Distributed Systems

CSC 343, Operating Systems

Topics covered in this lecture

- Distributed Systems
- Network File System
- Andrew File System

This slide deck covers chapters 48 - 50 in OSTEP.

File System Case Studies

- Local
  - FFS: Fast File System
  - LFS: Log-Structured File System
- Network
  - NFS: Network File System
  - AFS: Andrew File System

Atomicity

- Atomicity means we do not get interrupted when partially done (or at least that we can make it appear that way to the user.)
- Concurrency: we are worried about other threads
- Persistence: we are worried about crashes

Atomic Update

- Suppose we want to update a file `foo.txt`; if we crash we want one of the following:
  - all old data
  - all new data
- Strategy: write new data to `foo.tmp` and only after that is complete replace `foo.txt` by switching names

Protocols

- Bad protocol
  - copy `foo.txt` to `foo.tmp` (with changes)
  - rename `foo.tmp` to `foo.txt` (new data is in RAM and not on disk)
- Good protocol
  - copy `foo.txt` to `foo.tmp` (with changes)
  - fsync `foo.tmp`
  - rename `foo.tmp` to `foo.txt`

Local FS Comparison

- FFS + Journal:
  - must write data twice (writes expensive)
  - can put data exactly where we like (reads cheaper)
- LFS:
  - all writes sequential (writes cheaper)
  - reads may be random (reads expensive)

Distributed Systems

- Basic definition: more than one machine
- Examples:
  - client/server: web server and we client
  - cluster: page rank computation
- Why go distributed?
  - more compute power
  - more storage capacity
  - fault tolerance
  - data sharing

New Challenges

- System failure: need to worry about partial failure
- Communication failure: links unreliable

Communication

- All communication is inherently unreliable
- Need to worry about
  - bit errors
  - packet loss
  - node/link failure

Raw Messages: UDP

- API:
  - reads and writes over socket file descriptors
  - messages sent from/to ports to target a process on machine
- Provide minimal reliability features
  - messages may be lost
  - messages may be reordered
  - messages may be duplicated
  - only protection: checksums

Raw Messages: UDP

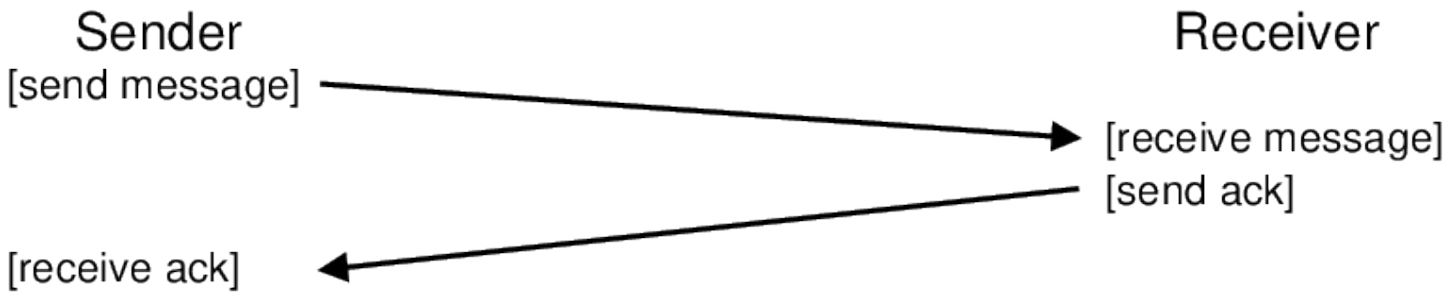
- Advantages
  - lightweight
  - some applications make better reliablity decisions themselves (for example, video conferencing programs)
- Disadvantages
  - more difficult to write the application correctly

Reliable Messages

- Strategy: using software, build reliable, logical connections over unreliable connections

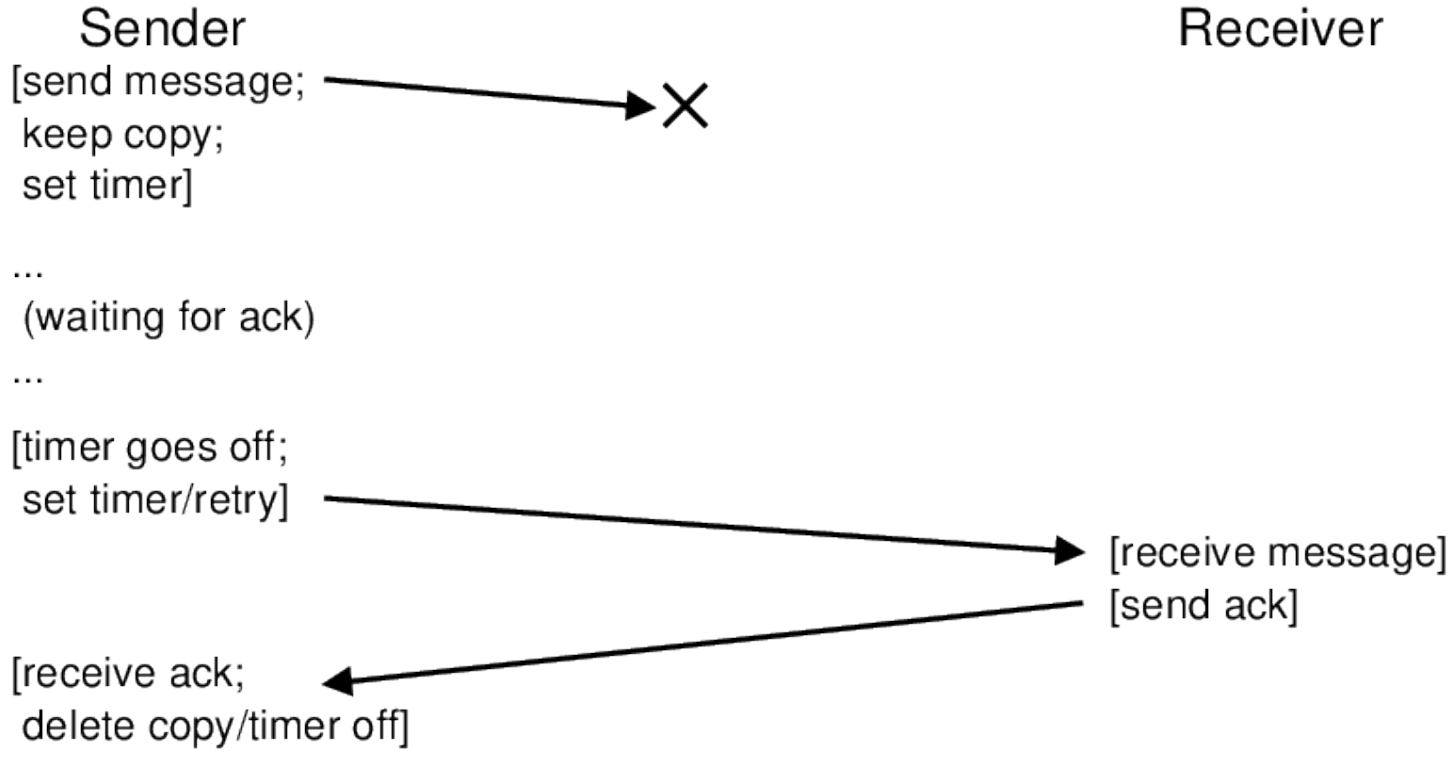
Acknowledgment (ACK)

- Sender knows message was received



Timeout

- Sender misses ACK; resend after timeout



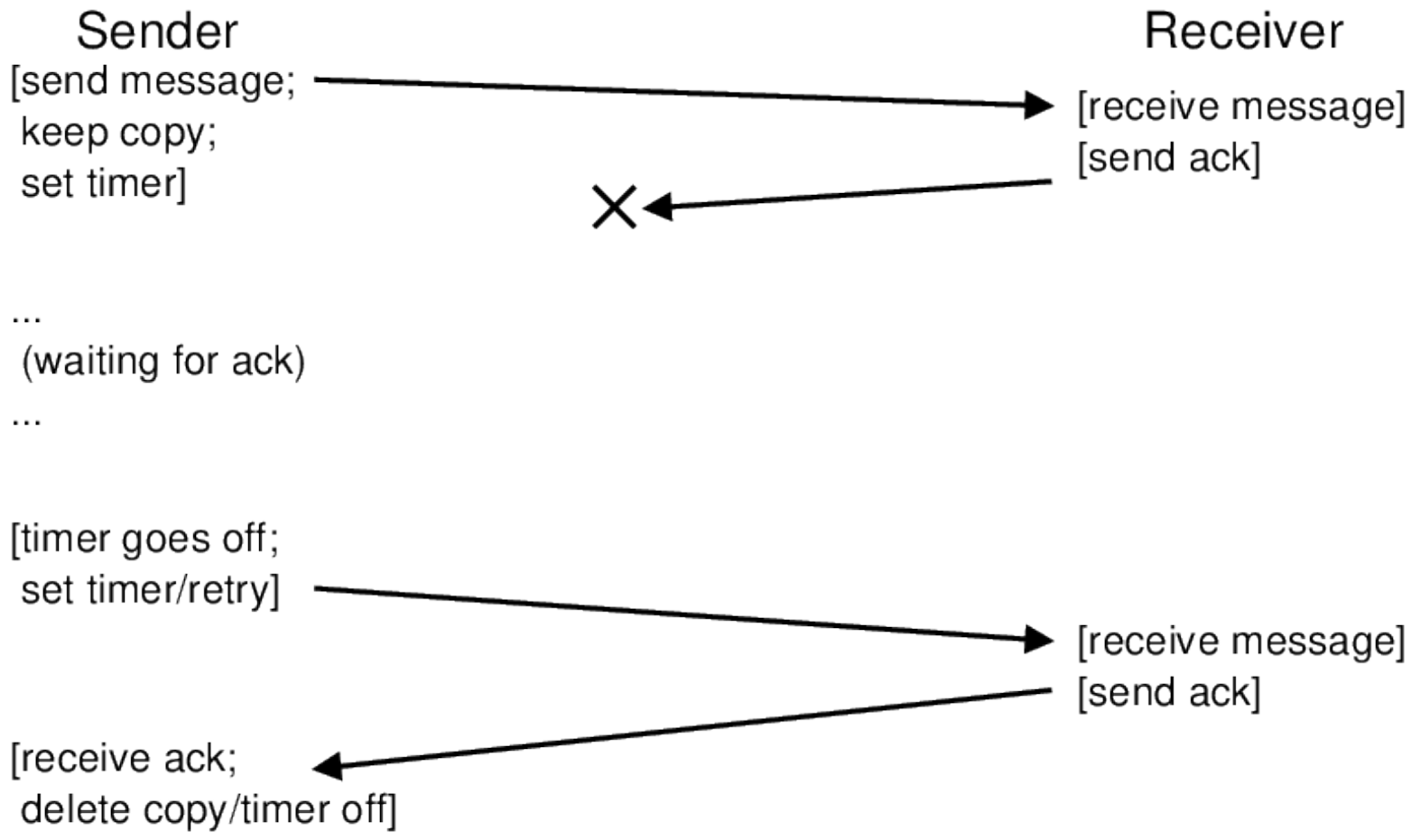
Timeout

- How long to wait?
  - too long: system feels unresponsive
  - too short: messages needlessly resent
- Messages may have dropped due to overloaded server; aggressive clients make the situation worse
- One strategy: be adaptive:
  - adjust time based on how long acks usually take
  - for each missing ack, wait longer between retries

Timeout

- What does a lost ack really mean?
- ACK: message received exactly once
- No ACK: message received at most once

Receiver Remembers Messages



Receiver Remembers Messages

- Solution 1: remember every message ever sent
- Solution 2: sequence numbers
  - give each message a sequence number, N
  - receiver knows all messages before N have been seen
  - receiver remembers message sent after N

TCP

- Most popular protocol based on sequence numbers
- Buffers messages so they arrive in order
- Timeouts are adaptive

Virtual Memory

- Inspiration: threads share memory
- Idea: processes on different machines share memory
- Strategy:
  - similar to swapping
  - instead of swap to disk, swap to another machine
  - sometimes multiple copies may be in memory on different machines

Virtual Memory Problems

- What if a machine crashes?
  - mapping disappears in other machines
  - how do we handle this?
- Performance
  - when to prefetch?
  - loads/stores expected to fast
- Distributed shared memory (DSM) not used today

Global File System

- Advantages
  - file access is already expected to be slow
  - use common API
  - no need to modify applications (for the most part, file locks over NFS do not work)
- Disadvantages
  - does not always make sense

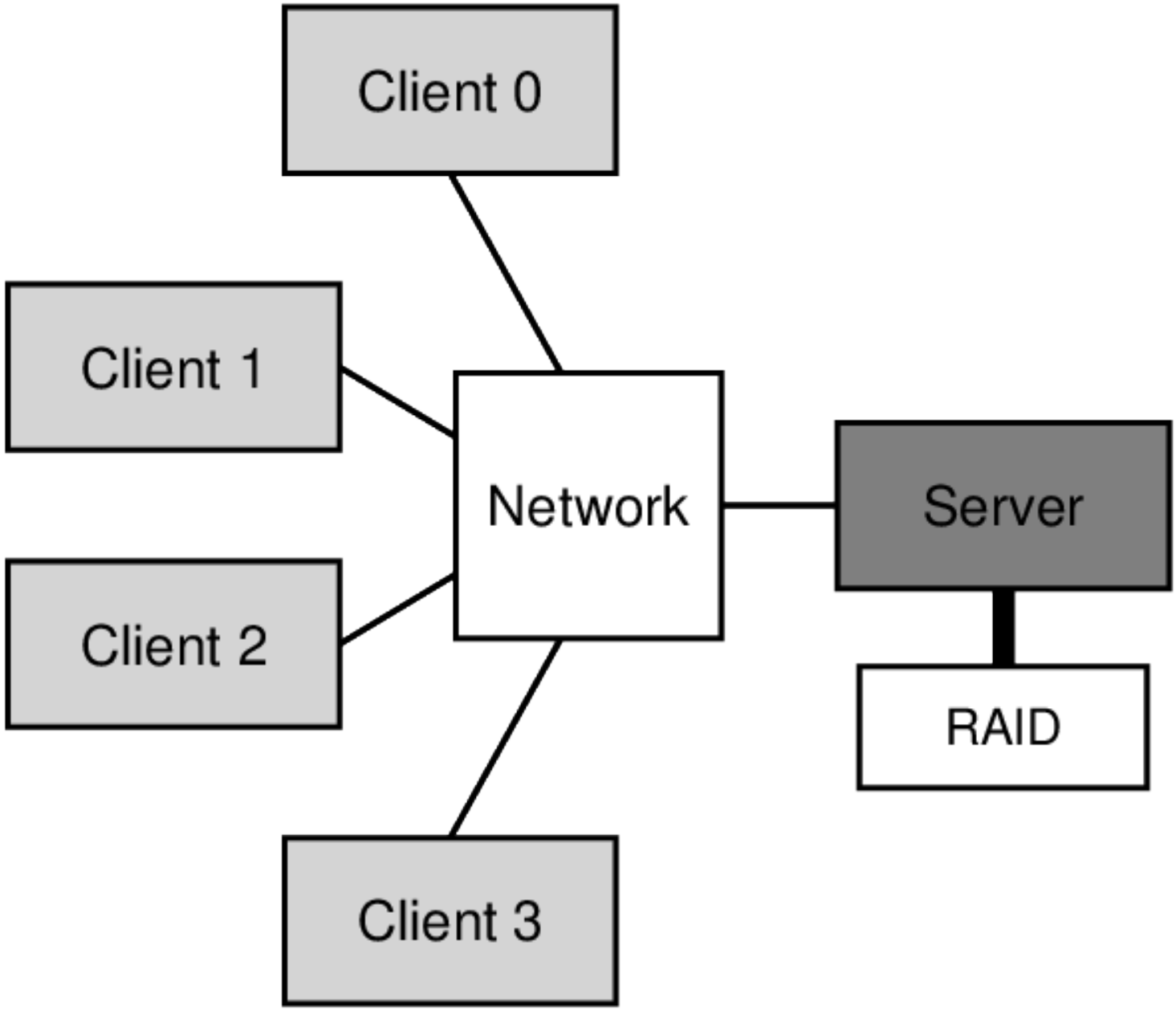
Remote Procedure Call (RPC)

- Strategy: create wrappers so calling a function on another machine feels like calling a local function
- This abstraction is common

RPC Tools

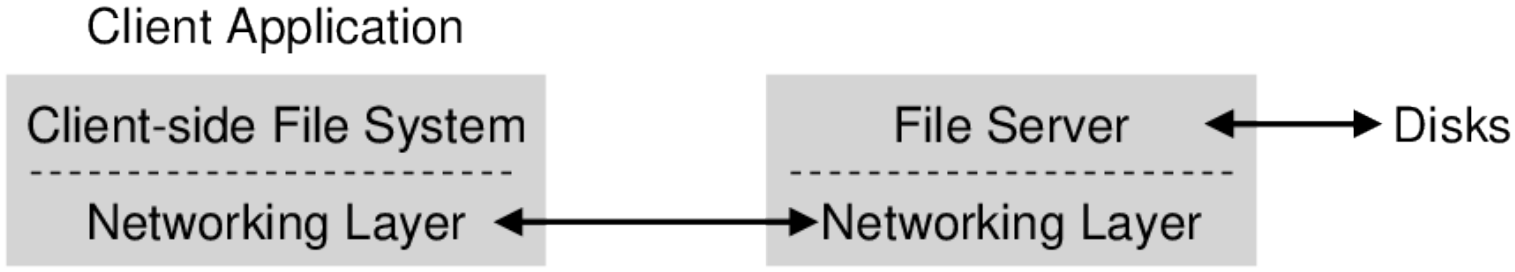
- RPC packages help with two components
- Stub generation
  - create wrappers automatically
- Runtime library
  - thread pool
  - socket listeners call functions on the server

Network File System (NFS) Architecture



Main Design Decisions

- What functions to expose via RPC?
- Think of NFS as more a protocol than a particular file system



Strategy 1

- Wrap regular UNIX system calls using RPC
  - `open()` on client calls `open()` on server
  - `open()` on server returns file descriptor to client
  - `read()` on client calls `read()` on server
  - `read()` on server returns data back to client

Strategy 1 Problems

- Imagine the server crashes and reboots during reads

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
...
read(fd, buf, MAX);
```

Subgoals

- Fast and simple crash recovery
  - both clients and file server may crash
- Transparent access
  - cannot tell that it is over the network
  - normal UNIX semantics
- Reasonable performance

Potential Solutions

- Run some crash recovery protocol upon reboot
  - complex
- Persist file descriptors on server disk
  - slow
  - what if client crashes instead?

Strategy 2

- Put all the information in the requests; use a "stateless" protocol
  - server maintains no state about clients
  - server keeps other state, of course
- Need API change, for example specify path and offset each time:
  - `pread(char *path, buf, size, offset)`
  - `pwrite(char *path, buf, size, offset)`
- Problem: too many path lookups

Strategy 3

- Request inode
  - `inode = open(char* path)`
  - `pread(inode, buf, size, offset)`
  - `pwrite(inode, buf, size, offset)`
- Problem: what if file is deleted and inode reused?

Strategy 4

- File handles
  - `fh = open(char* path)`
  - `pread(fh, buf, size, offset)`
  - `pwrite(fh, buf, size, offset)`
- File handle information
  - volume ID
  - inode number
  - generation number

Aside: append

- Would an `append()` be a good idea?
- Problem: if our RPC library retries if no ACK return what happens when `append()` is retried?
- Replica suppression is stateful; TCP is stateful – if the server crashes, it forgets what RPCs have been executed
- Solution: design the API so that there is no harm in executing a call more than once, that is, ensure idempotence

Strategy 5; Client Logic

- Build normal UNIX API on client side on top of the idempotent, RPC-based API we have described
- Client `open()` creates a local file descriptor containing the file handle and offset

Cache

- We can cache data in three places:
  - server memory
  - client disk
  - client memory
- How do we keep all versions in sync?

Cache Problems

- Problem: update visibility
  - a client may buffer a write
  - how can the server and other clients see it?
  - NFS solution: flush on closing a file descriptor
- Problem: stale cache
  - a client may have a cached copy that is obsolete
  - how can we get the latest?
  - if we were not trying to be stateless, the server could push out an update
  - NFS solution: clients recheck if cache is current before using it

Andrew File System (AFS)

- Primary goal: scalability (many clients per server)
- More reasonable semantics for concurrent file access
- Not good about handling some failure scenarios

AFS Design

- NFS: export local FS
- AFS: present big file tree, store across many machines
  - break tree into “volumes” (partial subtrees)

Volumes

- Volumes should be coalesced into a seamless file tree
- Volume leaves may point to other volumes
- A volume database (mapping volume names to physical locations) is replicated on each server
    - clients can ask any server in the system

Volume Movement

- What if we want to migrate a volume to another machine?
- Steps:
  - copy over data (without halting I/O)
  - update volume database (without it becoming stale)

Cache Update Visibility

- Client updates not seen on servers yet
- NFS solution: flush blocks
  - on `close()`
  - when low on memory
- Problems:
  - flushes not atomic (one block at a time)
  - two clients flush at once: mixed data

Cache Update Visibility

- AFS solution:
  - flush on `close()`
  - buffer whole files on local disk
- Concurrent writes?
  - last writer wins
- Never get mixed data

Stale Cache

- Clients have old version
- NFS rechecks cache entries before using them, assuming a check has not been done "recently"
  - "recent" is too long: you read old data
  - "recent" is too short: server overloaded with `stat()` calls

Stale Cache

- AFS solution: tell clients when data is overwritten
- When clients cache data, ask for "callback" from server
- No longer stateless

Callbacks

- What if client crashes?
  - On client reboot either: evict everything from cache or recheck before using
- What if server runs out of memory?
  - tell clients you are dropping their callback
  - clients mark entry for recheck
- What if server crashes?
  - Either: tell all clients to recheck everything before next read or persist callbacks

Prefetching

- AFS paper notes that most files are read in their entirety
- What are the implications for prefetching?
  - Aggressively prefetch whole files

Whole-File Caching

- Upon `open`, AFS fetches whole file (even if it is huge) and stores it in local memory or disk
- Upon `close`, whole file is flushed (if it was written)
- Convenient:
  - AFS needs to do work for open/close
  - reads/writes are local

Name Resolution

- What if the same inodes and directory entries are repeatedly read?
  - cache prevents too much disk I/O
  - too much CPU though
- Solution: server returns directory entries to the client and the client caches entries and inodes
  - pro: partial traversal is the common case
  - con: first lookup requires many round trips

Process Structure

- For each client, a different process ran on the server
  - context switching costs were high
- Solution: use threads
  - shared address space results in more useful TLB entries

File Locks

- AFS has a dedicated lock server
- A client can lock a file preventing another client from accessing it while the lock is held
- NFS does not have this capability