15-440 Distributed Systems

Lecture 7 – Distributed File Systems 1

1

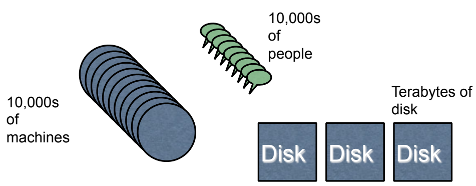
Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

2

andrew.cmu.edu

- Let’s start with a familiar example: andrew



Goal: Have a consistent namespace for files across computers. Allow any authorized user to access their files from any computer

Why DFSs are Useful

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

4

What Distributed File Systems Provide

- Access to data stored at servers using file system interfaces
- What are the file system interfaces?
  - Open a file, check status of 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, create/delete a directory
  - Delete a file, rename a file, add a symlink to a file
  - etc

5

Challenges

- Remember our initial list of challenges...
- Heterogeneity (lots of different computers & users)
- Scale (10s of thousands of peeps!)
- Security (my files! hands off!)
- Failures
- Concurrency
- oh no... we’ve got ‘em all.

How can we build this??

Just as important: non-challenges

- Geographic distance and high latency
  - Andrew and AFS target the campus network, *not* the wide-area

Prioritized goals? / Assumptions

- Often very useful to have an explicit list of prioritized goals. Distributed filesystems almost always involve trade-offs
- Scale, scale, scale
- User-centric workloads... how do users use files (vs. big programs?)
  - Most files are personally owned
  - Not too much concurrent access; user usually only at one or a few machines at a time
  - Sequential access is common; reads much more common that writes
  - There is locality of reference (if you've edited a file recently, you're likely to edit again)

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

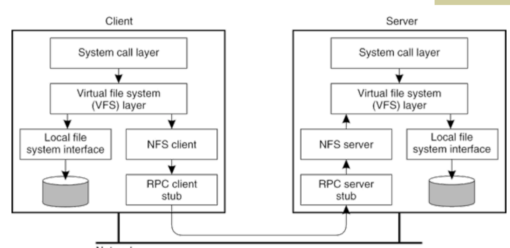
Components in a DFS Implementation

- Client side:
  - What has to happen to enable applications to access a remote file the same way a local file is accessed?
  - Accessing remote files in the same way as accessing local files → kernel support
- Communication layer:
  - Just TCP/IP or a protocol at a higher level of abstraction?
- Server side:
  - How are requests from clients serviced?

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
  - 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

VFS Interception



A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- For andrew context: bad bad bad: server would get hammered!

Lesson 1: Needing to hit the server for every detail impairs performance and scalability.

Question 1: How can we avoid going to the server for everything? *What* can we avoid this for? What do we lose in the process?

NFS V2 Design

- "Dumb", "Stateless" servers w/ smart clients
- Portable across different OSes
- Low implementation cost
- Small number of clients
- Single administrative domain

Some NFS V2 RPC Calls

- NFS RPCs using XDR over, e.g., 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)

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 should be 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 V2 Operations

- V2:
  - NULL, GETATTR, SETATTR
  - LOOKUP, READLINK, READ
  - CREATE, WRITE, REMOVE, RENAME
  - LINK, SYMLINK
  - READIR, MKDIR, RMDIR
  - STATFS (get file system attributes)

NFS V3 and V4 Operations

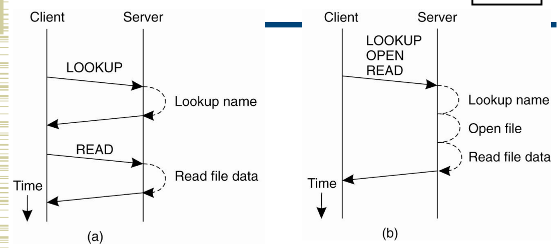
- V3 added:
  - READDIRPLUS, COMMIT (server cache!)
  - FSSTAT, FSINFO, PATHCONF
- V4 added:
  - COMPOUND (bundle operations)
  - LOCK (server becomes more stateful!)
  - PUTROOTFH, PUTPUBFH (no separate MOUNT)
  - Better security and authentication
  - Very different than V2/V3 → stateful

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

19

Remote Procedure Calls in NFS



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

20

AFS Goals

- Global distributed file system
  - "One AFS", like "one Internet"
    - Why would you want more than one?
- **LARGE** numbers of clients, servers
  - 1000 machines could cache a single file,
  - Most local, some (very) remote
- Goal: O(0) work per client operation
  - O(1) may just be too expensive!

21

AFS Assumptions

- Client machines are un-trusted
  - Must **prove** they act for a specific user
    - Secure RPC layer
  - Anonymous "system:anyuser"
- Client machines have disks(!!)
  - Can cache whole files over long periods
- Write/write and write/read sharing are rare
  - Most files updated by one user, on one machine

22

AFS Cell/Volume Architecture

- Cells correspond to administrative groups
  - /afs/andrew.cmu.edu is a ***cell***
- Cells are broken into ***volumes*** (miniature file systems)
  - One user's files, project source tree, ...
  - Typically stored on one server
  - Unit of disk quota administration, backup
- Client machine has cell-server database
  - ***protection server*** handles authentication
  - ***volume location server*** maps volumes to servers

23

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

24

Topic 1: Client-Side Caching

- Huge parts of systems rely on two solutions to every problem:
  1. "All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem." -- David Wheeler
  2. Cache it!

Client-Side Caching

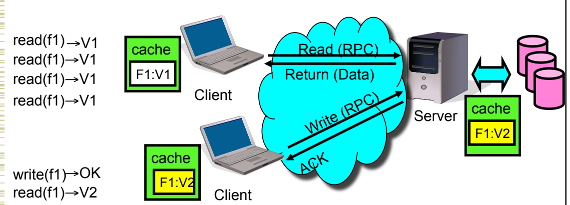
- So, uh, what do we cache?
  - Read-only file data and directory data → easy
  - Data written by the client machine → when is data written to the server? What happens if the client machine goes down?
  - Data that is written by other machines → how to know that the data has changed? How to ensure data consistency?
  - Is there any pre-fetching?
- And if we cache... doesn't that risk making things inconsistent?

26

Failures

- Server crashes
  - Data in memory but not disk lost
  - So... what if client does
    - seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail. Ditto for open(), read()
- Lost messages: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo anew?
- Client crashes
  - Might lose data in client cache

Use of caching to reduce network load



28

Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
- File attributes in the client cache expire after 60 seconds (file data doesn't expire)
- File data is checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 seconds to be reflected on another machine
- Dirty data are buffered on the client machine until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

29

Implication of NFS v2 Client Caching

- Advantage: No network traffic if open/read/write/close can be done locally.
- But…. Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Productivity apps tend to tolerate such loose consistency
- Generally clients do not cache data on local disks

30

NFS' s Failure Handling – Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of "who has this file open" (permission check on each operation on open file!)
  - No "pending transactions" across crash
- Crash recovery is "fast"
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocolStateless design
- Stateless protocol: requests specify exact state. read() → read([position]). no seek on server.

NFS' s Failure Handling

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It' s not delete("foo"), it' s delete(1337f00f), where that ID won' t be reused.
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can' t handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

NSF Results

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it' s gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

Let' s look back at Andrew

- NFS gets us partway there, but
  - Probably doesn' t handle scale (* - you can buy huge NFS appliances today that will, but they' re $$$-y).
  - Is very sensitive to network latency
- How can we improve this?
  - More aggressive caching (AFS caches on disk in addition to just in memory)
  - Prefetching (on open, AFS gets entire file from server, making later ops local & fast).
    - Remember: with traditional hard drives, large sequential reads are much faster than small random writes. So easier to support (client a: read whole file; client B: read whole file) than having them alternate. Improves scalability, particularly if client is going to read whole file anyway eventually.

Client Caching in AFS

- Callbacks! Clients register with server that they have a copy of file;
  - Server tells them: "Invalidate!" if the file changes
  - This trades state for improved consistency
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"

AFS v2 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

36

Topic 2: 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

37

Session Semantics in AFS v2

- What it means:
  - A file write is visible to processes on the same box immediately, but not visible to processes on other machines 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

38

AFS Write Policy

- Writeback cache
  - Opposite of NFS “every write is sacred”
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file?

39

Results for AFS

- Lower server load than NFS
  - More files cached on clients
  - Callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine’s memory over LAN
- For both:
  - Central server is bottleneck: all reads and writes hit it at least once;
  - is a single point of failure.
  - is costly to make them fast, beefy, and reliable servers.

Topic 3: Name-Space Construction and Organization

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

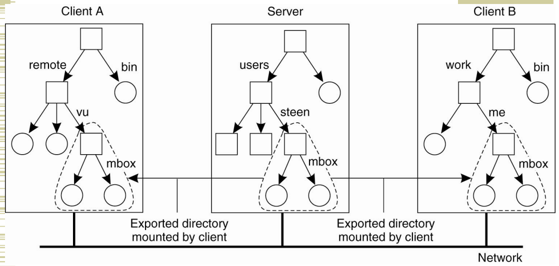
41

Implications on 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

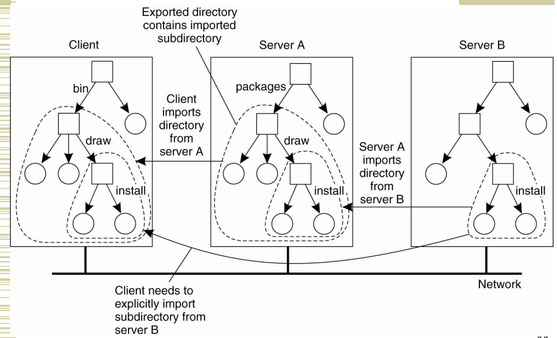
42

Naming in NFS (1)



43

Naming in NFS (2)



44

Topic 4: 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 → 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 → request packets can be faked easily

45

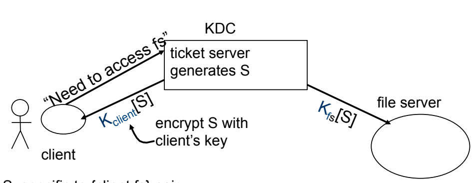
User Authentication (cont'd)

- How do we fix the problems in NFS v2
  - Hack 1: root remapping → strange behavior
  - Hack 2: UID remapping → no user mobility
  - Real Solution: use a centralized Authentication/ Authorization/Access-control (AAA) system

46

A Better AAA System: Kerberos

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



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

47

Today's bits

- Distributed filesystems almost always involve a tradeoff: consistency, performance, scalability.
- We've learned a lot since NFS and AFS (and can implement faster, etc.), but the general lesson holds. Especially in the wide-area.
- We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff. Consistency, Availability, Partition-resilience.

More bits

- Client-side caching is a fundamental technique to improve scalability and performance
  - But raises important questions of cache consistency
- Timeouts and callbacks are common methods for providing (some forms of) consistency.
- AFS picked close-to-open consistency as a good balance of usability (the model seems intuitive to users), performance, etc.
  - AFS authors argued that apps with highly concurrent, shared access, like databases, needed a different model

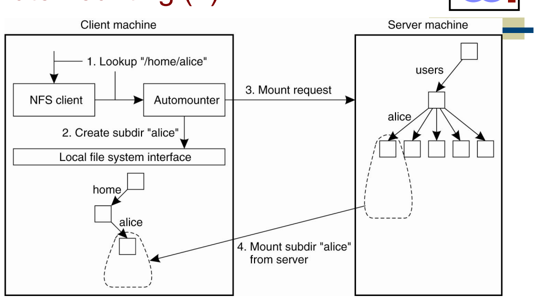
AFS Retrospective

- Small AFS installations are hard
  - Step 1: Install Kerberos
    - 2-3 servers
    - Inside locked boxes!
  - Step 2: Install ~4 AFS servers (2 data, 2 pt/vldb)
  - Step 3: Explain Kerberos to your users
    - Ticket expiration!
  - Step 4: Explain ACLs to your users

AFS Retrospective

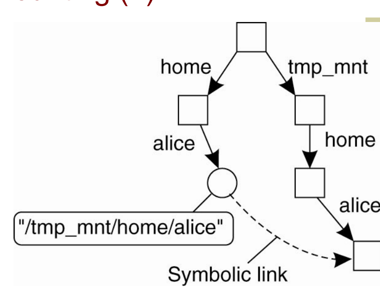
- Worldwide file system
- Good security, scaling
- Global namespace
- “Professional” server infrastructure per cell
  - Don't try this at home
  - Only ~190 AFS cells (2002-03)
    - 8 are cmu.edu, 14 are in Pittsburgh
- “No write conflict” model only partial success

Automounting (1)



- A simple automounter for NFS.

Automounting (2)



- Using symbolic links with automounting.

Access Consistency in the "Sprite" File System

- 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?

55

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

56

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?

57

"Tokens" in DCE DFS

- How does one implement sequential consistency in a file system that spans multiple sites over WAN
- 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

58

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?

59

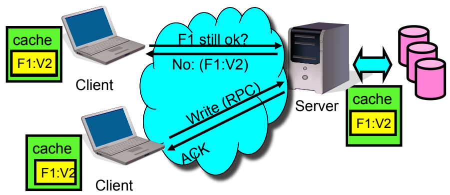
Token Manager

- 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

60

NFS Cache consistency

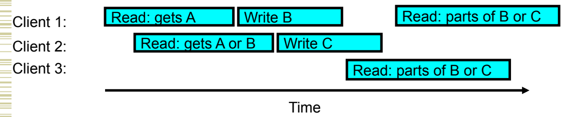
- Client polls server periodically to check for changes
  - Polls server if data hasn't been checked in last 3-30 seconds (exact timeout is tunable parameter).
  - When file is changed on one client, server is notified, but other clients use old version of file until timeout.



- What if multiple clients write to same file?
  - In NFS, can get either version (or parts of both)
  - Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - If read finishes before write starts, get old copy
    - If read starts after write finishes, get new copy
    - Otherwise, get either new or old copy
  - For NFS:
    - If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - As a result, do not get partial writes: all or nothing!
    - Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - Get file from server, set up callback with server
  - On write followed by close:
    - Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

Communication Layer Example: Remote Procedure Calls (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-issue

Extended Data Representation (XDR)

- Argument data and response data in RPC are packaged in XDR format
  - Integers are encoded in big-endian format
  - 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 data
- Extra overhead in data conversion to/from XDR

Client Caching in AFS v2

- 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
- Implication: file server must keep state!

67

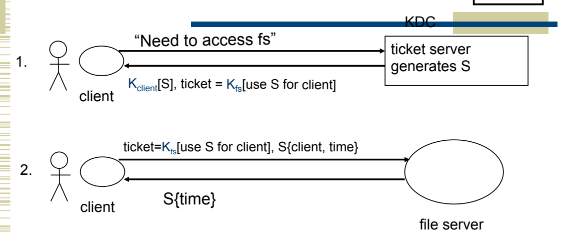
Semantics of File Sharing

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
Transactions	All changes occur atomically

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

68

Kerberos Interactions



- 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

69

AFS Security (Kerberos)

- Kerberos has multiple administrative domains (realms)
  - principal@realm
  - srini@cs.cmu.edu sseshan@andrew.cmu.edu
- Client machine presents Kerberos ticket
  - Arbitrary binding of (user,machine) to Kerberos (principal,realm)
    - dongsuh on grad.pc.cs.cmu.edu machine can be srini@cs.cmu.edu
- Server checks against access control list (ACL)

70

AFS ACLs

- Apply to directory, not to file
- Format:
  - sseshan rlidwka
  - srini@cs.cmu.edu rl
  - sseshan:friends rl
- Default realm is typically the cell name (here andrew.cmu.edu)
- Negative rights
  - Disallow "joe rl" even though joe is in sseshan:friends

71

Failure Recovery in AFS & NFS

- What if the file server fails?
- 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?

72

Key to Simple Failure Recovery

- Try not to keep any state on the server
- If you must keep some state on the server
  - Understand why and what state 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

73