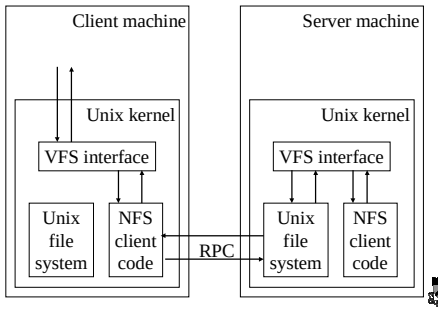
Distributed File Systems

- the most commonly used distributed programs
- illustrate many important issues
- today
  - NFS: (Sun) Network File System
  - AFS: Andrew File System
  - Coda: file system for disconnected operation

Features of Unix File Accesses

- most files small (<10k)
- reads outnumber writes by 6:1
- sequential access common; random rare
- 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 soon

NFS Structure



NFS Structure

- can mount an NFS filesystem into the namespace 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 Servers

- idea: server doesn't store any state, except
  - contents of files
  - hints to help performance (but not correctness)
- consequence: server can crash and recover without causing trouble
- all client RPC ops must be idempotent
- server doesn't maintain lists of who has which file open
  - always name file and check permission

Caching in NFS

- server caches contents of recent reads, writes, directory-ops in memory
  - server usually 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 in NFS

- clients cache results or reads, writes, 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
    - » check every three seconds at most
  - good enough?

NFS Performance

- generally pretty good
- problems
  - write-through in server
    - » solution: put crash-proof cache in front of disk
      - battery-backed RAM
      - flash-RAM
  - frequent timestamp checking
  - pathname lookup done one component at a time
    - » required by Unix semantics

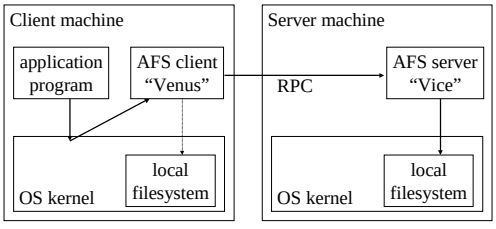
Andrew File System

- originally a research project at CMU
- now a commercial product from Transarc
- goal: a single, world-wide filesystem

Unusual Features of AFS

- whole-file transfers
  - don't divide into blocks or chunks
  - exception: divide huge files into huge chunks
- files cached on client's disk
  - client cache large, perhaps 1 Gig today
- whole files are cached

Structure of AFS



AFS

- internally, files identified by 96-bit Ids
  - directory info stored in flat filesystem
  - software provides hierarchical view to users
- design based on the hope that files will migrate to the desktops of people who use them
  - if sharing is rare, almost all accesses will be local

Cache Consistency in AFS

- key mechanism: callback promise
  - represents server's commitment to tell client when client's copy becomes obsolete
    - » granted to client when client gets 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 callback promises
    - » server must remember promises it made, even if server crashes

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: "funny" sharing semantics
  - opening file gets current snapshot of file
  - closing file creates a new version
  - concurrent write-sharing leads to unexpected results

AFS Performance

- works very well in practice
- the only known filesystem that scales to thousands of machine
- whole-file caching works well
- callbacks more efficient than the repeated consistency checking of NFS

Coda: Disconnected Operation

- observation: AFS client often goes a long time without communicating with servers
- Why not use an AFS-like implementation when disconnected from the network?
  - on an airplane
  - at home
  - during network failure
- Coda tries to do this

Disconnected Operation

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

Disconnected Operation

- problem: how to keep disconnected versions consistent
  - what if two disconnected users write the same file at the same time
    - » can't use callback promises, since communication is impossible
  - or, what if a write makes a disconnected version obsolete
    - » can't prevent this either

Consistency

- 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 same directory
- if all else fails, ask user what to do

Consistency

- amazing fact: unfixable conflicts almost never happen
  - typical user can go months without seeing an unfixable conflict
- so Coda works wonderfully
  - but: are workloads changing?
- can use these observations to improve AFS
  - exercise callback promises lazily
  - keep working despite network failures