Networked File System

CS 537 - Introduction to Operating Systems

Remote File Systems

- Often useful to allow many users access to the same file system
  - shared files everyone uses
  - can have a much larger storage space than a single computer
  - easier to protect against failure of the system

Remote File Systems

- Two major issues with remote file systems
  - performance can be awful
    - have to traverse a network to get data
  - recovery
    - what if the server crashes in the middle of a write
    - what if the client crashes
  - consistency
    - what if two people are simultaneously changing file

Networked File System (NFS)

- Major Goals
  - machine and OS independent
  - simple crash recovery
  - transparent access
    - don't need to re-write current programs to use NFS
  - support Unix semantics
    - this doesn't happen perfectly
  - reasonable performance
    - 80% of a local drive

Terminology

- Server
  - contains all of the files and directories
  - responsible for maintaining the file system
- Client
  - requester of directory and file information
  - does the actual reading and writing of files
- file handle
  - a way to access a file without giving the file name
  - similar to a file descriptor on a local file system

Remote Procedure Call (RPC)

- Method of getting one machine to run code on behalf of another machine
- Package up remote procedure name and parameters and send across the network
- Receiving machine runs procedure, packages up results, and sends them back
- Very similar to a function call in a high level programming language

RPC

- Initial implementations of RPC used the UDP communication protocol
  - if no response in a certain amount of time, just re-send the request
- Today both UDP and TCP are used
  - implemented on top of the IP protocol

NFS Protocol

- NFS is implemented using RPC
  - a client issues a request to the server by placing all necessary information to complete the request in the parameters
  - RPC calls are synchronous
    - client blocks until the server sends a response back
- This looks exactly like a procedure call on a local system
  - exactly what a user is used to seeing when they make a system call

NFS Protocol

- NFS protocol is stateless
  - each procedure call by a client contains all the necessary information to complete an operation
  - server doesn't need to maintain any information about what is at the clients site
  - server also doesn't keep track of any past requests
- This makes crash recovery very simple

Crash Recovery

- If a server crashes
  - just reboot the server
  - client just keeps sending its request until the server is brought back on-line
  - remember, RPC is synchronous
- If a client crashes
  - no recovery is necessary at all
  - when client comes back up it just starts running program again

Crash Recovery

- In a system that maintains state
  - both client and server must be able to detect a crash by the other
  - if client crashes
    - server discards all changes made by client
  - if server crashes
    - client must rebuild the servers state

NFS Protocol

- There are a set of standard NFS procedures
- Here are a few of the major ones
  - *lookup(dirfh, name)* returns (fh, attr)
  - *create(dirfh, name, attr)* returns (newfh, attr)
  - *remove(dirfh, name)* returns (status)
  - *read(fh, offset, count)* returns (attr, data)
  - *write(fh, offset, count, data)* returns (attr)
- Notice that *read* and *write* require the offset
  - this prevents server from maintaining a file ptr
    - a file ptr would be client state

File Handle

- Consists of the following
  - <inode #, inode generation #, file system id>
- NFS reuses inodes after a file has been deleted
- May be possible to hand out a file handle and then have the file deleted
- When original file handle comes back to server, it needs to know it is for an old, deleted file

Virtual File System

- Major goal of NFS is system independence
- Concept of the Virtual File System (VFS)
  - this is an interface that the client side must implement
  - if implemented properly, the client can then communicate with the NFS server regardless of what type of system each is
- Can allow different file systems to live on the same client

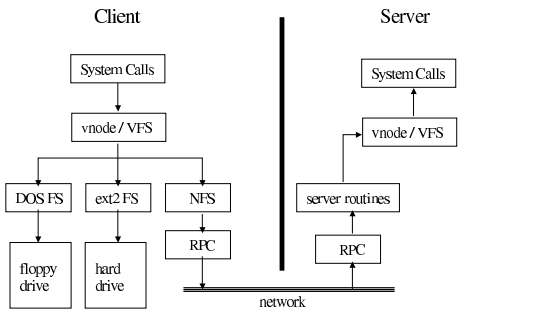
Virtual Node (vnode)

- An abstraction of a file or directory
  - a "virtual inode"
- Provides a common interface to a file
- This must also be implemented by the client
- Allows files on different types of file systems to accessed with the same system calls

vnode

- Here is a small sampling of the operations
  - *open(vnode, flags)*
  - *close(vnode, flags)*
  - *rdwr(vnode, uio, rwflag, flags)*
  - *create(dvnode, name, attr, excl, mode)*
  - *link(vnode, todvnode, toname)*
  - *symlink(dvnode, name, attr, to_name)*

VFS



Pathname Traversal

- Break name into components and call lookup for each component
- Requires multiple calls to lookup for a single pathname
  - don't pass entire path name into lookup because of mounting issues
  - mounting is independent protocol from NFS
    - can't be separated from the architecture
- Seems slow so…use cache of directory entries

Increasing Performance

- Client caches file and directory data in memory
- Use a larger packet size
  - less traffic for large reads or writes
- Fixed some routines to do less memory copying
- Cache client attributes
  - this prevents calls to server to get attributes
  - server notifies client if attributes change

Increasing Performance

- Cache directory entries
  - allows for fast pathname traversal
- For small executable files
  - send the entire file on execution
  - versus demand page-in of executable file
  - most small executable files touch all pages of the file
  - a form of read-ahead

Hard Issues

- Authentication
  - user passes uid and gid on each call
  - very large number of uid's and gid's on a distributed system
  - NFS uses a yellow pages
    - just a big database of users and their rights
- Concurrent Access
  - what if two users open a file at the same time?
  - could get interleaved writes
    - especially if they are large writes
    - this is different from Unix semantics

Hard Issues

- Open File Semantics
  - what if a user opens a file and then deletes it?
  - in Unix, just keep the file open and let the user read and write it
    - when the file is closed, the file is deleted
  - in NFS, rename the file
    - this sort of deletes the old version of it
  - when file is closed, client kernel is responsible for deleting it
  - if system crashes in between, end up with a garbage file in the file system

Major Problem with NFS

- Write performance is slow
- While clients may buffer writes, a write to the server is synchronous
  - no DMA to hide the latency of a write
- This is necessary to maintain statelessness of the server and client
- Could add non-volatile RAM to the server
  - expensive