Distributed Systems

14. Network File Systems

Paul Krzyzanowski

Rutgers University

Fall 2016

Accessing files

File sharing with socket-based programs

HTTP, 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 models

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?

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

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.
- Need to hide or lock file under modification from other clients
- Last process to modify the file wins.

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

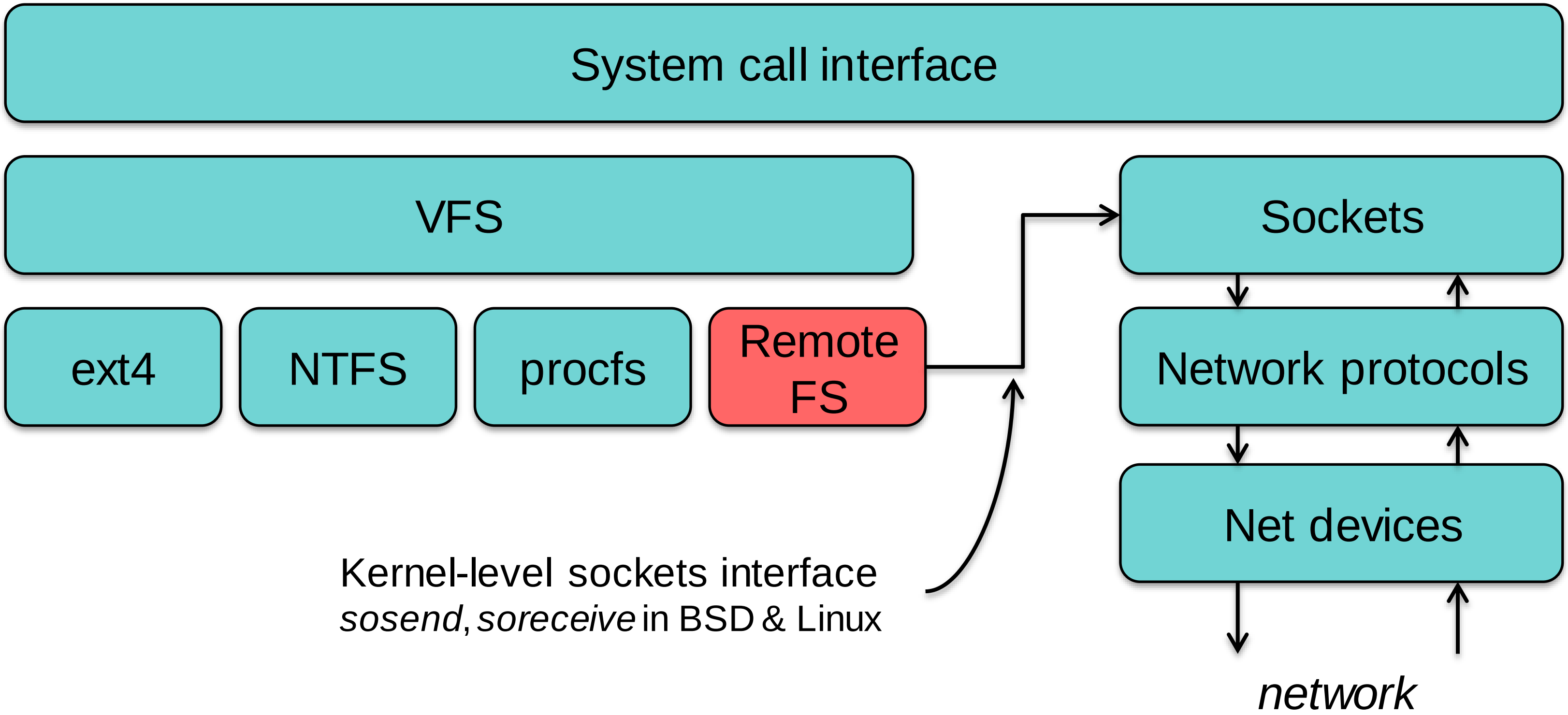
System design issues

System Design Issues

- Transparency
  - Integrated into OS or access via APIs?
- Consistency
  - What happens if more than one user accesses the same file?
  - What if files are replicated across servers?
- Security
- Reliability
  - What happens when the server or client dies?
- State
  - Should the server keep track of clients between requests?

Accessing Remote Files

For maximum transparency, 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

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:
risk of 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 wire 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
  - Device files refer back to local drivers
- Heterogeneous systems
  - Not 100% for all UNIX system call options
- Access transparency: normal file system calls
- Recovery from failure:
  - Stateless, UDP, client retries
  - Stateless → no locking!
- High Performance
  - use caching and read-ahead

NFS Design Goals

Transport Protocol

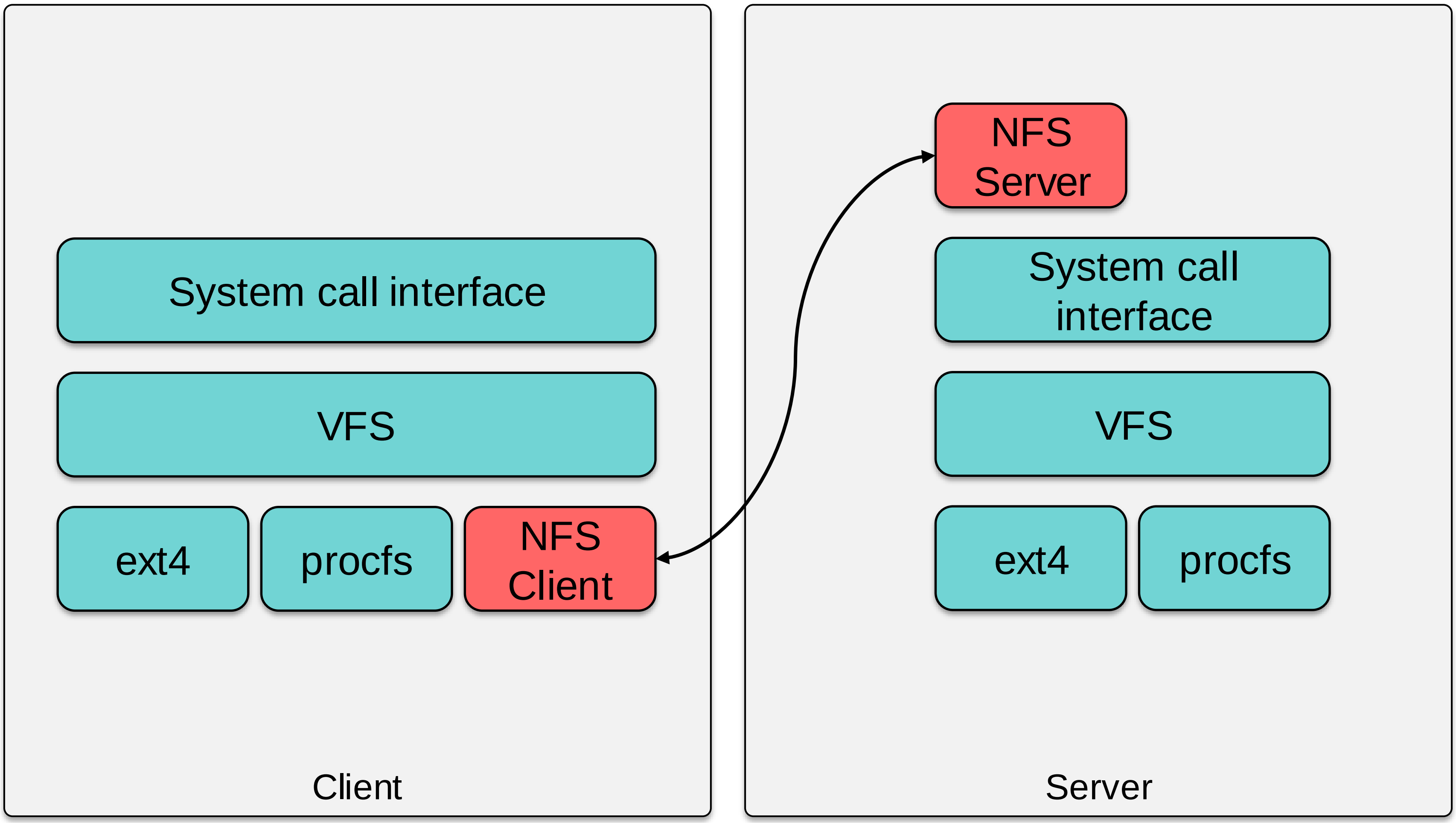
Initially NFS ran over UDP using Sun RPC

Why was UDP chosen?

- Slightly faster than TCP
- No connection to maintain (or lose)
- NFS is designed for Ethernet LAN environment – relatively reliable
- UDP has error detection (drops bad packets) but no retransmission

  NFS retries lost RPC requests

VFS on client; Server accesses local file system



NFS Protocols

Mounting protocol

Request access to exported directory tree

Directory & File access protocol

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

Mounting Protocol

static mounting

– <u>mount</u> request contacts server

Server: edit `/etc/exports`

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

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 ***rnode*** for remote files - *Client keeps state, not the server*

Directory and file access protocol

- First, perform a *lookup* RPC
  - returns file handle and attributes

- <u>lookup is ***not*** like *open*</u>
  - 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	link symlink readlink	getattr setattr
create remove rename	mkdir rmdir readdir	statfs
read write		

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

- 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 (*that used to be a large chunk!*)
- 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 (but this adds stateful behavior)
- 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: introduces *state* at server (but runs as a separate user-level process)
    - 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

Support Larger Environments: 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

- Alternative to static mounting
- 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
  - Describes how file systems below a mount point are mounted

Automounter maps

Example:

automount /usr/src srcmap

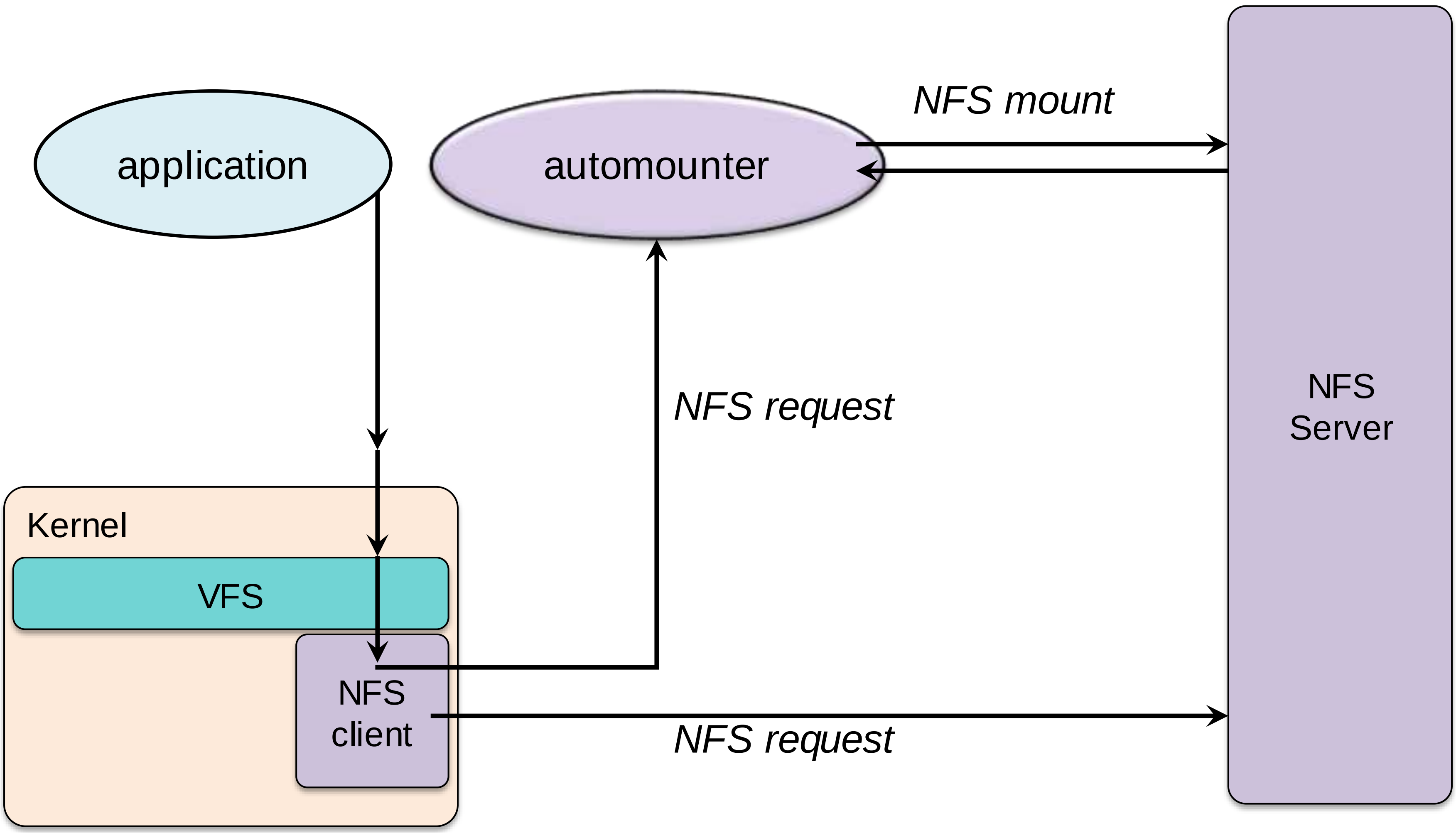
srcmap contains:

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

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

Access **/usr/src/kernel**:
ping frodo and bilbo, mount first response

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
- 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 to get attributes for validation

AFS

Andrew File System

Carnegie Mellon University

c. 1986(v2), 1989(v3)

AFS

- Design Goal
  - Support information sharing on a *large* scale
    e.g., 10,000+ clients

- History
  - Developed at CMU
  - Became a commercial spin-off: Transarc
  - IBM acquired Transarc
  - Open source under IBM Public License
  - OpenAFS (openafs.org)

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

Communication with 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*

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 invalidate
– 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

- Referrals
  - An administrator may move a volume to another server
  - If a client accesses the old server, it gets a *referral* to the new one

AFS key concepts

- Single global namespace
  - Built from a collection of volumes
  - Referrals for moved volumes
  - Replication of read-only volumes
- Whole-file caching
  - Offers dramatically reduced load on servers
- Callback promise
  - Keeps clients from having to poll the 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

1. Provide better support for replication than AFS
   – support shared read/write files

2. Support mobility of PCs

Mobility

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

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)**
- Replicated volumes
  - Volume ID used to identify files is a 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**

Disconnected volume servers

AVSG: Accessible 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

Reconnecting 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 = Ø

- If no servers are accessible
  - 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
  - Clients do the work of writing replicas (extra bandwidth)
  - Client-detected reintegration
- Disconnected operation
  - Client modification log
  - Hoard database for needed files
    - User-directed prefetch
  - Log replay on reintegration

DFS (AFS v3)
Distributed File System

DFS

- Goal
  - AFS: scalable performance but session semantics were hard to live with
  - Create a file system similar to AFS but with a strong consistency model
- History
  - 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

Caching and Server Communication

- Increase effective performance with
  - Caching data that you read
    - Safe if multiple clients reading, nobody writing
  - read-ahead
    - Safe if multiple clients reading, nobody writing
  - write-behind (delaying writes to the server)
    - Safe if only one client is accessing file

- Goal:
  - Minimize times client informs server of changes, use fewer messages with more data vs. lots of messages with little data

DFS Tokens

Cache consistency maintained by **tokens**

Token

- Guarantee from server that a client can perform certain operations on a cached file
- Server grants & revokes 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* tokens
  - 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 key points

- Caching
  - 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 before using it as with AFS
- File protection via access control lists (ACLs)
- Communication via authenticated RPCs
- 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 9x/NT/20xx/ME/XP/Vista/Windows 7/Windows 8/Windows 10 …
- 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 attributes
  - create/delete directories
  - search for file(s)
  - create/delete/rename file
  - lock/unlock file area
  - open/commit/close file
  - get/set file attributes
- 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

SMB Evolves

Common Internet File System (1996)

SMB 2 (2006)

SMB 3 (2012)

SMB Evolves

- History
  - SMB was reverse-engineered for non-Microsoft platforms
    - samba.org
  - Microsoft released SMB protocol to X/Open in 1992
  - Common Internet File System (CIFS)
    - SMB as implemented in 1996 for Windows NT 4.0

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 by the server at some future time
  - Changed by server at some future time

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 (multiple readers)

- Level 1 oplock is replaced with a Level 2 lock if another process tries to read the file
- 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 any 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 is exclusive: one client only
  - 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.

Leases (SMB ≥ 2.1; Windows ≥ 7)

- Same purpose as oplock: control caching
- Lease types
  - Read-cache (R) lease: cache results of *read*; can be shared
  - Write-cache (W) lease: cache results of writes; exclusive
  - Handle-cache (H) lease: cache file handles; can be shared
    - Optimizes re-opening files
- Leases can be combined: R, RW, RH, RWH
- Leases define oplocks:
  - *Read oplock* (R) – essentially same as Level 2
  - Read-handle (RH) – essentially same as Batch
  - Read-write (RW)– essentially the same as Level 1
  - Read-write-handle (RWH)

See https://blogs.msdn.microsoft.com/openspecification/2009/05/22/client-caching-features-oplock-vs-lease/

No oplock

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

Microsoft Dfs

- "Distributed File System"
  - Provides a logical view of files & directories
  - Organize multiple SMB shares into one file system
  - Provide location transparency & redundancy

- 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 via referrals

- 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

SMB (CIFS) Summary

- Stateful model with strong consistency
- Oplocks offer flexible control for distributed consistency
  - Oplocks mechanism supported in base OS: Windows NT/XP/Vista/7/8/9/10, 20xx
- Dfs offers namespace management

SMB2 and SMB3

- SMB was…
  - Chatty: common tasks often required multiple round trip messages
  - Not designed for WANs
- SMB2
  - Protocol dramatically cleaned up
  - New capabilities added
  - SMB2 is the default network file system in Apple Mavericks (10.9)
- SMB3
  - Added RDMA and multichannel support; end-to-end encryption
    - RDMA = Remote DMA (Direct Memory Access)
  - Windows 8 / Windows Server 2012: SMB 3.0
  - SMB3 was default on Apple Yosemite (10.10)

SMB2 Additions

- Reduced complexity
  - From >100 commands to 19
- Pipelining support
  - Send additional commands before the response to a previous one is received
  - Credit-based flow control
    - Goal: keep more data in flight and use available network bandwidth
    - Server starts with a small # of "credits" and scales up as needed
    - Server sends credits to client
    - Client needs credits to send a message and decrements credit balance
    - Allows server to control buffer overflow
    - Note: TCP uses congestion control, which yields to data loss and wild oscillations in traffic intensity

SMB2 Additions

- Compounding support
  - Avoid the need to have commands that combine operations
  - Send an arbitrary set of commands in one request
  - E.g., instead of *RENAME*:
    - CREATE (create new file or open existing)
    - SET_INFO
    - CLOSE
- Larger reads/writes
- Caching of folder & file properties
- "Durable handles"
  - Allow reconnection to server if there was a temporary loss of connectivity

Benefits

- Transfer 10.7 GB over 1 Gbps WAN link with 76 ms RTT
  - SMB: 5 hours 40 minutes: rate = 0.56 MB/s
  - SMB2: 7 minutes, 45 seconds: rate = 25 MB/s

SMB3

- Key features
  - Multichannel support for network scaling
  - Transparent network failover
  - “SMBDirect” – support for Remote DMA in clustered environments
    - Enables direct, low-latency copying of data blocks from remote memory without CPU intervention
  - Direct support for virtual machine files
    - Volume Shadow Copy
    - Enables volume backups to be performed while apps continue to write to files.
  - End-to-end encryption

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
  - Mirror servers can be configured
  - Administrator can distribute data across multiple servers
  - Clients don't need to know where the data is: server will send referrals
- No concurrent write sharing or distributed cache coherence

NFS version 4 enhancements

- Stateful locking
  - Clients inform servers of lock requests
  - Locking is lease-based; clients must renew leases
- Improved caching
  - Server can delegate specific actions on a file to enable more aggressive client caching
  - Close-to-open consistency
    - File changes propagated to server when file is closed
    - Client checks timestamp on open to avoid accessing stale cached copy
  - Similar to CIFS oplocks
    - Clients must disable caching to share files
- Callbacks
  - Notify client when file/directory contents change

Review: Core Concepts

- NFS
  - RPC-based access
- AFS
  - Long-term caching
- DFS
  - AFS + tokens for consistency and efficient caching
- CODA
  - Read/write replication & disconnected operation
- SMB/CIFS
  - RPC-like access with strong consistency
  - Oplocks (tokens) to support caching
  - Dfs: add-on to provide a consistent view of volumes (AFS-style)

The End