Last class: Distributed File Systems

- Issues in distributed file systems
- Sun’s Network File System case study

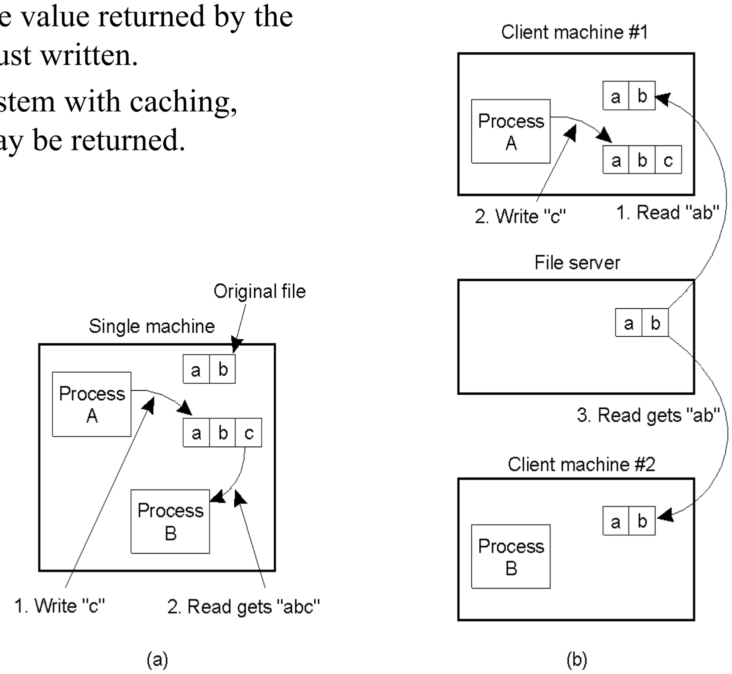
Today: NFS, Coda

- Case Study: NFS (continued)
- Case Study: Coda File System

Semantics of File Sharing

a) On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.

b) In a distributed system with caching, obsolete values may be returned.



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
Transaction	All changes occur atomically

- Four ways of dealing with the shared files in a distributed system.
  - NFS implements session semantics
    - Can use remote/access model for providing UNIX semantics (expensive)
    - Most implementations use local caches for performance and provide session semantics

File Locking in NFS

Operation	Description
Lock	Creates a lock for a range of bytes (non-blocking_
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 supports file locking
    - Applications can use locks to ensure consistency
    - Locking was not part of NFS until version 3
    - NFS v4 supports locking as part of the protocol (see above table)

File Locking: Share Reservations

Current file denial state

Request access	NONE	READ	WRITE	BOTH
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(a)

Requested file denial state

Current access state	NONE	READ	WRITE	BOTH
READ	Succeed	Fail	Succeed	Fail
WRITE	Succeed	Succeed	Fail	Fail
BOTH	Succeed	Fail	Fail	Fail

(b)

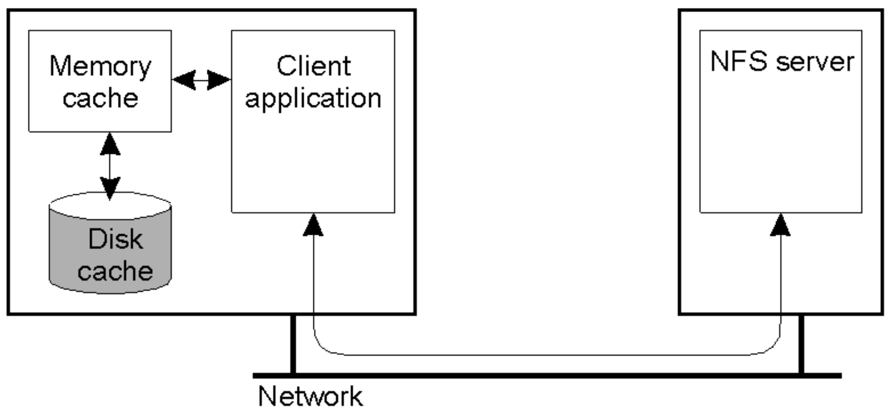
- The result of an *open* operation with share reservations in NFS.

a) When the client requests shared access given the current denial state.

b) When the client requests a denial state given the current file access state.

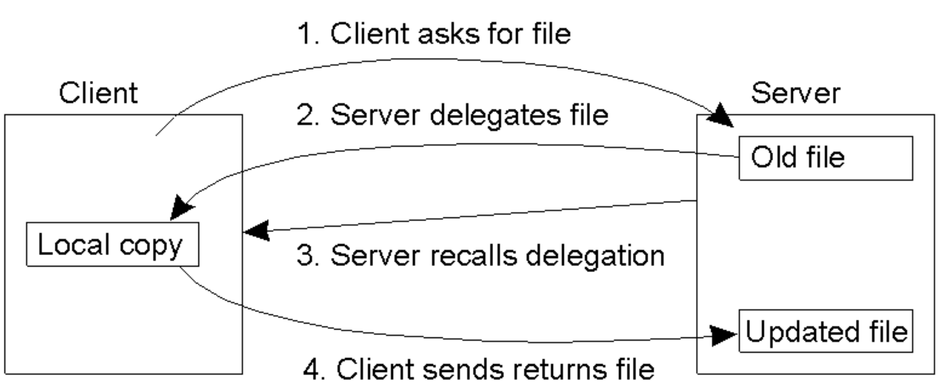
Client Caching

- Client-side caching is left to the implementation (NFS does not prohibit it)
  - Different implementation use different caching policies
    - Sun: allow cache data to be stale for up to 30 seconds

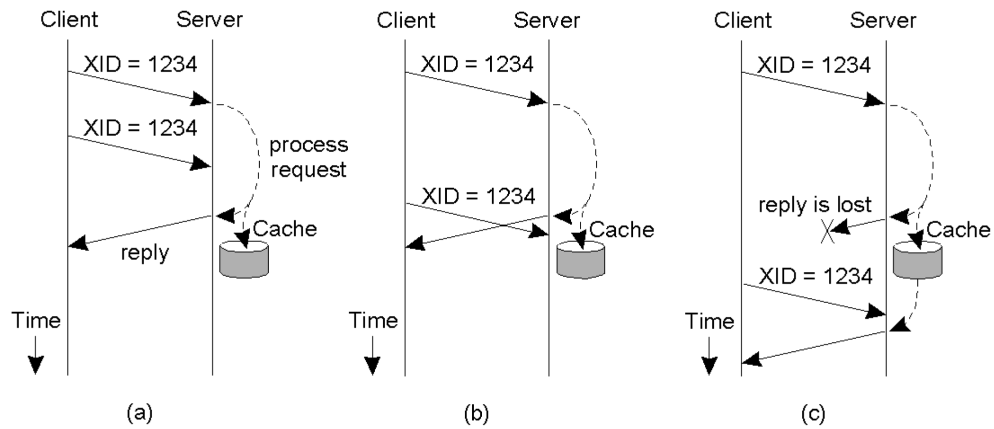


Client Caching: Delegation

- NFS V4 supports open delegation
  - Server delegates local open and close requests to the NFS client
  - Uses a callback mechanism to recall file delegation.



RPC Failures



- Three situations for handling retransmissions: use a duplicate request cache

a) The request is still in progress

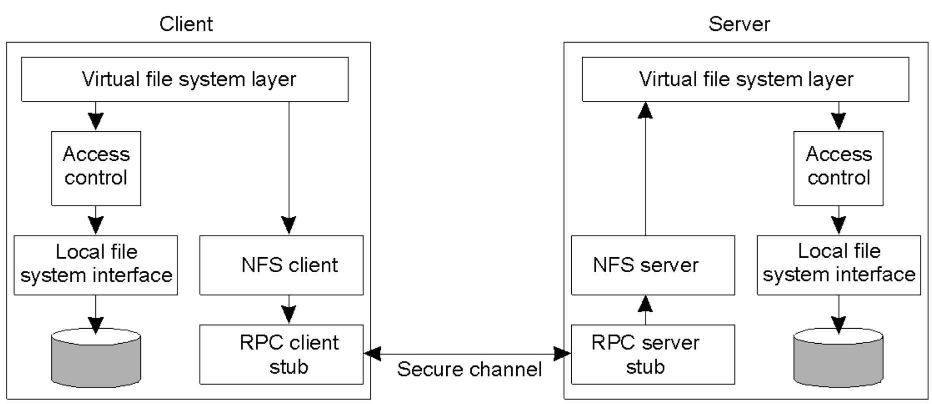
b) The reply has just been returned

c) The reply has been some time ago, but was lost.

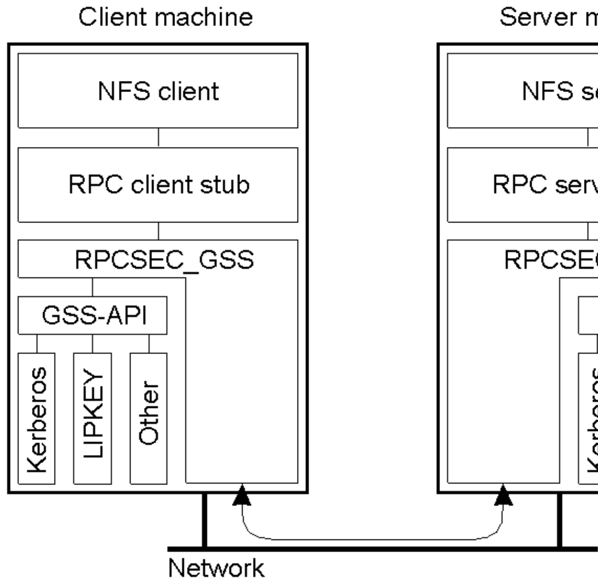
Use a duplicate-request cache: transaction Ids on RPCs, results cached

Security

- The NFS security architecture.
  - Simplest case: user ID, group ID authentication only



Secure RPCs



- Secure RPC in NFS version 4.

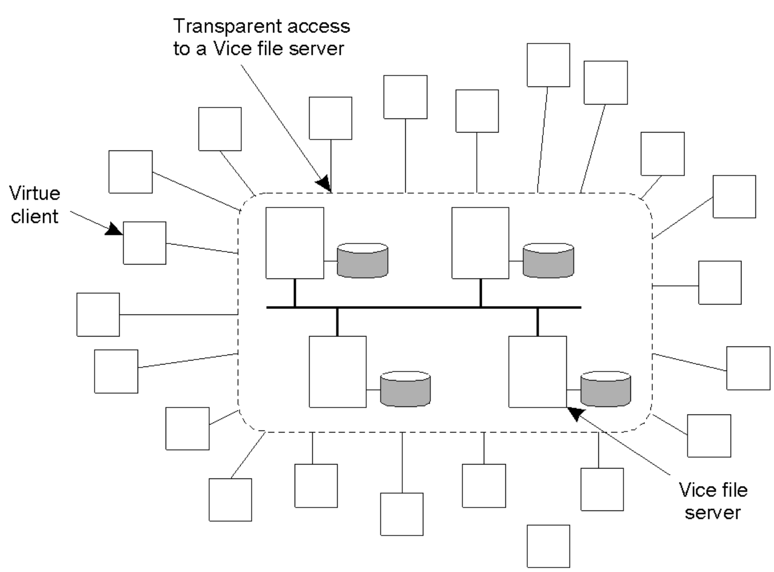
Replica Servers

- NFS ver 4 supports replications
- Entire file systems must be replicated
- FS_LOCATION attribute for each file
- Replicated servers: implementation specific

Coda

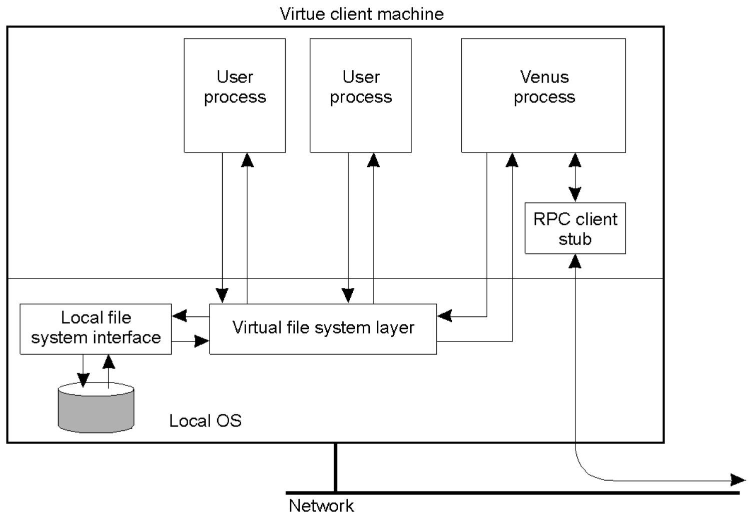
- Coda: descendent of the Andrew file system at CMU
  – Andrew designed to serve a large (global community)
- Salient features:
  – Support for disconnected operations
    - Desirable for mobile users
  – Support for a large number of users

Overview of Coda



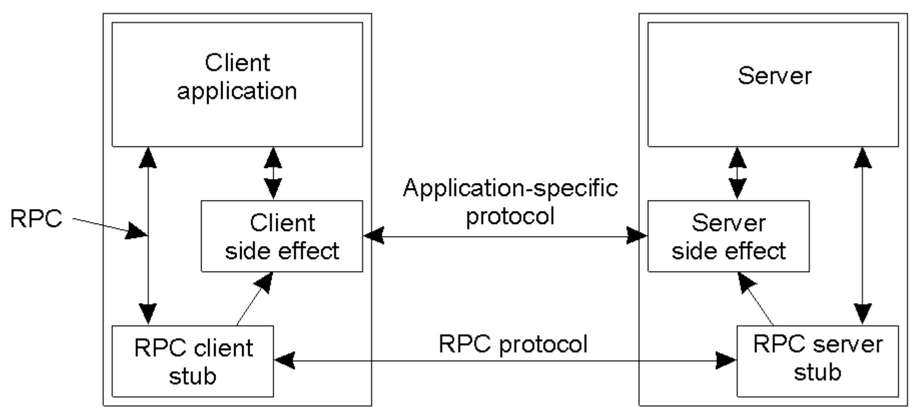
- Centrally administered Vice file servers
- Large number of virtue clients

Virtue: Coda Clients



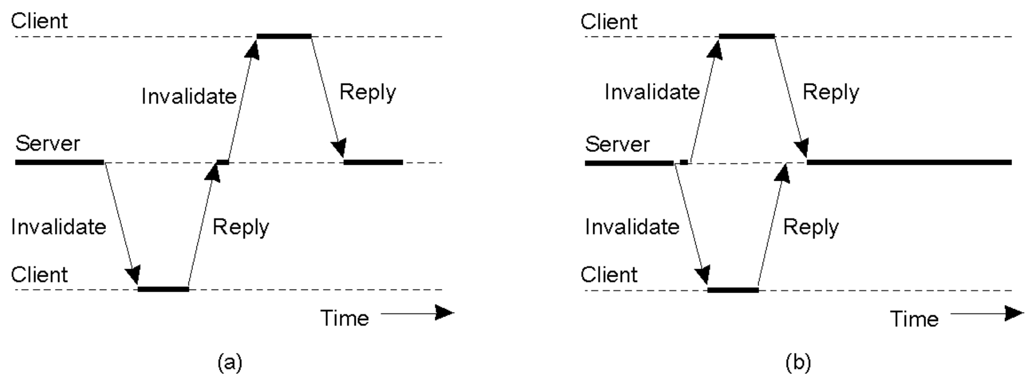
- The internal organization of a Virtue workstation.
  - Designed to allow access to files even if server is unavailable
  - Uses VFS and appears like a traditional Unix file system

Communication in Coda



- Coda uses RPC2: a sophisticated *reliable* RPC system
  - Start a new thread for each request, server periodically informs client it is still working on the request
- RPC2 supports *side-effects:* application-specific protocols
  - Useful for video streaming [where RPCs are less useful]
- RPC2 also has multicast support

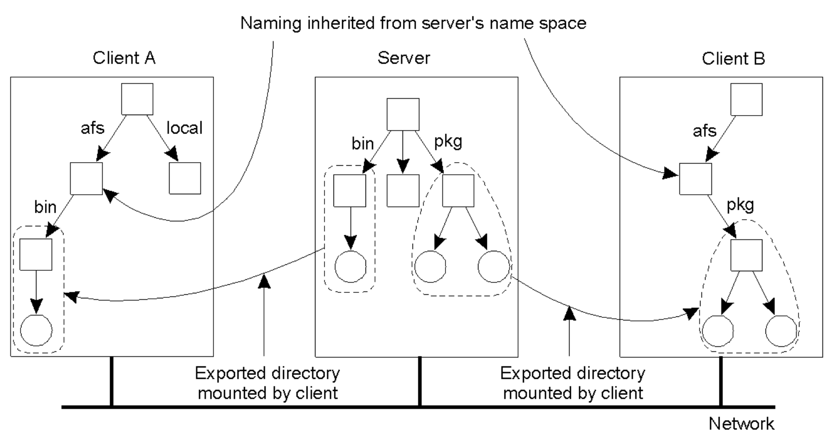
Communication: Invalidations



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

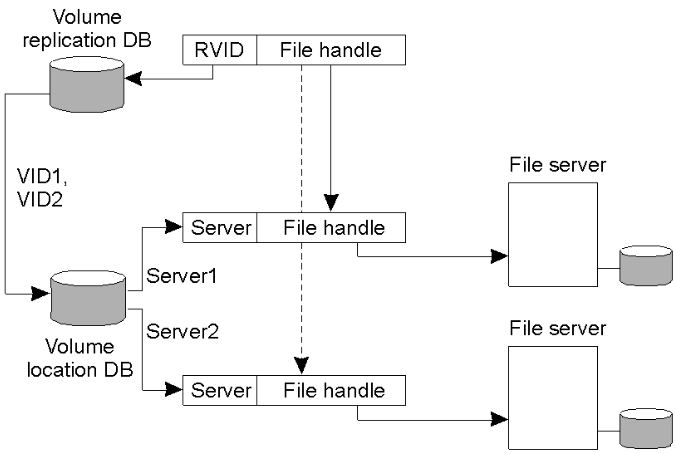
Can use MultiRPCs [Parallel RPCs] or use Multicast
- Fully transparent to the caller and callee [looks like normal RPC]

Naming



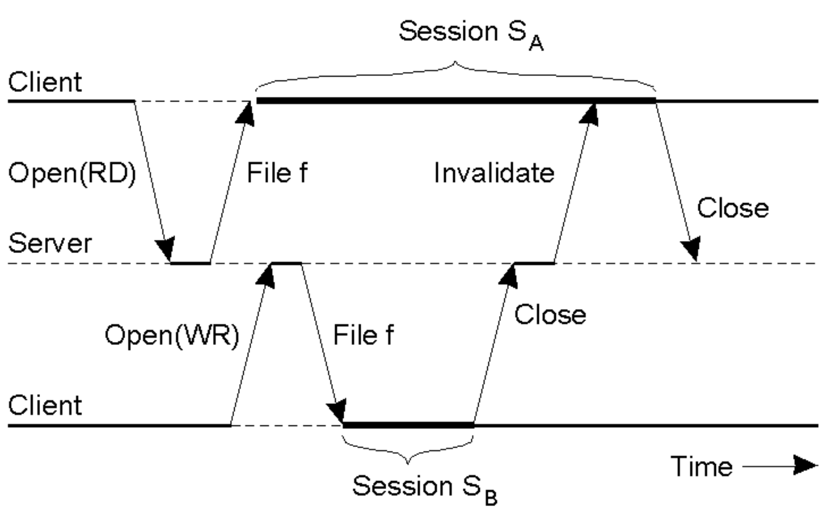
- Clients in Coda have access to a single shared name space
- Files are grouped into *volumes* [partial subtree in the directory structure]
  - Volume is the basic unit of mounting
  - Namespace: /afs/filesrv.cs.umass.edu [same namespace on all client; different from NFS]
  - Name lookup can cross mount points: support for detecting crossing and automounts

File Identifiers



- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical (replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

Sharing Files in Coda



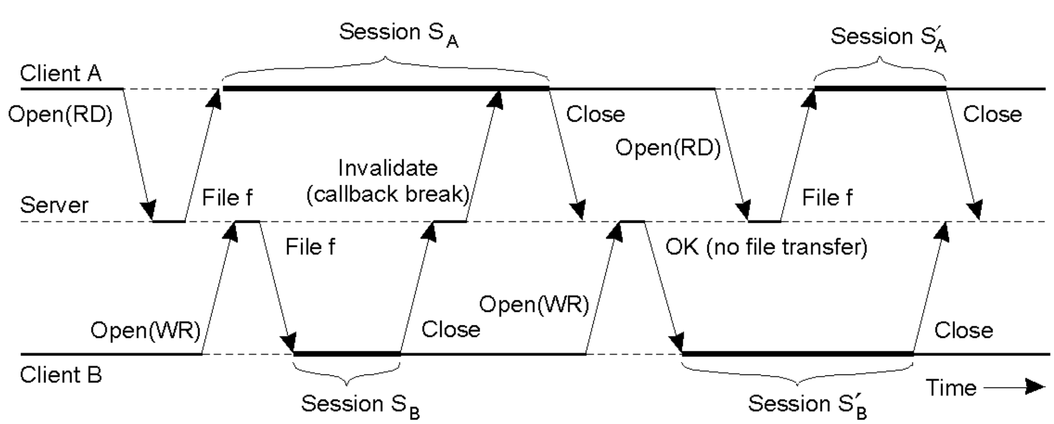
- Transactional behavior for sharing files: similar to share reservations in NFS
  - File open: transfer entire file to client machine [similar to delegation]
  - Uses session semantics: each session is like a transaction
    - Updates are sent back to the server only when the file is closed

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

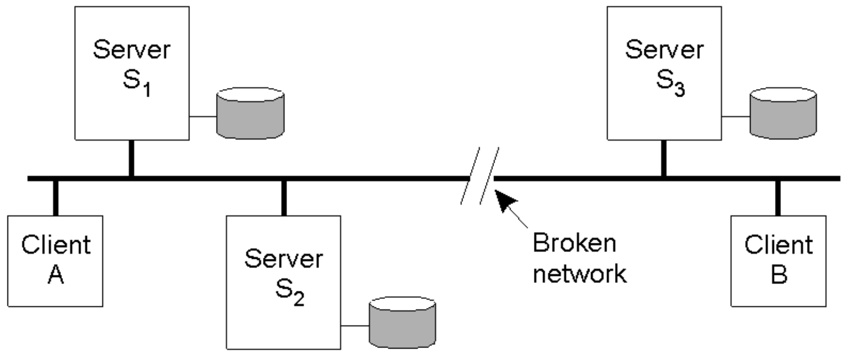
- Network partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    - Ensure that operations were serializable *after thay have executed*
  - Conflict => force manual reconciliation

Client Caching



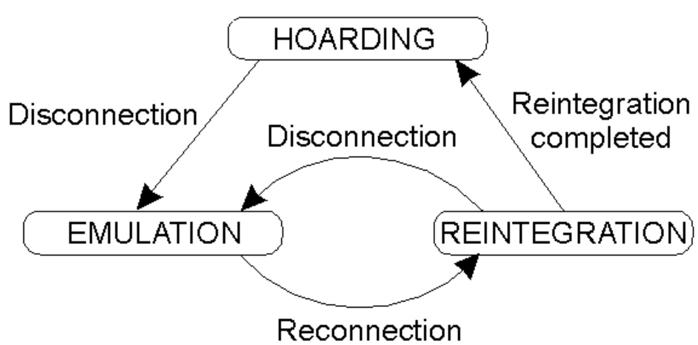
- Cache consistency maintained using callbacks
  - Server tracks all clients that have a copy of the file [provide *callback promise*]
  - Upon modification: send invalidate to clients

Server Replication



- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG (all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2,2,1] and [1,1,2] is a conflict => manual reconciliation

Disconnected Operation



- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection
  - Prefetch all files that may be accessed and cache (hoard) locally
  - If AVSG=0, go to emulation mode and reintegrate upon reconnection