Distributed File Systems

CS 239, Spring 1996

May 18, 1996

Outline

- Basic concepts
- NFS
- Andrew File System
- Replicated file systems
  - Ficus
  - Coda
- Serverless file systems

Basic Distributed File System Concepts

You're here, the file's there, what do you do about it?

Important questions -

- What files can I access?
- How do I name them?
- How do I get the data?
- How do I synchronize with others?

What Files Can Be Accessed?

Several possible choices -

- Every file in the world
- Every file stored in this kind of system
- Every file in my local installation
- Selected volumes
- Selected individual files

What Dictates the Proper Choice?

Why not make every file available?

- Naming issues
- Scaling issues
- Local autonomy
- Security
- Network traffic

Naming Files in a Distributed System

- How much transparency?
  - Does every user/machine/sub-network need its own namespace?
- How do I find a site that stores the file that I name? Is it implicit in the name?
- Can my naming scheme scale?
- Must everyone agree on my scheme?

How Do I Get Data For Non-Local Files?

- Fetch it over the network?
- How much caching?
- Replication?
- What security is required for data transport?

Synchronization and Consistency

- Will there be trouble if multiple sites want to update a file?
- Can I get any guarantee that I always see consistent versions of data?
  - i.e., will I ever see old data after new?
  - How soon do I see new data?

NFS

Network File System

Provide distributed filing by remote access

With a high degree of transparency

Method of providing highly transparent access to remote files

Developed by Sun

NFS Characteristics

- Volume-level access
- RPC-based
- Stateless remote file access
- Uses XDR
- Location (not name) transparent
- Implementations for many systems
  - All interoperate, even non-Unix ones
- Currently based on VFS

VFS/Vnode Review

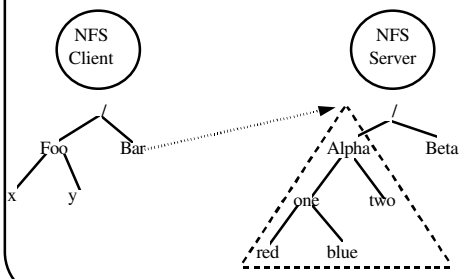
VFS - Virtual File System

Common interface allowing multiple file system implementations on one system

Plugged in below user level

Files represented by vnodes

NFS Diagram



File Handles

On the client site, files are represented by vnodes

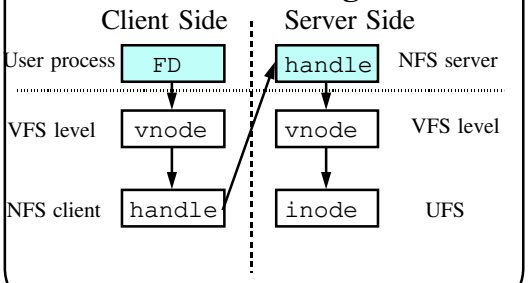
The client NFS implementation internally represents remote files as *handles*

Opaque to client

But meaningful to server

To name remote file, provide handle to server

NFS Handle Diagram



How To Make This Work?

- Could integrate it into the kernel
  - Non-portable, non-distributable
- Instead, use existing features to do the work
  - VFS for common interface
  - RPC for data transport

Using RPC For NFS

Must have some process at server that answers the RPC requests

Continuously running daemon process

Somehow, must perform mounts over machine boundaries

A second daemon process for this

NFS Processes

- **nfsd** daemons - server daemons that accept RPC calls for NFS
- **rpc.mountd** daemons- server daemons that handle mount requests
- **biod** daemons - optional client daemons that can improve performance

NFS From the Client's Side

User issues a normal file operation

Like **read()**

Passes through vnode interface to client-side NFS implementation

Client-side NFS implementation formats and sends an RPC packet to perform operation

Single client blocks until NFS RPC returns

NFS RPC Procedures

16 RPC procedures to implement NFS

Some for files, some for file systems

Including directory ops, link ops, read, write, etc.

lookup() is the key operation

Because it fetches handles

Other NFS file operations use the handle

Mount Operations

Must mount an NFS file system on the client before you can use it

Requires local and remote operations

Local operations indicate mount point has an NFS-type VFS at that point in hierarchy

Remote operations go to remote **rpc.mountd**

Mount provides "primal" file handle

NFS on the Server Side

The server side is represented by the local VFS actually storing the data

Plus **rpc.mountd** and **nfsd** daemons

NFS is stateless - servers do not keep track of clients

Each NFS operation must be self-contained

From server's point of view

Implications of Statelessness

- NFS RPC requests must completely describe operations
- NFS requests should be idempotent
- NFS should use a stateless transport protocol (e.g., UDP)
- Servers don't worry about client crashes
- Server crashes won't leave junk lying around

An Important Implication of Statelessness

Servers don't know what files clients think are open

Unlike in UFS, LFS, most local VFS file systems

Makes it much harder to provide certain semantics

Also scales nicely, though

Preserving Unix File Operation Semantics

NFS works hard to provide identical semantics to local UFS operations

Some of this is tricky

Especially given statelessness of server

E.g., how do you avoid discarding pages of unlinked file a client has open?

Sleazy NFS Tricks

Used to provide desired semantics despite statelessness of the server

E.g., if client unlinks open file, send rename to server rather than remove

- Perform actual remove when file is closed
- Won't work if file removed on server
- Won't work with cooperating clients

File Handles

Method clients use to identify files

Created by the server on file look up

Must be unique mappings of server file identifier to universal identifier

File handles become invalid when server frees or reuses inode

Inode generation number in handle shows when stale

NFS Daemon Processes

- **nfsd** daemon
- **biod** daemon
- **rpc.mount** daemon
- **rpc.lockd** daemon
- **rpc.statd** daemon

nfsd Daemon

Server daemon to handle incoming RPC requests

Often multiple **nfsd** daemons per site

Incoming NFS RPC requests go to one **nfsd** daemon

Which makes a kernel call to do the real work

Using daemons allows multiple threads

biod Daemon

Most client NFS operations go from VFS NFS implementation to the server

biod daemon does read-ahead for clients

- To make use of kernel file buffer cache

Only improves performance - NFS works correctly without **biod** daemons

Also flushes buffered writes for clients

rpc.mount Daemon

Runs on server to handle VFS-level operations for NFS

Particularly remote mount requests

Provides initial file handle for a remote volume

Also checks that incoming requests are from privileged ports (in UDP/IP packet source address)

rpc.lockd Daemon

NFS server is stateless, so it does not handle file locking

rpc.lockd provides locking

Runs on both client and server

Client side catches request, forwards it to server daemon

rpc.lockd handles lock recovery when server crashes

rpc.statd Daemon

Also runs on both client & server

Used to check status of a machine

Server's **rpc.lockd** asks **rpc.statd** to store permanent lock information (in file system)

And to monitor status of locking machine

If client crashes, clear its locks from server

Recovering Locks After a Crash

If server crashes and recovers, its **rpc.lockd** contacts clients to re-establish locks

If client crashes, **rpc.statd** contacts client when it becomes available again

Client has short grace period to revalidate locks

Then they're cleared

Caching in NFS

- What can you cache at NFS clients?
- How do you handle invalid client caches?

What Can You Cache?

- Data blocks read ahead by **biod** daemon
  - Cached in normal file system cache area

What Can You Cache, Con't?

- File attributes
  - Specially cached by NFS
  - Directory attributes handled a little differently than file attributes
  - Especially important because many programs get and set attributes frequently

Security in NFS

NFS inherits RPC mechanism security
- Some RPC mechanisms provide decent security
- Some don't

Mount security provided via knowing which ports are permitted to mount what

The Andrew File System

A different approach to remote file access

Meant to service a large organization
- Such as a university campus

Scaling is major goal

Basic Andrew Model

- Files are stored permanently at file server machines
- Users work from workstation machines
  - With their own private namespace
- Andrew provides mechanisms to cache user's files from shared namespace

User Model of AFS Use

Sit down at any AFS workstation anywhere

Log in and authenticate who I am

Access all files without regard to which workstation I'm using

The Local Namespace

Each workstation stores a few files

Mostly systems programs and configuration files

Workstations are treated as generic, interchangeable entities

Virtue and Vice

Vice is the system run by the file servers
- Distributed system

Virtue is the protocol client workstations use to communicate to Vice

Overall Architecture

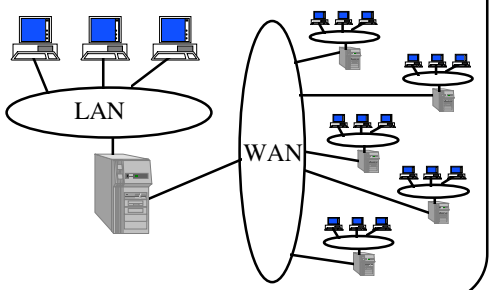
System is viewed as a WAN composed of LANs

Each LAN has a Vice cluster server

Which stores local files

But Vice makes all files available to all clients

Andrew Architecture Diagram



Caching the User Files

Goal is to offload work from servers to clients

When must servers do work?

To answer requests

To move data

Whole files cached at clients

Why Whole-File Caching?

- Minimizes communications with server
- Most files used in toto, anyway
- Easier cache management problem

Requires substantial free disk space on workstations

Doesn't address huge file problems

The Shared Namespace

An Andrew installation has global shared namespace

All clients name files in the namespace with the same names

High degree of name and location transparency

How Do Servers Provide the Namespace?

Files are organized into volumes

Volumes are grafted together into overall namespace

Each file has globally unique ID

Volumes are stored at individual servers

But a volume can be moved from server to server

Finding a File

At high level, files have names

Directory translates name to unique ID

If client knows where the volume is, it simply sends unique ID to appropriate server

Finding a Volume

What if you enter a new volume?

- How do you find which server stores the volume?

Volume-location database stored on each server

Once information on volume is known, client caches it

Moving a Volume

When a volume moves from server to server, update database

- Heavyweight distributed operation

What about clients with cached information?

Old server maintains forwarding info

- Also eases server update

Handling Cached Files

Client can cache all or part of a file

Files fetched transparently when needed

File system traps opens

- Sends them to local *Venus* process

The Venus Daemon

Responsible for handling single client cache

Caches files on open

Writes modified versions back on close

Cached files saved locally after close

Cache directory entry translations, too

Consistency For AFS

If my workstation has a locally cached copy of a file, what if someone else changes it?

Callbacks used to invalidate my copy

Requires servers to keep info on who caches files

Write Consistency in AFS

What if I write to my cached copy of a file?

Need to get write permission from server
- Which invalidates anyone else's callback

Permission obtained on open for write

Need not obtain new data at this point

Write Consistency in AFS, Con't

Initially, written only to local copy

On close, Venus sends update to server

Server will invalidate callbacks for other copies

Extra mechanism to handle failures

Storage of Andrew Files

Stored in Unix file systems

Client cache is a directory on local machine
- Low-level names do not match Andrew names

Venus Cache Management

Venus keeps two caches
- Status
- Data

Status cache kept in virtual memory
- For fast attribute lookup

Data cache kept on disk

Venus Process Architecture

Venus is single user process

But multithreaded

Uses RPC to talk to server
- RPC is built on low level datagram service

AFS Security

Only servers/Vice are trusted here
- Client machines might be corrupted

No client programs run on Vice machines

Clients must authenticate themselves to servers

Encryption used to protect transmissions

AFS File Protection

AFS supports *access control lists*

- Each file has list of users who can access it
- And permitted modes of access

Maintained by Vice

Used to mimic Unix access control

AFS Read-Only Replication

For volumes containing files that are used frequently, but not changed often

- E.g., executables

AFS allows multiple servers to store read-only copies