
7. Distributed File Systems

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2017/2018 Q2

Contents

- **Architecture**
- Communication
- Naming
- Synchronization
- Consistency & replication
- Fault tolerance

Contents

- Architecture
 - Client-server architectures
 - Cluster-based architectures
- Communication
- Naming
- Synchronization
- Consistency & replication
- Fault tolerance

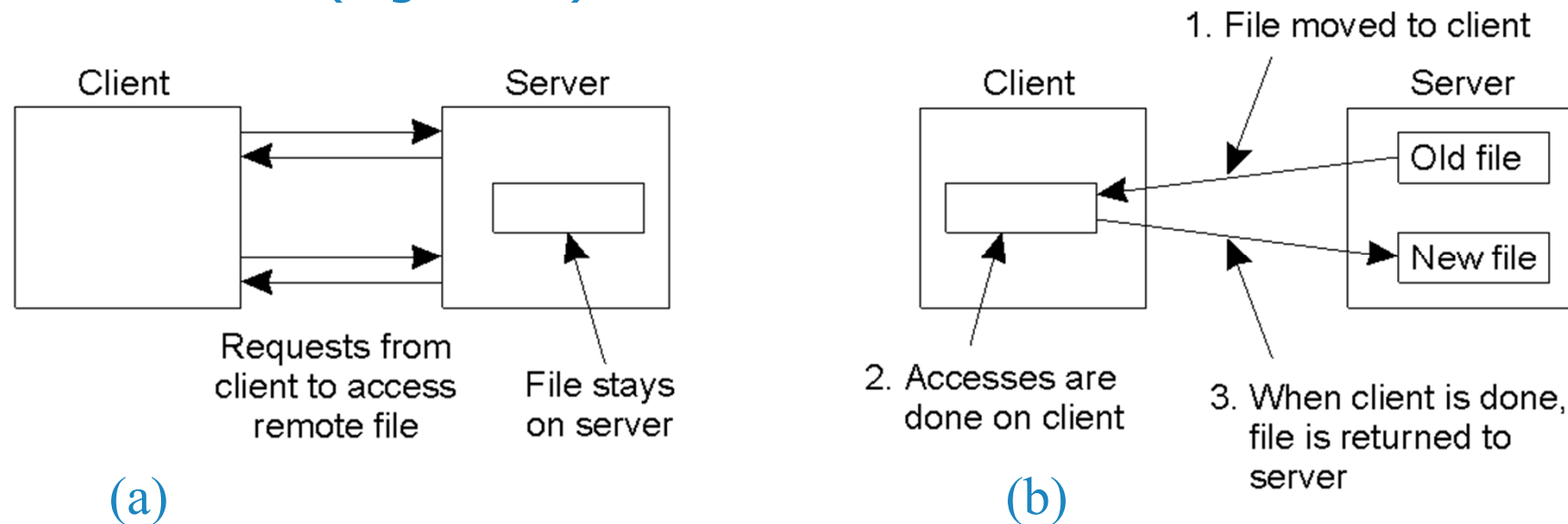
Client-server architectures

a) Remote access model (a.k.a. remote file service)

- Clients are offered transparent access to a file system that is managed by a remote server (e.g. NFS)

b) Upload/download model

- Client accesses locally to a file after downloading it from the server (e.g. Coda)



Client-server architectures

Remote access model

- Most of the work done at the server
- ↑ Consistency is easy
- ↑ Client does not need storage
- ↓ Server is a bottleneck
- ↓ Low tolerance to server crash and partitions
- Low communication cost to open files but high to operate on them

Upload/download model

- Most of the work done at the client
- ↓ Consistency is hard
- ↓ Client downloads file: needs storage
- ↑ Load is distributed
- ↑ Tolerance to server crash and partitions
- High communication cost to open files but low to operate on them

Client-server architectures

Stateful server

- Server provides open and close operations
- Server keeps client state between requests on the same file

↑ Better performance

↓ Server state must be recovered after a crash

Stateless server

- Server does not keep any client information between requests
 - Each request must provide complete data to execute operation

↓ Worse performance

↑ No need to restore any state after a crash

↓ Cannot be always followed to full extent

- e.g. file locking

NFS architecture

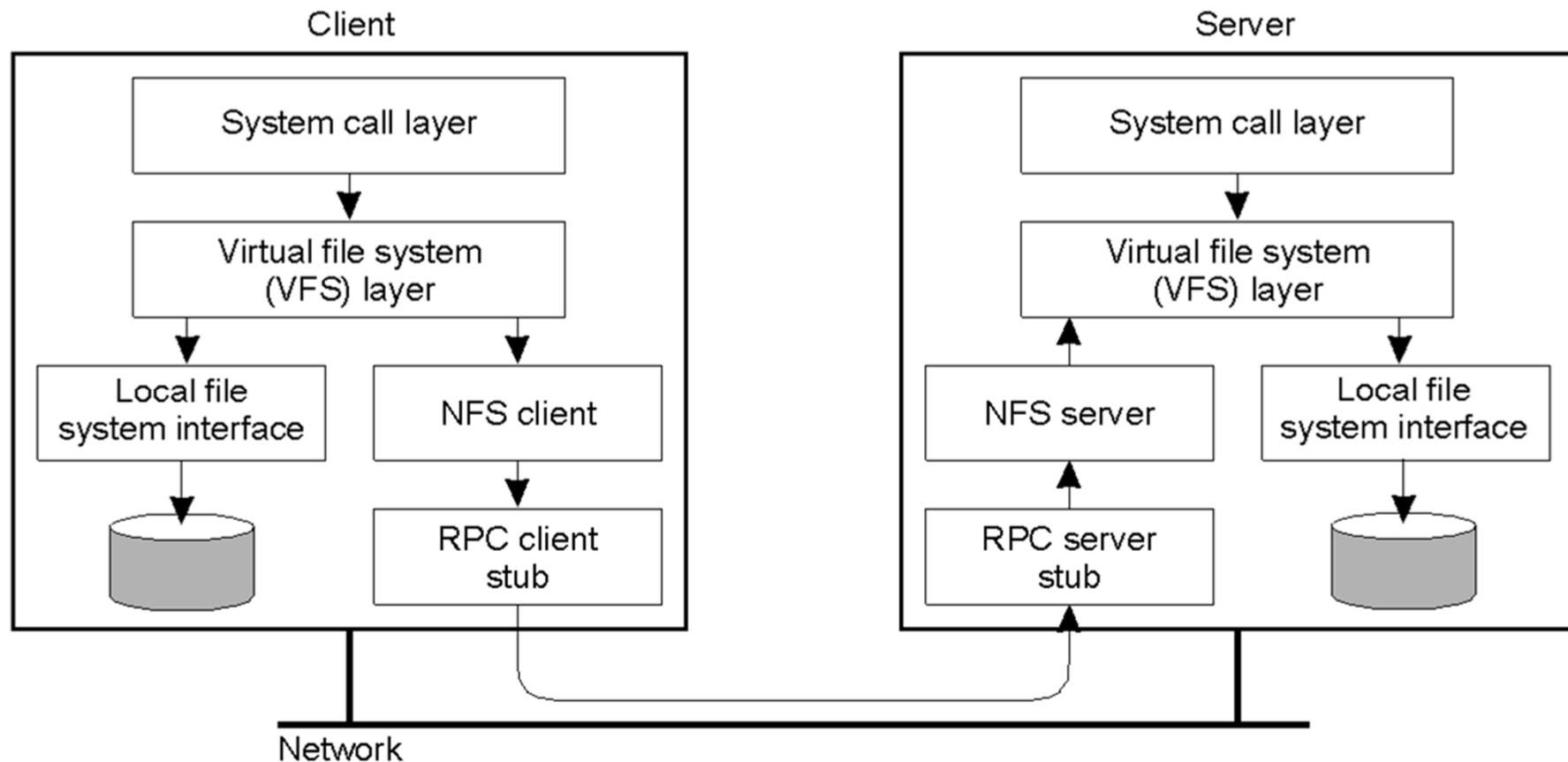
- **Sun Network File System (NFS)**
 - Designed for use on UNIX-based systems, but has been implemented for many OS
 - Client-server file system
 - Each server provides a standardized view of its local file system, no matter how it is implemented
 - Clients access files through a communication protocol
 - File system model similar to UNIX
 - Files treated as uninterpreted sequences of bytes
 - Files have name, but are referred by a file handle (i.e. file identifier)
 - First NFS versions were stateless, v4 is stateful

NFS architecture

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

NFS architecture

- **Virtual File System (VFS)** provides uniform & transparent access to local and remote files



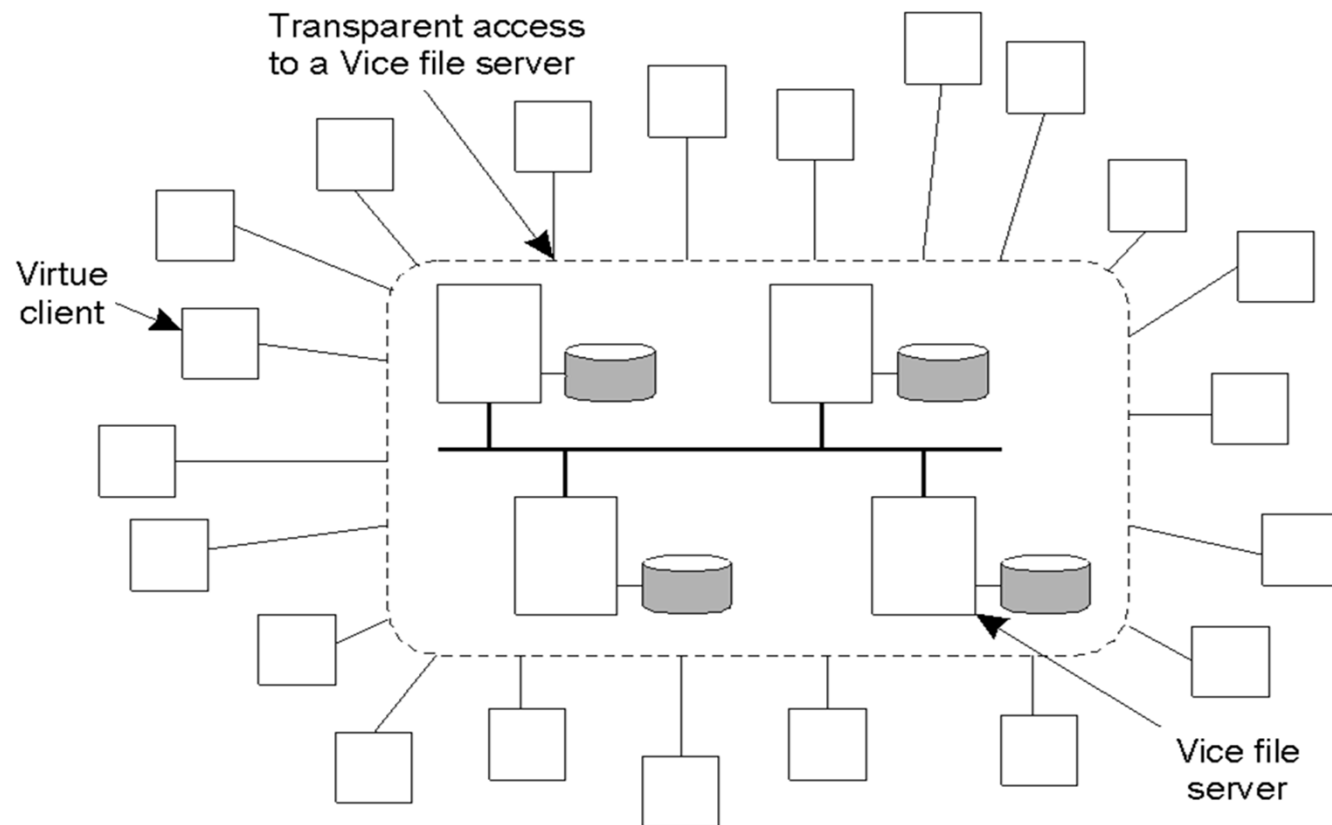
Coda architecture

- **Coda File System**

- Based on Andrew File System (AFS v2)
- Goals:
 - Scalability: support for a large number of users
 - Availability: tolerate server/communication failures and voluntary client disconnections (mobile users)
- Design philosophy: “Scalability and availability more important than consistency”
- ⇒ Use upload/download model
 - Whole-file serving and caching
 - AFS v3 allows file data to be transferred and cached in 64-kbyte blocks

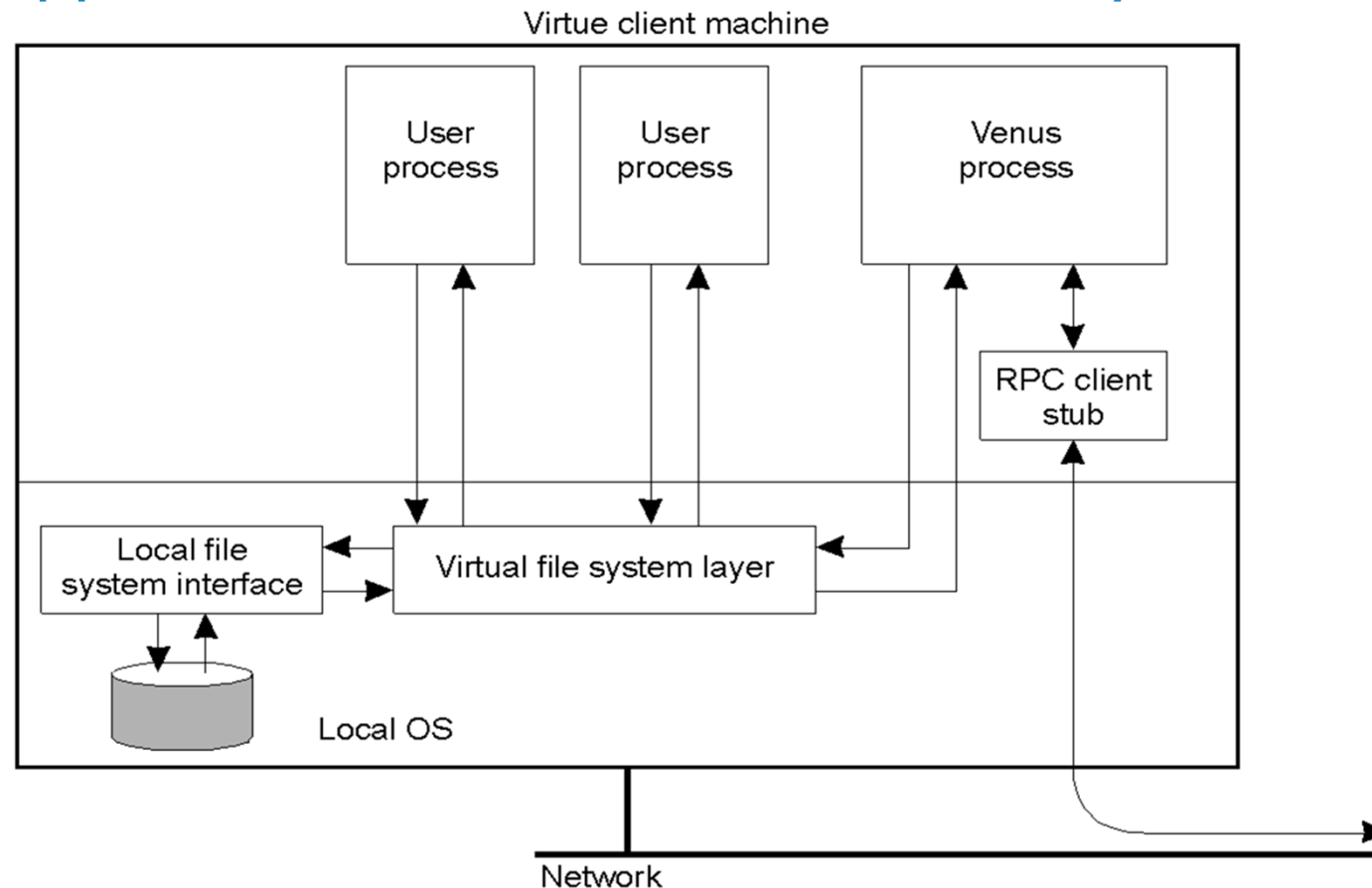
Coda architecture

- Few centrally administered **Vice** file servers
- Large number of **Virtue** clients
 - Hosting a user-level process **Venus** (like NFS client)



Coda architecture

- Virtue clients use VFS (like NFS does)
 - Appears like a traditional UNIX file system

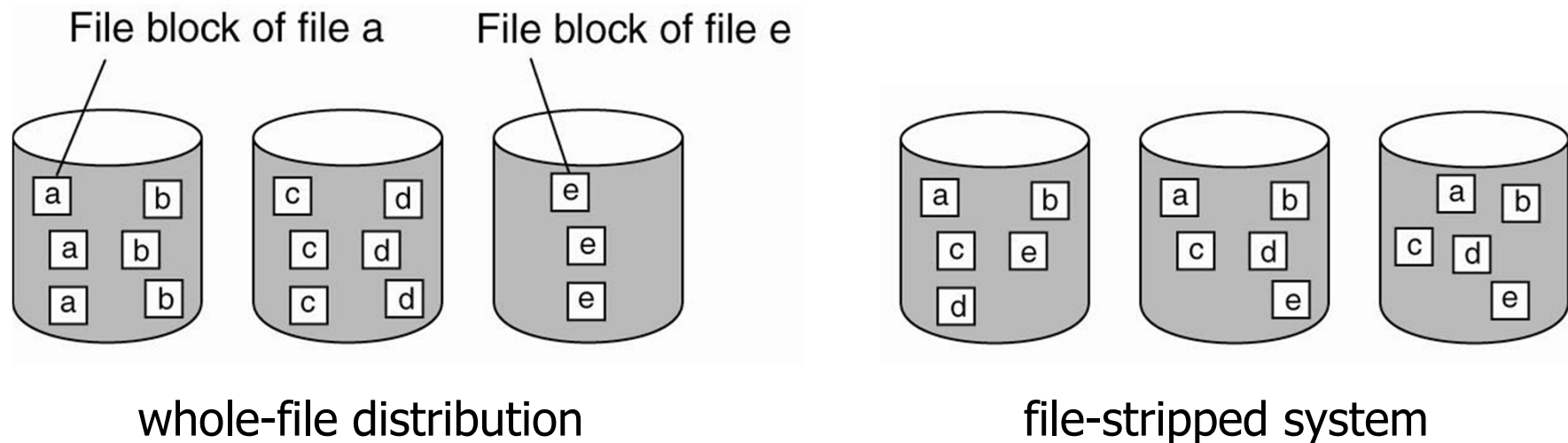


Contents

- Architecture
 - Client-server architectures
 - **Cluster-based architectures**
- Communication
- Naming
- Synchronization
- Consistency & replication
- Fault tolerance

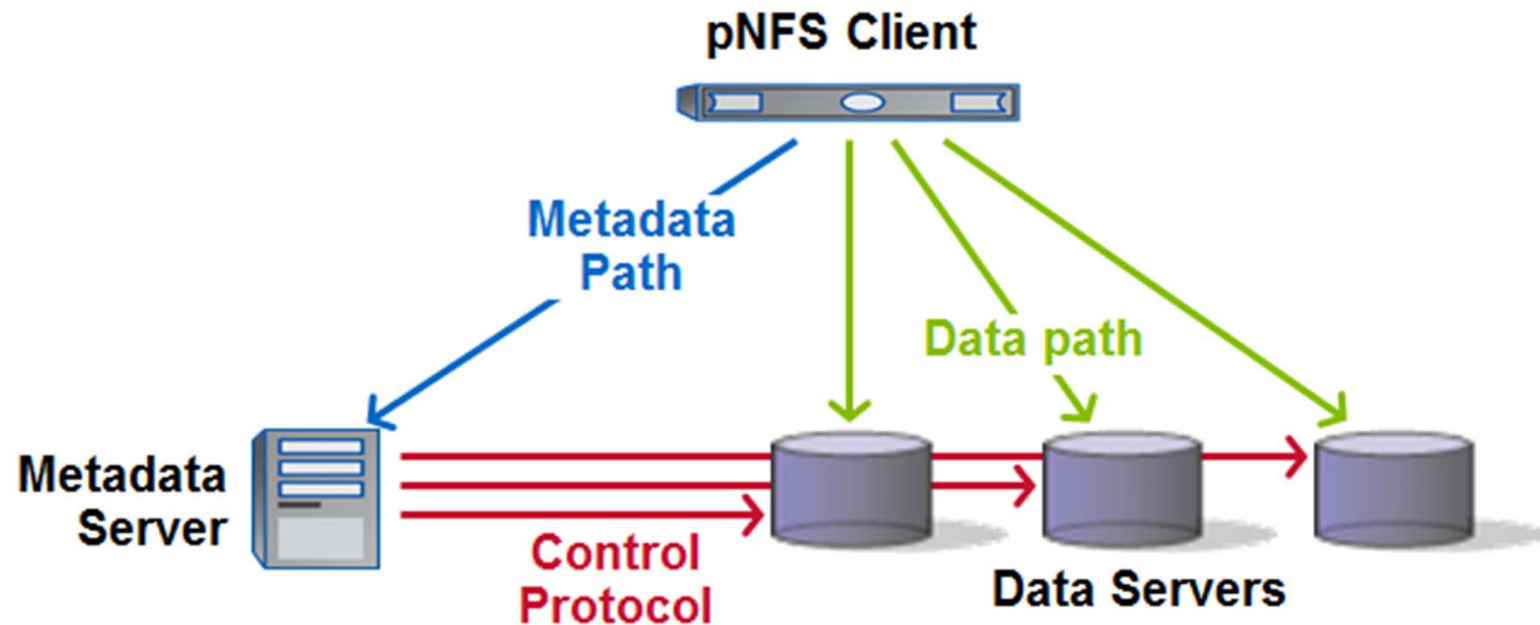
Cluster-based architectures

- How to improve performance in server clusters with very large data collections?
- **File-striping** techniques by which files can be fetched in parallel
 - Single file is distributed across multiple servers



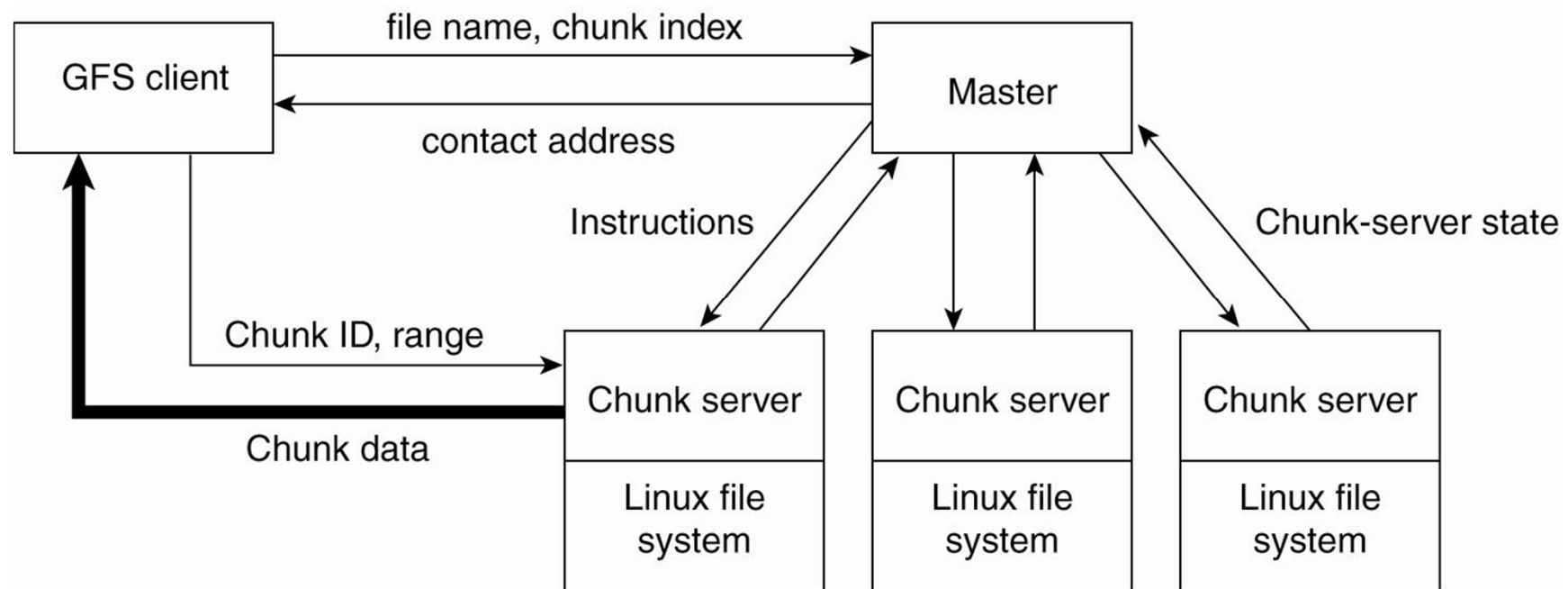
Cluster-based architectures

- Ex: pNFS (part of NFS v4.1)
 - Metadata and data separation
 - Metadata server is out of the data path
 - Files are striped across a number of data servers



Cluster-based architectures

- Ex: Google File System (GFS)
 - Each GFS cluster: 1 master, N chunk servers
 - Files divided into 64 MB chunks, which are replicated and distributed across chunk servers



Contents

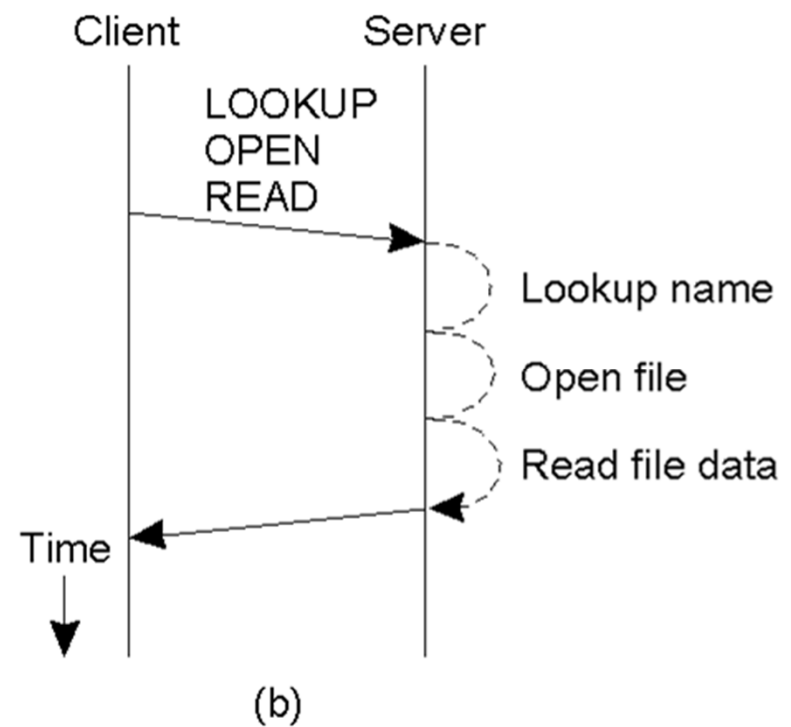
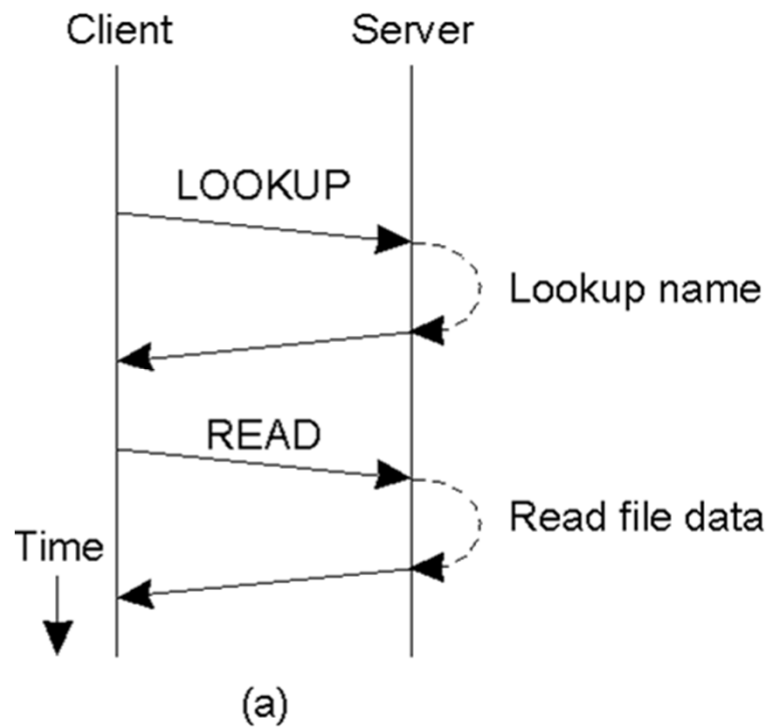
- Architecture
- **Communication**
- Naming
- Synchronization
- Consistency & replication
- Fault tolerance

NFS communication

- RPC-based communication
 - RPC hides the difference between different OS
 - Open Network Computing RPC (ONC RPC)
 - NFS v4 supports **compound procedures**
 - Group multiple RPCs into a single request
 - Handled in order as if sent separately
 - If one fails, the rest terminate
- ↑ Save latency of doing multiple RPCs
- Especially important in wide-area systems

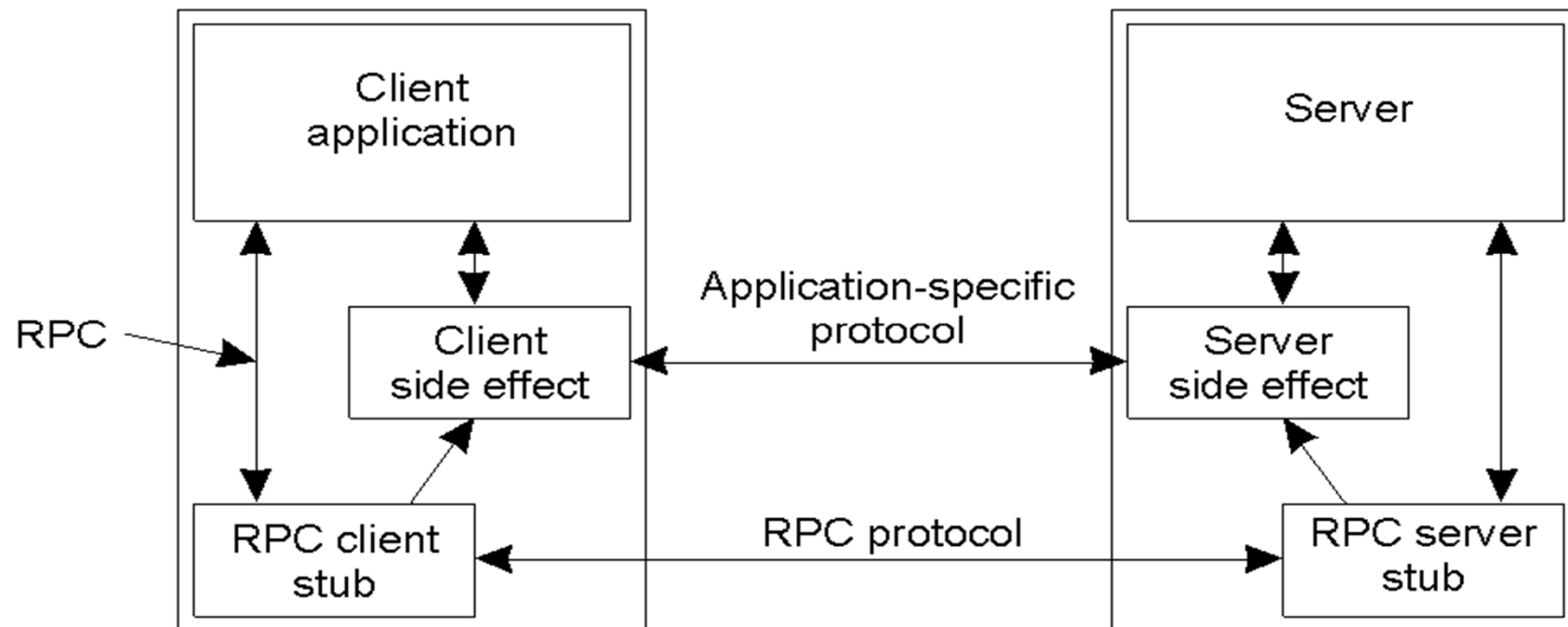
NFS communication

- a) One operation per RPC (NFS v1,2,3)
- b) Multiple operations per RPC (NFS v4)
 - Compound procedures



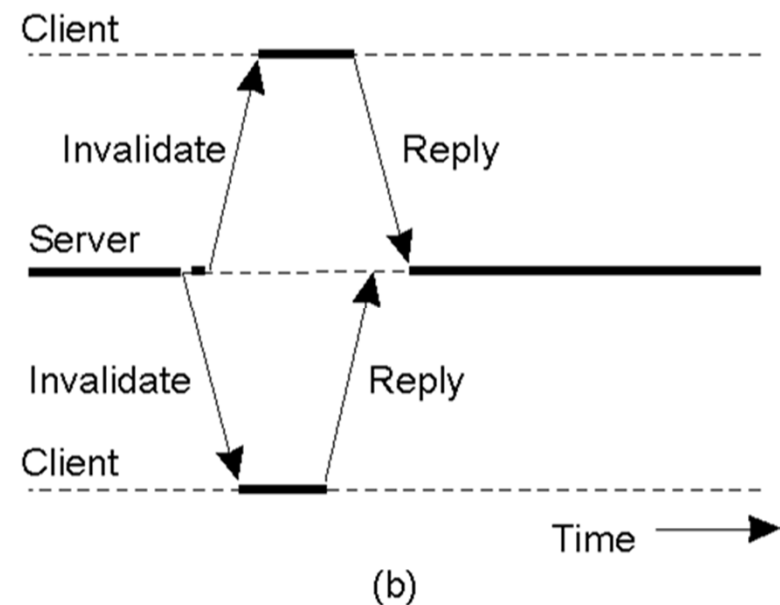
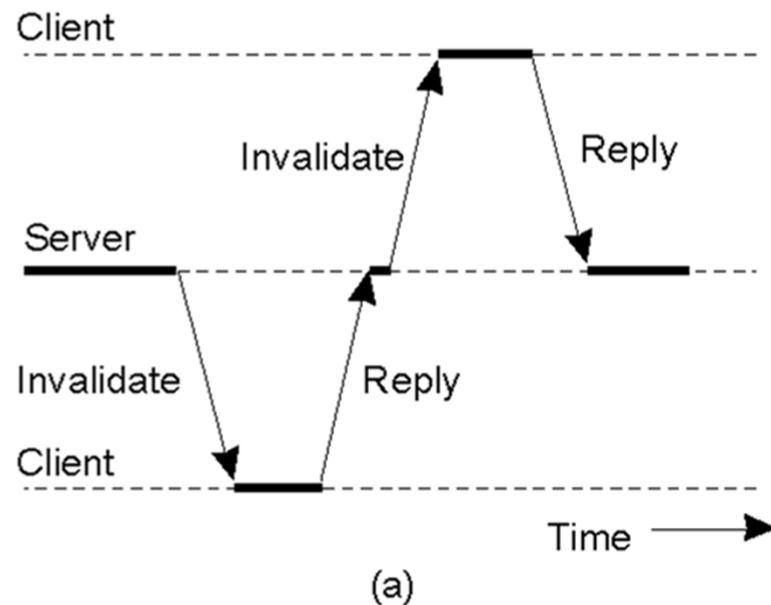
Coda communication

- Based on **RPC2**
 - Reliable transmission on top of UDP (unreliable)
 - Supports **side-effects**, i.e. user-defined protocols
 - e.g. isochronous mode for a video stream



Coda communication

- Support for multicast transparent to the client through **MultiRPC** (part of RPC2)
 - Send multiple RPCs in parallel (e.g. invalidations)



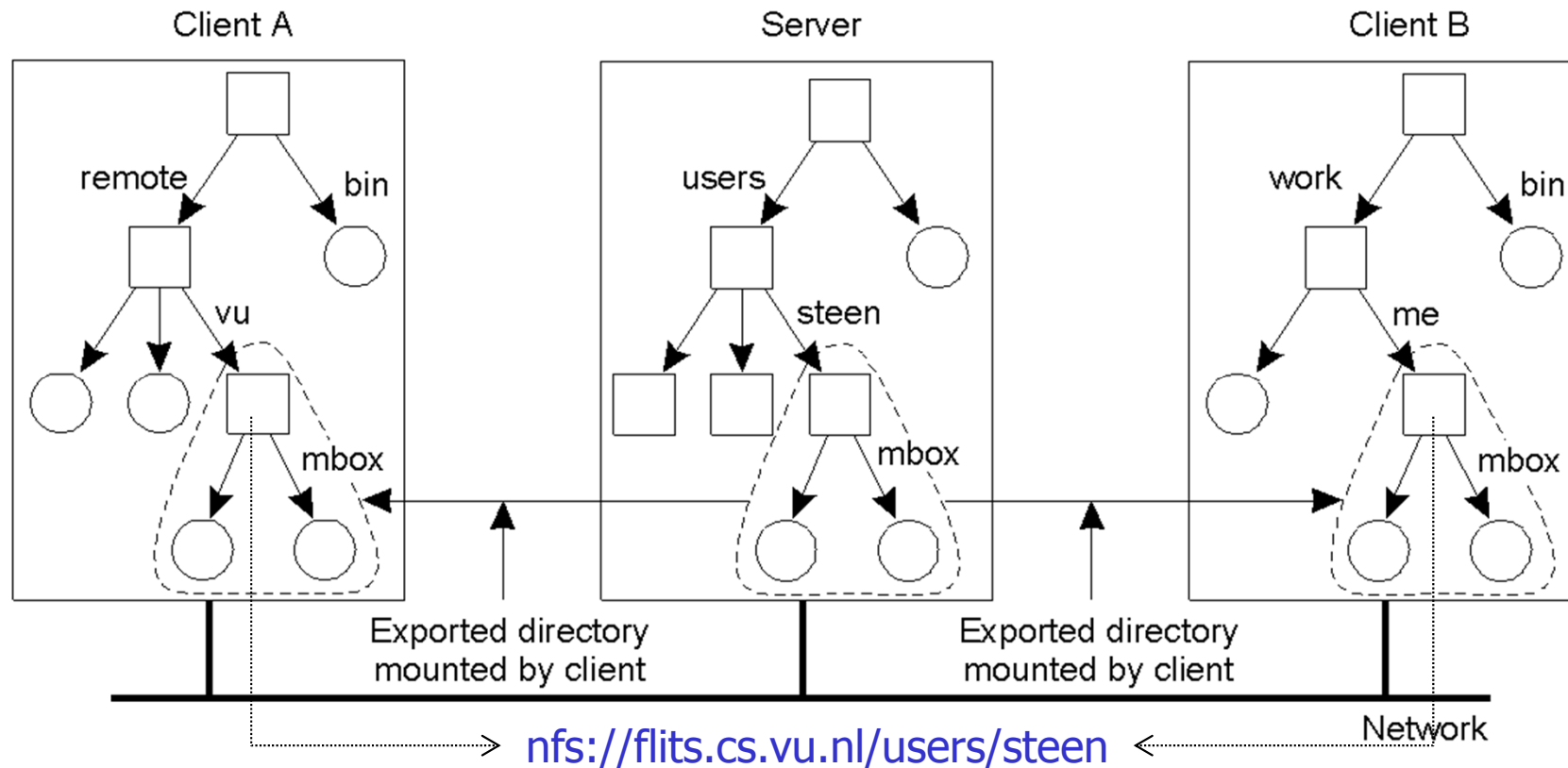
- a) Sending an invalidation message one at a time
- b) Sending invalidation messages in parallel

Contents

- Architecture
- Communication
- **Naming**
- Synchronization
- Consistency & replication
- Fault tolerance

NFS naming

- Transparent access to remote file system
 - By allowing a client to **mount** (part of) a remote file system into its own local file system

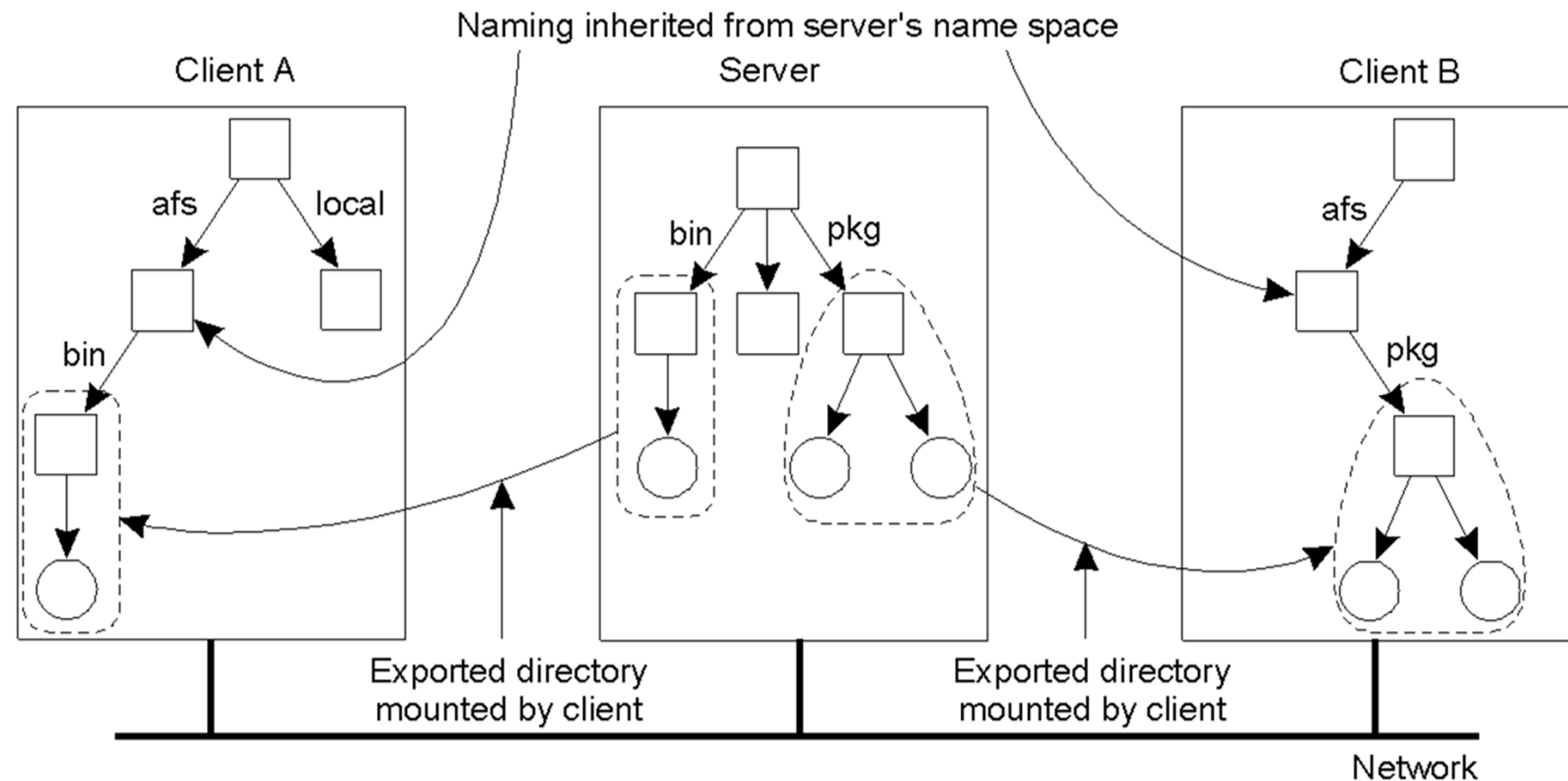


NFS naming

- Pathnames are not globally unique, as they depend on the clients' local name spaces
 - Referring to shared files is harder
 - Solution: Partly standardized name spaces
 - e.g. /usr/bin
- Name resolution in NFS v3 is iterative, NFS v4 also supports recursive name lookups
- File handle created by server hosting the file
 - Unique with respect to all its exported file systems
 - Does not change during file lifetime and it is not reused

Coda naming

- Single global shared name space at /afs
 - All clients see the same names (unlike NFS)



Coda naming

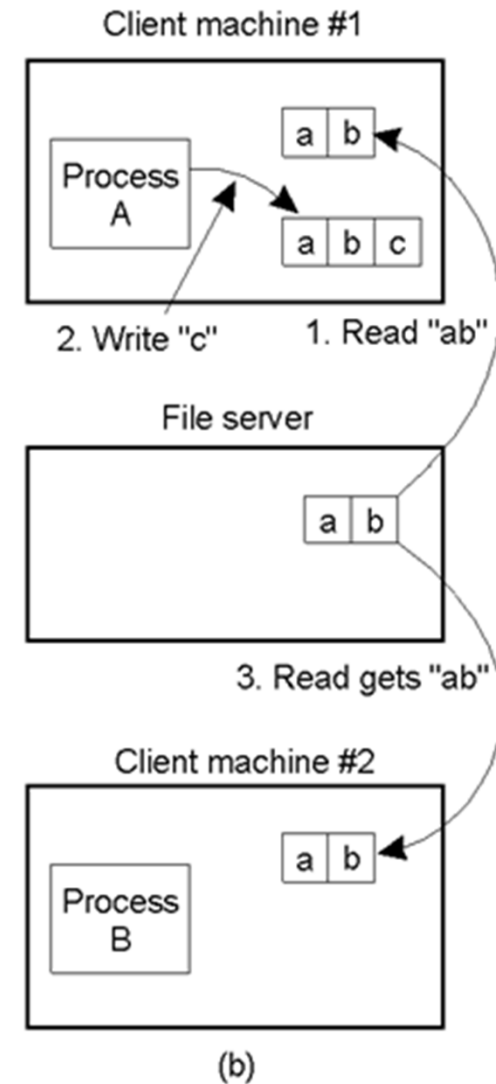
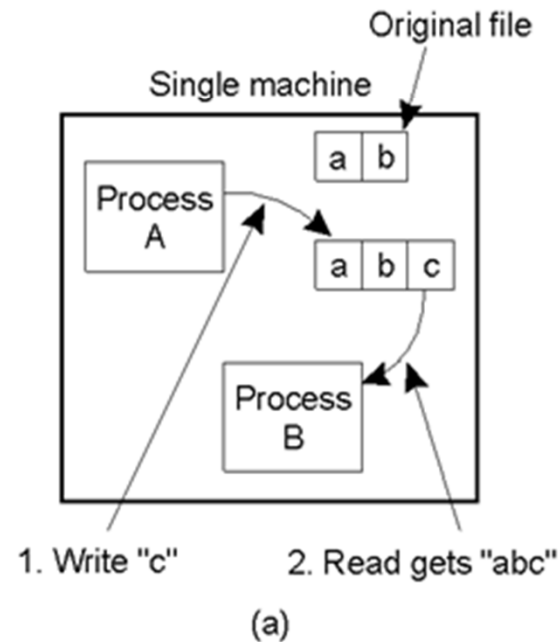
- File handles are globally unique
 - vs. server unique in NFS
- Files are grouped into volumes
 - Each file contained in exactly one volume
 - Volumes may be mounted
 - A volume is the mounting unit
 - Volumes may be replicated among several servers
 - A volume is the unit of server-side replication
 - Distinction between physical and logical volumes
 - A replicated volume consists of several physical volumes that are managed as one logical volume

Contents

- Architecture
- Communication
- Naming
- **Synchronization**
- Consistency & replication
- Fault tolerance

Semantics of file sharing

- a) In a single machine, when read follows write, value returned by read is value just written
- b) In distributed system with caching, stale values may be returned



Semantics of file sharing

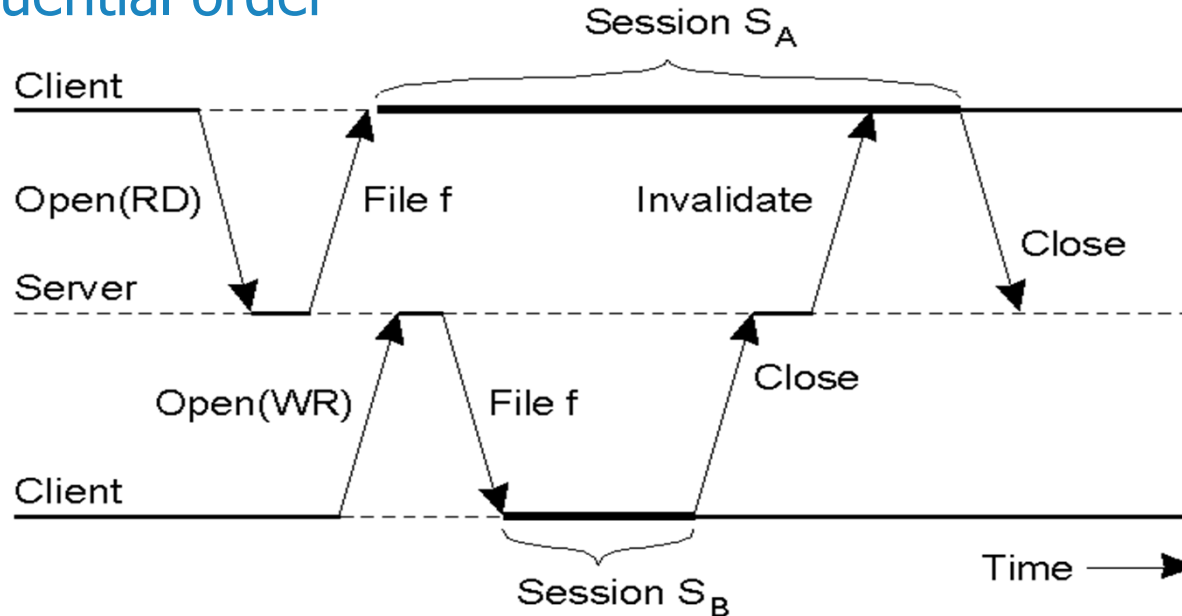
- 4 approaches for dealing with shared files in distributed systems

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 (BEGIN/END_TRANSACTION)

- NFS can use remote access model for providing UNIX semantics \Rightarrow performance problem
- Most implementations use local caches for performance implementing **session semantics**

Semantics of file sharing

- Coda checks file sharing conditions when it is opened
 - If file is opened for writing, no other clients may open it
 - Otherwise, other clients can open for reading or writing
- Coda treats sessions (open...close) like **transactions**
 - Concurrent sessions are considered to be scheduled in some sequential order



File locking

- NFS v1-v3: use a separate (stateful) lock manager
- NFS v4: integrated into NFS file access protocol
 - Distinguishes read locks from write locks
 - Locks granted for a specific time (i.e. lease)

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

- In addition, **share reservations** can be used
 - Implicit way to lock a file, independent from locks

File locking

- Result of open operation with share reservations
 - Client requests shared access given current denial state
 - Client requests denial state given current file access state

Current file denial state

(a)
**Request
access**

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

Requested file denial state

(b)
**Current
file
access
state**

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

Contents

- Architecture
- Communication
- Naming
- Synchronization
- **Consistency & replication**
- Fault tolerance

NFS client-side caching

- Preferred technique to attain performance
 - a) Data and metadata caching
 - Cache the results of read, write, getattr, lookup, and readdir operations
 - i.e. file data, file handles, attributes, directories
 - Carry out write operations locally
 - Modified blocks are marked as 'dirty'
 - **Closing** the file (or explicit sync) implies flushing dirty blocks to the server
 - Validate cached blocks when they are used using a timestamp-based procedure

NFS client-side caching

- Each cache block has two timestamps
 - T_c : time when cache block was last validated
 - T_m : time when block was last modified at the server
- A block is valid at time T if either:
 - i. $T - T_c < F$ (a.k.a. freshness interval)
 - Can be evaluated without accessing the server
 - F is typically 3-30s for files, 30-60s for directories
 - ii. $T_m \text{ (at server)} = T_m \text{ (at client)}$
 - $T_m \text{ (at server)}$ checked only if (i) is false
 - If (ii) is true, T_c is set to T
 - If (ii) is false, get fresh data from the server and set $T_m \text{ (at client)}$ and T_c

NFS client-side caching

- Close-to-open cache consistency
 - Extends the previous procedure to support the typical file sharing scenario
 - i.e. sequential file sharing: client A opens a file, writes something and closes it; then client B opens the same file and reads the changes
 - Client forces a cache validity check with the server when the file is opened
 - Ignoring any cache time remaining
 - On closing the file, pending changes are flushed to the server so that the next opener can view them

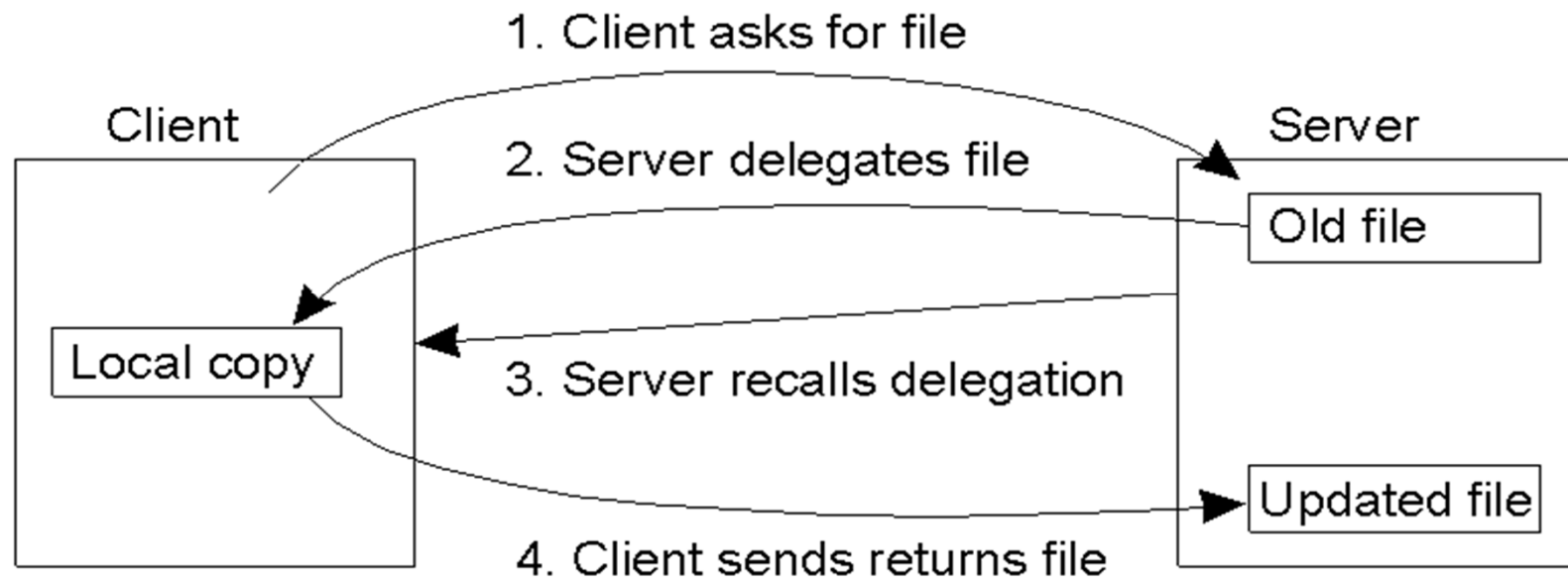
NFS client-side caching

b) NFS v4 supports **open delegation**

- Server voluntarily and temporarily transfers the control of operations on a file to the client
 - Decision is made by the server based on a set of conditions such as the recent history of the file
- During the duration of the delegation, client can treat the file as if no other conflicting accesses are performed by other clients
 - Client can locally service operations such as open, close, or lock without interaction with the server
 - Client can access the file in the cache without sending validation requests to the server

NFS client-side caching

- Delegations can be recalled by the server using callbacks (i.e. RPC to the client)
 - When another client requests access to the file that conflicts with the granted delegation
 - They have also a lease that is subject to renewal



NFS client-side caching

- Read delegation
 - Awarded on a file opened for reading (not denying read access to others)
 - Multiple read delegations to different clients can be outstanding simultaneously
 - Allows the client to handle locally requests to:
 - Open for reading (not denying read access to others)
 - Read from the cache without checking validity
 - Requests to open the file for writing or to lock the file must still be sent to the server
 - Recalled when another client requests to open the file for writing

NFS client-side caching

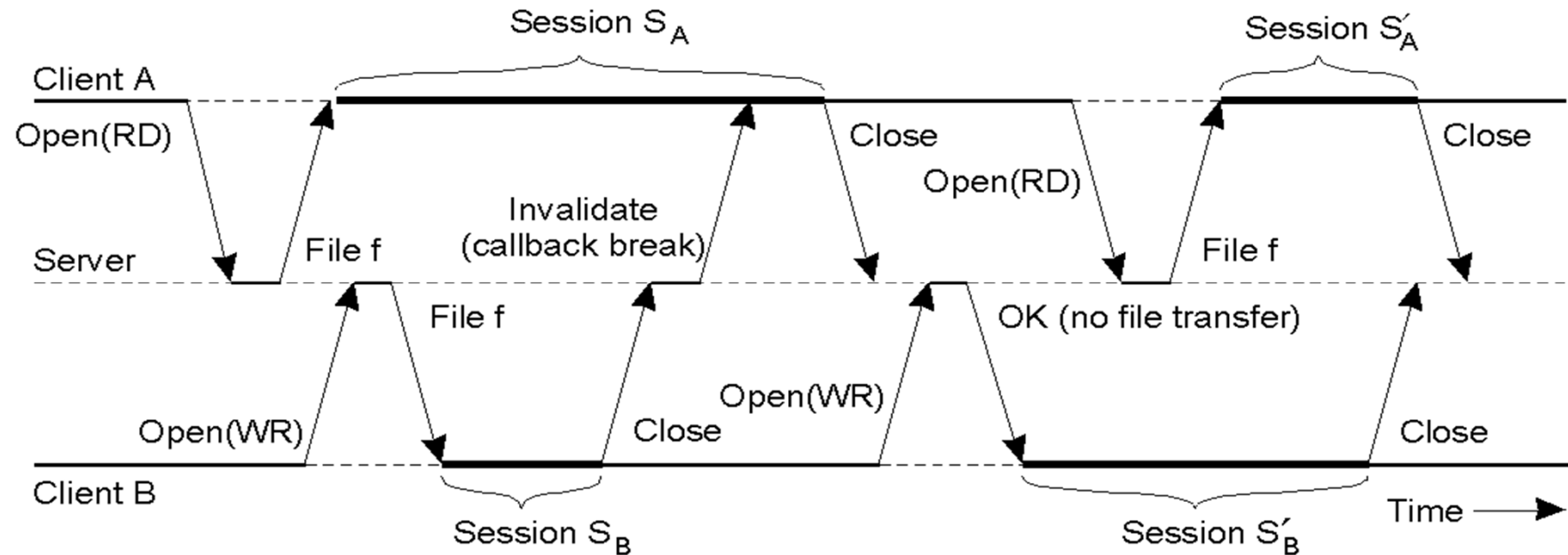
- Write delegation
 - Awarded on a file opened for writing (or r/w)
 - Only one write delegation may exist for a given file at a time, and it is inconsistent with any read delegations
 - While the delegation is outstanding, all the operations in the file can be handled locally
 - Recalled when another client requests to open the file (independently on the requested access)
 - On returning the delegation, the client will commit all dirty data to the server

Coda client-side caching

- Coda creates a local copy of the file when it is opened (entire file is cached)
 - All read/write operations done on the local copy
 - Updates are sent back to server when the file is closed
- Consistency maintained using **callbacks**
 - On opening a file, client gets a callback promise for the file from the server
 - When another client updates the file, the server breaks the promise (it can also expire by timeout)
 - Server tracks all clients that have a copy of the file

Coda client-side caching

- If the promise is valid, a cached file can be re-opened without referring to the server
- If the promise has been canceled, re-opening implies fetching a fresh copy of the file

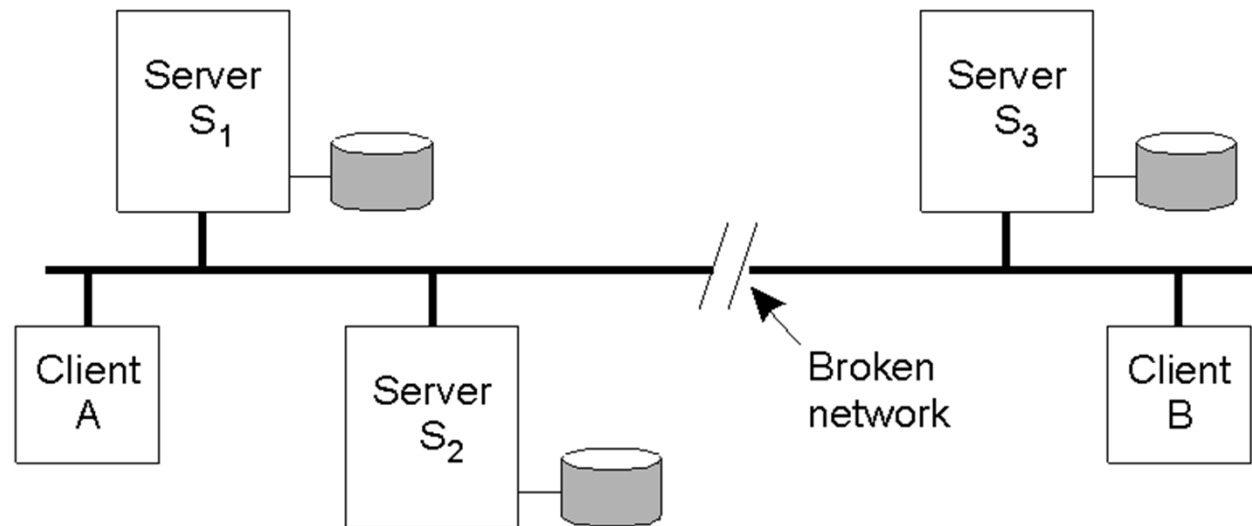


Coda server-side replication

- This is generally used for fault tolerance
- VSG: Volume Storage Group
 - Collection of servers that have a copy of a volume
- AVSG: Accessible VSG
 - Servers in the VSG that the client can contact
 - If AVSG is empty, the client is disconnected
- Replicated-write protocol to keep consistency
 - Variant of ROWA: Read-One Write-All
 - Client opens a file: contact one server in the AVSG
 - Client closes an updated file: multicast file to all the servers in the AVSG using MultiRPC

Coda server-side replication

- What happens in the presence of partitions?
 - Possible inconsistency: use versioning to detect
 - Versioning vector when partition happens: $[1,1,1]$
 - A updates file, versioning vector in its partition: $[2,2,1]$
 - B updates file, versioning vector in its partition: $[1,1,2]$
 - Partition repaired, compare versioning vectors: **conflict!**



Contents

- Architecture
- Communication
- Naming
- Synchronization
- Consistency & replication
- **Fault tolerance**

NFS fault tolerance

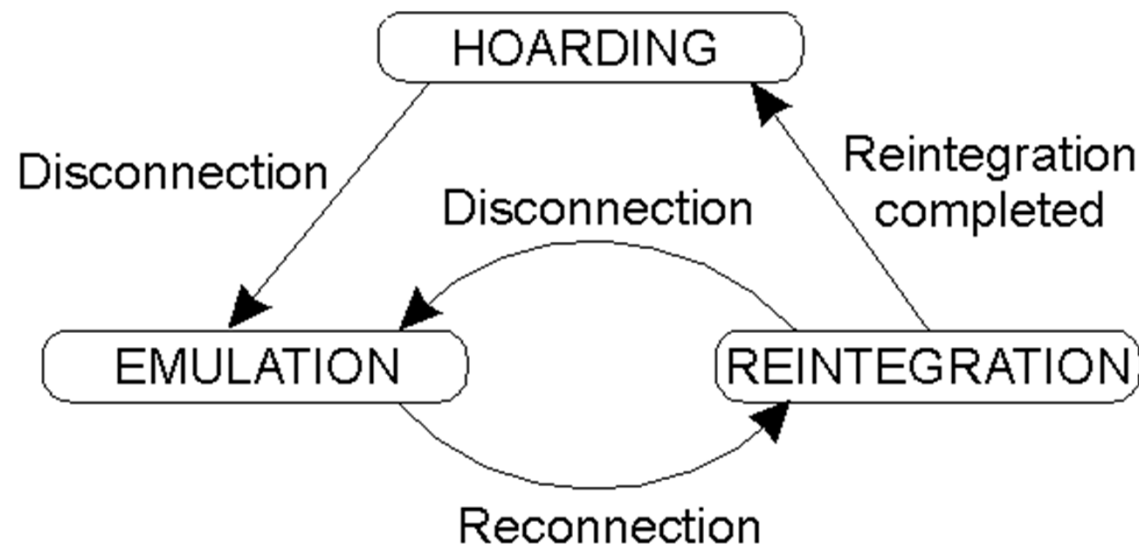
- Communication failures
 - Most operations are idempotent: can be retried until receiving a reply (at-least-once semantics)
 - Number requests to deal with duplicated non-idempotent operations (as discussed in Lesson 2)
- Client failures (addressed with leases)
 - Server can recover client's locks and delegations when the associated leases have expired
 - Delegations may need to be reclaimed after client restarts to reestablish the file state on the server
 - Client may have unsaved file data stored locally

NFS fault tolerance

- Server failure (addressed with grace periods)
 - When the server restarts, it initiates a grace period of length equal to the lease period
 - During the grace period, clients and server work to reestablish the server state that existed prior to the failure
 - Clients send reclaim-type open and lock requests
 - Corresponding to the locks and open files they had prior to the failure
 - Server rejects new open and lock requests
 - Unless it can reliably determine that granting any such request cannot conflict with a subsequent reclaim

Coda fault tolerance

- Disconnected operation \Rightarrow high availability
 - HOARDING: Cache in advance all files that may be accessed when disconnected
 - EMULATION: When disconnected, behavior of server emulated at client (locally cached files allow operation across failure)
 - REINTEGRATION: transfer updates to server; resolve conflicts



Summary

Issue	NFS	Coda
Design goals	Access transparency	High availability
Access model	Remote	Upload/Download
Communication	RPC (ONC RPC)	RPC (RPC2)
Client process	Thin/Fat	Fat
Server groups	No	Yes
Mount granularity	Directory	File system (volume)
Name space	Per client	Global
File ID scope	File server	Global
Sharing semantics	Session	Transactional
Cache consistency	Write-back	Write-back
Replication	Minimal	ROWA
Fault tolerance	Reliable communication	Replication & caching
Recovery	Client-based	Reintegration

Summary

- Further details:
 - [Tanenbaum]: chapter 11
 - [Coulouris]: chapters 12, 18.4.3, and 21.5.1