CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2003

Lecture 23: Distributed File Systems

23.0 Main Points

- Examples of distributed file systems
- Cache coherence protocols

23.1 Concepts

A **distributed file system** provides transparent access to files stored on a remote disk

Issues:

- Naming: (always an issue). Some choices (no clear winners) –
  - *Hostname:*localname
    - Simple, but no location or migration transparency.
  - *Mounting* of remote file systems (a la NFS)
    - Transparency (to user, at least) but strange failure behavior, and manageability problems.
  - *A single, global name space* – hard to implement in practice.
- Failures: what happens when server crashes, but client doesn't? Or vice versa?
- Performance => caching: use caching at both the clients and the server to improve performance.

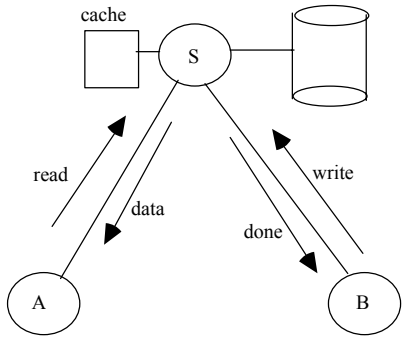
Cache coherence: how do we make sure each client sees most up to date copy?

23.2 No caching

Simple approach: use RPC to forward every file system request to remote server (Novell Netware, Mosaic Web browser).

Example operations: open, seek, read, write, and close

Server implements each operation as it would for a local request and sends back result to client



Advantage: server provides consistent view of file system to both A and B.

Problems? Performance can be lousy:

- Going over network is slower than going to local memory!
- Lots of network traffic
- Server can be a bottleneck – what if lots of clients?

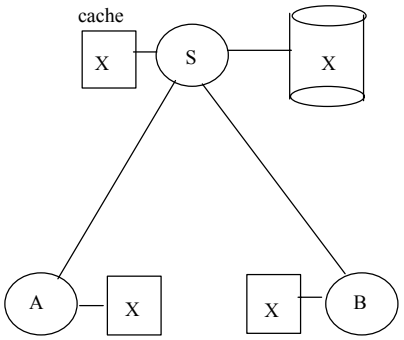
23.3 NFS (Sun Network File System)

Idea: Use caching to reduce network load

Cache file blocks, file headers, etc. at both clients and servers:

Client memory

Server memory



Advantage: if open/read/write/close can be done locally, no network traffic.

Issues: failures and cache consistency.

23.3.1 Motivation, part 1: Failures

What if server crashes? Can client wait until server comes back up, and continue as before?

1. Any data in server memory but not yet on disk can be lost.
2. Shared state across RPCs. Ex: open, seek, read. What if server crashes after seek? Then when client does "read", it will fail.
3. Message retries – suppose server crashes after it does UNIX "`rm foo`", but before acknowledgment? Message system will retry – send it again. How does it know not to delete it again? (Could solve this with two-phase commit protocol, but NFS takes a more ad hoc approach – sound familiar?)

What if client crashes?

Might lose modified data in client cache

23.3.2 NFS Protocol (part 1): stateless

1. 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.
2. Stateless protocol – server keeps no state about client, except as hints to help improve performance (Ex: a cache)

   Each read request gives enough information to do entire operation - ReadAt(inumber, position), not Read(openfile).

   When server crashes and restarts, can start again processing requests immediately, as if nothing happened.
3. Operations must be made "idempotent": all requests are ok to repeat (i.e., performing the operation multiple times has the same effect as performing it exactly once). So if server crashes between disk I/O and message send, client can resend message, server just does operation all over again.

   Read and write file block are easy – just re-read or re-write file block – no side effects.

What about "remove"? NFS just ignores this – does the remove twice, second time returns an error if file not found.

4. Failures are transparent to client system
   Is this a good idea? What should happen if server crashes? Suppose you are an application, in the middle of reading a file, and server crashes?

   Options:
   a. Hang until server comes back up (next week)?
   b. Return an error? Problem is: most applications don't know they are talking over the network – we're transparent, right?
      Many UNIX applications simply ignore errors! Crash if there's a problem.

   NFS does both options – can select which one. Usually, hang and only return error if really must – if you see "NFS stale file handle", that's why.

23.3.3 Motivation, part 2: cache consistency

What if multiple clients are sharing the same files? Easy if they are both reading – each gets a copy of the file.

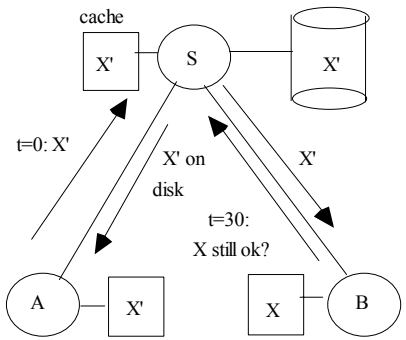
What if one is writing? How do updates happen?

Remember: NFS has write-through cache policy. If one client modifies file, writes through to server.

How does other client find out about the change?

23.3.4 NFS protocol, part 2: weak consistency

In NFS, client polls server periodically, to check if file has changed. Polls server if data hasn't been checked in last 3-30 seconds (exact timeout is tunable parameter).



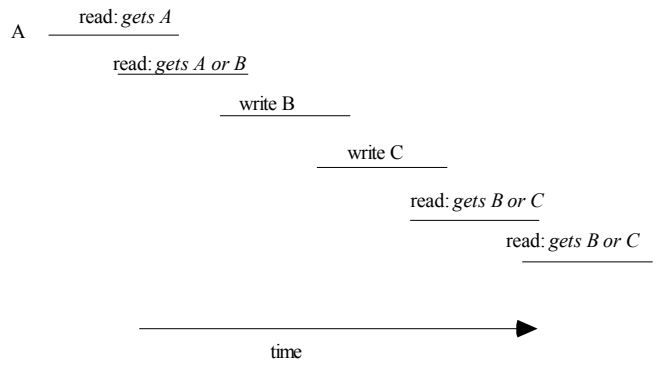
Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout. They then check server, and get new version.

What if multiple clients write to same file? In NFS, can get either version (or parts of both). Completely arbitrary!

23.3.5 Sequential Ordering Constraints

Cache coherence: What should happen? What if one CPU changes file, and before it's done, another CPU reads file?

Note that every operation takes time: actual read could occur anytime between when system call is started, and when system call returns.



Assume what we want is distributed system to behave exactly the same as if all processes are running on a single UNIX system.

If read finishes before write starts, then get old copy

If read starts after write finishes, then get new copy

Otherwise: get either new or old copy.

Similarly, if write starts before another write finishes, may get either old or new version. (Hence in above diagram, non-deterministic as to which value you end up with!)

In NFS, if read starts more than 30 seconds after write finishes, get new copy. Otherwise, who knows? Could get partial update

23.3.6 NFS Summary

NFS pros & cons:

+ Simple
+ Highly portable
– Sometimes inconsistent
– Doesn't scale to large # of clients

Might think NFS is really stupid, but Netscape does something similar: caches recently seen pages, and re-fetches them if they are too old. Cache coherence wasn't supported in early versions of HTTP.

23.4 Andrew File System

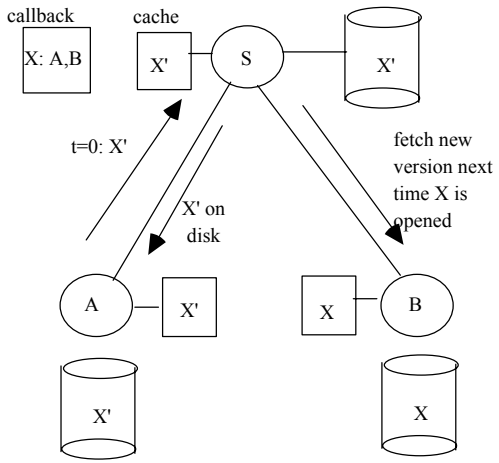
AFS (CMU, late 80's) -> DCE DFS (commercial product)

1. Callbacks: Server records who has copy of file
2. Write through on close
   If file changes, server is updated (on close)
   Server then immediately tells all those with the old copy.
3. Session semantics – updates visible only on close.
   In UNIX (single machine), updates visible immediately to other programs who have the file open.

   In AFS, everyone who has file open sees old version; anyone who opens file again will see new version. (this is a type of "multi-version" scheme).

   In AFS:
   a. On open and cache miss: get file from server, set up callback
   b. On write close: send copy to server; tells all clients with copies, to fetch new version from server on next open

4. Files cached on local disk; NFS caches only in memory



What if server crashes? Lose all your callback state!

Reconstruct callback information from client – go ask everyone "who has which files cached"

AFS pros & cons:

Relative to NFS, less server load:

+ Disk as cache -> more files can be cached locally
+ Callbacks -> server not involved if file is read-only
– On fast LANs, local disk is much slower than remote memory

In both AFS and NFS:

Central server is a bottleneck

Performance bottleneck:

– All data is written through to server
– All cache misses go to server

Availability bottleneck:

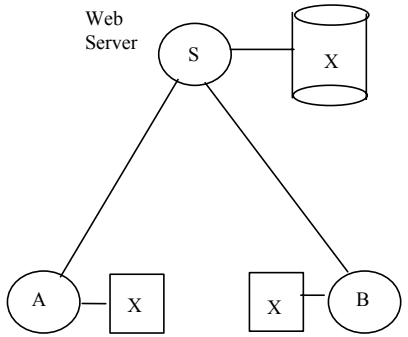
– Server is single point of failure

Cost bottleneck:

– Server machine's high cost relative to workstation

23.5 World Wide Web

Key idea: graphical front-end to RPC protocol.



23.5.1 No caching

Initial version had no caching. Didn't scale well – easy to overload servers.

23.5.2 Web server failures

What happens when the server fails?

- System breaks!
- Transport-layer redirection:
  - Invisible to applications
  - Can help with scalability
  - Must handle "sessions"

23.5.3 Cache consistency for the Web

Reduce number of interactions between clients and servers and/or reduce the size of the interactions:

- Time-to-Live (TTL) fields – HTTP "Expires" header from server
- Client polling – HTTP "If-Modified-Since" request headers from clients
- Server refresh – HTML "META Refresh tag" causes periodic client poll

What is the polling frequency for clients and servers?

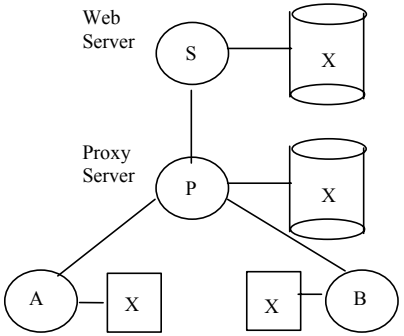
Could be adaptive based upon a page's age and its rate of change

Server load is still significant!

23.5.4 Proxy caches

Place caches *in* the network.

- Reduces server load
- But, increases latency in lightly loaded case



Can cache static traffic easily (but only gets around 40% of the traffic)

Dynamic and multimedia is harder (but multimedia is a big win: Megabytes vs Kilobytes).

Same cache consistency problems as before.

Caches are also placed near servers (called "reverse proxy caches") in order to offload busy server machines, and at the "edges" of the network (called "content distribution networks")

These caches are effectively changing the architecture of the Internet, by placing functionality at higher levels of the communications protocols.