Distributed Systems Lecture 15

Professor Louise E. Moser

Spring 2013

Distributed File Systems

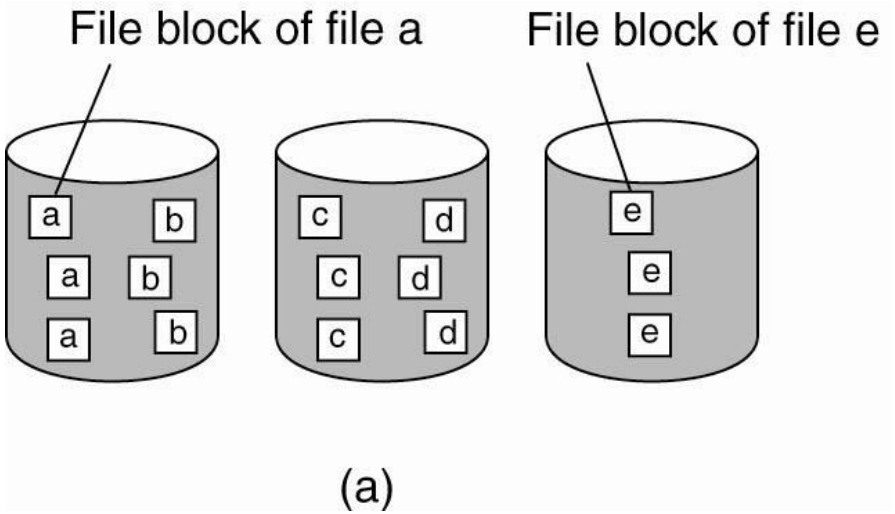
- Cluster-Based Distributed File Systems, such as the Google File System (GFS)
- Network File System (NFS), originally developed by Sun Microsystems
- Coda, based on the Andrew File System (AFS) at Carnegie Mellon University (CMU)

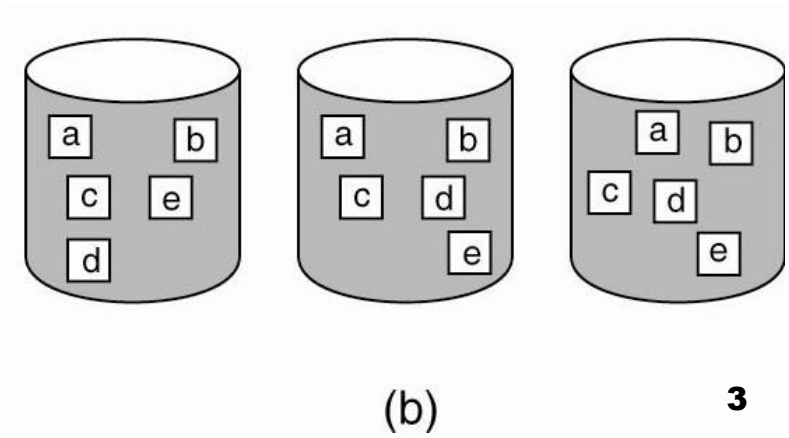
Cluster-Based Distributed File Systems

- Two different ways of distributing blocks of files in a distributed file system

  (a) Distributing blocks of *different* files across different servers, with all of the blocks of a file at the same server

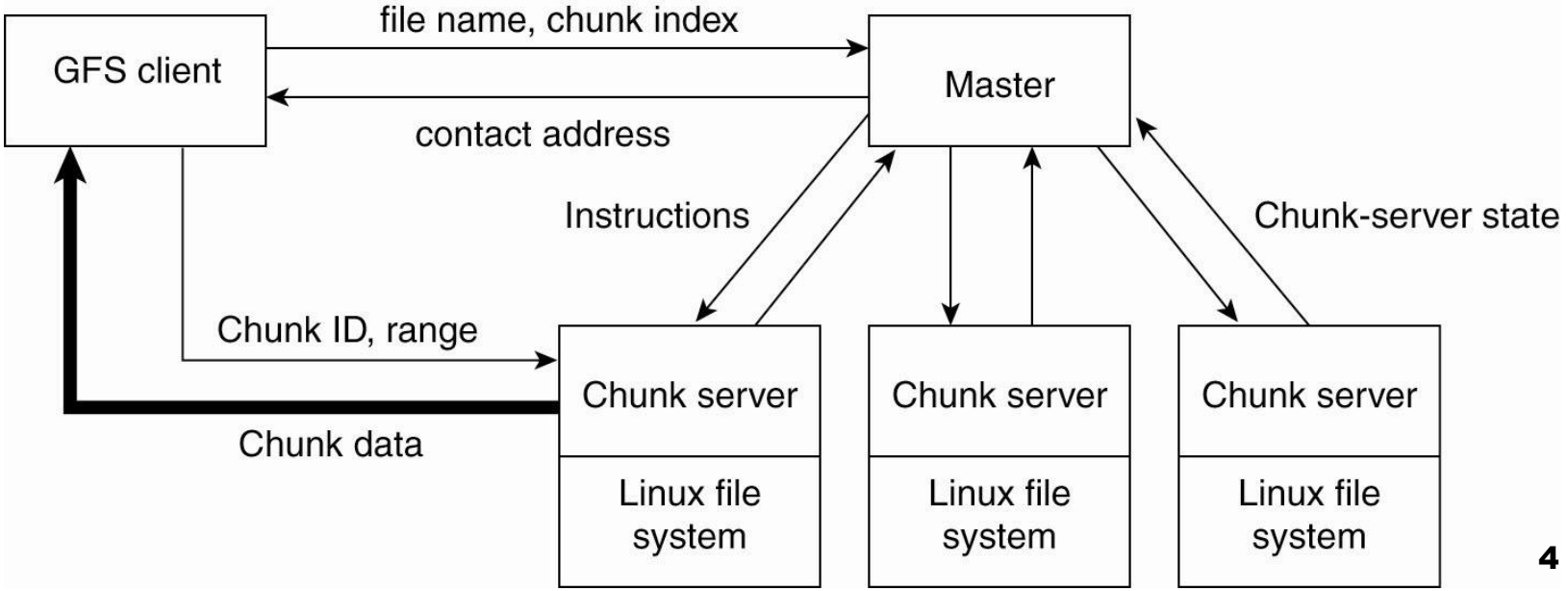
  (b) Striping files, i.e., distributing blocks of the *same* file across different servers, so that different blocks of the same file can be accessed in parallel





Cluster-Based Distributed File Systems

- **Google File System (GFS) uses striping**
  - A GFS file is divided into 64 Mbyte chunks; each chunk has an identifier
  - The chunks of a file are distributed to Chunk servers
  - The GFS Master maintains a table with the chunk identifiers and their locations, and also metadata for the files



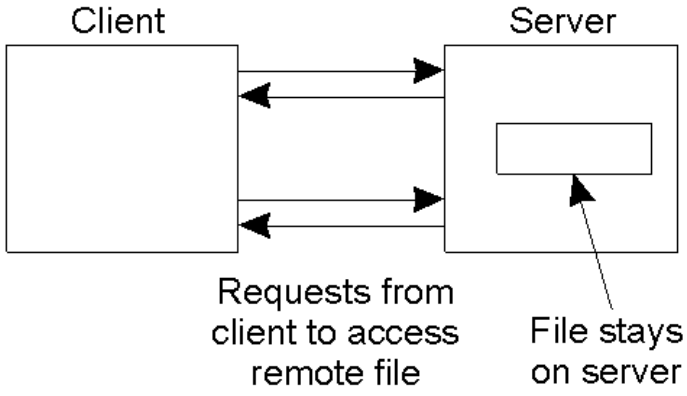
Global Name Space

- **Global Name Space (GNS)** decouples the naming of files from their locations and, thus, allows files to be moved without making their names unresolvable
- GNS maintains a virtual tree in which each node is either a directory or a junction
- A **junction** indicates that name resolution is to be taken over by another process and is somewhat like a mount point

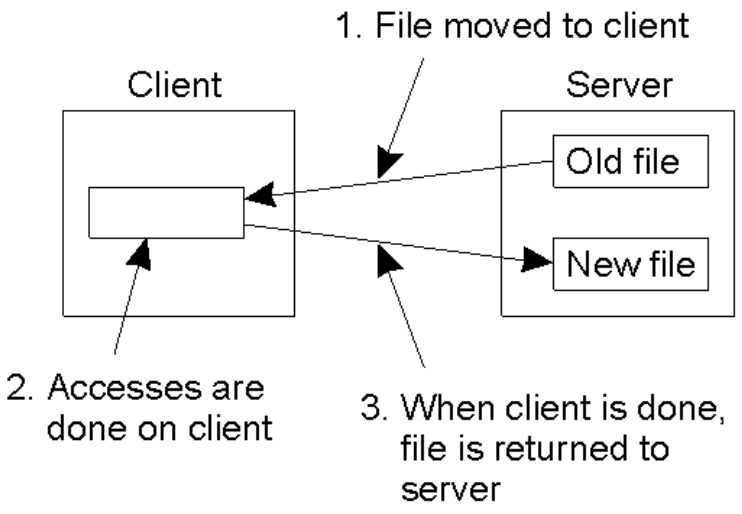
Junction	Description
GNS junction	Refers to another GNS instance
Logical file-system name	Reference to subtree to be looked up in a location service
Logical file name	Reference to a file to be looked up in a location service
Physical file-system name	Reference to directly remote-accessible subtree
Physical file name	Reference to directly remote-accessible file

Network File System

- NFS originated with diskless workstations at Sun Microsystems, based on a **remote file system** that is transparently available to remote clients
- Follows **(a)** the **remote access model**, rather than **(b)** the **upload/download model** (caching) which is better for mobile devices



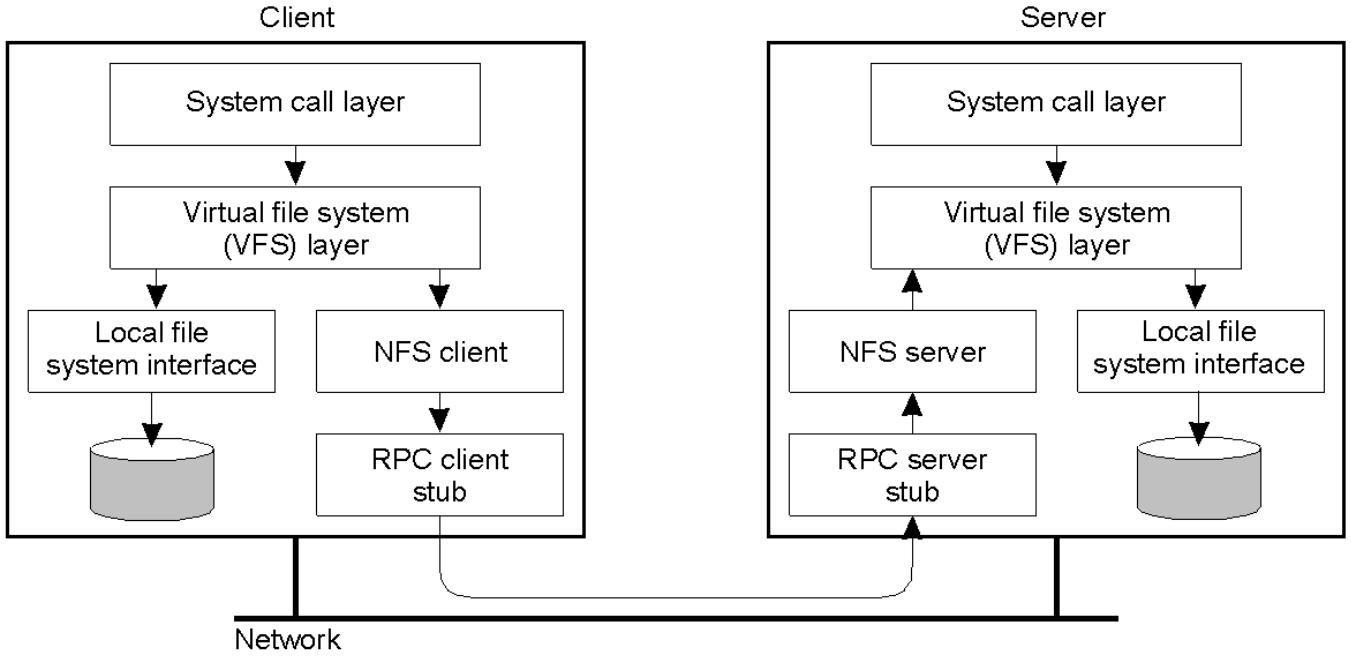
(a)



(b)

Network File System

- NFS uses the **Virtual File System (VFS)** abstraction of Unix, which is now used by many different operating systems
- VFS provides a standard file system interface that hides the difference between local and remote file access



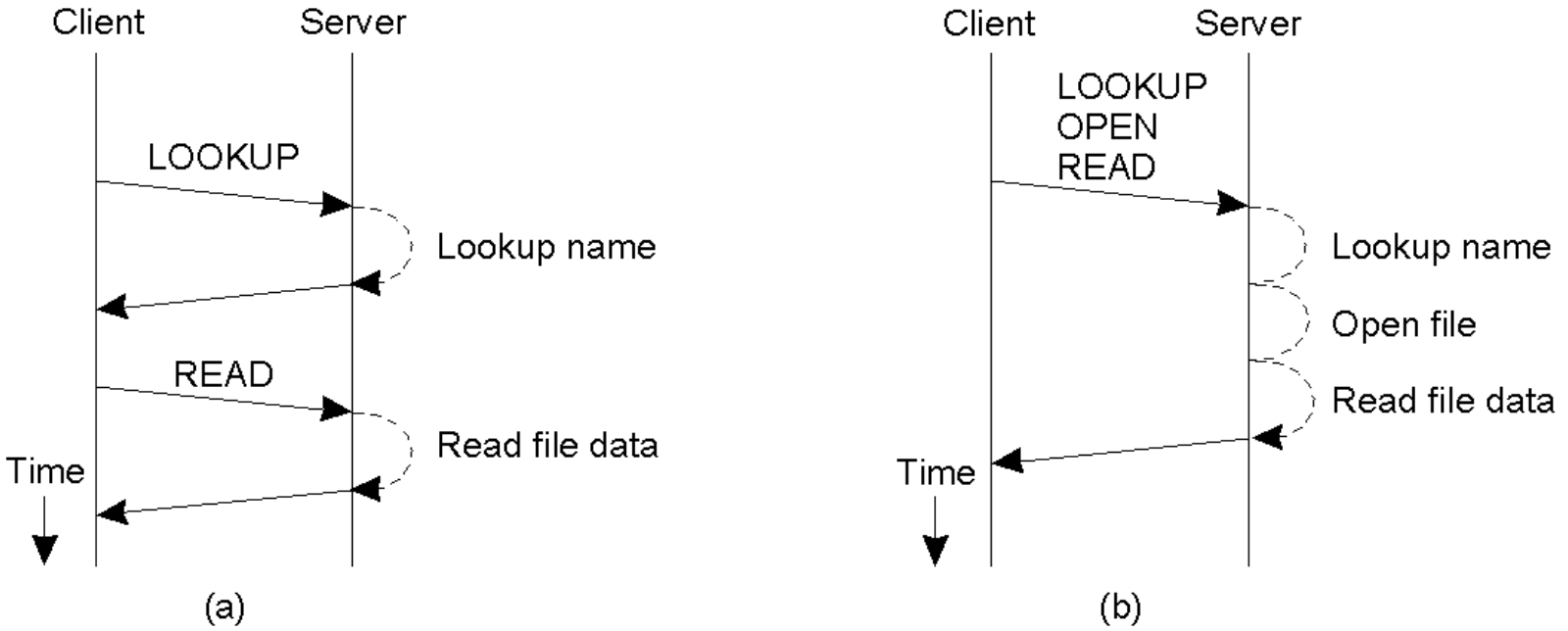
NFS Operations

- Some file system operations supported by NFS

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a non-regular 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

Communication

- Communication in NFS is best-effort Open Network Computing (ONM) **Remote Procedure Call (RPC)**
- NFSv4 also supports **compound procedures**

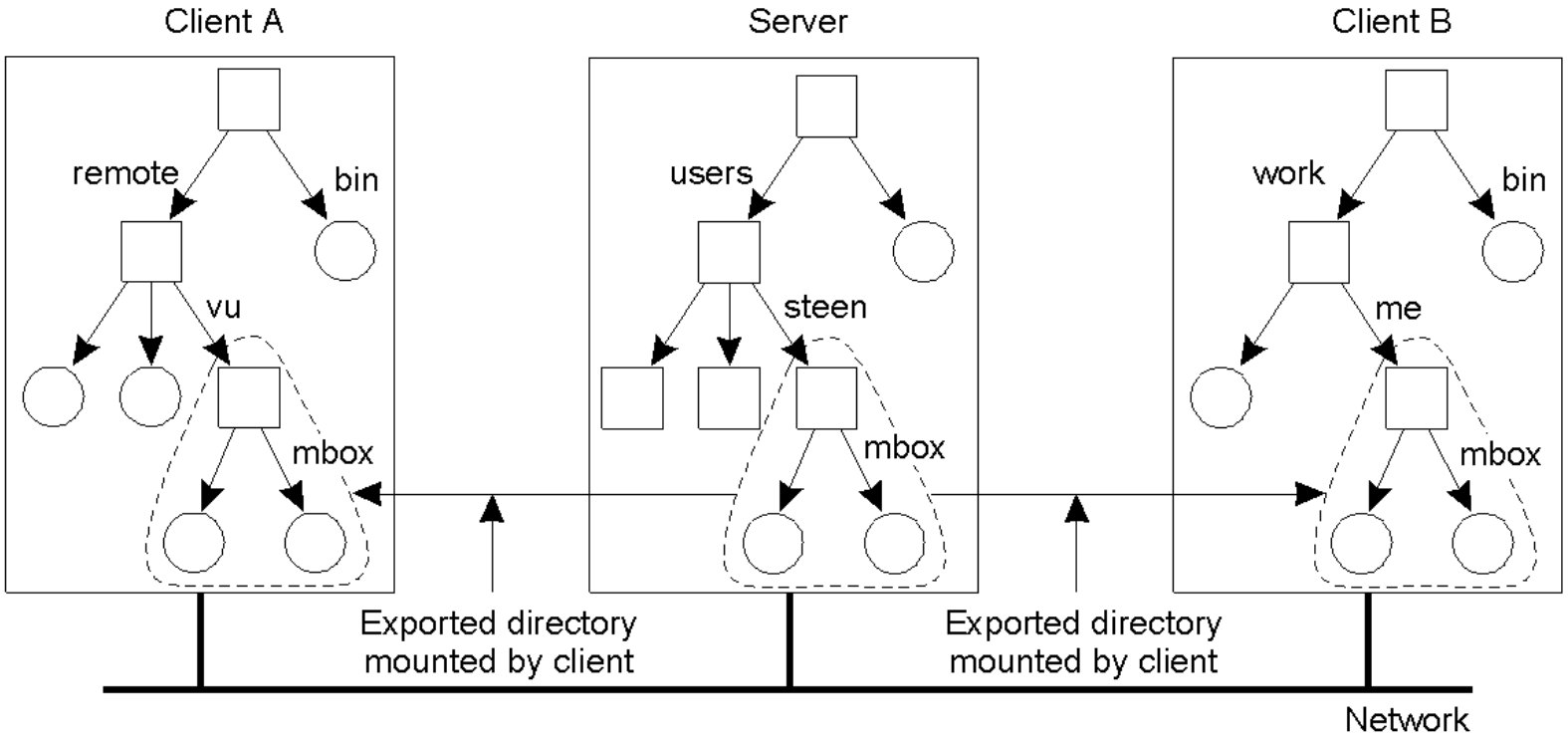


(a) Reading data from a file in NFSv3

(b) Reading data using a compound procedure in NFSv4
The first failure breaks execution of the rest of the RPC

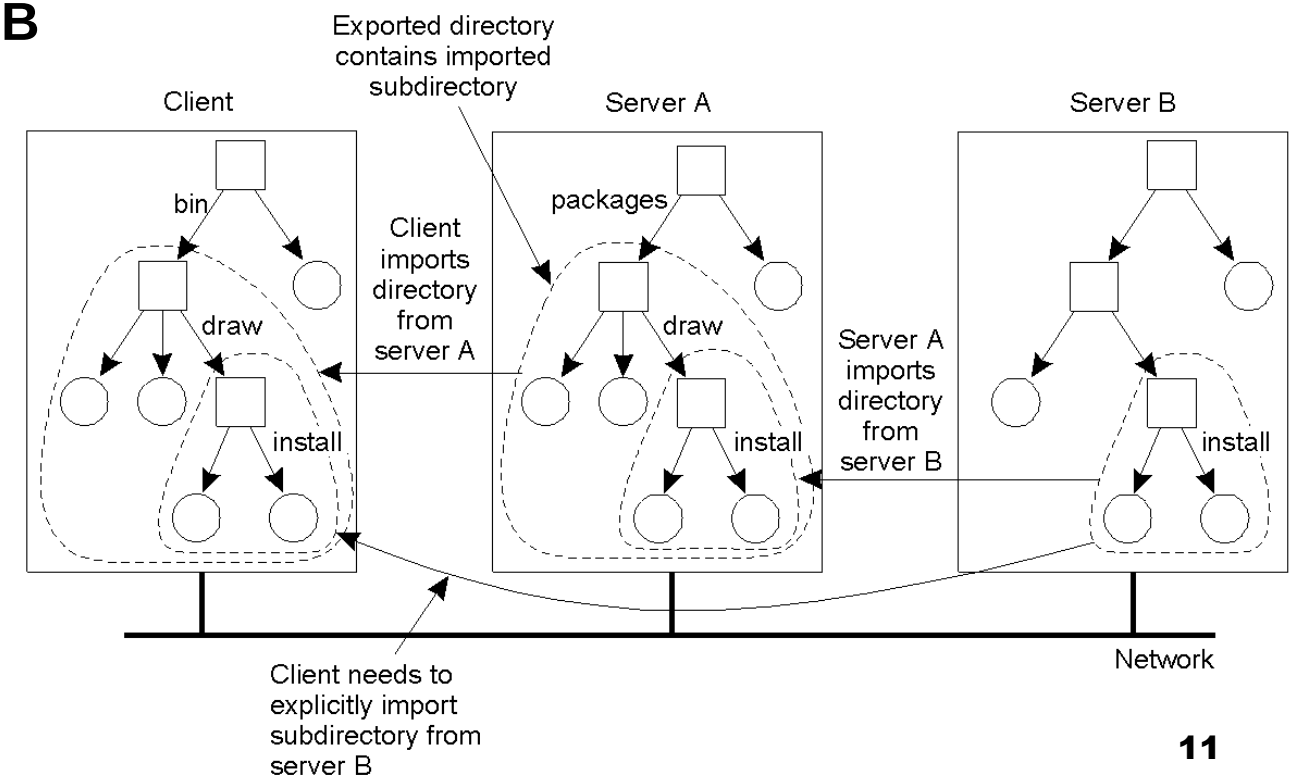
Naming and Mounting

- NFS supports mounting of remote file systems and directories into a client's local name space
- Typically, directories are copies but files are not
- Different clients may have different local name spaces,



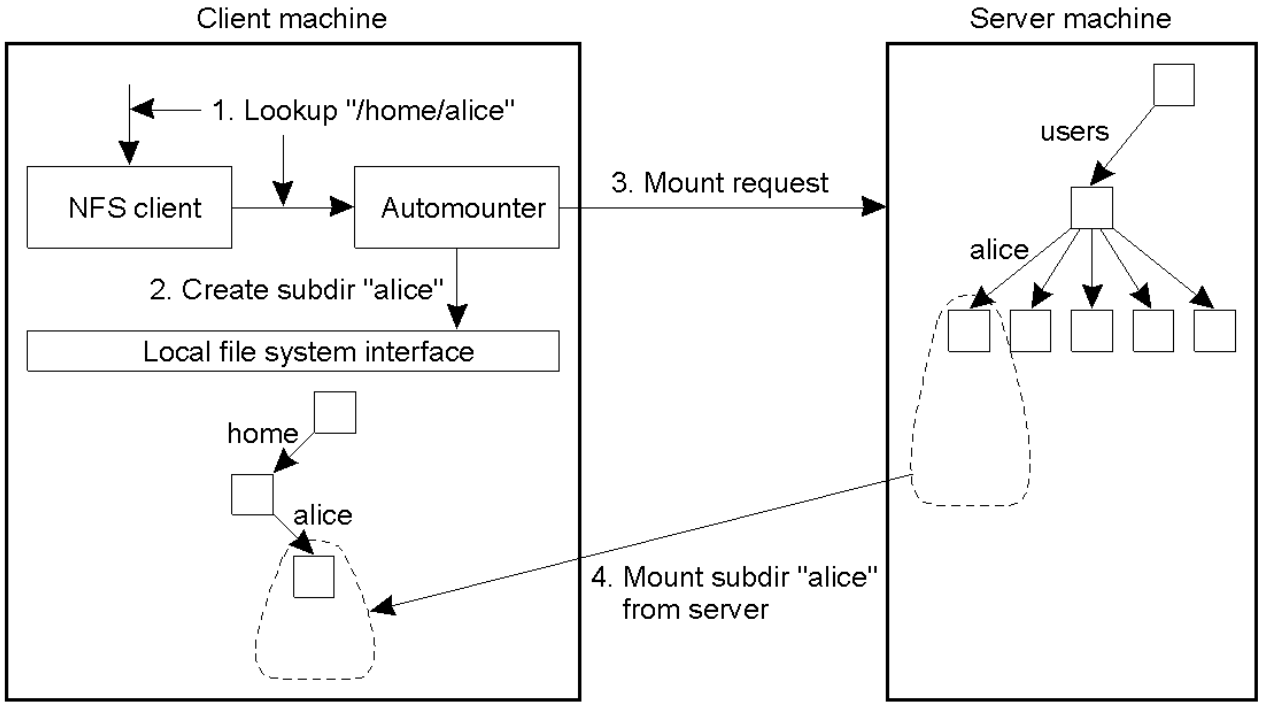
Naming and Mounting

- Mounting nested directories from multiple servers in NFS
  - **A server A cannot export a directory that it imported from another server B**
  - **A client must itself mount server A's imported directory from server B**



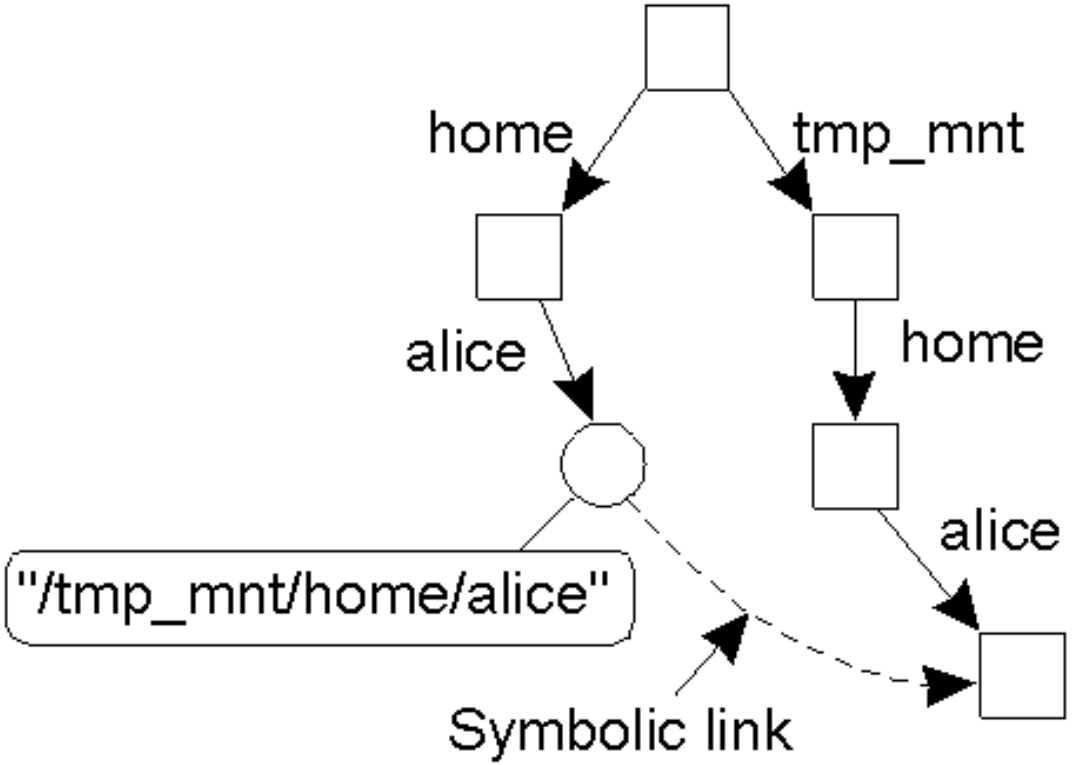
Auto-Mounting

- **Problem:** Mounting lots of subdirectories (e.g., all of the subdirectories in `/home/users`) can take a lot of time
- **Solution:** On login, **auto-mount** the user's home directory and subdirectories; on use, mount system directories or other users' files or directories using symbolic links



Auto-Mounting

- Using **temporary mount points** and **symbolic links** with auto-mounting in NFS



File Attributes

- Some **mandatory** file attributes in NFS

Attribute	Description
TYPE	Type of this file (regular, directory, symbolic link)
SIZE	Length of this file in bytes
CHANGE	Indicator that allows a client to see if and/or when this file changed
FSID	Server-unique identifier of the file system of this file

- Some **recommended** file attributes in NFS

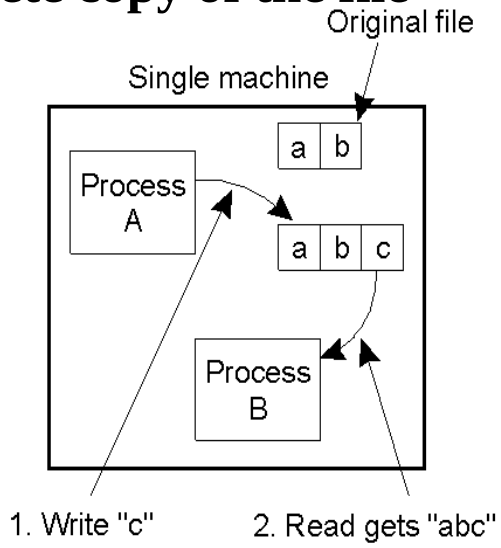
Attribute	Description
ACL	Access control list associated with this file
FILEHANDLE	Server-provided file handle for this file
FILEID	File-system unique identifier for this file
FS_LOCATIONS	Locations in the network where this file system can be found
OWNER	Character-string name of the file's owner
TIME_ACCESS	Time when the file data were last accessed
TIME_MODIFY	Time when the file data were last modified
TIME_CREATE	Time when this file was created

Semantics of File Sharing

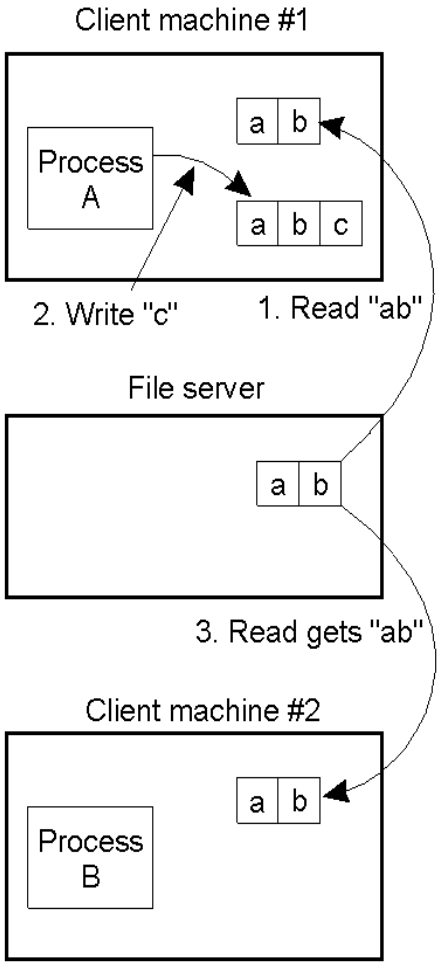
- **Problem:** With distributed file systems, we need to take into account the order of concurrent read/write operations, and the expected consistency semantics

  (a) On a single processor, when a *read* follows a *write*, the value returned by the *read* includes the value just written (for sequential file access)

  (b) In a distributed system with the upload/download model (*caching*), an obsolete copy of the file might be read



(a)



(b)

Semantics of File Sharing

- Four different kinds of semantics for updating shared files in a distributed system

Semantics	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes A **read** operation returns the effect of the last **write** operation Implementable only for remote access models that allow **only a single copy of the file**
Session semantics	No changes are visible to other processes until the file is **closed** The effects of **read** and **write** operations are seen only by the client that has **opened** (a local copy of) the file When the file is closed, **only one client's writes** remain
Immutable files	**No updates** are possible; simplifies sharing and replication
Transaction semantics	All changes to a file occur **atomically** The file system supports transactions on a *single* file Issue: How to allow concurrent access to a physically distributed file?

File Locking

- NFS supports an explicit (stateful) locking protocol for files
- NFS distinguishes **read locks** from **write locks**
- In NFS, locks are granted for a certain period of time, determined by the server, i.e., a lock has an associated **lease**
- NFSv4 operations related to file locking

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 lease on a specified lock

File Locking

- **Observation:** They could have kept it simple, but they didn't
- NFS also supports an implicit **share reservations** approach
  - When a client opens a file, it specifies the type of **access it requires** (READ, WRITE, BOTH) and also the type of **access denied other clients** by the server (NONE, READ, WRITE, BOTH)
  - What happens when a client tries to open a file

Current denial state specified by previous client

Request access by this client	**NONE**	**READ**	**WRITE**	**BOTH**
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(a)

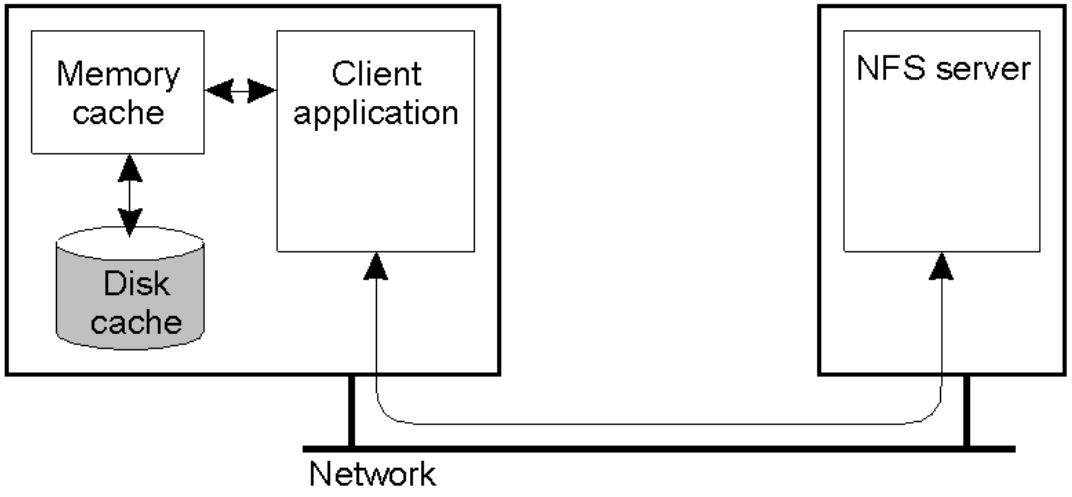
Denial state requested by this client

Current access of previous client	**NONE**	**READ**	**WRITE**	**BOTH**
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(b)

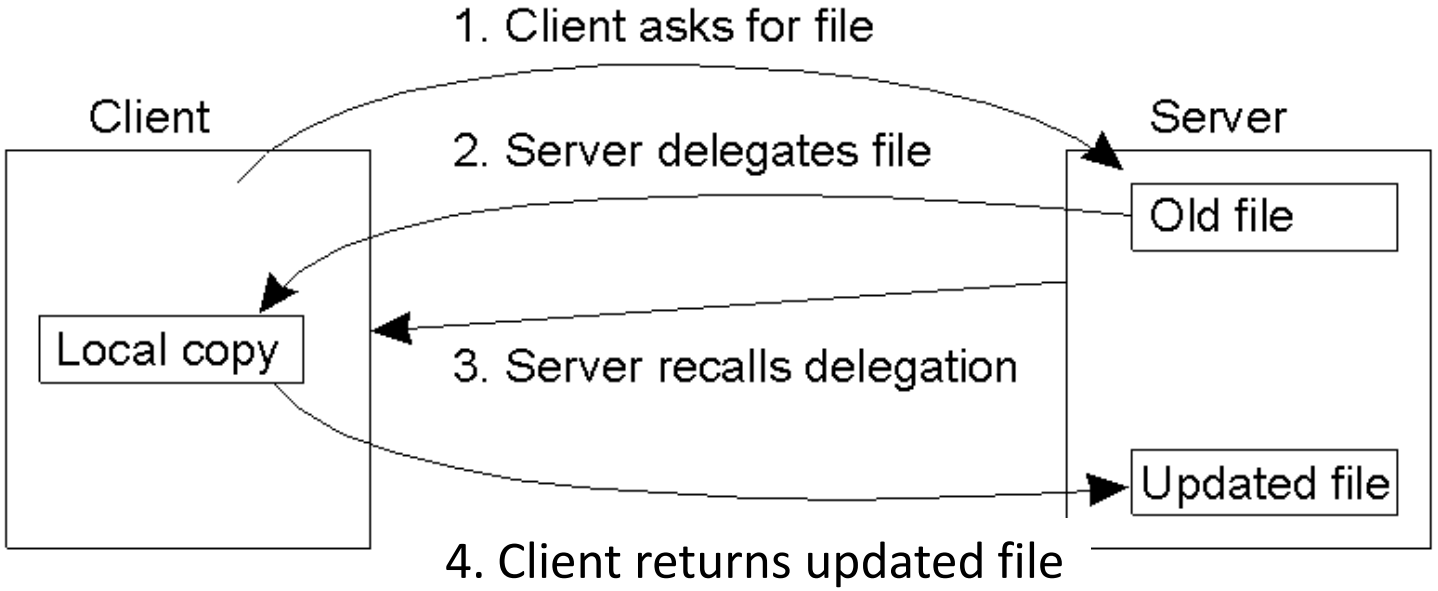
Client-Side Caching

- Client-side caching in NFS
  - **Essence: Clients are on their own**
- **Open file delegation:** Server explicitly permits a client machine to handle local operations for *other* clients on the client's machine
  - **Good for performance**
  - **Requires the server to take over when necessary**



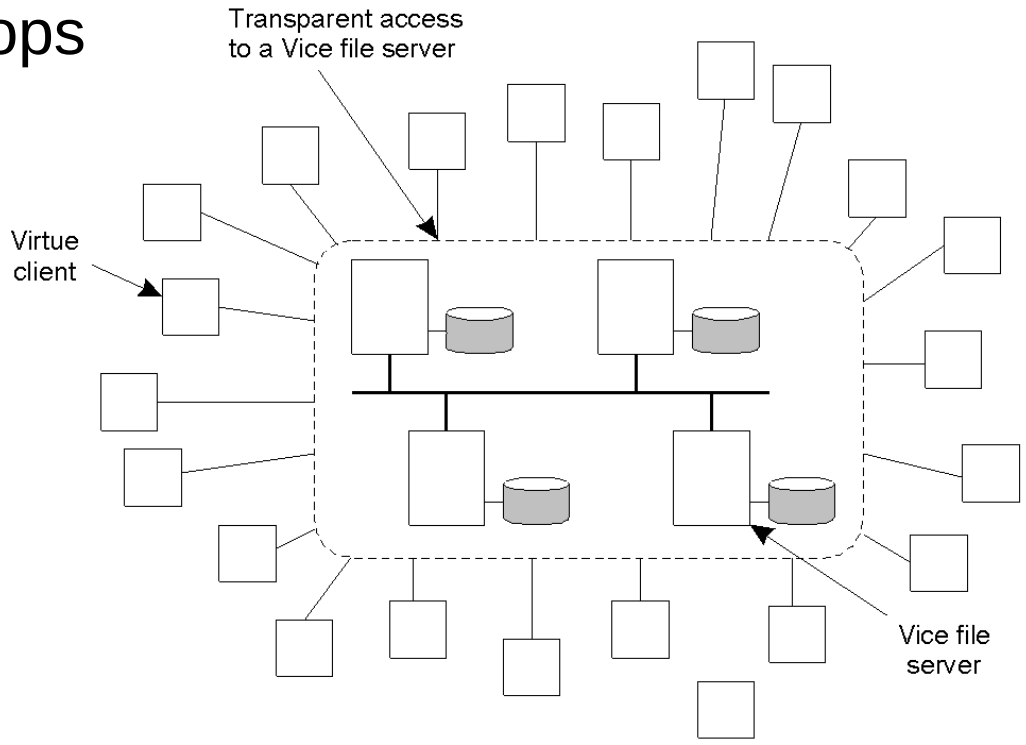
Client-Side Caching

- Server uses the NFSv4 callback mechanism to recall delegation of the file to the client; the client must then return the file to the server



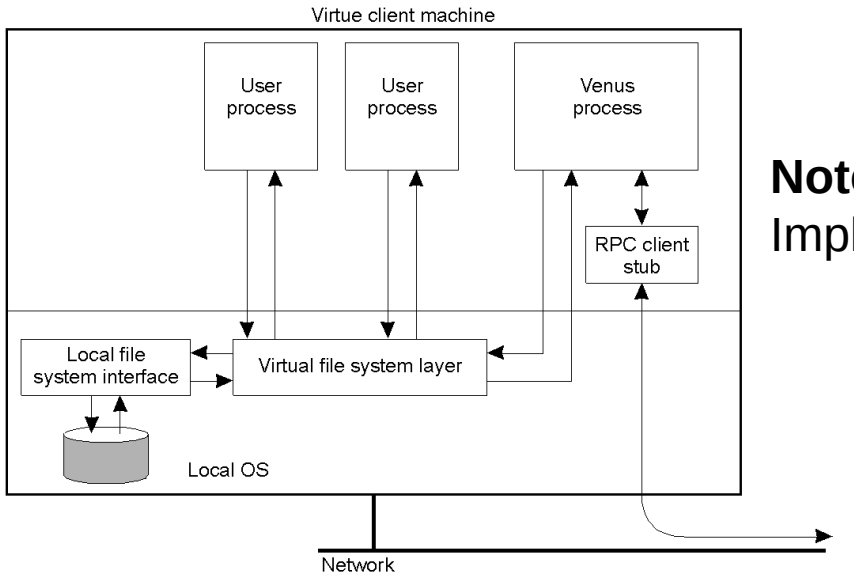
Coda File System

- Developed in the 1990s as a descendant of the Andrew File System (AFS) at CMU
- Now included in Linux distributions
- **Feature:** Support for disconnected operation, e.g., mobile laptops



Coda File System

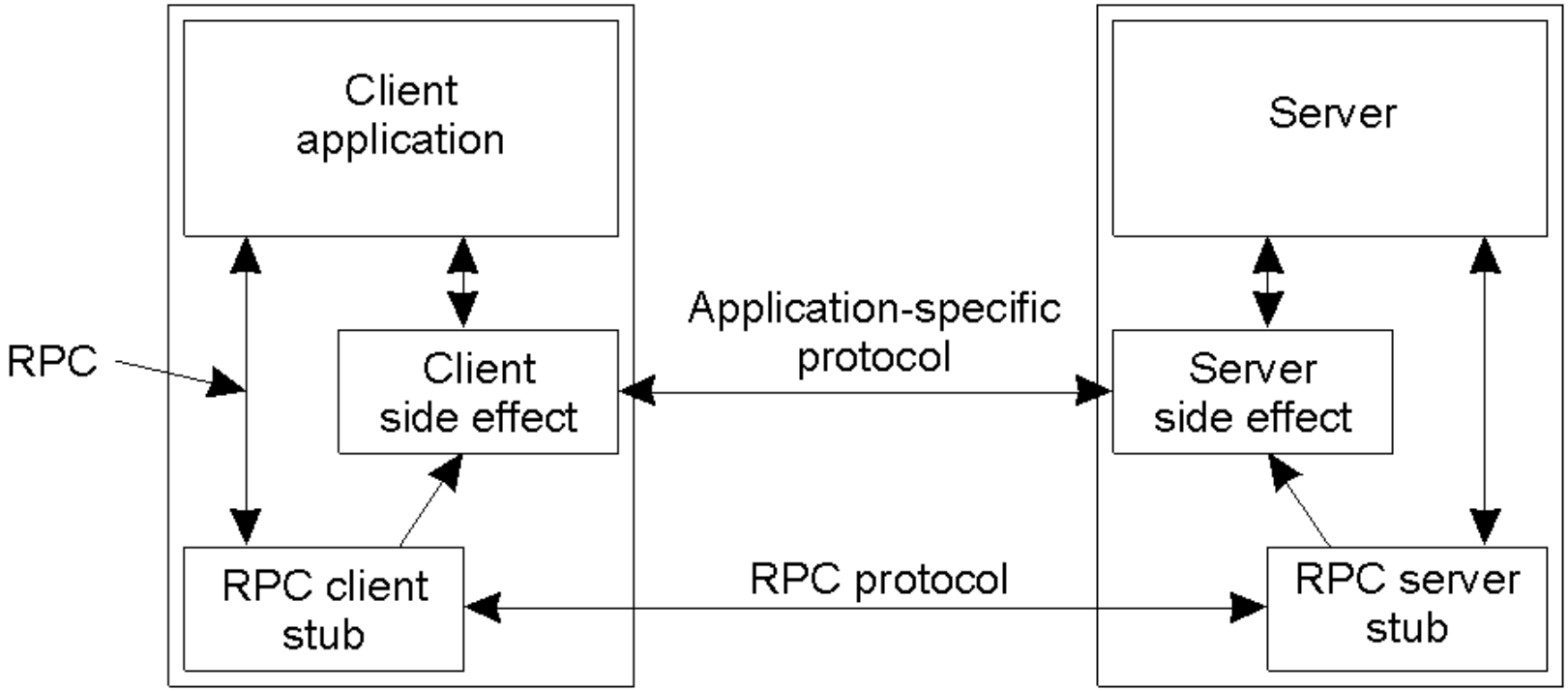
- **Virtue client** is like an NFS client, but the client can continue locally even if remote server is not accessible
- **VFS layer** intercepts client file calls, and directs them to either the local file system or to the Venus process
- **Venus process** use RPC to communicate with a remote Vice server
- **Vice file server** maintains a globally shared name space, unlike NFS



Note: Virtue client is mostly Implemented in user space

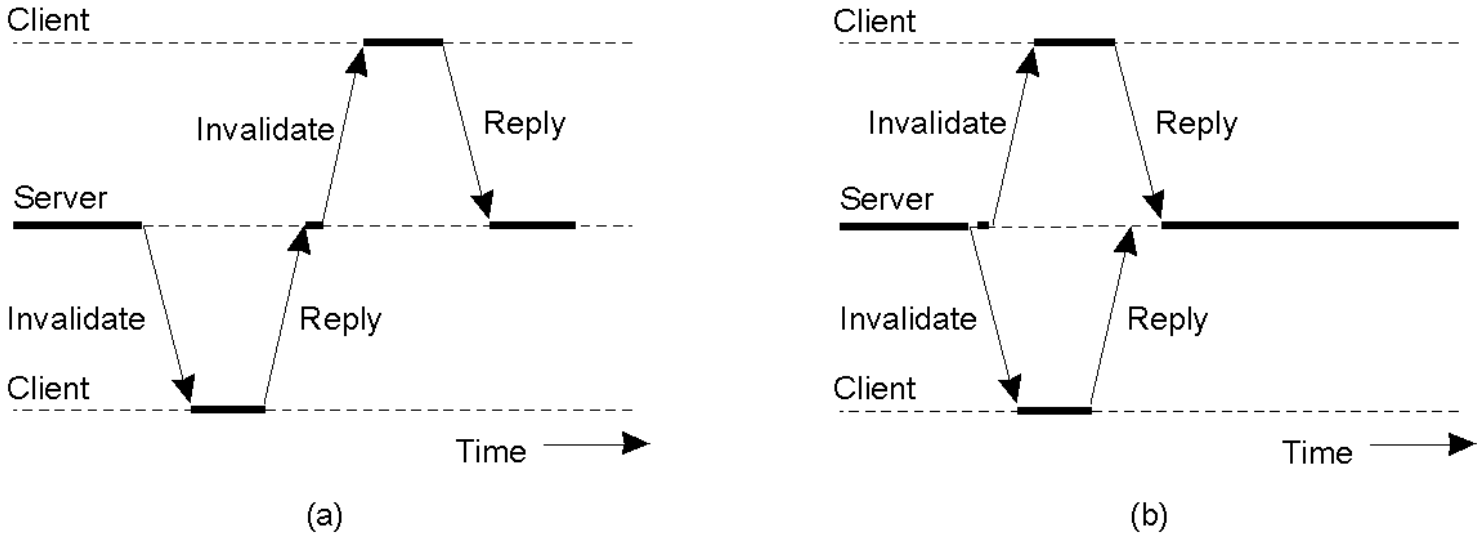
Communication

- In Coda, all client-server communication (and server-server communication) is handled by a reliable RPC subsystem
- Coda RPC supports **side-effects**, which allow specific protocols to handle multimedia streams, etc.



Communication

- In Coda, clients cache entire files; a client can modify a file locally but must send a **notification message** to the file server
- The file server then sends **invalidation messages** to the other clients => a need for **multicast RPC**

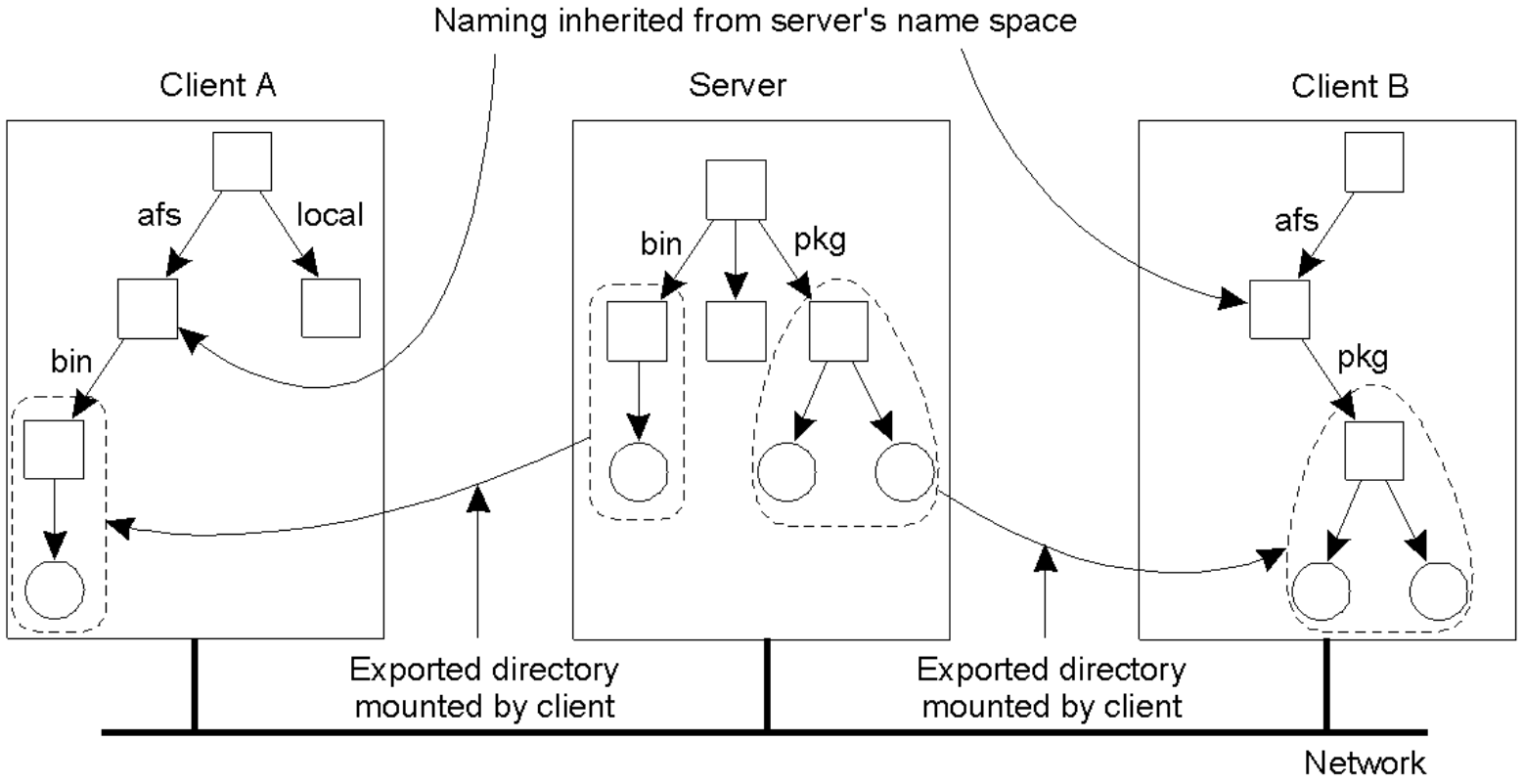


(a) Sending an invalidation message one at a time

(b) Sending invalidation messages in parallel

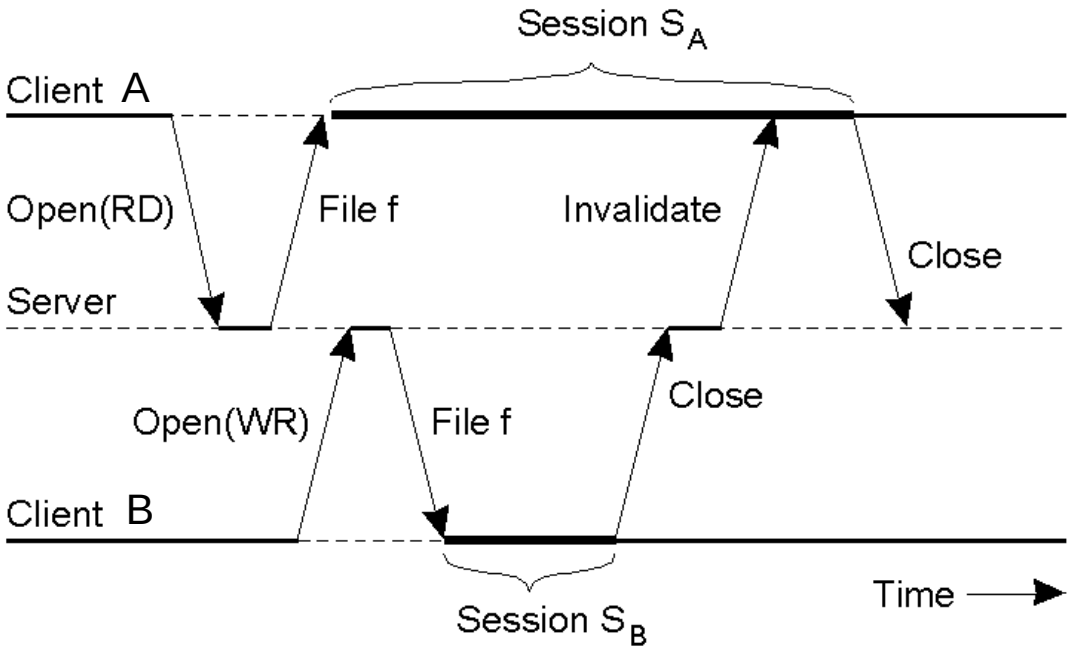
Naming and Mounting

- In Coda, clients have access to a single shared name space
- Mounting mechanisms are similar to those of NFS



Sharing Files

- Coda uses **transactional semantics** for sessions, without the ACID properties of real transactions
- Transactional properties take the form of "this ordering could have taken place"



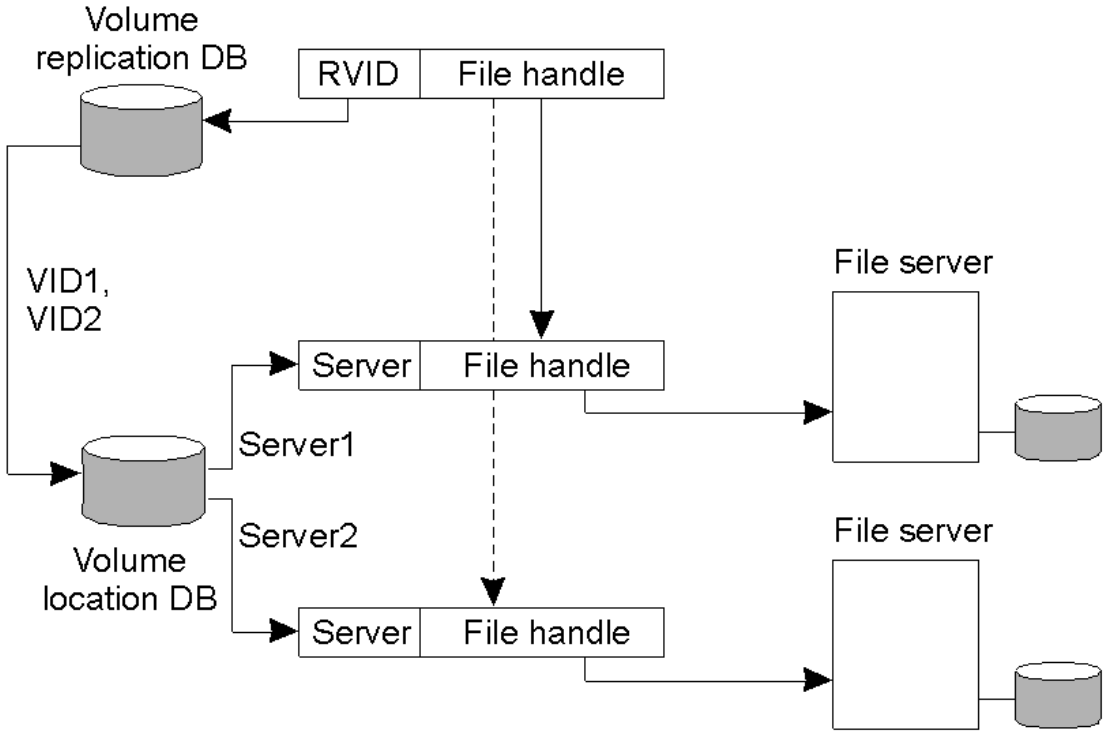
Transactional Semantics

- The metadata that are read and modified for a ***store*** session type in Coda

File-associated data	Read?	Modified?
File identifier	Yes	No
Access rights	Yes	No
Last modification time	Yes	Yes
File length	Yes	Yes
File contents	Yes	Yes

File Handles and Volume Identifiers

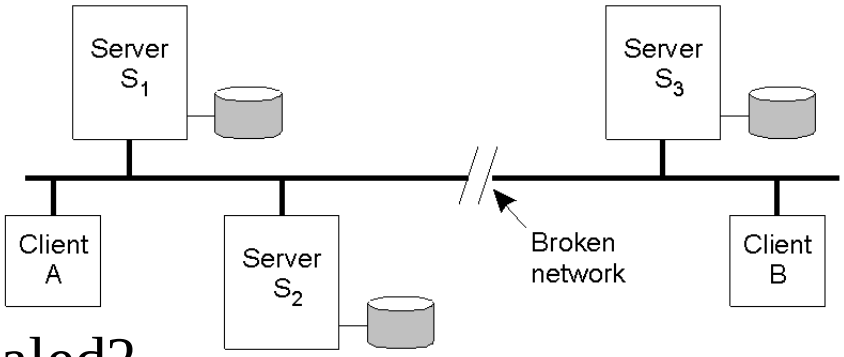
- **Issue:** How to track files in a location-transparent way in Coda
- A collection of files is a **volume** (like a UNIX disk partition but smaller); a volume is the unit of replication in Coda
  - **Physical volume has a Volume Identifier (VID)**
  - **Replicated Volume has a Replicated Volume Identifier (RVID)**



Server Replication

- Coda allows file servers to be replicated, using a **Read One Write All (ROWA)** replicated write protocol
- The collection of servers that have a copy of a volume form a **Volume Storage Group (VSG)**
  - **Reads are done from one server in a file's VSG**
  - **Writes are propagated to all servers in the file's VSG**
- In the presence of failures, a client might not have access to all servers in a VSG
- Those servers in a VSG that a client can contact at the moment form the client's **Accessible Volume Storage Group (AVSG)**
  - **When a client needs to read a file, it contacts one of the servers in its AVSG**
  - **When a client updates a file and closes a session, it multicasts the updated file to all servers in its AVSG**

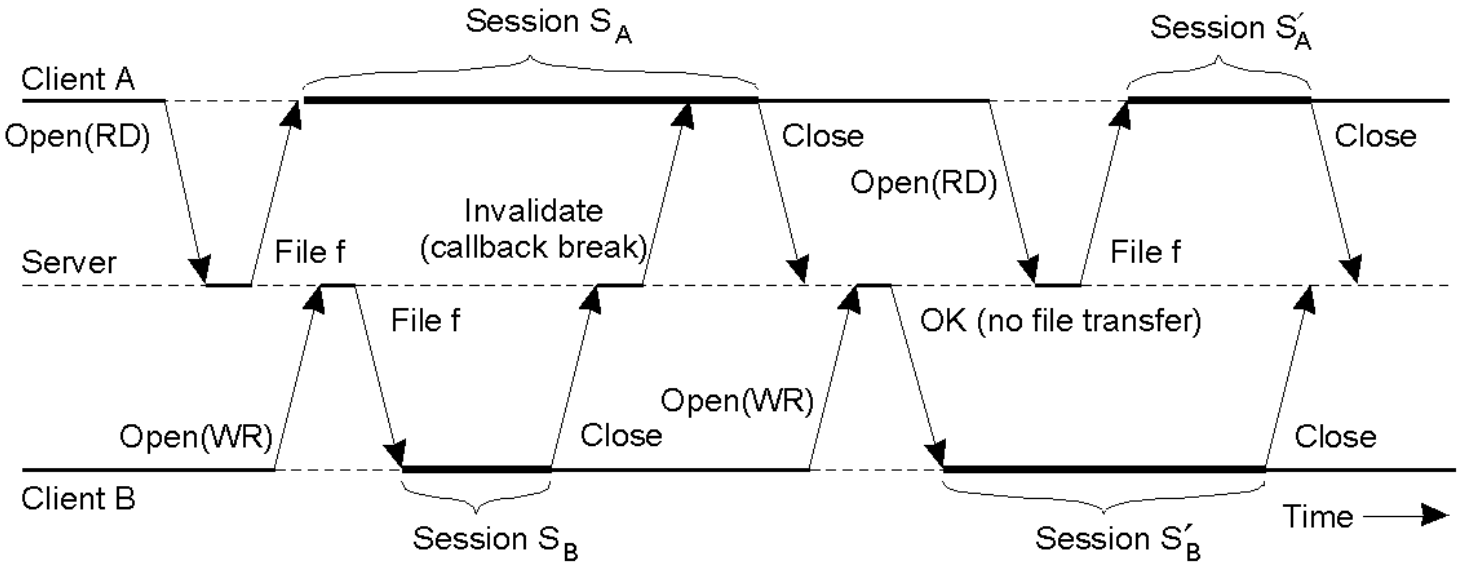
Server Replication



- **Problem:** What to do when a VSG partitions and the partition is later healed?
- **Solution:** Detect inconsistencies using **Coda Version Vectors** (like vector timestamps)
- ***CVVi(f)[j] = k* means that server *Si* knows that server *Sj* has seen (at least) version *k* of file *f***
- When a client reads file *f* from server *Si*, it receives the version vector *CVVi*(*f*) from server *Si*
- When server *Si* receives an update for file *f*, it increments *CVVi(f)[i]*
- When a partition is healed, comparison of version vectors allows detection of conflicts and possible reconciliation (see the example in the text on page 525)

Client-Side Caching

- Clients cache entire files and operate on their local copies (using the upload/download model)
- The file server keeps track of which clients have cached a file locally, and records a **callback promise** when a client does so
- When a client updates a file the first time, it sends a **notification message** to the server
- The server sends an **invalidation message**, containing a **callback break**, to the other clients and discards the callback promise it held for them



NFS vs. Coda Summary

Issue	NFS	Coda
Design goals	Access transparency	High availability
Access model	Remote	Up/Download
Communication	RPC	RPC
Client process	Thin/Fat	Fat
Server groups	No	Yes
Mount granularity	Directory	File system
Name space	Per client	Global
File ID scope	File server	Global
Sharing sem.	Session	Transactional
Cache consist.	write-back	write-back
Replication	Minimal	ROWA
Fault tolerance	Reliable comm.	Replication and caching
Recovery	Client-based	Reintegration
Secure channels	Existing mechanisms	Needham-Schroeder
Access control	Many operations	Directory operations