Google Cluster Computing Faculty Training Workshop

Module VI: Distributed Filesystems

Outline

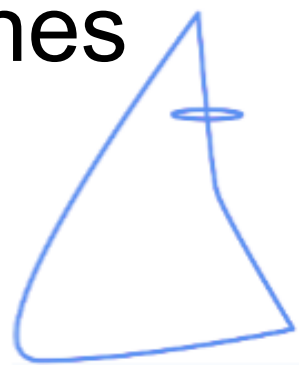
- Filesystems overview
- NFS & AFS (Andrew File System)
- GFS

File Systems Overview

- System that permanently stores data
- Usually layered on top of a lower-level physical storage medium
- Divided into logical units called “files”
  - Addressable by a *filename* (“foo.txt”)
  - Usually supports hierarchical nesting (directories)

File Paths

- A file *path* joins file & directory names into a **relative** or **absolute** address to identify a file
  - Absolute: /home/aaron/foo.txt
  - Relative: docs/someFile.doc
- The shortest absolute path to a file is called its **canonical** path
- The set of all canonical paths establishes the **namespace** for the filesystem



What Gets Stored

- User data itself is the bulk of the file system's contents
- Also includes *meta-data* on a drive-wide and per-file basis:

Drive-wide:	Per-file:
Available space	name
Formatting info	owner
character set	modification date
...	physical layout...

High-Level Organization

- Files are organized in a “tree” structure made of nested directories
- One directory acts as the “root”
- “links” (symlinks, shortcuts, etc) provide simple means of providing multiple access paths to one file
- Other file systems can be “mounted” and dropped in as sub-hierarchies (other drives, network shares)

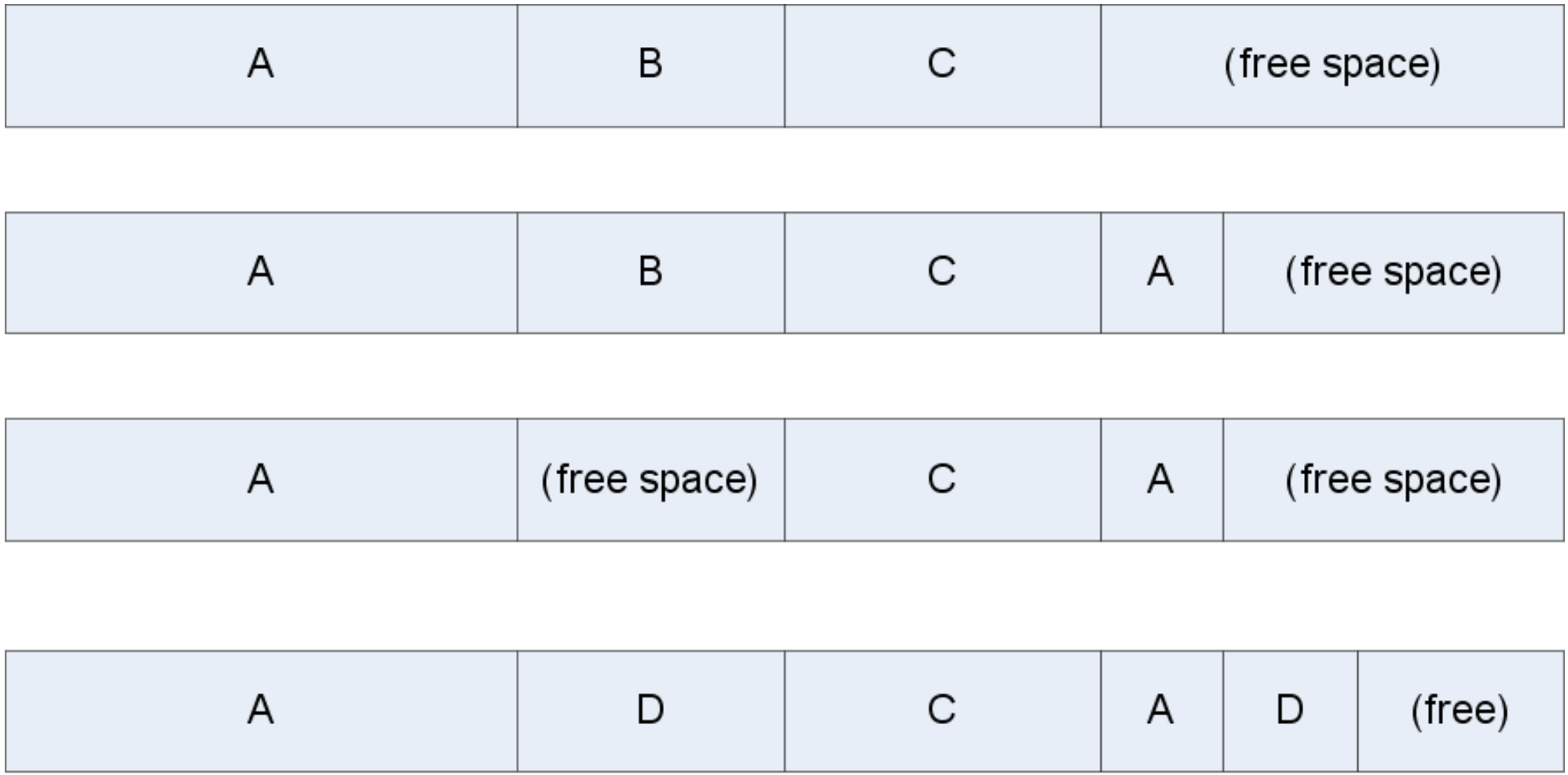
Low-Level Organization (1/2)

- File data and meta-data stored separately
- File descriptors + meta-data stored in *inodes*
  - Large tree or table at designated location on disk
  - Tells how to look up file contents
- Meta-data may be replicated to increase system reliability

Low-Level Organization (2/2)

- "Standard" read-write medium is a hard drive (other media: CDROM, tape, ...)
- Viewed as a sequential array of blocks
- Must address ~1 KB chunk at a time
- Tree structure is "flattened" into blocks
- Overlapping reads/writes/deletes can cause **fragmentation**: files are often not stored with a linear layout
  - inodes store all block ids related to file

Fragmentation



Design Considerations

- Smaller inode size reduces amount of wasted space
- Larger inode size increases speed of sequential reads (may not help random access)
- Should the file system be **faster** or **more reliable**?
- But faster at what: Large files? Small files? Lots of reading? Frequent writers, occasional readers?

Filesystem Security

- File systems in multi-user environments need to secure private data
  - Notion of username is heavily built into FS
  - Different users have different access writes to files

UNIX Permission Bits

- World is divided into three scopes:
  - User – The person who owns (usually created) the file
  - Group – A list of particular users who have "group ownership" of the file
  - Other – Everyone else
- "Read," "write" and "execute" permissions applicable at each level

UNIX Permission Bits: Limits

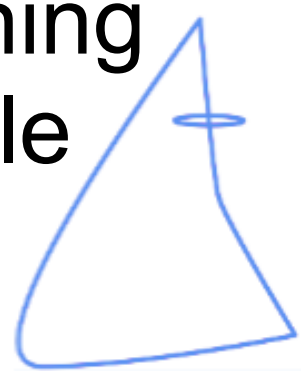
- Only one group can be associated with a file
- No higher-order groups (groups of groups)
- Makes it difficult to express more complicated ownership sets

Access Control Lists

- More general permissions mechanism
- Implemented in Windows
- Richer notion of privileges than r/w/x
  - e.g., SetPrivilege, Delete, Copy…
- Allow for **inheritance** as well as **deny** lists
  - Can be complicated to reason about and lead to security gaps

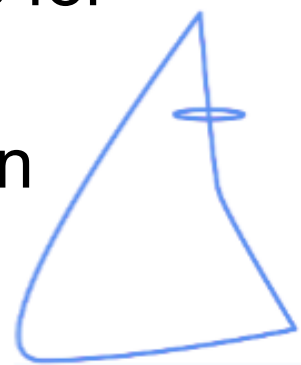
Process Permissions

- Important note: processes running on behalf of user X have permissions associated with X, not process file owner Y
- So if root owns ls, user aaron can not use ls to peek at other users' files
- Exception: special permission "setuid" sets the user-id associated with a running process to the owner of the program file



Disk Encryption

- Data storage medium is another security concern
  - Most file systems store data in the clear, rely on runtime security to deny access
  - Assumes the physical disk won't be stolen
- The disk itself can be encrypted
  - Hopefully by using separate passkeys for each user's files
  - (Challenge: how do you implement read access for group members?)
  - Metadata encryption may be a separate concern



Distributed Filesystems

- Support access to files on remote servers
- Must support concurrency
  - Make varying guarantees about locking, who “wins” with concurrent writes, etc...
  - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations sit in different places on complexity/feature scale

NFS

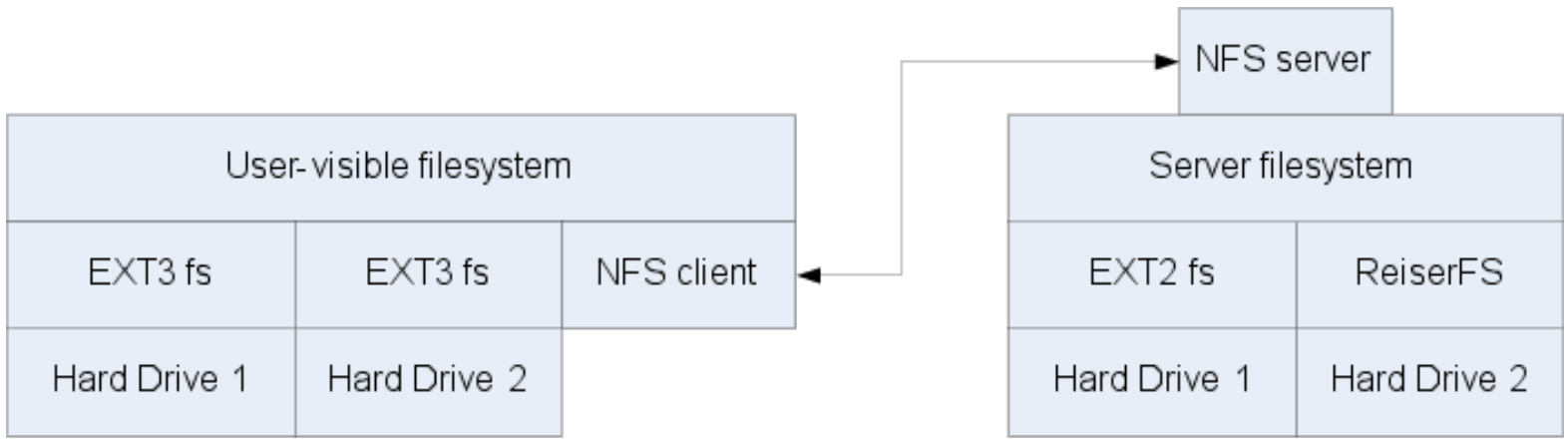
- First developed in 1980s by Sun
- Presented with standard UNIX FS interface
- Network drives are *mounted* into local directory hierarchy
  - Your home directory on attu is NFS-driven
  - Type 'mount' some time at the prompt if curious

NFS Protocol

- Initially completely stateless
  - Operated over UDP; did not use TCP streams
  - File locking, etc, implemented in higher-level protocols
- Modern implementations use TCP/IP & stateful protocols

Server-side Implementation

- NFS defines a *virtual file system*
  - Does not actually manage local disk layout on server
- Server instantiates NFS volume on top of local file system
  - Local hard drives managed by concrete file systems (EXT, ReiserFS, ...)
  - Other networked FS's mounted in by...?



NFS Locking

- NFS v4 supports stateful locking of files
  - Clients inform server of intent to lock
  - Server can notify clients of outstanding lock requests
  - Locking is lease-based: clients must continually renew locks before a timeout
  - Loss of contact with server abandons locks

NFS Client Caching

- NFS Clients are allowed to cache copies of remote files for subsequent accesses
- Supports *close-to-open* cache consistency
  - When client A closes a file, its contents are synchronized with the master, and timestamp is changed
  - When client B opens the file, it checks that local timestamp agrees with server timestamp. If not, it discards local copy.
  - Concurrent reader/writers must use flags to disable caching

NFS: Tradeoffs

- NFS Volume managed by single server
  - Higher load on central server
  - Simplifies coherency protocols
- Full POSIX system means it “drops in” very easily, but isn’t “great” for any specific need

Distributed FS Security

- Security is a concern at several levels throughout DFS stack
  - Authentication
  - Data transfer
  - Privilege escalation
- How are these applied in NFS?

Authentication in NFS

- Initial NFS system trusted client programs
  - User login credentials were passed to OS kernel which forwarded them to NFS server
  - … A malicious client could easily subvert this
- Modern implementations use more sophisticated systems (e.g., Kerberos)

Data Privacy

- Early NFS implementations sent data in "plaintext" over network
  - Modern versions tunnel through SSH
- Double problem with UDP (connectionless) protocol:
  - Observers could watch which files were being opened and then insert "write" requests with fake credentials to corrupt data

Privilege Escalation

- Local filesystem username is used as NFS username
  - Implication: being “root” on local machine gives you root access to entire NFS cluster
- Solution: “root squash” – NFS hard-codes a privilege *de-escalation* from “root” down to “nobody” for all accesses.

AFS (The Andrew File System)

- Developed at Carnegie Mellon
- Strong security, high scalability
  - Supports 50,000+ clients at enterprise level

Security in AFS

- Uses Kerberos authentication
- Supports richer set of access control bits than UNIX
  - Separate “administer”, “delete” bits
  - Allows application-specific bits

Local Caching

- File reads/writes operate on locally cached copy
- Local copy sent back to master when file is closed
- Open local copies are notified of external updates through *callbacks*

Local Caching - Tradeoffs

- Shared database files do not work well on this system
- Does not support *write-through* to shared medium

Replication

- AFS allows read-only copies of filesystem volumes
- Copies are guaranteed to be atomic checkpoints of entire FS at time of read-only copy generation
- Modifying data requires access to the sole r/w volume
  - Changes do not propagate to read-only copies

AFS Conclusions

- Not quite POSIX
  - Stronger security/permissions
  - No file write-through
- High availability through replicas, local caching
- Not appropriate for all file types

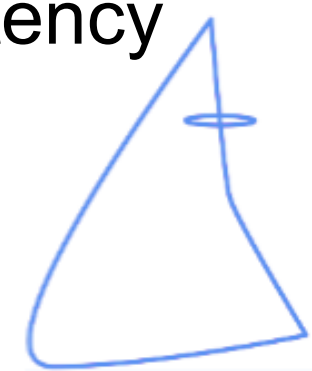
The Google File System

Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers
- Why not use an existing file system?
  - Google's problems are different from anyone else's
    - Different workload and design priorities
  - GFS is designed for Google apps and workloads
  - Google apps are designed for GFS

Assumptions

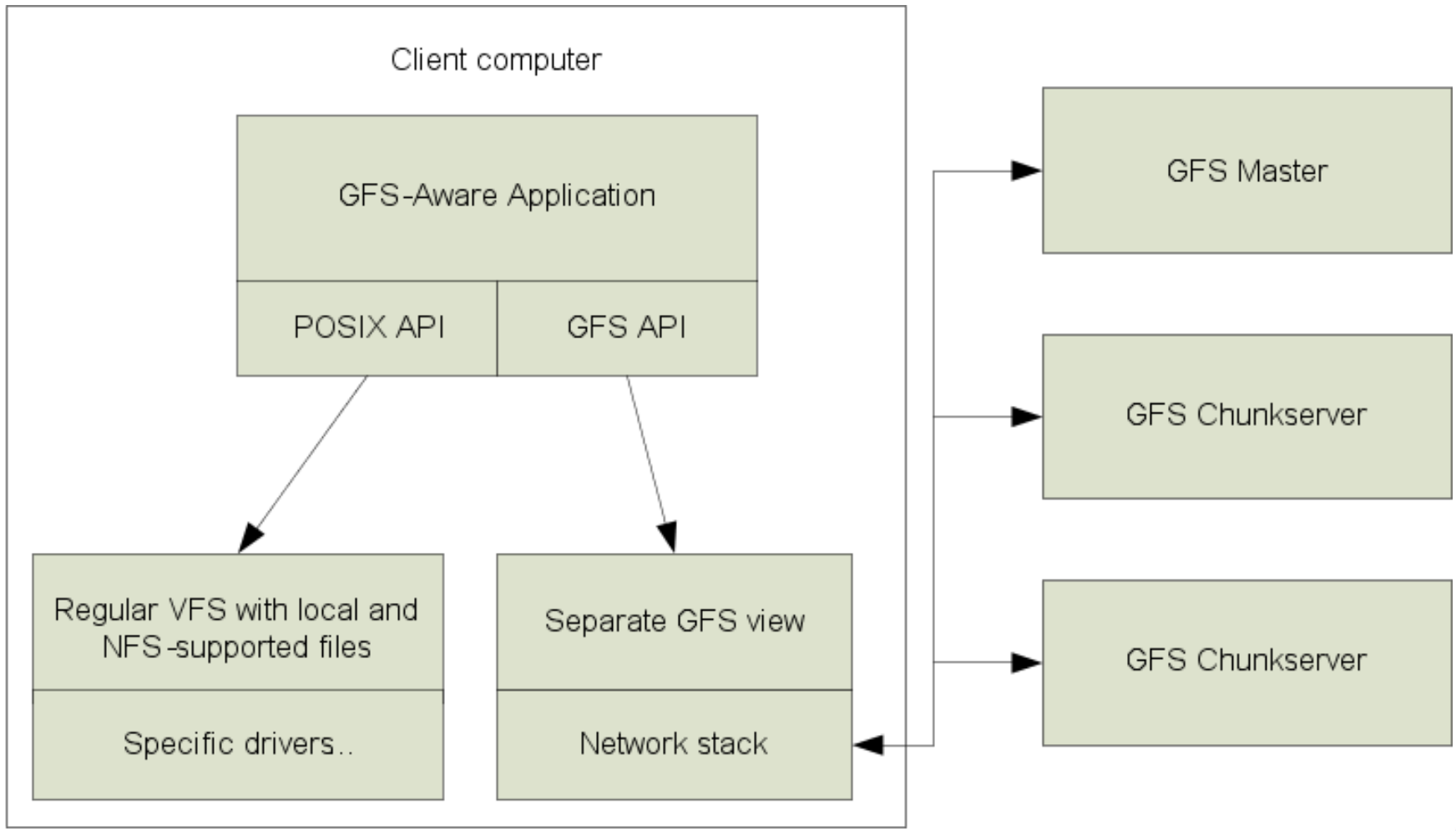
- High component failure rates
  - Inexpensive commodity components fail often
- “Modest” number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency



GFS Design Decisions

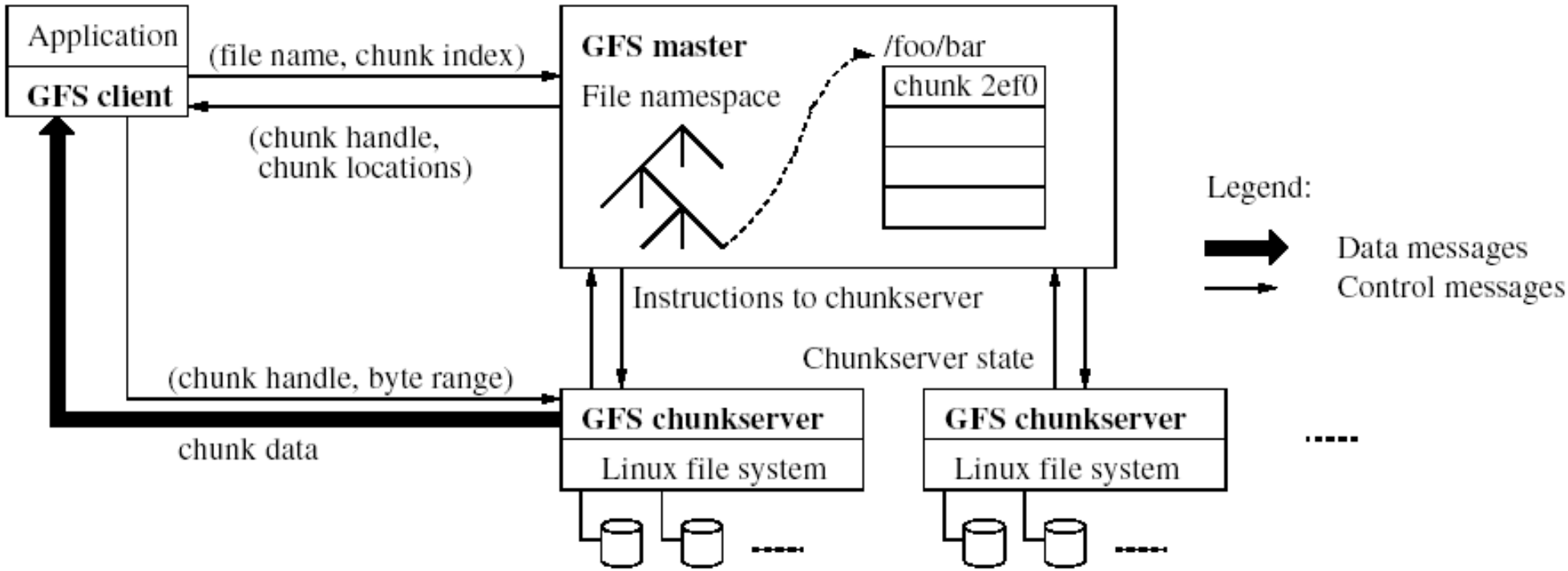
- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on Google apps
  - Add *snapshot* and *record append* operations

GFS Client Block Diagram



GFS Architecture

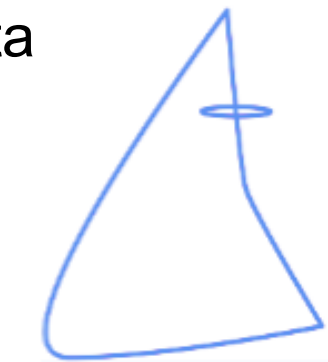
- Single master
- Mutiple chunkservers



…Can anyone see a potential weakness in this design?

Single master

- From distributed systems we know this is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters
  - Minimize master involvement
    - never move data through it, use only for metadata
      - and cache metadata at clients
    - large chunk size
    - master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!



Metadata (1/2)

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
  - Fast
  - Easily accessible

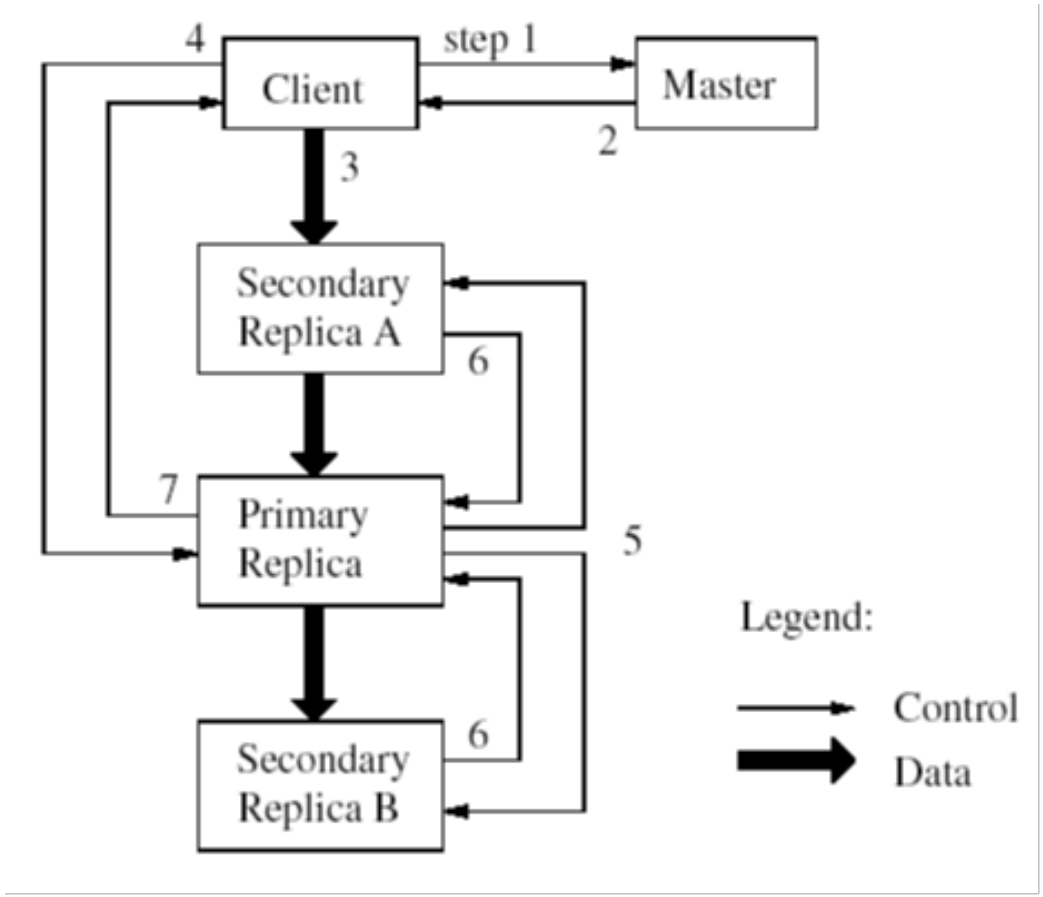
Metadata (2/2)

- Master has an *operation log* for persistent logging of critical metadata updates
  - persistent on local disk
  - replicated
  - checkpoints for faster recovery

Mutations

- Mutation = write or append
  - must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism:
  - master picks one replica as primary; gives it a “lease” for mutations
  - primary defines a serial order of mutations
  - all replicas follow this order
- Data flow decoupled from control flow

Mutations Diagram



Mutation Example

1. Client 1 opens "foo" for modify. Replicas are named A, B, and C. B is declared primary.
2. Client 1 sends data X for chunk to chunk servers
3. Client 2 opens "foo" for modify. Replica B still primary
4. Client 2 sends data Y for chunk to chunk servers
5. Server B declares that X will be applied before Y
6. Other servers signal receipt of data
7. All servers commit X then Y
8. Clients 1 & 2 close connections
9. B's lease on chunk is lost

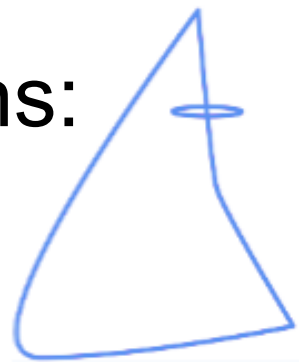
Atomic record append

- Client specifies data
- GFS appends it to the file atomically at least once
  - GFS picks the offset
  - works for concurrent writers
- Used heavily by Google apps
  - e.g., for files that serve as multiple-producer/single-consumer queues

Relaxed consistency model (1/2)

- "Consistent" = all replicas have the same value
- "Defined" = replica reflects the mutation, consistent

- Some properties:
  - concurrent writes leave region consistent, but possibly undefined
  - failed writes leave the region inconsistent
- Some work has moved into the applications:
  - e.g., self-validating, self-identifying records

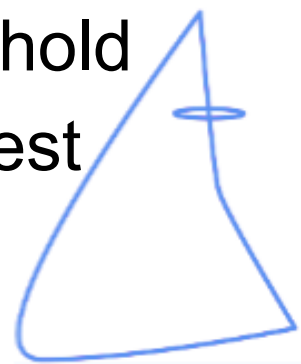


Relaxed consistency model (2/2)

- Simple, efficient
  - Google apps can live with it
  - what about other apps?
- Namespace updates atomic and serializable

Master's responsibilities (1/2)

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
  - give instructions, collect state, track cluster health
- Chunk creation, re-replication, rebalancing
  - balance space utilization and access speed
  - spread replicas across racks to reduce correlated failures
  - re-replicate data if redundancy falls below threshold
  - rebalance data to smooth out storage and request load



Master's responsibilities (2/2)

- Garbage Collection
  - simpler, more reliable than traditional file delete
  - master logs the deletion, renames the file to a hidden name
  - lazily garbage collects hidden files
- Stale replica deletion
  - detect "stale" replicas using chunk version numbers

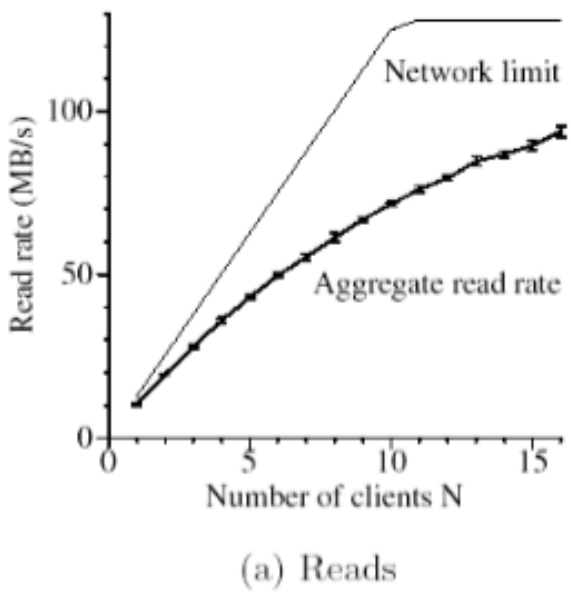
Fault Tolerance

- High availability
  - fast recovery
    - master and chunkservers restartable in a few seconds
  - chunk replication
    - default: 3 replicas.
  - shadow masters
- Data integrity
  - checksum every 64KB block in each chunk

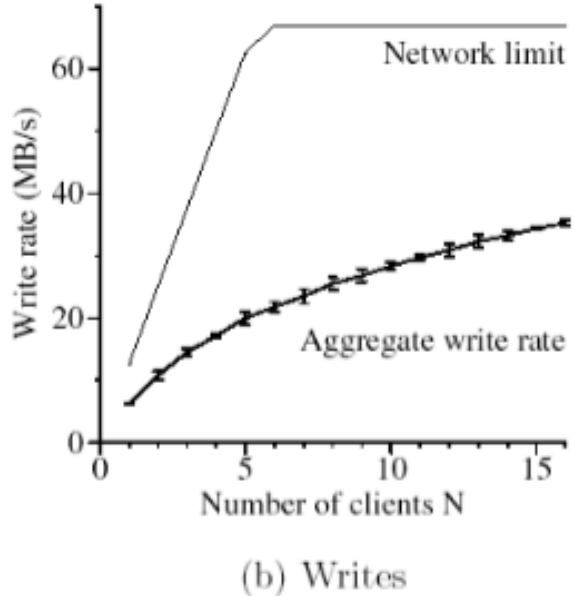
Scalability

- Scales with available machines, subject to bandwidth
  - Rack- and datacenter-aware locality and replica creation policies help
- Single master has limited responsibility, does not rate-limit system

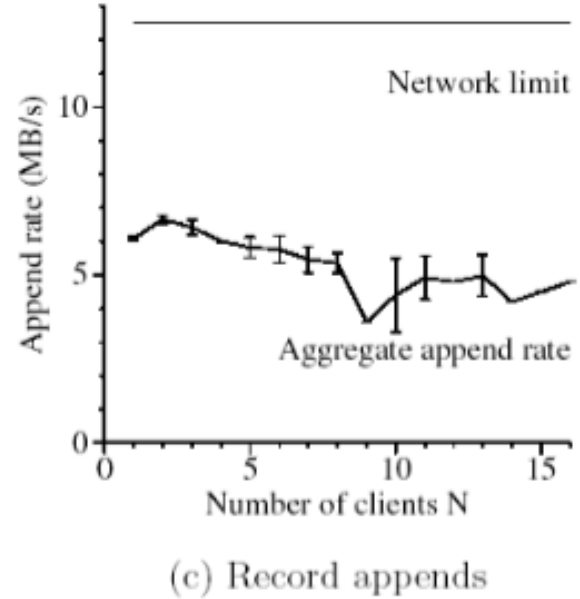
Scalability



(a) Reads



(b) Writes



(c) Record appends

- Microbenchmarks: 1—16 servers
  - Read performance 75—80% efficient (good!)
  - Write performance ~50% (network stack overhead)

Performance

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Security

- … Basically none
- Relies on Google's network being private
- File permissions not mentioned in paper
  - Individual users / applications must cooperate

Deployment in Google

- 50+ GFS clusters
- Each with thousands of storage nodes
- Managing petabytes of data
- GFS is under BigTable, etc.

Conclusion

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
  - design to tolerate frequent component failures
  - optimize for huge files that are mostly appended and read
  - feel free to relax and extend FS interface as required
  - go for simple solutions (e.g., single master)

- GFS has met Google's storage needs… it must be good!