Winter 2025

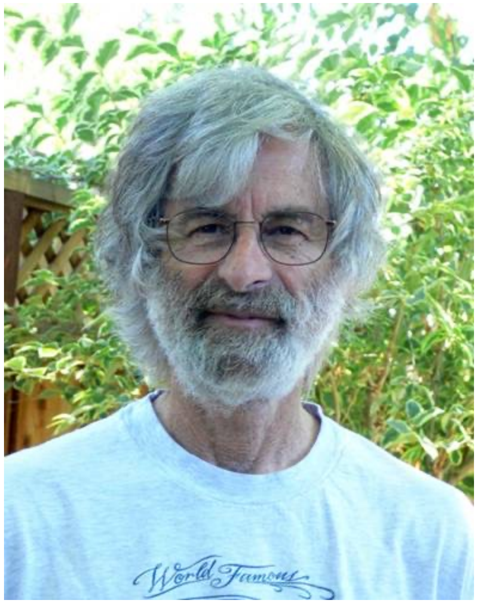
EECS 482: Introduction to Operating Systems

Lecture 24: Distributed Systems

Prof. Ryan Huang

What is a distributed system?

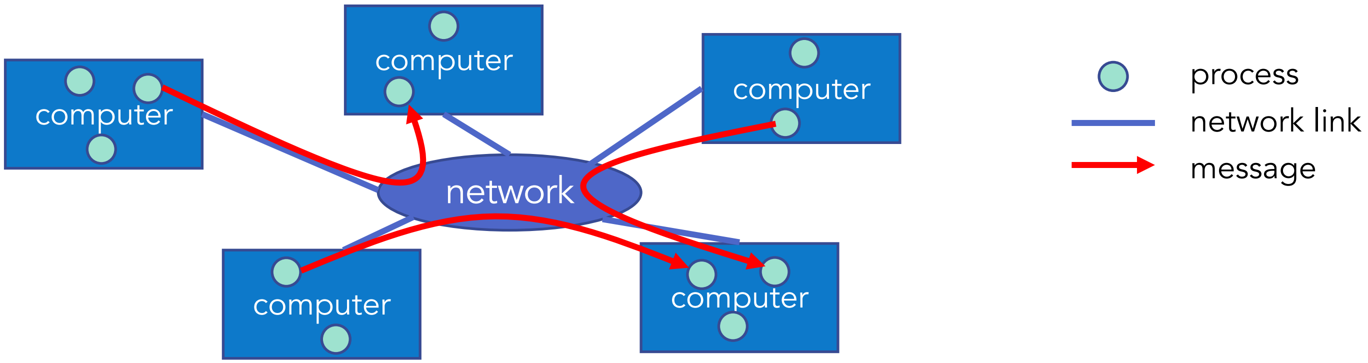
"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."



Leslie Lamport

What is a distributed system?

A collection of processes in a computer network



Distributed systems today

- Proprietary: GFS, Spanner, MapReduce, etc.
- Open-source: Hadoop, ZooKeeper, Cassandra, Kafka, etc.

Why distributed systems?

Expected benefits

- Performance: parallelism across multiple nodes
- Scalability: by adding more nodes
- Reliability: leverage redundancy to provide fault tolerance
- Cost: cheaper and easier to build lots of simple computers
- Collaboration: collaborate through network resources

May not be the reality!

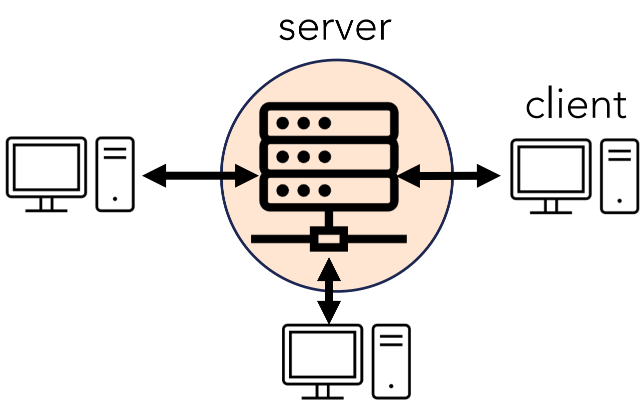
- Worse performance due to comm. costs, stragglers, etc.
- No speed-up after adding more nodes
- A single node crash leads to service unavailability, data loss
- ...

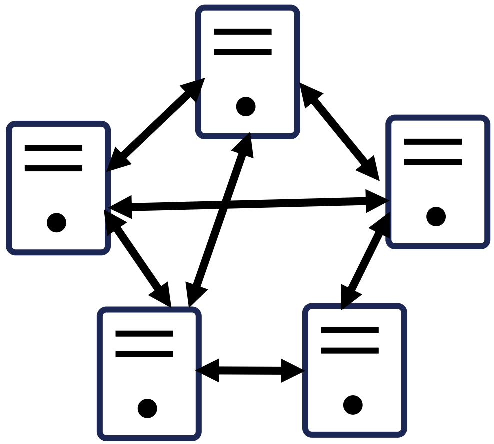
Models of distributed systems?

Degree of integration

- Loosely-coupled: Internet apps, email, web browsing
- Mediumly-coupled: remote execution, remote file systems
- Tightly-coupled: distributed file systems

Client/server vs. cluster/peer-to-peer





Transparency in distributed systems

Transparency is a key requirement/goal:

- The ability for the system to mask its complexity behind a simple interface

Possible transparencies:

- Location: can't tell where resources are located
- Migration: resources may move without the user knowing
- Replication: can't tell how many copies of resource exist
- Concurrency: can't tell how many users there are
- Parallelism: a jobs may be split into smaller pieces
- Fault Tolerance: may hide various things that go wrong

Distributed system is hard!

Coordination

- Must coordinate multiple copies of shared state info
- What would be easy in a centralized system becomes a lot more difficult
  - E.g., agreeing on some value

Scale

- A solution that works for a small-scale system may no longer suffice

Heterogeneity

- Machines with different configurations, architectures, speed, ...

Failures

- Typical first year for a new cluster:
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
  - ~5 racks go wonky (40-80 machines see 50% packet loss)
  - ~thousands of hard drive failures
  - ...

Source: [Building Software Systems at Google and Lessons Learned](), Jeff Dean

Case study: distributed file system

Main abstraction: remote storage looks like local storage

Examples?

Basic implementation: single client, single server (Project 4)

Advanced implementations: multiple clients, multiple servers

- Client-side caching
- Splitting data across multiple servers
- Replicating data across multiple servers

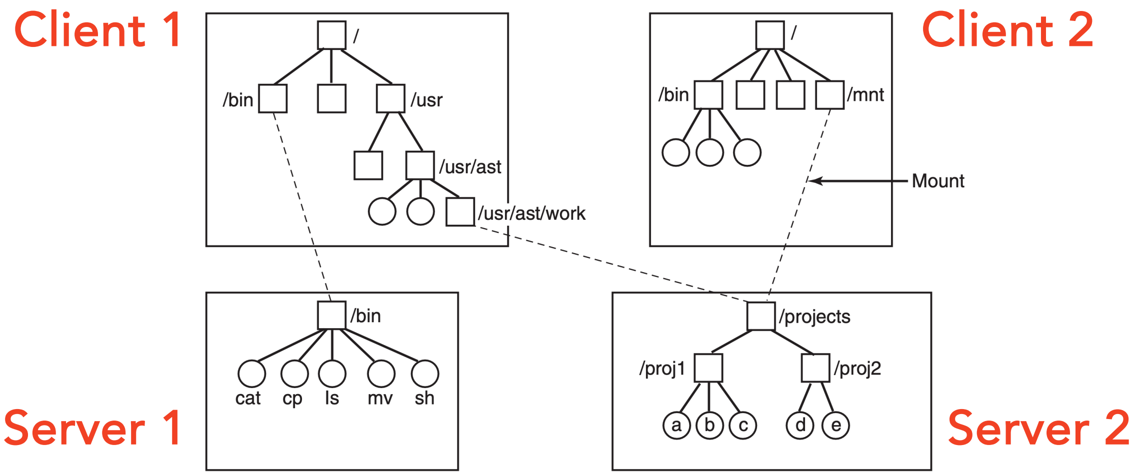
Example: Network File System (NFS)

A widely-used distributed file system

- Developed by Sun Microsystems
- An open *protocol*

Allow a remote directory to be "mounted" onto a local directory using the mount protocol

- Giving access to that remote directory *and all its descendants* as if they were part of the local FS hierarchy



NFS implementation

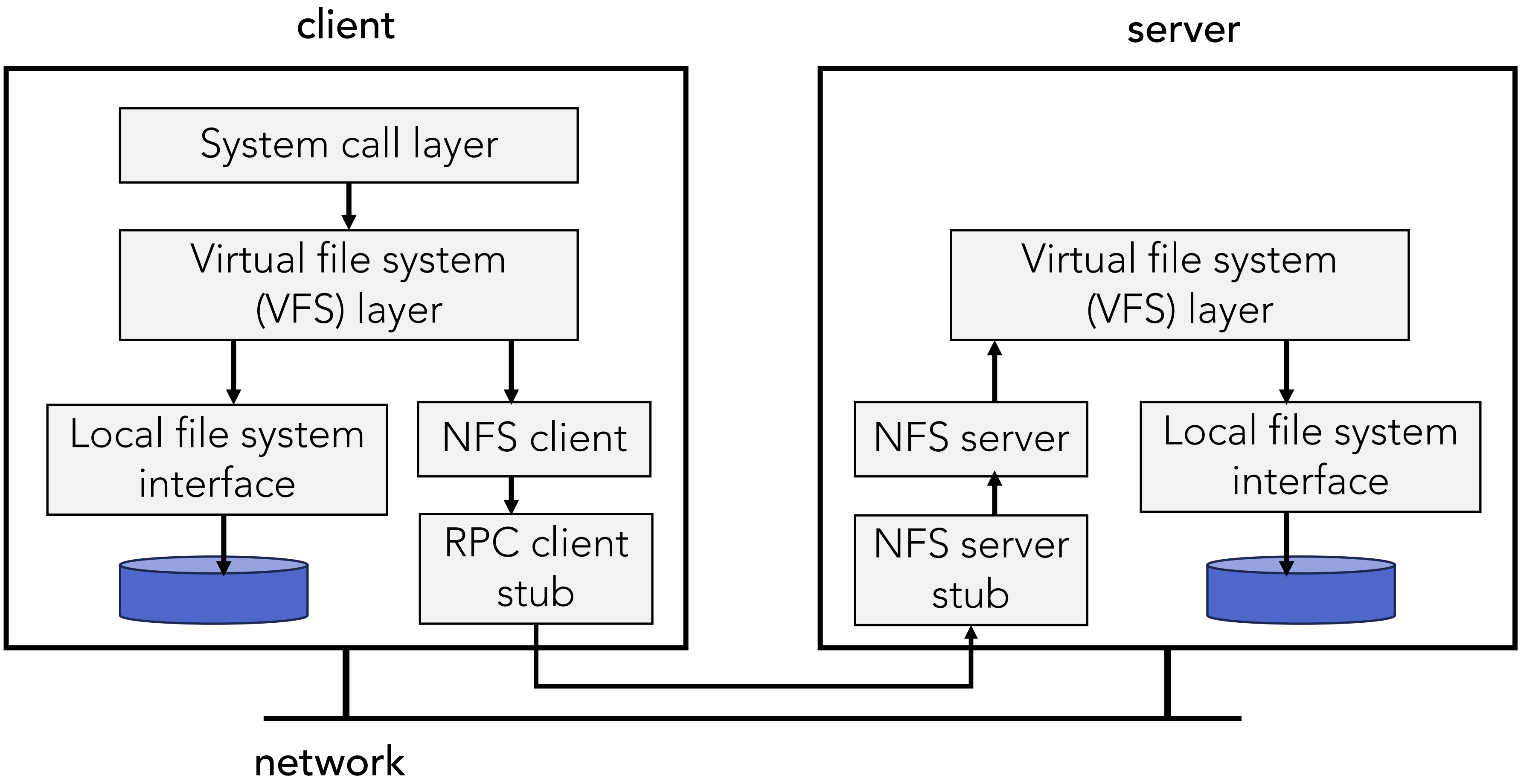
Build on UNIX file-system interface

Introduce a Virtual File System (VFS) layer

- Using v-nodes as file handles
- A v-node describes either a local or remote file
- VFS allows the same system call interface to be used for different types of file systems
- Modern Linux systems now support VFS as an integral part of their file systems, *even if NFS is not used*

Client and server communicate using RPCs

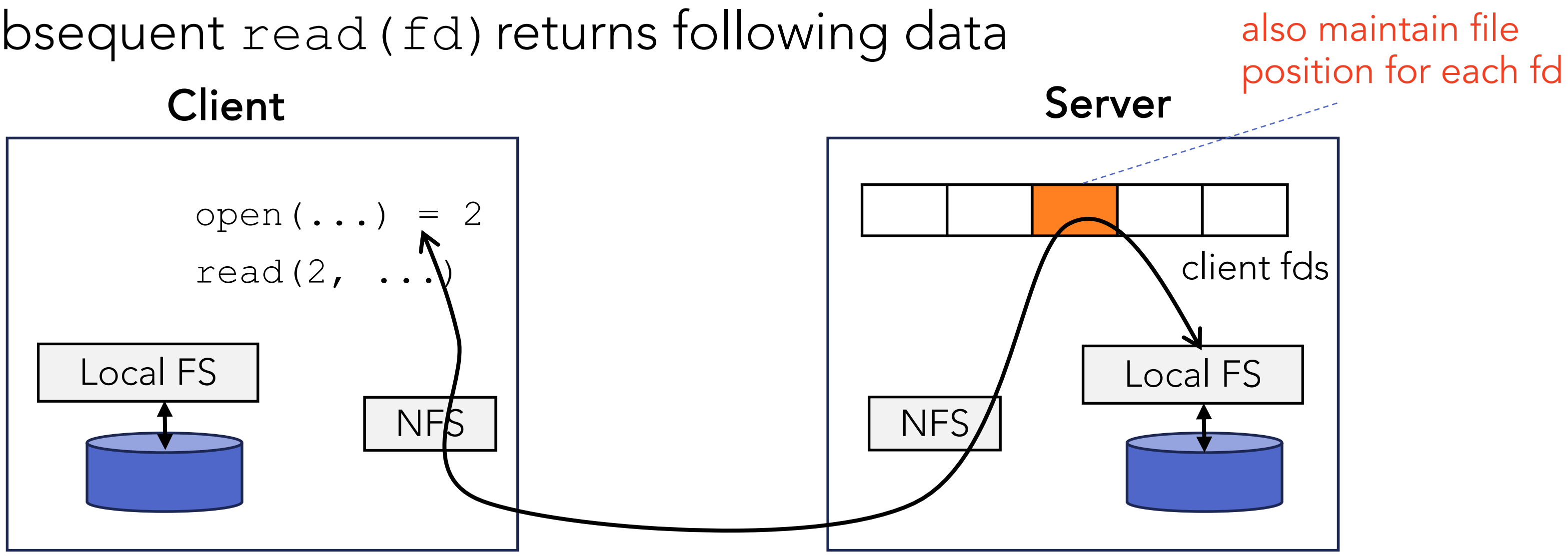
NSF architecture



NFS design - attempt 1

Wrap regular UNIX system calls using RPC

- `open()` on client calls `open()` on server
- `open()` on server returns file descriptors (fd) back to client
- `read(fd)` on client calls `read(fd)` on server
- `read(fd)` on server returns data back to client
- Subsequent `read(fd)` returns following data



Problem?

- What if the server crashes in between?

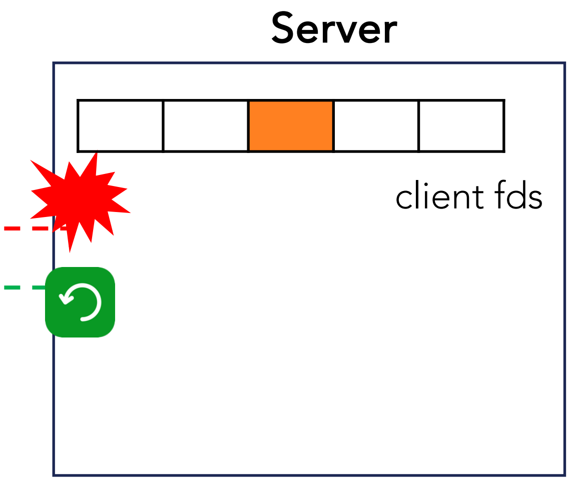
Failure handling in attempt 1

NSF server crashes and reboots in between:

Client

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


read(fd, buf, LEN);
read(fd, buf, LEN);
```



Solutions?

1. Run some crash recovery protocol upon reboot
   - Complex
2. Persist client fd states on server disk
   - Slow
   - What if client crashes? When can fds be garbage collected?

NFS design - attempt 2

Use “stateless” protocol!

- Server maintains no state about clients
  - Server still keeps other state, of course
- Clients include all information in their requests

Need API changes

One possibility:

```
read(char *path, buf, size, offset);
write(char *path, buf, size, offset);
```

- Pros?
  - Server need not remember anything from clients
- Cons?
  - Many path lookups

NFS design - attempt 3

Use inode numbers in requests

- Minimizes path lookups

APIs:

```
inumber = open(char *path);
read(inumber, buf, size, offset);
write(inumber, buf, size, offset);
```

How is this different from attempt 1?

- Use inode number vs. a file descriptor
- Client specifies offset in requests vs. server records file position

Problem?

- If file is deleted, the inumber could be reused
  - inumber not guaranteed to be unique over time

NFSv2 design

Use a compound file handle (FH)

- File handle = <volume ID, inode #, generation #>
- Opaque to client (client should not interpret internals)

Client side tracks relevant state (e.g., FD to FH)

Client

1 `fd = open("foo");`

Send LOOKUP (rootdir FH, "foo")

Receive LOOKUP reply, Allocate FD,
Store foo's FH, store current file position,
Return FD to application

2 `read(fd, buf, LEN);`

Find corresponding FH and file position
Send READ (FH, offset=0, count=LEN)

Receive READ reply
Update file position, Return data

Server

Receive LOOKUP request
Look for "foo" in root dir
Return foo's FH + attributes

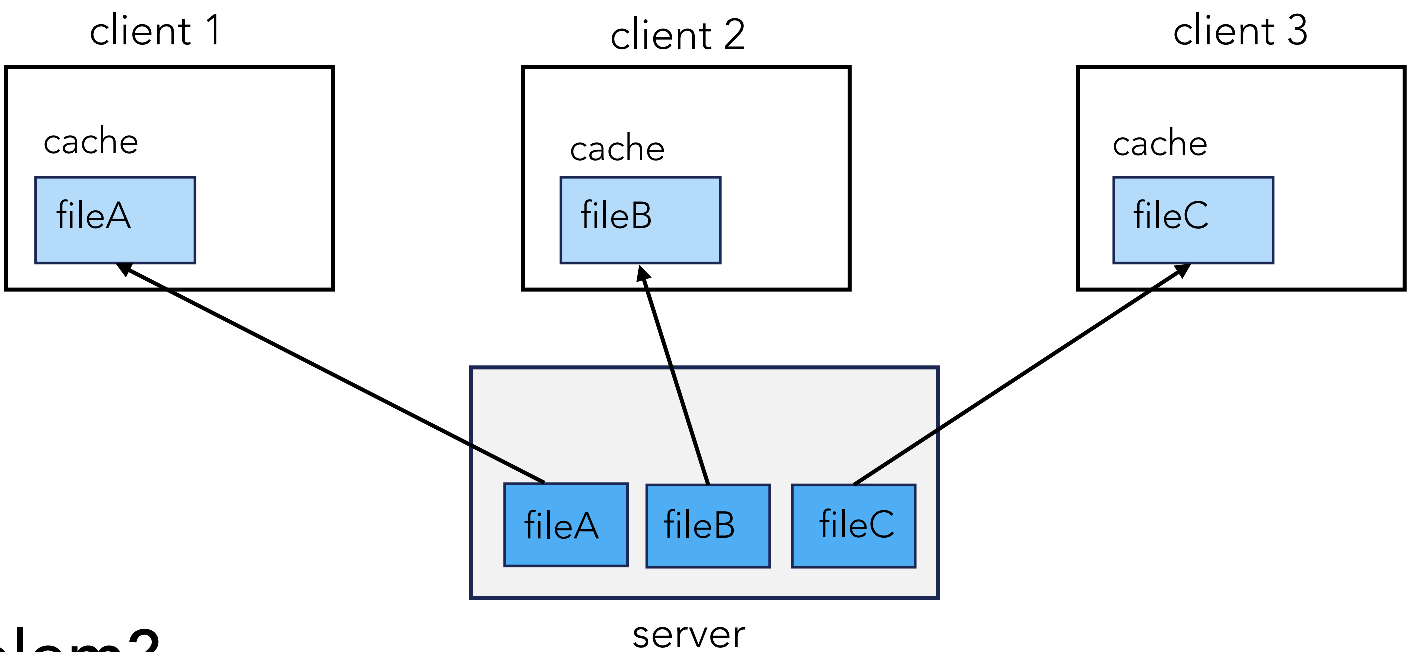
Receive READ request
Use FH to get block info
Read data from disk or cache
Return data to client

Client-side caching

Sending all read/write requests to server is slow…

- With more clients, what will be the bottleneck?

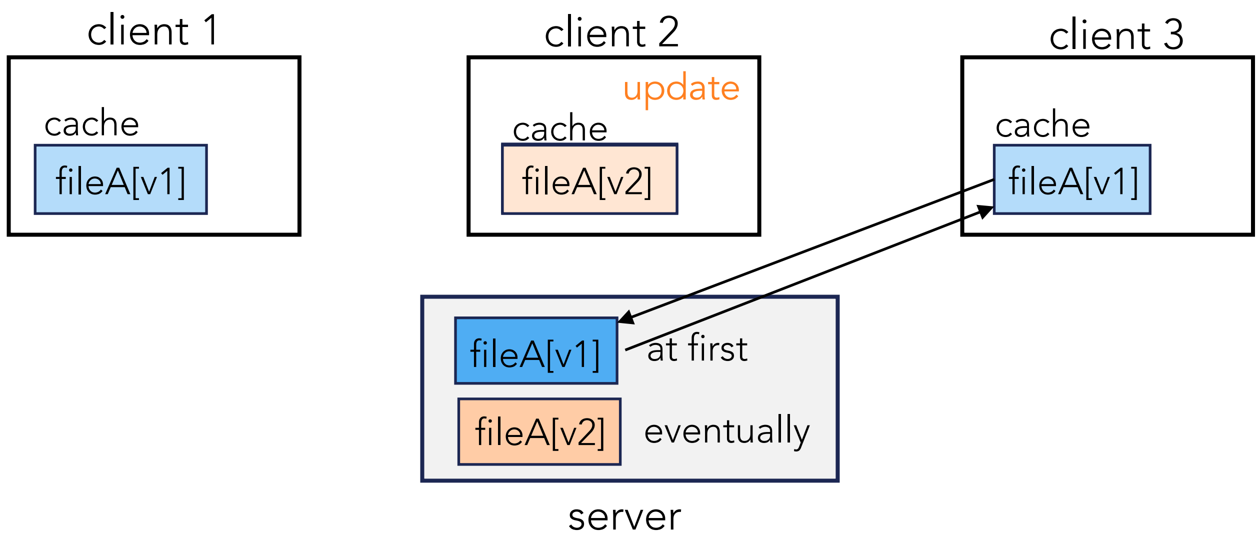
Solution: add cache to clients



Problem?

- Cache consistency

Cache consistency



Problem 1: update visibility

- When does a client's update become visible to other clients?

Problem 2: stale cache

- What happens to old copies of data in some client?

Cache consistency in NFS

Weak consistency

- Client polls server periodically to check for changes
- When file is changed on one client, server is notified, but other clients use old version of file until timeout

What if multiple clients write to same file?

- In NFS, can get either version (or parts of both)
  - File data is cached at block granularity
  - The server file can contain blocks from different clients
- Completely arbitrary

Cache consistency in AFS

Andrew File System (AFS): developed at CMU in 1980s

Use whole-file caching

- Entire file is fetched from server upon `open()` and stored in local disk

Callbacks: Server records who has copy of file

- On changes, server immediately tells all with old copy
  - No polling bandwidth needed

Cache consistency in AFS (cont'd)

Write through on close

- Changes not propagated to server until close()
- Updates visible to other clients only after the file is closed
  - No partial writes (all or nothing)
  - But updates are visible immediately to other programs on local machine that have file open
- Program that has file open sees old version until reopen

Crash recovery more complicated than NFS

- Server crashes: lose callback states; reconstruct by asking the clients
- Client crashes: need to check with server about whether cache is still valid

Multiple file servers (w/o replication)

Assign different clients to different servers?

Assign different files to different servers?

- How to know which server to contact?

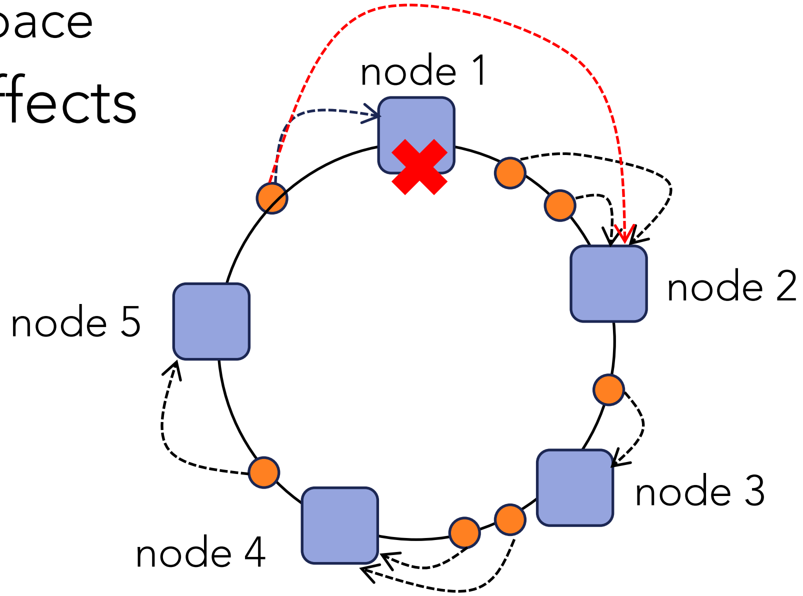
Hash-based partitioning

Assign file *f* to server [*hash(f) % n*]

What happens if you add or remove a server?

Solution: consistent hashing

- Map nodes and keys (files) to a virtual ring
  - Each node owns a range of the keyspace
- Adding/removing a server only affects a portion of the keys
  - Vs. having to remap all keys



Replicated file servers

Using a single file server is problematic

- In terms of both performance and reliability

Using replication to keep replicas of a file (at some block granularity) in multiple servers

Example: Google file system

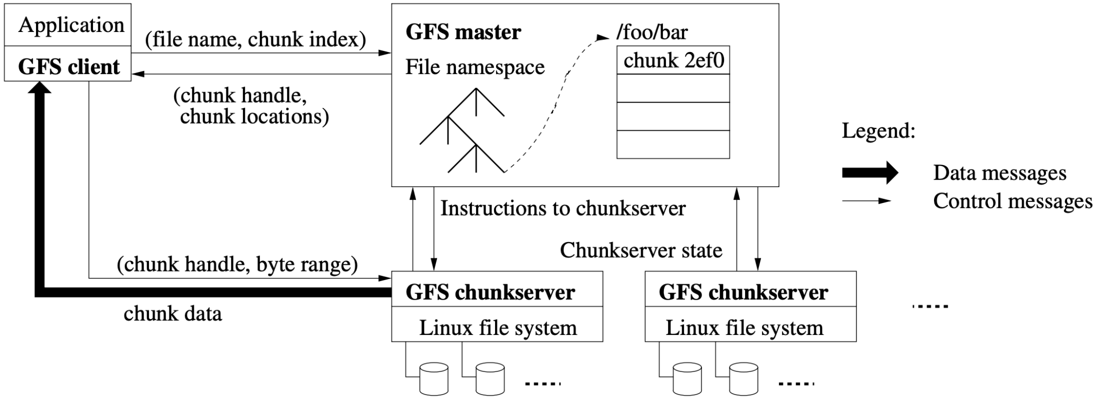
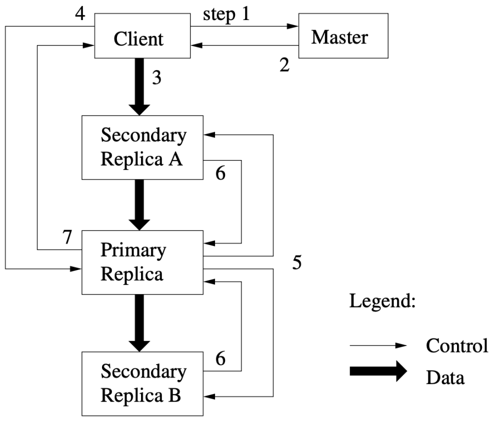


Figure 1: GFS Architecture



Source: [The Google File System]