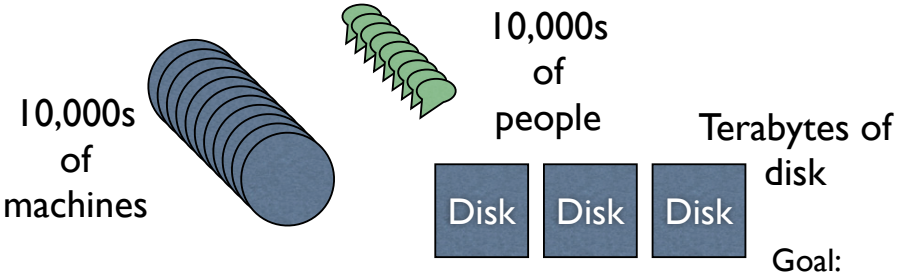
Distributed Filesystems

andrew.cmu.edu

- Let's start with a familiar example: andrew



Goal:

Have a consistent namespace for files across computers

Allow any authorized user to access their files from any computer

Challenges

- Remember our initial list of challenges...
- Heterogeneity (lots of different computers & users)
- Scale (10s of thousands of peeps!)
- Security (my files! hands off!)
- Failures
- Concurrency
- oh no... we've got 'em all.

How can we build this??

Just as important: non-challenges

- Geographic distance and high latency
  - Andrew and AFS target the campus network, *not* the wide-area

Prioritized goals?

- Often very useful to have an explicit list of prioritized goals. Distributed filesystems almost always involve trade-offs
- *Scale, scale, scale*
- *User-centric workloads*... how do users use files (vs. big programs?)
  - Most files are personally owned
  - Not too much concurrent access; user usually only at one or a few machines at a time
  - Sequential access is common; reads much more common that writes
  - There is locality of reference (if you've edited a file recently, you're likely to edit again)

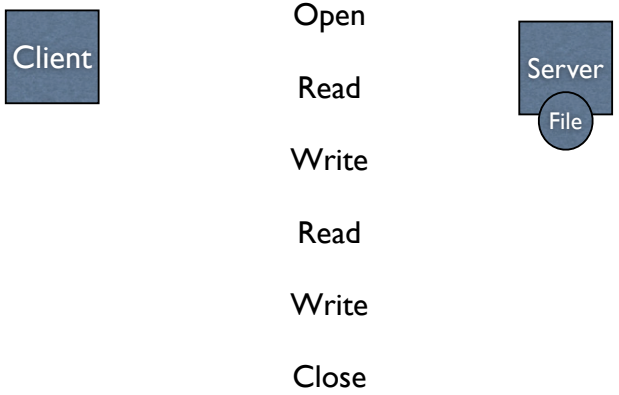
We'll see over several lectures how these design goals play out.

Fault Tolerance

- Many options...
  - Do nothing --> NFS
  - Hot, consistent replicas (every change affects multiple servers in case one dies)
  - Consistent snapshots (think "a backup of the filesystem" made easier with help from the filesystem) --> AFS initial design

How?

Single file



and directory ops

- Create file
- create directory
- rename file
- delete file
- delete directory

Approach 1: Simple

- Use RPC to forward *every* filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- For andrew context: bad bad 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?

- Huge parts of systems rely on two solutions to every problem:
  - 1) "All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem." -- David Wheeler
  - 2) Cache it!
- So, uh, what do we cache?
- And if we cache... doesn't that risk making things inconsistent?

Sun NFS

- Cache file blocks, file headers, etc., at both clients and servers.
- Advantage: No network traffic if open/read/write/close can be done locally. Woot.
- But: failures and cache consistency.
  - NFS trades some consistency for increased performance... let's look at the protocol.

Failures

- Server crashes
  - Data in memory but not disk lost
  - So... what if client does seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail. Ditto for open(), read()
- Lost messages: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo anew?
- Client crashes
  - Might lose data in client cache

NFS's answers

- Stateless design
  - Write-through caching: 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.

NFS and Failures

- You can choose -
  - retry until things get through to the server
  - return failure to client
- Most client apps can't handle failure of close() call. NFS tries to be a transparent distributed filesystem -- so how can a write to local disk fail? And what do we do, anyway?
- Usual option: hang for a long time trying to contact server

But... caching?

- If we allow client to cache parts of files, file headers, etc.
  - What happens if another client modifies them?
  - [picture]
- 2 readers: no problem!
- But now .. timeline of 1 reader, 1 writer

NFS: Weak Consistency

- NFS writes through at 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 before open()
  - Trades a bit of scalability 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!

Results

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

Let's look back at Andrew

- NFS gets us partway there, but
  - Probably doesn't handle scale (* - you can buy huge NFS appliances today that will, but they're $$$-y).
  - Is very sensitive to network latency
- How can we improve this?
  - More aggressive caching (AFS caches on disk in addition to just in memory)
  - Prefetching (on open, AFS gets entire file from server, making later ops local & fast).
    - Remember: with traditional hard drives, large sequential reads are much faster than small random writes. So easier to support (client a: read whole file; client B: read whole file) than having them alternate. Improves scalability, particularly if client is going to read whole file anyway eventually.

How to cope with that caching?

- Close-to-open consistency only (remember: user-centric!)
- Callbacks! Clients register with server that they have a copy of file;
  - Server tells them: "Invalidate!" if the file changes
  - This trades state for improved consistency
  - Soooo: What if server crashes?
    - Reconstruct: Ask all clients "dude, what files you got?"

Results

- Lower server load than NFS
  - More files cached on clients
  - 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 LAN
- For both:
  - Central server is bottleneck: all reads and writes hit it at least once;
  - is a single point of failure.
  - is spendy: make them fast, beefy, and reliable. $$$ servers.

Today's bits

- Distributed filesystems almost 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
- Timeouts and callbacks are common methods for providing (some forms of) consistency.
- AFS picked close-to-open consistency as a good balance of usability (the model seems intuitive to users), performance, etc.
  - AFS authors argued that apps with highly concurrent, shared access, like databases, needed a different model