Network File Systems

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: *Computing Systems and Concurrency*
Lecture 4

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.

Abstraction, abstraction, abstraction!

- Local file systems
  - Disks are terrible abstractions: low-level blocks, etc.
  - Directories, files, links much better
- Distributed file systems
  - Make a remote file system look local
  - Today: NFS (Network File System)
    - Developed by Sun in 1980s, still used today!

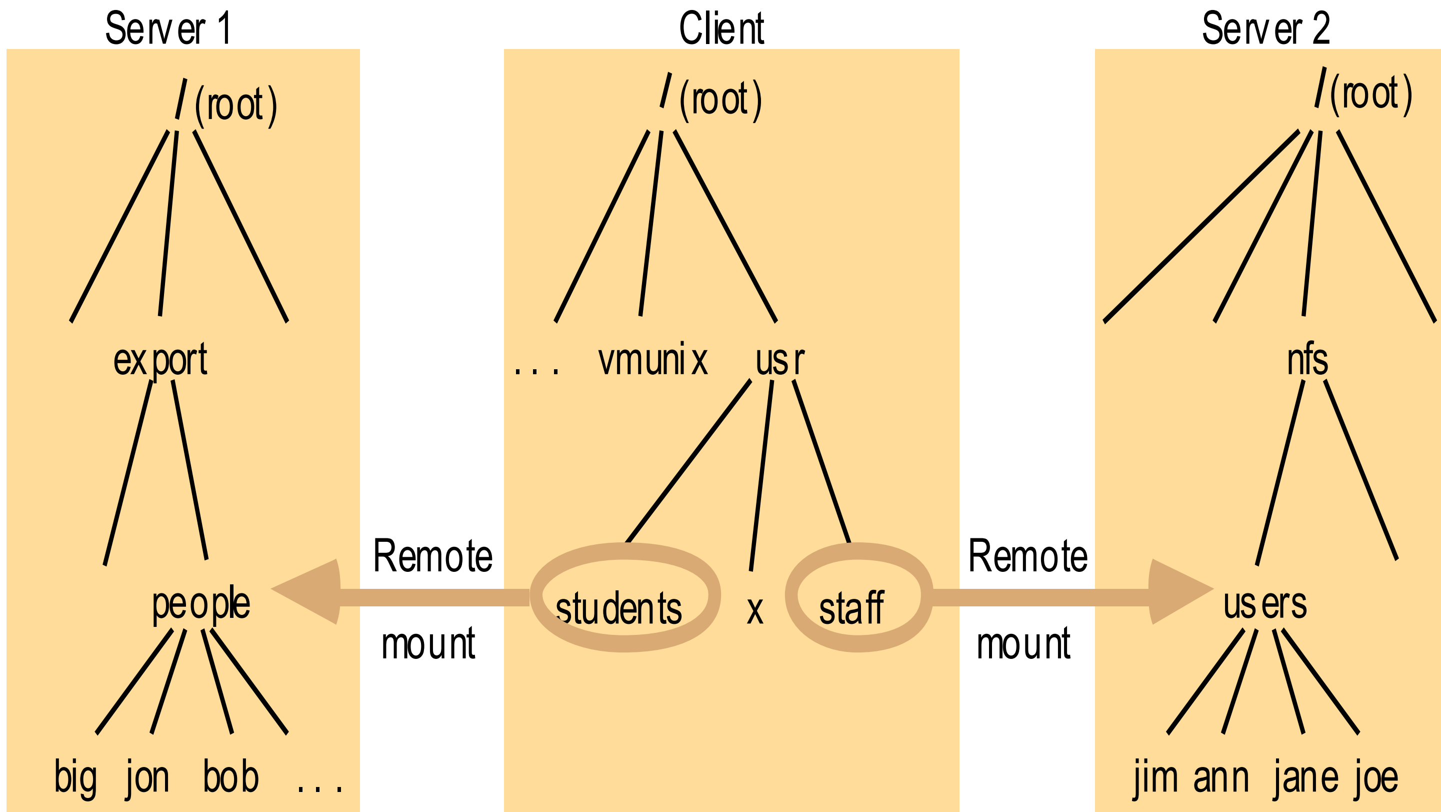
3 Goals: Make operations appear:

Local

Consistent

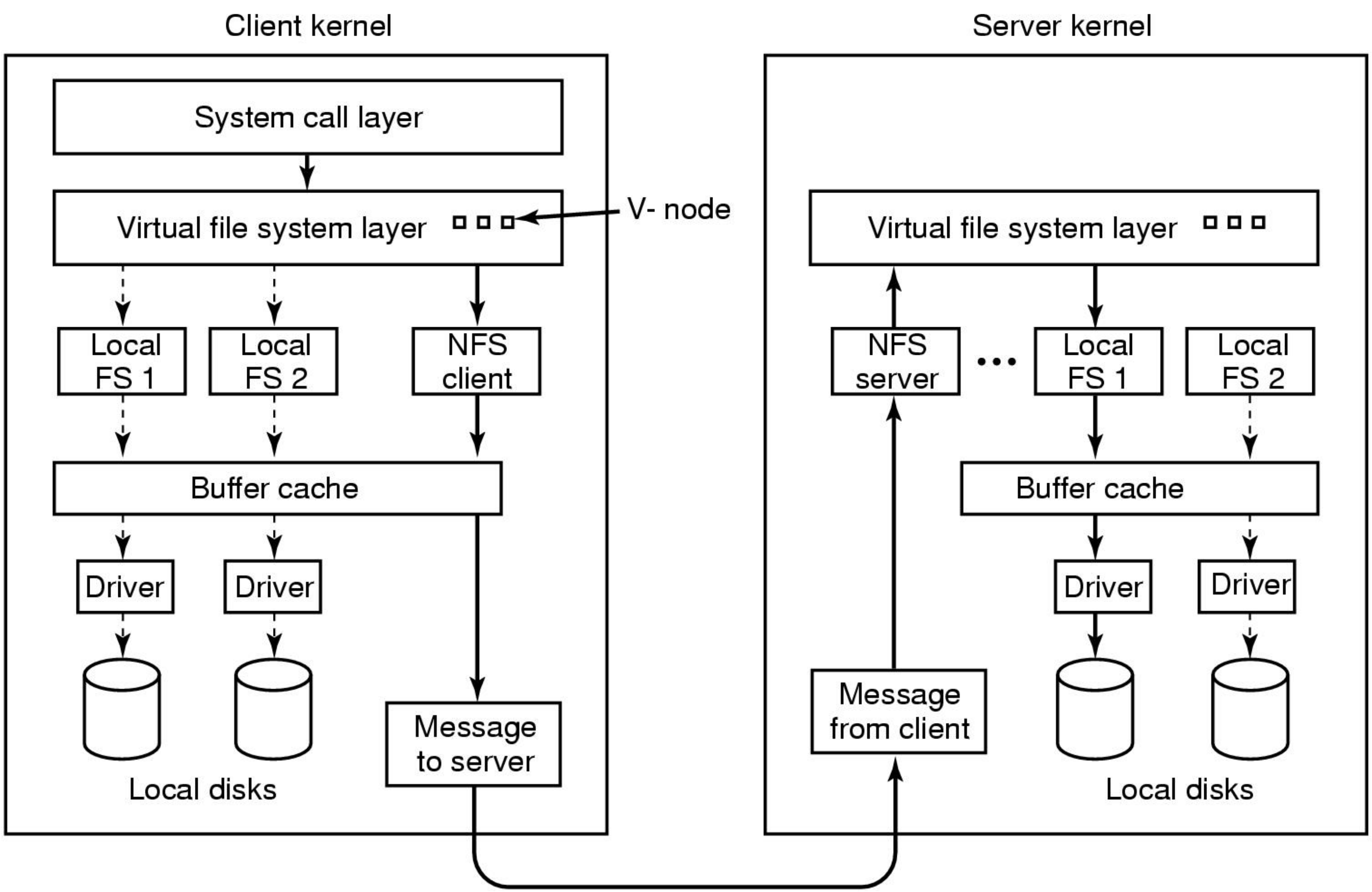
Fast

NFS Architecture



"Mount" remote FS (host:path) as local directories

Virtual File System enables transparency



Interfaces matter

VFS / Local FS

```
fd = open("path", flags)

read(fd, buf, n)

write(fd, buf, n)

close(fd)
```

Server maintains state that maps `fd` to inode, offset

Stateless NFS: Strawman 1

```
fd = open("path", flags)
```

```
read("path", buf, n)
```

```
write("path", buf, n)
```

```
close(fd)
```

Stateless NFS: Strawman 2

```
~~fd = open("path", flags)~~

read("path", offset, buf, n)

write("path", offset, buf, n)

~~close(fd)~~
```

Embed pathnames in syscalls?

	Program 1 on client 1	Program 2 on client 2
1	CHDIR ("dir1")	
2	*fd* ← OPEN ("f", READONLY)	
3		RENAME ("dir1", "dir2")
4		RENAME ("dir3", "dir1")
5	READ (*fd*, *buf*, *n*)	

Time ↓

- Should read refer to current `dir1/f` or `dir2/f` ?
- In UNIX, it's `dir2/f`. How do we preserve in NFS?

Stateless NFS (for real)

```
fh = lookup("path", flags)

read(fh, offset, buf, n)

write(fh, offset, buf, n)

getattr(fh)
```

Implemented as Remote Procedure Calls (RPCs)

NFS File Handles (`fh`)

- Opaque identifier provider to client from server

- Includes all info needed to identify file/object on server

```
volume ID |  inode # | generation #
```

- It's a trick: "store" server state at the client!

NFS File Handles (and versioning)

	Program 1 on client 1	Program 2 on client 2	Time
1		$fd \leftarrow$ OPEN ("f", READONLY)	
2	UNLINK ("f")		
3	$fd \leftarrow$ OPEN ("f", CREATE)		
4		READ (fd, buf, n)	

- With generation #'s, client 2 continues to interact with "correct" file, even while client 1 has changed " f "
- This versioning appears in many contexts, e.g., MVCC (multiversion concurrency control) in DBs

NFS example

```
fd = open(”/foo”, ...);
 Send LOOKUP (rootdir FH, ”foo”)

                          Receive LOOKUP request
                           look for ”foo” in root dir
                           return foo’s FH + attributes

Receive LOOKUP reply
 allocate file desc in open file table
 store foo’s FH in table
 store current file position (0)
 return file descriptor to application
```

NFS example

```
read(fd, buffer, MAX);
 Index into open file table with fd
  get NFS file handle (FH)
  use current file position as offset
 Send READ (FH, offset=0, count=MAX)

                 Receive READ request
                  use FH to get volume/inode num
                  read inode from disk (or cache)
                  compute block location (using offset)
                  read data from disk (or cache)
                  return data to client

Receive READ reply
 update file position (+bytes read)
 set current file position = MAX
 return data/error code to app
```

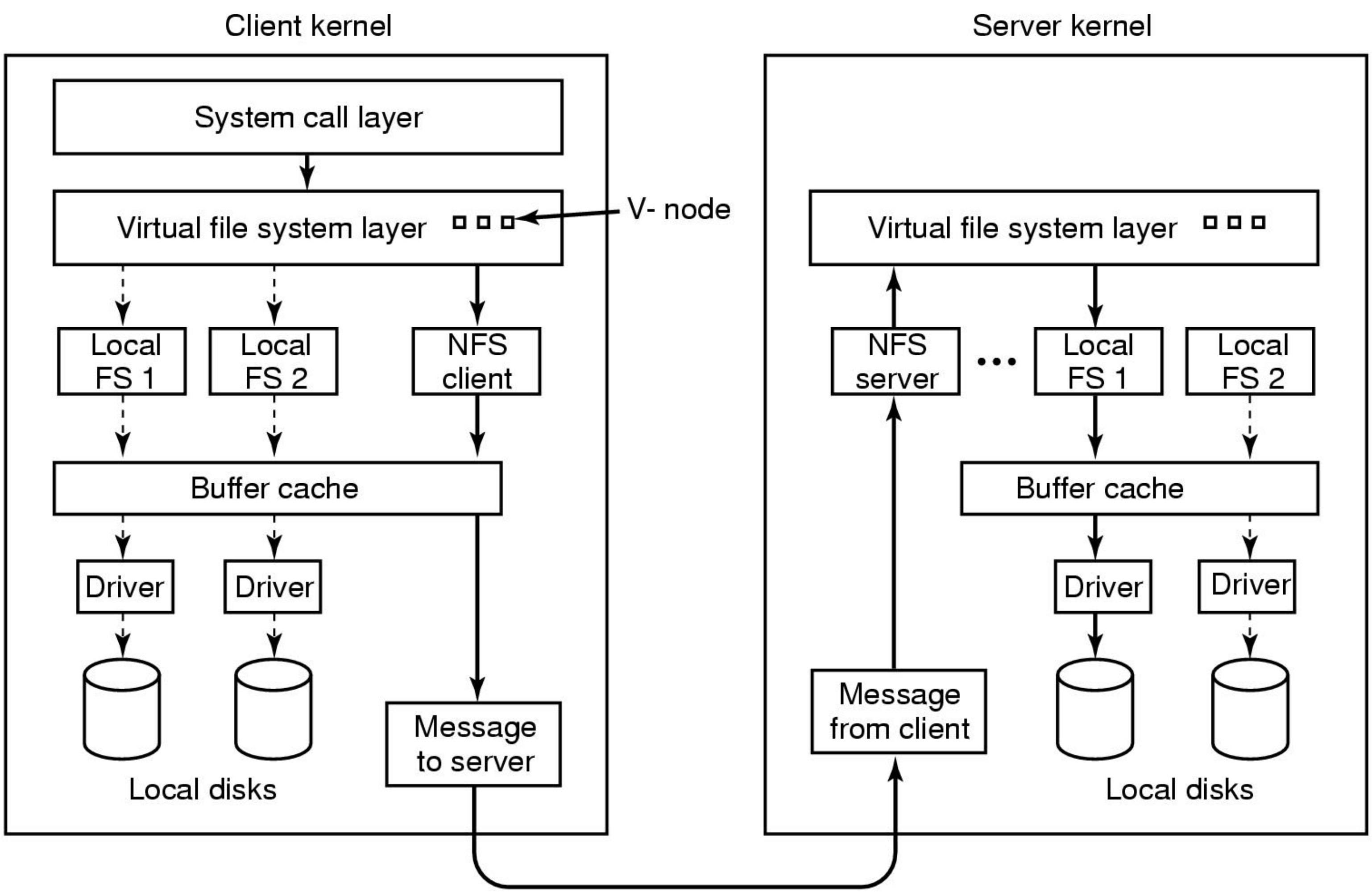
NFS example

```
read(fd, buffer, MAX);
 Same except offset=MAX and
 set current file position = 2*MAX

read(fd, buffer, MAX);
 Same except offset=2*MAX and
 set current file position = 3*MAX

close(fd);
 Just need to clean up local structures
 Free descriptor "fd" in open file table
 (No need to talk to server)
```

Handling server failures

- What to do when server is not responding?
  - Retry again!
    - set a timer; a reply before cancels the retry; else retry
- Is it safe to retry operations?
  - NFS operations are idempotent
    - the effect of multiple invocations is same as single one
  - LOOKUP, READ, WRITE: message contains all that is necessary to re-execute
  - What is not idempotent?
    - E.g., if we had INCREMENT
    - Real example: MKDIR is not

Are remote == local?



TANSTANFL

(There ain't no such thing as a free lunch)

- With local FS, `read` sees data from "most recent" `write`, even if performed by different process
  - "Read/write coherence", linearizability

All operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations. (Herlihy & Wing, 1991)

- Achieve the same w
  - Perform all reads
  - Huge cost: high la
- And what if the server doesn't return?
  - Options: hang indefinitely, return ERROR

Caching GOOD

Lower latency, better scalability

Consistency HARDER

No longer one single copy of data,

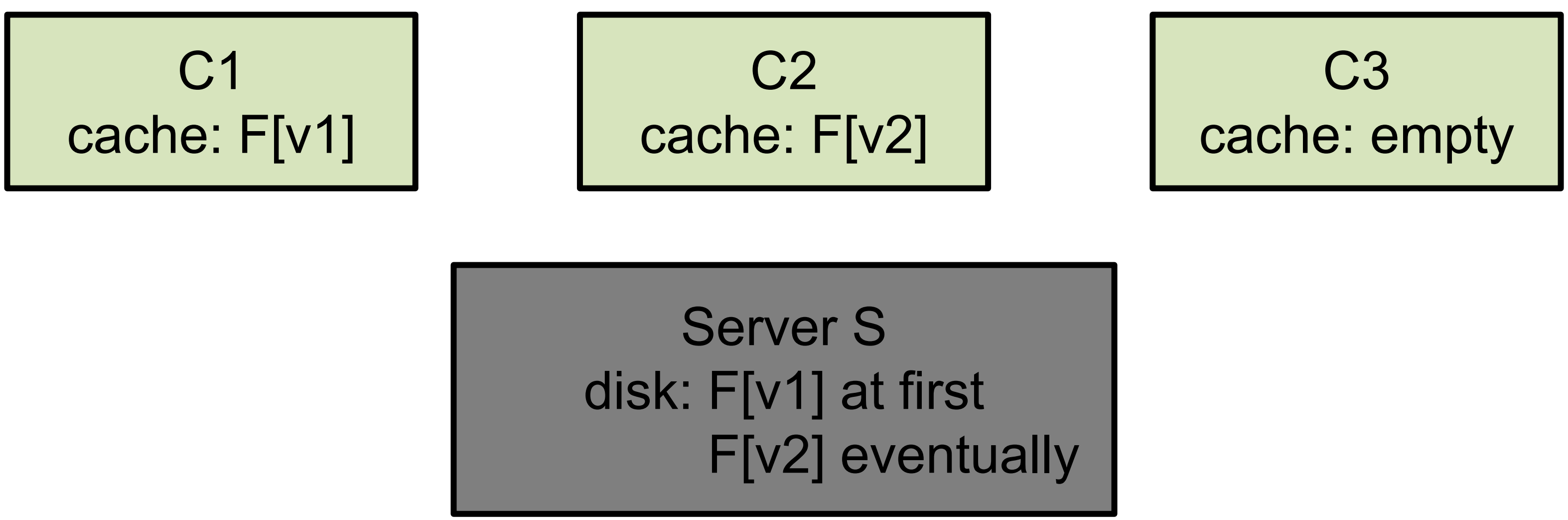
to which all operations are serialized

Caching options

- Centralized control: Record status of clients (which files open for reading/writing, what cached, …)
- Read-ahead: Pre-fetch blocks before needed
- Write-through: All writes sent to server
- Write-behind: Writes locally buffered, send as batch

Cache consistency problem

- Consistency challenges:
  - When client writes, how do others caching data get updated? (Callbacks, …)
  - Two clients concurrently write? (Locking, overwrite, …)



Should server maintain per-client state?

Stateful

- Pros
  - Smaller requests
  - Simpler req processing
  - Better cache coherence, file locking, etc.
- Cons
  - Per-client state limits scalability
  - Fault-tolerance on state required for correctness

Stateless

- Pros
  - Easy server crash recovery
  - No open/close needed
  - Better scalability
- Cons
  - Each request must be fully self-describing
  - Consistency is harder, e.g., no simple file locking

It’s all about the state, ’bout the state, …

- Hard state: Don’t lose data
  - Durability: State not lost
    - Write to disk, or cold remote backup
    - Exact replica or recoverable (DB: checkpoint + op log)
  - Availability (liveness): Maintain online replicas

- Soft state: Performance optimization
  - Then: Lose at will
  - Now: Yes for correctness (safety), but how does recovery impact availability (liveness)?

NFS

- Stateless protocol
  - Recovery easy: crashed == slow server
  - Messages over UDP (unencrypted)
- Read from server, caching in NFS client
- NFSv2 was write-through (i.e., synchronous)
- NFSv3 added write-behind
  - Delay writes until `close` or `fsync` from application

Exploring the consistency tradeoffs

- Write-to-read semantics too expensive
  - Give up caching, require server-side state, or …
- Close-to-open "session" semantics
  - Ensure an ordering, but only between application `close` and `open`, not all `writes` and `reads`.
  - If B opens after A closes, will see A's writes
  - But if two clients open at same time? No guarantees
    - And what gets written? "Last writer wins"

NFS Cache Consistency

- Recall challenge: Potential concurrent writers

- Cache validation:
  - Get file's last modification time from server: `getattr(fh)`
  - Both when first open file, then poll every 3-60 seconds
    - If server's last modification time has changed, flush dirty blocks and invalidate cache

- When reading a block
  - Validate: (current time – last validation time < threshold)
  - If valid, serve from cache. Otherwise, refresh from server

Some problems…

- "Mixed reads" across version
  - A reads block 1-10 from file, B replaces blocks 1-20, A then keeps reading blocks 11-20.
- Assumes synchronized clocks. Not really correct.
  - We'll learn about the notion of logical clocks later
- Writes specified by offset
  - Concurrent writes can change offset

Server-side write buffering

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

```
write(fd, a_buffer, size); // fill first block with a’s
write(fd, b_buffer, size); // fill second block with b’s
write(fd, c_buffer, size); // fill third block with c’s
```

Expected:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

But assume server buffers 2nd write, reports OK but then crashes:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Server **must** commit each write to stable (persistent) storage before informing the client of success

When statefulness helps

Callbacks
Locks + Leases

NFS Cache Consistency

- Recall challenge: Potential concurrent writers
- Timestamp invalidation: NFS
- Callback invalidation: AFS, Sprite, Spritely NFS
  - Server tracks all clients that have opened file
  - On write, sends notification to clients if file changes; client invalidates cache
- Leases: Gray & Cheriton ’89, NFSv4

Locks

- A client can request a lock over a file / byte range
  - Advisory: Well-behaved clients comply
  - Mandatory: Server-enforced
- Client performs writes, then unlocks
- Problem: What if the client crashes?
  - Solution: Keep-alive timer: Recover lock on timeout
    - Problem: what if client alive but network route failed?
      - Client thinks it has lock, server gives lock to other: "Split brain"

Leases

- Client obtains ***lease*** on file for read or write
  - "A lease is a ticket permitting an activity; the lease is valid until some expiration time."
- Read lease allows client to cache clean data
  - *Guarantee:* no other client is modifying file
- Write lease allows safe delayed writes
  - Client can locally modify then batch writes to server
  - *Guarantee:* no other client has file cached

Using leases

- Client requests a lease
  - May be implicit, distinct from file locking
  - Issued lease has file version number for cache coherence
- Server determines if lease can be granted
  - *Read leases* may be granted concurrently
  - *Write leases* are granted exclusively
- If conflict exists, server may send *eviction* notices
  - Evicted write lease must write back
  - Evicted read leases must flush/disable caching
  - Client acknowledges when completed

Bounded lease term simplifies recovery

- Before lease expires, client must *renew* lease
- Client fails while holding a lease?
  - Server waits until the lease expires, then unilaterally reclaims
  - If client fails during eviction, server waits then reclaims
- Server fails while leases outstanding? On recovery,
  - Wait *lease period* + *clock skew* before issuing new leases
  - Absorb renewal requests and/or writes for evicted leases

Requirements dictate design

Case Study: AFS

Andrew File System (CMU 1980s-)

- Scalability was key design goal
  - Many servers, 10,000s of users
- Observations about workload
  - Reads much more common than writes
  - Concurrent writes are rare / writes between users disjoint
- Interfaces in terms of files, not blocks
  - *Whole-file serving*: entire file and directories
  - *Whole-file caching:* clients cache files to local disk
    - Large cache and permanent, so persists across reboots

AFS: Consistency

- Consistency: Close-to-open consistency
  - No mixed writes, as whole-file caching / whole-file overwrites
  - Update visibility: Callbacks to invalidate caches

- What about crashes or partitions?
  - Client invalidates cache iff
    - Recovering from failure
    - Regular liveness check to server (heartbeat) fails.
  - Server assumes cache invalidated if callbacks fail + heartbeat period exceeded

Next lecture topic:

Google File System (GFS)