Distributed File Systems

Chien-Min Wang
Institute of Information Science
Academia Sinica

Contents

- File System Overview
- Distributed File Systems: Issues
- Distributed File Systems: Case Studies
- Distributed File Systems for Clouds

Lecture 2
Distributed File Systems: Issues

Outline

- Introduction
- Basic Implementation Mechanisms
- Design Choices

Why distributed file systems?

- The data may be much larger than the storage space of a computer.
- The data may survive much longer than the life of a computer.
- A user may access his/her data from different machines at different geographic locations.
- A user may want to share his/her data with users around the world.

Accessing Files on Remote Sites

- FTP
  - Explicit access
  - User-directed connection to access remote resources
- We want more transparency
  - Allow user to access remote files just as local ones

File Service Types$_1$

- Upload/Download Model
  - Read file: copy file from server to client
  - Write file: copy file from client to server
- Advantage
  - Simple
- Problems
  - Wasteful: what if client needs small piece?
  - Problematic: what if client doesn’t have enough space?
  - Consistency: what if others need to modify the same file?

File Service Types$_2$

- Remote Access Model
- File service provides functional interface:
  - create, delete, read bytes, write bytes, etc…
- Advantages:
  - Client gets only what's needed
  - Server can manage coherent view of the file system
- Problem:
  - Possible server and network congestion
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly

File Servers

- File Directory Service
  - Maps textual names for file to internal locations that can be used by file service
- File service
  - Provides file access interface to clients
- Client module (driver)
  - Client side interface for file and directory service
  - if done right, helps provide access transparency
    - e.g. under vnode layer of Linux virtual file system

Distributed File Systems

- Provide accesses to data 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 symbolic link to a file;
  - etc;

Why is DFS useful?

- Data sharing of multiple users
- User mobility
- Location transparency
- Location independence
- Backups and centralized management

- Not all DFS are the same:
  - High-speed network DFS vs. low-speed network DFS

Interface: File vs. Block

- Data are organized in files, which in turn are organized in directories
- Compare these with disk-level access or “block” access interface: [Read/Write, LUN, block#]
- Key differences:
  - Implementation of the directory/file structure and semantics
  - Synchronization

Digression: Buzz Word Discussion

	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

Sequential Semantics of File Sharing

- Read returns result of last write
- Easily achieved if
  - Only one server
  - Clients do not cache data
- BUT
  - Performance problems if no cache
    - Obsolete data
  - We can write-through
    - Must notify clients holding copies
    - Requires extra state, generates extra traffic

Session Semantics of File Sharing

- Relax the rules
- Changes to an open file are initially visible only to the process (or machine) that modified it.
- Last process to modify the file wins.

Other solutions

- Make files immutable
  - Aids in replication
  - Does not help with detecting modification
- Use atomic transactions
  - Each file access is an atomic transaction
  - If multiple transactions start concurrently
    - Resulting modification is serial

File Usage Patterns

- We can’t have the best of all worlds
- Where to compromise?
  - Semantics vs. efficiency
  - Efficiency = client performance, network traffic, server load
- Understand how files are used

File Usage

- Most files are <10 Kbytes
  - 2005: average size of 385,341 files =197 KB
  - 2007: average size of 440,519 files =451 KB
  - Feasible to transfer entire files (simpler)
  - Still have to support long files
- Most files have short lifetimes
  - Perhaps keep them local
- Few files are shared
  - Overstated problem
  - Session semantics will cause no problem most of the time

Outline

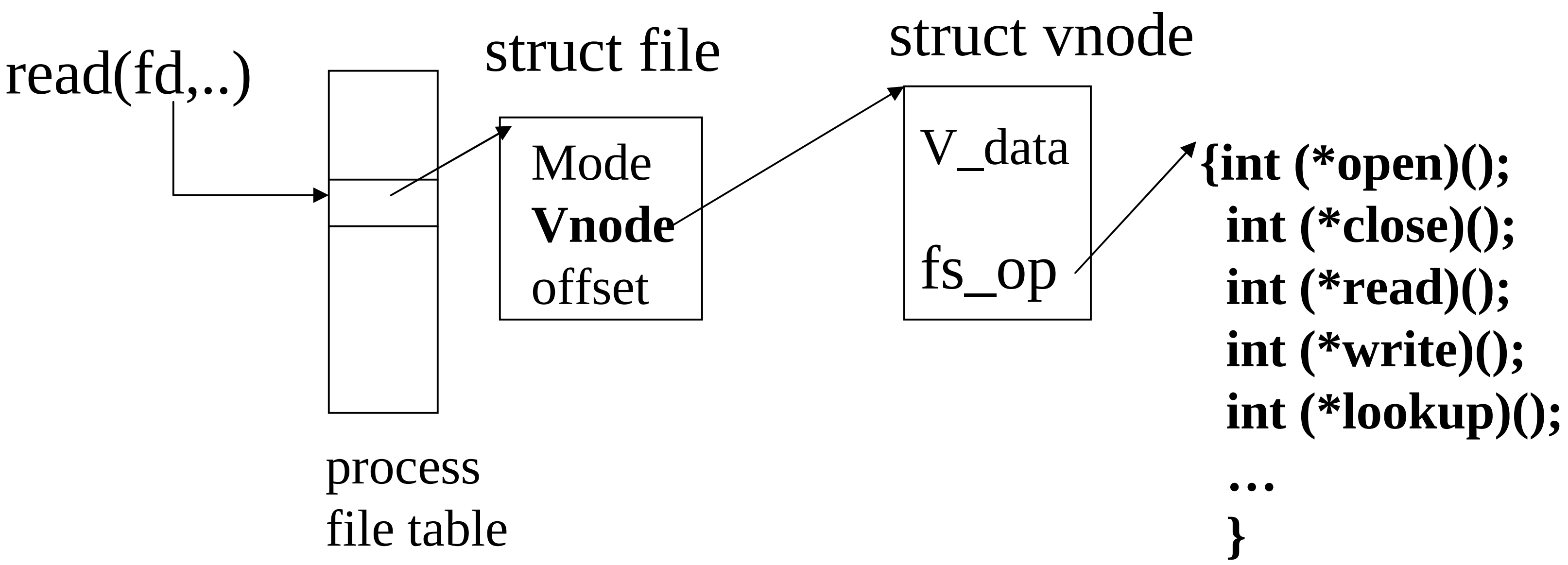
- Introduction
- Basic Implementation Mechanisms
- Design Choices

Components in a DFS

- Client side:
  - What has to happen to enable applications access a remote file in the same way as accessing a local file
- Communication layer:
  - Just TCP/IP or some protocol at higher abstraction
- Server side:
  - How does it service requests from the client

Client Side Example: UNIX

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



read(fd,..)

struct file

Mode
Vnode
offset

struct vnode

V_data
fs_op

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

process
file table

VFS Interception

- VFS provides “pluggable” file systems
- Standard flow of remote access
  - User process calls read()
  - Kernel dispatches to VOP_READ() in some VFS
  - nfs_read()
    - check local cache
    - send RPC to remote NFS server
    - put process to sleep

VFS Interception

- Standard flow of remote access (continued)
  - server interaction handled by kernel process
    - retransmit if necessary
    - convert RPC response to file system buffer
    - store in local cache
    - wake up user process
  - nfs_read()
    - copy bytes to user memory

Communication Layer Example: RPC

RPC call
xid
“call”
service
version
procedure
auth-info
arguments …

RPC reply
xid
“reply”
reply_stat
auth-info
results …

- Failure handling: timeout and re-issuance
- RPC over UDP vs. RPC over TCP

Extended Data Representation (XDR)

- Argument data and response data in RPC are packaged in XDR format
  - Integers are encoded in big-endian
  - Strings: len followed by ascii bytes with NULL padded to four-byte boundaries
  - Arrays: 4-byte size followed by array entries
  - Opaque: 4-byte len followed by binary data
- Marshalling and un-marshalling
- Extra overhead in data conversion to/from XDR

NFS RPC Calls

- NFS / RPC using XDR / TCP/IP

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

- fhandle: 32-byte opaque data (64-byte in v3)
  - What’s in the file handle

NFS Operations

- V2:
  - NULL, GETATTR, SETATTR
  - LOOKUP, READLINK, READ
  - CREATE, WRITE, REMOVE, RENAME
  - LINK, SYMLINK
  - READIR, MKDIR, RMDIR
  - STATFS
- V3: add
  - *READDIRPLUS, COMMIT*
  - FSSTAT, FSINFO, PATHCONF

Server Side Example: mountd and nfsd

- Mountd: provides the initial file handle for the exported directory
  - Client issues nfs_mount request to mountd
  - Mountd checks if the pathname is a directory and if the directory is exported to the client
- nfsd: answers the rpc calls, gets reply from local file system, and sends reply via rpc
  - Usually listening at port 2049
- Both mountd and nfsd use underlying RPC implementation

NFS Client Server Interactions

- Client machine:
  - Application à nfs_vnops à nfs client code à rpc client interface
- Server machine:
  - rpc server interface à nfs server code à ufs_vnops à ufs code à disks

NFS File Server Failure Issues

- Semantics of file write in V2
  - Bypass UFS file buffer cache
- Semantics of file write in V3
  - Provide “COMMIT” procedure
- Server-side retransmission cache
  - Idempotent vs. non-idempotent requests

Outline

- Introduction
- Basic Implementation Mechanisms
- Design Choices

Topic 1: Naming

- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1 à 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 à server_ip mappings

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?

Topic 2: User Authentication and Access Control

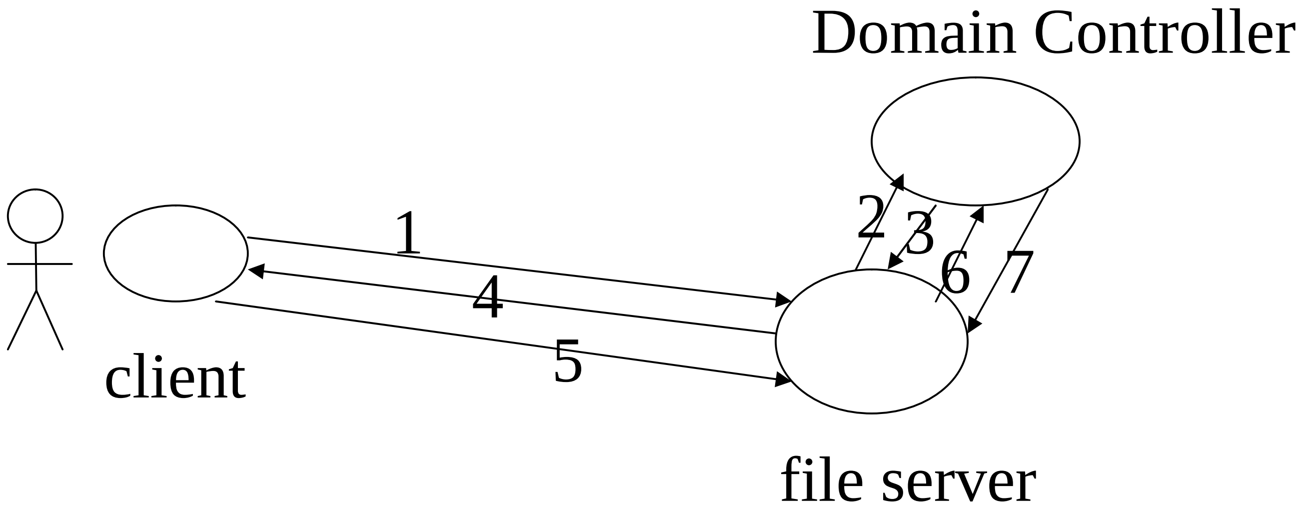
- User X logs onto workstation A, wants to access files on server B
  - How does A tell B who X is
  - Should B believe A
- Choices made in NFS v2
  - All servers and all client workstations share the same <uid, gid> name space à B send X’s <uid,gid> to A
    - Problem: root access on any client workstation can lead to creation of users of arbitrary <uid, gid>
  - Server believes client workstation unconditionally
    - Problem: if any client workstation is broken into, the protection of data on the server is lost;
    - <uid, gid> sent in clear-text over wire à request packets can be faked easily

User Authentication

- How do we fix the problems in NFS v2
  - Hack1: root remapping à strange behavior
  - Hack 2: UID remapping à no user mobility
  - Real Solution: use a centralized Authentication/Authorization/Access-control (AAA) system

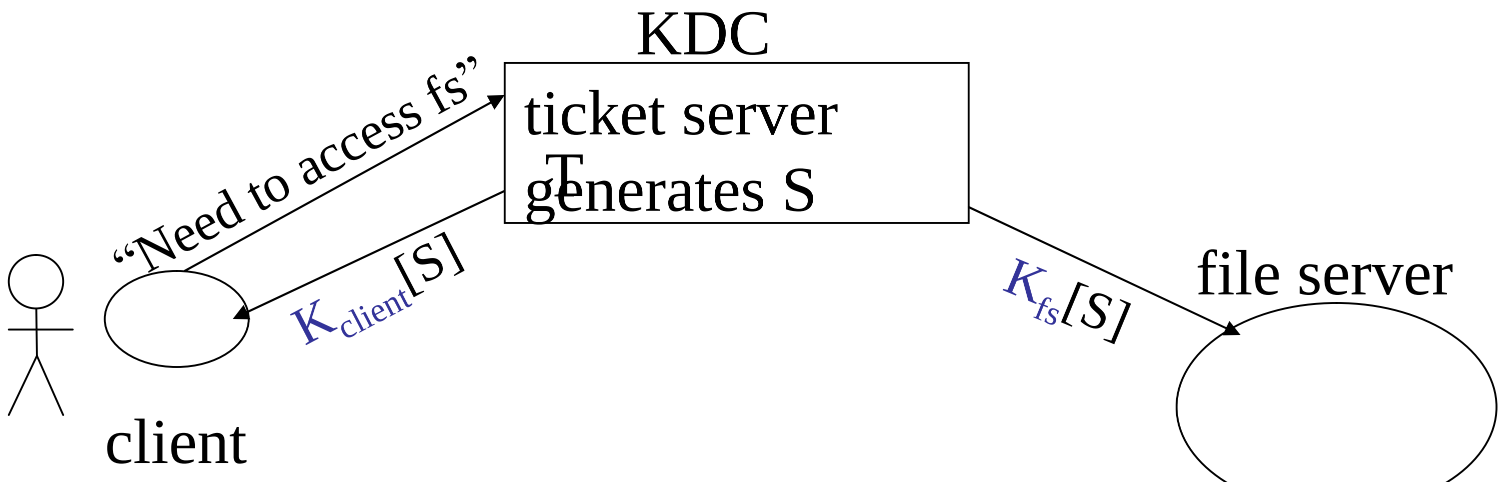
Example AAA System: NTLM

- Microsoft Windows Domain Controller
  - Centralized AAA server
  - NTLM v2: per-connection authentication



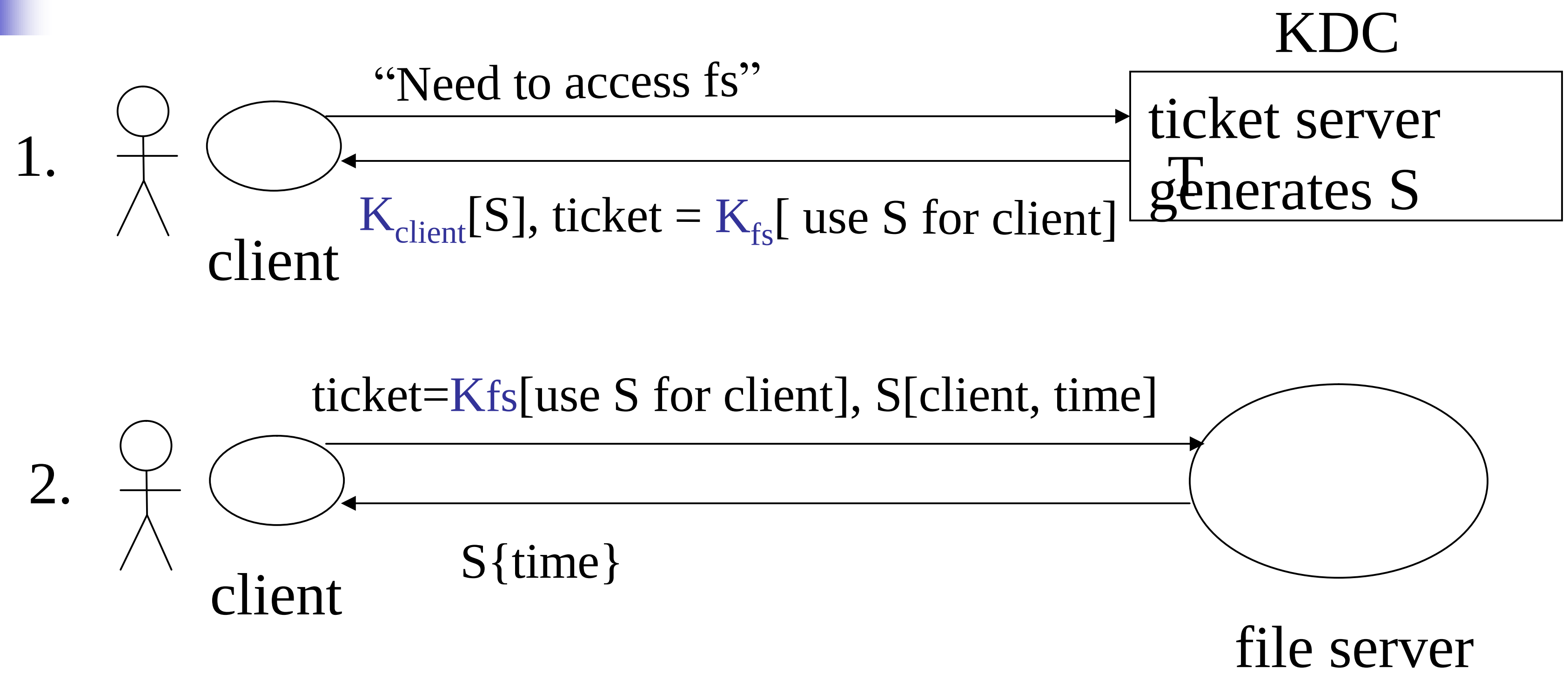
A Better AAA System: Kerberos

- Basic idea: shared secrets
  - User prove to KDC who he is; KDC generates shared secret between client and file server



S: specific to {client,fs} pair;
"short-term session-key"; has expiration time (e.g. 8 hours);

Kerberos Interactions



KDC

1\.

client

“Need to access fs”

K_{client}[S], ticket = K_{fs}[use S for client]

ticket server

T

generates S

2\.

client

ticket=Kfs[use S for client], S[client, time]

S{time}

file server

- why “time”: guard against replay attack
- mutual authentication
- File server doesn’t store S, which is specific to {client, fs}
- Client doesn’t contact “ticket server” every time it contacts fs

Kerberos: User Log-on Process

- How does user prove to KDC who the user is
  - Long-term key: 1-way-hash-func(passwd)
  - Long-term key comparison happens once only, at which point the KDC generates a shared secret for the user and the KDC itself à ticket-granting ticket, or “logon session key”
  - The “ticket-granting ticket” is encrypted in KDC’s long-term key

Topic 3: Operator Batching

- Should each client/server interaction accomplish one file system operation or multiple operations?
- Advantage of batched operations
- How to define batched operations

Examples of Batched Operators

- NFS v3:
  - Readdirplus
- NFS v4:
  - Compound RPC calls
- CIFS:
  - “AND-X” requests

Topic 4: Client-Side Caching

- Why is client-side caching necessary
- What are cached
  - Read-only file data and directory data à easy
  - Data written by the client machine à when are data written to the server? What happens if the client machine goes down?
  - Data that are written by other machines à how to know that the data have been changed? How to ensure data consistency?
  - Is there any pre-fetching?

Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
- File attributes in the client cache are expired after 60 seconds
- File data are checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 secs to be reflected on another machine
- Dirty data are buffered on the client machine till file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost
  - Similar to UNIX FFS local file system behavior

Implication of NFS v2 Client Caching

- Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Productivity apps tend to tolerate such loose consistency
- Different client implementations implement the “prefetching” part differently
- Generally clients do not cache data on local disks

Client Caching in AFS

- Client caches both clean and dirty file data and attributes
    - The client machine uses local disks to cache data
    - When a file is opened for read, the whole file is fetched and cached on disk
        - Why? What's the disadvantage of doing so?
- However, when a client caches file data, it obtains a "callback" on the file
- In case another client writes to the file, the server "breaks" the callback
    - Similar to invalidations in distributed shared memory implementations
- Implications: file server must keep states!

AFS RPC Procedures

- Procedures that are not in NFS
  - Fetch: return status and optionally data of a file or directory, and place a callback on it
  - RemoveCallBack: specify a file that the client has flushed from the local machine
  - BreakCallBack: from server to client, revoke the callback on a file or directory
    - What should the client do if a callback is revoked?
  - Store: store the status and optionally data of a file
- Rest are similar to NFS calls

Failure Recovery in AFS

- What if the file server fails
  - Two candidate approaches to failure recovery
- What if the client fails
- What if both the server and the client fail
- Network partition
  - How to detect it? How to recover from it?
  - Is there anyway to ensure absolute consistency in the presence of network partition?
    - Reads
    - Writes
- What if all three fail: network partition, server, client

Key to Simple Failure Recovery

- Try not to keep any state on the server
- If you must keep some states on the server
  - Understand why and what states the server is keeping
  - Understand the worst case scenario of no state on the server and see if there are still ways to meet the correctness goals
  - Revert to this worst case in each combination of failure cases

Topic 5: File Access Consistency

- In UNIX local file system, concurrent file reads and writes have "sequential" consistency semantics
  - Each file read/write from user-level app is an atomic operation
    - The kernel locks the file vnode
  - Each file write is immediately visible to all file readers
- Neither NFS nor AFS provides such concurrency control
  - NFS: "sometime within 30 seconds"
  - AFS: session semantics for consistency

Session Semantics in AFS

- What it means:
  - A file write is visible to processes on the same box immediately, but not visible to processes on other machine until the file is closed
  - When a file is closed, changes are visible to new opens, but are not visible to "old" opens
  - All other file operations are visible everywhere immediately
- Implementation
  - Dirty data are buffered at the client machine until file close, then flushed back to server, which leads the server to send "break callback" to other clients
  - Problems with this implementation

Access Consistency in “Sprite”

- Sprite: a research file system developed in UC Berkeley in late 80’s
- Implements “sequential” consistency
  - Caches only file data, not file metadata
  - When server detects a file is open on multiple machines but is written by some client, client caching of the file is disabled; all reads and writes go through the server
  - “Write-back” policy otherwise
    - Why?

Implementing Sequential Consistency

- How to identify out-of-date data blocks
  - Use file version number
  - No invalidation
  - No issue with network partition
- How to get the latest data when read-write sharing occurs
  - Server keeps track of last writer

Implication of “Sprite” Caching

- Server must keep states!
  - Recovery from power failure
  - Server failure doesn’t impact consistency
  - Network failure doesn’t impact consistency
- Price of sequential consistency: no client caching of file metadata; all file opens go through server
  - Performance impact
  - Suited for wide-area network?

Access Consistency in AFS v3

- Motivation
  - How does one implement sequential consistency in a file system that spans multiple sites over WAN
    - Why Sprite's approach won't work
    - Why AFS v2 approach won't work
    - Why NFS approach won't work
- What should be the design guidelines?
  - What are the common share patterns?

“Tokens” in AFS v3

- Callbacks are evolved into 4 kinds of “Tokens”
  - Open tokens: allow holder to open a file; submodes: read, write, execute, exclusive-write
  - Data tokens: apply to a range of bytes
    - “read” token: cached data are valid
    - “write” token: can write to data and keep dirty data at client
  - Status tokens: provide guarantee of file attributes
    - “read” status token: cached attribute is valid
    - “write” status token: can change the attribute and keep the change at the client
  - Lock tokens: allow holder to lock byte ranges in the file

Compatibility Rules for Tokens

- Open tokens:
  - Open for exclusive writes are incompatible with any other open, and “open for execute” are incompatible with “open for write”
  - But “open for write” can be compatible with “open for write” --- why?
- Data tokens: R/W and W/W are incompatible if the byte range overlaps
- Status tokens: R/W and W/W are incompatible
- Data token and status token: compatible or incompatible?

Token Manager

- Resolve conflicts: block the new requester and send notification to other clients' tokens
- Handle operations that request multiple tokens
  - Example: rename
  - How to avoid deadlocks

Failure Recovery in Token Manager

- What if the server fails
- What if a client fails
- What if network partition happens

Topic 6: File Locking

- Issues
  - Whole file locking or byte-range locking
  - Mandatory or advisory
    - UNIX: advisory
    - Windows: if a lock is granted, it’s mandatory on all other accesses
- NFS: network lock manager (NLM)
  - NLM is not part of NFS v2, because NLM is stateful
  - Provides both whole file and byte-range locking
  - Advisory
  - Relies on “network status monitor” for server monitoring

Issues in Locking Implementations

- Synchronous and Asynchronous calls
  - NLM provides both
- Failure recovery
  - What if server fails
    - Lock holders are expected to re-establish the locks during the “grace period”, during which no other locks are granted
  - What if a client holding the lock fails
  - What if network partition occurs

Wrap up: Comparing the File Systems

- Caching:
  - NFS
  - AFS
  - Sprite
- Consistency
  - NFS
  - AFS
  - Sprite
  - AFS v3
- Locking

Wrap up: Comparison with the Web

- Differences:
  - Web offers HTML, etc. DFS offers binary data only
  - Web has a few but universal clients; DFS is implemented in the kernel
- Similarities:
  - Caching with TTL is similar to NFS consistency
  - Caching with IMS-every-time is similar to Sprite consistency
    - As predicted in AFS studies, there is a scalability problem here
- Security mechanisms
  - AAA similar
  - Encryption?

Topic 7: Stateful or stateless design?

- Stateful
  - Server maintains client-specific states
- Shorter requests
- Better performance in processing requests
- Cache coherence is possible
  - Server can know who’s accessing what
- File locking is possible

Topic 7: Stateful or stateless design?

- Stateless
  - Server maintains no information on client accesses
- Each request must identify file and offsets
- Server can crash and recover
  - No state to lose
- Client can crash and recover
- No open/close needed
  - They only establish state
- No server space used for state
  - Don’t worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible