Networked File System

CS333

Williams College

This Video

NFS: Network File System

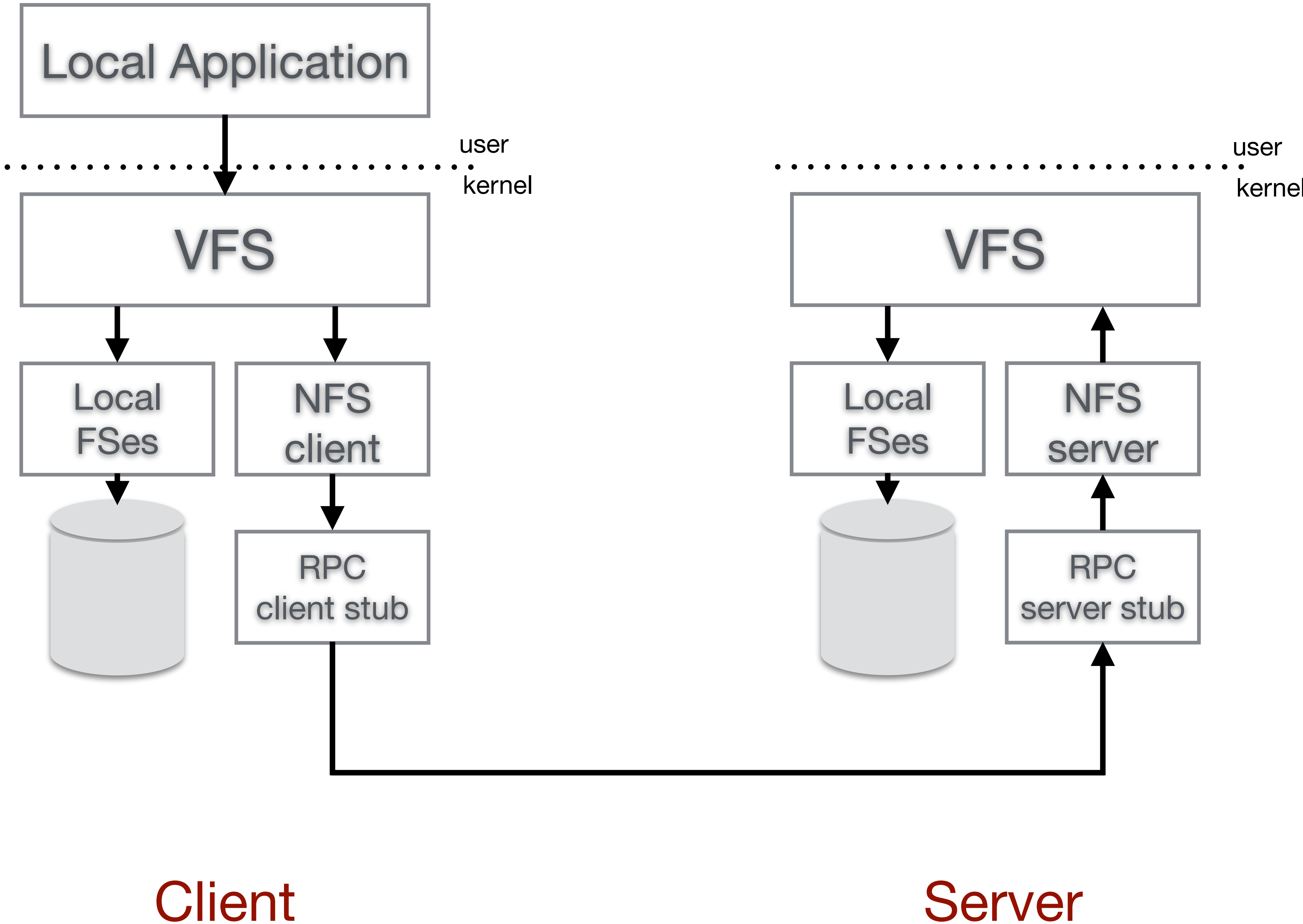
- Statelessness vs. statefulness?
- What is idempotency and why are idempotent operations desirable?
- Who caches what, and what are the implications?
- What consistency guarantees does NFS give to clients?
- What happens when an NFS client crashes? An NFS server?

NFS v2: High-level idea

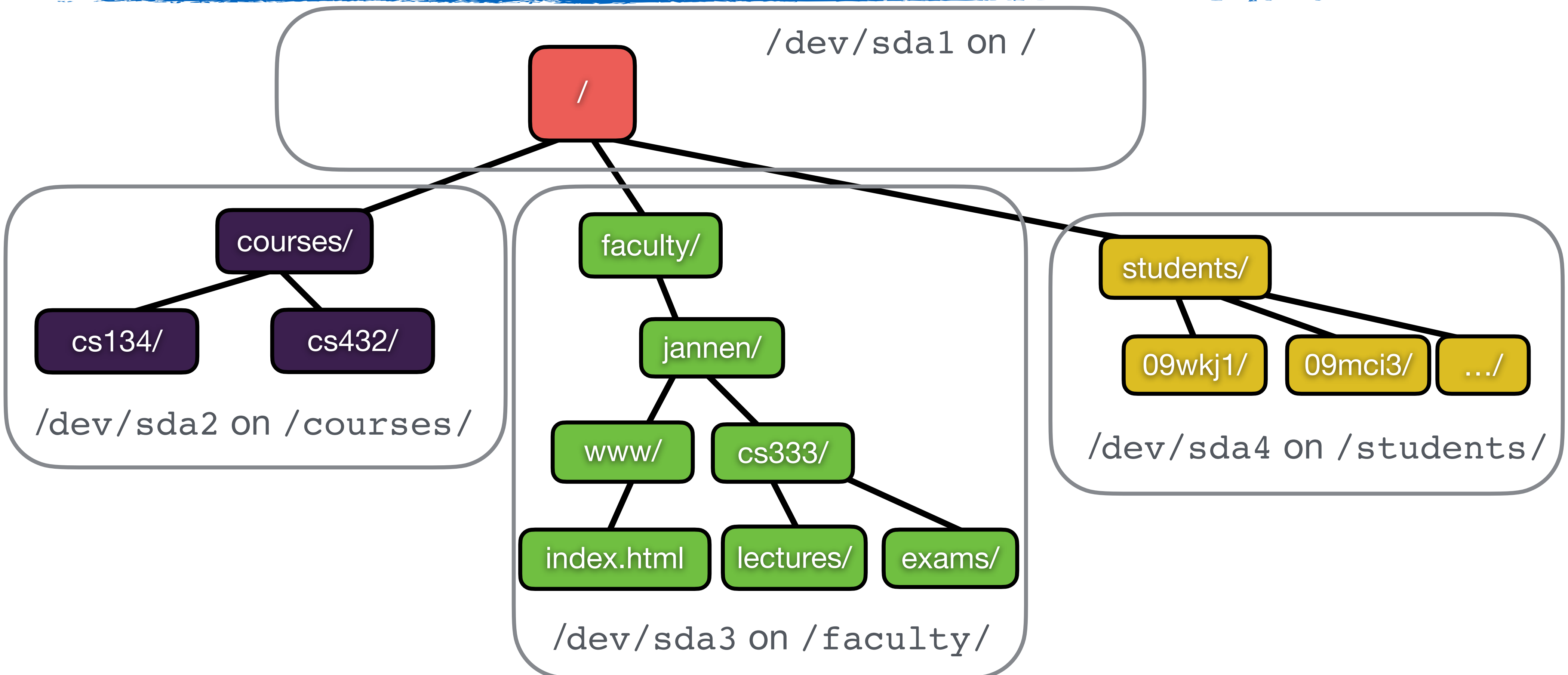
- **Clients connect to an NFS server over a network**
- **Each client's local VFS operations are translated into a series of network requests**
- **The server receives requests, makes corresponding changes to its local file system, and sends responses back to clients**
  - Can be the acknowledgement of a successful write, data from a read request, an error, etc.

The fact that NFS uses the client-server model is transparent to client applications: they think they are running on a local FS

NFS Big Picture



NFS Big Picture: CS Dept. Server

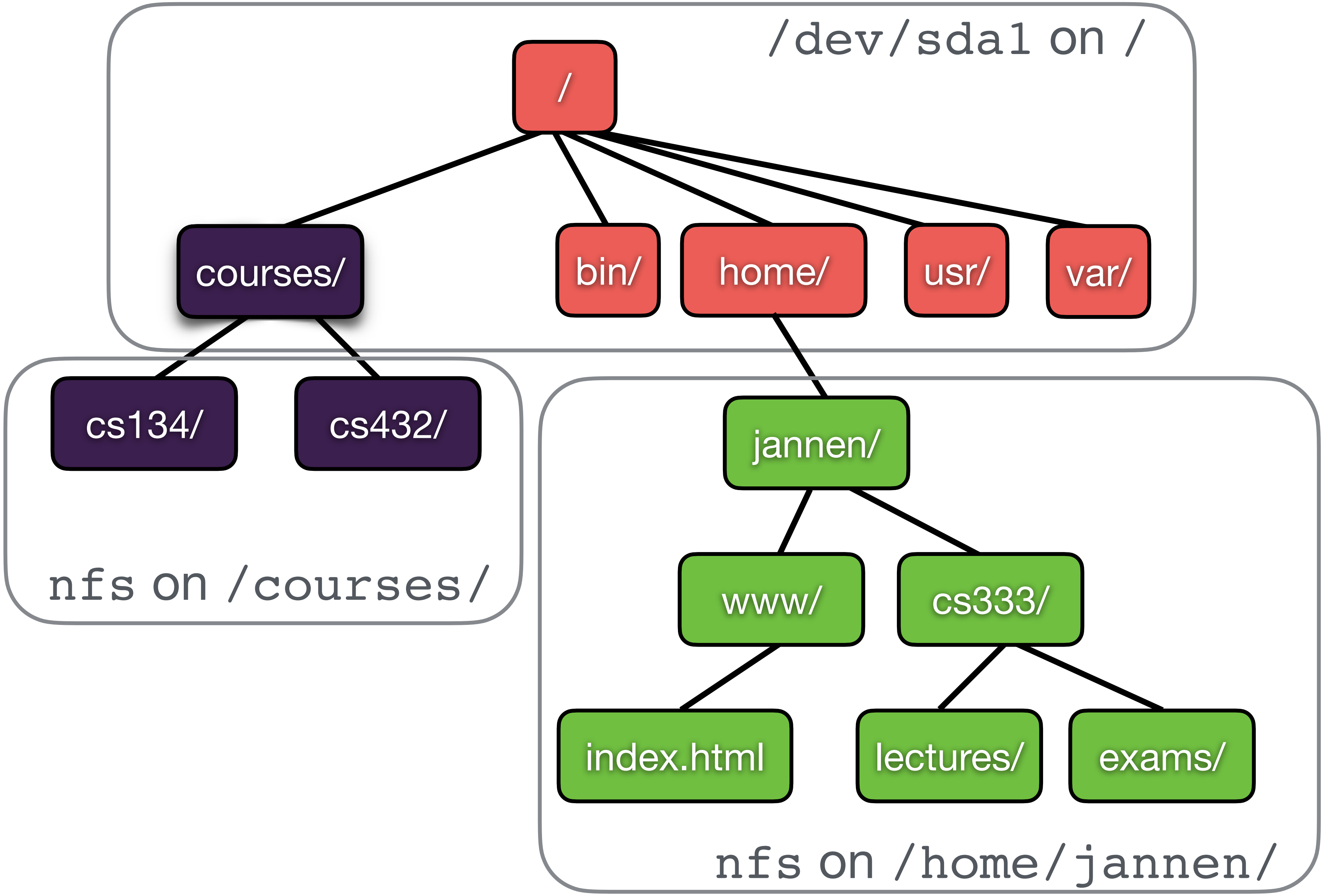


NFS servers export subtrees to clients.

/etc/exports contains nfs server settings:

- `/faculty/jannen` exported RW to userid:jannen on dexter.cs.williams.edu, speckle.cs.williams.edu, …
- `/courses/`exported RW to userid:jannen on dexter.cs.williams.edu, speckle.cs.williams.edu, …
- `/students/09mci3/`exported RW to userid:09mci3 on dexter.cs.williams.edu, speckle.cs.williams.edu, …

NFS Big Picture: CS Lab Client



Clients mount nfs volumes into their FS namespace

NFSv2 Statelessness

POSIX is stateful, NFSv2 is stateless

- What challenges does this bring?

An NFSv2 server has no notion of a "connection"

- The client bundles all necessary "state" into each message
  - NFS **file handle** stores volume, inode #, generation #
- The server can then view each client message in isolation
  - If a client disconnects, the server does not know or care
  - If the server crashes, the client does not lose any relied-upon state
- Statelessness makes recovery and error-handling easier

NFSv2 Statelessness

Idempotent operations

- An operation is idempotent if performing the operation multiple times has the same effect as performing it once
  - Many POSIX system calls are not idempotent:
    - read(fd, buf, nbytes), write(fd, buf, nbytes), lseek(fd, offset, SEEK_CUR), etc.
  - By specifying the starting offset with each read/write, many operations can be made idempotent
    - Notable exceptions?
      - mkdir fails if the file exists

Idempotency simplifies the protocol considerably

- If a message is dropped, send it again
- If the server crashes, clients can resend all unacknowledged operations

Caching: Who and What?

Network round trips are expensive, so clients would like to cache data locally

- Satisfy reads from cache, rather than pay for read+RTT
- Buffer writes locally, and send updates in large messages

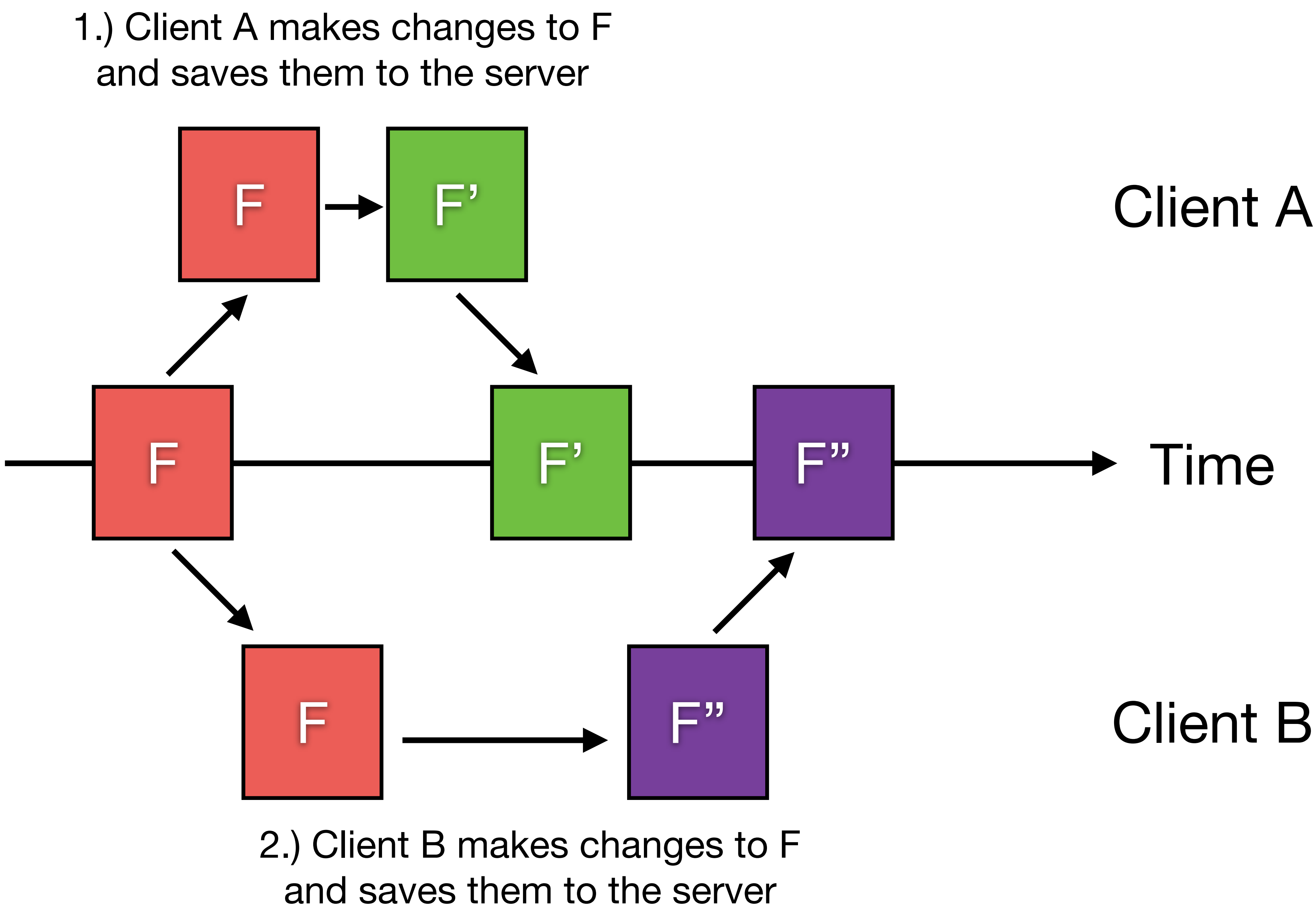
There are challenges to client-side caching: multiple clients can't share data effectively without coordination

- NFSv2 is supposed to appear as a local FS
- When is local cache stale w.r.t. server?
- When should you push your updates to server?

NFSv2 solutions:

- Flush-on-close / close-to-open consistency
- Attribute cache

Close-to-open Consistency



Attribute Cache

NFS sends GETATTR requests to check the validity of its cache

- The result indicates whether or not the server has a more recent version of the file than the client's cached version.

Problem: GETATTR messages can flood the network just to confirm the common case

Clients often issue GETATTR requests every N seconds

- This reduces GETATTR traffic to a (tunable) volume
- This also bounds the cache's staleness, but introduces a window of vulnerability

NFS is an Evolving Protocol

NFS has evolved over many years:

- NFS v1: internal SUN protocol
- NFS v2: 1989
- NFS v3: 1995
- NFS v4: 2000
- NFS v4.1: 2010
- NFS v4.2: 2016

What's changed?

- Statelessness has given way to statefulness in NFSv4
  - The "purity" of the v2 design has eroded in favor of performance & security

Takeaways

NFS fills a very common need: efficient access to shared files on a reliable, low-latency network

Building a stateless protocol that implements a stateful API adds challenges

- Caching helps performance but creates coordination challenges
  - Close-to-open consistency
  - Attribute caching
- Idempotent operations simplify the protocol by bundling state with each request
  - This also simplifies the recovery process:
    - If a message is lost, send it again
    - If the server crashes, clients can resend all unacknowledged requests