Distributed Systems

Lec 9: Distributed File Systems – NFS, AFS

Slide acks: Dave Andersen

(http://www.cs.cmu.edu/~dga/15-440/F10/lectures/08-distfs1.pdf)

Homework 3 Update

- Some folks have argued that:
  1) HW 3 is too heavy for 2 weeks → Disagree!
  2) HW 3 has too much local-FS boilerplate to distill the distributed systems aspects → Agree!

- For reason 2), we are changing HW3:
  - HW3 (due 10/9): implement the basic extent server, basic file-oriented FS operations (create, lookup, readdir, setattr, open, read, write)
  - HW3.5 (due 10/16): implement directory operations (mkdir, remove) and distributed locking
  - This way, you experience DS aspects (esp. locking) with more focus

- If you've already done HW3 + HW3.5, or you wish to do them together, you're welcome to submit them as one before HW3 deadline (10/9)

VFS and FUSE Primer

- Some have asked for some background on Linux FS structure and FUSE in support of homework 3
- We'll talk about them now on whiteboard

Today

- Finish up distributed mutual exclusion from last lecture

- Distributed file systems (start)
  – Sun's Network File System (NFS)
  – CMU's Andrew File System (AFS)

Distributed Mutual Exclusion (Reminder)

- Ensure that only one thread can interact with shared resource (shared memory, file) at the same time

- Algorithms:
  - Centralized algorithm (A1)
  - Distributed algorithms
    - A2: Token ring
    - A3: Lamport’s priority queues
    - A4: Ricart and Agrawala (today)
    - A5: Voting (today)

- Quiz: Explain algorithms A1-A3

Lamport's Algorithm (Reminder)

- Each process keeps a priority queue Q_i, to which it adds any outstanding lock request it knows of, in order of logical timestamp
- To enter critical section at time T, process i sends REQUEST to everyone and waits for REPLYs from all processes and for all earlier requests in Q_i to be RELEASEd
- To exit critical section, sends RELEASE to everyone
- Process i delays its REPLY to process j's REQUEST until j has answered any earlier REQUESTs that i has outstanding to j

Solution 4: Ricart and Agrawala

- An improved version of Lamport's shared priority queue
  - Combines function of REPLY and RELEASE messages

- Delay REPLY to any requests later than your own
  - Send all delayed replies after you exit your critical section

Solution 4: Ricart and Agrawala

- To enter critical section at process *i* :
  - Stamp your request with the current time *T*
  - Broadcast REQUEST(*T*) to all processes
  - Wait for all replies

- To exit the critical section:
  - Broadcast REPLY to all processes in Q_i
  - Empty *Qi*

- On receipt of REQUEST(*T'*):
  - If waiting for (or in) critical section for an earlier request *T*, add *T'* to *Qi*
  - Otherwise REPLY immediately

Ricart and Agrawala Safety

- Safety and fairness claim: If $T1<T2$, then process P2 requesting a lock at *T2* will enter its critical section after process P1, who requested lock at *T1*, exits

- Proof sketch:
  - Consider how P*2* collects its reply from P1:
  - *T1* must have already been time-stamped when request *T2* was received by P1, otherwise the Lamport clock would have been advanced past time *T2*
  - But then P1 must have delayed reply to *T2* until after request *T1* exited the critical section
  - Therefore *T2* will not conflict with *T1*.

Solution 4: Ricart and Agrawala

- Advantages:
  - Fair
  - Short synchronization delay
  - Simpler (therefore better) than Lamport’s algorithm

- Disadvantages
  - Still very unreliable
  - *2(N-1)* messages for each entry/exit

Solution 5: Majority Rules

- Instead of collecting REPLYs, collect VOTEs
  - Each process VOTEs for which process can hold the mutex
  - Each process can only VOTE once at any given time

- You hold the mutex if you have a majority of the VOTEs
  - Only possible for one process to have a majority at any given time!

Solution 5: Majority Rules

- To enter critical section at process *i* :
  - Broadcast REQUEST(T), collect VOTEs
  - Can enter crit. sec. if collect a majority of VOTEs (N/2+1)
- To leave:
  - Broadcast RELEASE to all processes who VOTEd for you
- On receipt of REQUEST(*T'*) from process *j*:
  - If you have not VOTEd, VOTE for *T'*
  - Otherwise, add *T'* to *Qi*
- On receipt of RELEASE*:*
  - If Q_i not empty, VOTE for pop(Q_i)

Solution 5: Majority Rules

- Advantages:
  - Can progress with as many as *N/2 – 1* failed processes
- Disadvantages:
  - Not fair
  - Deadlock!
    - No guarantee that anyone receives a majority of votes

Solution 5’: Dealing with Deadlock

- Allow processes to ask for their vote back
  - If already VOTEd for *T’* and get a request for an earlier request *T*, RESCIND-VOTE for *T’*
- If receive RESCIND-VOTE request and not in critical section, RELEASE-VOTE and re-REQUEST
- Guarantees that some process will eventually get a majority of VOTEs → liveness
  - Assuming network messages eventually get to destination
- But still not fair…

Algorithm Comparison

Algorithm	Messages per entry/exit	Synchronization delay (in RTTs)	Liveness
Central server	3	1 RTT	Bad: coordinator crash prevents progress
Token ring	N	<= sum(RTTs)/2	Horrible: any process' failure prevents progress
Lamport	3*(N-1)	max(RTT) across processes	Horrible: any process' failure prevents progress
Ricart & Agrawal	2*(N-1)	max(RTT) across processes	Horrible: any process' failure prevents progress
Voting	>= 2*(N-1) (might have vote recalls, too)	max(RTT) between the fastest N/2+1 processes	Great: can tolerate up to N/2-1 failures

(sync delay: you request the lock; no one else has it; how long till you get it?)

So, Who Wins?

- Well, none of the algorithms we’ve looked at thus far
- But the closest one to industrial standards is…

So, Who Wins?

- Well, none of the algorithms we've looked at thus far
- But the closest one to industrial standards is…
  - The centralized model (e.g., Google's Chubby, Yahoo's ZooKeeper)

So, Who Wins?

- The closest to the industrial standards is…
  - The centralized model (e.g., Google’s Chubby, Yahoo’s ZooKeeper)
  - But replicate it for fault-tolerance across a few machines
  - Replicas coordinate closely via mechanisms similar to the ones we’ve shown for the distributed algorithms (e.g., voting) – we’ll talk later about generalized voting alg.
  - For manageable load, app writers must avoid using the centralized lock service as much as possible!

Take-Aways

- Lamport and Ricart & Agrawala's algorithms demonstrate utility of logical clocks
- Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue)!
  - We'll talk about replica consistency in the future
- If you build your distributed system wrong, then you get worse properties from distribution than if you didn't distribute at all

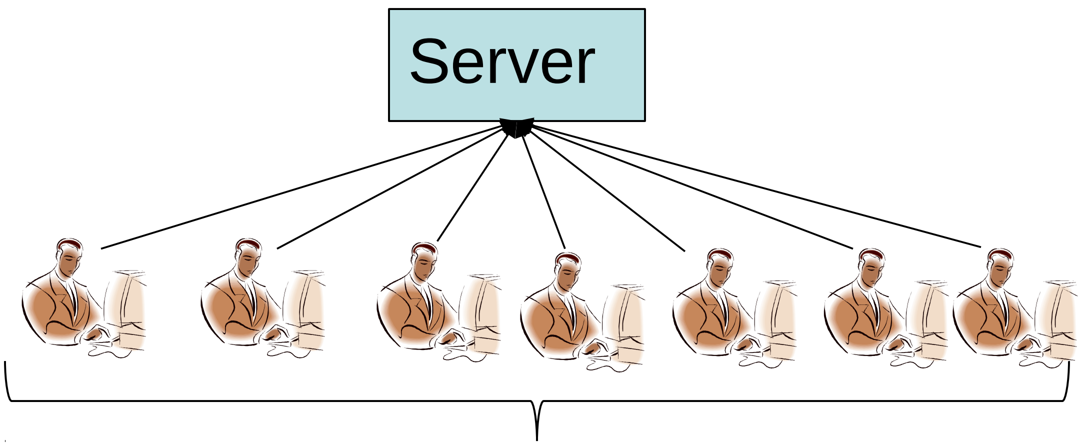
Today

- Finish up distributed mutual exclusion from last lecture

- Distributed file systems (start)
  - Sun's Network File System (NFS)
  - CMU's Andrew File System (AFS)

NFS and AFS Overview

- Networked file systems
- Their goals:
  - Have a consistent namespace for files across computers
  - Let authorized users access their files from any computer
- These FSes are different in properties and mechanisms, and that's what we'll discuss



10,000s of users from 10,000 of machines

Distributed-FS Challenges

- Remember our initial list of distributed-systems challenges from the first lecture?
  - Interfaces
  - Scalability
  - Fault tolerance
  - Concurrency
  - Security

- Oh no... we've got 'em all…
  - Can you give examples?

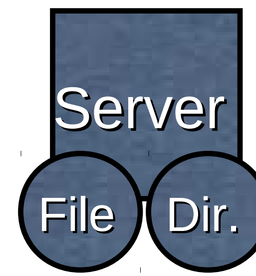
- How can we even start building such a system???

How to Start?

- Often very useful to have a prioritized list of goals
  - Performance, scale, consistency – what's most important?
- Workload-oriented design
  - Measure characteristics of target workloads to inform the design
- E.g., AFS and NFS are user-oriented, hence they optimize to how users use files (vs. big programs)
  - Most files are privately owned
  - Not too much concurrent access
  - Sequential is common; reads more common than writes
- Other distributed FSes (e.g., Google FS) are geared towards big-program/big-data workloads (next time)

The FS Interface

File Ops	Directory Ops
Open	Create file
Read	Mkdir
Write	Rename file
Read	Rename directory
Write	Delete file
Close	Delete directory



Naïve DFS Design

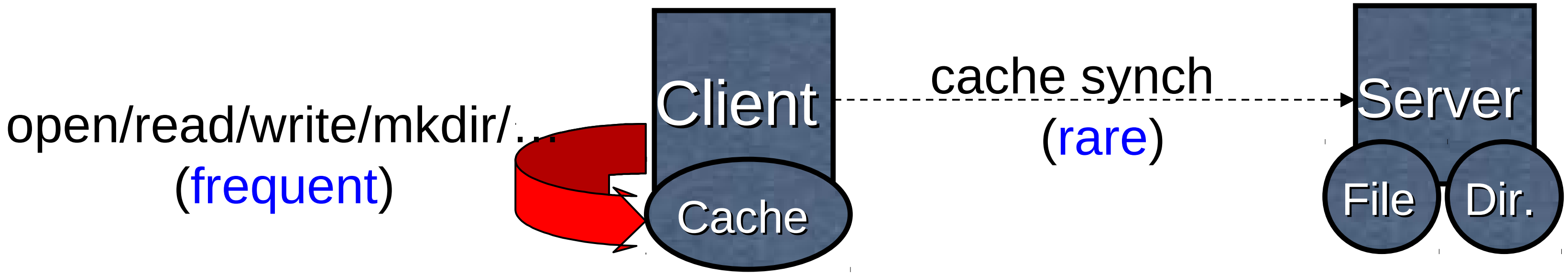
- Use RPC to forward *every* FS operation to the server
  - Server orders all accesses, performs them, and sends back result
- Good: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance will stink. Latency of access to remote server often much higher than to local memory.
- Really bad: Server would get hammered!

Lesson 1: Needing to hit the server for every detail impairs performance and scalability.

Question 1: How can we avoid going to the server for everything? *What* can we avoid this for? What do we lose in the process?

Solution: Caching

- Lots of systems problems are solved in 1 of 2 ways:
  1) Adding a level of indirection
     - "All problems in computer science can be solved by adding a level of indirection; but this will usually cause other problems" -- David Wheeler
  2) Caching data



- Questions:
  – What do we cache??
  – If we cache, don't we risk making things inconsistent?

Sun NFS

- Cache file blocks, directory metadata in RAM at both clients and servers.

- Advantage: No network traffic if open/read/write/close can be done locally.

- But: failures and cache consistency are big concerns with this approach
  - NFS trades some consistency for increased performance...

Caching Problem 1: Failures

- Server crashes
  - Any data that's in memory but not on disk is lost
  - What if client does seek(); /* SERVER CRASH */; read()
    - If server maintains file position in RAM, the read will return bogus data
- Lost messages
  - What if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo anew?
  - The first client might retry the delete and delete new file
- Client crashes
  - Might lose data updates in client cache

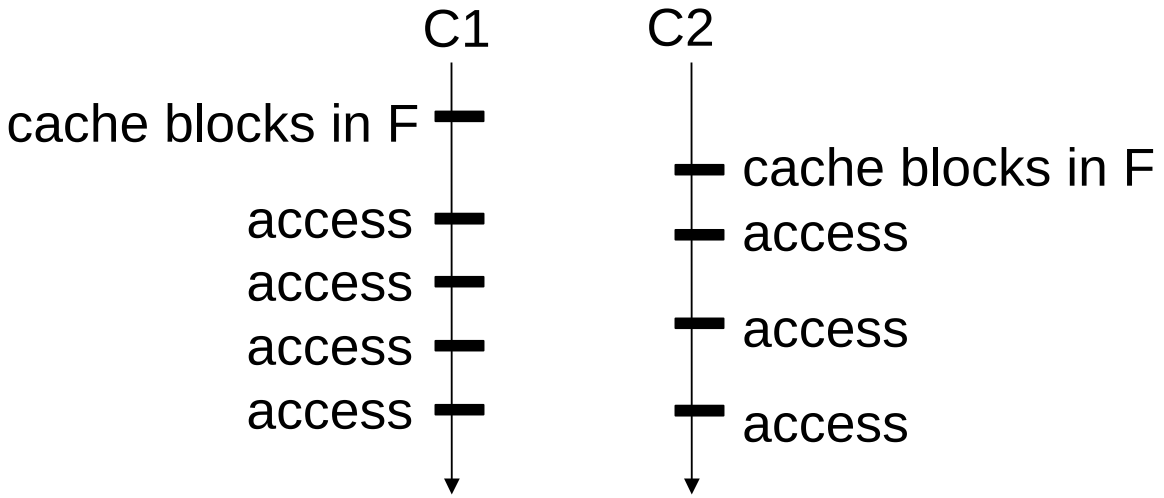
NFS's Solutions

- Stateless design
  - Flush-on-close: When file is *closed*, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Stateless protocol: requests specify exact state. read() -> read([position]). no seek on server.

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
  - (See the level of indirection we've added with this ID? ☺)

Caching Problem 2: Consistency

- If we allow client to cache parts of files, directory metadata, etc.
  – What happens if another client modifies them?



- 2 readers: no problem!
- But if 1 reader, 1 writer: inconsistency problem!

NFS's Solution: Weak Consistency

- NFS flushes updates on close()
- How does other client find out?
- NFS's answer: It checks periodically.
  - This means the system can be inconsistent for a few seconds: two clients doing a read() at the same time for the same file could see different results if one had old data cached and the other didn't.

Design Choice

- Clients can choose a stronger consistency model:
  close-to-open consistency
  - How? Always ask server for updates before open()
  - Trades a bit of scalability / performance for better consistency (getting a theme here? ☺)

What about Multiple Writes?

- NFS provides no guarantees at all!
- Might get one client's writes, other client's writes, or a mix of both!

NFS Summary

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it’s gotten a little more complex over time)
- Weak consistency semantics
- Requires hefty server resources to scale (flush-on-close, server queried for lots of operations)

Let's Look at AFS Now

- NFS addresses some of the challenges, but
  - Doesn't handle scale well (one server only)
  - Is very sensitive to network latency
- How does AFS improve this?
  - More aggressive caching (AFS caches on disk in addition to RAM)
  - Prefetching (on open, AFS gets entire file from server, making subsequent ops local & fast)

How to Cope with That Caching?

- Close-to-open consistency only
  - Why does this make sense? (Hint: user-centric workloads)

- Cache invalidation callbacks
  - Clients register with server that they have a copy of file
  - Server tells them: “Invalidate!” if the file changes
  - This trades server-side state (read: scalability) for improved consistency

AFS Summary

- Lower server load than NFS
  - More files cached on clients
  - Cache invalidation callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine's memory over a LAN
- For both, central server is:
  - A bottleneck: reads and writes hit it at least once per file use;
  - A single point of failure;
  - Expensive: to make server fast, beefy, and reliable, you need to pay $$$.

Today's Bits

- Distributed filesystems always involve a tradeoff: consistency, performance, scalability.

- We've learned a lot since NFS and AFS (and can implement faster, etc.), but the general lesson holds. *Especially* in the wide-area.
  – We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff (Consistency, Availability, Partition-resilience)

More Bits

- Client-side caching is a fundamental technique to improve scalability and performance
  – But raises important questions of cache consistency
- Periodic refreshes and callbacks are common methods for providing (some forms of) consistency
  – We’ll talk about consistency more formally in future
- AFS picked close-to-open consistency as a good balance of usability (the model seems intuitive to users), performance, etc.
  – Apps with highly concurrent, shared access, like databases, needed a different model

Next Time

- Another distributed file system, oriented toward other types of workloads (big-data/big-application workloads): the Google File System