Caching: File Systems

Outline

- File Usage Patterns
- Concurrent Write Sharing
- Availability

Distributed File Systems

- Most Commonly used Distributed Program
- Issues
  - naming
  - caching
  - availability
  - security
- Examples
  - NSF: Sun's Network File System
  - AFS: Andrew File System
  - Sprite: Berkeley research project; stressed caching
  - Coda: CMU research project; stressed availability

Questions

- Semantics
  - How much does it behave like Unix on a single system?
- Performance
  - How close is its perform to Unix on a single system?
- Scale
  - How big can it grow?

Unix File Usage

- Most files are small (<10k)
- Reads outnumber writes 6:1
- Sequential access is common; random is rare
- Files open short periods (75% < .5s, 90% < 10s)
- Most files accessed by only one user
- Most shared files written by only one user
- Temporal locality: recently accessed files are most likely to be accessed again

NSF Structure

- Can mount an NFS file system into the name space of a Unix machine
  - hard mount: RPC failures block client
  - soft mount: RPC failures return error to client
- One NFS client module on each machine
  - in kernel for efficiency
- One NFS server module on each server
  - in kernel for efficiency

Stateless Server

- Idea: server doesn't store any state except
  - contents of files
  - hints to help performance (but not correctness)
- Consequence: server can crash and recover easily
- All client RPC ops must be idempotent
- Server doesn't maintain lists of who has files open
  - always name file and check permission

Stateless Server (cont)

- fhandle, attr = **lookup**(fhandle, name)
  - *uid, gid, size...* — *opaque* — *token from pathname*
- attr, data = **read**(fhandle, offset, count)
  - *current file pointer* — *client defines block size*
- attr = **write**(fhandle, offset, data)
  - *when returns, guaranteed to have written data to disk*

Server Caching

- Server caches recent reads, writes, directory-ops
  - server typically has lots of memory
- Server cache is write-through
  - all modifications immediately written to disk
- Pro:
  - stateless, no data lost on server crash
- Con:
  - slow, client must wait for disk write

Client Caching

- Clients cache results of reads, writes, and directory-ops in memory
- Raises cache consistency problem
- Half-Solution
  - server keeps last-modification time per file
  - client remembers time it got file data
  - on file open or new block access, client checks its timestamp against server's (checks every 3 seconds)
  - good enough?

Performance Problems

- Write-through in server
  - put crash-proof cache in front of disk (e.g., battery-backed RAM, flash RAM)
- Frequent timestamp checking
- Pathname lookup done one component at a time
  - required by Unix semantics

Andrew File System

- Originally a research project at CMU
- Now a commercial product from Transarc
- Goal: a single, world-wide file system

AFS Features

- Files identified by 96-bit Ids
  - 32-bit VolumeId, 32-bit VnodeId, 32-bit UserId
- Directory info stored in flat file system
  - software provides hierarchical view to users
- Files cached on client's disk
- Whole-file transfers/cached
- Designed so files will migrate to desktop of people that use them; assumes sharing is rare

Callback Promise

- Server commits to warn client about obsolete files
  - granted to client when client gets a copy of file
  - client accesses file only if it has a callback promise
  - once exercised by server, callback promise vanishes
- Fault-Tolerance
  - on client reboot, client discards all callback promises
  - server must remember promises made, even if it reboots

Cache Consistency

- Client checks for callback promise only when file is opened
- When writing, new version of file made visible to the world only on file close
- Result: concurrent read sharing semantics
  - opening file gets current snapshot
  - closing file creates new snapshot
  - concurrent write sharing leads to unexpected results

AFS Performance

- Only known file system that scales to thousands of machines
- Whole-file caching works well
- Callbacks more efficient than the repeated consistency checks of NFS

Sprite File System

- Research project at UC Berkeley
- Supports concurrent write sharing
- Block-based
  - fixed sized (4KB)
  - unique block number (not physical address)
  - client can create new blocks w/out contacting server
- Stateful servers
  - server notified when file opened/closed
  - all interactions are client-to-server; no client-to-client

Algorithm

- Write policy: write back all dirty blocks every 30s
  - client-to-server
  - server-to-disk
- When server detects concurrent write sharing
  - notifies clients to write back all dirty blocks
  - caching disabled
- Server serializes access to its cache

Algorithm (cont)

- When client opens, may have out-of-date cache
  - server keeps version number for all files
  - increment each time file opened for writing
  - client keeps version number of all cached files
  - compare version numbers on open; flush if old
- Server keeps track of last writer (at most one)
  - when someone opens, server asks last writer to flush any dirty blocks
  - block open until done

Coda File System

- Observation: AFS client often goes a long time without communicating with servers
- Coda use AFS-like implementation when disconnected from the network
  - on an airplane
  - at home
  - during network failure

Disconnected Operation

- Problem: how to get the right files onto the client before disconnecting
- Solution
  - AFS does a decent job already
  - let user make a list of files to keep around
  - have system learn which files user tends to use

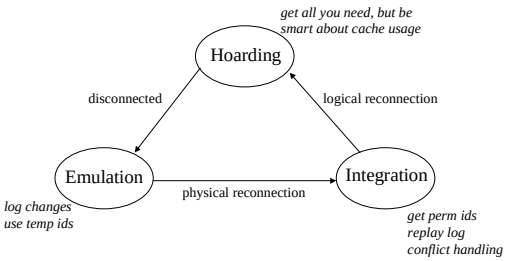
Consistency

- Problem: how to keep disconnected versions consistent
  - what if two disconnected users write the same file at the same time?
  - what if a writer makes a disconnected version obsolete?
- Can't use callback promise since communication is impossible

Consistency (cont)

- Strategy
  - hope it doesn't happen
  - if it happens, hope it doesn't matter
- If it does happen, try to patch things up automatically
  - example: creating two files in the same directory
- If all else fails, ask the user what to do

Implementation



Experience

- Unfixable conflicts almost never happen
  - typical user can go months without seeing one
  - but are workloads changing?
- Can Coda experience be applied to AFS?
  - exercise callback promises lazily
  - keep working despite network failures