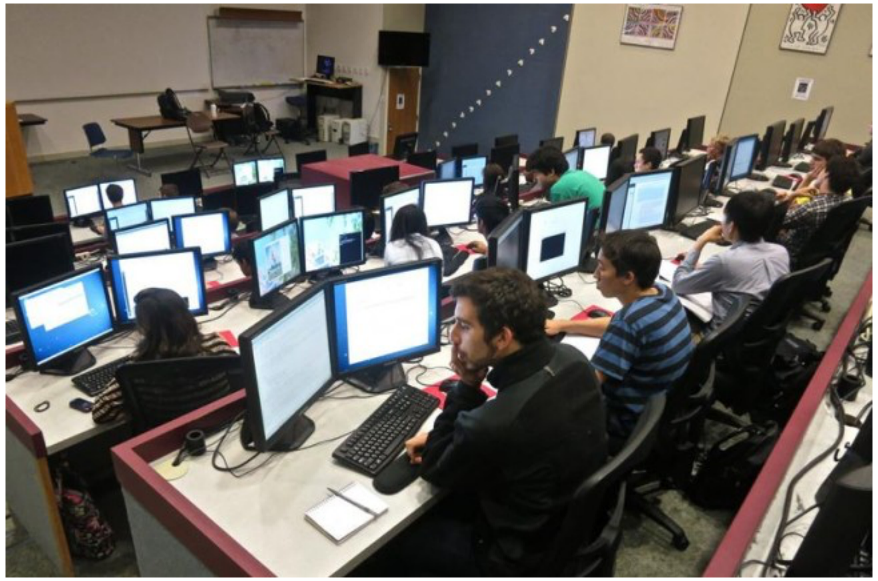
CLOUD SCALE STORAGE: THE GOOGLE FILE SYSTEM

CS6410

Harjasleen Malvai

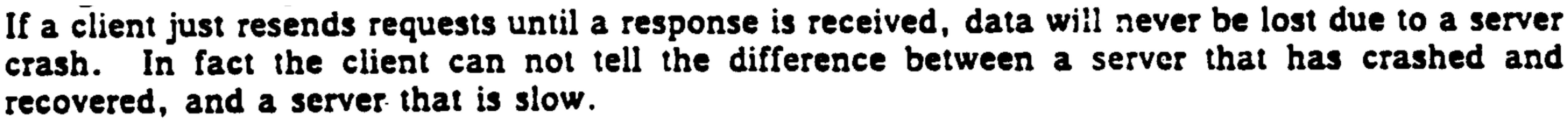
Where do the files go?

- Machines placed in a network need to share and use data.
- Introduces a few problems:
  - Plain old access
  - Consistency/Reliability
  - Availability


Source: Brown Daily Herald

Version 1.0: Network File System

- Introduced by Sun in 1985 (Sandberg et al. at USENIX).
- Interface looks like Unix File System: machine actually holding the file becomes "server", machine requesting becomes "client".
- Single copies stored.
- No locks, which might cause problems with concurrent modifications.
- There is a cache.
- Unreliable due to the fact that the strategy for getting files from server is based on:


If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact the client can not tell the difference between a server that has crashed and recovered, and a server that is slow.

Source: Sandberg, Russel, et al. "Design and implementation of the Sun network filesystem." *Proceedings of the Summer USENIX conference*. 1985.

Version 2.0: Sharing is Caring (p2p)

- Many untrusted nodes which can come and go store files. E.g. Napster, Limewire for p2p filesharing.
- Napster (1999) and its contemporaries had to maintain some centralized store of where files were or search all nodes for them, limiting scalability.
- Concurrent proposals (~2001) of various distributed hash tables: hash "keys" (e.g. file IDs) and/or node names, use some structure to speed up search for key locations (Chord, CAN, Tapestry, Pastry).
- Applications could include any distributed system with nodes leaving such as distributing nonce ranges to nodes in a mining pool!
- Using the distributed hash tables (among other new tools), the issues from Napster could be overcome: Systems such as Pond (2003) implemented scalable p2p data storage.
- Did not trust the hosts!

Source: Website

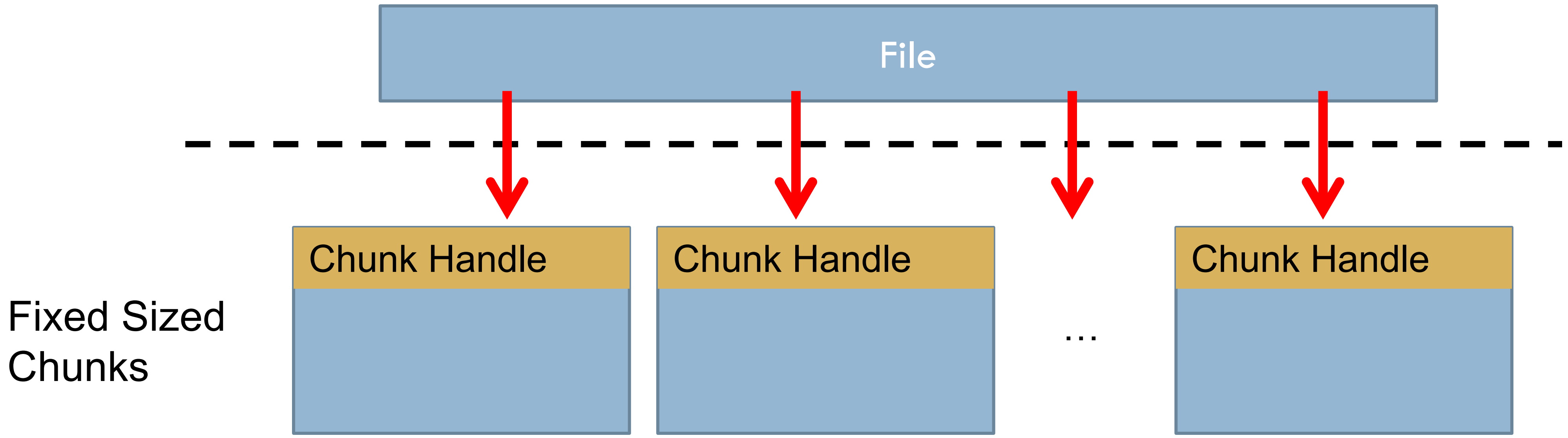
Why Google File System?

- Datacenter! Cheap commodity machines to run Google's operations with high bandwidth.
- Machines owned by Google, within data center, hence trusted!
- Need to design file system which accounted for:
  - Large scale distributed storage
  - Reliability
  - Availability

ASSUMPTIONS

- Hardware:
  - Using commodity hardware.
  - Component failures are common and need to be accounted for.
- Files:
  - Huge files are common so design needs to accommodate.
- Writes:
  - Most mutations are *appends* and *not overwrites.*
  - Concurrent modifications are to be accommodated.
- Reads:
  - Primarily large streaming reads and small random reads.
- Efficiency:
  - High bandwidth > low latency: Most applications process data at a high rate but do not have fast response requirements.

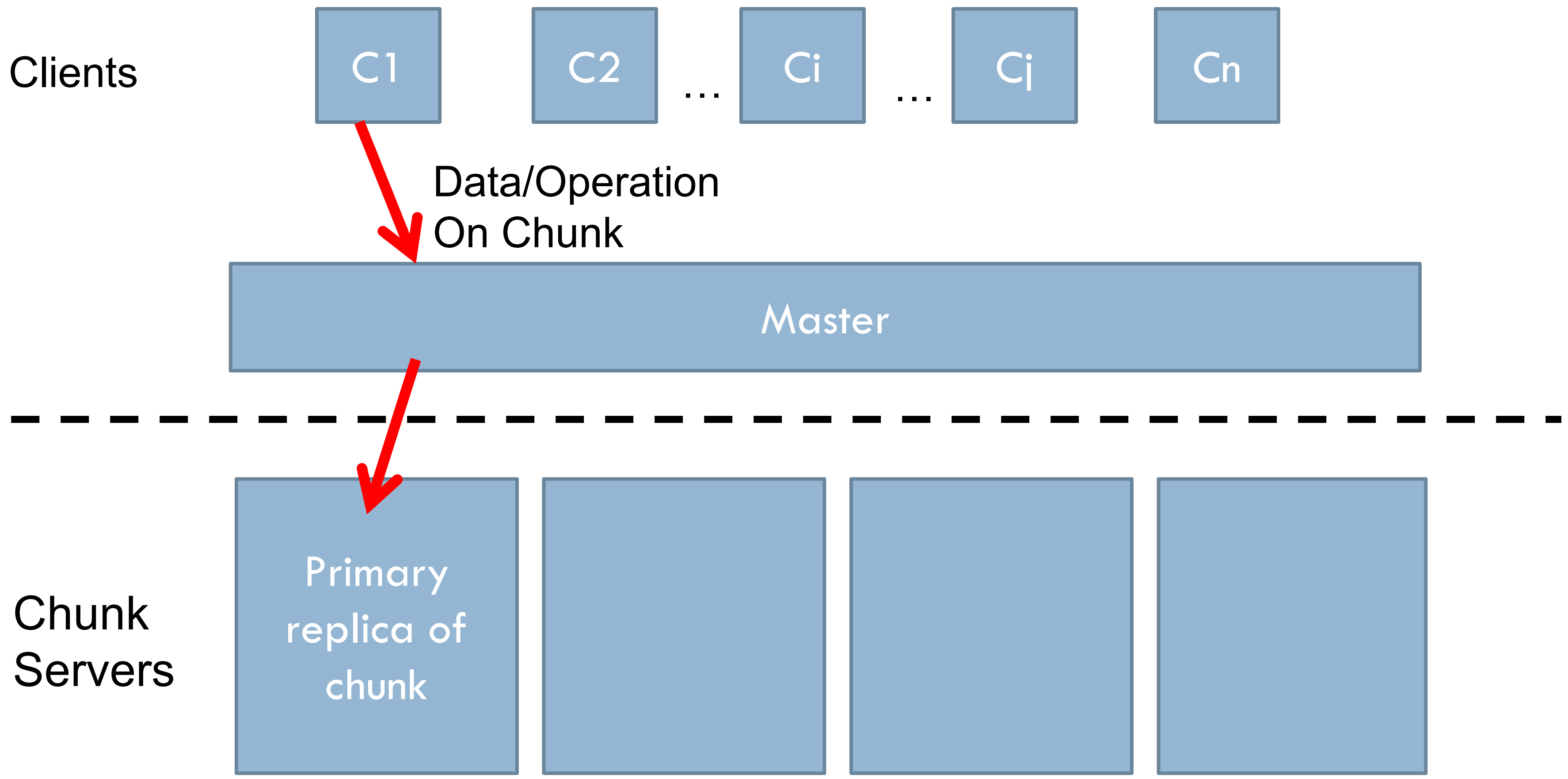
Data Under The Hood



Salient features:

- Chunk is treated as a Linux file on the hardware, Linux caching is implicitly used.
- Data is written at an offset within a chunk.
- Size is a parameter. They chose 64 MB.
- Many replicas (more on this later).

Architecture



Client Interaction

1. Client wants to *mutate* a chunk (write or append).
2. Master grants an arbitrarily extendible 60s lease for the chunk to a random *primary* with an up to date version (version checked with master metadata).
3. Replies to client with primary and replicas.
4. Client caches the primary and other chunk servers with replicas *(secondaries)*.
5. All edits are pushed to all replicas and write request is sent to the primary by the client.
6. Primary mutates and also makes an ordered list of write requests, accounting for multiple users sending write requests to the chunk.
7. Primary forwards list of writes, hence ensuring consistency.
8. Any errors from secondary writes are sent to client which handles re-tries.

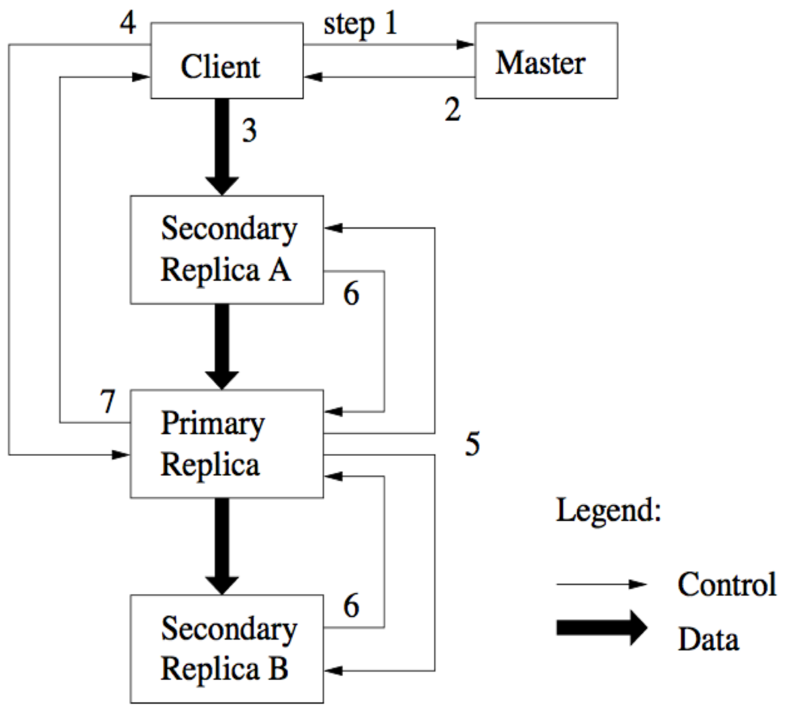


Figure 2: Write Control and Data Flow

Source: The Google File System

Problems Posed By GFS

Synchronization I

- Filesystem itself (namespace):
  - File/directory names saved as full pathnames in a lookup table, each with read/write locks.
  - File manipulation requires no locks from directory!
    - Why? "Because the old directory is dead!"
  - This implies:
    - Ability to snapshot while still writing to "directory".
    - Ability to write concurrently to "directory".

Synchronization II

- Multiple users editing a chunk
  - Atomic record appends:
    - Since primary is the authority on write operations, if multiple users send write requests, it is just treated as a multi-user write queue.
    - Details about chunk size being exceeded/needing new chunk.
    - Checksums contained in records to deal with resulting inconsistencies.
- Snapshots for versioning:
    - If snapshot requested, leases revoked, new copies created.
    - Copies created on the same machines to reduce network cost.
    - Revoked lease prevents new writes without master in the mean time.
- *Heartbeat* messages to keep master knowledge about chunks/servers current.
- *Operation Log* of mutations stored to replicated persistent memory for the *master.*

Availability

- Chunk replications via chunk-servers
  - Multi-level distribution
  - Multiple copies per *rack*.
  - Aim to keep copies on multiple *racks* in case specific routers fail.
- Master replication and logging
- Re-replication in case of failure:
  - Priority depending on degree of failure.
  - Trying to reduce bottlenecks by distributing new replicas.

Recovery

- Primary down!
  - Reconnect or new lease
  - Heartbeat messages keep track
- Master recovery
  - All mutations are saved to disk and not considered complete till replicated to all the backup masters.
  - Only background operations running in memory most of the time.
  - This means re-start or start of new master is seamless.*

Integrity

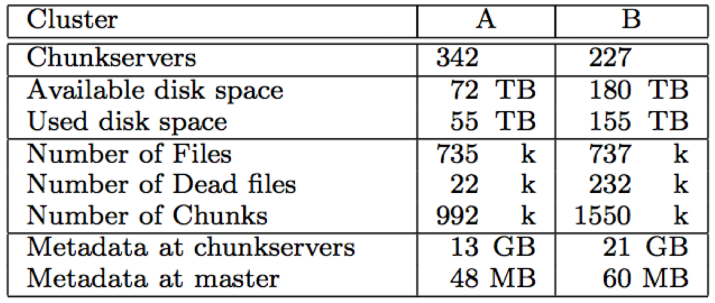
- Correctness of chunk mutations from *mutation order.*
- Checksums on chunk servers and checksum version numbers stored on master. Corroboration with client to ensure integrity.

Server Efficiency

- Memory efficiency:
  - Garbage collection
  - Load balancing
- Data flow efficiency (utilizing bandwidth)
- Diagnostics
- *Atomic record appends for* fast concurrent mutation.
- Avoiding bottlenecks by reducing role of master:
  - Once primary assigned, client only interacts with primary and secondaries.
  - Memory used only for “maintenance” operations such as garbage collection and load balancing.

Measurements

- Included measurements from real use cases!
- Low memory overhead for filesystem (see fig).
- It would appear memory bounds master but experiments show not an issue in practice.
- Some experiments with recovery:
  - Killed a single chunkserver (new replicas made in ~23 min).
  - Killed 16,000 chunkservers, leaving some chunks with single replica, hence high copy priority (all new replicas in ~2mins).



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

Table 2: Characteristics of two GFS clusters

Comments/Questions

- Application design specific to assumptions! How does this extend? What assumptions can we drop/need to drop?
- Chunk server recovery is analyzed but master recovery is not. Since the centralized controller in itself seems like a dangerous idea from an availability perspective, to what extent is this worrisome?
- Seems like the trust model is that the clients are somehow *internal* and will not try to launch a DoS on the master. Is this a good assumption? Provided, they do have the caveat of not trying to generalize.

CLOUD SCALE STORAGE: SPANNER: GOOGLE'S GLOBALLY DISTRIBUTED DATABASE

CS6410

Harjasleen Malvai

Why Spanner?

- Based on Colossus (successor to GFS)!
- Predecessors:
  - BigTable: Low functionality (no transactions), not strongly consistent. [Also uses GFS]
  - Megastore: Strong consistency but low write throughput.
- Google needed a (third!) tool which addressed these drawbacks.
- In addition on a global scale:
  - Client proximity matters for read latency.
  - Replica proximity matters for write latency.
  - Number of replicas matters for availability.

Spanner Solution

- Spanner solves this problem by implementing a derivative of BigTable with Paxos commits to support transactions.
- Spanner is “chunked” by rows having same or similar keys which they call “tablets”.
- Spanner deployments termed “universe” with physically isolated units known as “zones”.
- Zones have zonemasters and placemasters which serve values and move data around respectively.
- Since no longer in one physical location with single master, time synchronization poses a problem. They address this using their new API TrueTime.

TrueTime

- Each datacenter has various servers which provide time using GPS and atomic clocks.
- Time is no longer returned as an absolute but rather as an interval with real time guaranteed to be within the interval.
- Spanner holds off on certain serialized transactions if it is required with certainty that it is after a given time.
- Allows externally consistent snapshots.
- Now Paxos leaders can be selected disjointly.

Comments/Questions

- Fast distributed file systems and databases are possible but may need to limit assumptions!
- To what extent are corporate scale assumptions widely useful?