CS425/CSE424/ECE428 – Distributed Systems

Distributed File Systems

Material derived from slides by Dave Eckhart and Bruce Maggs (CMU), I. Gupta, K. Nahrtstedt, S. Mitra, N. Vaidya, M. T. Harandi, J. Hou (UIUC)

Outline

- Why remote file systems?
- VFS interception
- NFS vs. AFS
  - Architectural assumptions & goals
  - Namespace
  - Authentication, access control
  - I/O flow
  - Rough edges

Why?

- Why remote file systems?
- Lots of "access data everywhere" technologies
  - Laptop
  - Multi-gigabyte flash-memory keychain USB devices
  - 4G Hitachi MicroDrive fits in a CompactFlash slot
  - iPod
- Are remote file systems dinosaurs?

Remote File System Benefits

- Reliability
  - Not many people carry multiple copies of data
    - Multiple copies ***with you*** aren't much protection
  - Backups are nice
    - Machine rooms are nice
      - Temperature-controlled, humidity-controlled
      - Fire-suppressed
    - Time travel is nice too
- Sharing
  - Allows multiple users to access data
  - May provide authentication mechanism

Remote File System Benefits

- Scalability
  - Large disks are cheaper
- Locality of reference
  - You don't use every file every day...
    - Why carry *everything* in expensive portable storage?
- Auditability
  - Easier to know who said what when with central storage...

Distributed File System (DFS) Requirements

- **Transparency** - server-side changes should be invisible to the client-side.
  - Access transparency: A single set of operations is provided for access to local/remote files.
  - Location Transparency: All client processes see a uniform file name space.
  - Migration Transparency: When files are moved from one server to another, users should not see it
  - Performance Transparency
  - Scaling Transparency
- File Replication
  - A file may be represented by several copies for service efficiency and fault tolerance.
- Concurrent File Updates
  - Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing the same file.

DFS Requirements (2)

- Concurrent File Updates
  - **One-copy update** semantics: the file contents seen by all of the processes accessing or updating a given file are those they would see if only a single copy of the file existed.
- Fault Tolerance
  - At most once invocation semantics.
  - At least once semantics. OK for a server protocol designed for idempotent operations (i.e., duplicated requests do not result in invalid updates to files)
- Security
  - Access Control list = per object, list of allowed users and access allowed to each
  - Capability list = per user, list of objects allowed to access and type of access allowed (could be different for each (user,obj))
  - User Authentication: need to authenticate requesting clients so that access control at the server is based on correct user identifiers.
- Efficiency
  - Whole file v.s. block transfer

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)
  - client kernel process manages call to server
    - retransmit if necessary
    - convert RPC response to file system buffer
    - store in local cache
    - wake up user process
  - back to nfs_read()
    - copy bytes to user memory

NFS Assumptions, goals

- Workgroup file system
  - Small number of clients
  - Very small number of servers
- Single administrative domain
  - All machines agree on "set of users"
    - ...which users are in which groups
  - Client machines run mostly-trusted OS
    - "User #37 says read(...)"

NFS Assumptions, goals

- "Stateless" file server
  - Of course files are "state", but...
  - Server ***exports*** files without creating extra state
    - No list of "who has this file open"
    - No "pending transactions" across crash
  - Result: crash recovery "fast", protocol "simple"

NFS Assumptions, goals

- "Stateless" file server
  - Of course files are "state", but...
  - Server ***exports*** files without creating extra state
    - No list of "who has this file open"
    - No "pending transactions" across crash
  - Result: crash recovery "fast", protocol "simple"
- Some inherently "stateful" operations
  - File locking
  - Handled by "separate service" "outside of NFS"
    - Slick trick, eh?

AFS Assumptions, goals

- Global distributed file system
  - *Uncountable* clients, servers
  - "One AFS", like "one Internet"
    - Why would you want more than one?
- Multiple administrative domains
  - username*@cellname*
  - bmm@andrew.cmu.edu
  - bmm@cs.cmu.edu

AFS Assumptions, goals

- 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
  - Most users on one machine at a time

AFS Assumptions, goals

- Support ***many*** clients
  - 1000 machines could cache a single file
  - Some local, some (very) remote

NFS Namespace

- Constructed by client-side file system mounts
  - mount server1:/usr/local /usr/local
- Group of clients ***can achieve*** common namespace
  - Every machine can execute same mount sequence at boot
  - If system administrators are diligent

NFS Namespace

- "Auto-mount" process based on "maps"
  - /home/dae means server1:/home/dae
  - /home/owens means server2:/home/owens

NFS Security

- Client machine presents credentials
  - user #, list of group #s – from Unix process
- Server accepts or rejects credentials
  - "root squashing"
    - map uid 0 to uid -1 unless client on special machine list
- Kernel process on server "adopts" credentials
  - Sets user #, group vector based on RPC
  - Makes system call (e.g., read()) with those credentials

AFS Namespace

- Assumed-global list of AFS cells
- Everybody sees same files in each cell
  - Multiple servers inside cell invisible to user
- Group of clients ***can achieve*** private namespace
  - Use custom cell database

AFS Security

- Client machine presents Kerberos ticket
  - Allows arbitrary binding of (machine,user) to (realm,principal)
    - bmm on a cs.cmu.edu machine can be bmm@andrew.cmu.edu
    - iff the password is known!
- Server checks against ***access control list***

AFS ACLs

- Apply to directory, not to individual files
- ACL format
  - bmm rlidwka
  - bmm@cs.cmu.edu rl
  - bmm:friends rl
- Negative rights
  - Disallow "joe rl" even though joe is in bmm:friends

AFS ACLs

- AFS ACL semantics are not Unix semantics
  - Some parts obeyed in a vague way
    - Cache manager checks for files being executable, writable
  - Many differences
    - Inherent/good: can name people in different administrative domains
    - "Just different"
      - ACLs are per-directory, not per-file
      - Different privileges: create, remove, lock
  - Not exactly Unix / not tied to Unix

NFS protocol architecture

- root@client executes mount-filesystem RPC
  - returns "file handle" for root of remote file system
- client RPC for each pathname component
  - /usr/local/lib/emacs/foo.el in /usr/local file system
    - h = lookup(root-handle, "lib")
    - h = lookup(h, "emacs")
    - h = lookup(h, "foo.el")
  - Allows disagreement over pathname syntax
    - Look, Ma, no "/"!

NFS protocol architecture

- I/O RPCs are ***idempotent***
  - multiple repetitions have same effect as one
  - lookup(h, "emacs") generally returns same result
  - read(file-handle, offset, length) ⇒ bytes
  - write(file-handle, offset, buffer, bytes)
- RPCs do not create server-memory state
  - no RPC calls for open()/close()
  - write() succeeds (to disk) or fails before RPC completes

NFS file handles

- Goals
  - Reasonable size
  - Quickly map to file on server
  - "Capability"
    - Hard to forge, so possession serves as "proof"
- Implementation (inode #, inode generation #)
  - inode # - small, fast for server to map onto data
  - "inode generation #" - must match value stored in inode
    - "unguessably random" number chosen in create()

NFS Directory Operations

- Primary goal
  - Insulate clients from server directory format
- Approach
  - readdir(dir-handle, cookie, nbytes) returns list
    - name, inode # (for display by ls -l), cookie

Client Caching

- A timestamp-based method is used to validate cached blocks before they are used.
- Each data item in the cache is tagged with
  - Tc: the time when the cache entry was last validated.
  - Tm: the time when the block was last modified at the server.
  - A cache entry at time T is valid if
    - (T-Tc < t) or (Tm client = Tm server).
  - t=freshness interval
    - Compromise between consistency and efficiency
    - Sun Solaris: t is set adaptively between 3-30 seconds for files, 30-60 seconds for directories

Client Caching (Cont'd)

- **When a cache entry is read, a validity check is performed.**
  - **If the first half of validity condition (previous slide) is true, the the second half need not be evaluated.**
  - **If the first half is not true, Tm_{server} is obtained (via *getattr()* to server) and compared against Tm_{client}**
- **When a cached <u>page</u> (not the whole file) is modified, it is marked as dirty and scheduled to be flushed to the server.**
  - **Modified pages are flushed when the file is closed or a *sync* occurs at the client.**
- **Does not guarantee one-copy update semantics.**
- **More details in textbook – please read up**

AFS protocol architecture

- *Volume* = miniature file system
  - One user's files, project source tree, ...
  - Unit of disk quota administration, backup
  - *Mount points* are pointers to other volumes
- Client machine has Cell-Server Database
  - /afs/andrew.cmu.edu is a *cell*
  - *protection server* handles authentication
  - *volume location server* maps volumes to *file servers*

AFS protocol architecture

- Volume location is ***dynamic***
  - Moved between servers transparently to user
- Volumes may have multiple ***replicas***
  - Increase throughput, reliability
  - Restricted to "read-only" volumes
    - /usr/local/bin
    - /afs/andrew.cmu.edu/usr

AFS Callbacks

- Observations
  - Client disks can cache files indefinitely
    - Even across reboots
  - Many files nearly read-only
    - Contacting server on each open() is wasteful
- Server issues ***callback promise***
  - If this file changes in 15 minutes, I will tell you
    - ***callback break*** message
  - 15 minutes of free open(), read() for that client
    - More importantly, 15 minutes of peace for server

AFS file identifiers

- Volume number
  - Each file lives ***in a volume***
  - Unlike NFS "server1's /usr0"
- File number
  - inode # (as NFS)
- "Uniquifier"
  - allows inodes to be re-used
  - Similar to NFS file handle inode generation #s

AFS Directory Operations

- Primary goal
  - Don't overload servers!
- Approach
  - Server stores directory as hash table on disk
  - Client fetches whole directory as if a file
  - ***Client*** parses hash table
    - Directory maps name to fid
  - Client caches directory (indefinitely, across reboots)
    - Server load reduced

AFS access pattern

open("/afs/cs.cmu.edu/service/systypes")

- VFS layer hands off "/afs" to AFS client module
- Client maps cs.cmu.edu to pt & vldb servers
- Client authenticates to pt server
- Client volume-locates root.cell volume
- Client fetches "/" directory
- Client fetches "service" directory
- Client fetches "systypes" file

AFS access pattern

open("/afs/cs.cmu.edu/service/newCSDB")

- VFS layer hands off "/afs" to AFS client module
- Client fetches "newCSDB" file

open("/afs/cs.cmu.edu/service/systypes")

- Assume
  - File is in cache
  - Server hasn't broken callback
  - Callback hasn't expired
- Client can read file with ***no server interaction***

AFS access pattern

- Data transfer is by ***chunks***
  - Minimally 64 KB
  - May be whole-file
- Write***back*** cache
  - Opposite of NFS "every write is sacred"
  - Store chunk back to server
    - When cache overflows
    - On last user close()

AFS access pattern

- Is writeback crazy?
  - Write conflicts "assumed rare"
  - Who needs to see a half-written file?

NFS "rough edges"

- Locking
  - Inherently stateful
    - lock must ***persist across client calls***
      - lock(), read(), write(), unlock()
  - "Separate service"
    - Handled by same server
    - Horrible things happen on server crash
    - Horrible things happen on client crash

NFS "rough edges"

- Some operations not really idempotent
  - unlink(file) returns "ok" ***once***, then "no such file"
  - server caches "a few" client requests
- Cacheing
  - No real consistency guarantees
  - Clients typically cache attributes, data "for a while"
  - No way to know when they're wrong

NFS "rough edges"

- Large NFS installations are brittle
  - Everybody must agree on ***many*** mount points
  - Hard to load-balance files among servers
    - No volumes
    - No atomic moves
- Cross-realm NFS access basically nonexistent
  - No good way to map uid#47 from an unknown host

AFS "rough edges"

- Locking
  - Server refuses to keep a waiting-client list
  - Client cache manager refuses to poll server
  - User program must invent polling strategy
- Chunk-based I/O
  - No real consistency guarantees
  - close() failures surprising

AFS "rough edges"

- ACLs apply to ***directories***
  - "Makes sense" if files will inherit from directories
    - Not always true
  - Confuses users
- Directories inherit ACLs
  - Easy to expose a whole tree accidentally
  - What else to do?
    - No good solution known
    - DFS horror

AFS "rough edges"

- Small AFS installations are punitive
  - 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

Summary - NFS

- Workgroup network file service
- Any Unix machine can be a server (easily)
- Machines can be both client & server
  - My files on my disk, your files on your disk
  - Everybody in group can access all files
- *Serious* trust, scaling problems
- "Stateless file server" model only partial success

Summary – AFS

- Worldwide file system
- Good security, scaling
- Global namespace
- “Professional” server infrastructure per cell
  - Don't try this at home
  - Only ~190 AFS cells (2005-11, also 2003-02)
    - 8 are cmu.edu, ~15 are in Pittsburgh
- “No write conflict” model only partial success

Further Reading

- NFS
  - RFC 1094 for v2 (3/1989)
  - RFC 1813 for v3 (6/1995)
  - RFC 3530 for v4 (4/2003)

Further Reading

- AFS
  - "The ITC Distributed File System: Principles and Design", Proceedings of the 10th ACM Symposium on Operating System Principles, Dec. 1985, pp. 35-50.
  - "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 51-81.
  - IBM AFS User Guide, version 36
  - http://www.cs.cmu.edu/~help/afs/index.html