CS 425 / ECE 428
Distributed Systems
Fall 2026

Aishwarya Ganesan & Ram Kesavan

Lecture 24: Remote & Distributed File Systems

Ack: Indranil Gupta

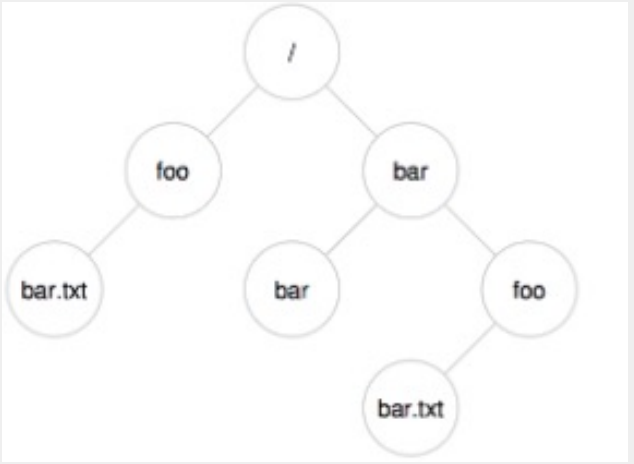
All slides © IG

Announcements

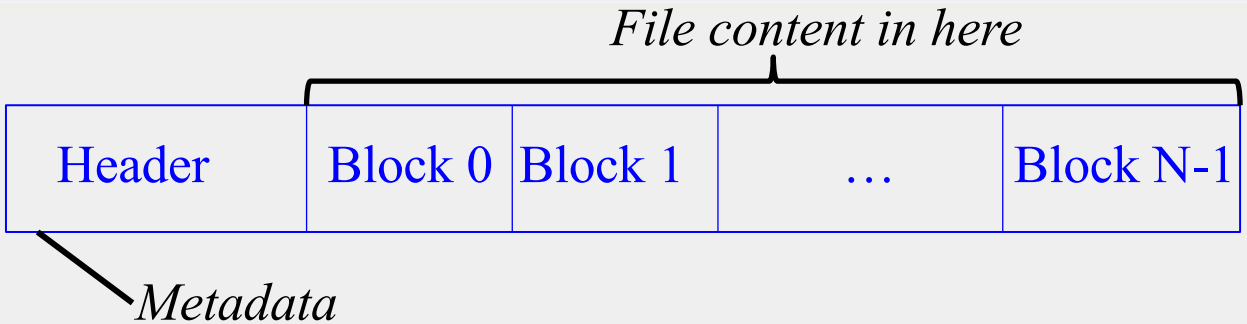
- HW4 released, due 12/3, Thursday!
- MP4 will be released today
  - Due 12/6, Sunday 11:59pm
  - Demos 12/7, Monday
- HW4 & MP4 due after Thanksgiving/Fall Break, so please start soon!
  - They're **both due the same week**!
- HW3 grades released (regrade window until Nov 16, 11:59pm)

File System

- File: sequence of persisted bytes
- File system: collection of files arranged into a directory (folder) namespace
  - /foo/bar.txt
- Higher level of abstraction
  - Users and processes can avoid dealing with blocks on storage and memory blocks
  - Instead, use POSIX file system API



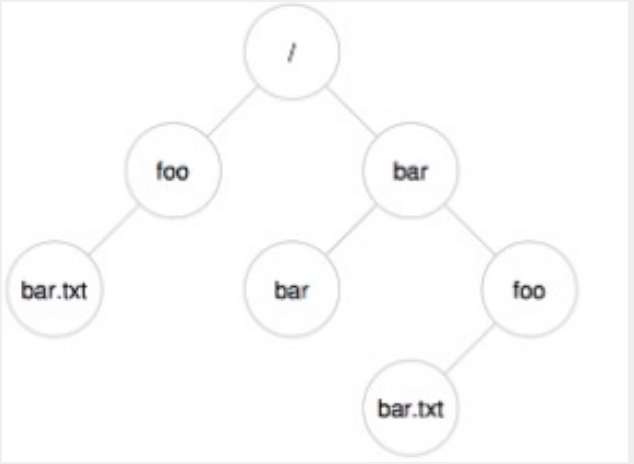
File Contents



- Timestamps: creation, most-recent read, most-recent-write
- File type: .c, .o, .exe, .java
- Ownership
- Access Control List (ACLs): more fine-grained permission info
  - Who can access the file and in what mode
- Reference count: Number of directories containing this file
  - May be > 1 (hard linking of files)
  - File is deleted when reference count = 0
- Locations of the N blocks of content

What about Directories?

- They're just special files!
- Similar metadata includes timestamps, ownership, etc.
- Their "data" contains:
  - Information about files in the directory, e.g. file name
  - Pointers (on disk) to the actual files' header
  - Information about children directories, e.g. dir name
- Files are accessed via system calls



System Calls: Opening and Closing Files

- Uses notion of *file descriptors (fd)*
  - Handle for the caller process to access a file
- Each process: has to open the file before reading/writing it
  - OS creates an internal data structure for a file descriptor, returns it
- *fd* = open(*name, mode*)
  - mode = access mode, e.g., r, w, x
- *fd* = creat(*name, mode*)
  - Create a new file, return the file descriptor
- close(*fd*)

Unix File System: Reading and Writing

- *error* = read(*fd, buffer, nbytes*)
  - File descriptor maintains a *pointer* to an offset within file
  - read( ) reads *nbytes* from that offset into *buffer*, and automatically advances *pointer* by *nbytes*
  - error returns the #bytes read, or 0 if EOF, or -1 if error (errno is set)
- *error* = write(*fd, buffer, nbytes*)
  - Writes from *nbytes* from *buffer* to the file starting at the offset, and automatically advances pointer by *nbytes*
- *pos* = lseek(*fd, offset, whence*)
  - Moves the pointer to *offset* within file
  - *whence* says whether *offset* absolute or relative (relative to current pointer)

Remote & Distributed File Systems

Earlier, FTP protocol used to access files over the network

Mid ’80s (workstations and PCs) led to…

Accessing file system from a remote server

- Clients use RPCs to access files/directories from a server machine
- Protocols: NFS, SMB, AFS, etc.

Mid ’90s multi-server file system architecture led to…

Distributed File Systems

- Multiple server collaborate to export a single file system
- GFS/Colossus, Azure Files, HDFS, Ceph, GlusterFS, GPFS, Lustre, etc

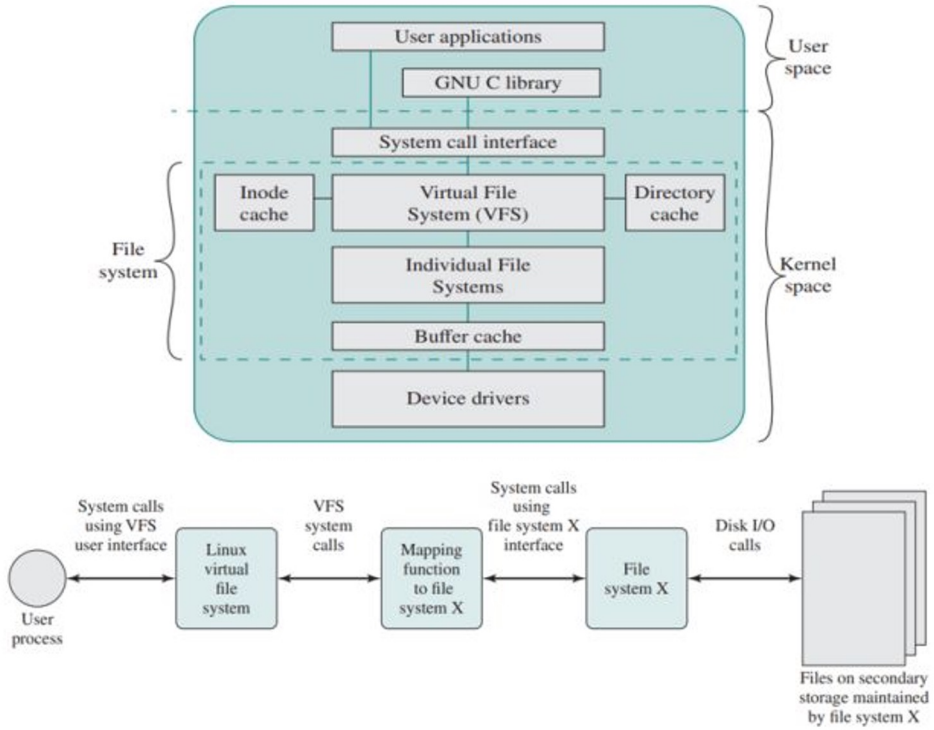
Conflation: *Remote file access is often called a distributed file system*

Remote File Systems

Desirable Properties from a Remote FS

- Transparency: clients access remote files as if they are local
  - Same API as local files, i.e., client code doesn't change
  - Make location, replication, etc. invisible to client
- Support concurrent clients
  - Multiple client processes reading/writing the file concurrently

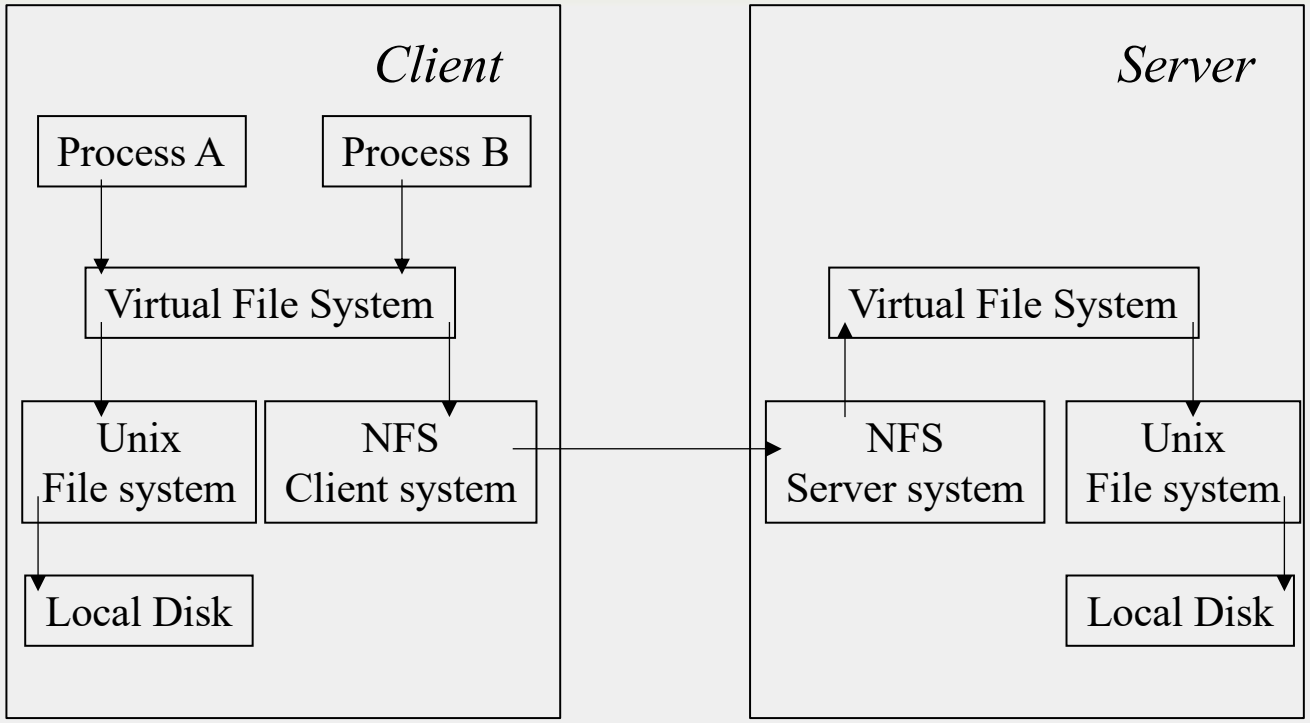
Virtual File System Layer



VFS

- Used by many OSes
- Abstraction layer to different FS types
- VFS translates each system call to the underlying FS's API
- Maintains info from all FSs in local caches: files, dirs, data.
- Allows a single machine to "mount" many local file systems

NFS Architecture



Remote File Access Protocols

- NFS: Network File System; v1 thru' v4.2
  - '84: Originated at Sun Microsystems for Solaris workstations
  - Published it and made it an open standard ➔ most popular
- SMB: Server Message Block; v1 thru' v3.1.1
  - Originated at IBM '83 for DOS PCs as an open standard
  - Microsoft rebranded it as CIFS (Common Internet File Protocol)
  - Implementations were kept proprietary; Open-source Samba project
  - '07 European anti-trust ruling: CIFS folded back into the open SMB standard
- AFS: Andrew File System
  - Originated at CMU (partnered with IBM) in '83
  - Pioneered: aggressive caching, global namespace, Kerberos security
  - Used only in some academic/research institutions

NFS: Mount & Export

- NFS Client system
  - Typically, runs as a kernel module in the client's OS
  - Enables client to *mount* a remote file system *exported* by NFS server
  - Local POSIX file system calls → NFS client sends RPCs to NFS server
- NFS Server system
  - *Exports* a local file system over the network; multiple clients can *mount* it
  - Executes received RPCs by reading/modifying this exported file system

Mount /server/bar from server into /usr/local/foo

⇒ Now, /usr/local/foo/baz on client refers to /server/bar/baz on server

Mount **does NOT clone or copy files**, just points to the remote file now

Polls



Virtual File System Module

- Allows processes to access files via file descriptors
  - Just like local Unix files. So, local and remote files are indistinguishable (i.e., gives transparency)
  - For a given file access, decides whether to route to local file system or to NFS client system

- If processes use file descriptors to access files, does that require the NFS server to be stateful?

Virtual File System Module (contd)

- Maps the *fd* (file descriptor) to an opaque NFS data structure called the file handle
  - File handle: an opaque blob of bytes that isn't parsed by client
  - Returned by NFS server as response to the RPC for an open( ) call on the client
- NFS server guarantees the mapping of file handle to its local file never changes
  - Allows NFS server to remain stateless

Concurrent Access to a file/dir



More examples for at most once and at least once?

- Changes to a file from one client should not interfere with another client's access to the same file
- Techniques
  - Best effort: client-side & server-side caching (can be unpredictable)
  - Session-based: changes invisible to others until file is closed (last write wins)
  - Immutable: file can never be modified; new version created
  - Lease/delegations and byte-range locking
- At-most-once vs at-least-once operation: choose carefully
  - E.g., append operations cannot be repeated
  - *Idempotent* operations can be at-least-once, e.g., read at absolute position in file

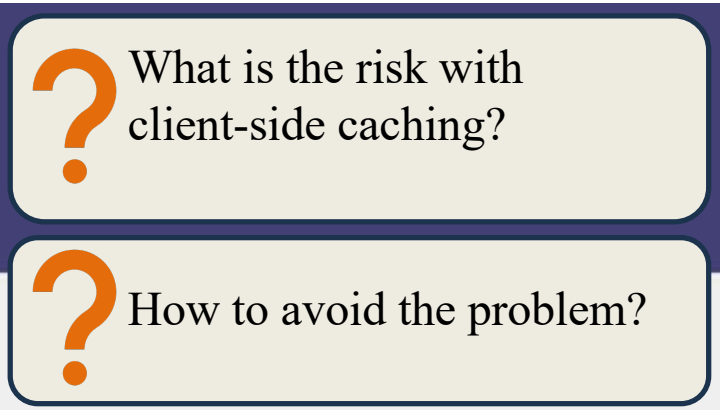
Remote Access: Other Aspects

- Authentication
  - Verify that a given user is who they claim to be
- Authorization
  - After a user is authenticated, verify they have permissions to access the file
  - Two flavors
    - **Access Control Lists (ACLs)**: per file, list of allowed users and access allowed to each
    - **Capability Lists:** per user, list of files allowed to access and type of access allowed
- Stateless vs Stateful
- Performance: caching

Server-side Caching

- Server caching is one of the big reasons NFS is so fast with reads
  - Caching in memory of recently-accessed blocks (of files and directories)
  - Most programs tend to have *locality of access even across clients*
    - Blocks accessed recently will be accessed soon in the future
- Writes: two flavors
  - Delayed write: write in memory, flush to disk every 30s (e.g., via Unix sync operation)
    - Fast but not consistent
  - Write-through: Write to disk immediately before ack'ing client
    - Consistent but may be slow

Client-side Caching



- Client also caches recently-accessed blocks
- Each block in cache is tagged with
  - *Tc*: the time when the cache entry was last validated.
  - *Tm*: the time when the block was last modified at the server.
  - A cache entry at time *T* is valid if

    (T - Tc < t) or (Tm_{client} = Tm_{server}) // has file been modified since I last cached it?
  - *t is the freshness interval*
    - Compromise between consistency and efficiency
    - Sun Solaris: *t* is set adaptively between 3-30 s for files, 30-60 s for directories
- On a write: client buffers the "dirty" blocks locally, sends RPC later
  - When: close( ), memory pressure, fsync( ), timeout
  - Aka write-back caching

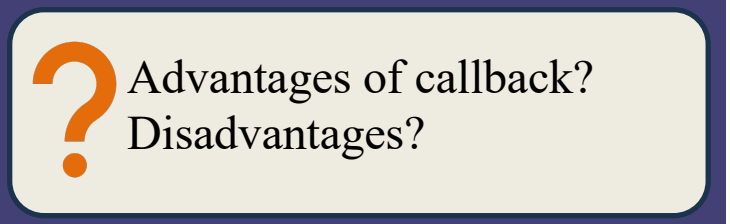
Contrasting Design Decisions in AFS

- Two unusual design principles:
  - Whole file serving
    - Not just specific blocks
  - Whole file caching
    - Permanent cache, survives reboots
- Based on (validated) assumptions that
  - Most file accesses are by a single user
  - Most files are small
  - A “large” client cache is possible (e.g., in RAM & local disk)
  - File reads are much more often that file writes, and typically sequential

AFS: Unified Namespace

- One top level directory "/" mounted by all AFS clients
  - Various administrative domains manage their own servers
  - But their file systems are stitched under the same "/" using a VLDB (volume location database)
  - E.g., */afs/cs.cmu.edu*, */afs/cern.ch*, etc.
  - Client can navigate to any file system from "/"
- NFS uses centralized directory services (LDAP, NIS, AD)
  - Fixed the annoyance of maintaining each client's */etc/fstab*
  - Automounter (*autofs*)
    - A client automatically mounts file systems on-demand using the (same) local directory path
    - If administrator moves Alice's homedir to a different server, they update only the LDAP map, and the client's access to */home/alice* remains correct

AFS Details



- Reads and writes are optimistic
  - Done on local copy of file at client
  - When file closed, writes propagated to server
- When a client opens a file, server:
  - Sends it entire file
  - Gives client a *callback promise*
- Callback promise
  - Promise that if another client modifies and closes the file, a callback will be sent from server to client
  - Callback state at client only binary: valid or canceled

Stateful Servers: AFS, NFSv4, & SMB

- SMB was always stateful (and connection-based)
- NFSv2 & v3 designed for UDP, and so different
- NFSv4 onwards: server stateful
  - Maintains info about clients, lock state, etc.
  - Hands out file delegations to client
  - Client can acquire read/write locks on a byte-range of the file
  - Callbacks from server to reclaim them
  - pNFS: parallel NFS

Distributed File Systems

- Many servers collaborate to export a single file system
- Concurrent client access
- Transparency: a client is unaware about location, failure, replication

Why Distributed File Systems?

- Limited by how much a single server can "scale-up"
  - Total storage attachable to that server
  - CPU + memory needed for server to execute remote access (NFS, SMB, etc.) protocol RPCs → performance
- Also, design for failures
  - Availability
  - Can keep adding cheap/commodity servers to "scale-out" storage and performance
  - Ideally, IOPS load & storage is evenly distributed across the servers
  - Ideal for pNFS style high-throughput access of files

Many Popular Distributed File Systems

Name	Year	Open Source?	Company	POSIX?
GPFS	1998	No	IBM	Yes
GFS/Colossus	2003	No	Google	No
HDFS	2006	Yes	Yahoo	No
CephFS & GlusterFS	2006	Yes	Red Hat	Yes
Pangu	2009	No	Alibaba	No
Tectonic	2021	No	Meta	No

GFS Architecture

GFS: designed for reading/writing large files

No actual directory structure in GFS

- Even though filenames can encode a hierarchy: */a/b/c/foo*
- File is just a sequence of chunks (64MiB), each chunk gets a unique ID

Single Metadata server & many chunk servers; can keep adding chunk servers

Metadata server contains

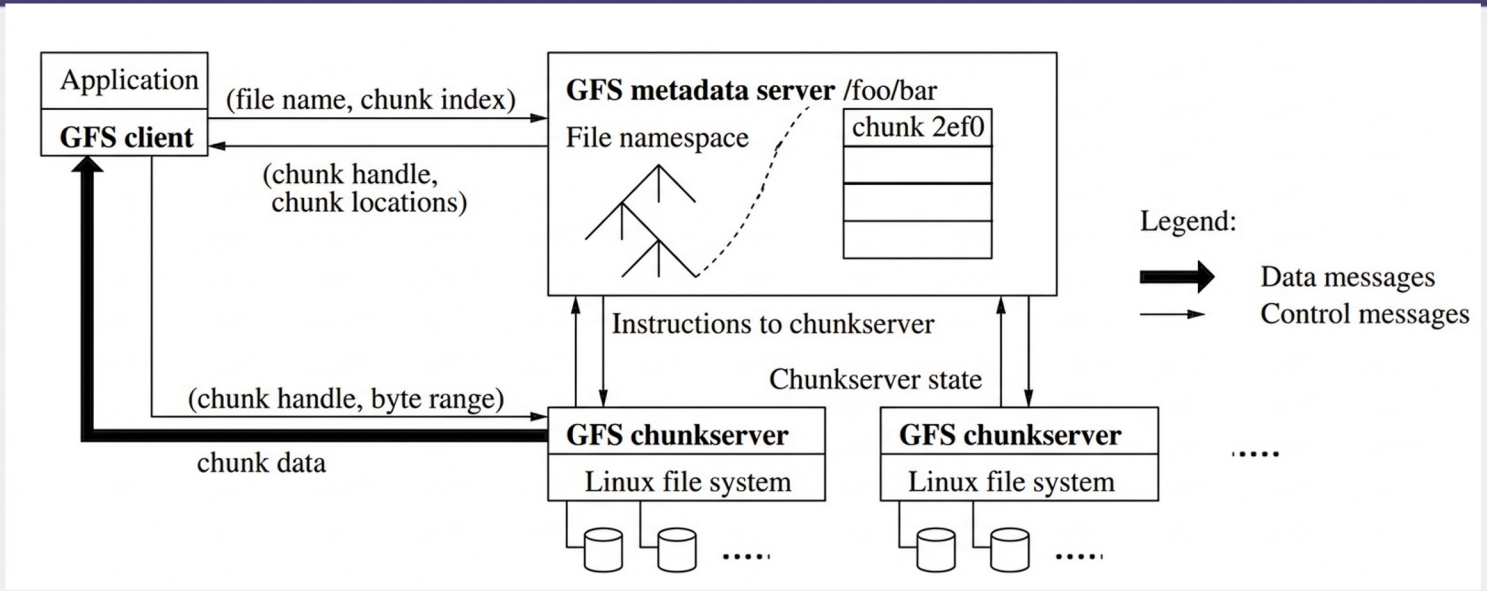
- Flat map of <filename + offset> → <chunkID>
- ChunkID → chunk server replicas

Chunk server: stores chunks as files in its local file system (e.g., Ext4)

- 3 replicas of each chunk; each stored in a different chunk server

Heartbeats between metadata & chunk servers

GFS



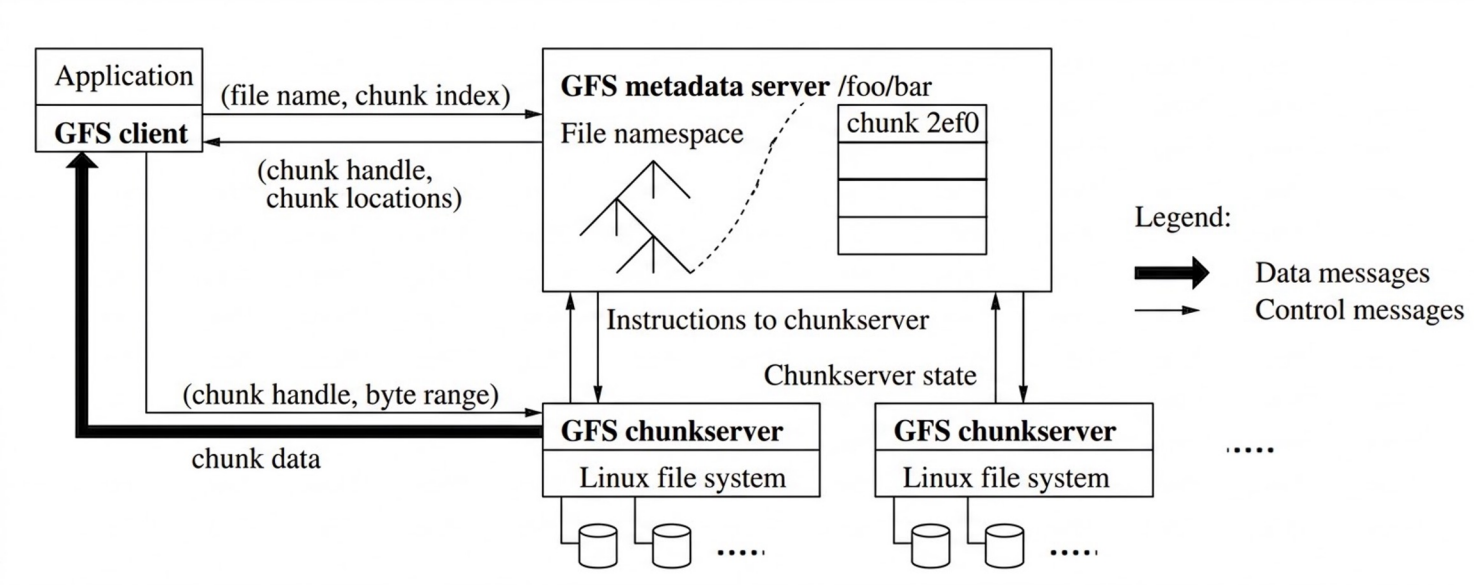
Application is compiled with GFS client; client hides all complexity
Single metadata server + many data chunkservers
Chunkserver uses Linux file system to store the chunks

GFS: A Few Important Details

- GFS is not POSIX compliant
  - True for most scale-out distributed file systems
  - No directories
  - Files are ***append-only***
    - A file is written only once, sequentially
    - When updated, a new version is created; file versioning is supported
- Greatly simplifies GFS (and scale-out distributed file system) design
  - Easy to provide *one-copy update semantics**
- Erasure encoding of chunks
  - 3 (or more) replicas becomes expensive (3x storage)

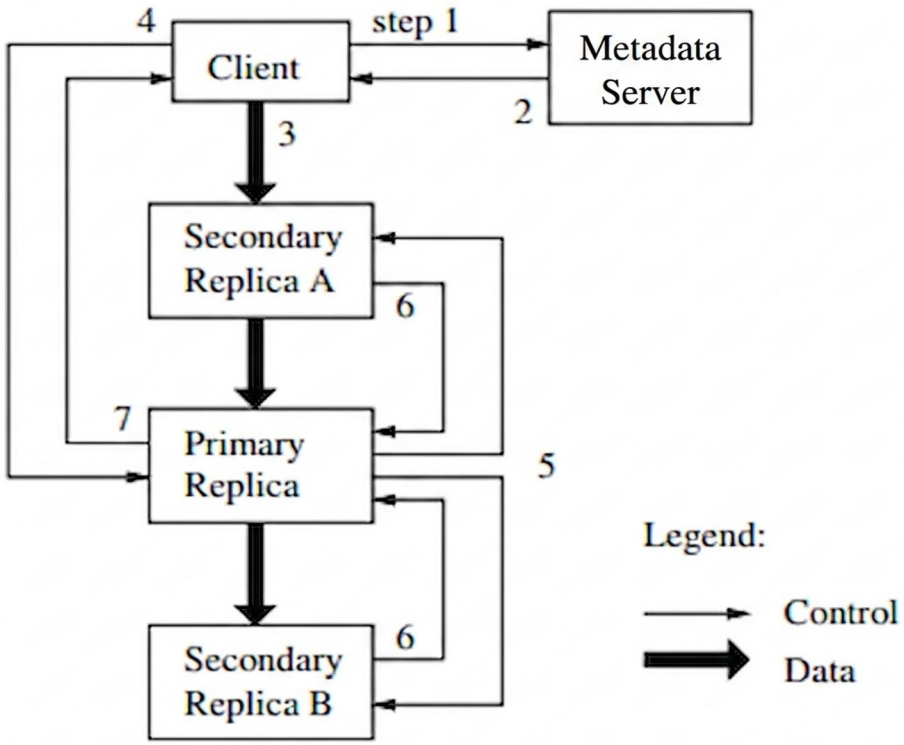
**when file is replicated, its contents, as visible to clients, are no different from when the file has exactly 1 replica*

GFS: Read Path



Client→MS: server filename + offset
MS→client: chunk-handle + replica-locations
Client caches metadata information
    Client picks a chunkserver (typically, closest) and directly reads from it
Future reads: client uses cached metadata, directly communicates with chunk server

GFS: Write Path



1. Give me replicas for chunks
2. Primary + 2ndary
   - Client caches metadata
3. Chunks to replicas
   - Daisy chained across the replicas
   - Wait for all replicas to ack
4. Formal write request
5. Assigns serial order
6. Replicas apply & ack
7. Primary acks

After GFS

HDFS has similar limitations

- Single metadata server (performance bottleneck)
- Eliminated SPOF (single point of failure) metadata server using ZooKeeper

Google now uses Colossus, a successor to GFS successor

Tectonic & Colossus

- Metadata service is also distributed and “scales out”
- Client writes chunks in parallel writes to all replicas instead of daisy-chaining

Reed-Solomon erasure encoding (1.25x--1.5x instead of 3x in GFS)

- Based on Galois Field math
- RS(n, k): k data shards + (n-k) parity shards
- Tolerates and recovers from n-k failed shards

Summary

- Local file systems
- Accesing remote file systems over the network
- How VFS hides the complexity
- NFS, AFS, SMB
  - Concurrent Access
  - Performance: caching
- Distributed file systems
  - Append-only files
  - GFS