15-440 Distributed Systems

GFS / HDFS / Spanner

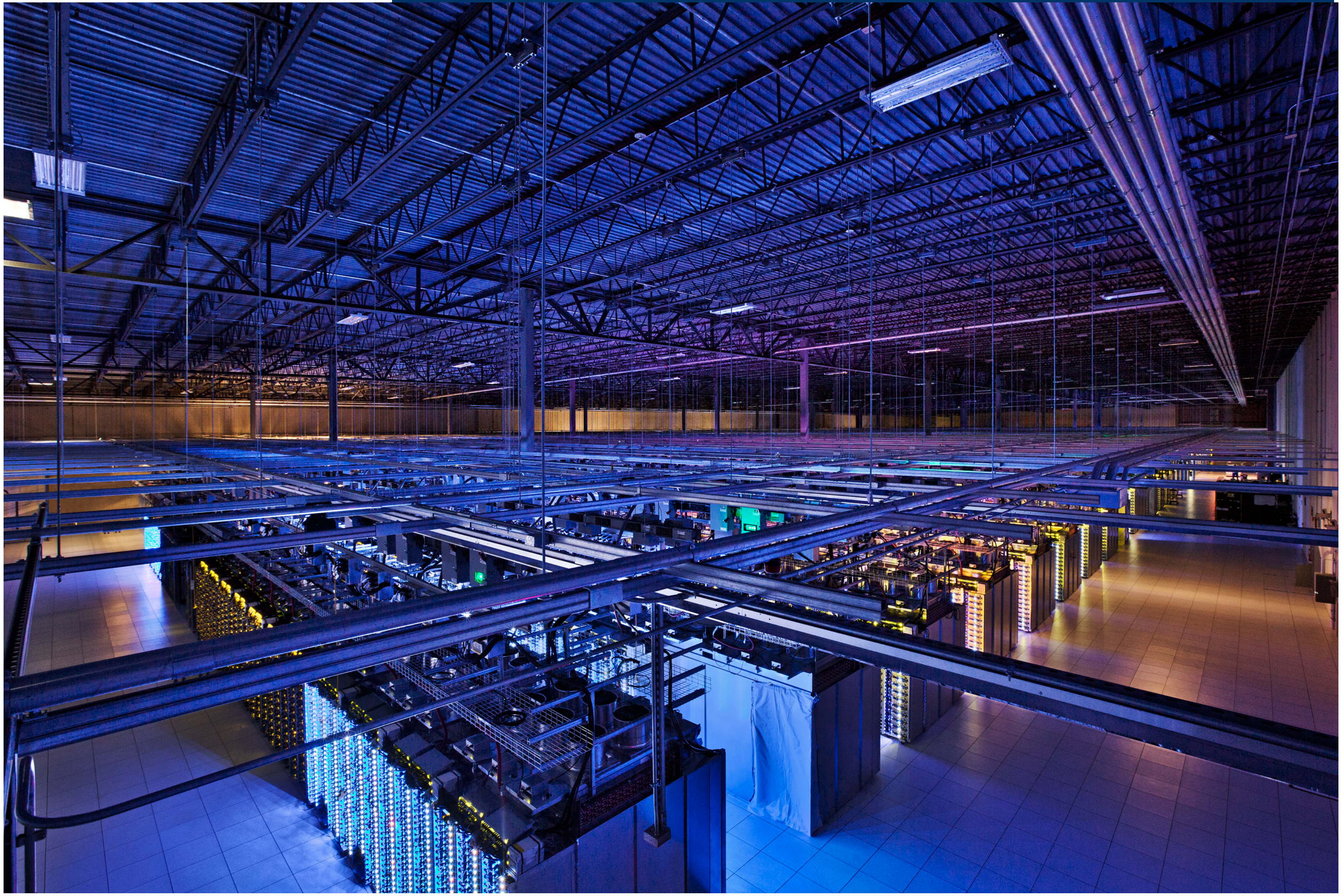
Agenda

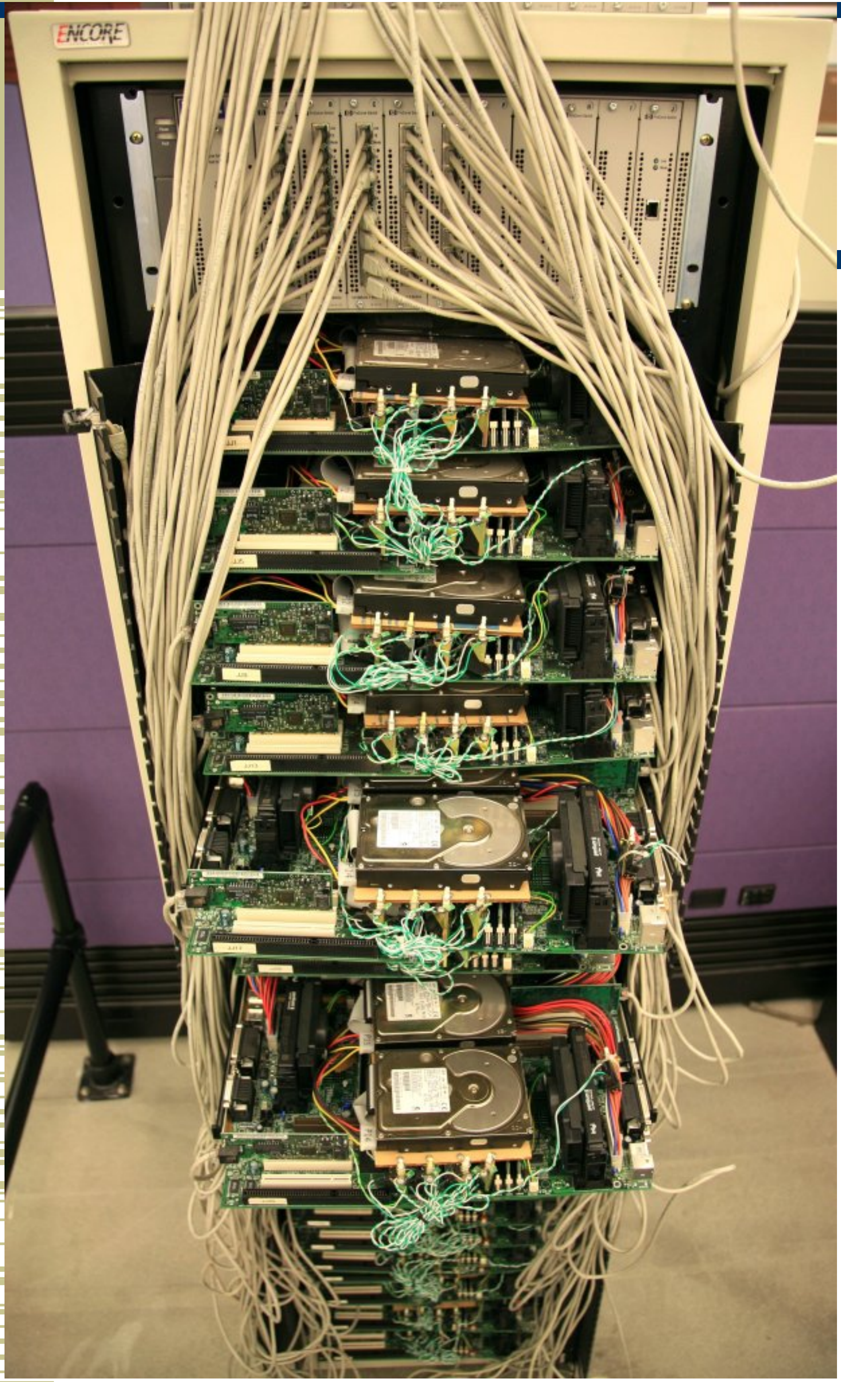
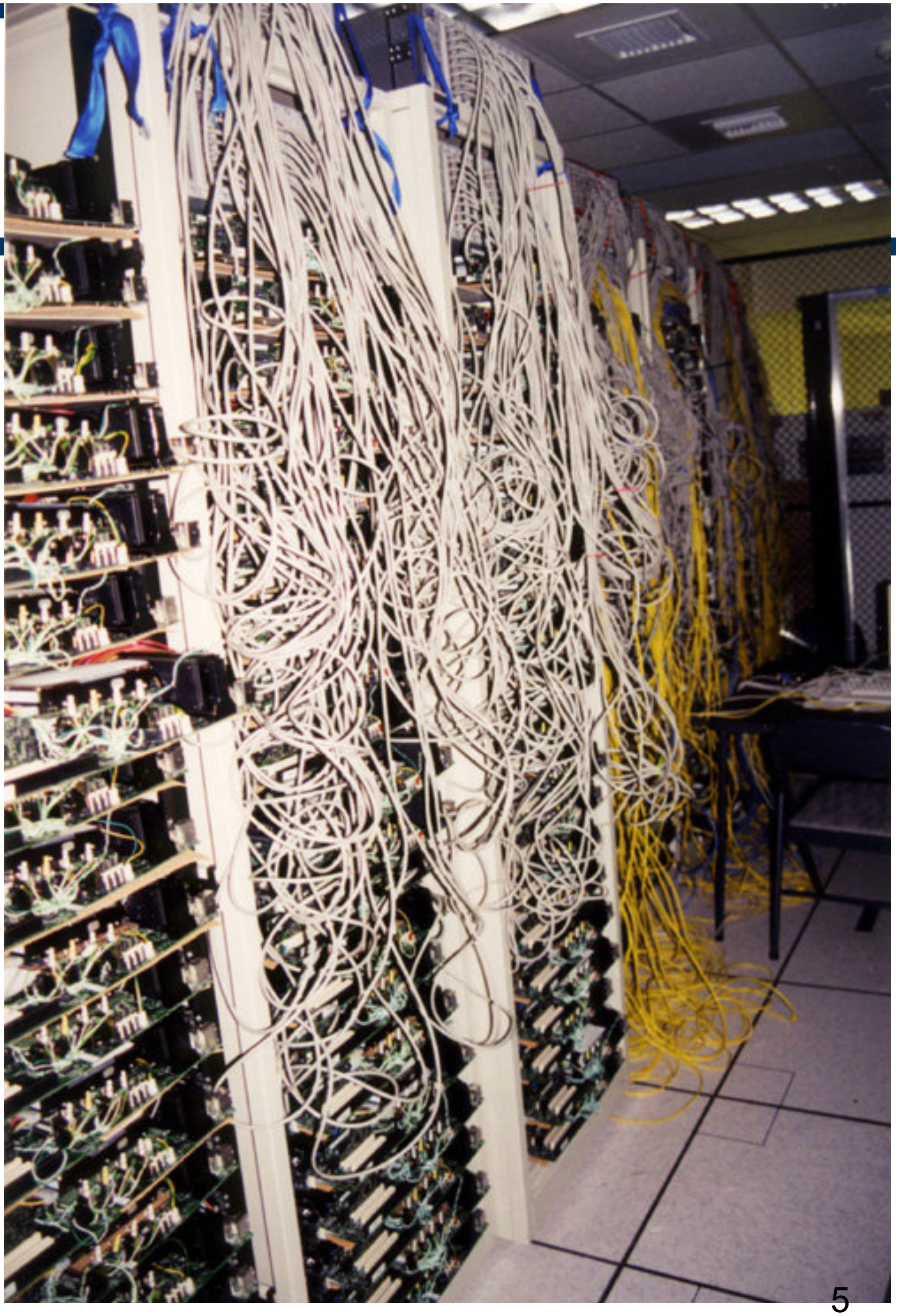
- Google File System (GFS)
- Hadoop Distributed File System (HDFS)
  - Distributed File Systems
  - Replication
- Spanner
  - Distributed Database System
  - Paxos
  - Replication
- Colossus (GFS v2)

GFS Motivation

- Google applications exercise specific read / write patterns (Gmail, YouTube, etc.)
- General purpose file systems (like Ext4, NTFS, etc.) are not designed to exploit specific workloads
- POSIX API (standard for file system communication) is an overkill for specific applications and their requirements
- Solution – design your own storage system
- **GFS is a distributed fault-tolerant file system**

GFS Operation Environment




ENCORE


GFS Operation Environment

- Hundreds of thousands of commodity servers
- Millions of commodity disks
- Failures are **normal (expected)**:
  - App bugs, OS bugs
  - Human error
  - Disk failure, memory failure, net failure, power supply failure
  - Connector failure
- Huge number of concurrent readers / writers

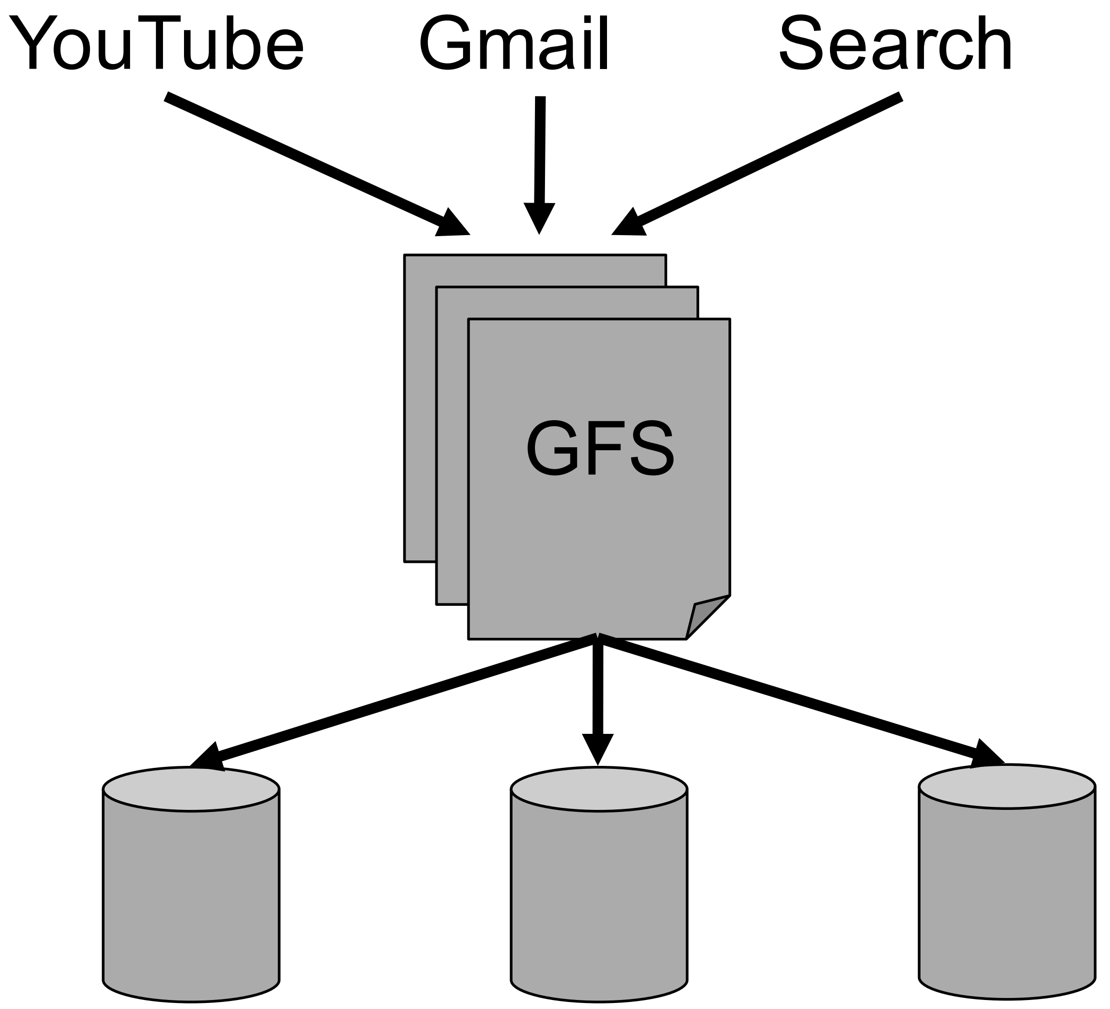
GFS Workload Assumptions

- (Relatively) Small (in the millions) # of large files
- Large files are >= 100 MB in size (multi-GB files common)
- Large, streaming reads (>= 1 MB in size)
- Large, sequential writes that append
- Concurrent appends by multiple clients (e.g., producer-consumer queues)

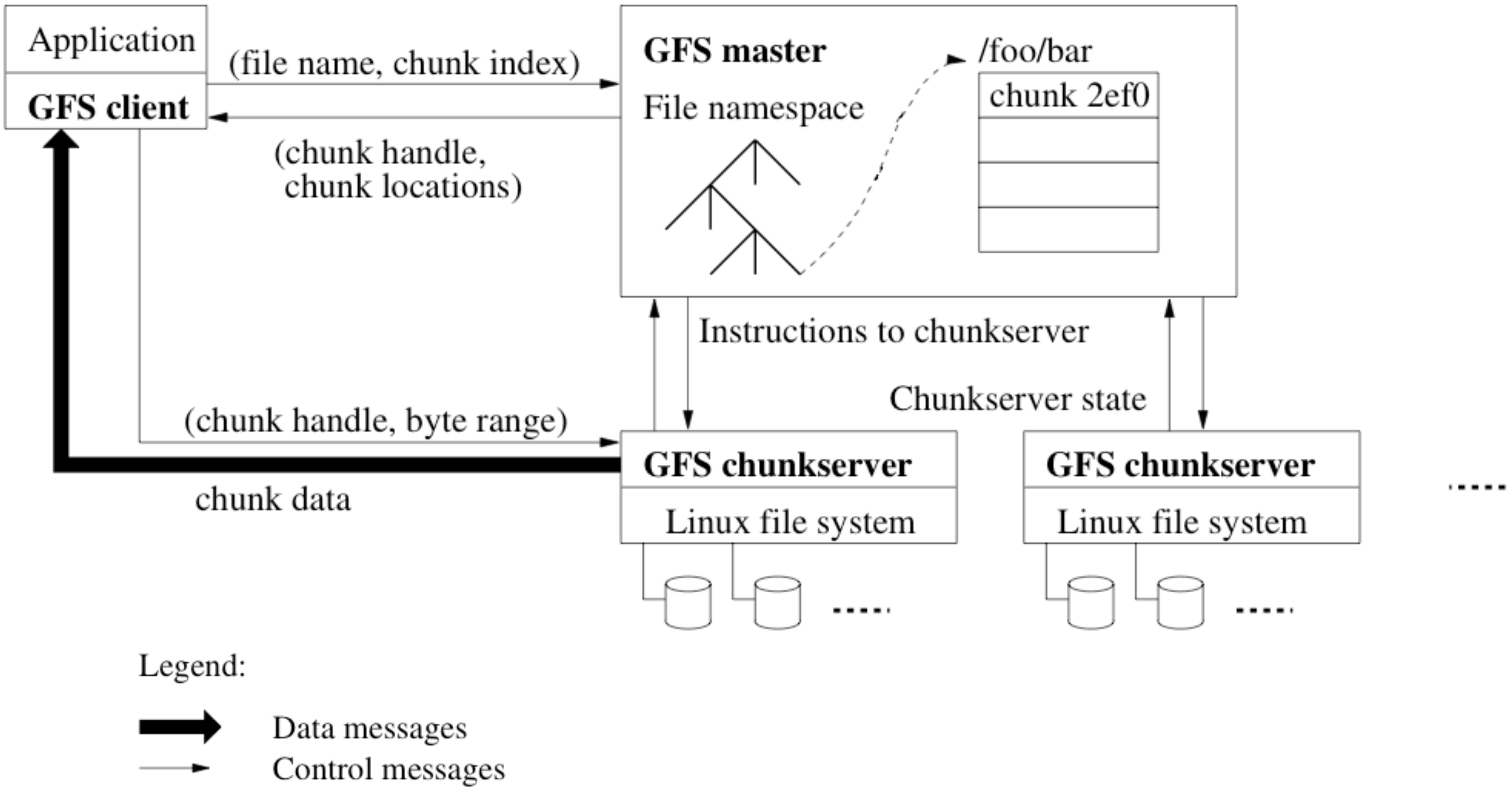
GFS Design Aims

- Maintain data and system availability
- Handle failures gracefully and transparently
- Low synchronization overhead between entities of GFS
- Exploit parallelism of numerous entities
- Ensure high sustained throughput over low latency for individual reads / writes

GFS Context



GFS Architecture



GFS Architecture

- One master server (state replicated on backups)
- Many chunk servers (100s – 1000s)
  - Spread across racks; intra-rack b/w greater than inter-rack
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
- Many clients accessing same and different files stored on same cluster

GFS Architecture Master Server

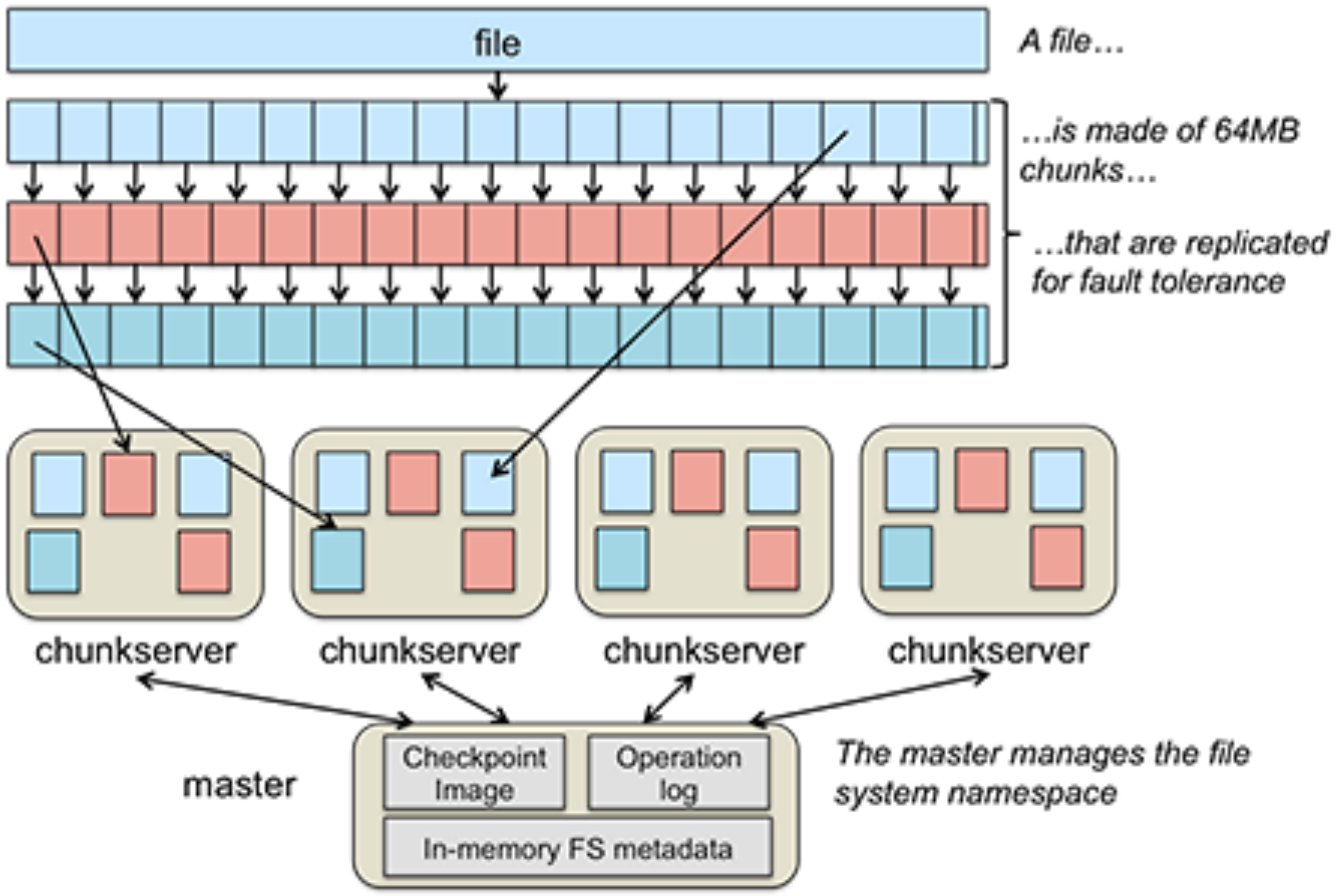
- Holds all metadata:
  - Namespace (directory hierarchy)
  - Access control information (per-file)
  - Mapping from files to chunks
  - Current locations of chunks (chunkservers)
- Delegates consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers
  - Why is migration needed?

Holds all metadata in RAM; very fast operations on file system metadata

GFS Architecture Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem (like Ext4), each with version number and checksum
  - What is the traditional file system chunk / block size?
  - Why 64 MB?
- Has no understanding of overall file system (just deals with chunks)
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data (beyond standard Linux buffer cache)
- Send periodic heartbeats to Master

GFS Architecture File Layout



GFS Architecture Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
  - This exploits parallelism and reduces master bottleneck
- Caches metadata
- Does no caching of data
  - No consistency difficulties among clients
  - Streaming reads (read once) and append writes (write once) don’t benefit much from caching at client

GFS Architecture Client

- No file system interface at the operating-system level (e.g., under the VFS layer
  - User-level API is provided
  - Does not support all the features of POSIX file system access – but looks familiar (i.e. open, close, read…)

- Two special operations are supported.
  - **Snapshot**: is an efficient way of creating a copy of the current instance of a file or directory tree.
  - **Append**: allows a client to append data to a file as an atomic operation without having to lock a file. Multiple processes can append to the same file concurrently without fear of overwriting one another's data

GFS Working Client Read

- Client sends master:
  - **read(file name, chunk index)**
- Master’s reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends “closest” chunkserver w/replica:
  - **read(chunk ID, byte range)**
  - “Closest” determined by IP address on simple rack-based network topology
- Chunkserver replies with data

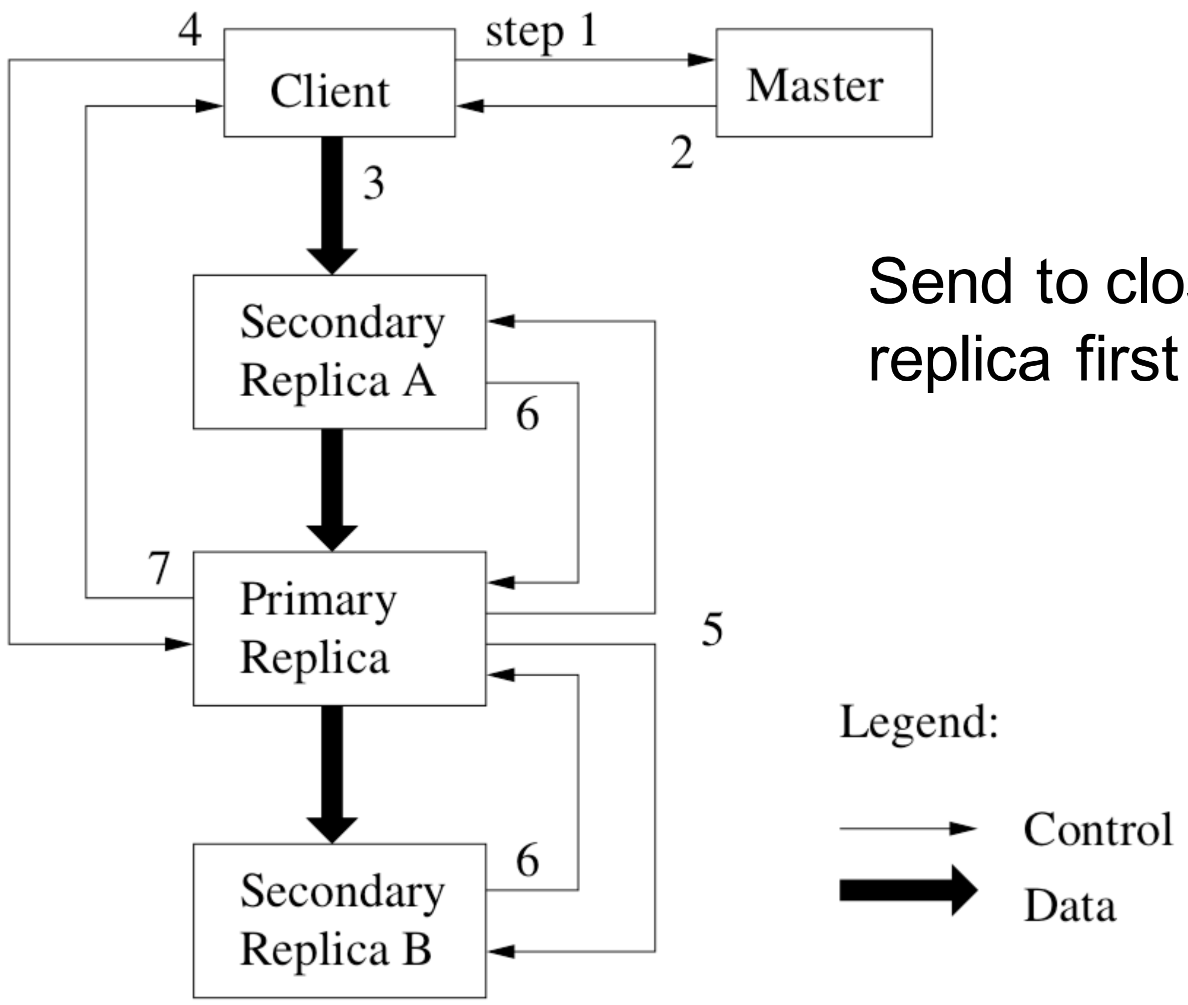
GFS Working Client Write

- 3 replicas for each block → must write to all
- When block created, Master decides placements
  - Default: two within single rack, third on a different rack
  - Why?
    - Access time / safety tradeoff

GFS Working Client Write

- Some chunkserver is primary for each chunk
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed using periodic heartbeat messages between master and chunkservers
- Client asks master for primary and secondary replicas for each chunk
- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives
  - Takes advantage of full-duplex Ethernet links

GFS Working Client Write



Send to closest replica first

GFS Working Client Write

- All replicas acknowledge data write to client
  - Don't write to file → just get the data
- Client sends write request to primary (commit phase)
- Primary assigns serial number to write request, providing ordering
- Primary forwards write request with same serial number to secondary replicas
- Secondary replicas all reply to primary after completing writes **in the same order**
- Primary replies to client

GFS Working Client Record Append

- Google uses large files as queues between multiple producers and consumers
- Same control flow as for writes, except…
- Client pushes data to replicas of last chunk of file
- Client sends request to primary
- Common case: request fits in current last chunk:
  - Primary appends data to own replica
  - Primary tells secondaries to do same at same byte offset in theirs
  - Primary replies with success to client

GFS Working Client Record Append

- When data won’t fit in last chunk:
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, “retry on next chunk”
- If record append fails at any replica, client retries operation
  - So replicas of same chunk may contain different data—even duplicates of all or part of record data
- What guarantee does GFS provide on success?
  - Data written at least once in atomic unit

GFS Working File Deletion

- When client deletes file:
  - Master records deletion in its log
  - File renamed to hidden name including deletion timestamp
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable)
  - In-memory metadata erased
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers

GFS Working Logging at Master

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups!
- Logs cannot be too long – why?
- Periodic checkpoints as an on-disk Btree

GFS Working Chunk Leases and Version Numbers

- If no outstanding lease when client requests write, master grants new one
- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease, increments version, informs all replicas
- Master can revoke leases
  - e.g., when client requests rename or snapshot of file

GFS Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses log to define global total order of namespace-changing operations

GFS Consistency Model

- Changes to data are ordered as chosen by a primary
  - But multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients
- Record append completes at least once, at offset of GFS's choosing
  - Applications must cope with possible duplicates
- Failures can cause inconsistency
  - Behavior is worse for writes than appends

GFS Fault Tolerance (Master)

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping (but not location of chunks)
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale
  - Chunk server down at lease renewal
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease

GFS Fault Tolerance (Chunkserver)

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing greatest number of replicas

GFS Limitations

- Security?
  - Trusted environment, trusted users
  - But that doesn't stop users from interfering with each other…
- Does not mask all forms of data corruption
  - Requires application-level checksum

GFS Limitations

- Master biggest impediment to scaling
  - Performance bottleneck
  - Holds all data structures in memory
  - Takes long time to rebuild metadata
  - Must vulnerable point for reliability
  - Solution:
    - Have systems with multiple master nodes, all sharing set of chunk servers.
    - Not a uniform name space.
- Large chunk size.
  - Can’t afford to make smaller, since this would create more work for master.

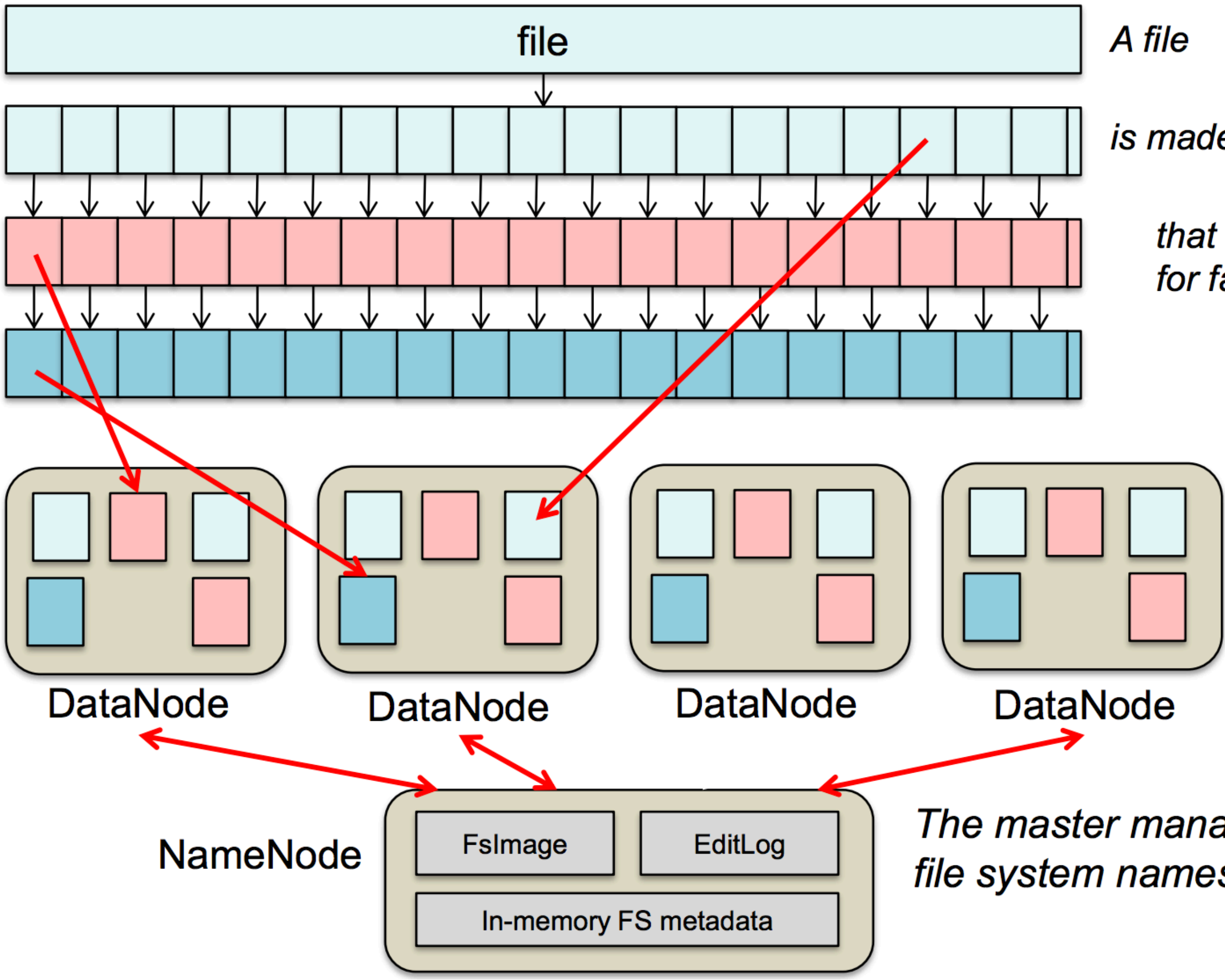
GFS Summary

- Success: used actively by Google to support search service and other applications
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)

- Replaced in 2010 by Colossus
  - Eliminate master node as single point of failure
  - Targets latency problems due to more latency sensitive applications
  - Reduce block size to be between 1~8 MB
  - Few details public ☹

HDFS

HDFS



A file

is made of 128 MB blocks

that are replicated for fault tolerance

Hmm… looks familiar

Blocks live on DataNodes

The master manages the file system namespace

GFS vs. HDFS

GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
random file writes possible	only append is possible
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 64KB data and 32bit checksum pieces	per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp)
default block size: 64MB	default block size: 128MB

Spanner

Spanner

- Spanner is a scalable globally-distributed database system.
- Replication is used for global availability and geographic locality.
- Why do you need Spanner when you have GFS?
  - Transactions - Google argues that it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.

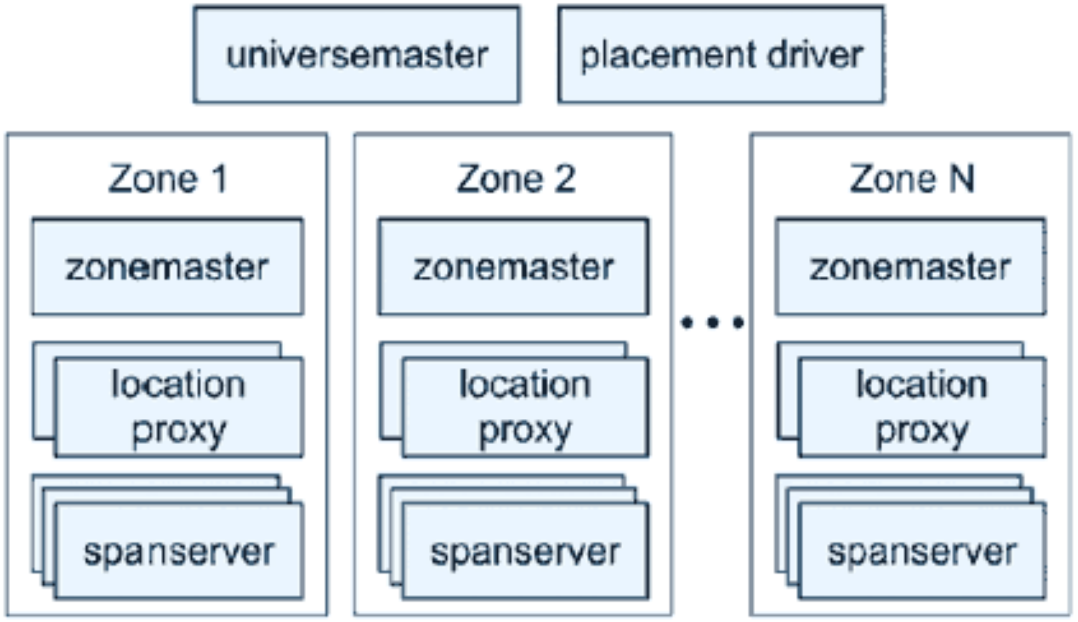
Spanner Operation Environment

- Replication across continents
- Data is sharded 1000s of ways within each data center
- Paxos state machines used across the world manage the sharded data
  - Why is Paxos needed in a database system?
- Expected to operate using million of machines across hundreds of data centers containing trillions of rows

Spanner Motivation

- Applications requiring
  - complex, evolving schemas
  - strong consistency in wide-area replication
  - high availability in wide-area natural disasters
  - fine grained control over data replication and location
- SQL like interface to query data
  - Since Dremel (an interactive data analysis tool) was very popular

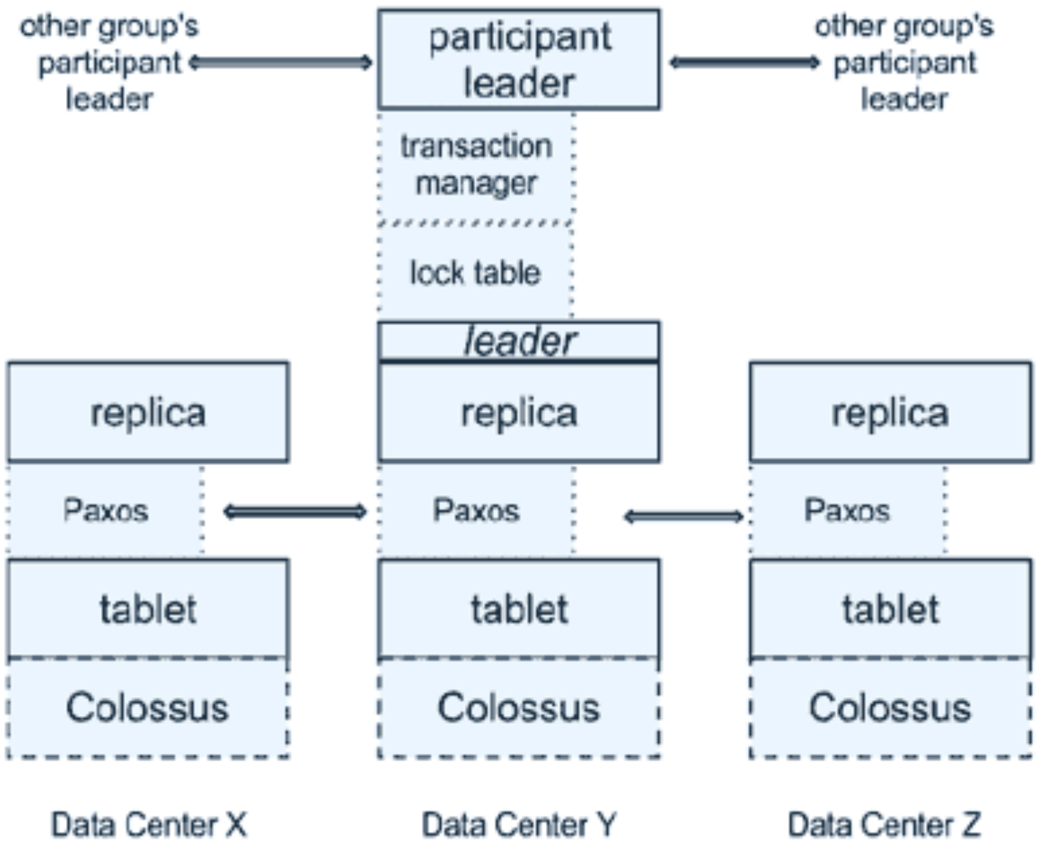
Spanner Deployment



Spanner Deployment

- Spanner deployment is called **universe**
- **Zone** is the unit of administrative deployment and physical isolation (i.e. different data centers will be in different zones)
- 100 to 1000s of **spanservers** in each zone

Spanserver Architecture



Spanserver Architecture

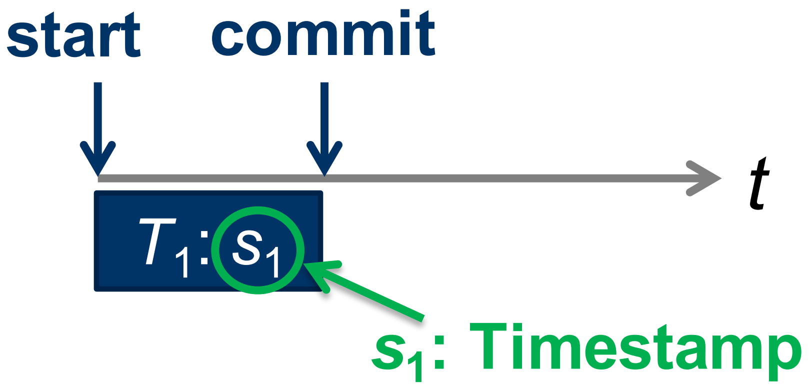
- 100 – 1000 BigTable tablets at the lowest layer writing to **Colossus** (successor to GFS)
- Atop each tablet runs a **Paxos** state machine
- Collection of Paxos state machines is called a **Paxos Group** and every Paxos Group has a **leader**
- Leader implements the **lock table** to maintain the 2-phase locking state

Spanner Directories

- Spanner is essentially a glorified hash map
- A **directory** is a bucket of contiguous keys that share a common prefix
- A directory is the unit of data placement in Spanner
- It is also the smallest unit whose replication and geographic location settings can be customized

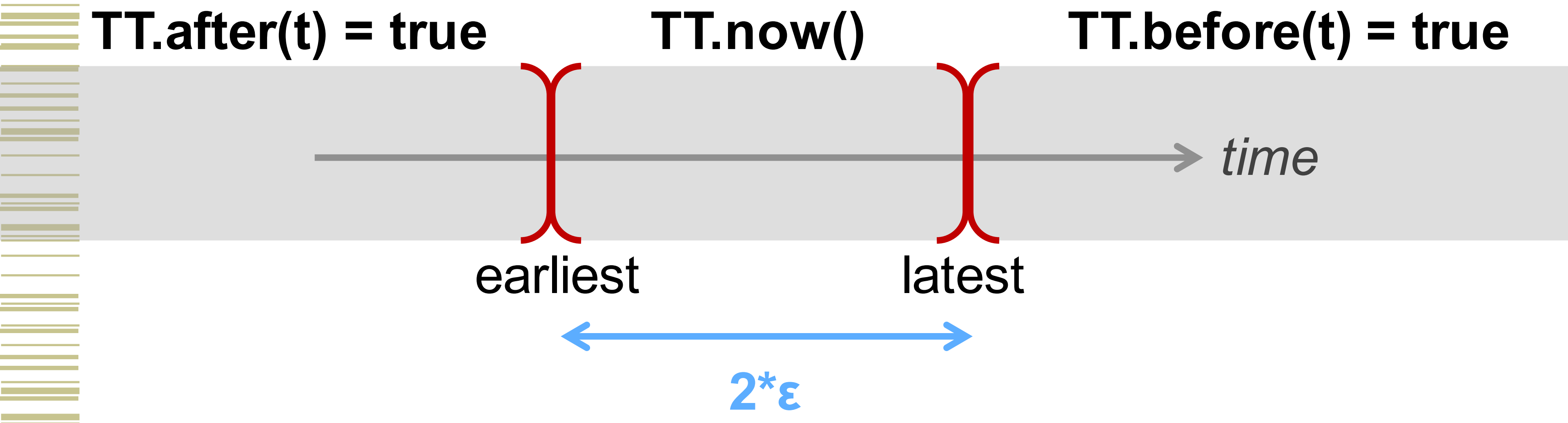
Spanner Concurrency Control

- Key aspect of differentiating Spanner – using globally meaningful timestamps for distributed transactions in achieving external consistency



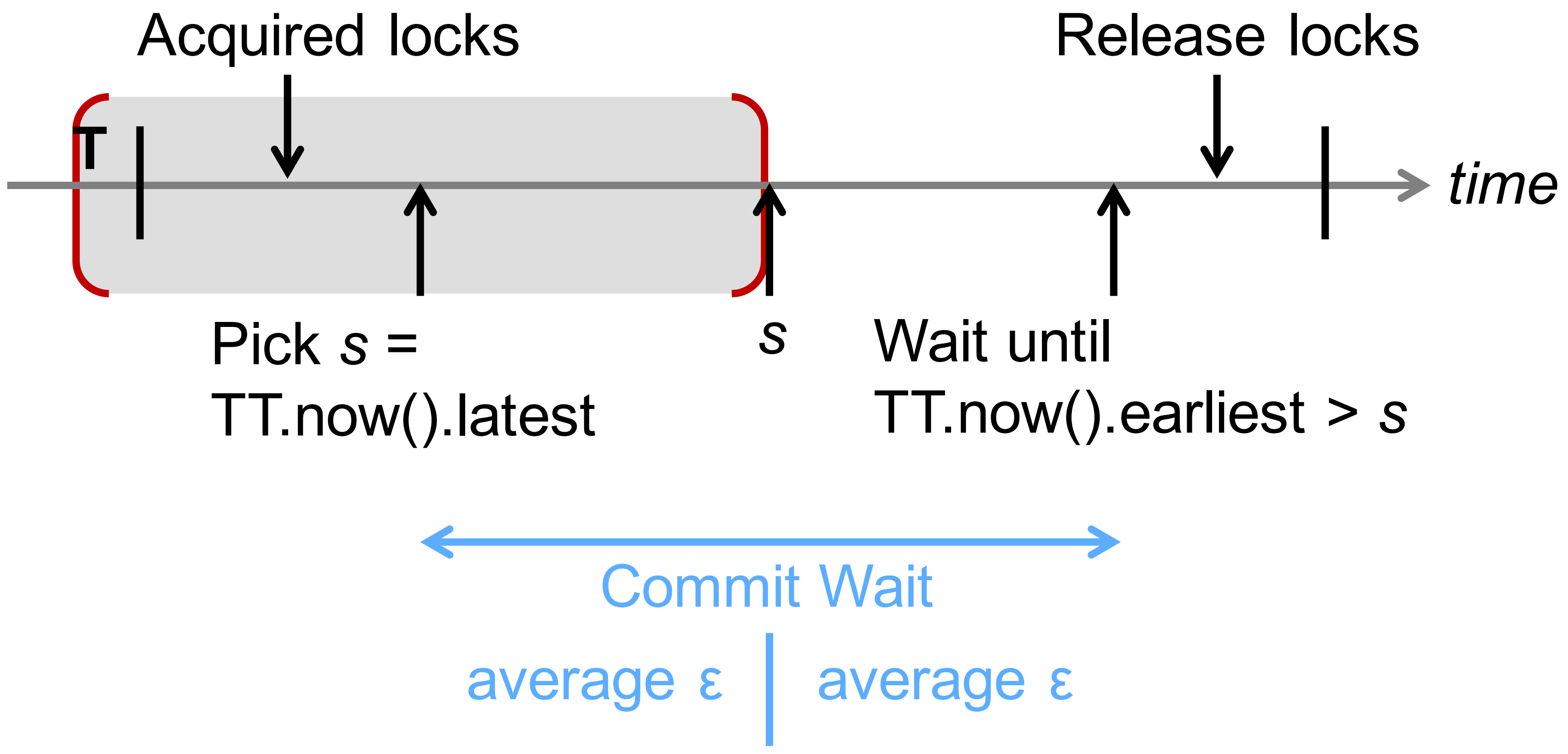
Spanner TrueTime API

TT.now() – TTInterval: [earliest, latest]
(Why is time an interval and not a specific instant?)
TT.after(t) – true if **t** has definitely passed
TT.before(t) – true if **t** has definitely not arrived



"Global wall-clock time" with bounded uncertainty

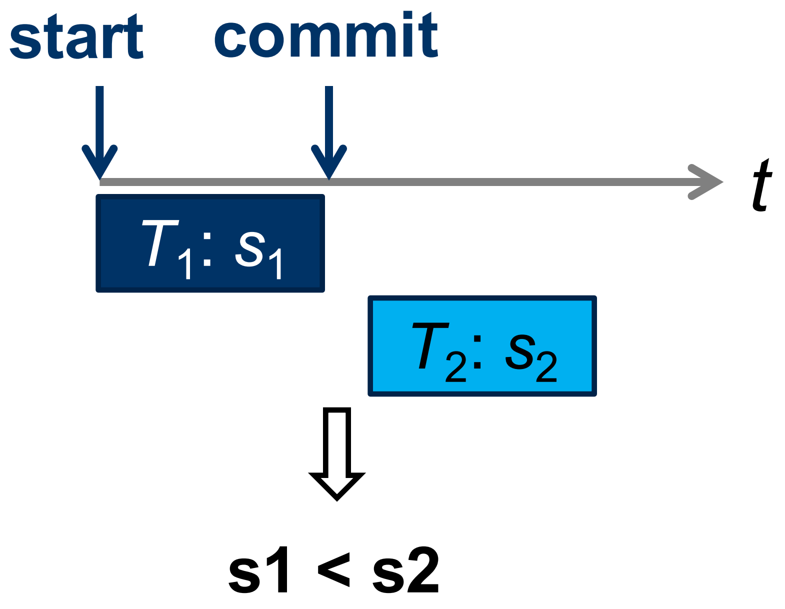
Spanner Commit Wait

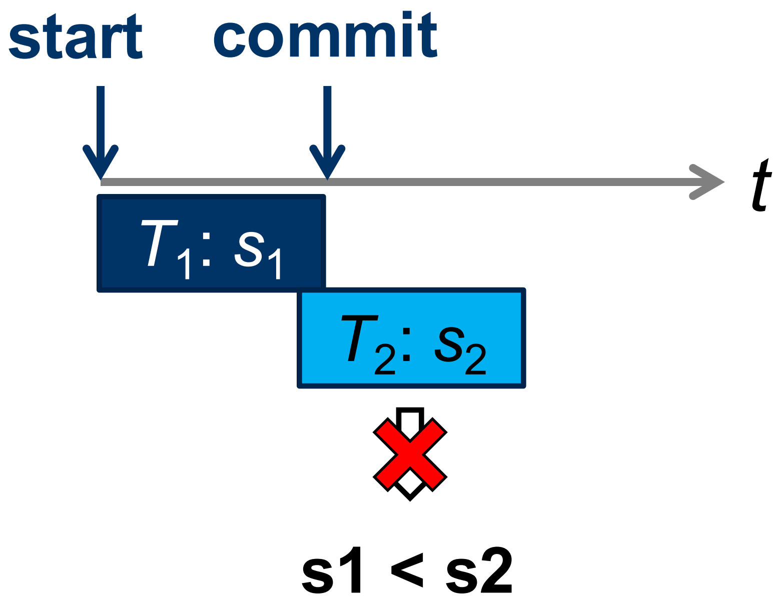


Why do you need to wait for TT.now().earliest > s before releasing locks?

Spanner External Consistency

- If a transaction T_1 commits before another transaction T_2 starts, then T_1's commit timestamp is smaller than T_2
- Similar to how we reason with wall-clock time





Summary

- GFS / HDFS
  - Data-center customized API, optimizations
  - Append focused DFS
  - Separate control (filesystem) and data (chunks)
  - Replication and locality
  - Rough consistency → apps handle rest
- Spanner
  - Globally consistent replicated database system
  - Implements distributed transactions
  - Lock-free reads and Paxos based writes
  - Implements external consistency using TrueTime API
  - Able to survive data center wipeouts

Colossus of today

Distributed File System Allocation

- Data centers consist of millions of disks
- All of them not added at the same time
- Advances in technology → larger capacity disks
  - Kryder's Law
- Disk speeds don't increase at the rate of capacity
- Disk capacity agnostic allocation algorithms result in hot-spots and thus bottlenecks
  - What are hot-spots?

Multidimensional Bin-packing

- Data can be categorized by temperature
  - Hot data is data read / written frequently
  - Cold data is data rarely read / written
- Smarter allocation algorithms can be devised keeping heat and disk capacity in mind
- Essentially bin-packing as per multiple criteria