Distributed File Systems

Jonathan Walpole

CSE515 Distributed Computing Systems

Design Issues

- Naming and name resolution
- Architecture and interfaces
- Caching strategies and cache consistency
- File sharing semantics
- Disconnected operation and fault tolerance

Naming Files

Transparency Issues

- Can clients distinguish between local and remote files?
- Location transparency
  - file name does not reveal the file's physical storage location.
- Location independence
  - the file name does not need to be changed when the file's physical storage location changes.

Global vs Local Name Spaces

- Global name space
  - file names are globally unique
  - any file can be named from any node

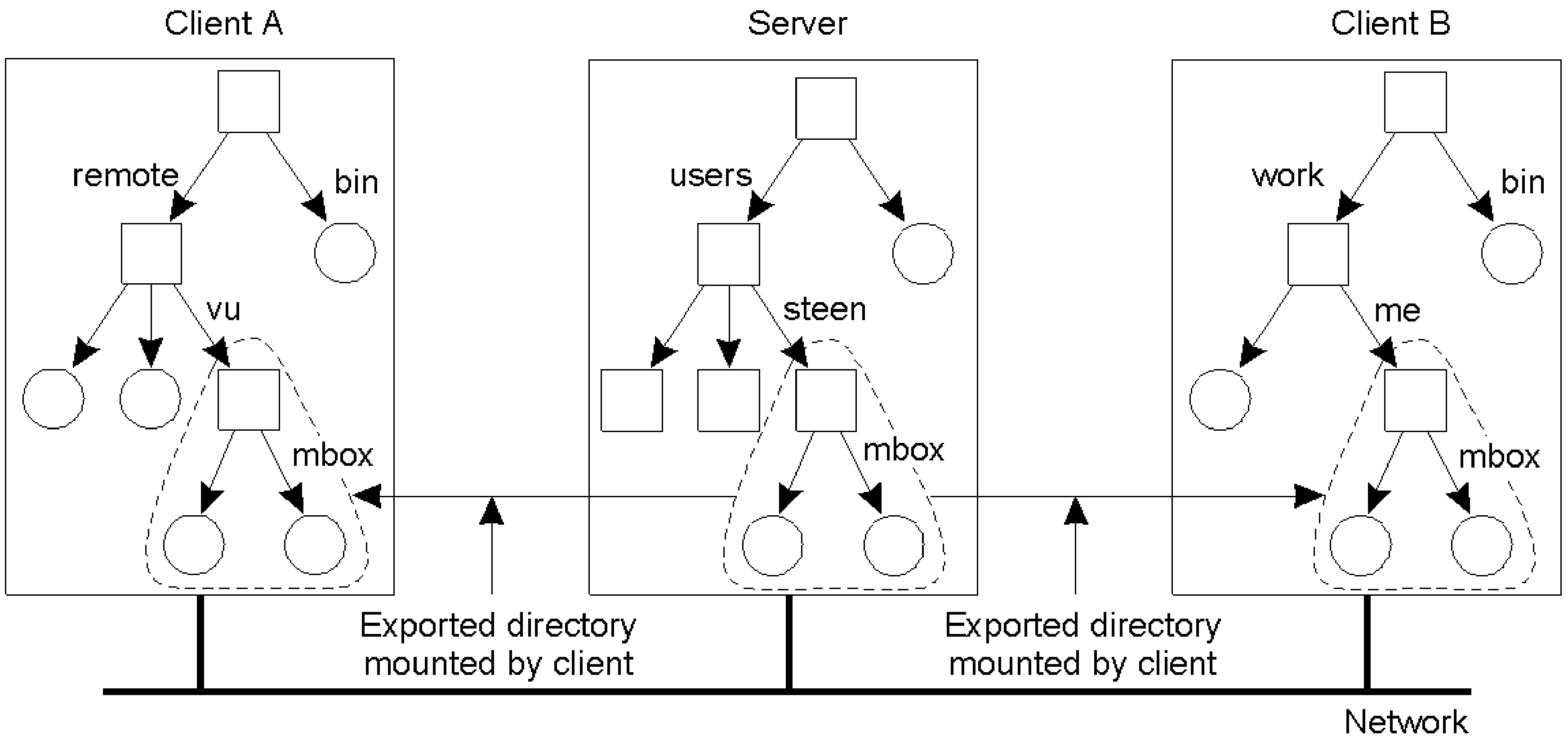
- Local name spaces
  - remote files must be inserted in the local name space
  - file names are only meaningful within the calling node
  - how do you refer to remote files in order to insert them?
  - globally unique file handles can be used to map remote files to local names

How to Build a Name Space?

- Super-root / machine name approach
  - concatenate the host name to the names of files stored on that host
  - system-wide uniqueness guaranteed
  - simple to located a file
  - not location transparent or location independent

- Mounting remote file systems
  - *exported* remote directory is *imported* and mounted onto local directory
  - accesses require a globally unique file handle for the remote directory
  - once mounted, file names are location-transparent
    - location can be captured via naming conventions
  - are they location independent?
    - location of file vs location of client?
    - files have different names from different places

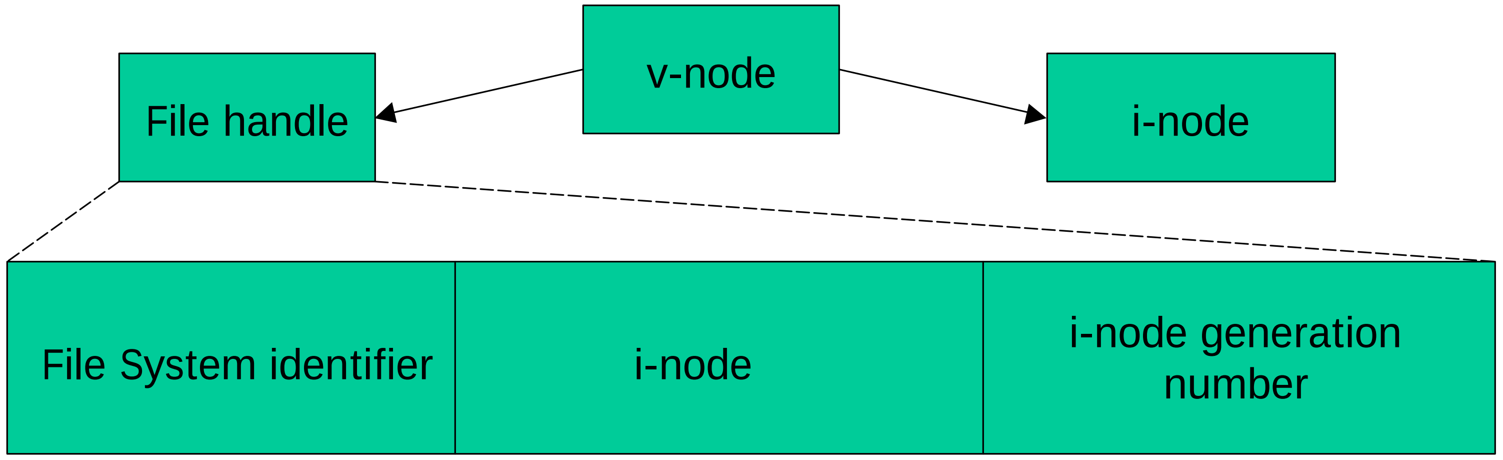
Local Name Spaces with Mounting


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

NSF Name Space

- Server `exports` a directory
- *mountd*: provides a unique file handle for the exported directory
- Client uses RPC to issue `nfs_mount` request to server
- *mountd* receives the request and checks whether
  - the pathname is a directory?
  - the directory is exported to this client?

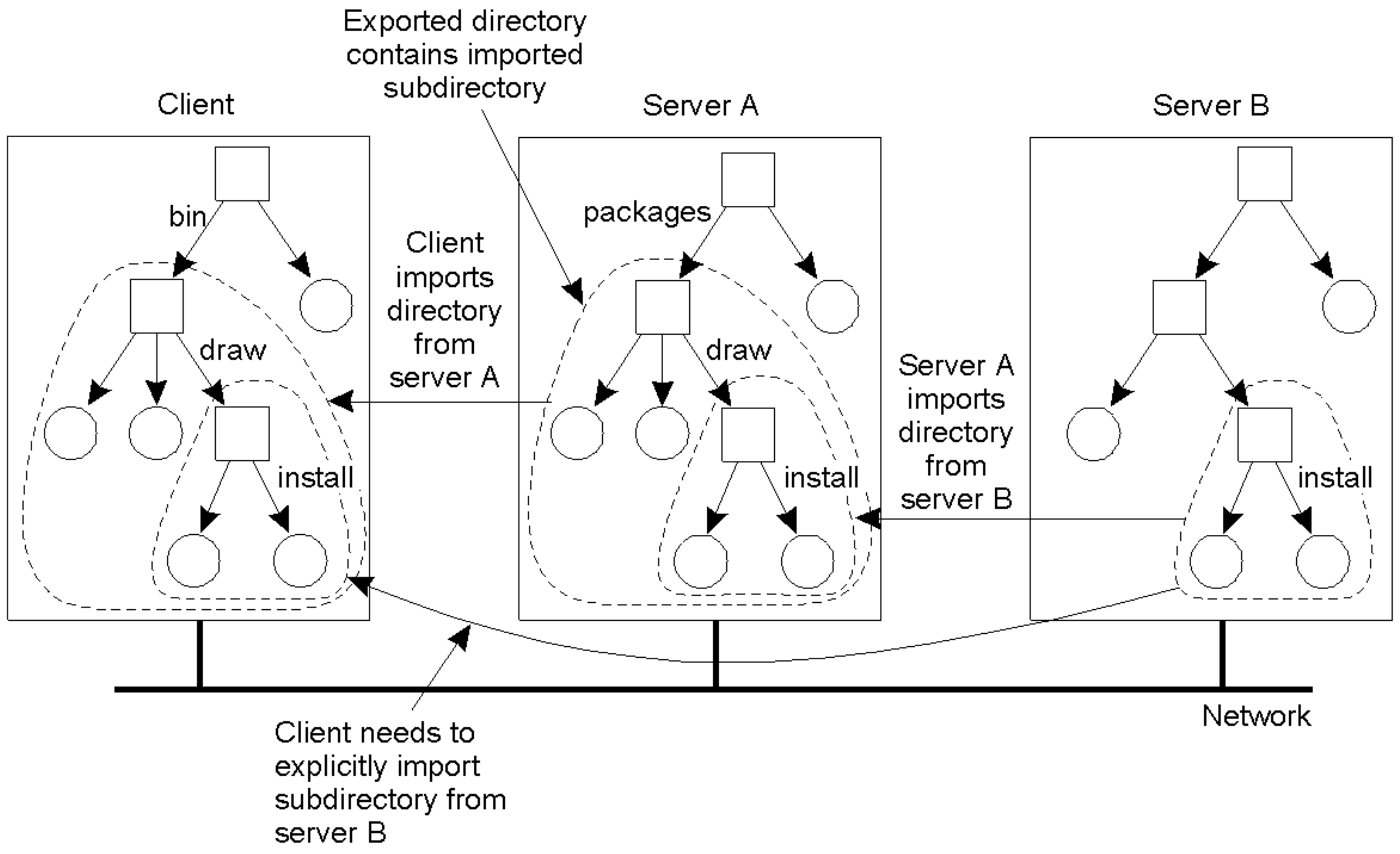
NFS File Handles



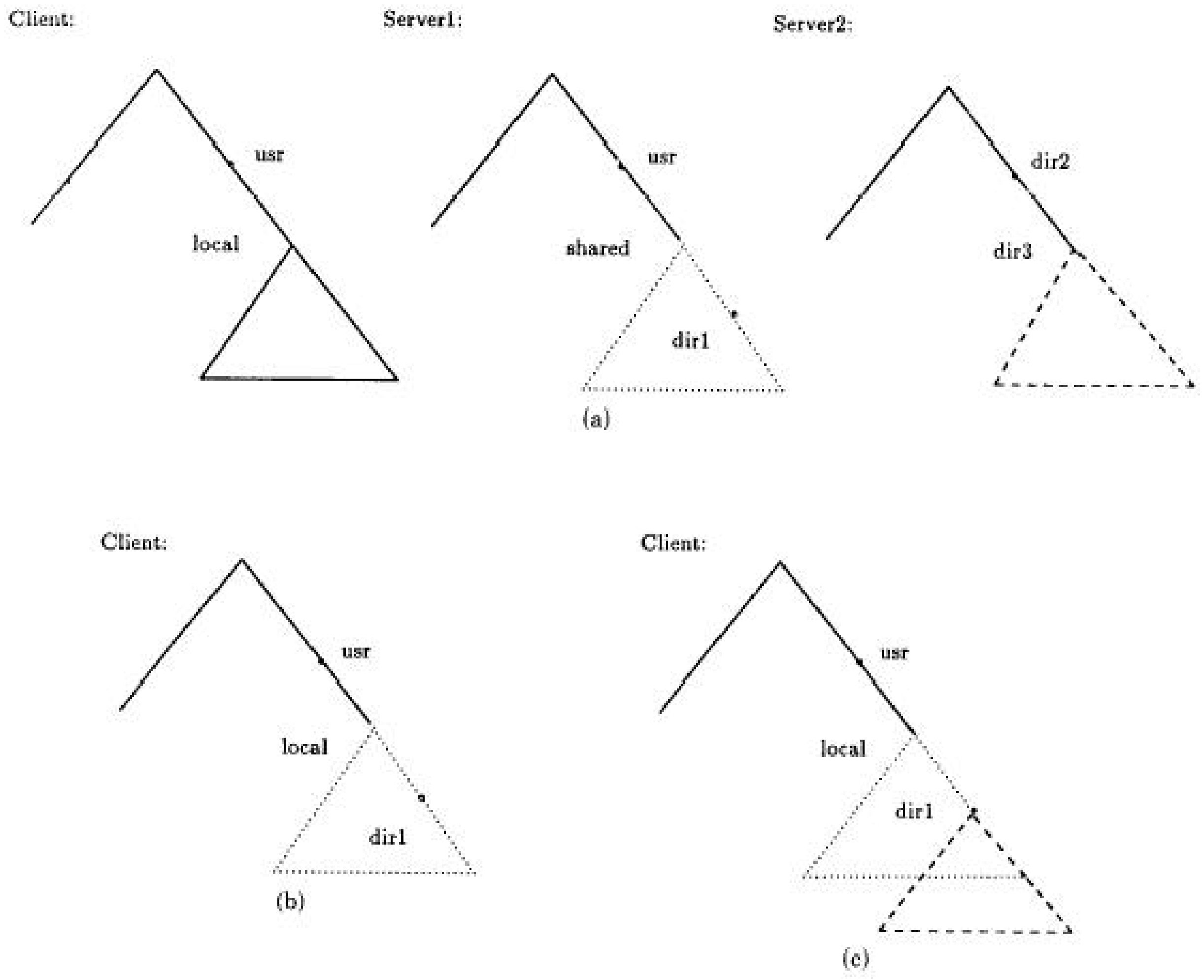
- V-node contains
  - reference to a file handle for mounted remote files
  - reference to an i-node for local files

- File handle uniquely names a remote directory
  - file system identifier: unique number for each file system (in UNIX super block)
  - i-node and i-node generation number

Nested Mounting on Multiple Servers


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

NFS Pathname Translation Across Mount Points



NFS joins independent file systems (a), by mounts (b), and cascading mounts (c).

NFS Pathname Translation Across Mount Points

- Is done *iteratively* by client
- Example: /usr/local/dir1/myfile
  - *Lookup(/ I-node, usr) → /usr I-node*
  - *Lookup(/usr I-node, local) → /usr/local file handle*
    - *Server 1 is contacted*
  - *Lookup(/usr/local file handle, dir1) → /usr/local/dir1 file handle*
    - *Server 2 is contacted*
  - *Lookup(/usr/local/dir1 file handle, myfile) → /usr/local/dir1/myfile file handle*
    - *Server 2 is contacted*
- Server 1 cannot lookup dir1 for client because dir1 is something else on server 1 than on client
- Lookups are cached

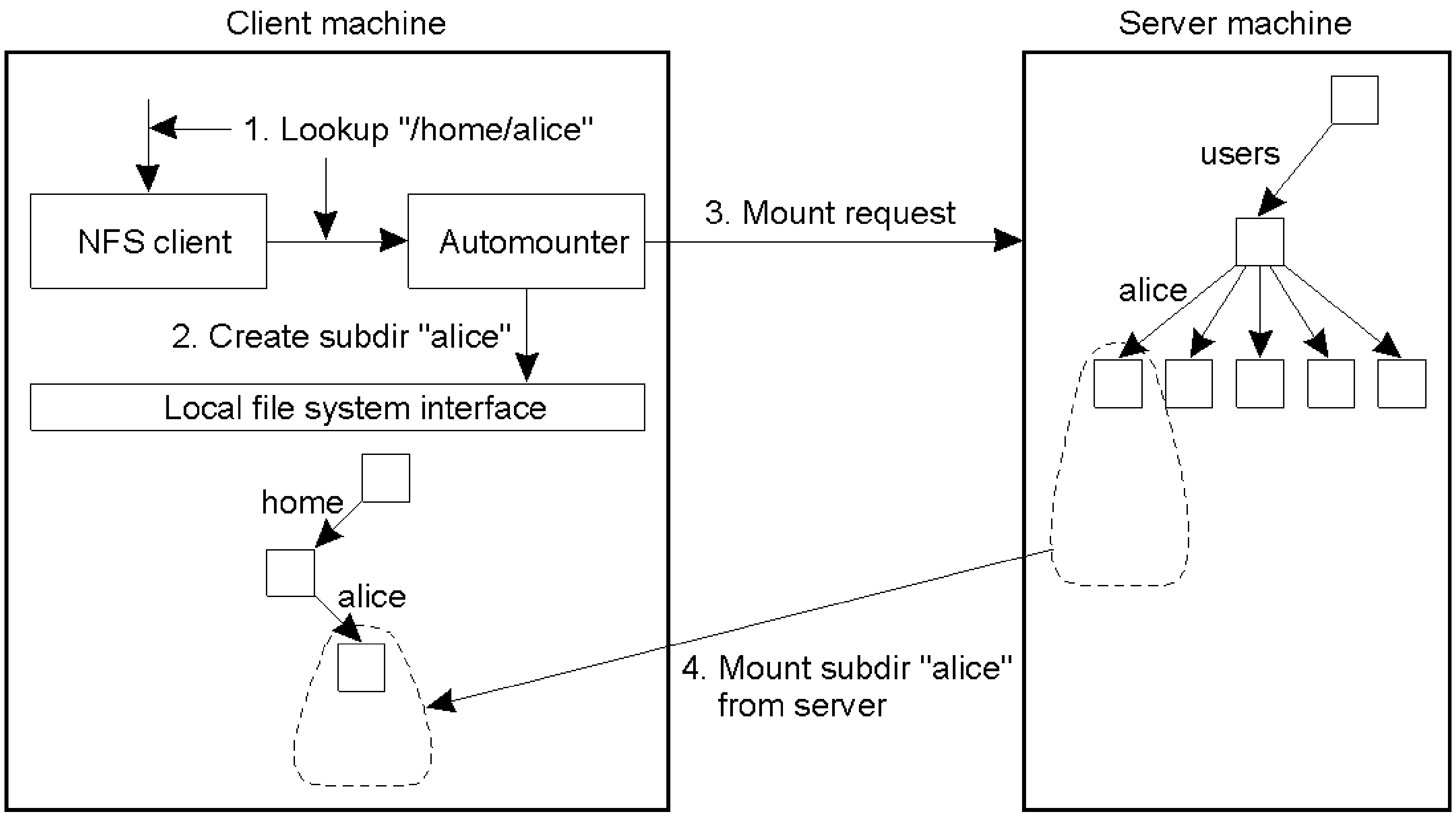
Mounting On-Demand

- Need to decide *where* and *when* to mount remote directories
- Where? - Can be based on conventions to standardize local name spaces (ie., /home/username for user home directories)
- When? - boot time, login time, access time, …?
- What to mount when?
  - – How long does it take to mount everything?
  - – Do we know what everything is?
  - – Can we do mounting on-demand?

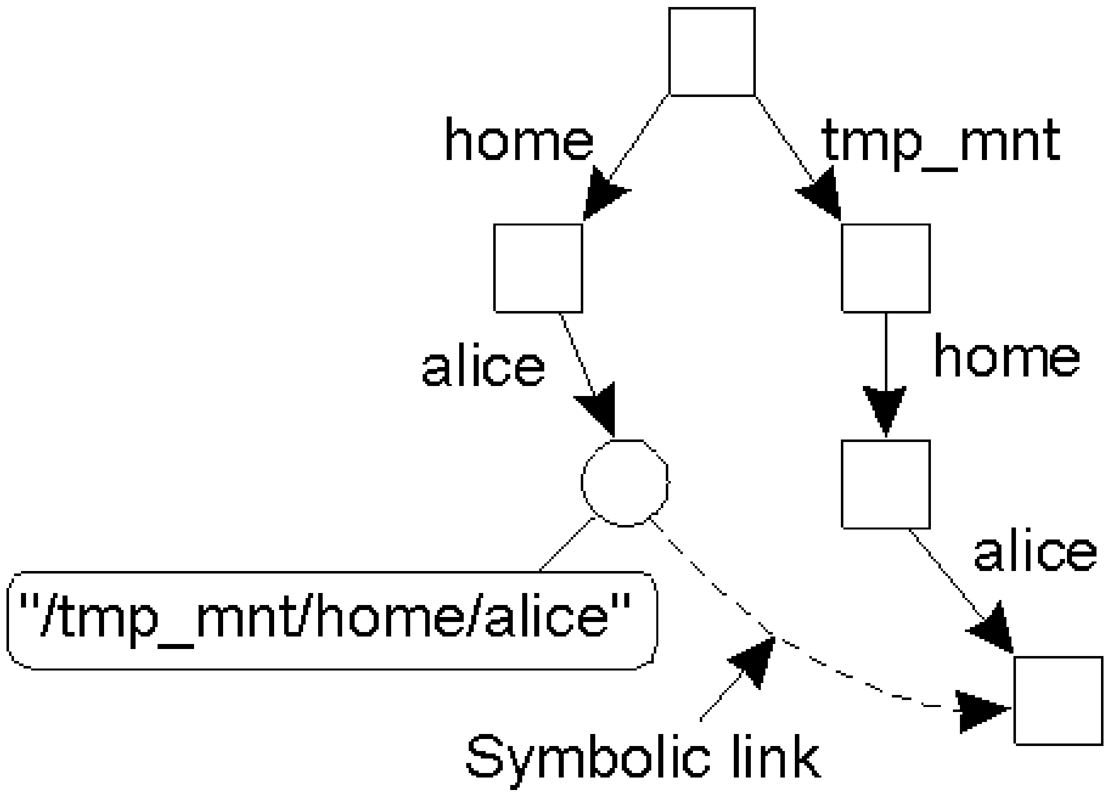
Automounting

- An *automounter* is a client-side process that handles on-demand mounting
  - it acts like a local NFS server
- Intercepts accesses to unmounted remote directories (/home)
  - accesses to /home go to local automounter
  - /home/walpole causes automounter to create local directory /home/walpole and mount remote directory /users/walpole there
- Overhead of automounter indirection?
  - Takes itself out of the loop by mounting remote /users/walpole under /tmp_mnt/home/walpole and making /home/walpole a symbolic link to it

Automounting in NSF



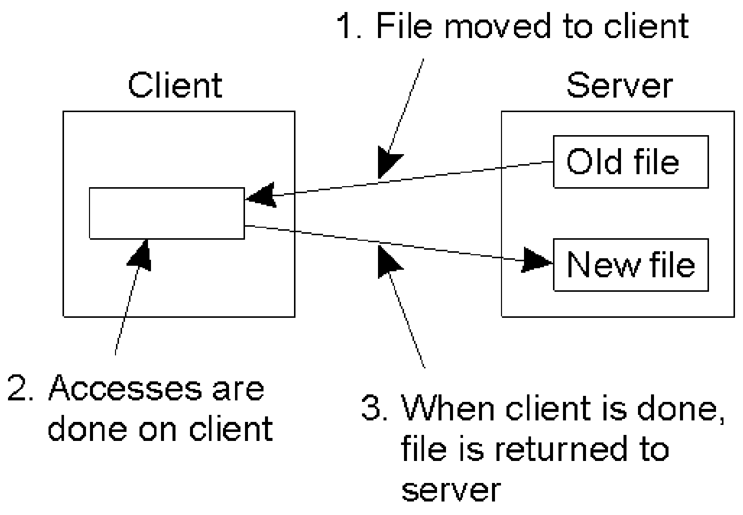
Using Symbolic Links with Automounting



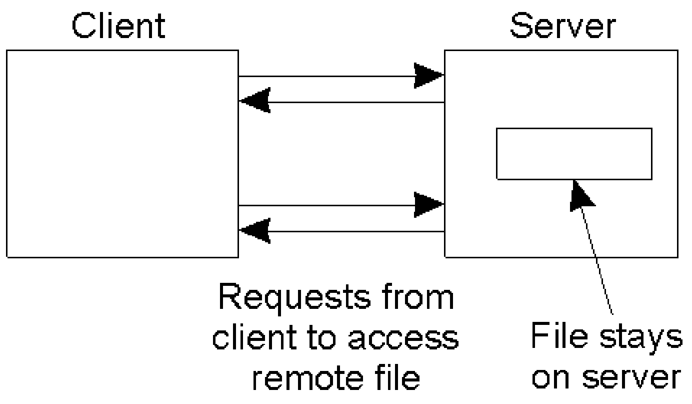
Architecture and Interfaces

Local Access Architectures

- Local access approach
  - move file to client
  - local access on client
  - return file to server
  - data shipping approach


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 Architectures



- Remote access
  - leave file on server
  - send read/write operations to server
  - return results to client
  - function shipping approach

File vs Block-Level Interface

- File-level client-server interface
  - local access model with whole file movement and caching
  - remote access model client-server interface at system call level
  - client performs remote open, read, write, close calls

- Block-level client-server interface
  - client-server interface at file system or disk block level
  - server offers virtual disk interface
  - client file accesses generate block access requests to server
  - block-level caching of parts of files on client

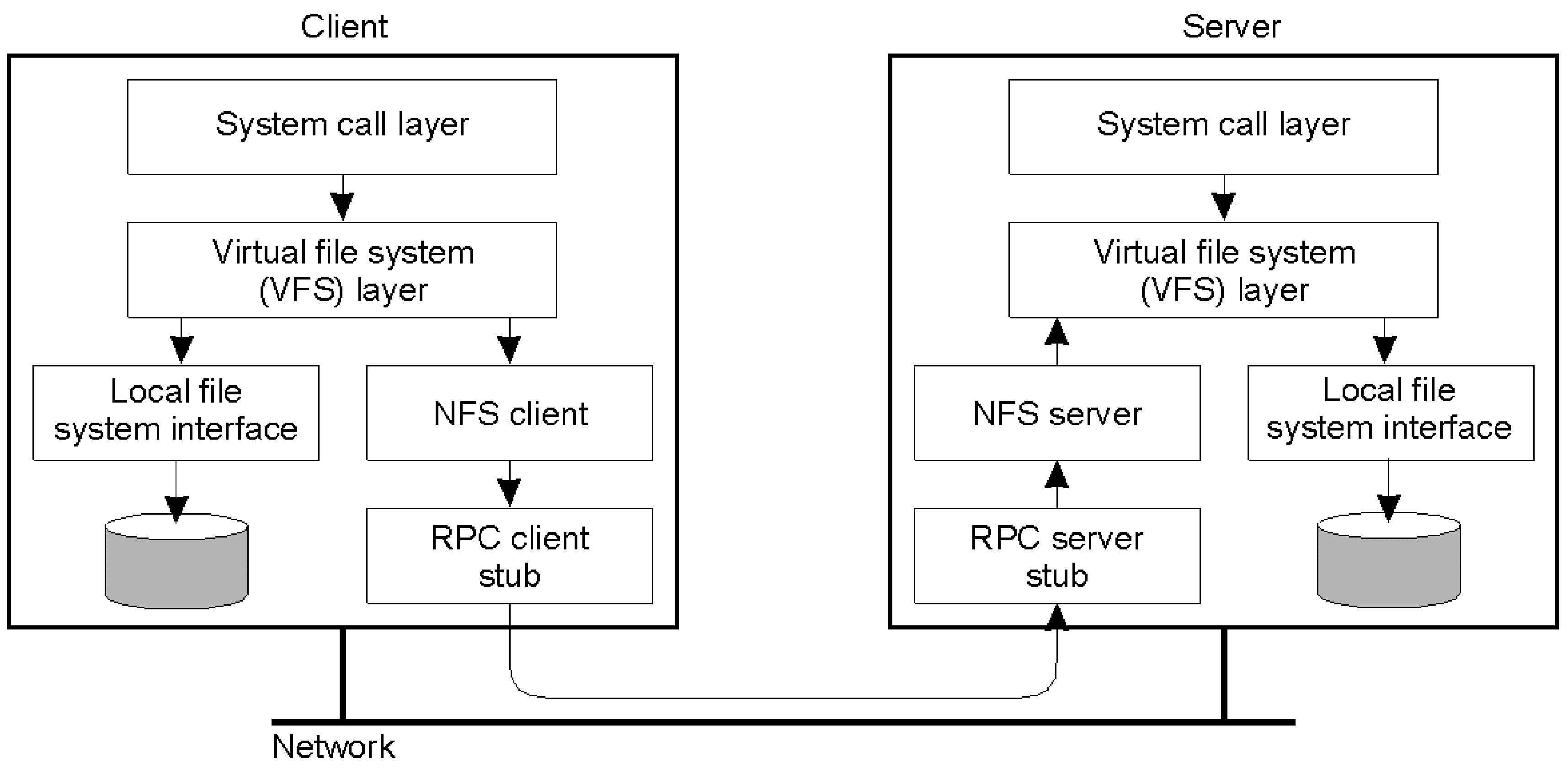
Hybrid Approaches

- Remote file operations
- Client-side caching of blocks

Distributed File System Architectures

- Server side
  - how do servers export files
  - how do servers handle requests from clients?
- Client side
  - how do applications access a remote file in the same way as a local file?
- Communication layer
  - how do clients and servers communicate?

NFS Architecture



Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

NFS Server Side

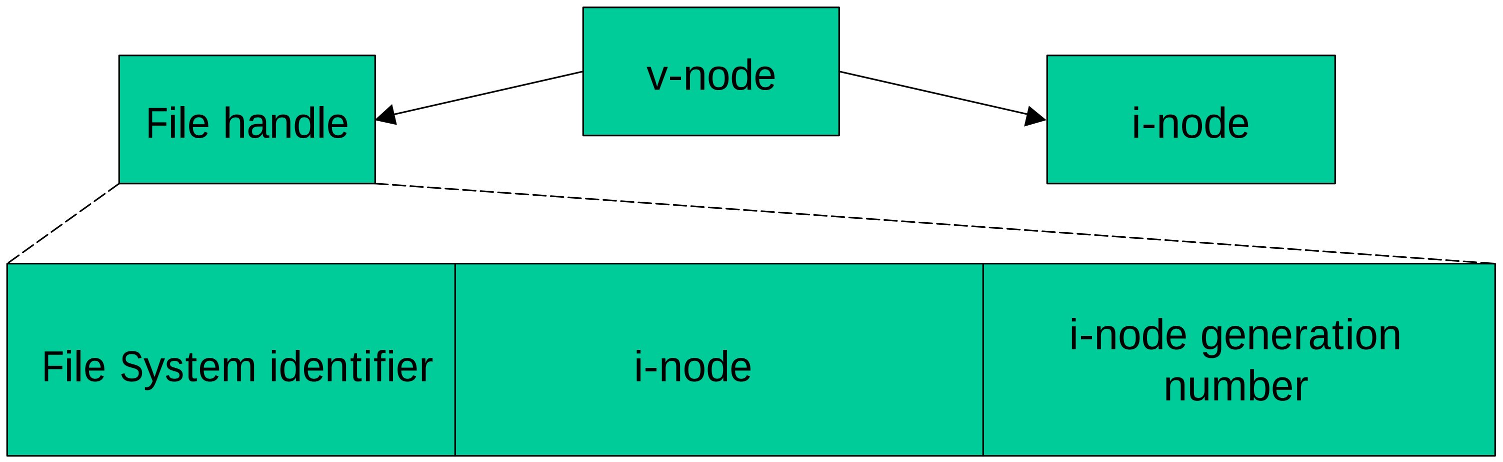
- *Mountd*
  - server exports directory via *mountd*
  - *mountd* provides the initial file handle for the exported directory
  - client issues `nfs_mount` request via RPC to *mountd*
  - *mountd* checks if the pathname is a directory and if the directory is exported to the client
- *nfsd*: services NFS RPC calls, gets the data from its local file system, and replies to the RPC
  - Usually listening at port 2049
- Both *mountd* and *nfsd* use RPC

Communication Layer: NFS RPC Calls

Proc.	**Input args**	**Results**
lookup	dirfh, name	status, fhandle, fattr
read	fhandle, offset, count	status, fattr, data
create	dirfh, name, fattr	status, fhandle, fattr
write	fhandle, offset, count, data	status, fattr

- NFS / RPC uses XDR and TCP/IP
- fhandle: 64-byte opaque data (in NFS v3)
  - what’s in the file handle?

NFS File Handles

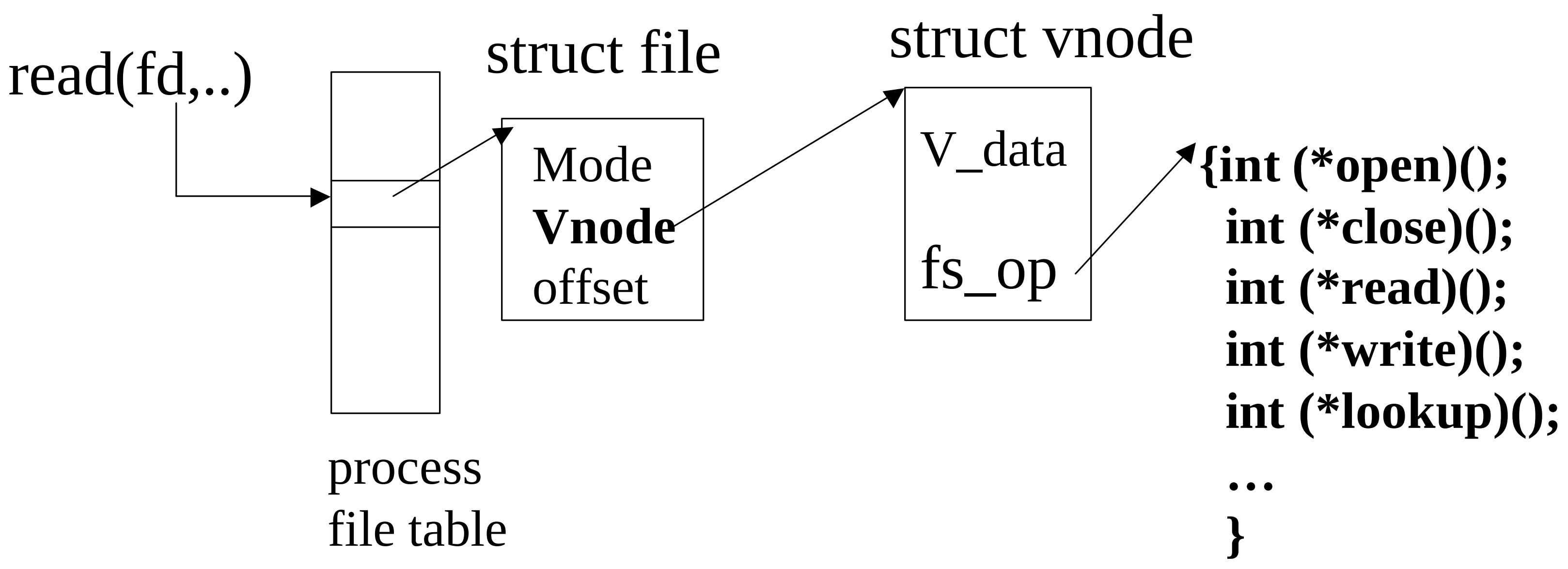


- V-node contains
  - reference to a file handle for mounted remote files
  - reference to an i-node for local files

- File handle uniquely names a remote directory
  - file system identifier: unique number for each file system (in UNIX super block)
  - i-node and i-node generation number

NFS Client Side

- Accessing remote files in the same way as accessing local files requires kernel support
  - Vnode interface



```
{int (*open)();
  int (*close)();
  int (*read)();
  int (*write)();
  int (*lookup)();
  …
  }
```

Caching

Caching vs Pure Remote Service

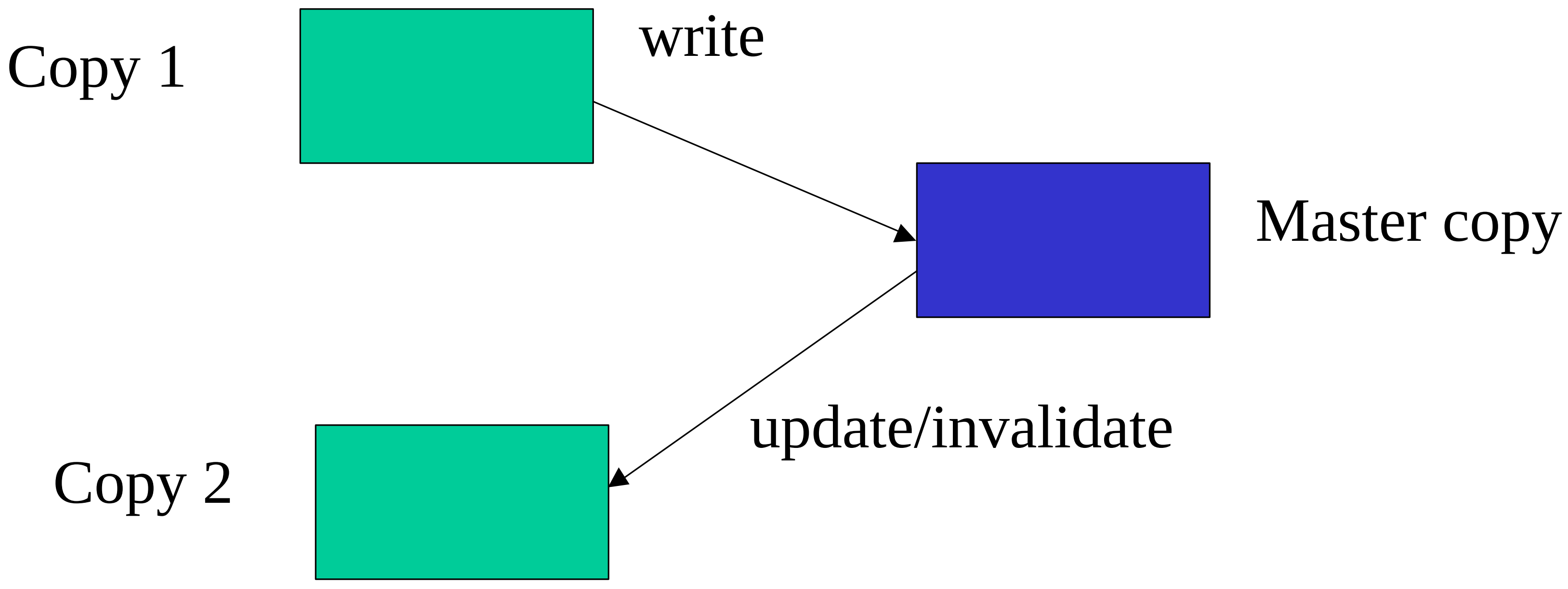
- Network traffic?
  - caching reduces remote accesses ⇒ reduces network traffic
  - caching generates fewer, larger, data transfers
- Server load?
  - caching reduces remote accesses ⇒ reduces server load
- Server disk throughput?
  - optimized better for large requests than random disk blocks
- Data integrity?
  - cache-consistency problem due to frequent writes
- Operating system complexity?
  - simpler for remote service.

Caching

- Four places to store files
  - Server's disk: slow performance
  - Server's memory
    - cache management, how much to cache, replacement strategy
    - still slow due to network delay
  - Client's disk
    - access speed vs server memory?
    - large files can be cached
    - supports disconnected operation
  - Client's memory
    - fastest access
    - can be used by diskless workstations
    - competes with the VM system for physical memory space

Cache Consistency

– Reflecting changes to local cache to master copy
– Reflecting changes to master copy to local caches



Common Update Algorithms for Client Caching

- Write-through: all writes are carried out immediately
  - Reliable: little information is lost in the event of a client crash
  - Slow: cache not useful for writes
- Delayed-write: writes do not immediately propagate to server
  - batching writes amortizes overhead
  - wait for blocks to fill
  - if data is written and then deleted immediately, data need not be written at all (20-30 % of new data is deleted with 30 secs)
- Write-on-close: delay writing until the file is closed at the client
  - semantically meaningful delayed-write policy
  - if file is open for short duration, works fine
  - if file is open for long, susceptible to losing data in the event of client crash

Cache Coherence

- How to keep locally cached data up to date / consistent?
- Client-initiated approach
  - check validity on every access: too much overhead
  - first access to a file (e.g., file open)
  - every fixed time interval
- Server-initiated approach
  - server records, for each client, the (parts of) files it caches
  - server responds to updates by propagation or invalidation
- Disallow caching during concurrent-write or read/write sharing
  - allow multiple clients to cache file for read only access
  - flush all client caches when the file is opened for writing

NFS – Server Caching

- Reads
  - use the local file system cache
  - prefetching in UNIX using read-ahead
- Writes
  - write-through (synchronously, no cache)
  - commit on close (standard behaviour in v4)

NFS – Client Caching (reads)

- Clients are responsible for validating cache entries (stateless server)
- Validation by checking last modification time
  - time stamps issues by server
  - automatic validation on open (with server??)
- A cache entry is considered valid if one of the following are true:
  - cache entry is less than *t* seconds old (3-30 s for files, 30-60 s for directories)
  - modified time at server is the same as modified time on client

NFS – Client Caching (writes)

- Delayed writes
  - modified files are marked dirty and flushed to server on close (or sync)

- Bio-daemons (block input-output)
  - read-ahead requests are done asynchronously
  - write requests are submitted when a block is filled

File Sharing Semantics

Consistency Semantics for File Sharing

- What value do reads see after writes?
- UNIX semantics
  - value read is the value stored by last write
  - writes to an open file are visible immediately to others with the file open
  - easy to implement with one server and no cache
- Session semantics
  - writes to an open file are not visible immediately to others with the file opened already
  - changes become visible on close to sessions started later
- Immutable-Shared-Files semantics
  - A sharable file cannot be modified
  - File names cannot be reused and its contents may not be altered
  - Simple to implement.
- Transactions
  - All changes have all-or-nothing property
  - W1,R1,R2,W2 not allowed where P1 = W1;W2 and P2 = R1;R2

NFS – File Sharing Semantics

- Not UNIX semantics!
- Unspecified in NFS standard
- Not clear because of timing dependencies
- Consistency issues can arise
  - Example: Jack and Jill have a file cached. Jack opens the file and modifies it, then he closes the file. Jill then opens the file (before t seconds have elapsed) and modifies it as well. Then she closes the file. Are both Jack's and Jill's modifications present in the file? What if Jack closes the file after Jill opens it?
- Locking part of v4 (byte range, leasing)

File Locking in NFS

•**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

- NFS version 4 operations related to file locking.

File Locking in NFS

Current file denial state

Request access

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

Requested file denial state

Current access state

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

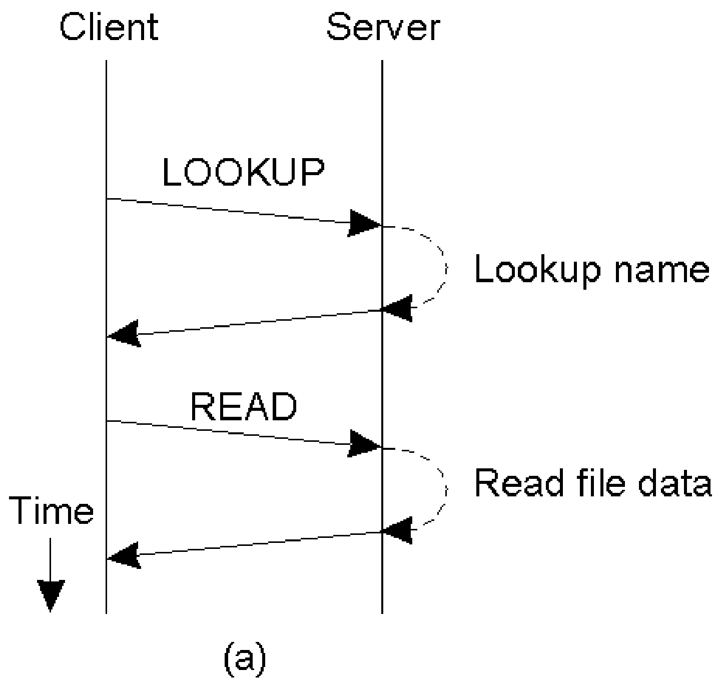
- The result of an *open* operation with share reservations in NFS.
- When the client requests shared access given the current denial state.
- When the client requests a denial state given the current file access state.

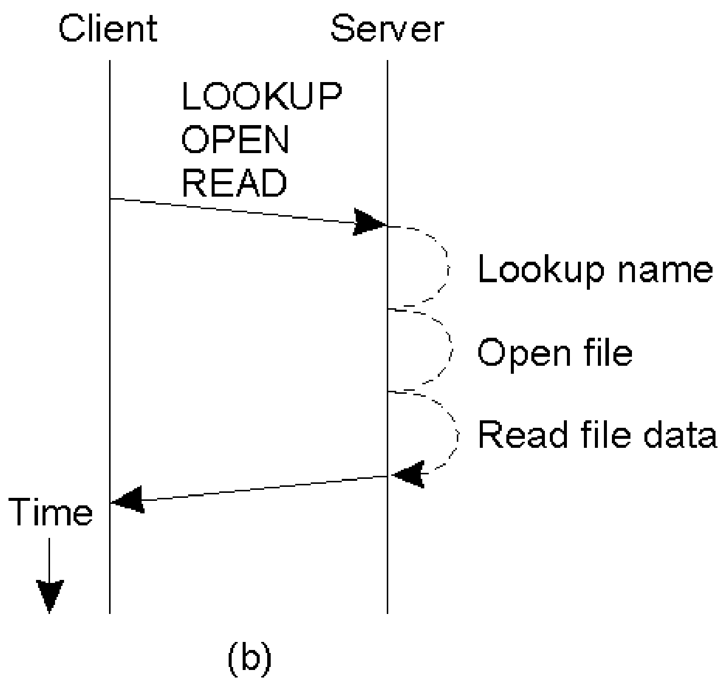
Fault Tolerance

Stateful versus Stateless Service

- Two approaches to server-side information
  1. stateful file server
     - a client performs open on a file
     - the server keeps file information (e.g., file descriptor entry, offset)
     - Adv: increased performance
     - On server crash, it looses all its volatile state information
     - On client crash, the server needs to know to claim state space
  2. stateless file server -- each request is self-contained
     - each request identifies the file, the position, read/write.
     - server failure is identical to slow server (client retries...)
     - each request must be idempotent.
     - NFS employs this.

NFS Stateless V3 & Stateful V4





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

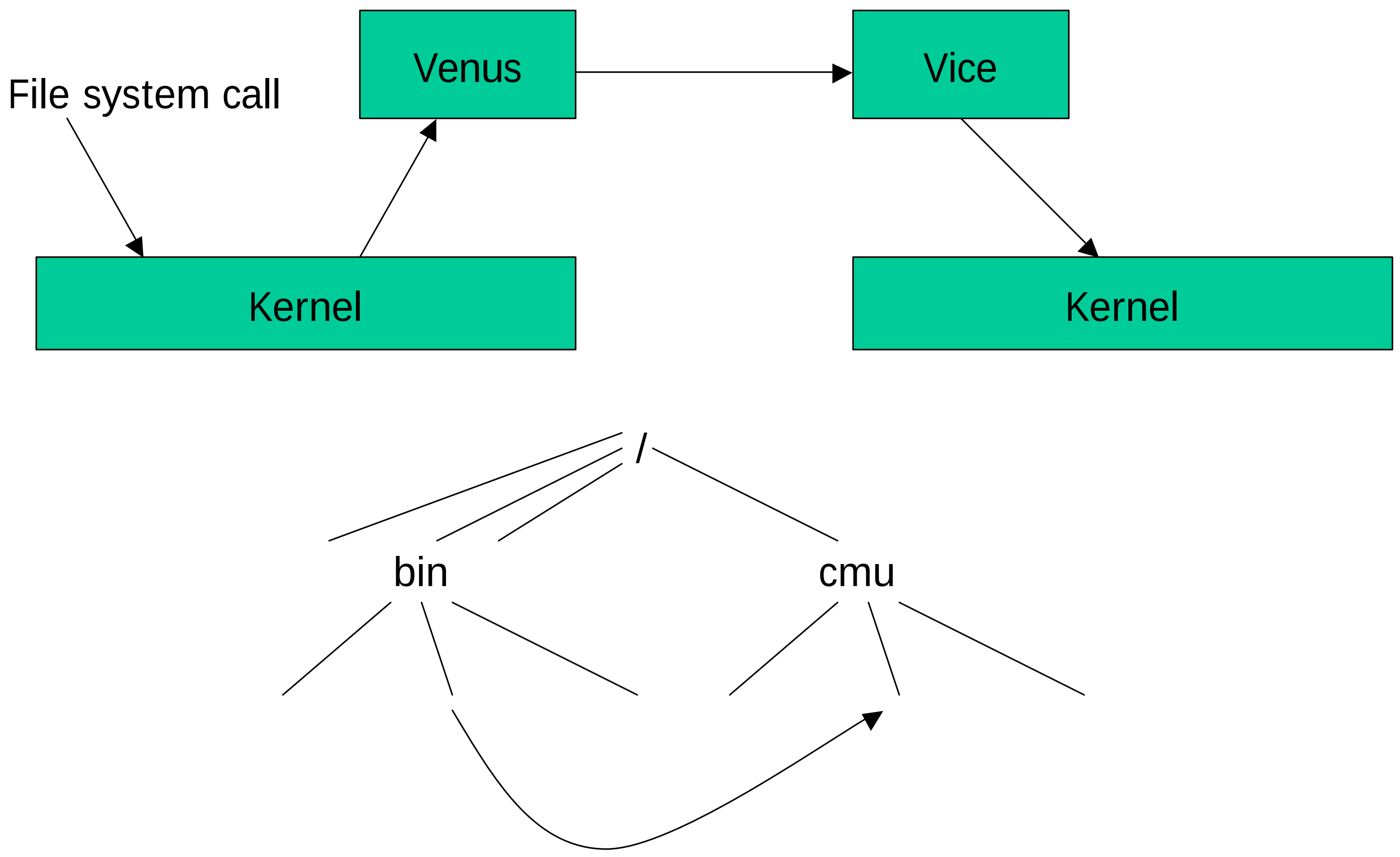
Other Case Studies

Andrew File System (AFS)

AFS

- Developed at Carnegie Mellon University starting in 1984
- Design goal is scalability

AFS – overview



AFS - transparency

- Access transparency
  - API same as for UNIX
- Location independency
  - Global name space (/cmu)
  - File identifier:

Volume number	File handle	Uniquifier

  - File /cmu/foo located at machine A can be moved to machine B
  - A volume location map is replicated at each server

AFS – scalability

- Scalability is achieved through
  - Whole-file serving
    - Entire files are transmitted to clients (64 KB blocks)
  - Whole-file caching
    - Large disk cache
  - Clustering

AFS – caching

- Validation of cache entries is done locally
- Servers job to invalidate clients cache entries (makes server stateful)
- Invalidation is done through callbacks.
- Callbacks are initially set up for all files in client cache. If modification to a file occur from another client the server breaks the callback. If callback exist a cache entry is valid.

AFS – update semantics

- Session semantics
- Successful open
  - latest(F,S) or
    (lostCallback(S,T) and inCache(F) and latest(F,S,T))
  - lostCallback(S,T) = A callback from the server has been lost during the last T seconds
  - T is typically 10 minutes

AFS – conclusion

- Session semantics
- Scalable
- Location independency
- Stateful server
- Consistency issues
  - Two simultaneous writes to the same position in a file. Former write is not present after latter write.
- Replication of read-only files

Coda

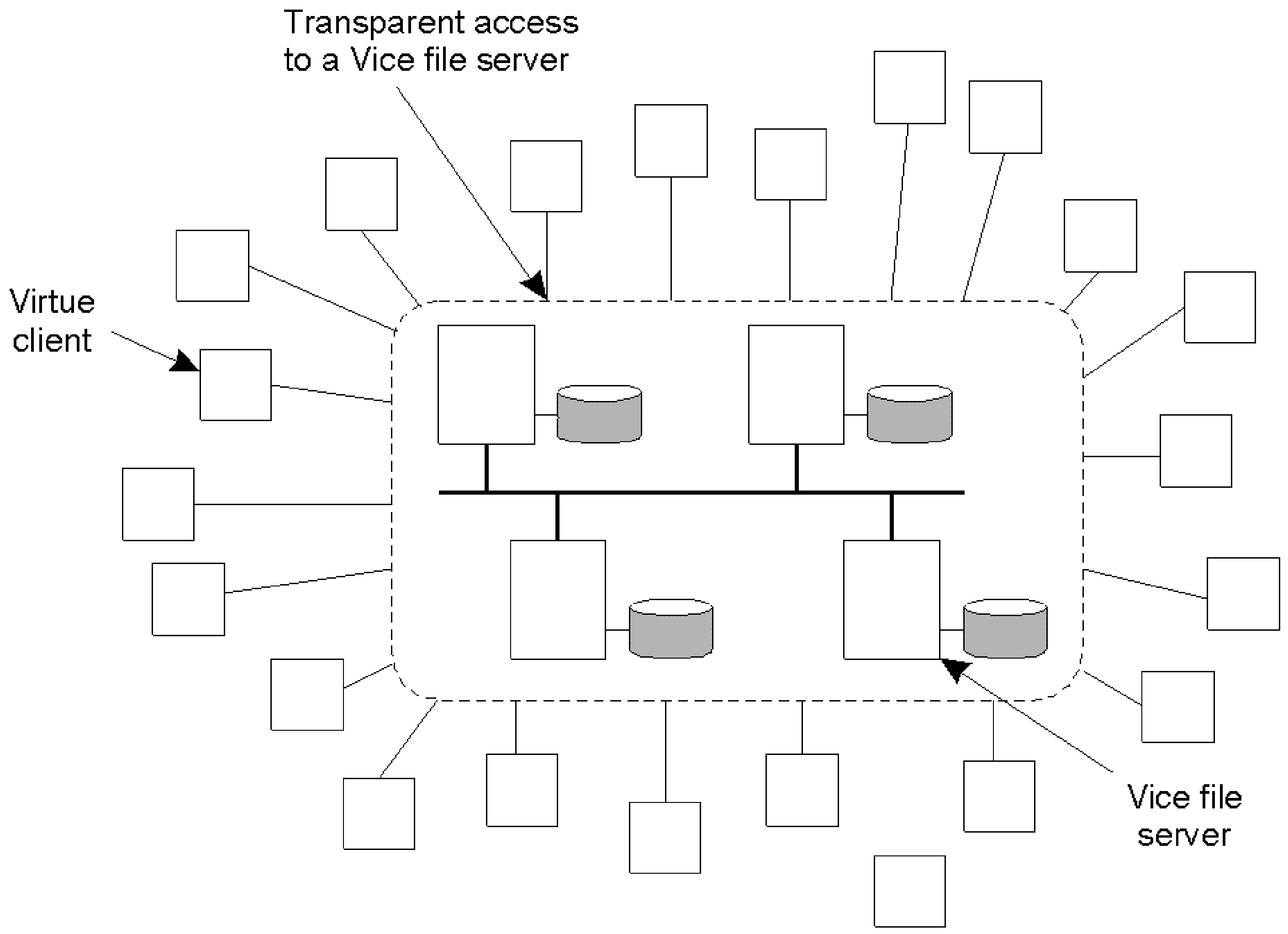
Coda

- Descendent of AFS
- Developed at Carnegie Mellon University since 1987
- Primary design goal is constant data availability
  - Achieved through
    - Server replication
    - Disconnected operation

Coda – overview

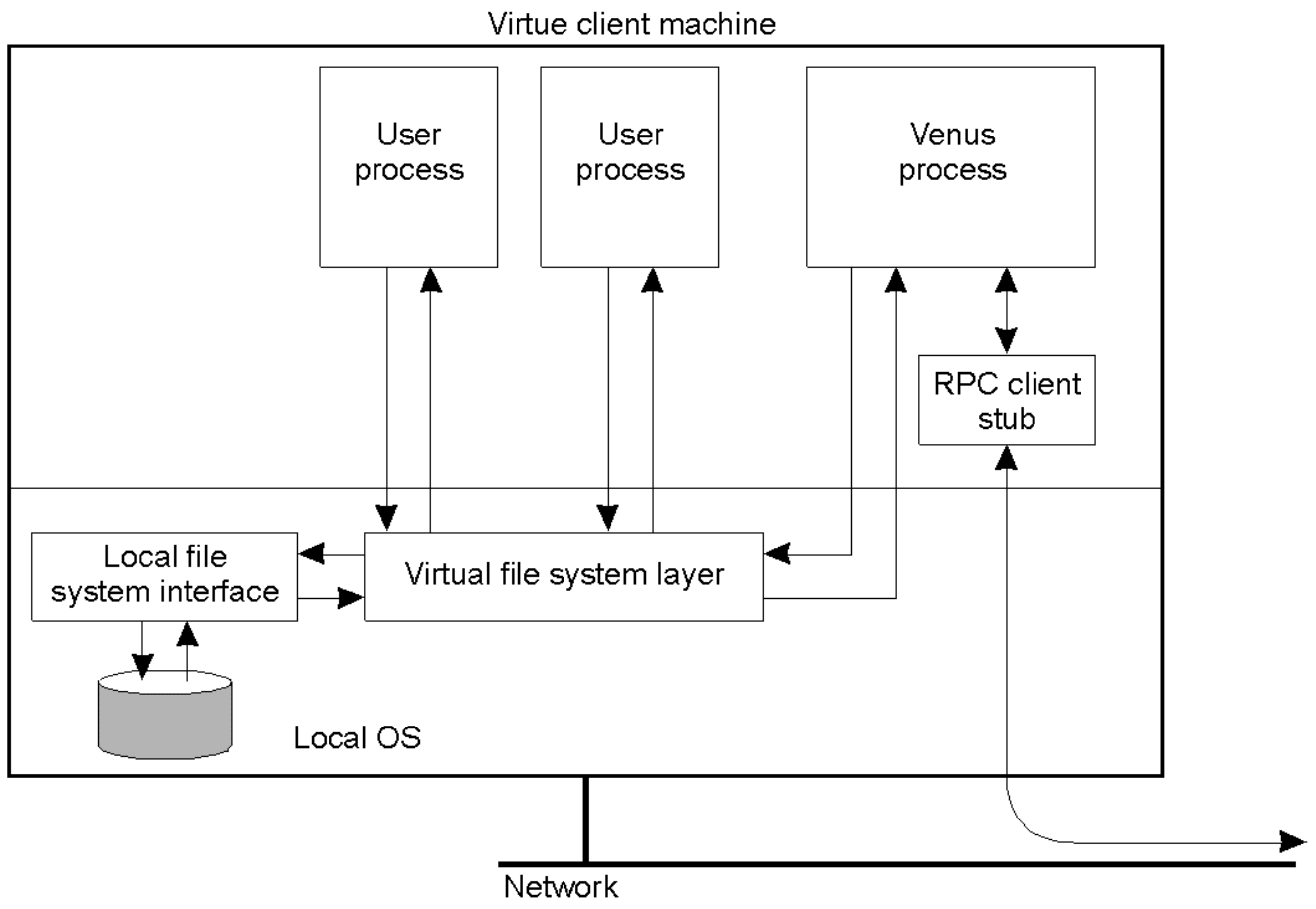
- A volume is replicated at a set of servers (volume storage group, VSG)
- Each Venus process has access to a subset of VSG (available VSG, AVSG)
- Each Venus process has a preferred server in AVSG.

Overview of Coda (1)



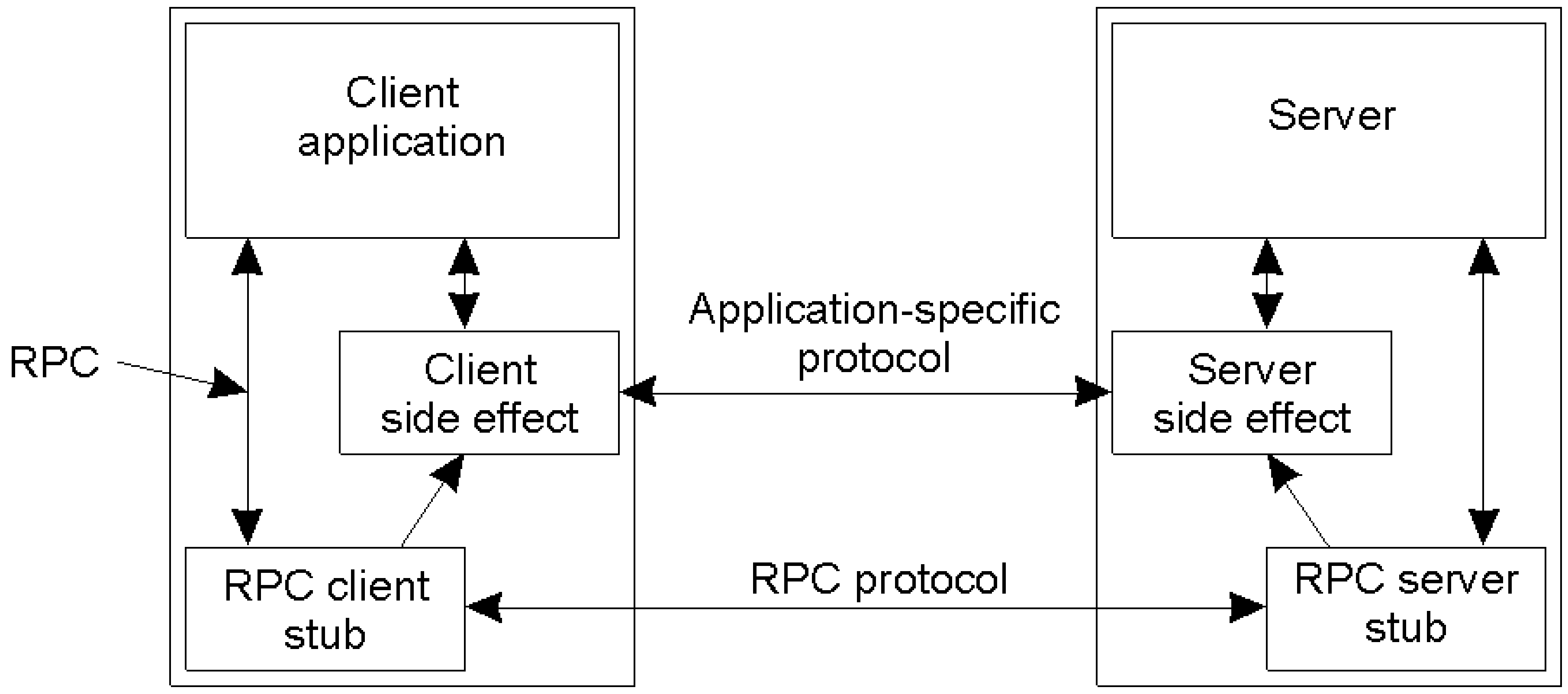
- The overall organization of AFS.

Overview of Coda (2)

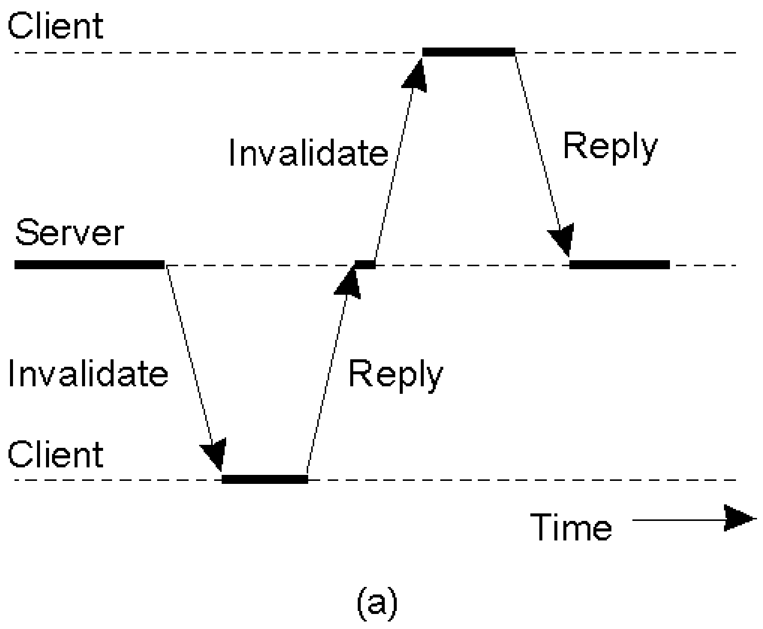


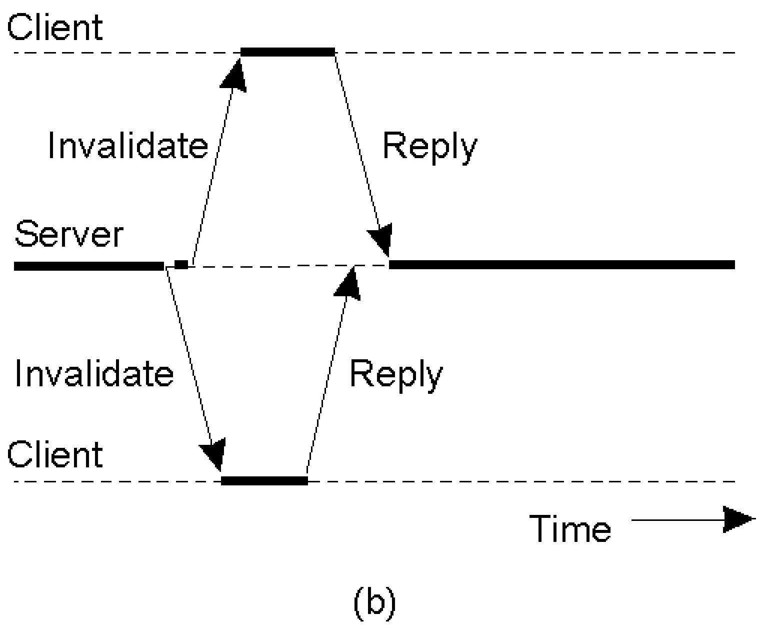
- The internal organization of a Virtue workstation.

Communication (1)



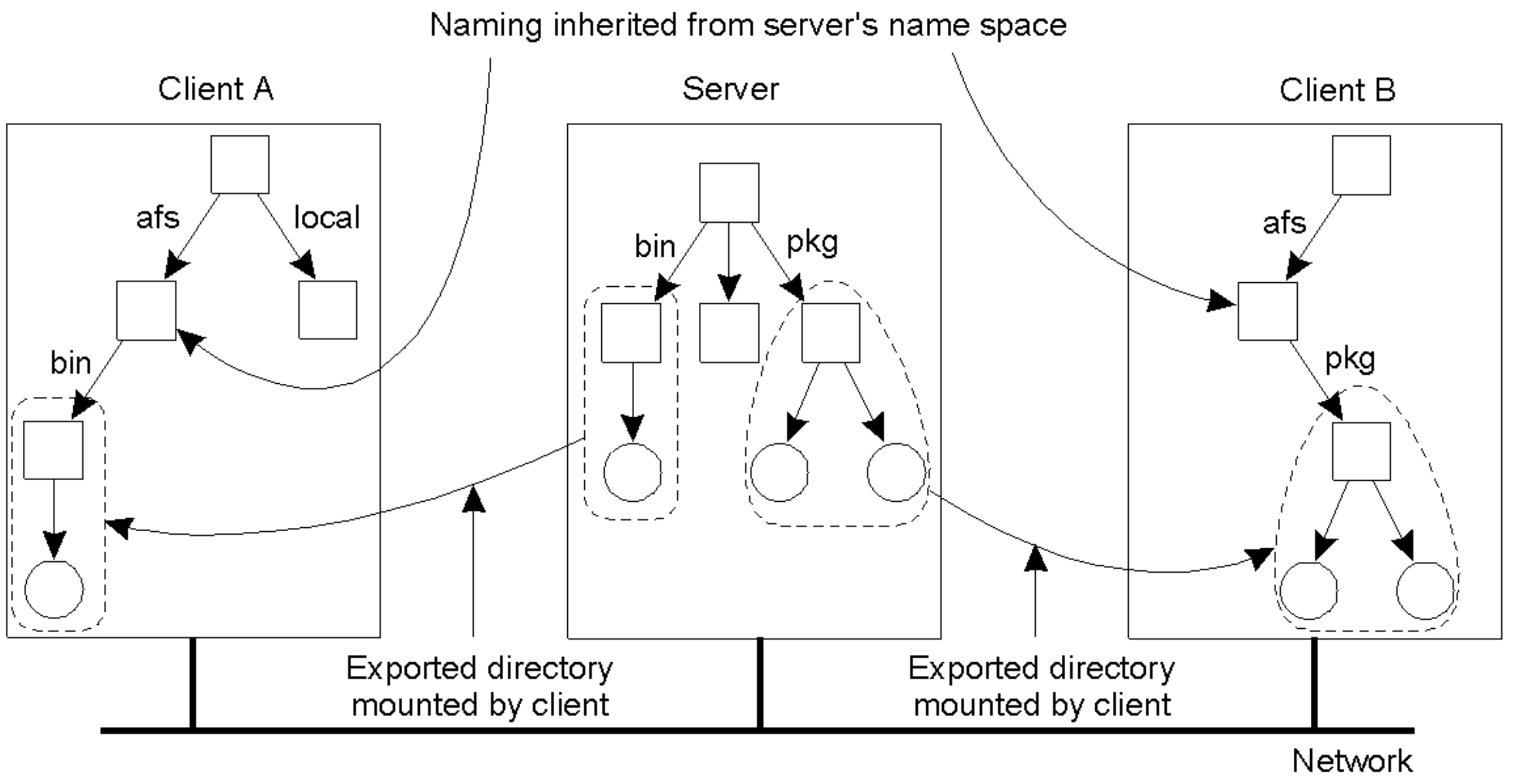
Communication (2)





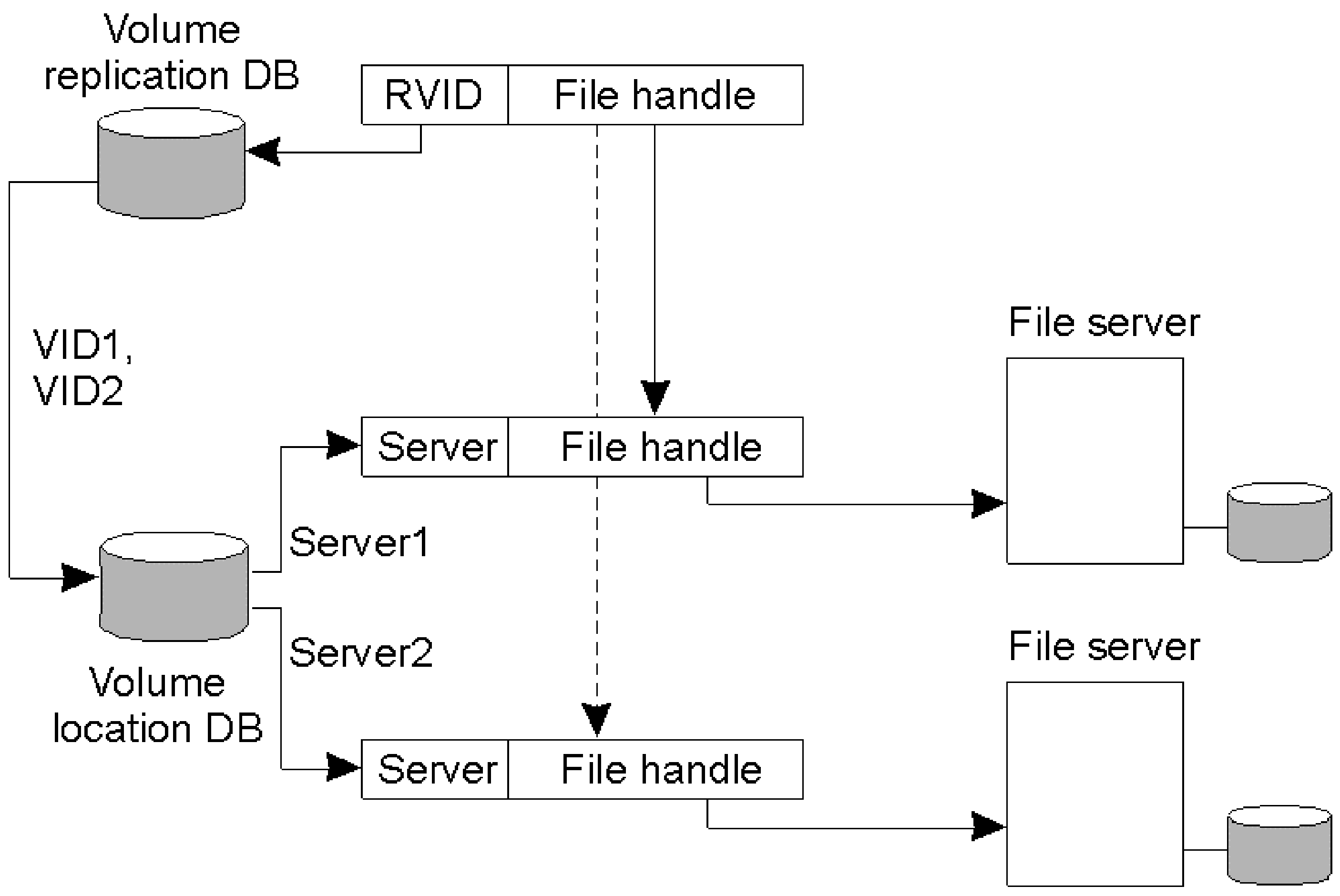
- Sending an invalidation message one at a time.
- Sending invalidation messages in parallel.

Naming


Naming inherited from server's name space
Client A
Server
Client B
afs
local
bin
bin
pkg
afs
pkg
Exported directory mounted by client
Exported directory mounted by client
Network

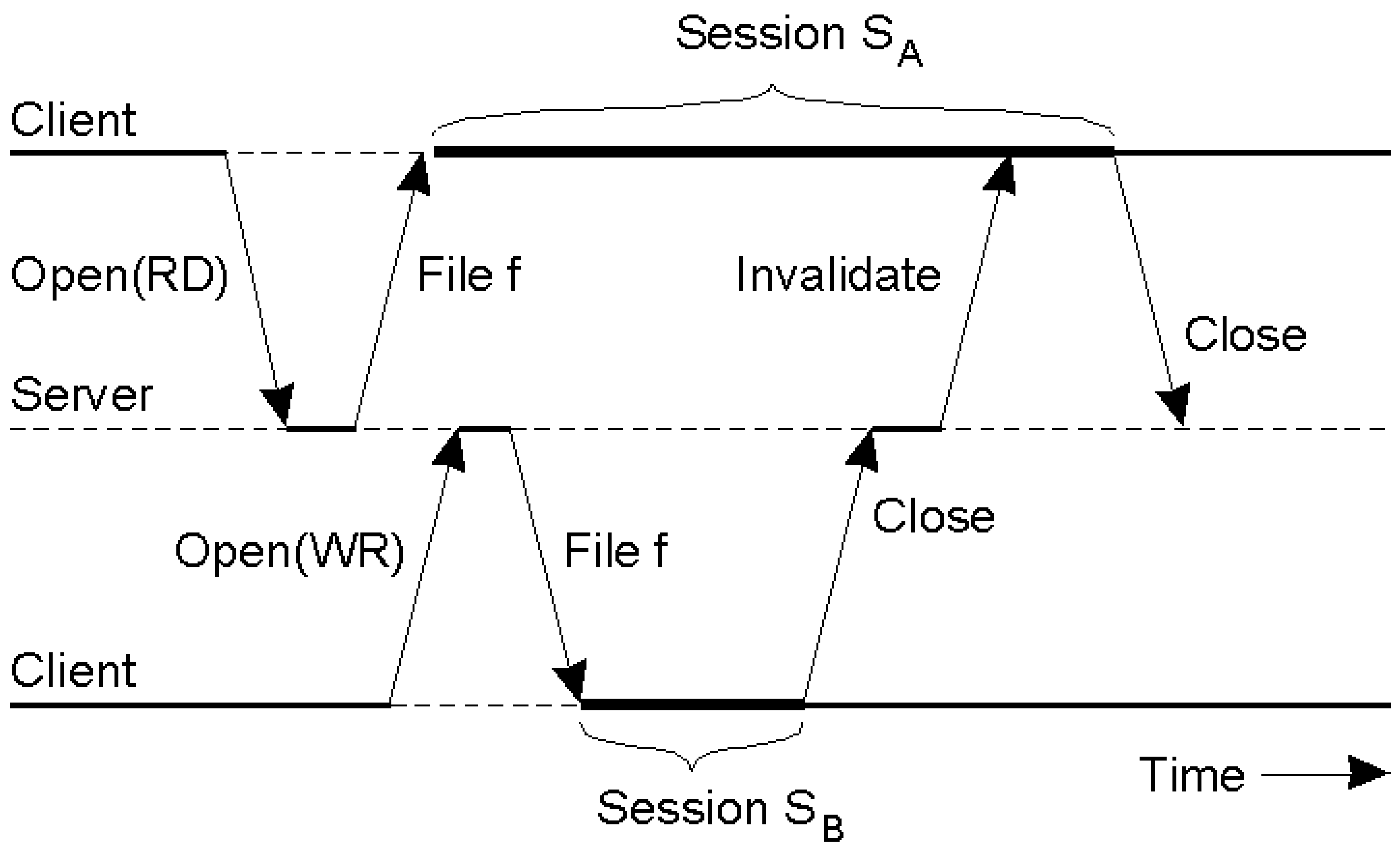
- Clients in Coda have access to a single shared name space.

File Identifiers



- The implementation and resolution of a Coda file identifier.

Sharing Files in Coda



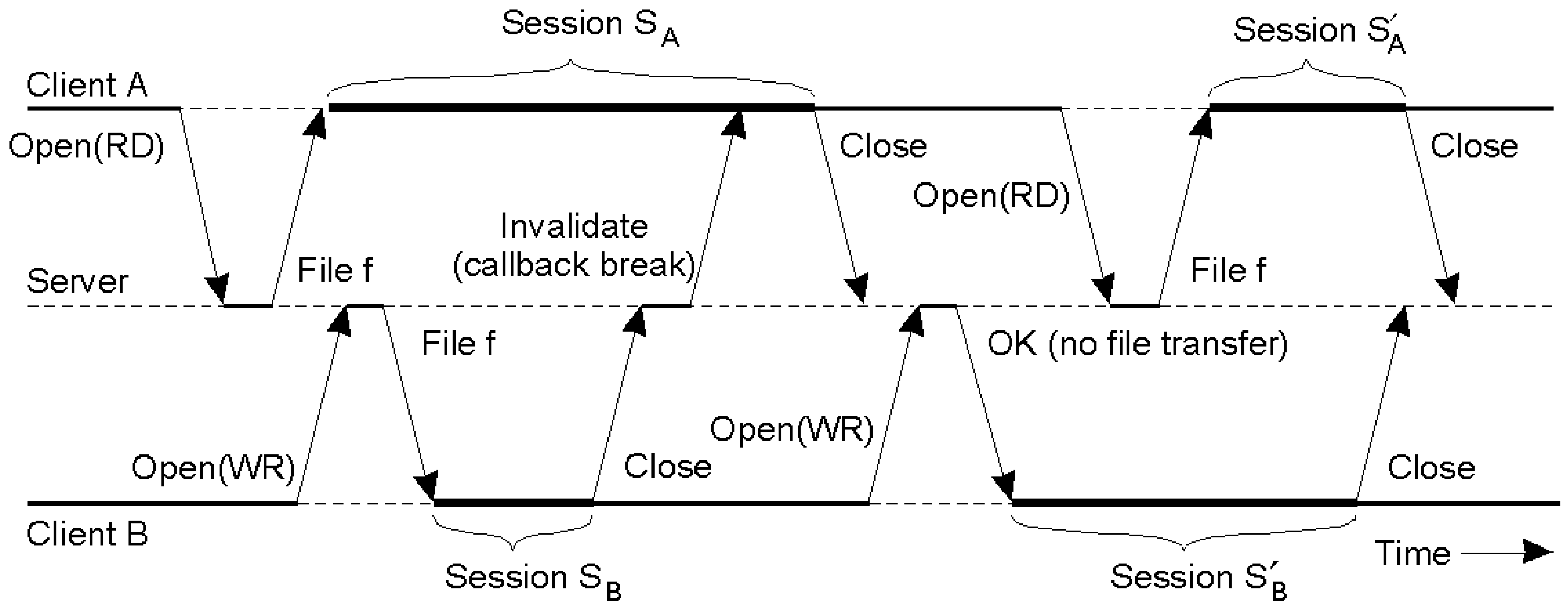
- The transactional behavior in sharing files in Coda.

Transactional Semantics

•**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

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

Client Caching



- The use of local copies when opening a session in Coda.

Coda – update semantics

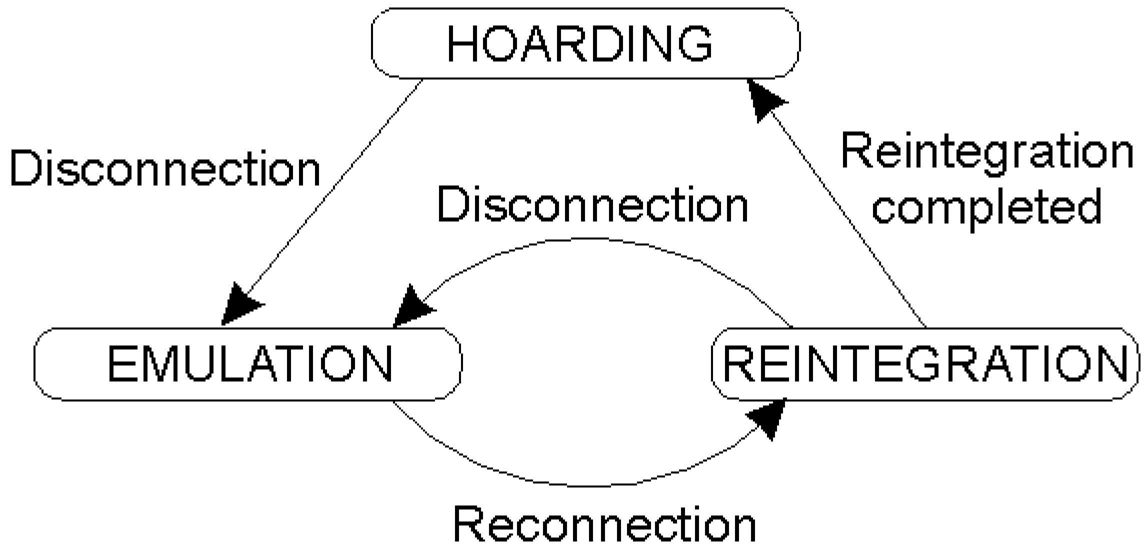
- Weaker than AFS semantic (not session semantics)
- Successful open
  - AVSG <> Ø and (latest(F,AVSG) or (lostCallback(AVSG,T) and inCache(F) and latest(F,AVSG,T))) or
    (AVSG = Ø and incache(F))
  - lostCallback(AVSG,T) = A callback from the server has been lost during the last T seconds
  - T is typically 10 minutes

Coda – disconnected operation

- Disconnected users can operate on files in their cache
- Can specify a list of files which Venus will try to keep in cache at all times
- Servers in VSG\AVSG are periodically polled
- Modified files are automatically transferred to preferred server upon reconnection

Disconnected Operation

- The state-transition diagram of a Coda client with respect to a volume.



Coda - conflicts

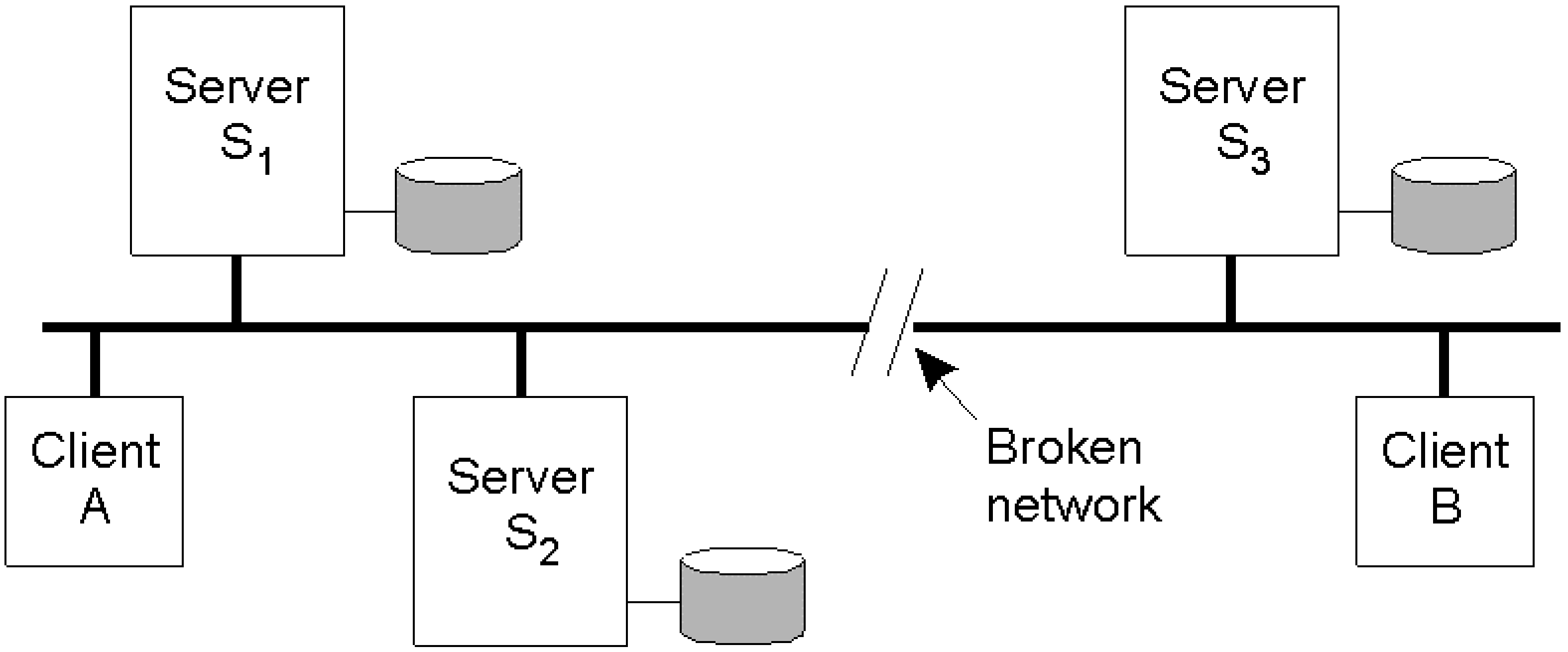
- Coda Version Vectors (CVV) are used to detect conflicts
- Each file has a CVV associated with it
- Example: 3 servers and 2 clients
  - Two partitions {S1,S2,C1} and {S3,C2}
  - Initially CVV is (1,1,1)
  - C1 updates file → (2,2,1)
  - C2 updates file → (1,1,3)
  - When the partitions merge we have a conflict because neither $CVV_{C1} \geq CVV_{C2}$ nor $CVV_{C2} \geq CVV_{C1}$
- Automatic conflict detection
  - Some directory conflicts can be automatically resolved

Coda – server replication

- Upon close modifications propagate to AVSG with multiRPC (multicast)
- All servers are contacted when opening a file to make sure the preferred server has the latest copy and that all replicas are in sync
- So, clients are responsible for server replication

Server Replication

- Two clients with different AVSG for the same replicated file.



Access Control in Coda

•**Operation**	•**Description**
•Read	•Read any file in the directory
•Write	•Modify any file in the directory
•Lookup	•Look up the status of any file
•Insert	•Add a new file to the directory
•Delete	•Delete an existing file
•Administer	•Modify the ACL of the directory

- Classification of file and directory operations recognized by Coda with respect to access control.

Background Reading Material

- NFS:
  - rfc 1094 for v2 (3/1989)
  - rfc 1813 for v3 (6/1995)
  - rfc 3530 for v4 (4/2003)

- AFS: “Scale and Performance in a Distributed File System”, TOCS Feb 1988
  - http://www-2.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/s11.pdf

- “Sprite”: “Caching in the Sprite Network File Systems”, TOCS Feb 1988
  - http://www.cs.berkeley.edu/projects/sprite/papers/caching.ps

More Reading Material

- CIFS spec:
  - http://www.itl.ohiou.edu/CIFS-SPEC-0P9-REVIEW.pdf
- CODA file system:
  - http://www-2.cs.cmu.edu/afs/cs/project/coda/Web/docdir/s13.pdf

- RPC related RFCs:
  - XDR representation: RFC 1831
  - RPC: RFCS 1832
  - RPC security: RFC 2203

Spare Slides

What Distributed File System Provides

- Provide accesses to date stored at servers using *file system interfaces*
- What are the file system interfaces?
  - Open a file, check status on a file, close a file;
  - Read data from a file;
  - Write data to a file;
  - Lock a file or part of a file;
  - List files in a directory, delete a directory;
  - Delete a file, rename a file, add a symlink to a file;
  - etc;

Buzz Words: NAS vs SAN

	NAS	SAN
Access Methods	File access	Disk block access
Access Medium	Ethernet	Fiber Channel and Ethernet
Transport Protocol	Layer over TCP/IP	SCSI/FC and SCSI/IP
Efficiency	Less	More
Sharing and Access Control	Good	Poor
Integrity demands	Strong	Very strong
Clients	Workstations	Database servers

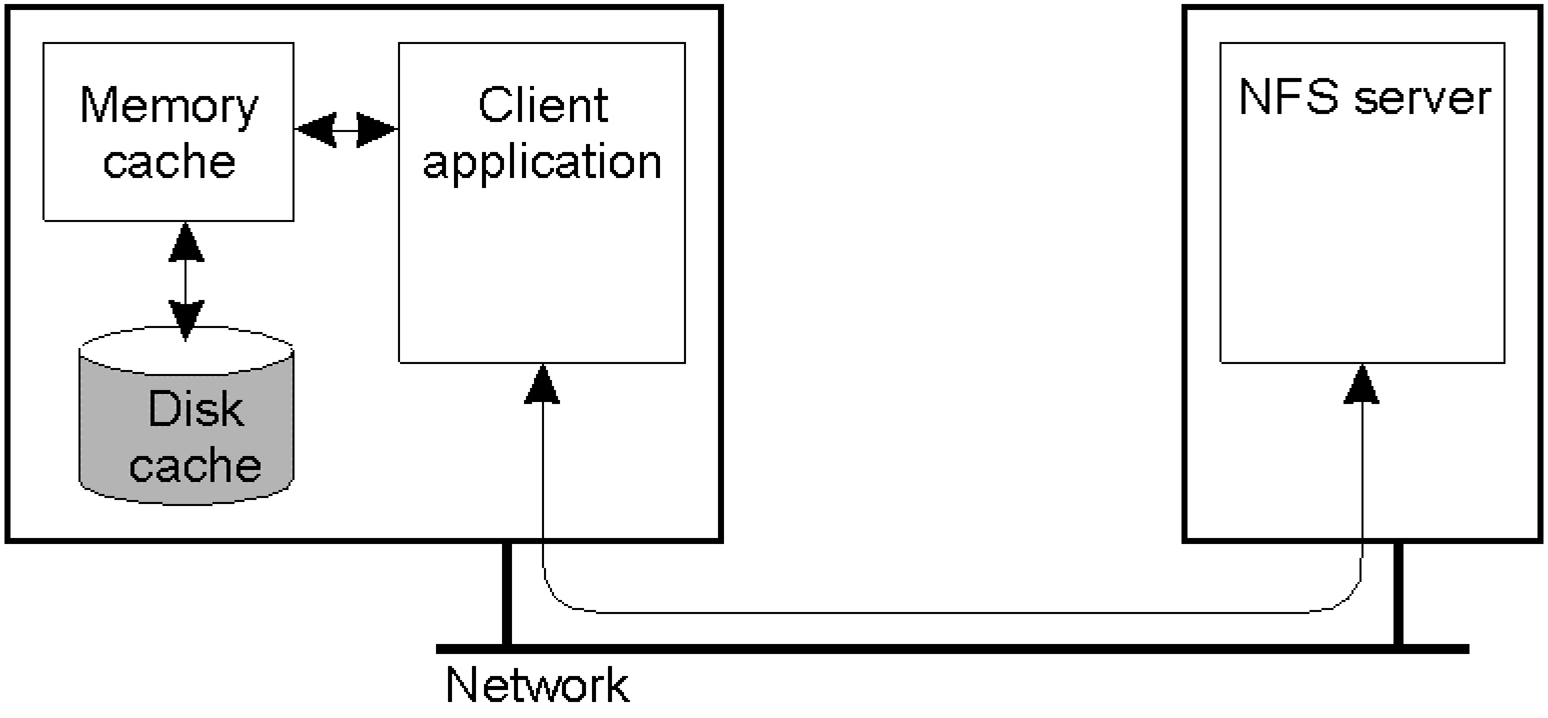
NFS

- A specification for a distributed file system (by Sun, 1984)
- Implemented on various OS’s
- De facto standard in the UNIX community
- Latest version is 4 (2000, RFC3010)
- Client-server file system

NFS Client Server Interactions

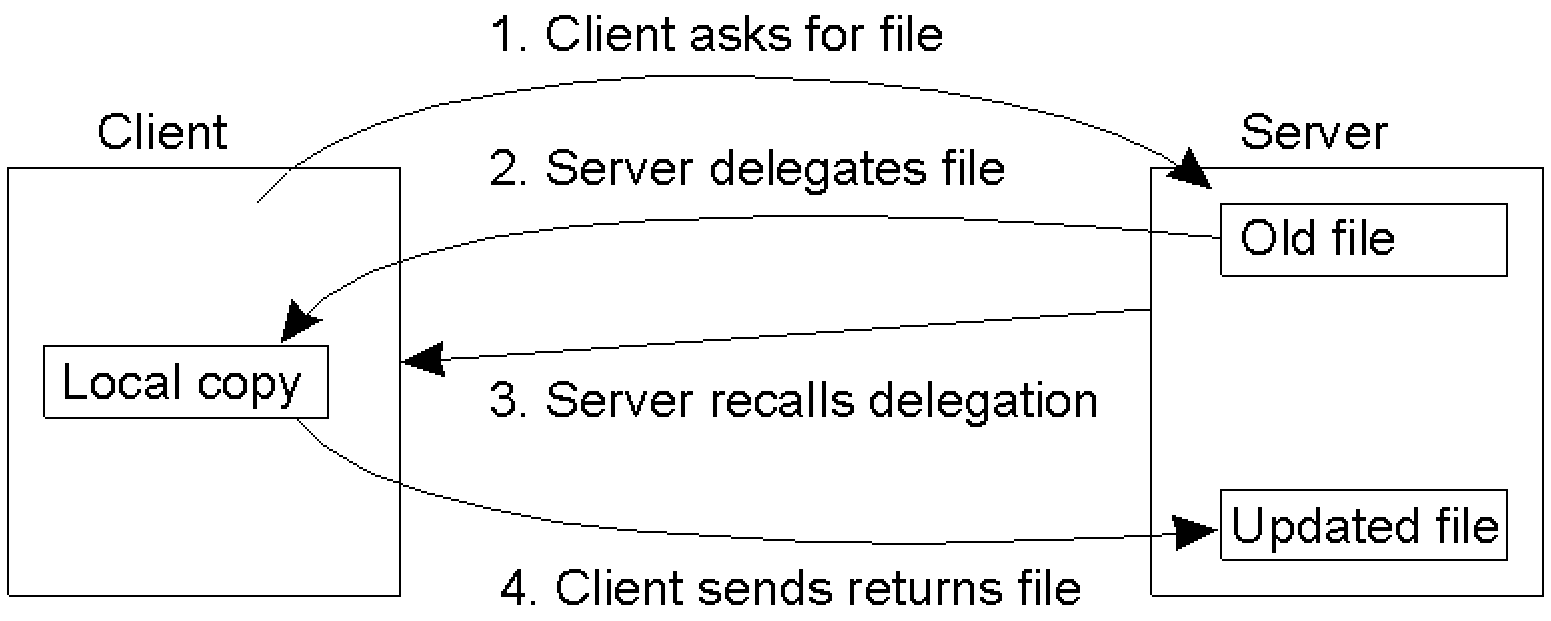
- Client machine
  - Application → nfs_vnops-> nfs client code -> rpc client interface
- Server machine
  - rpc server interface → nfs server code → ufs_vops -> ufs code -> disks

Client Caching (1)



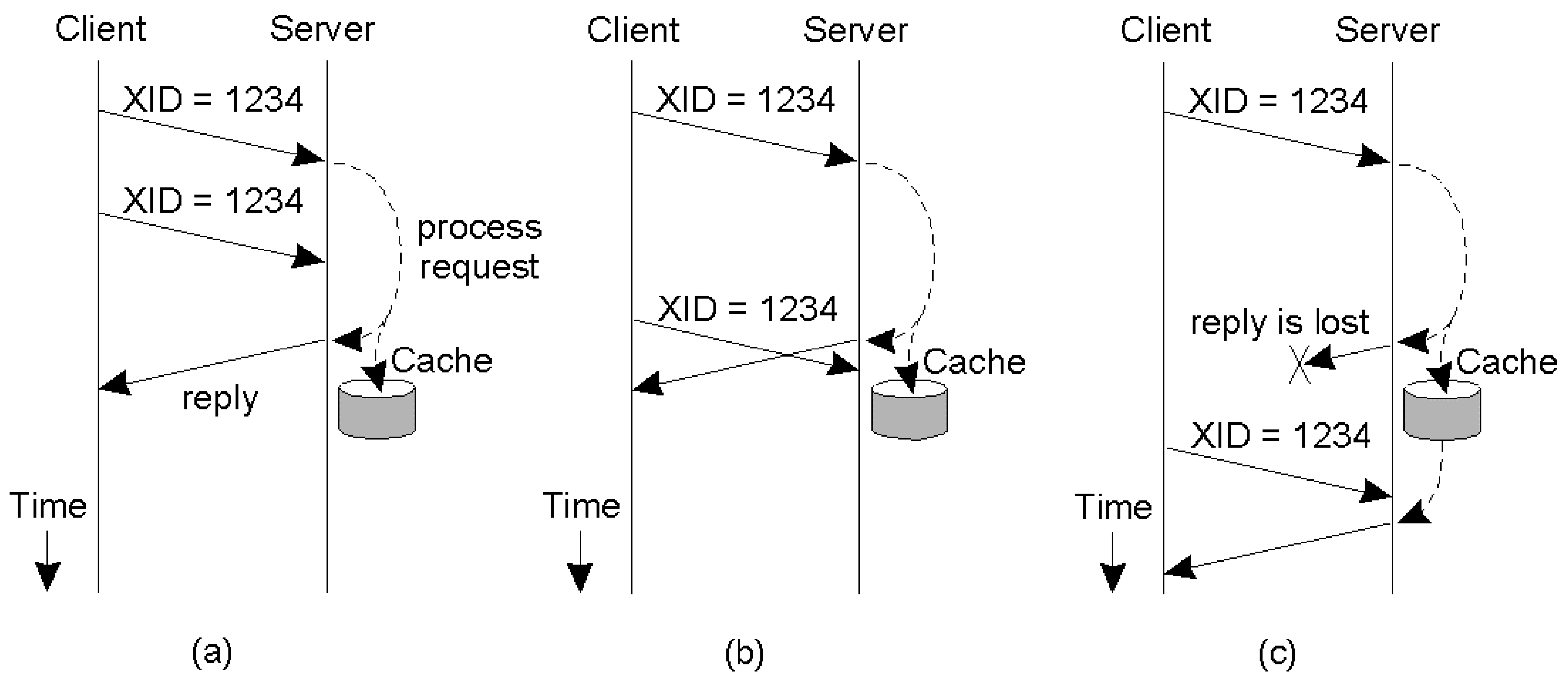
- Client-side caching in NFS.

NFS V4 Delegation



- Using the NFS version 4 callback mechanism to recall file delegation.

RPC Failures



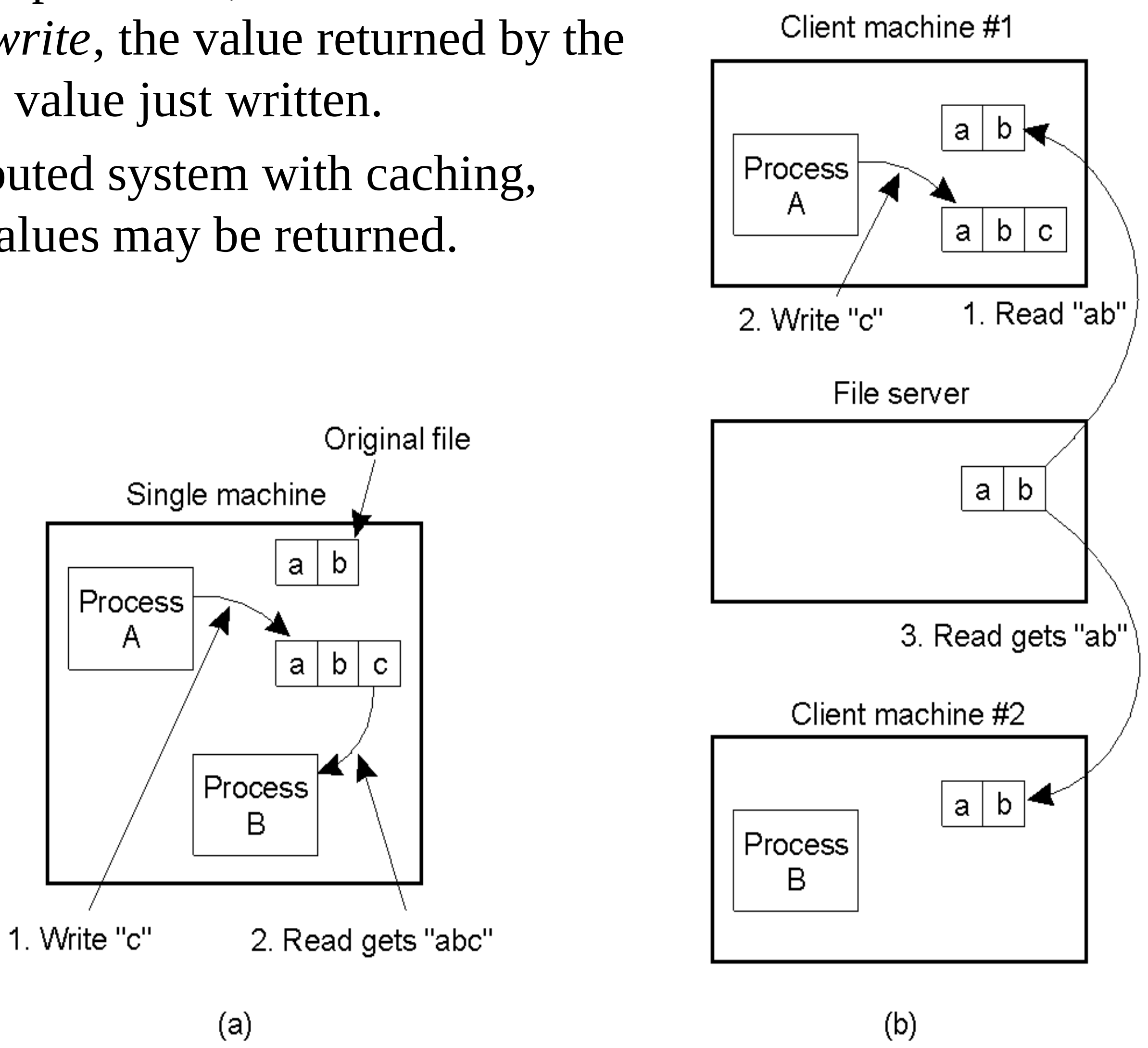
- Three situations for handling retransmissions.
- The request is still in progress
- The reply has just been returned
- The reply has been some time ago, but was lost.

Replication

- Reasons:
  - Increase reliability
  - improve availability
  - balance the servers’ workload
- how to make replication transparent (Fig. 13-12)
- how to keep the replicas consistent
  - Problems -- mainly with updates
    1. a replica is not updated due to its server failure
    2. network partitioned
- Replication Management:
  1. weighted vote for read and write
  2. current synchronization site for each file group to control access

Semantics of File Sharing (1)

- On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.
- In a distributed system with caching, obsolete values may be returned.



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.

NFS vs AFS: Name-Space

- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1 → fhandle
- AFS: global name space
  - Name space is organized into Volumes
    - Global directory /afs;
    - /afs/cs.wisc.edu/vol1/…; /afs/cs.stanfod.edu/vol1/…
  - Each file is identified as <vol_id, vnode#, vnode_gen>
  - All AFS servers keep a copy of "volume location database", which is a table of vol_id→ server_ip mappings

NFS vs AFS: Implications for Location Transparency

- NFS: no transparency
  - If a directory is moved from one server to another, client must remount
- AFS: transparency
  - If a volume is moved from one server to another, only the volume location database on the servers needs to be updated
  - Implementation of volume migration
  - File lookup efficiency
- Are there other ways to provide location transparency?

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
•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

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

File Attributes (1)

•**Attribute**	•**Description**
•TYPE	•The type of the file (regular, directory, symbolic link)
•SIZE	•The length of the file in bytes
•CHANGE	•Indicator for a client to see if and/or when the file has changed
•FSID	•Server-unique identifier of the file's file system

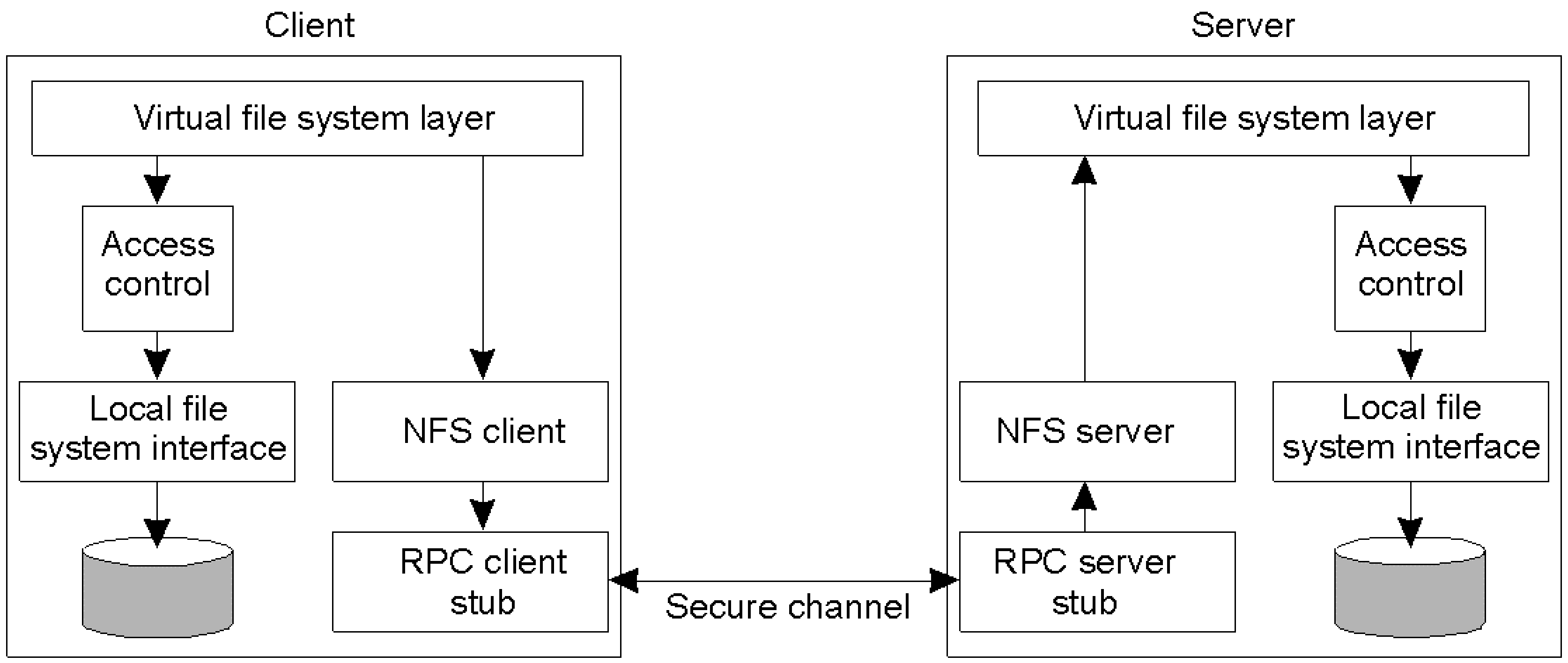
- Some general mandatory file attributes in NFS.

File Attributes (2)

•Attribute	•Description
•ACL	•an access control list associated with the file
•FILEHANDLE	•The server-provided file handle of this file
•FILEID	•A file-system unique identifier for this file
•FS_LOCATIONS	•Locations in the network where this file system may be found
•OWNER	•The 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 the file was created

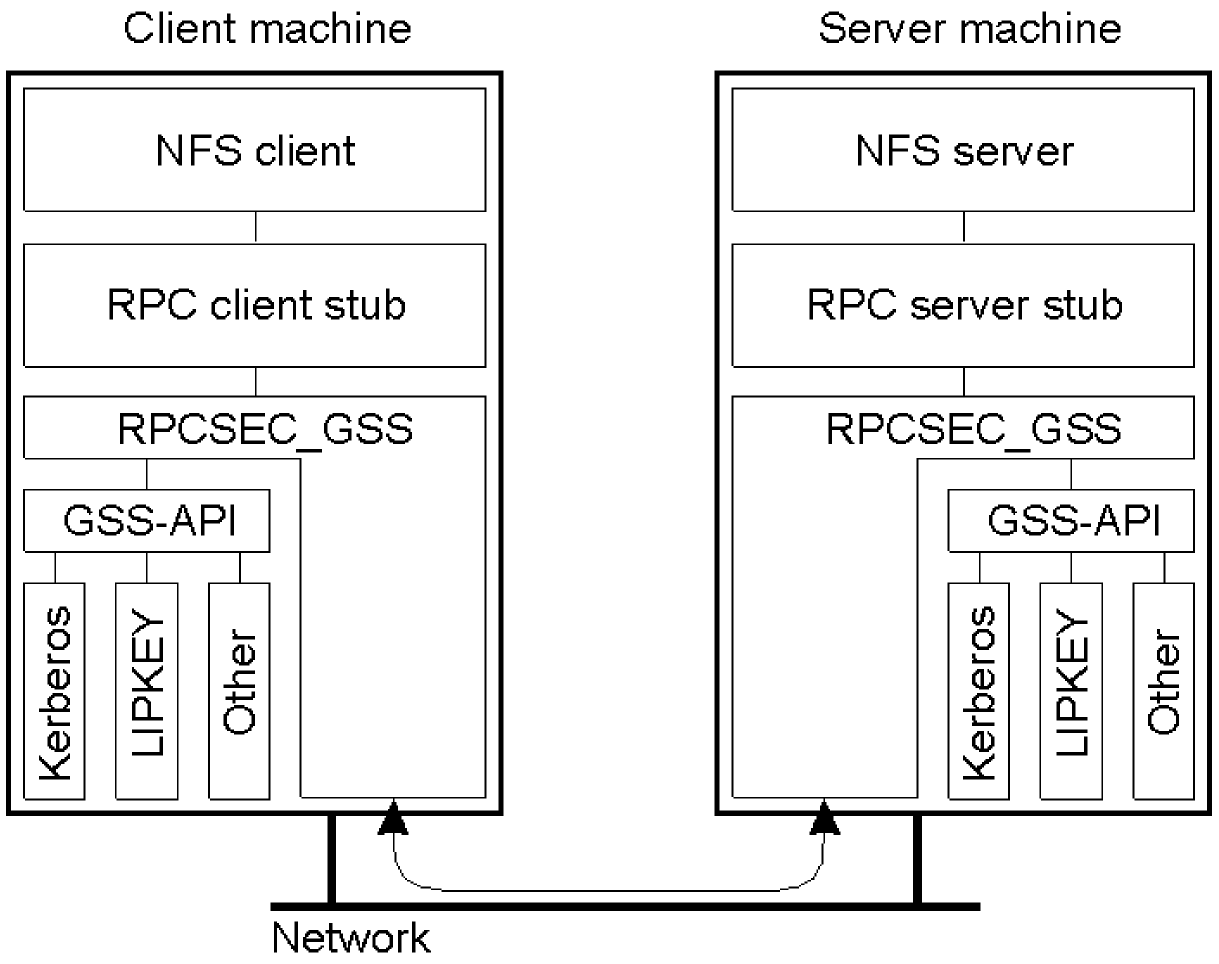
- Some general recommended file attributes.

Security


Client
Server
Virtual file system layer
Access control
Local file system interface
NFS client
RPC client stub
Secure channel
Virtual file system layer
Access control
NFS server
Local file system interface
RPC server stub

- The NFS security architecture.

Secure RPCs



- Secure RPC in NFS version 4.

NFS Access Control

•**Operation**	•**Description**
•Read_data	•Permission to read the data contained in a file
•Write_data	•Permission to to modify a file's data
•Append_data	•Permission to to append data to a file
•Execute	•Permission to to execute a file
•List_directory	•Permission to to list the contents of a directory
•Add_file	•Permission to to add a new file t5o a directory
•Add_subdirectory	•Permission to to create a subdirectory to a directory
•Delete	•Permission to to delete a file
•Delete_child	•Permission to to delete a file or directory within a directory
•Read_acl	•Permission to to read the ACL
•Write_acl	•Permission to to write the ACL
•Read_attributes	•The ability to read the other basic attributes of a file
•Write_attributes	•Permission to to change the other basic attributes of a file
•Read_named_attrs	•Permission to to read the named attributes of a file
•Write_named_attrs	•Permission to to write the named attributes of a file
•Write_owner	•Permission to to change the owner
•Synchronize	•Permission to to access a file locally at the server with synchronous reads and writes

- The classification of operations recognized by NFS with respect to access control.

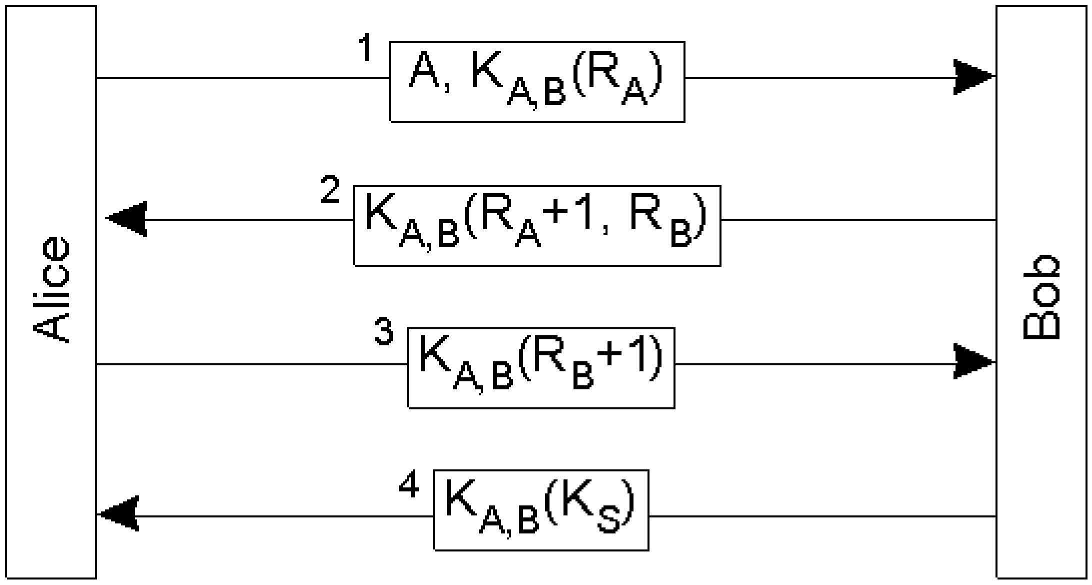
NFS Access Control

•**Type of user**	•**Description**
•Owner	•The owner of a file
•Group	•The group of users associated with a file
•Everyone	•Any user of a 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 a batch job
•Anonymous	•Anyone accessing the file without authentication
•Authenticated	•Any authenticated user of a process
•Service	•Any system-defined service process

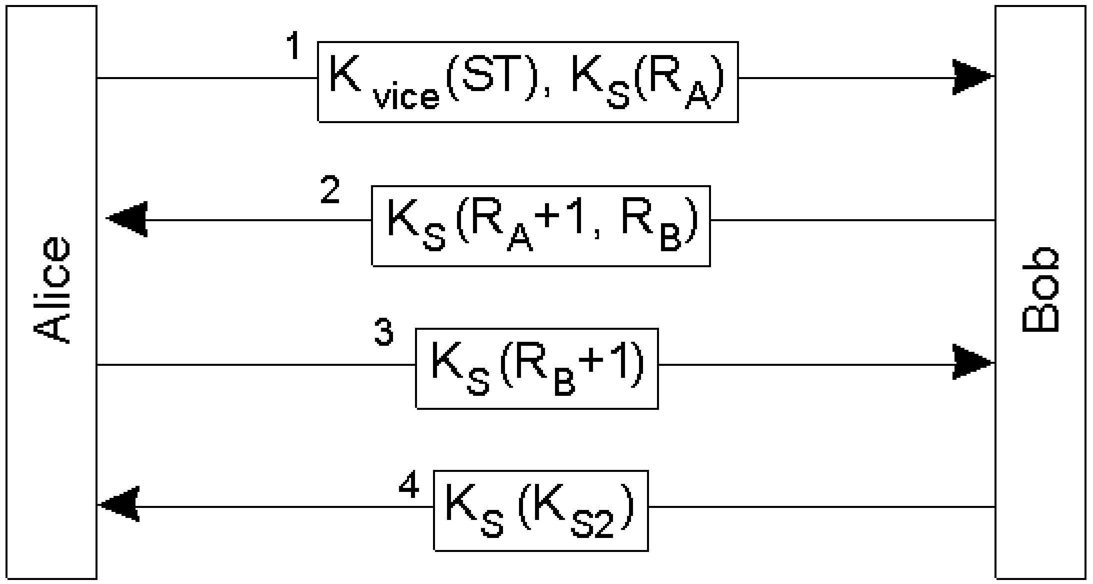
- The various kinds of users and processes distinguished by NFS with respect to access control.

Secure Channels in Coda

- Mutual authentication in RPC2.



Secure Channels in Coda



- Setting up a secure channel between a (Venus) client and a Vice server in Coda.