Ch. 7 Distributed File Systems

File service architecture
Network File System
Coda file system

Tanenbaum, van Steen: Ch 10
CoDoKi: Ch 8

21-Apr-06 1

File Systems

- Traditional tasks of a FS
  - organizing, storing, accessing of data
  - naming
  - sharing
  - protection
- Requirements
  - reliability, persistence
  - scalability
- Distributed systems: requirements
  - old requirements: increased importance
  - various transparencies
  - consistency
  - fault tolerance
  - performance
  - mobility support

21-Apr-06 2

Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache / replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore

Figure 8.1 1: strict one-copy, ✓ weaker, 2: essentially weaker, ✗ user actions

21-Apr-06 3

File service architecture (1)

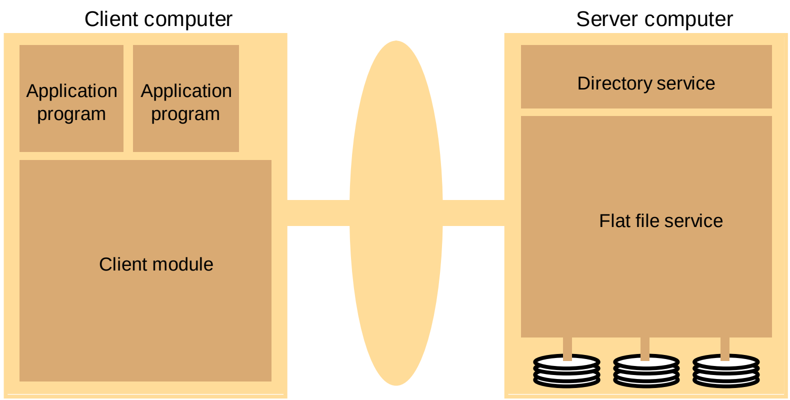
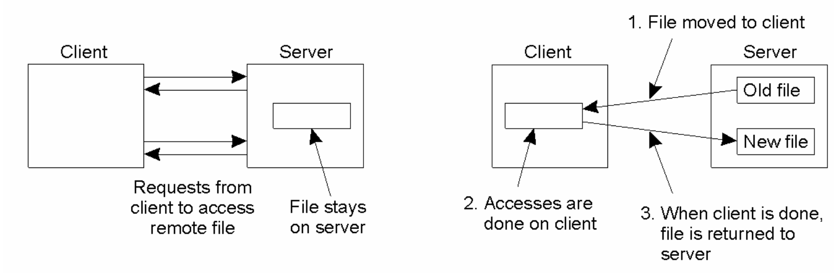


Figure 8.5

21-Apr-06 4

File Service Architecture (2)



The remote access model (e.g., NFS)

The upload/download model (e.g., Coda)

Distributed FS: Requirements (1)

Transparency

- Access
  - local/remote
  - heterogeneity of local servers
- Location
  - uniform name space
  - relocation of file(group)s without an effect on pathnames
  - name space independent of user location
- Mobility
  - client-node tables independent of file movings
- Performance
  - server load
- Scaling

DFS: Requirements (2)

- Concurrency control of file updates
  (file/record-level locking)
- File replication
  => availability, fault tolerance
  => performance, scalability

HW/SW heterogeneity

Fault tolerance
- causes: node failures, communication failures
- tolerance through replication of
  - data (=> consistency problems)
  - operations (idempotent ops or duplicate-operation problems)
- tolerance through stateless servers

21-Apr-06 7

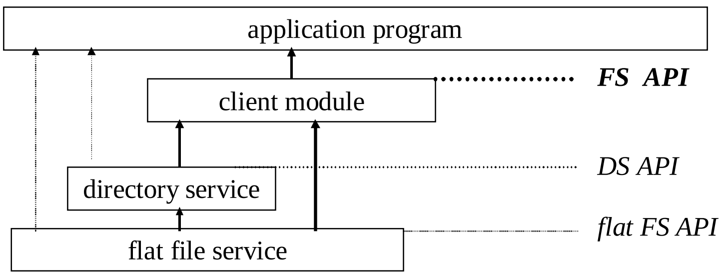
DFS: Requirements (3)

- Consistency (of replicates)
  - unix semantics: *one-copy-semantics*
    (update with immediate effect)
  - session semantics: update after closing the file
    - transaction (all at the same time)
    - lazy update (update propagation as a background activity)
  - see: Tanenbaum, Ch 6
- Security
  - authentication (each message!)
  - access control; protection of message contents
  - means: passwords, digital signatures, capabilities, encryption of data
- Efficiency
  - comparable to a local filestore

21-Apr-06 8

FS Architecture (1)

- Architecture: division of responsibilities => separation of concerns, modularity, openness (CoDoKi, Fig. 8.5)



21-Apr-06 9

FS Architecture (2)

- Flat file service
  - operations on file contents
  - unique file identifiers
- Directory service: text name => UFID
- Client module
  - runs in the client node
  - provides a unified local API
  - responsibilities:
    - implements operations not provided by the flat file service
    - takes care of remote operation invocations
      - management of data communication
      - control of data communication

21-Apr-06 10

Service Interfaces

- **Flat file service**
  - Operations of the model flat file service: CoDoKi, Fig. 8.6 (compare with UNIX!)
  - Reasons for differences
    - repeatable (*idempotent*) operations
    - stateless servers
    => easier implementation of fault tolerance
- **Directory service**
  - Operations of the model directory service: Fig. 8.7

Flat file service operations

Read(FileId, i, n) -> Data — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data.*
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in).

Figure 8.6

Directory service operations

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

Figure 8.7

21-Apr-06 13

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**.

21-Apr-06 14

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

Fig. 10-9 (a) Some general mandatory file attributes in **NFS**.

21-Apr-06 15

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

Fig. 10-9 (b) Some general recommended file attributes in **NFS**.

21-Apr-06 16

File Grouping

- *File group:* a collection of files located on a given server
- Groups can be moved between servers
  (origins: removable disk cartridges)
- Use: managerial purposes (capacity allocation, …)
- Naming of files: {File group ID, File UFID}
  (textual name => file group => file server => file)
- File group ID's must be globally unique
  (e.g., *creating server*'s IP-address, creation timestamp)

21-Apr-06 17

Sun Network File System (1)

- A long history …
  - 2^{nd} version in mid 80's
  - 3^{rd} version beginning of 90's
  - 4^{th} version beginning 2000's
- Architecture: CoDoKi, Fig. 8.8;
  the server is (almost) stateless
- Virtual file system
  - access transparency:
    - UNIX API <=> "any" fs
    - local or remote
  - location transparency
  - administration: available file services
  - communication service (RPC; TCP or UDP)

21-Apr-06 18

NFS architecture

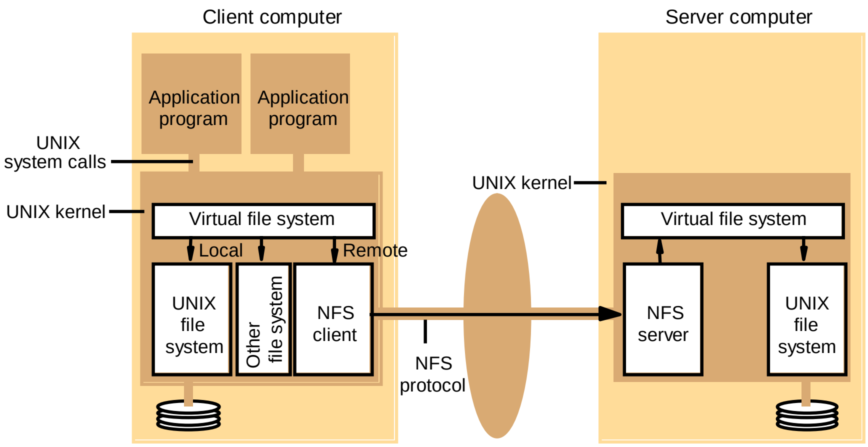
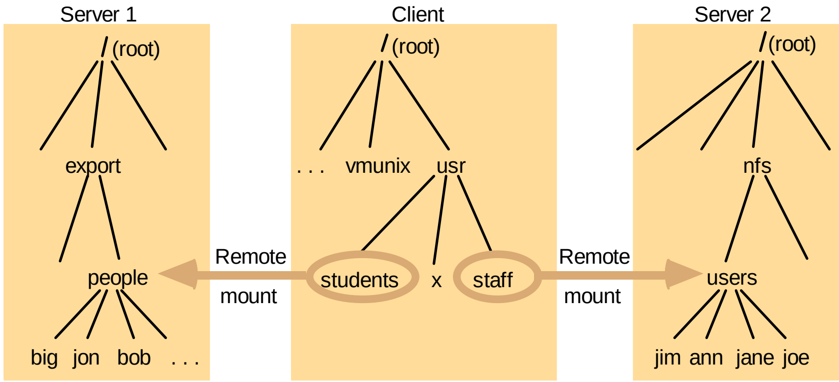


Figure 8.8

Sun Network File System (2)

- Client integration
  - "the client module" in the CoDoKi model
  - emulates standard UNIX FS semantics
  - *integrated* with the UNIX kernel, benefits (wrt. routine library):
    - a single client module serves all user-level processes, with a shared cache
    - the encryption key used to authenticate user IDs passed to the server can be retained in the kernel
- Access control and authentication (NFS v.3)
  - a stateless server: all operations independent
  - all operations authenticated (within the Sun RPC)
  - Kerberos integrated

Local and remote file systems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Figure 8.10

21-Apr-06 21

NFS Architecture

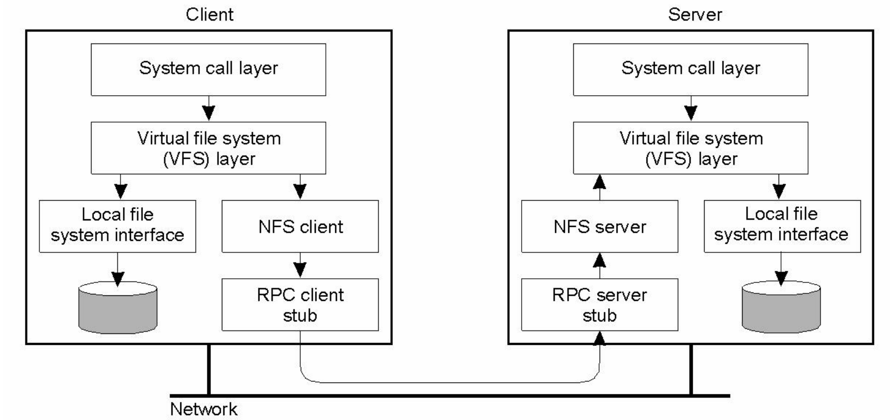
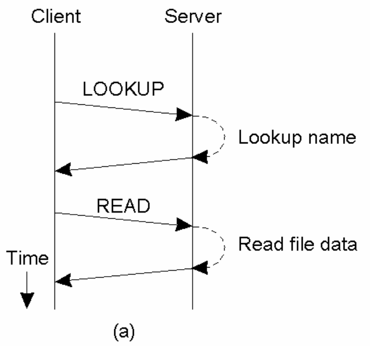


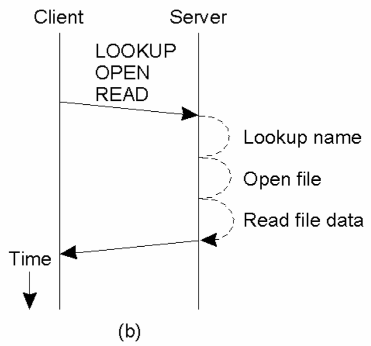
Fig. 10-2. The basic NFS architecture for UNIX systems.

21-Apr-06 22

Communication



Reading data from a file in NFS version 3.



Reading data using a compound procedure in version 4.

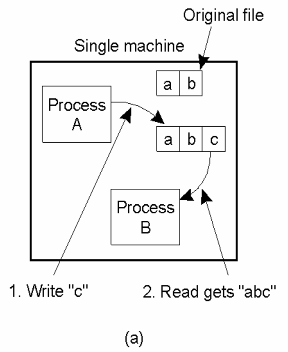
Fig. 10-4. Operations: Open Network Computing RPC

21-Apr-06 23

Semantics of File Sharing (1)

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

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



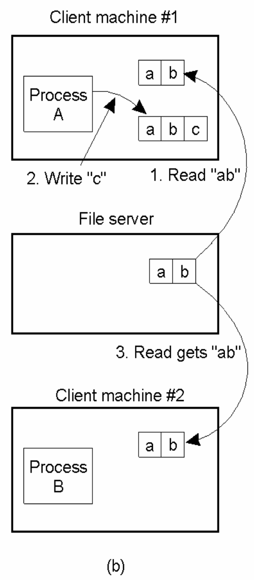


Fig. 10-10

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

Immutable files

- ”write” => create a new file under the old name (directory update)
- problem solved: read-write conflict
- problem created: two concurrent replacements of a file
- problem created: concurrent reading & replacement

Fig. 10-11. Four ways of dealing with the shared files in a distributed system.

File Locking in NFS (1)

Stateless server => separate locking service needed
NFS traditionally: ”yes, but ...”
NFS v4: locking integrated into the access protocol

Operation	Description
Lock	Creates a lock for a range of bytes - unblocking (failure => start polling) - possibility: queuing of requests (to be refreshed) Grant: for a specific time (cont.: renew operation)
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

Fig. 10-12. NFS version 4 operations related to file locking.

File Locking in NFS (2)

- Share reservation
  - an implicit way to lock a file
  - independent from locking
  - usage: implementation of NFS for Windows-based systems
- Open file specifications:
  - required type of access (READ, WRITE, BOTH)
  - access types to be denied for other clients (NONE, READ, WRITE, BOTH)

File Locking in NFS (3)

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

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

Fig. 10-13. The result of an *open* operation with share reservations in NFS.

Caching in NFS

- Caching: file data, attributes, handles, directories
- Server caching: write-through OR write on commit (commit on closing)
- Client caching: write on commit (session semantics)

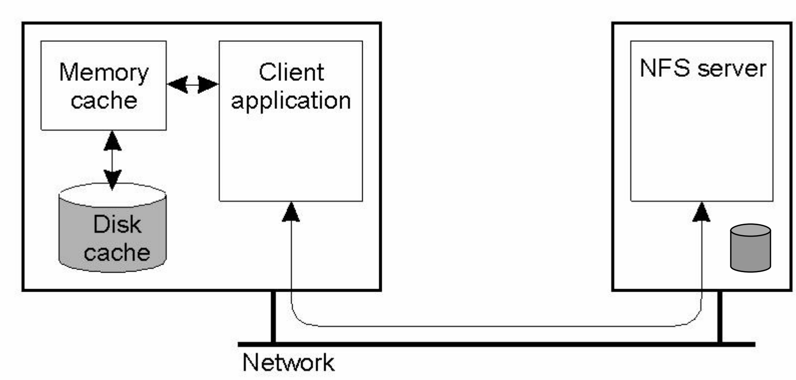


Fig. 10-14. Client-side caching in NFS.

Cache Validity

- Reopen a closed file =>validity check
- NFS v3: validation of **each** *read* !
  - recently checked => accept, otherwise check at server
  - performance vs. consistency: what is “recent”?
- NFS v4: delegation of rights to a client
  - the client is allowed to locally handle *open* and *close* from other clients on the same node
  - requests from other nodes => the server denies
  - recalling of delegation: a *callback* operation
- Consistency of cached of attribute values ??

NFS: Delegation of Rights

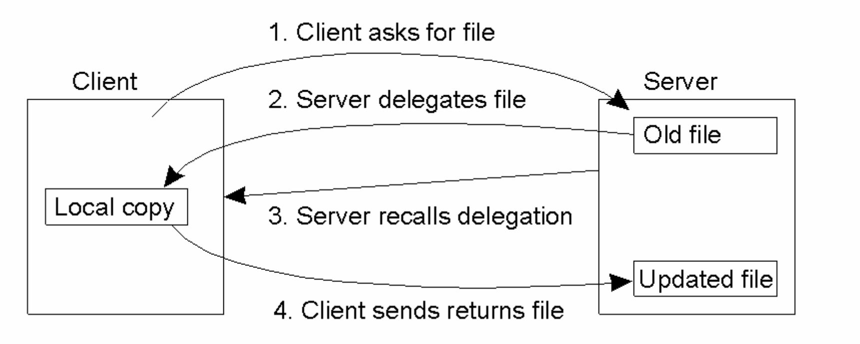
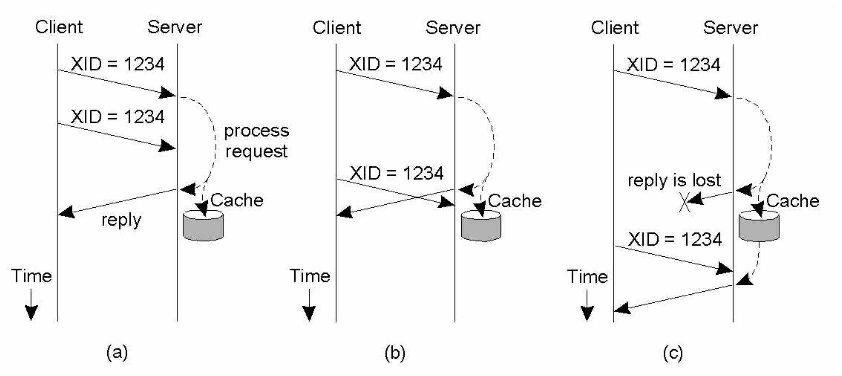


Fig. 10-15. Using the NFS version 4 callback mechanism to recall file delegation.

21-Apr-06 31

Fault Tolerance: RPC Failures



The request is still in progress

The reply has just been returned

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

Fig. 10-16. Three situations for handling retransmissions.

21-Apr-06 32

Fault Tolerance: Locking

- Client / server crashes => granted locks?
- Client:
  - the server issues a **lease** on every lock
  - the lease expires => lock is removed
  - the client can renew its lease (before it expires)
- Server crashes and recovers:
  - the server enters a **grace period**
  - a client can reclaim its old locks
  - (no new locks are granted)
  - => the previous state wrt locks is rebuilt
- **Problems**: non-synchronized clocks; network partitioning

21-Apr-06 33

Security in NFS (1)

- NFS file system: remote ~ local => *communication* is the issue (security: secure RPC's)
- File access control: access control attributes of the file & FS access control

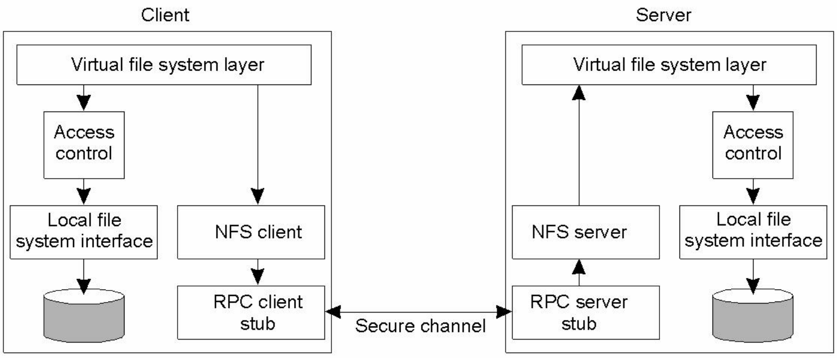


Fig. 10-17. The NFS security architecture.

21-Apr-06 34

Security in NFS (2)

Version 3

1. System authentication:
   user ID, group ID, memberships in groups => server (as plaintext)
2. **Secure NFS**: public-key cryptosystem
   (problems: key distribution, length of the key)
3. Kerberos authentication

Version 4: A general security framework RPCSEC_GSS

- GSS-API (Generic Security Service)
- user-chosen security mechanisms, for example:
  - Kerberos
  - LIPKEY
    authentication
    - clients: password
    - servers: public key

Secure RPCs

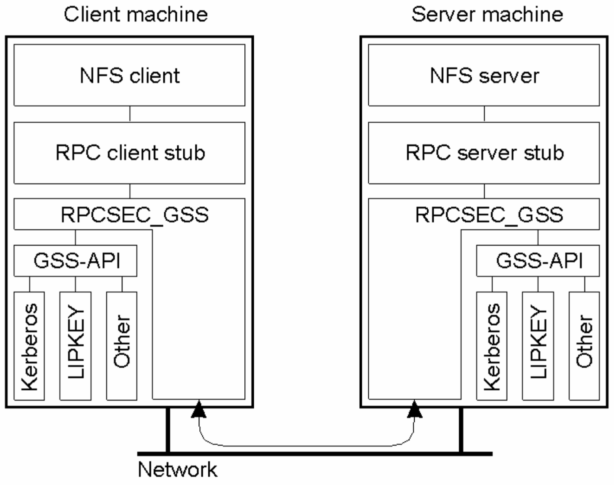


Fig. 10-18. Secure RPC in NFS version 4 (with Generic Security Service framework)

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 to 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

Fig. 10-19. The classification of operations recognized by NFS with respect to access control.

21-Apr-06 37

Access Control: User Types

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

Fig. 10-20. The various kinds of users and processes distinguished by NFS with respect to access control.

21-Apr-06 38

Performance Problems

- Cache validation (frequent use of *getattr*)
- Write with the write-through
  (for large files; in typical UNIX workloads, only 5% of calls to the server are writes)
- Name resolving (50% of ops are lookups !)

- Benchmark results: see www.spec.org
  measurements:
  - ~ 5 ms response times
  - 12.000 – 300.000 ops / sec

21-Apr-06 39

Transparencies (1)

- Access: normal UNIX
- Location: individual mountings => single network-wide name spaces not enforced
- Mobility: filesystems move => mount tables must be updated
- Scalability: managerial problem (yes - but …)
- File replication:
  - read-only OK
  - updates:? (Sun Network Information System supports)

21-Apr-06 40

Transparencies (2)

- Hardware operating system heterogeneity:
  NFS is widely implemented
- Fault tolerance
  - stateless service, idempotent operations
  - remote failures ~ local failures
  - restart "at point of interruption"
- Consistency: not for close coordination needs
- Security: secure RPC available (but not always used)
- Efficiency: widely accepted in heavy-use environments

21-Apr-06 41

The Coda File System

- CMU: campus-wide Workstation net (1983);
  user mobility: anybody anywhere anytime
- Workstation
  - desktop, with a disk
  - BSD Unix (modified)
  - homogeneous file system (distributed, transparent)
  - otherwise: totally independent computers
- Scalability: up to 10.000 workstations, most of which may be active
- History
  - Andrew File System: AFS-1 **AFS-2**, AFS-3
  - AFS-2 => **Coda**

21-Apr-06 42

AFS / Coda Architecture

- AFS/Coda vs NFS:
  - upload/download model
  - consistency control
  - Coda: disconnected operation allowed
- Assumptions made
  - most files are small
  - read operations are more common than write operations
  - most files have only one user (at a time)
  - sequential access is common, random access is rare
  - files are referenced in bursts
- Architecture: Figures
  - Tanenbaum 10-21, CoDoKi 8-11

Overview of AFS (1)

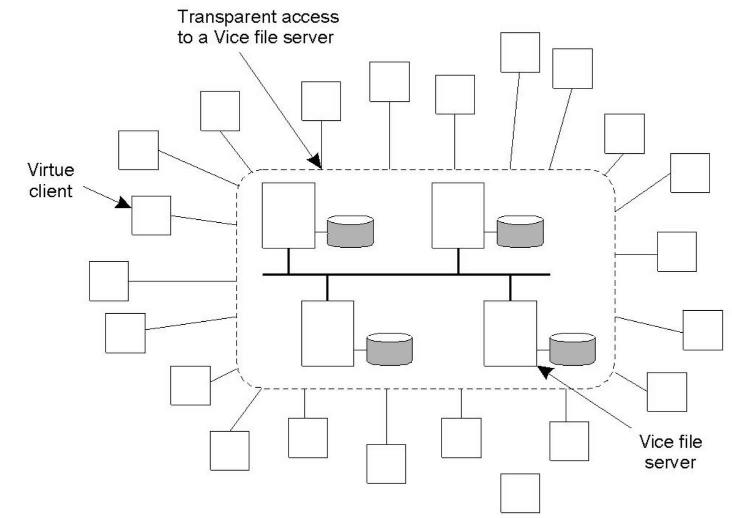


Fig. 10-21. The overall organization of AFS / Coda.

Distribution of processes in AFS

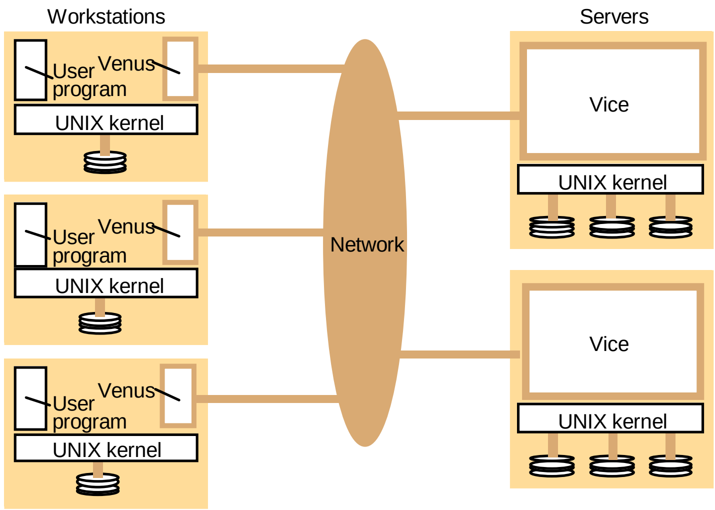


Figure 8.11

21-Apr-06 45

System call interception in AFS

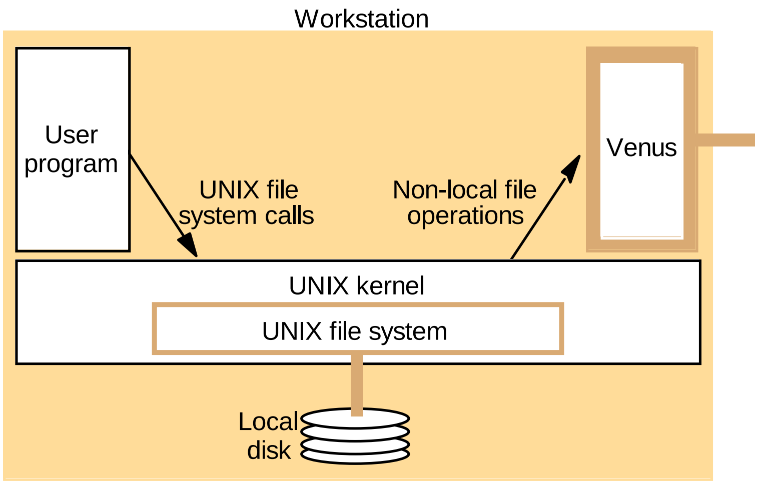


Figure 8.13

21-Apr-06 46

Overview of AFS (2)

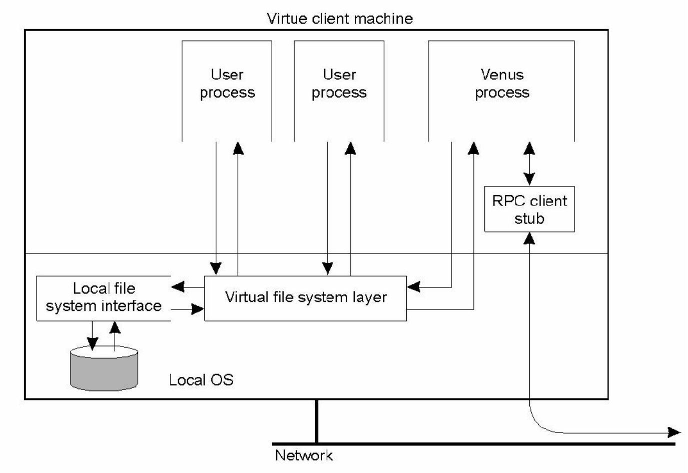


Fig. 10.22. The internal organization of a Virtue workstation.

21-Apr-06 47

AFS: the Implementation

- Files
  - local: normal UNIX files, on the WS disk
  - shared: stored on servers, copies in local caches
- System call interception (open, close): Fig. 8.13
- Implementation of systems calls: Fig. 8.14
- The name space: CoDoKi, Fig. 8.12
  - local: tmp
  - user's directories: cmu
    (=> location transparency for moving users)

21-Apr-06 48

Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open (FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
		Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
	Open the local file and return the file descriptor to the application.			

Figure 8.14

21-Apr-06 49

Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
read (FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write (FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close (FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback break* to all other clients holding *callback promises* on the file.

Figure 8.14

21-Apr-06 50

File name space seen by clients of AFS

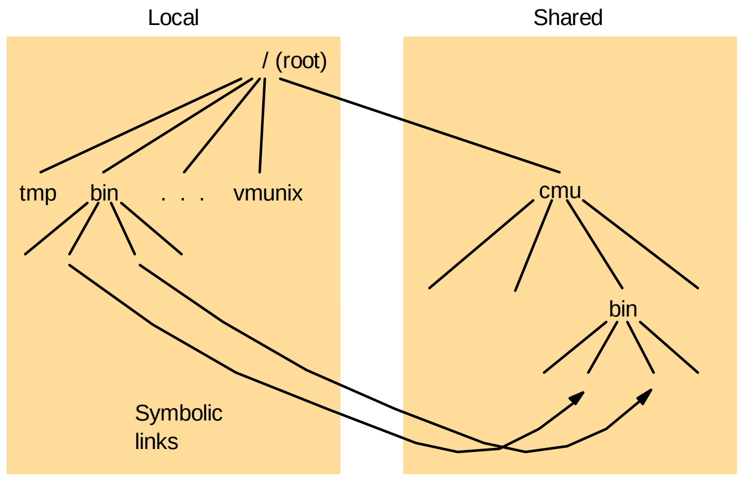


Figure 8.12

Compare with NFS name space

Communication in Coda

Communication method: "advanced RPC"

- reliable RPC on top of UDP
- client: thread per call
- server: "still working" messages (=> "fail stop")

- Side effect (see: Fig. 10-23)
  - interface for application-dependent protocols
  - example: create an isochronous stream connection
- Support for multicasting (see: Fig. 10-24)

  need: implementation of cache consistency
  (notification of invalidity)

Communication: Side Effects

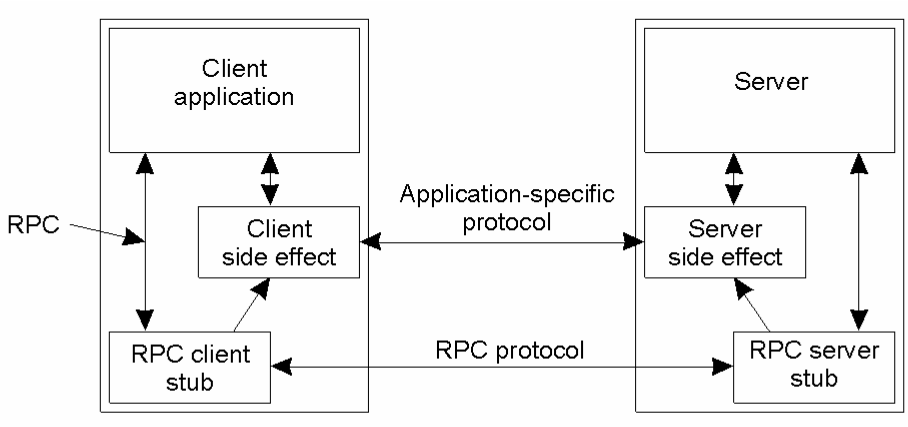
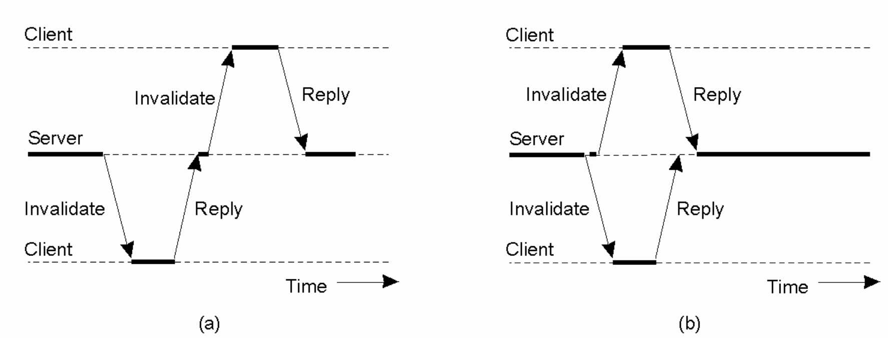


Fig. 10-23. Side effects in Coda's RPC2 system.

21-Apr-06 53

Communication: MultiRPC



Sending an invalidation message one at a time.

Sending invalidation messages in parallel.

Transparency:
- the callee: fully transparent
- the caller: "largely transparent"

Implementation:
multiple RPCs in parallel

21-Apr-06 54

FS Organization: Volumes

- Volume (see: Fig. 10-25)
  - a subtree in the shared name space
  - volume ~ user
- Mounting
  - a mount point is a leaf node of a volume … … that refers to the root node of another volume
- Unit of server-side replication (AFS: only read-only volumes replicated)

Note: when a volume is mounted, Venus follows the structure of the shared name space (unlike NFS) .

Naming

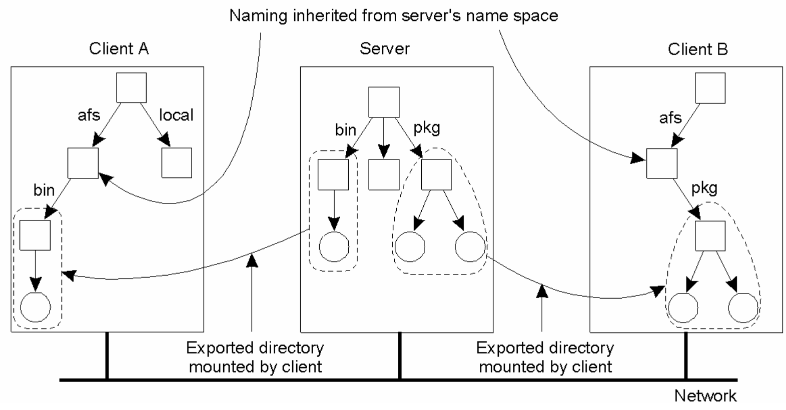
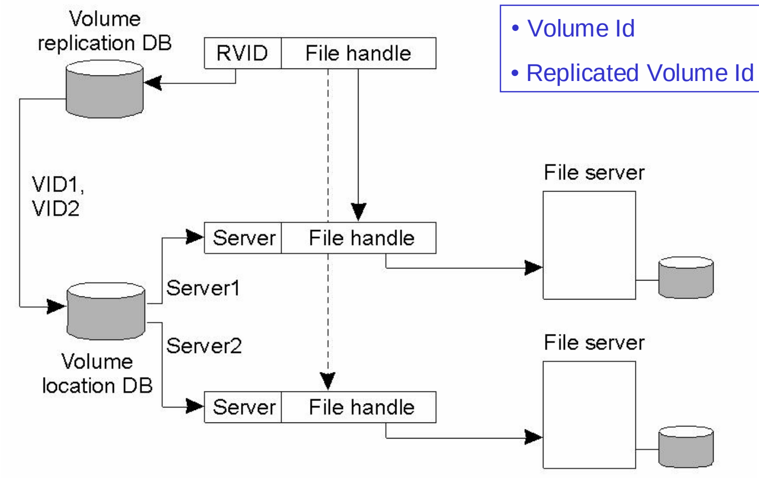


Fig. 10-25. Clients in Coda have access to a single shared name space.

File Identifiers



- Volume Id
- Replicated Volume Id

Fig. 10-26. The implementation and resolution of a Coda file identifier.

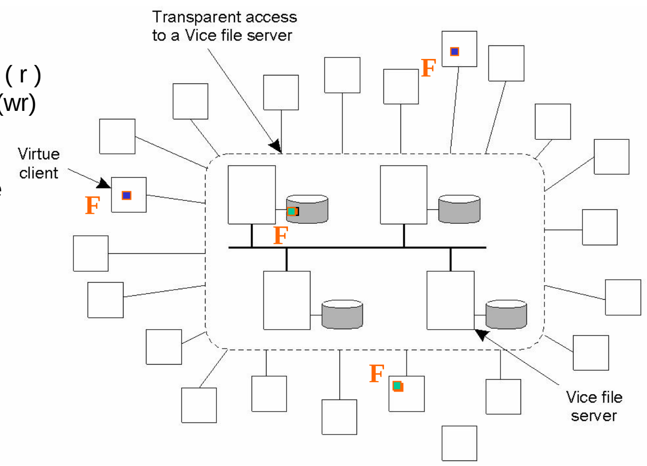
21-Apr-06 57

Sharing Files (1)

1. A,B download F (r)
2. C downloads F (wr)
3. C updates F
4. C closes F
5. F: Virtue -> Vice

A,B: old versions ?

Coda:
1. Vice -> A,B : "F invalid"



21-Apr-06 58

Sharing Files (2)

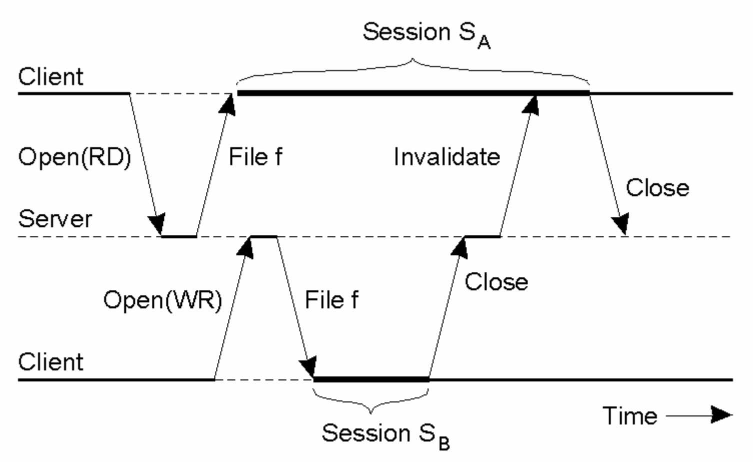


Fig. 10-27. The transactional behavior in sharing files in Coda:
session ~ transaction

21-Apr-06 59

Sharing Files (3)

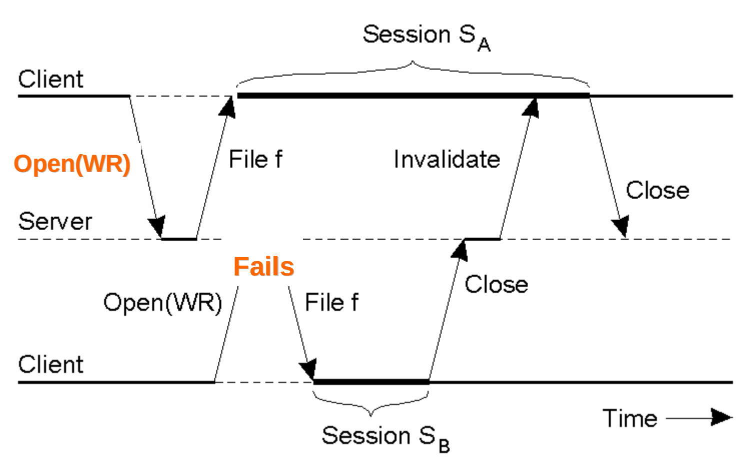


Fig. 10-27. The transactional behavior in sharing files in Coda:
session ~ transaction

21-Apr-06 60

Caching in AFS

- Caching is crucial
  - scalability
  - fault tolerance

  => entire file caching

- Open => download (to cache)

- Close => upload, a copy remains in the cache

21-Apr-06 61

AFS: Cache Consistency

- Re-open: the copy still valid?
- AFS-1: Ask Vice (=> a performance problem!)
- AFS-2 / Coda :
  - open => Venus to Vice: make a **callback promise**
  - update => Vice to all on the callback list: **callback break**
  - re-open => Venus:
    - check the callback promise
    - valid => use the file
    - cancelled => fetch the file from the server

21-Apr-06 62

Client Caching

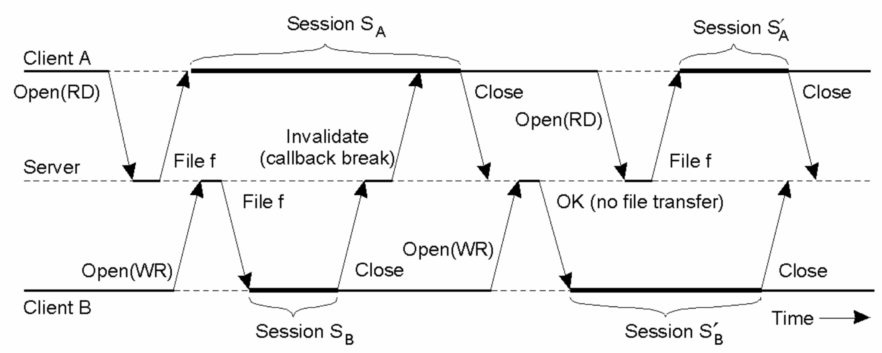


Fig. 10-29. The use of local copies when opening a session in Coda.

Server replication: Coda

Coda

- scalability of AFS: only read-only volumes replicated
- availability in spite of disconnections in the network
- availability for portable workstations

⇒ **constant data availability**

- **Volume Storage Group:**
  the servers which have a copy of the volume
- **Accessible VSG:**
  the servers of the VSG which the client can contact
- **Disconnected client:** the AVSG is empty

Use of a Replicated File

- Open for read: *read any* (in AVSG)
- Close an updated file ("write all available") : send the file to AVSG using *multiRPC*
- **Problem**: AVSG != VSG (network partitioned)

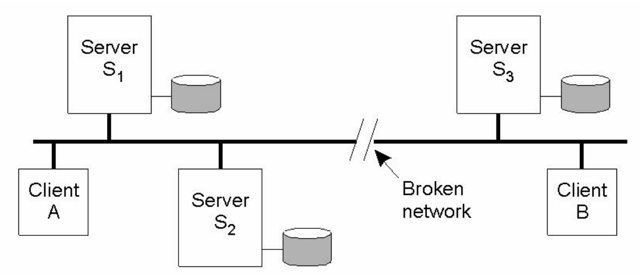


Fig. 10-30

File Consistency

- Concurrent usage allowed:
  - transaction semantics
  - AVSG != VSG => an optimistic approach: use it
- Consistency checking at reconnection
  - detect conflicts
  - recovery: application / manager dependent
- Detection:
  - file f: version number k
  - server i , file f : version vector $CVV_i(f)$

Coda Version Vector

- $CVV_i(f)[j] = k$ ó
  - S_i **knows** that
  - S_j **has seen** at least **version k** of the file f
  - $CVV_i[i]$: the current version of the local copy
- Update: increment CVV[i] for all i: S_i in AVSG
  (file transfer: a reliable multicast wrt. AVSG, see Ch. 6)
- Consistency check at reintegration of S_i and S_j :

  if $CVV_i(f)$ =< $CVV_j(f)$ or $CVV_j(f)$ >= $CVV_i(f)$ then
  - no conflicts
  - the newer replica is based on the older one, which can be brought up to date with the newer one

21-Apr-06 67

Disconnected Operation (1)

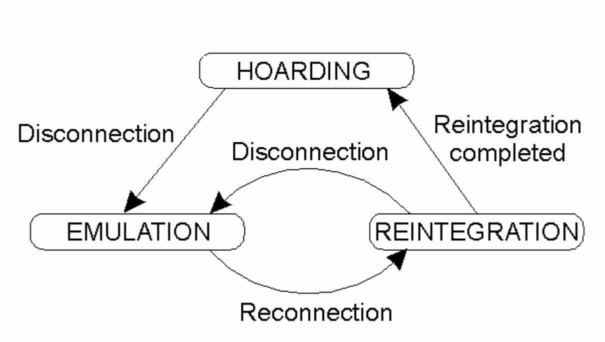


Fig. 10-31. The state-transition diagram of a Coda client with respect to a volume.

21-Apr-06 68

Disconnected Operation (2)

- Disconnected operation OK if all needed files in the local cache => **prefetch**!
- Hoarding state
  - the client is connected to (at least) one server
  - keep the cache full of useful data (files, directories, …)
  - selection: some intelligence is used
- Emulation state
  - use the local cache
  - in case of a miss, try to (re-)contact a server
- Reintegration state
  - updated files => servers

AFS: Other Aspects

- Location database: fully replicated (in all servers)
- Vice, Venus: thread based (=> concurrent ops)
- Bulk transfers (64 kbytes): to minimize latency
- Security:
  - all data transfer encrypted
  - directories: access control lists (incl: *negative* rights)
- Performance (method: benchmarking)
  - important features
    - whole-file caching, callbacks
    - workload: write-sharing hardly ever occurs
  - outperforms NFS;
    reason: load transfer from the server to the WS

For more information, see:

M. Satyanarayanan, The Evolution of Coda; ACM TOCS, May 2002

A distributed system

is
a collection of independent computers
that appears to its users
as a single coherent system.

The End