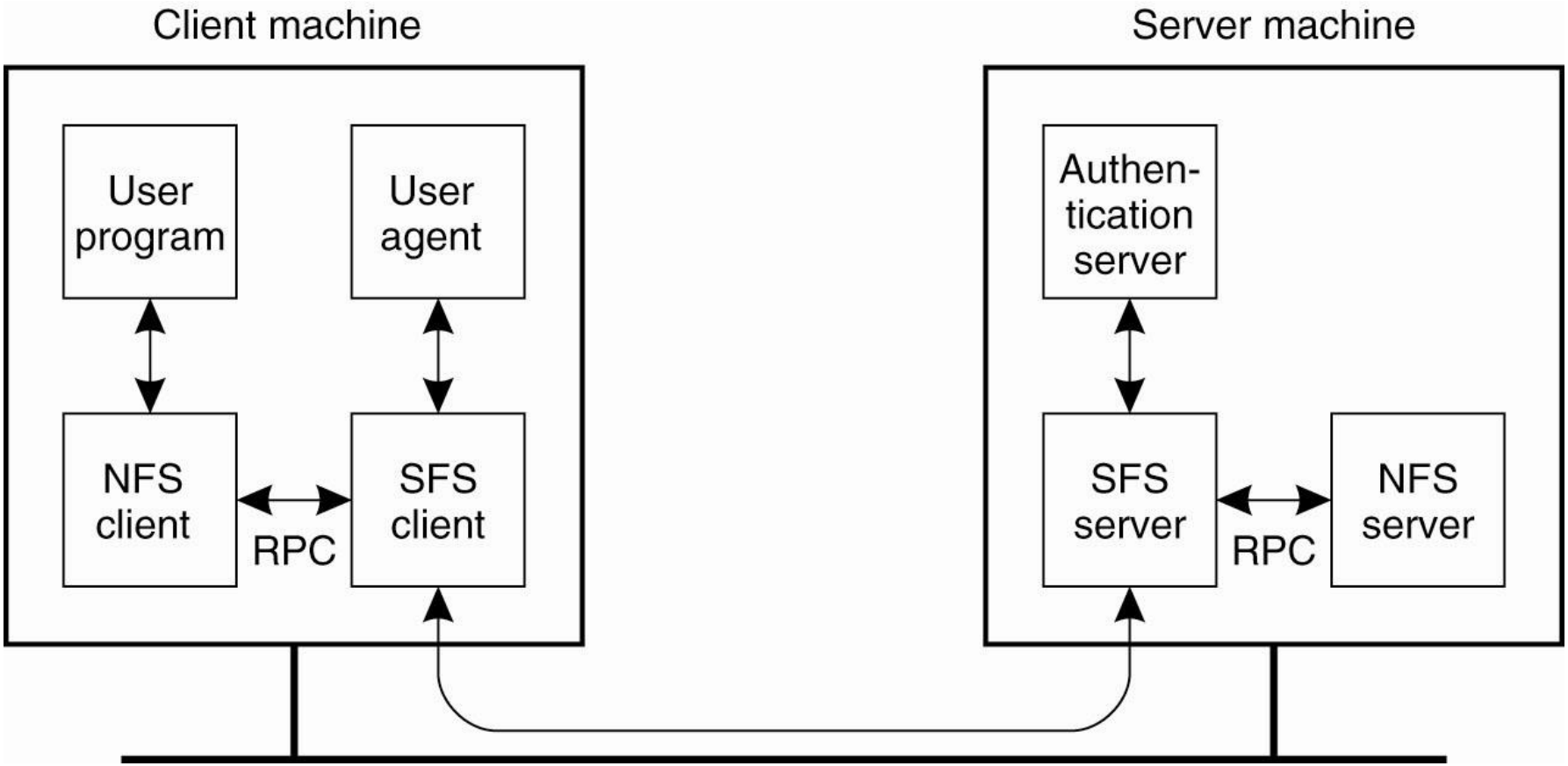

Figure 11-31. The organization of SFS.