



Distributed File Systems

Arvind Krishnamurthy
Spring 2004



Distributed File Systems

- A distributed file system provides transparent access to files stored on a remote disk
- Usage scenario:
 - You login into any zoo machine, access your home directory
 - Your project partner works on the workstation next to you, shares files with you
- How do you support it? Or rather, how do you implement:
 - Open(filename) → file-descriptor
 - Read(file-descriptor, position, size) → array of bytes
 - Write(file-descriptor, position, size, array of bytes) → status



Issues

1) System size: what is the target system size?

- NFS: dozens of workstations
- Sprite: 100's of workstations
- AFS: 1000's of workstations
- As the scale increases, what extra issues do you have to worry at larger scale?

2) Sharing/transparency:

- Name transparency: all three systems support
- AFS: local disks, shared component
- NFS: diskless, local/shared
- Sprite: single shared file system, supports remote devices



3) Locating File Servers

- NFS approach
 - Extend mount table: add host name
 - Has to be on a local disk, replicated
- AFS:
 - System-wide table, special protocol for obtaining current copy
 - Forwarding when files move
- Extreme approach: broadcast every file name
 - Servers know what files they serve, respond to broadcast
 - Can be cached to minimize broadcasts
 - On failure, broadcast request



4) Name lookup

- Translate /a/b/c to some kind of file identifier
 - One element at a time (NFS, AFS)
 - Option 1: Client makes a remote procedure call
 - Option 2: Read the directory and find the appropriate entry
 - Send the whole path name to the server (Sprite)
 - Server iterates through the directories and returns the final file identifier



5) Caching

A) Where?

- Disk (AFS)
- Memory (others)

B) What?

- Whole files (AFS)
- Blocks (others)

C) Writing policies?

- Write-through on close (AFS)
- NFS: starts pushing blocks (as soon as possible in background) and does write-through on close
- Sprite: delayed writes (after 30 secs push to next level)



Consistency

- Consistency is different from synchronization
 - Weaker than synchronization
 - Reads return the value of previous write
- AFS:
 - Freeze the file on open
 - If file changes, client gets notified. Next time client refetches the file
- NFS:
 - Version number, checks occasionally, flushes if different
- Sprite:
 - Version numbers, check on open, callbacks to disable caching when there is write-sharing



Failures

- What if server crashes?
- Can client wait until server comes back up, and continue as before?
- Issues:
 - Any data in server memory but not yet on disk can be lost
 - Shared state across RPCs.
 - Example: open, seek, read. What if server crashes after seek?
 - Message re-tries – suppose server crashes after it does “rm foo”, but before acknowledgement
- What if client crashes?
 - Might lose modified data in client cache



NFS Protocol: Stateless

- Write-through caching – when a file is closed, all modified blocks are sent immediately to the server disk
- To the client, “close” doesn’t return until all bytes are stored on disk
- Stateless protocol: server keeps no state about client, except as hints to help improve performance
 - Each request gives enough information to do entire operation
ReadAt(inumber, position) not Read(openfile)
 - When server crashes and restarts, can start processing requests immediately as if nothing happened



NFS Protocol (contd.)

- Most operations are “idempotent”
 - All requests are ok to repeat
 - If server crashes between disk I/O and message reply, client can resend message, server just does operation all over again
 - Read and write file block is easy (just re-read or re-write)
 - What about “remove”? NFS does the remove twice, and returns an error the second time
- Failures are transparent to client system
 - Is this a good idea? What should happen if server crashes?
Suppose you are in the middle of reading a file, and server crashes
 - Option 1: hang until server comes back up
 - Option 2: return an error
 - NFS does both – can select which one



AFS and Sprite Failures

- Client failure:
 - AFS might have to flush stuff on its local disk back to the server
 - Sprite loses data on client cache
- Server failure:
 - Need to rebuild callbacks
 - When a server comes back up, clients tell the server what files it has opened



Coda and disconnected operation

- AFS users often go a long time without any communication between their desktop client and any AFS server
- Coda says: “why can’t we use AFS-like implementation when disconnected from the network?”
 - On an airplane
 - At home
 - During network failure
- Issues
 - Which files to get before disconnection
 - Consistency



Hoarding

- AFS keeps recently used files on local disk
 - Most of what you need will be around
- Users can specify “hoard lists” to tell Coda to cache a bunch of other things even if not already stored locally
- System can also learn over time which files a user tends to use



Consistency

- What if two disconnected users write the same file at the same time?
 - No way to use callback promises since server and client cannot communicate
- Coda's solution: cross your fingers, hope it does not happen, and pick up pieces if it does
 - Log of changes kept while disconnected
 - Apply changes upon reconnect
 - If conflict detected, try to resolve automatically, else ask the user
- In practice, unfixable conflicts almost never happen