Distributed File Systems
Case Studies

NFS
Network File System
Sun Microsystems

c. 1985

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 migration transparency

If resource moves to another server, client must remount resource.

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 UDP?

- 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, …)

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-code vnode at mount point.
(points to inode for local files)
points to **rnode** for remote files
- stores state on client

Mounting Protocol

static mounting

– *mount* request contacts server

Server: `/etc/exports`
Client: `mount fluffy:/users/paul /home/paul`

Directory and file access protocol

- Initially perform ***lookup*** RPC
  - returns **file handle** and attributes
- *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)

null
lookup

create
remove
rename

read
write

link
symlink
readlink

mkdir
rmdir
readdir

getattr
setattr

statfs

Accessing files

Parse **component at a time** via *namei*

- At each point, see if mount point
  - Yes? Continue on the mounted file system
  - Remote? Perform NFS RPC *lookup*

- Ensures that **..** is processed locally and future mount points are processed
- Final lookup returns handle
- Create in-core vnode, rnode

Accessing files

Application can now access file

file descriptor → in-core vnode (VFS layer)
↓
in-core rnode (NFS client)

Perform NFS read/write RPCs using state in rnode

RPCs include user ID and group ID
- security hole

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 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 block is modified
  - Marked *dirty*
  - Scheduled to be written
  - Flushed on 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

- No reference counting of open files
  - You can delete a file you (or others) have open!
- Common practice
  - Create temp file, delete it, continue access
  - Sun's hack:
    - If same process with open file tries to delete it
    - Move to temp name
    - Delete on close

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
- 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

Improving NFS: version 2

- Enhanced 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

The automounter

Problem with mounts

- If a client has many remote resources mounted, boot-time can be excessive
- Each machine has to maintain its own name space
  - Painful to administer on a large scale

Automounter

- Allows administrators to create a global name space
- Support *on-demand* mounting

Automounter

- Solve static mounting problems with…

  the **automounter**
- Mount and unmount in response to client demand
  - Set of directories are associated with a local directory
  - None are mounted initially
  - When local directory is **referenced**
    - OS sends a message to **each** server
    - First reply wins
  - Attempt to unmount every 5 minutes

Automounter maps

Automounter maps

- Provide mapping:

  client pathname → server file system
- Automounter converts maps into mounts that are added to the client's file system tree

Alternative to automounter maps

- X/Open Federated Naming Specification (XFN)
- Name service
- All resources under **/xfn/*pathname***

Automounter maps

Example:

automount /usr/src srcmap

srcmap contains:

```
cmd       -ro  doc:/usr/src/cmd
kernel         -rw  frodo:/release/src \
               bilbo:/library/source/kernel
lib       -ro  sneezy:/usr/local/lib
```

Access **/usr/src/cmd**: request goes to doc

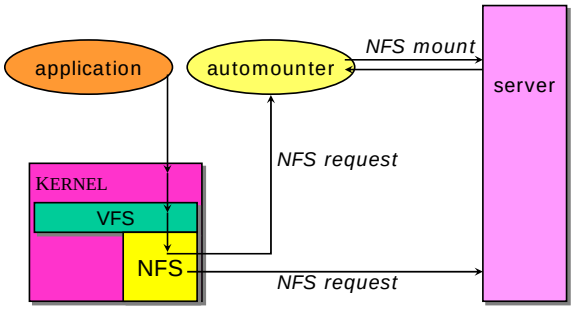
Access **/usr/src/kernel**:

ping frodo and bilbo, mount first response

Inside the automounter

- automounter speaks NFS and mount protocols
- Performs NFS mount of given directory
  - Mount has address and port number of automounter, *not* nfsd server
- All NFS requests go to automounter
- When automounter gets a *lookup* RPC of a directory in the map
  - Mounts the directory under a directory in **/tmp_mnt**
  - Returns a symbolic link to the directory

The automounter



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
- Server checks access for entire path from client

More improvements… NFS v3

- Negotiate for optimal transfer size
- 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

RFS

Remote File Sharing

AT&T Unix System V

c. 1986

Design Goals

NFS

- Support lowest common denominator in file system interfaces

RFS

- Support *every* feature of the UNIX System V file system
  - More complex interface
  - Harder to port to other operating systems/file systems

Design Goals

NFS

- Connectionless
- Ambiguous semantics
- Stateless

RFS

- Connection-oriented, stateful
- Server keeps track of
  - Which files are open
  - Which data blocks are cached on each client
- Provides for strong consistency ("UNIX semantics")

Resource access

NFS

- Client has to know which directories are exported by which server
- Export list is a file in `/etc/exports` on server

- RFS
  - Name server
  - Server advertises resource
  - *mount* command contacts name server

Resource access

NFS

- Devices on remote file system refer to *local* devices

RFS

- Supports remote devices

What do you get?

- File and record locking works
- Open with append works
- Strong consistency

But…

- Sensitive to server crashes
- UNIX System V file system only

AFS

Andrew File System

Carnegie-Mellon University

c. 1986(v2), 1989(v3)

AFS

- Developed at CMU
- Commercial spin-off
  - Transarc
- IBM acquired Transarc
- Currently open source under IBM Public License

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: volumes

Disk partition contains
file and directories
grouped into **volumes**

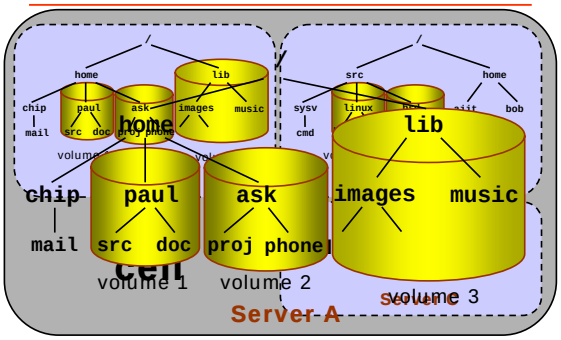
Volume

- 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 100's of volumes

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**

Files, directories, volumes, cells



Namespace management

Clients get information via cell directory server

Goal:
everyone sees the same namespace

/afs/cellname/path

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

Internally on the server…

Each file and directory identified by three 32-bit numbers:

File ID = {volumeID, vnodeID, uniquifier}

client caches server address of volume but server keeps mapping. If volume moves to another server, server forwards the request

vnodeID: "handle" on server

Unique number to ensure that vnode IDs are not reused

Internally on the server

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

Authentication and access

Kerberos authentication
- Trusted third party issues tickets
- Mutual authentication

Before a user can access files
- Authenticate to AFS with *klog* command
  - "Kerberos login" – centralized authentication
- Get a token (ticket) from Kerberos
- Present it with each file access

Unauthorized users have id of
`system:anyuser`

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

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

CODA

COnstant 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?
- Each file copy has a version stamp
- Before fetching a file
  - Client requests version stamps from all available servers

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
  - handled entirely by servers
  - Administrative intervention may be required (if conflicts)

AVSG = Ø

- 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 (AFS v3.x)

Distributed File System

Open Group

DFS

- Part of Open Group's Distributed Computing Environment
- Descendant of AFS - AFS version 3.x

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 (UNIX semantics)

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 <u>and</u> 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 (AFS v3) Sumary

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

SMB

Server Message Blocks

Microsoft

c. 1987

SMB Goals

- File sharing protocol for Windows 95/98/NT/2000/ME/XP
- 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

<u>Design Priority</u>:

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
  - 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

Locating Services

- Clients can be configured to know about servers
- Each server broadcasts info about its presence
  - Clients listen for broadcast
  - Build list of servers
- Fine on a LAN environment
  - Does not scale to WANs
  - Microsoft introduced *browse servers* and the *Windows Internet Name Service* (WINS)
  - or … explicit pathname to server

Security

- Share level
  - Protection per "share" (resource)
  - Each share can have password
  - Client needs password to access all files in share
  - Only security model in early versions
  - Default in Windows 95/98
- User level
  - protection applied to individual files in each share based on access rights
  - Client must login to server and be authenticated
  - Client gets a UID which must be presented for future accesses

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)

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

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

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
- Enhancement of DFS tokens

Client must request an oplock
- oplock may be
  - Granted
  - Revoked
  - Changed by server

Level 1 oplock

- 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

- Request 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

- 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

- Open file for read or write
- Locks file so other clients cannot open for write or delete
  - All clients can share read access
- Allow other clients to perform non-intrusive (read) operations

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

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

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
- SMB + ability to mount server shares on other server shares

CIFS Summary

- Proposed standard has not yet fully materialized
  - Future direction uncertain
- Oplocks mechanism supported in base OS: Windows NT, 2000, XP
- Oplocks offer flexible control for distributed consistency

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

WebDAV

- *Not a file system - just a 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: locak a resource (shared or exclusive)
  - UNLOCK: remove a lock

Who uses WebDAV?

- Davfs2: Linux file system driver to mount a DAV server as a file system
  - Coda kernel driver and neon for WebDAV communication
- Apache HTTP servier
- 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)
- User-level
  - Python application
  - FUSE userland file system interface
- Supports
  - Read, write, open, close, stat, symlink, link, unlink, truncate, rename, directories
- Metadata stored in the subject of emails sent by the gmail user
- File data stored in attachments
  - Files can span multiple attachments
- Subject lines have a file system name.

File servers

- 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
  - (120 GB disks commodity items @ $120)

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

Motivation

Fast switched LANs
- Scalable bandwidth, lower latency

Expanding demands of user
- Need increased performance for multimedia

Limitations of central servers

Local disk caching

Made sense on Ethernet (non-switched) networks

- On today's LANs, fetching data from local disk can be an order of magnitude slower than fetching it from a server's memory or remote striped disk

Design

- Build a truly distributed network file system
  - No central bottleneck
- Distribute control processing across the system on a per-file granularity
- Distribute data storage across storage server disks with software RAID
- Cooperative caching
  - Use portions of client memory as a large global file cache
- Built for a restricted/trusted environment
  - Machines communicate over a fast network
  - Trust one another's kernels to enforce security

The End.