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

Themes:

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.

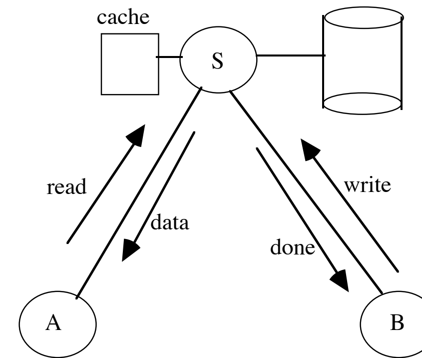
Cache coherence: how do we make sure each client sees most up to date copy?

23.1 No caching

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

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

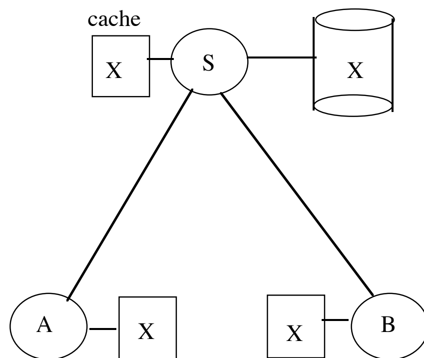
- going over network is slower than going to local memory!
- lots of network traffic
- server can be a bottleneck -- what if lots of clients?

23.2 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.

23.2.1 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 re-tries -- 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 aproach -- sound familiar?)

What if client crashes?

1. Might lose modified data in client cache

23.2.2 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 are "idempotent": all requests are ok to repeat (all requests done at least 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 app's 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 see "NFS stale file handle", that's why.

23.2.3 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.

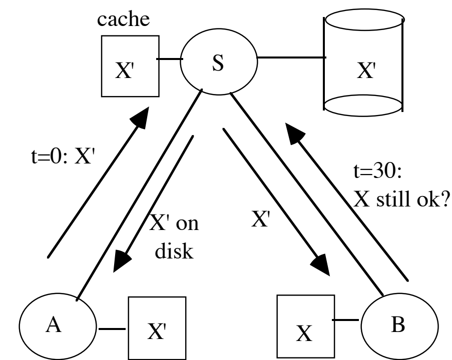
What if one is writing? How do updates happen?

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?

23.2.4 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).



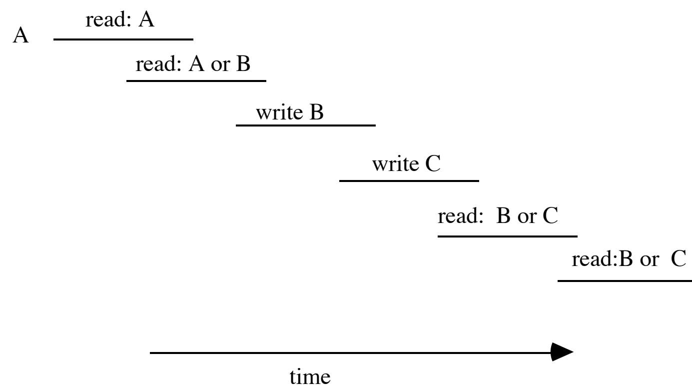
Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout. They then check server, and get new version.

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

22.2.5 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

22.2.6 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 refetches them if they are too old. Nothing in the WWW to help with cache coherence.

22.3 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.

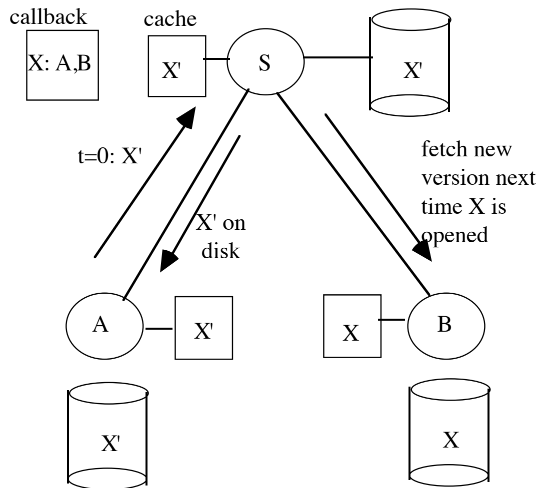
3. Session semantics -- updates visible only on close.

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.

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 machines high cost relative to workstation

22.4 xFS: Serverless Network File Service

Key idea: file system as a parallel program; exploit opportunity provided by fast LANs.

Four key ideas:
- Cooperative caching
- Write ownership cache coherence
- Software RAID
- Distributed control

22.4.1 Cooperative caching

Use remote memory to avoid going to disk (manage client memory as a global shared resource)

a. on cache miss, get file from someone else's cache instead of from disk
b. on replacement, if last copy of file, send to idle client, instead of discarding

+ better hit rate for read-shared data
+ active clients get to use memory on idle clients

22.4.2 Write ownership cache coherence

Does server have to get all updates, to keep everything consistent? Answer is no.

Almost all machines today have disks. Why write data back to the server? Why not just put them on your local disk?

Key idea: a machine can "own" a file. Owner has the most up to date copy; no one else has copy. Server keeps track of who

"owns" file; any request for the file goes to the server, who then forwards to the owner.

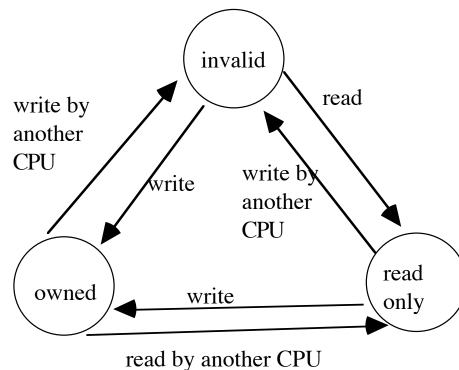
Either:

One "owned" copy of file (ex: file you are editing)
Multiple "read-only" copies (ex: emacs executable)

To modify file, must make sure you have only copy. Once you have only copy, ok to modify multiple times. If someone else reads file, forward the up to date version and mark read-only.

Each file block in the cache has three states:

invalid
read-only
owned (read-write)




```
Read file block:
    if (invalid)
      ask server who has up to date version
      fetch up to date version
      if any remote machine has it owned:
         on remote machine: read-only
      on local machine: read-only
```

```
Write file block
    if (invalid)
      ask server who has up to date version,
         and who has copies
      fetch up to date version
    else if (read-only)
      ask server who has copies

    if any remote machine has copy:
         on remote machine: invalid
    on local machine: owned
```

Example: three CPU's, one server.

What happens?
P1 read A
P2 write B
P3 read A
P1 read A
P2 write B
P3 write A

Perfectly ok for that machine to keep modifying the file, without telling the server. For instance, your machine is likely to be the only machine with your sub-directory. You'd like it to work if you move to another spot, but why tell the server every time you modify a file, just because there is the possibility someone might need it elsewhere.

22.4.3 Software RAID

We've made the availability story a whole lot worse. Now pieces of the file system are spread all over everywhere. If any machine in the system goes down, part of the file system unavailable.

xFS solution: stripe data redundantly over multiple disks, using software RAID. Each client writes its modifications to a log stored on redundant stripe of disks.

On failure, others can reconstruct data from the other disks in order to figure out missing data. Logging makes reconstruction easy.

A detail: need to be able to find things on disk. Done as in LFS via an inode/file header map, containing the location of every inode on disk. (This map is spread over all machines, kept by the last writer.)

Inode map is checkpointed to disk periodically. On failure, read checkpoint from disk, then update from logs written after checkpoint.

22.4.4 Distributed control

We've decentralized the cache, the disk, writes and reads. But there's still a central server to record who has which copies of data.

xFS solution: spread manager over all machines. If anyone fails, poll clients to know who has what, and then shift its responsibilities to a new client.

22.4.5 Summary

xFS: build large system out of large numbers of small, unreliable components.

Key: everything dynamic -- data, metadata, control can live anywhere, on any machine, in any memory, on any location on disk. Also, this means easy to migrate to tape: anything can be located anywhere.

Started with promise vs. reality of distributed systems: reality is lower performance,

xFS is an example of how distributed systems will look in the future: higher performance, higher availability than any centralized system. Improves performance as you add more machines: more CPUs, more DRAM, more disks, ought to mean better performance, better availability, not worse!

Also: automatic reconfiguration. Machine goes down, everything continues to work. Machine gets added, start using its disk and CPU. (In hardware, called "hot swap" -- key to high availability.)

Still some challenges: how do you upgrade software to new OS, new version of xFS, new version of disk, CPU, etc., while system continues to operate? Can we build systems that operate continuously for a decade?