Distributed File Systems

CS 239, Spring 1995

May 23, 1995

Outline

- NFS
- Replicated file systems
  - Ficus
  - Coda

NFS

Network File System

Method of providing highly transparent access to remote files

In the Unix context

Basic NFS strategy -

Remote file service via RPC

Developed by Sun

NFS Characteristics

- Volume-level access
- Stateless remote file access
- Uses XDR
- Location transparent, not name transparent
- Implementations for many types of systems
  - Which all interoperate, even non-Unix ones
- Currently based on VFS

VFS/Vnode Review

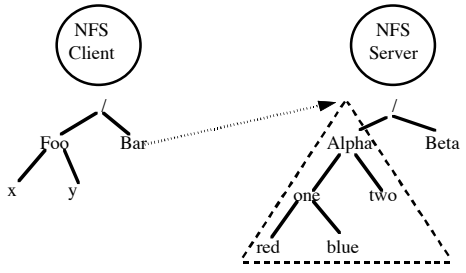
VFS - Virtual File System

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 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 **chmod()**

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

Mounting 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 need not 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

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 discarded 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

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
- 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 systems provide decent security

Other measures -

- Superuser mapping
- Unknown user mapping
- Restricted host access
- Port checking

Replicated File Systems

File replication can be very valuable

- For performance
- For reliability
- For availability
- For mobility

Optimistic replication increases these advantages

Ficus

Optimistic file replication based on *peer-to-peer* model

In peer-to-peer model, all replicas are equals

Updates can be made to any replica

Any replica can propagate updates to any other replica

Ficus is built using stackable layers

Basic Ficus Architecture

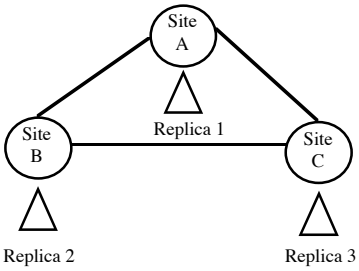
Ficus replicates at the volume level

File volumes can be replicated any number of times (but doesn't scale well beyond several replicas)

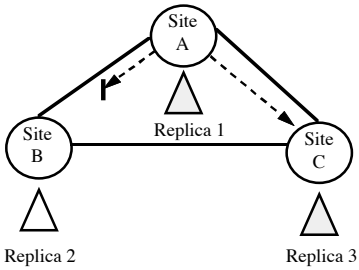
Updates propagated on a single best-effort basis

Consistency achieved by periodic *reconciliation* process

Ficus Diagram



Update Diagram



Reconciliation in Ficus

Run periodically on each Ficus site

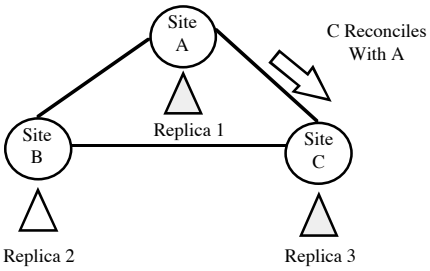
For each local volume replica

1). Get information from a remote replica

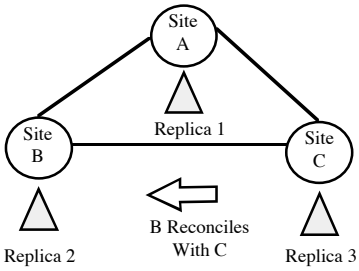
2). Compare it to the local replica

3). Change local replica to reflect remote changes

Reconciliation Diagram



Reconciliation Diagram, Con't



Reconciliation Topology

Reconciliation is a pairwise operation

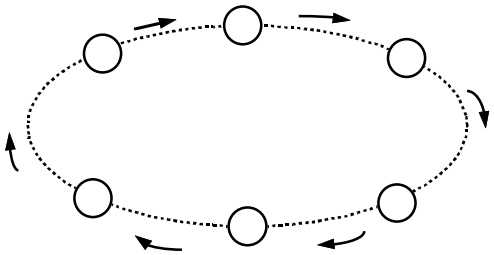
In the general case, how to choose which pairs to reconcile?

Reconciling all pairs unnecessary due to gossip

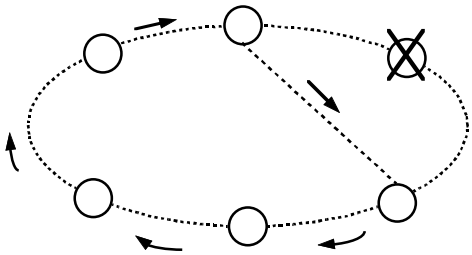
Want to minimize number of recons to pass data

But propagate data quickly

Ring Reconciliation Topology



Adaptive Ring Reconciliation Topology



Problems in File Reconciliation

- Recognizing updates
- Recognizing update conflicts
- Handling conflicts
- Recognizing name conflicts
- Update/remove conflicts
- Garbage collection

Ficus has solutions for all of these problems

Recognizing Updates

Ficus keeps per-file version vectors

Comparison of an updated replica's version vector to the version vector of an un-updated file detects the update

The data for updated files can then be propagated

Ficus propagates full files

Recognizing Update Conflicts

Concurrent update leads to update conflict

Same version vector mechanism will detect update conflicts

Will work equally well on an n-way conflict

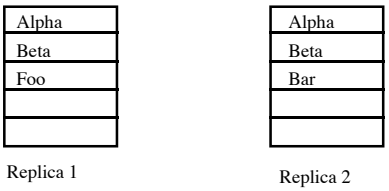
Handling Directory Conflicts

Ficus uses special mechanisms for conflicts in directories

Directories are essentially lists of file names

Unix permits only limited operations on file names

Directory Conflict Diagram



How Did This Directory Get Into This State?

Possibility 1:

i). Alpha and Beta exist

ii). Create Foo at replica 1

iii). Create Bar at replica 2

Possibility 2:

i). Alpha, Beta, and Foo exist

ii). Delete Foo at replica 2

iii). Create Bar at replica 2

The Create/Delete Ambiguity

This is an example of a general problem in replicated data

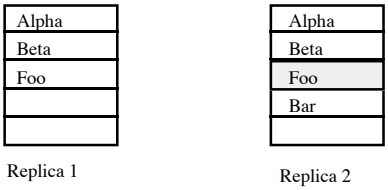
Cannot be solved with per-file version vectors

Requires per-entry information

Ficus keeps per-entry directory information for that purpose

And does not immediately destroy removed entries

Ficus "Internal" Representation of Problem Directory



Handling Update Conflicts

Ficus uses *resolver* programs to handle conflicts

Resolvers attempt to deduce the file type

Then apply semantic knowledge about type to resolve conflict

Ficus has a couple of dozen resolvers

If no resolvers work, mark file as "in conflict"

And make it unusable

Recognizing Name Conflicts

A *name conflict* occurs when two different files are concurrently given the same name

Ficus recognizes them with its per-entry directory info

Then what?

Handle similarly to update conflicts

Add disambiguating suffixes to each name

Update/Remove Conflicts

Say file “foo” has two replicas

1). Replica 1 receives an update

2). Replica 2 is removed

What should happen?

A matter of system semantics, basically

Ficus supports “no lost update” semantics

So don’t toss “foo” completely

Garbage Collection

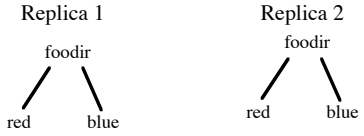
When can you discard the bits of a removed file?

Why not at once?

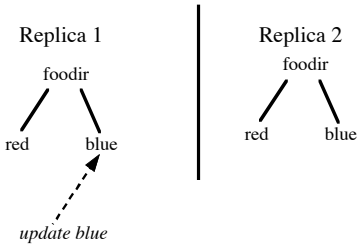
Ah, but what about hard links?

In Unix, files don’t disappear until all their hard links are removed

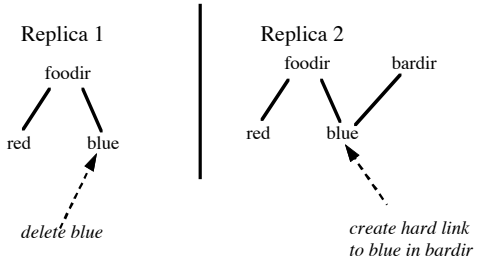
Link Example



Link Example, Con’t



Link Example, Part III



What Should Happen Here?

- Clearly, the link named “foodir/blue” should disappear
- And the link in “bardir” should remain
- But what version of the data does the bardir link point to?
- If you want true “no lost update” semantics, it must be the latest update at replica 1

So When Can I Throw Away My Data?

- So a replica cannot throw away data when all local links disappear
- Only when it knows that all links at all sites have disappeared
- And, moreover, when it knows that all replicas know
- Requires two trips around recon ring to find this out