Operating Systems

19. Network Attached Storage

Paul Krzyzanowski

Rutgers University

Spring 2015

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

Remote File Service Components

Remote file access network protocol

- Request access to, look up, and access remote files and directories

Remote file server

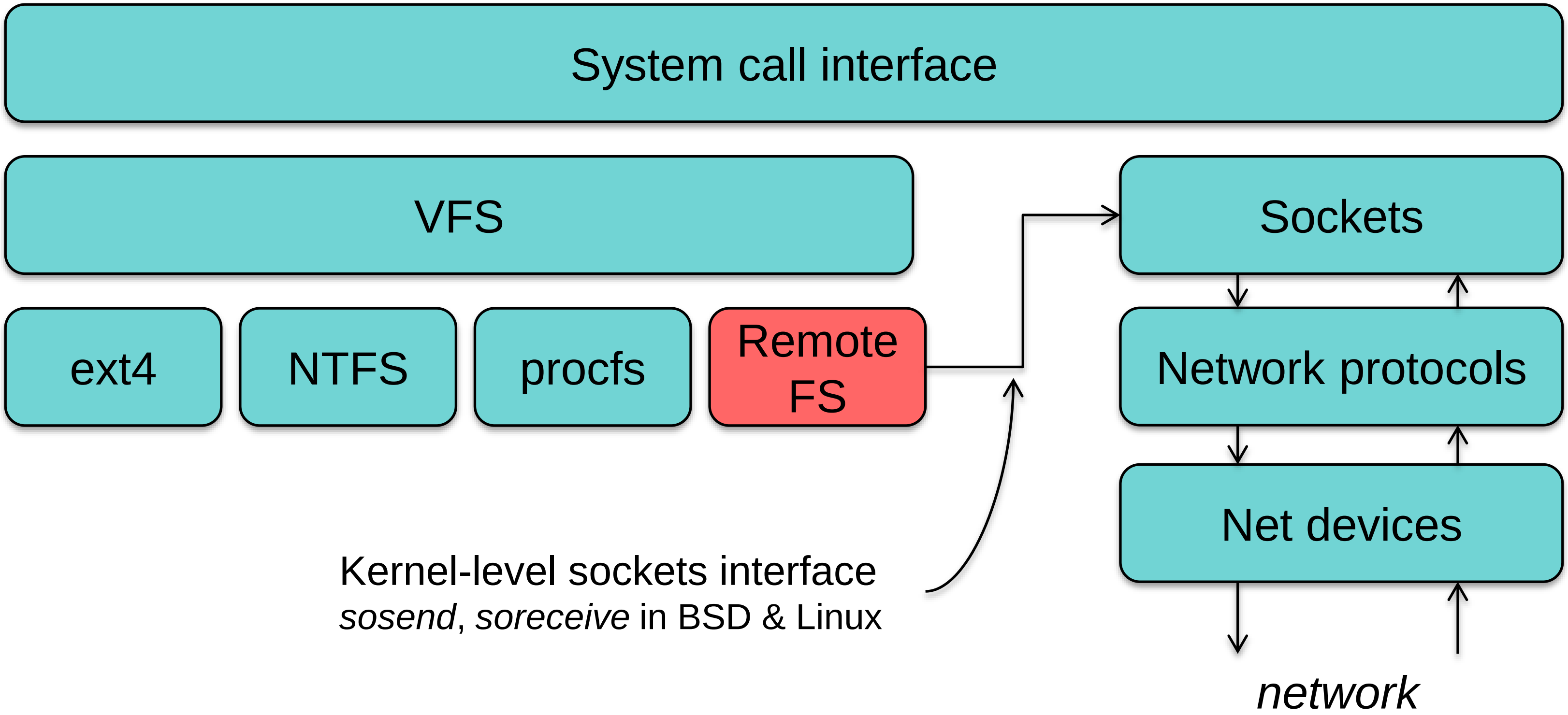
- Provides file access interface to clients

Remote file client (driver)

- Client side interface for file and directory service
- File system driver under VFS layer will provide access transparency
  - Remote files will be accessed in the same way as local files

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

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.

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.

Case Study: NFS

Network File System

Sun Microsystems

NFS Design Goals

- Any machine can be a client or server
- Must support diskless workstations
  - Remote device files refer back to local drivers so we can access our devices
- Heterogeneous systems
  - Not 100% for all UNIX system call options
- Access transparency: normal file system calls
- Recovery from failure:
  - Stateless, UDP, client responsible for retransmission
  - Stateless → no file locking possible!
- High Performance
  - use caching and read-ahead

NFS Design Goals

Transport Protocol

Initially NFS ran over UDP using Sun Remote Procedure Calls

Why was UDP chosen?

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

static mounting

- mount 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

- 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)

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

Improving NFS: version 2

- User-level lock manager
  - Monitored locks: introduces *state* at server
    (but runs as a separate user-level process)
    - 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

- Support 64-bit file sizes
- TCP support and large-block transfers
  - All traffic can be multiplexed on one connection
    - Minimizes connection setup
- 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
  - Reduces 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

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 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

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

The End