

Distributed Systems

14. Distributed File Systems

Paul Krzyzanowski
pxk@cs.rutgers.edu

Accessing files

FTP, telnet:

- Explicit access
- User-directed connection to access remote resources

We want more transparency

- Allow user to access remote resources just as local ones

NAS: Network Attached Storage

File service types

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

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 file system

Problem:

- Possible server and network congestion
 - Servers are accessed for duration of file access
 - Same data may be requested repeatedly

Remote File Service

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. implement the file system under the **VFS** layer

Semantics of file sharing

Sequential semantics

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

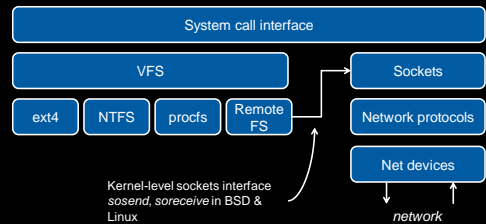
Relax the rules

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

System design issues

Accessing Remote Files

Implement the client module as a file system type under VFS



Stateful or stateless design?

Stateful

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

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

Caching

Hide latency to improve performance for repeated accesses

Four places

- Server's disk
- Server's buffer cache
- Client's buffer cache
- Client's disk

WARNING:
cache consistency
problems

Approaches to caching

- Write-through
 - What if another client reads its own (out-of-date) cached copy?
 - All accesses will require checking with server
 - Or ... server maintains state and sends invalidations
- Delayed writes (write-behind)
 - Data can be buffered locally (watch out for consistency – others won't see updates!)
 - Remote files updated periodically
 - One bulk write is more efficient than lots of little writes
 - Problem: semantics become ambiguous

Approaches to caching

- Read-ahead (prefetch)
 - Request chunks of data before it is needed.
 - Minimize wait when it actually is needed.
- Write on close
 - Admit that we have session semantics.
- Centralized control
 - Keep track of who has what open and cached on each node.
 - Stateful file system with signaling traffic.

NFS

Network File System
Sun Microsystems

NFS Design Goals

- Any machine can be a **client or server**
- Must support **diskless workstations**
- **Heterogeneous systems** must be supported
 - Different HW, OS, underlying file system
- **Access transparency**
 - Remote files accessed as local files through normal file system calls (via VFS in UNIX)
- **Recovery from failure**
 - Stateless, UDP, client retries
- **High Performance**
 - use caching and read-ahead

NFS Design Goals

No support for UNIX file access semantics

Stateless design: file locking is a problem.

All UNIX file system controls may not be available.

NFS Design Goals

Devices

Must support diskless workstations where *every* file is remote.

Remote devices refer back to local devices.

NFS Design Goals

Transport Protocol

Initially NFS ran over UDP using Sun RPC

Why was UDP chosen?

- Slightly faster than TCP
- No connection to maintain (*or lose*)
- NFS is designed for Ethernet LAN environment – relatively reliable
- Error detection but no correction.
 - NFS retries requests

NFS Protocols

Mounting protocol

Request access to exported directory tree

Directory & File access protocol

Access files and directories
(read, write, mkdir, readdir, ...)

Mounting Protocol

- Send pathname to server
- Request permission to access contents

client: parses pathname
contacts server for file handle

- Server returns **file handle**
 - File device #, inode #, instance #

client: create in-memory VFS inode at mount point.
internally points to **node** for remote files
– *Client keeps state, not the server*

Mounting Protocol

static mounting

- **mount** request contacts server

Server: `edit /etc/exports`

Client: `mount fluffy:/users/paul /home/paul`

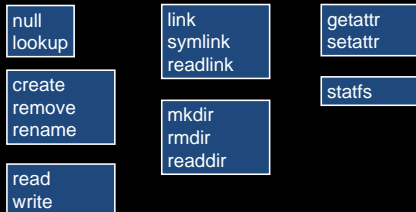
Directory and file access protocol

- First, perform a **lookup** RPC
 - returns **file handle** and attributes
- **lookup is *not* like open**
 - No information is stored on server
- handle passed as a parameter for other file access functions
 - e.g. `read(handle, offset, count)`

Directory and file access protocol

NFS has 16 functions

- (version 2; six more added in version 3)



NFS Performance

- Usually slower than local
- Improve by caching at client
 - Goal: reduce number of remote operations
 - Cache results of *read, readlink, getattr, lookup, readdir*
 - Cache file data at client (buffer cache)
 - Cache file attribute information at client
 - Cache pathname bindings for faster lookups
- Server side
 - Caching is "automatic" via buffer cache
 - All NFS writes are *write-through* to disk to avoid unexpected data loss if server dies

Inconsistencies may arise

Try to resolve by **validation**

- Save timestamp of file
- When file opened or server contacted for new block
 - Compare last modification time
 - If remote is more recent, invalidate cached data

Validation

- Always invalidate data after some time
 - After 3 seconds for open files (data blocks)
 - After 30 seconds for directories
- If data block is modified, it is:
 - Marked *dirty*
 - Scheduled to be written
 - Flushed on file close

Improving read performance

- Transfer data in **large chunks**
 - 8K bytes default
- **Read-ahead**
 - Optimize for sequential file access
 - Send requests to read disk blocks before they are requested by the application

Problems with NFS

- File consistency
- Assumes clocks are synchronized
- Open with append cannot be guaranteed to work
- Locking cannot work
 - Separate lock manager added (stateful)
- No reference counting of open files
 - You can delete a file you (or others) have open!
- Global UID space assumed

Problems with NFS

- File permissions may change
 - Invalidating access to file
- No encryption
 - Requests via unencrypted RPC
 - Authentication methods available
 - Diffie-Hellman, Kerberos, Unix-style
 - Rely on user-level software to encrypt

Improving NFS: version 2

- **User-level lock manager**
 - Monitored locks
 - status monitor: monitors clients with locks
 - Informs lock manager if host inaccessible
 - If server crashes: status monitor reinstates locks on recovery
 - If client crashes: all locks from client are freed
- **NV RAM support**
 - Improves write performance
 - Normally NFS must write to disk on server before responding to client *write* requests
 - Relax this rule through the use of non-volatile RAM

Improving NFS: version 2

- **Adjust RPC retries dynamically**
 - Reduce network congestion from excess RPC retransmissions under load
 - Based on performance
- **Client-side disk caching**
 - **cacheFS**
 - Extend buffer cache to disk for NFS
 - Cache in memory first
 - Cache on disk in 64KB chunks

More improvements... NFS v3

- Updated version of NFS protocol
- Support **64-bit file sizes**
- **TCP support and large-block transfers**
 - UDP caused more problems on WANs (errors)
 - All traffic can be multiplexed on one connection
 - Minimizes connection setup
 - No fixed limit on amount of data that can be transferred between client and server
- Negotiate for optimal **transfer size**
- Server **checks access for entire path** from client

More improvements... NFS v3

- New **commit** operation
 - Check with server after a *write* operation to see if data is committed
 - If *commit* fails, client must **resend** data
 - Reduce number of *write* requests to server
 - Speeds up *write* requests
 - Don't require server to write to disk immediately
- **Return file attributes with each request**
 - Saves extra RPCs

AFS

Andrew File System
Carnegie Mellon University

c. 1986(v2), 1989(v3)

AFS

- Developed at CMU
- Became a commercial spin-off
 - Transarc
- IBM acquired Transarc

Currently open source under IBM Public License

Also:

OpenAFS, Arla, and Linux version

AFS Design Goal

Support information sharing
on a **large** scale

e.g., 10,000+ systems

AFS Assumptions

- Most files are small
- Reads are more common than writes
- Most files are accessed by one user at a time
- Files are referenced in bursts (locality)
 - Once referenced, a file is likely to be referenced again

AFS Design Decisions

Whole file serving

- Send the entire file on *open*

Whole file caching

- Client caches entire file on local disk
- Client writes the file back to server on *close*
 - if modified
 - Keeps cached copy for future accesses

AFS Design

- Each client has an **AFS disk cache**
 - Part of disk devoted to AFS (e.g. 100 MB)
 - Client manages cache in LRU manner
- Clients communicate with **set of trusted servers**
- Each server presents **one, identical name space** to clients
 - All clients access it in the same way
 - Location transparent

AFS Server: cells

- Servers are grouped into administrative entities called **cells**
- **Cell**: collection of
 - Servers
 - Administrators
 - Users
 - Clients
- Each cell is autonomous but cells may cooperate and present users with one **uniform name space**

AFS Server: volumes

Disk partition contains
file and directories

Grouped into **volumes**

Volume

- Administrative unit of organization
 - e.g. user's home directory, local source, etc.
- Each volume is a directory tree (one root)
- Assigned a name and ID number
- A server will often have 100s of volumes

Namespace management

Clients get information via **cell directory server** (Volume Location Server) that hosts the **Volume Location Database** (VLDB)

Goal:
everyone sees the same namespace

`/afs/cellname/path`

`/afs/mit.edu/home/paul/src/try.c`

Internally on the server

- Communication is via **RPC over UDP**
- Access control lists used for protection
 - Directory granularity
 - UNIX permissions ignored (except execute)

AFS cache coherence

On **open**:

- Server sends entire file to client
and provides a **callback promise**:
- *It will notify the client when any other process modifies the file*

AFS cache coherence

If a client modified a file:

- Contents are **written to server on close**

When a server gets an update:

- it **notifies all clients** that have been issued the callback promise
- Clients invalidate cached files

AFS cache coherence

If a client was down, on startup:

- Contact server with timestamps of all cached files to decide whether to invalidate

If a process has a file open, it continues accessing it even if it has been invalidated

- Upon close, contents will be propagated to server

AFS: Session Semantics
(vs. sequential semantics)

AFS: replication and caching

- Read-only volumes may be replicated on multiple servers
- Whole file caching not feasible for huge files
 - AFS caches in 64KB chunks (by default)
 - Entire directories are cached
- Advisory locking supported
 - Query server to see if there is a lock

AFS summary

Whole file caching

- offers dramatically reduced load on servers

Callback promise

- keeps clients from having to check with server to invalidate cache

AFS summary

AFS benefits

- AFS scales well
- Uniform name space
- Read-only replication
- Security model supports mutual authentication, data encryption

AFS drawbacks

- Session semantics
- Directory based permissions
- Uniform name space

SMB

Server Message Blocks

Microsoft

c. 1987

SMB Goals

- File sharing protocol for Windows 95/98/NT/200x/ME/XP/Vista/Windows 7
- Protocol for sharing:
 - Files, devices, communication abstractions (named pipes), mailboxes
- Servers: make file system and other resources available to clients
- Clients: access shared file systems, printers, etc. from servers

Design Priority:

locking and consistency over client caching

SMB Design

- Request-response protocol
 - Send and receive **message blocks**
 - name from old DOS system call structure
 - Send *request* to server (machine with resource)
 - Server sends response
- Connection-oriented protocol
 - Persistent connection – “session”
- Each message contains:
 - Fixed-size header
 - Command string (based on message) or reply string

Message Block

- Header: [fixed size]
 - Protocol ID
 - Command code (0..FF)
 - Error class, error code
 - Tree ID – unique ID for resource in use by client (handle)
 - Caller process ID
 - User ID
 - Multiplex ID (to route requests in a process)
- Command: [variable size]
 - Param count, params, #bytes data, data

SMB Commands

- Files
 - Get disk attr
 - create/delete directories
 - search for file(s)
 - create/delete/rename file
 - lock/unlock file area
 - open/commit/close file
 - get/set file attributes

SMB Commands

- **Print-related**
 - Open/close spool file
 - write to spool
 - Query print queue
- **User-related**
 - Discover home system for user
 - Send message to user
 - Broadcast to all users
 - Receive messages

Protocol Steps

- Establish connection

Protocol Steps

- Establish connection
- Negotiate protocol
 - *negprot* SMB
 - Responds with version number of protocol

Protocol Steps

- Establish connection
- Negotiate protocol
- Authenticate/set session parameters
 - Send *sesssetupX* SMB with username, password
 - Receive NACK or UID of logged-on user
 - UID must be submitted in future requests

Protocol Steps

- Establish connection
- Negotiate protocol - *negprot*
- Authenticate - *sesssetupX*
- Make a connection to a resource (similar to *mount*)
 - Send *tcon* (tree connect) SMB with name of shared resource
 - Server responds with a **tree ID** (TID) that the client will use in future requests for the resource

Protocol Steps

- Establish connection
- Negotiate protocol - *negprot*
- Authenticate - *sesssetupX*
- Make a connection to a resource – *tcon*
- Send open/read/write/close/... SMBs

CODA

CConstant Data Availability
Carnegie-Mellon University

c. 1990-1992

CODA Goals

Descendant of AFS
CMU, 1990-1992

Goals

Provide better support for replication than AFS
- support shared read/write files

Support mobility of PCs

Mobility

- Provide **constant** data availability in disconnected environments
- Via **hoarding** (user-directed caching)
 - Log updates on client
 - Reintegrate on connection to network (server)
- Goal: Improve fault tolerance

Modifications to AFS

- Support replicated file volumes
- Extend mechanism to support disconnected operation
- A **volume can be replicated** on a group of servers
 - **Volume Storage Group (VSG)**

Volume Storage Group

- Volume ID used in the File ID is
 - **Replicated volume ID**
- One-time lookup
 - Replicated volume ID → list of servers and *local* volume IDs
 - Cache results for efficiency
- Read files from *any* server
- Write to **all available servers**

Disconnection of volume servers

- AVSG:** Available Volume Storage Group
- Subset of VSG

What if some volume servers are down?

On first download, contact everyone you can and get a version timestamp of the file

Disconnected servers

If the client detects that some servers have old versions

- Some server resumed operation
- Client initiates a **resolution process**
 - Updates servers: notifies server of stale data
 - Resolution handled entirely by servers
 - Administrative intervention may be required (if conflicts)

AVSG = \emptyset

- If no servers are available
 - Client goes to **disconnected operation mode**
- If file is not in cache
 - Nothing can be done... fail
- Do not report failure of update to server
 - Log update locally in **Client Modification Log** (CML)
 - User does not notice

Reintegration

Upon reconnection

- Commence **reintegration**

Bring server up to date with **CML log playback**

- Optimized to send latest changes

Try to resolve conflicts automatically

- Not always possible

Support for disconnection

Keep important files up to date

- Ask server to send updates if necessary

Hoard database

- Automatically constructed by monitoring the user's activity
- And user-directed prefetch

CODA summary

- Session semantics as with AFS
- Replication of read/write volumes
 - Client-driven reintegration
- Disconnected operation
 - Client modification log
 - Hoard database for needed files
 - User-directed prefetch
 - Log replay on reintegration

DFS
Distributed File System
Open Group

DFS

- Part of Open Group's Distributed Computing Environment
- Descendant of AFS - AFS version 3.x
- Development stopped c. 2005
- Assume (like AFS):
 - Most file accesses are sequential
 - Most file lifetimes are short
 - Majority of accesses are whole file transfers
 - Most accesses are to small files

DFS Goals

Use **whole file caching** (like original AFS)

But...

session semantics are hard to live with

Create a **strong consistency** model

DFS Tokens

Cache consistency maintained by **tokens**

Token:

- Guarantee from server that a client can perform certain operations on a cached file

DFS Tokens

- **Open** tokens
 - Allow token holder to open a file.
 - Token specifies access (read, write, execute, exclusive-write)
- **Data** tokens
 - Applies to a byte range
 - *read* token - can use cached data
 - *write* token - write access, cached writes
- **Status** tokens
 - *read*: can cache file attributes
 - *write*: can cache modified attributes
- **Lock** token
 - Holder can lock a byte range of a file

Living with tokens

- **Server grants and revokes tokens**
 - Multiple *read* tokens OK
 - Multiple *read* and a *write* token or multiple *write* tokens not OK if byte ranges overlap
 - Revoke all other *read* and *write* tokens
 - Block new request and send revocation to other token holders

DFS design

- Token granting mechanism
 - Allows for long term caching and strong consistency
- Caching sizes: 8K – 256K bytes
- Read-ahead (like NFS)
 - Don't have to wait for entire file
- File protection via ACLs
- Communication via authenticated RPCs

DFS Summary

Essentially AFS v2 with server-based token granting

- Server keeps track of who is reading and who is writing files
- Server must be contacted on each *open* and *close* operation to request token

CIFS

Common Internet File System
Microsoft, Compaq, ...

c. 1995?

SMB evolves

SMB was reverse-engineered

- **samba** under Linux

Microsoft released protocol to X/Open in 1992

Microsoft, Compaq, SCO, others joined to develop an enhanced public version of the SMB protocol:

**Common Internet File System
(CIFS)**

Original Goals

- Heterogeneous HW/OS to request file services over network
- Based on SMB protocol
- Support
 - Shared files
 - Byte-range locking
 - Coherent caching
 - Change notification
 - Replicated storage
 - Unicode file names

Original Goals

- Applications can register to be notified when file or directory contents are modified
- Replicated virtual volumes
 - For load sharing
 - Appear as one volume server to client
 - Components can be moved to different servers without name change
 - Use *referrals*
 - Similar to AFS

Original Goals

- Batch multiple requests to minimize round-trip latencies
 - Support wide-area networks
- Transport independent
 - But need reliable connection-oriented message stream transport
- DFS support (compatibility)

Caching and Server Communication

- Increase effective performance with
 - Caching
 - Safe if multiple clients reading, nobody writing
 - read-ahead
 - Safe if multiple clients reading, nobody writing
 - write-behind
 - Safe if only one client is accessing file
- Minimize times client informs server of changes

Oplocks

Server grants **opportunistic locks (oplocks)** to client

- Oplock tells client how/if it may cache data
- Similar to DFS tokens (but more limited)

Client must request an oplock

- oplock may be
 - Granted
 - Revoked
 - Changed by server

Level 1 oplock (exclusive access)

- Client can open file for exclusive access
- Arbitrary caching
- Cache lock information
- Read-ahead
- Write-behind

If another client opens the file, the server has former client *break its oplock*:

- Client must send server any lock and write data and acknowledge that it does not have the lock
- Purge any read-aheads

Level 2 oplock (one writer)

- Level 1 oplock is replaced with a Level 2 lock if another process tries to read the file
- Request this if expect others to read
- Multiple clients may have the same file open as long as none are writing
- Cache reads, file attributes
 - Send other requests to server
- Level 2 oplock revoked if another client opens the file for writing

Batch oplock (remote open even if local closed)

- Client can keep file open on server even if a local process that was using it has closed the file
 - Exclusive R/W open lock + data lock + metadata lock
- Client requests batch oplock if it expects programs may behave in a way that generates a lot of traffic (e.g. accessing the same files over and over)
 - Designed for Windows batch files
- Batch oplock revoked if another client opens the file

Filter oplock (allow preemption)

- Open file for read or write
- Allow clients with *filter oplock* to be suspended while another process preempted file access.
 - E.g., indexing service can run and open files without causing programs to get an error when they need to open the file
 - Indexing service is notified that another process wants to access the file.
 - It can abort its work on the file and close it or finish its indexing and then close the file.

No oplock

- All requests must be sent to the server
- Can work from cache only if byte range was locked by client

Naming

- Multiple naming formats supported:
 - N:\junk.doc
 - \\myserver\users\paul\junk.doc
 - file://grumpy.pk.org/users/paul/junk.doc

Microsoft Dfs

- "Distributed File System"
 - Provides a logical view of files & directories
- Each computer hosts **volumes**
 - \\servername\dfsname

Each Dfs tree has one root volume and one level of leaf volumes.
- A volume can consist of multiple shares
 - Alternate path: load balancing (read-only)
 - Similar to Sun's automounter
- Dfs = SMB + naming/ability to mount server shares on other server shares

Redirection

- A share can be replicated (read-only) or moved through **Microsoft's Dfs**
- Client opens old location:
 - Receives **STATUS_DFS_PATH_NOT_COVERED**
 - Client requests referral: **TRANS2_DFS_GET_REFERRAL**
 - Server replies with new server

CIFS Summary

- A "standard" SMB
- Oplocks mechanism supported in base OS: Windows NT/XP/Vista/7/8, 20xx
- Oplocks offer flexible control for distributed consistency
- Dfs offers namespace management

NFS version 4
Network File System
Sun Microsystems

NFS version 4 enhancements

- Stateful server
- Compound RPC
 - Group operations together
 - Receive set of responses
 - Reduce round-trip latency
- Stateful open/close operations
 - Ensures atomicity of share reservations for windows file sharing (CIFS)
 - Supports exclusive creates
 - Client can cache aggressively

NFS version 4 enhancements

- create, link, open, remove, rename
 - Inform client if the directory changed during the operation
- Strong security
 - Extensible authentication architecture
- File system replication and migration
 - To be defined
- No concurrent write sharing or distributed cache coherence

NFS version 4 enhancements

- Server can delegate specific actions on a file to enable more aggressive client caching
 - Similar to CIFS oplocks
- Callbacks
 - Notify client when file/directory contents change

Other
(less conventional)
Distributed File Systems

WebDAV

- *Not a file system – but a network protocol*
- Web-based Distributed Authoring [and Versioning] RFC 2518
- **Extension to HTTP** to make the Web writable
- New HTTP Methods
 - **PROPFIND**: retrieve properties from a resource, including a collection (directory) structure
 - **PROPPATCH**: change/delete multiple properties on a resource
 - **MKCOL**: create a collection (directory)
 - **COPY**: copy a resource from one URI to another
 - **MOVE**: move a resource from one URI to another
 - **LOCK**: lock a resource (shared or exclusive)
 - **UNLOCK**: remove a lock

Who uses WebDAV?

- File systems:
 - davfs2: Linux file system driver to mount a DAV server as a file system
 - Coda kernel driver and neon for WebDAV communication
 - Native filesystem support in OS X (since 10.0)
 - Microsoft web folders (since Windows 98)
- Apache HTTP server
- Apple iCal & iDisk
- Jakarta Slide & Tomcat
- KDE Desktop
- Microsoft Exchange & IIS
- SAP NetWeaver
- Many others...
- Check out webdav.org

An *ad hoc* file system using Gmail

- Gmail file system (Richard Jones, 2004; Dave Hansen, 2009)
 - User-level
 - Python application
 - FUSE userland file system interface
 - Supports
 - Read, write, open, close, stat, symlink, link, unlink, truncate, rename, directories
 - Each message represents a file
 - Subject headers contain:
 - File system name, filename, pathname, symbolic link info, owner ID, group ID, size, etc.
 - File data stored in attachments
 - Files can span multiple attachments
- See <http://sr71.net/projects/gmailfs/>

Client-server file systems

- Central servers
 - Point of congestion, single point of failure
- Alleviate somewhat with replication and client caching
 - E.g., Coda
 - Limited replication can lead to congestion
 - Separate set of machines to administer
- But ... user systems have LOTS of disk space
 - (500 GB disks commodity items @ \$45)

Serverless file systems?

- Use workstations cooperating as peers to provide file system service
- Any machine can share/cache/control any block of data

Prototype serverless file system

- **xFS** from Berkeley demonstrated to be scalable
- Others:
 - See Fraunhofer FS (www.fhgfs.com)

The End