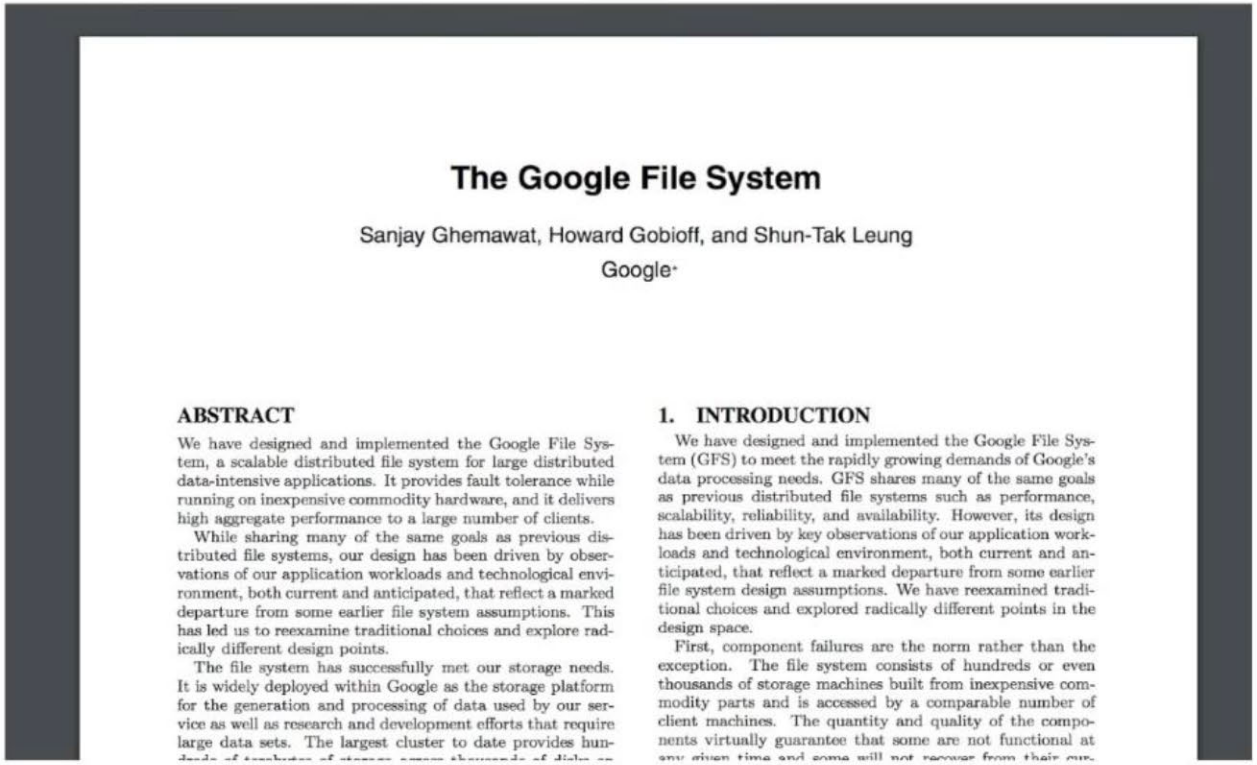
Chapter 2
Google File System

Bal Krishna Nyaupane
Assistant Professor
Department of Electronics and Computer Engineering
Institute of Engineering, Tribhuvan University
bkn@wrc.edu.np

Google File System (GFS)



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hun-

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

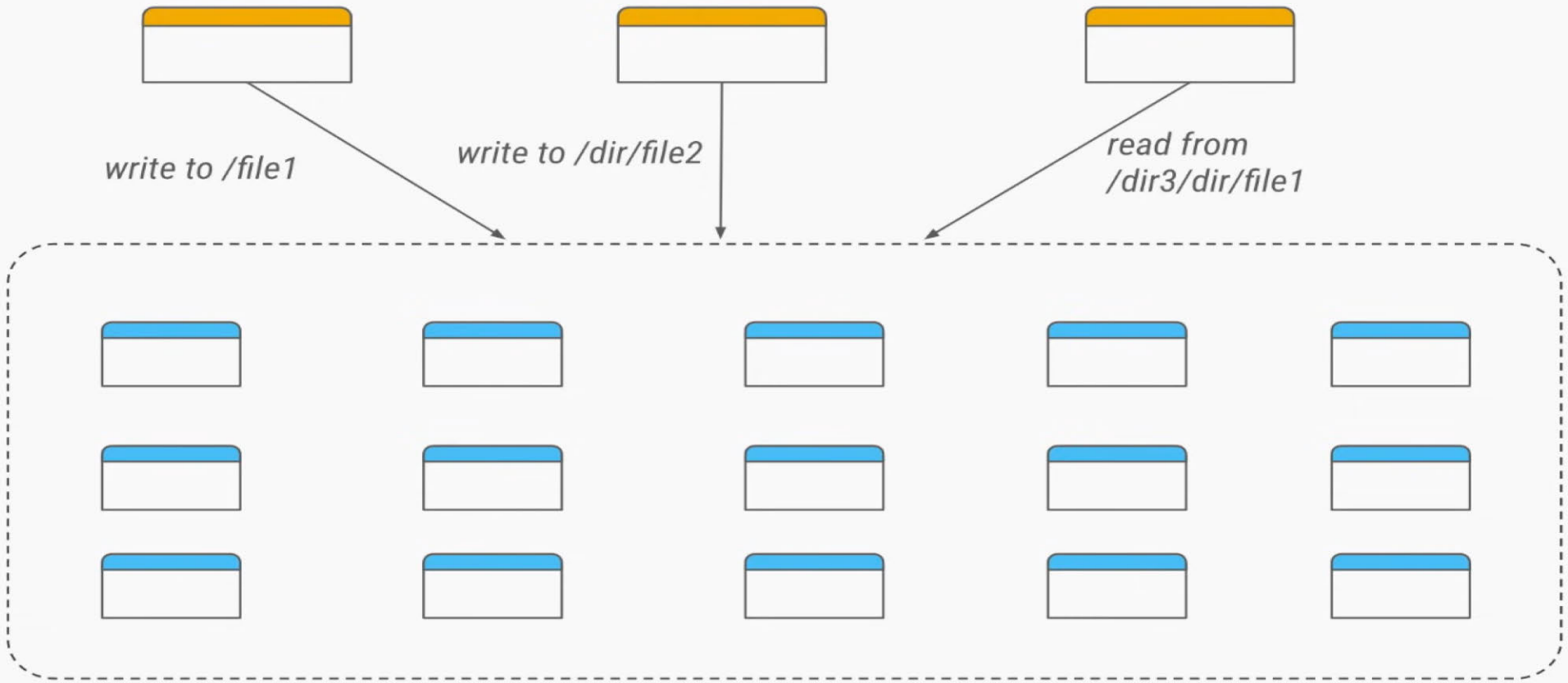
First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at

Paper describing Google File System

Used as a basis to design Hadoop

Google File System (GFS)

What is Google File System



Distributed File Storage

1 cluster = 100s of commodity servers

Google File System (GFS)

- A ***scalable distributed file system for large distributed data-intensive applications.***
- ***Provides fault tolerance*** while running on inexpensive commodity hardw are and delivers high aggregate performance to a large number of clients.
- Designed and implemented ***to meet the rapidly growing demands of Google's*** data processing needs.
- Shares many of the same goals as previous distributed file systems such as ***performance, scalability, reliability, and availability.***
- The design of GFS is driven by **four key observations**
  - ***Component failures***, ***huge files***, ***mutation of files***, and ***benefits of co-designing the applications and file system API***.

Google File System (GFS)

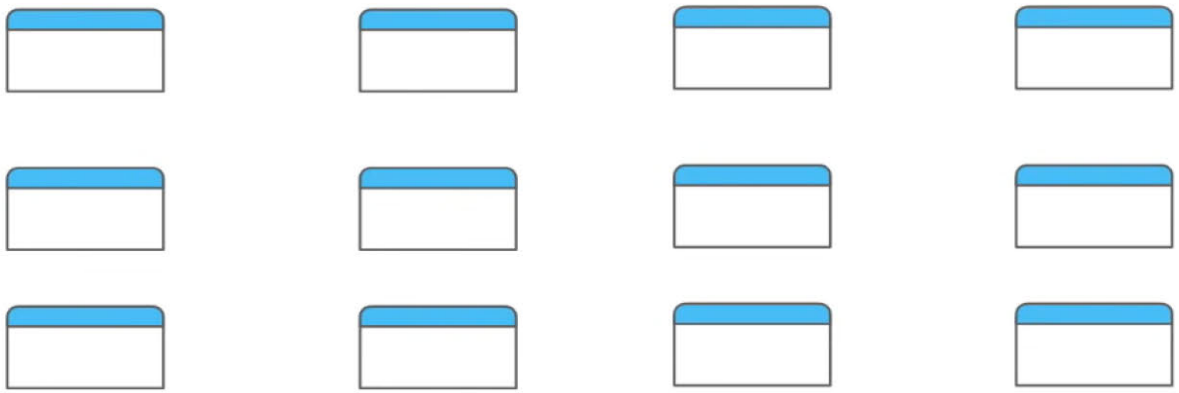
- ***Component failures*** are the norm rather than the exception
  - The system is built from ***many inexpensive commodity components*** that often fail.
  - Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.
- ***Files are huge***
  - A few million files, each typically 100 MB or larger in size are expected.
  - Multi GB files are the common case and should be managed efficiently.

GFS Design Overview: Assumptions

Commodity Hardware

Failures are common

- disk / network / server
- OS bugs
- human errors



Commodity servers are cheap & can be made to scale horizontally with right software

GFS Design Overview: Assumptions

- ***The system is built from many inexpensive commodity components that often fail.***
  - It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- ***The system stores a modest number of large files.***
  - Expect a few million files, each typically 100 MB or larger in size.
  - Multi-GB files are the common case and should be managed efficiently.
  - Small files must be supported, but we need not optimize for them.

GFS Design Overview: Assumptions

- The workloads primarily consist of two kinds of reads***: large streaming reads*** and ***small random reads.***
  - ***In large streaming reads***, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file.
  - ***A small random read typically*** reads a few KBs at some arbitrary offset. Performance-conscious applications ***often batch and sort their small reads to advance steadily through the file*** rather than go back and forth.

GFS Design Overview: Assumptions

- ***The workloads also have many large, sequential writes that append data to files.***
  - Once written, files are seldom modified again.
  - Small writes at arbitrary positions in a file are supported ***but do not have to be efficient.***
- ***High sustained bandwidth*** is more important ***than low latency.***
  - Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.
  - *Having **users wait a little longer for their requests** to be processed is better than **having to focus on only a small number of users at once** and leaving everyone else waiting.*

GFS Design Overview: Interface

- Files are organized ***hierarchically in directories*** and ***identified by pathnames***.
- Support the usual operations to ***create, delete, open, close, read, and write files***.
- GFS has ***snapshot and record append operations***.
- ***Record append operations***
  - GFS provides an ***atomic append operation called record append***.
  - ***Allow multiple clients to append data to the same file concurrently.***
  - Client specifies only the data. GFS appends it to the file at least once atomically a (i.e., As one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client.

GFS Design Overview: Interface

- ***Snapshot***
  - Snapshot ***creates a copy of a file*** or ***a directory tree at low cost.***
  - The snapshot operation ***makes a copy of a file or a directory tree (the "source") almost instantaneously***, while minimizing any interruptions of ongoing mutations.
  - GFS use ***standard copy-on-write techniques to implement snapshots***.
  - **When the master receives a snapshot request**, it first revokes any outstanding leases on the chunks in the files it is about to snapshot.
  - This ***ensures that any subsequent writes to these chunks will require an interaction*** with the master to find the lease holder.

GFS Design Overview: Architecture

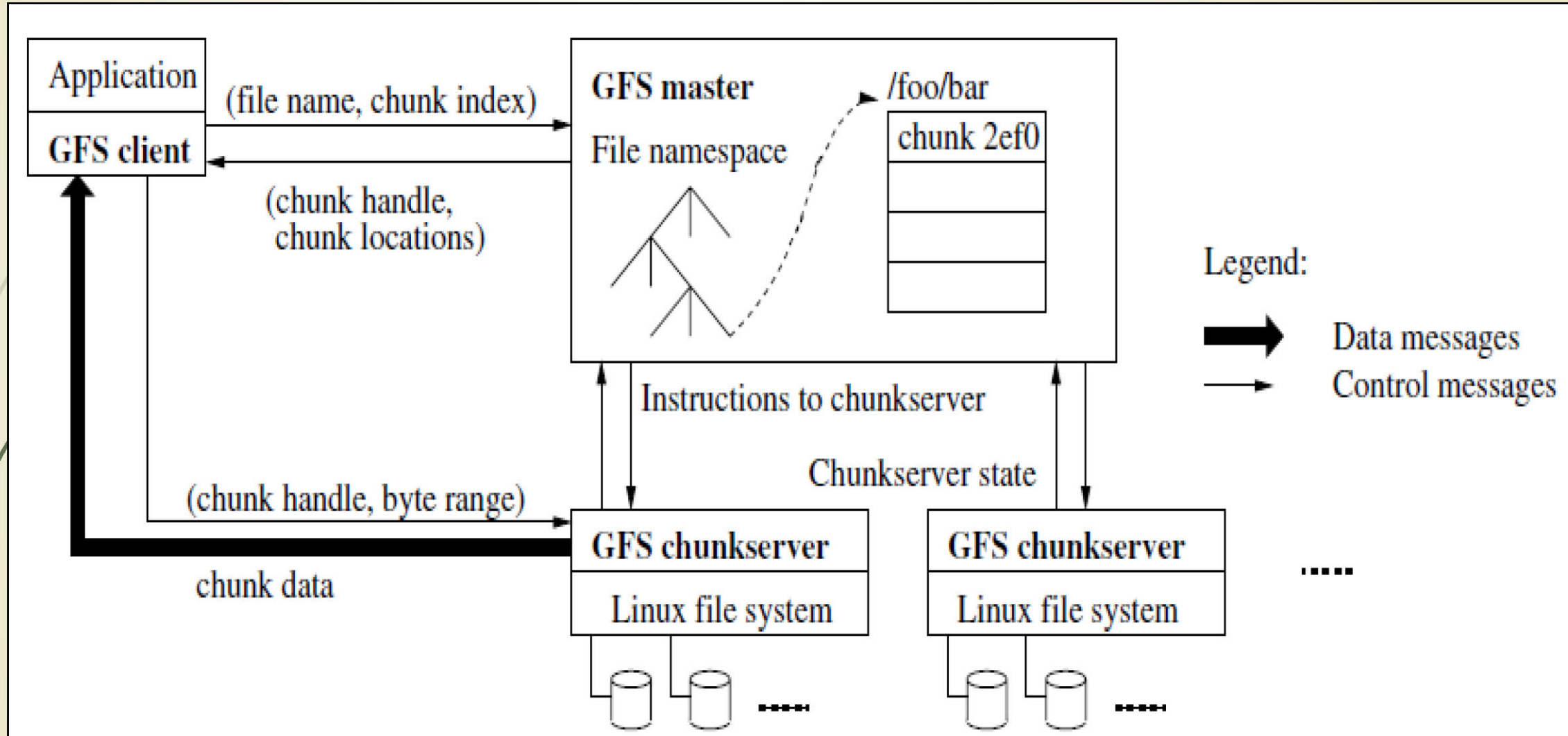


Figure 1: GFS Architecture

GFS Design Overview: Architecture

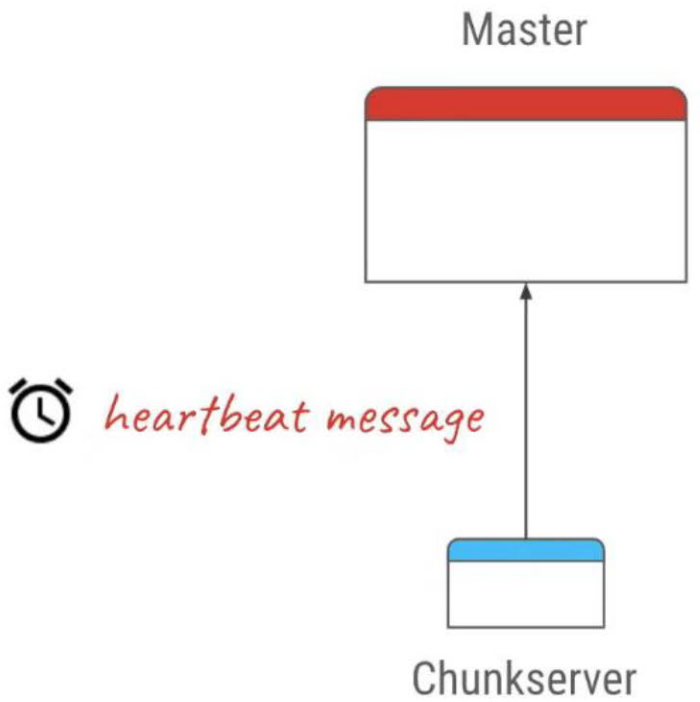
- A GFS cluster consists of ***a single master*** and ***multiple chunkservers*** and is accessed by ***multiple clients.***
- Each of these is typically a commodity Linux machine.
- Files are divided ***into fixed-size chunks***[64 MB by default but can be customized].
- Each ***chunk is identified*** by an immutable and ***globally unique 64-bit chunk*** handle assigned by the master at the time of chunk creation.
- ***Chunkservers store chunks*** on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range.

GFS Design Overview: Architecture

- For reliability, ***each chunk is replicated*** on multiple chunkservers.
- ***By default***, ***three replicas are stored***, though users can designate different replication levels for different regions of the file namespace.
- ***The Master Node:*** ***The master maintains all file system metadata***
  - This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks.
  - It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers.
  - The master ***periodically communicates*** with each chunkserver in ***HeartBeat*** messages to give it ***instructions and collect its state***.

Heartbeats

Regular heartbeats to ensure chunkservers are alive



GFS Design Overview: Architecture

- ***GFS Client***
  - GFS client code linked into each application ***implements the file system API*** and ***communicates with the master*** and ***chunkservers*** to ***read or write data on behalf of the application***.
  - ***Clients interact with the master for metadata*** operations, but all data-bearing ***communication goes directly to the chunkservers.***

Why Single Master?

- The master now has ***global knowledge of the whole system***, which ***drastically simplifies the design***.
- ***But the master is (hopefully) never the bottleneck***
  - Clients ***never read and write file data through the master***; client only requests from master which chunkservers to talk to.
  - Master can ***also provide additional information about subsequent chunks*** to further reduce latency.
  - ***Further reads of the same chunk don't involve the master***, either.

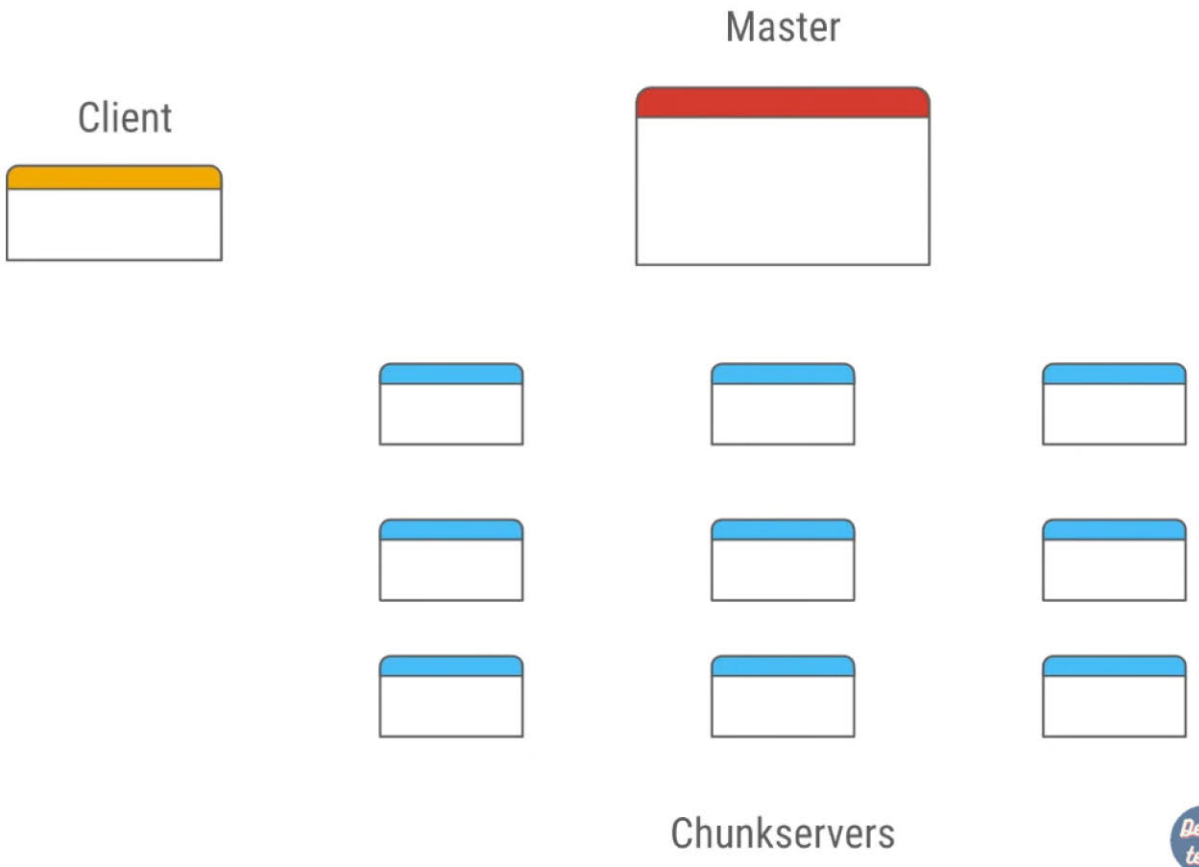
Why Single Master?

- Master state is also ***replicated for reliability on multiple machines***, using the ***operation log and checkpoints***
  - If ***master fails***, GFS can ***start a new master process at any of these replicas*** and modify DNS alias accordingly
  - ***"Shadow" masters also provide read-only access to the file system***, even when primary master is down
    - They ***read a replica of the operation log*** and apply the same sequence of changes
    - ***Not mirrors of master*** – they lag primary master by fractions of a second. This means we can **still read up-to-date file contents while master is in recovery**!

Single master for multi-TB cluster

Large chunk size

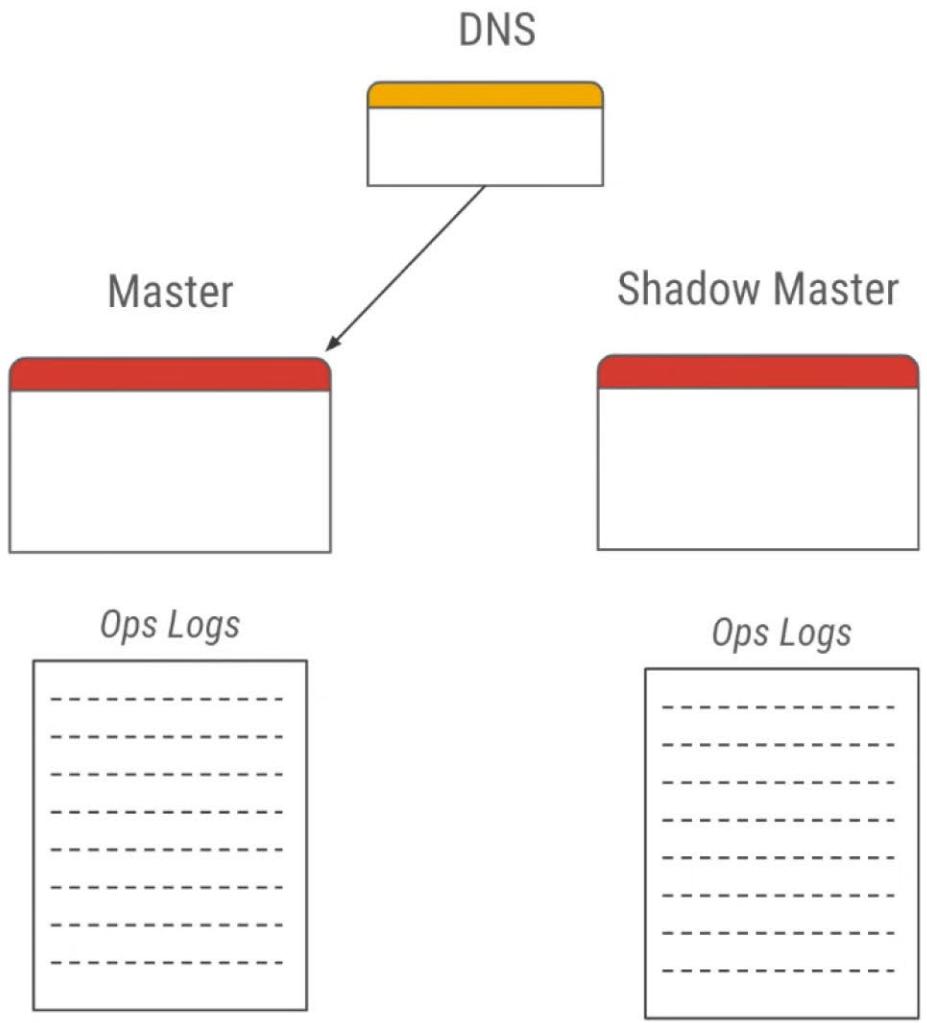
- 64MB chunk
- Reduced meta-data
- Reduces client interactions
- Client caches location data



Shadow Master

Single Point of Failure

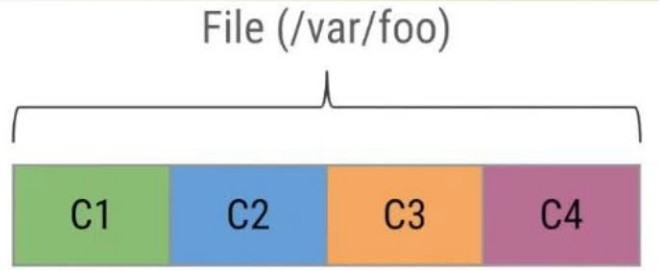
- Ops log is replicated remotely
- Shadow master uses the logs
- DNS change can change master
- Shadow master may lag slightly



GFS Design Overview: Chunk Size

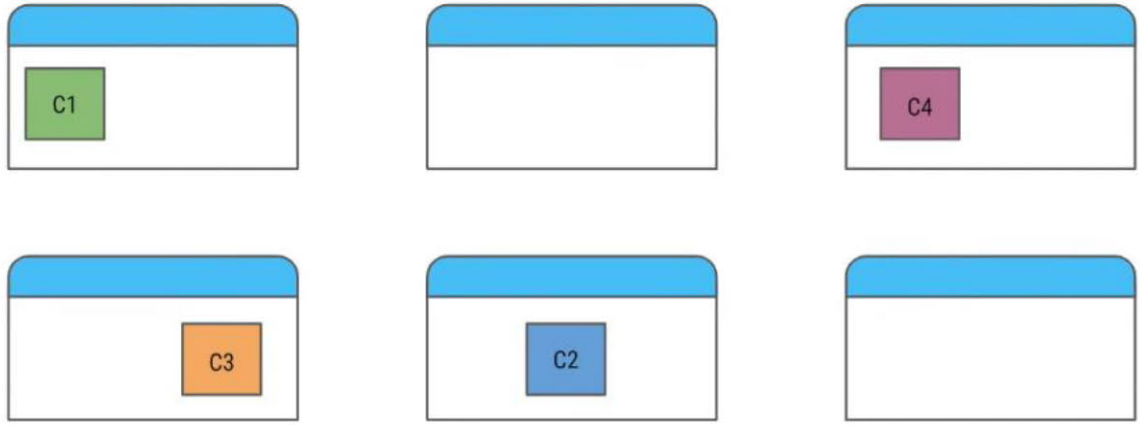
- ***Chunk size is 64 MB***, which is ***much larger*** than ***typical file*** system blocks sizes.
- ***Advantages***
  - It ***reduces clients' need to interact with the master*** because reads and writes on the ***same chunk require only one initial request*** to the master for chunk location information.
  - Since on a large chunk, ***a client is more likely to perform many operations on a given chunk***, it can ***reduce network overhead*** by keeping a persistent TCP connection to the chunkserver over an extended period of time.
  - It reduces the size of the metadata stored on the master.
- ***Disdvantages***
  - A ***small file consists of a small number of chunks***, perhaps just one.
  - The chunkservers storing those chunks may become hot spots if many clients are accessing the same file.

Chunks



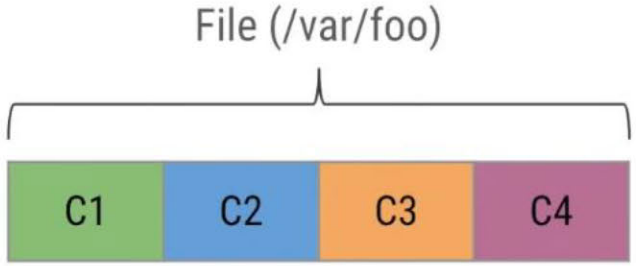
Files split into chunks

- Each chunk of 64MB
- Identified by 64 bit ID
- Stored in Chunkservers



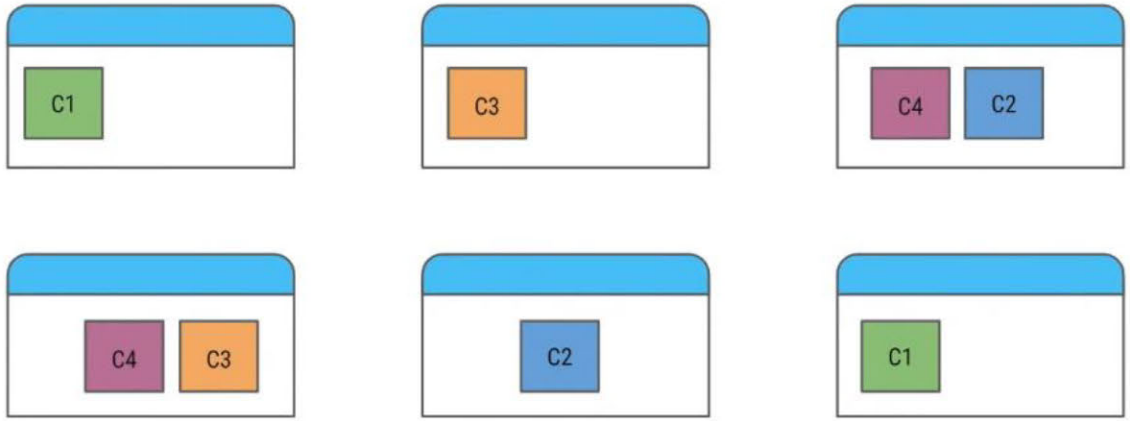
Chunkservers - Chunks of single file are distributed on multiple machines

Replicas



Files split into chunks

- Replica count by client
- commodity server failures



Replicas ensure durability of data if chunkserver goes down

GFS Design Overview: Metadata

- The master ***stores three major types of metadata***: the ***file and chunk namespaces***, the ***mapping from files to chunks***, and the ***locations of each chunk's replicas***.
- All metadata is kept in the master's memory.
- ***In-Memory Data Structures***
  - Since metadata is stored in memory, ***master operations are fast.***
  - Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background.
  - This periodic scanning is ***used to implement chunk garbage collection, re-replication in the presence of chunkserver failures***, and chunk migration to balance load and disk space usage across chunkservers.

GFS Design Overview: Metadata

- ***Chunk Locations***
  - The master ***does not keep a persistent record*** of which chunkservers have a ***replica of a given chunk***.
  - It simply polls ***chunkservers for that information at startup***.
  - The master can ***keep itself up-to-date thereafter*** because it controls all chunk placement and ***monitors chunkserver status*** with ***regular HeartBeat messages.***
- ***Operation Log***
  - The operation log ***contains a historical record of critical metadata changes***.
  - It is ***central to GFS***. Not only is it the only persistent record of metadata, but it also ***serves as a logical timeline that defines the order of concurrent operations***.
  - Since the operation log is critical, we must ***store it reliably and not make changes visible to clients*** until metadata changes are made persistent.
  - The ***master recovers*** its file system state ***by replaying the operation log.***

GFS Design Overview: Consistency Model

- The ***state of a file region*** after a data mutation depends on ***the type of mutation***, whether it ***succeeds or fails*** and whether ***there are concurrent mutations.***

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

- A ***file region is consistent*** if all clients will ***always see the same data***, regardless of ***which replicas they read from***.
- A ***region is defined*** after a file data mutation ***if it is consistent*** and clients ***will see what the mutation writes in its entirety.***

GFS Design Overview: Consistency Model

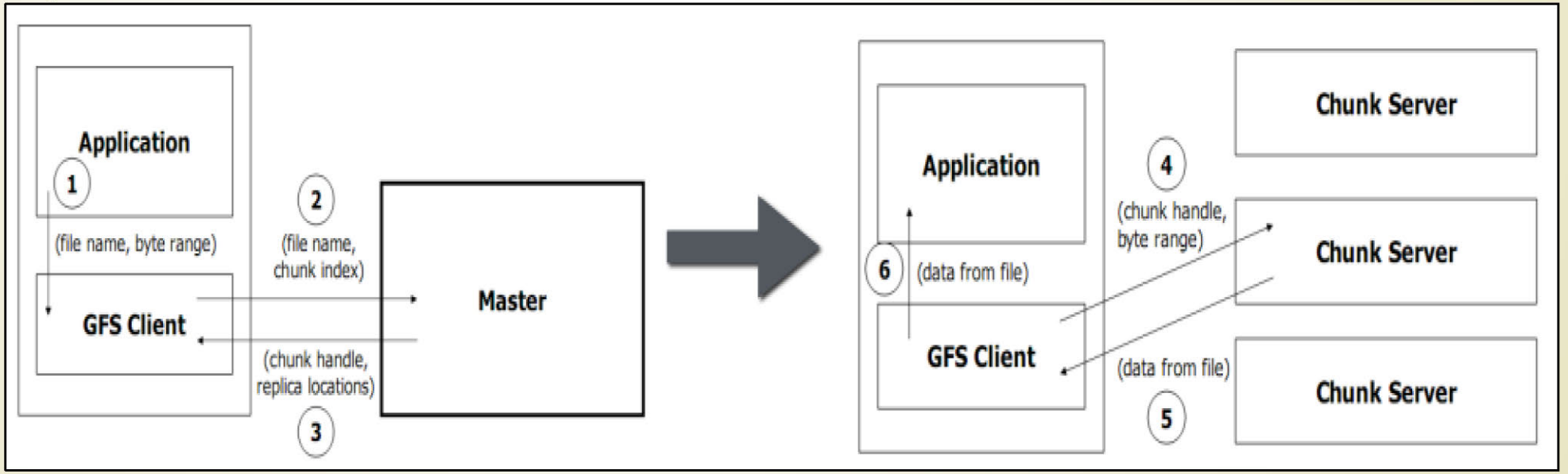
- When ***a mutation succeeds without interference*** from concurrent writers, the ***affected region is defined*** (and by implication consistent): ***all clients will always see what the mutation has written.***
- Concurrent ***successful mutations*** leave the region undefined but consistent: ***all clients see the same data***, but it may not reflect what any one mutation has written.
  - Typically, it consists of mingled fragments from multiple mutations.
- A ***failed mutation makes the region inconsistent*** (hence also undefined): different clients may see different data at different times.

GFS Design Overview: Consistency Model

- After ***a sequence of successful mutations***, the mutated file region is ***guaranteed to be defined and contain the data*** written by the last mutation.
- ***GFS achieves this by***
  - applying ***mutations to a chunk in the same order*** on all its replicas
  - using ***chunk version numbers to detect any replica*** that has become stale ***because it has missed mutations*** while its chunk server was down. ***Stale replicas will never be involved in a mutation*** or given to clients asking the master for chunk locations.
  - They are ***garbage collected at the earliest opportunity.***

GFS Design Overview: Read Operation

1. First, ***using the fixed chunk size***, the client translates the file name and byte offset specified by the application into a chunk index within the file.
2. Then, it ***sends the master a request*** containing the ***file name and chunk index.***
3. The ***master replies*** with the ***corresponding chunk*** handle and ***locations of the replicas.***



GFS Design Overview: Read Operation

4. The ***client caches this information*** using the file name and chunk index as the key.
5. The client ***then sends a request to one of the replicas***, most likely the ***closest one.***
6. The request ***specifies the chunk handle and a byte range within*** that chunk.
7. Further ***reads of the same chunk require no more client-master interaction until the cached information expires*** or the file is reopened.

- In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested.
- This extra information sidesteps several future client-master interactions at practically no extra cost.

GFS System Interactions: Leases and Mutation Order

- A ***mutation is an operation*** that changes the ***contents or metadata of a chunk*** such as a write or an append operation.
- ***Leases*** are used ***to maintain a consistent mutation order*** across replicas.
- The ***master grants a chunk lease to one of the replicas***, called ***primary.*** The primary ***picks a serial order*** for all mutations to the chunk. ***All replicas follow this order*** when applying mutations.
- The ***lease mechanism*** is designed ***to minimize management overhead*** at the master. ***A lease has an initial timeout of 60 seconds.***
- However, ***as long as the chunk is being mutated***, the primary can request and typically receive extensions from the master indefinitely.
- The master ***may sometimes try to revoke a lease before it expires*** (e.g., when the master wants to disable mutations on a file that is being renamed).

GFS Data Write Operation

1. Client ***asks the master which chunk server holds the current lease*** for the chunk and the locations of the other replicas. If ***no one has a lease***, the master ***grants one to a replica it chooses.***
2. Master ***replies with the identity of the primary and the locations of the other (secondary) replicas.*** The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.
3. The ***client pushes the data to all the replicas***. ***A client can do so in any order***. Each ***chunkserver will store the data in an internal LRU buffer*** cache until the data is used or aged out.

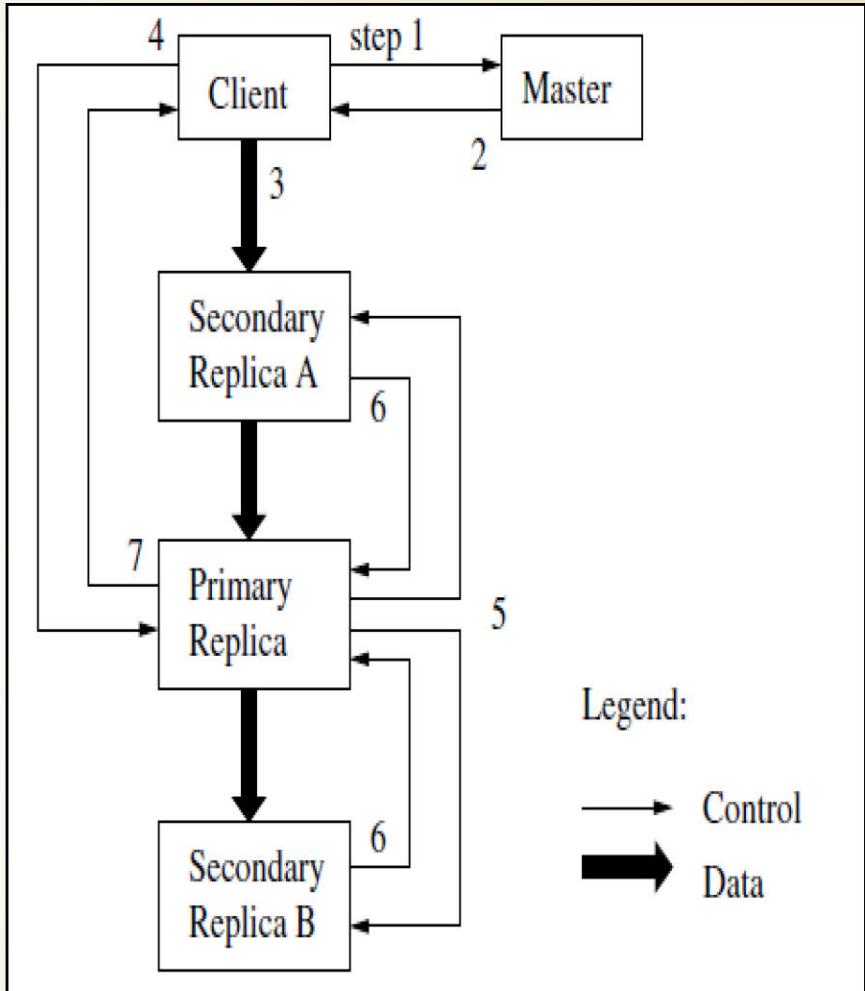
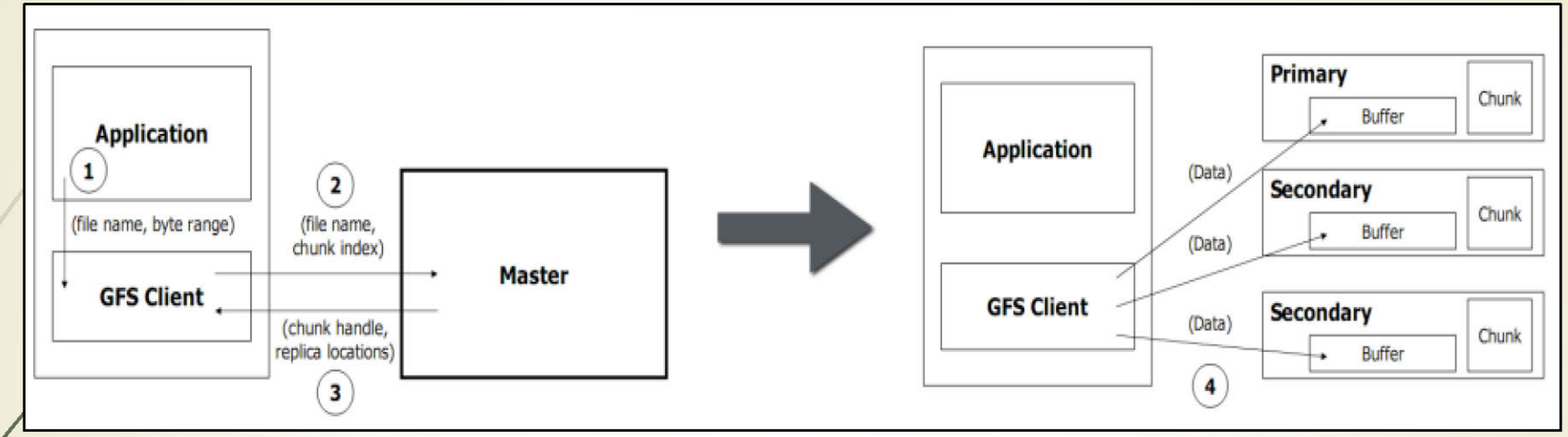


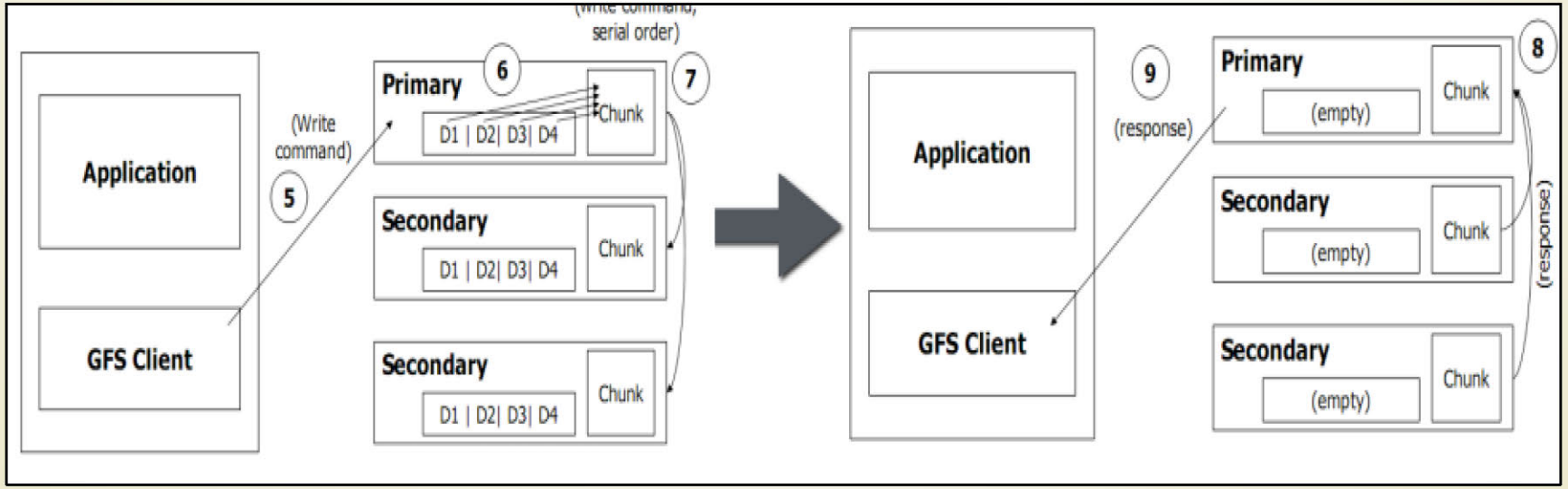
Figure 2: Write Control and Data Flow

GFS Data Write Operation

4. ***Once all the replicas have acknowledged*** receiving the data, the ***client sends a write request to the primary***. The request identifies the data pushed earlier to all of the replicas. The ***primary assigns consecutive serial numbers to all the mutations it receives***, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
5. The ***primary forwards the write request*** to all secondary replicas. Each secondary ***replica applies mutations in the same serial number*** order assigned by the primary.
6. The ***secondaries all reply to the primary*** indicating that they have completed the operation.
7. The ***primary replies to the client***. Any errors encountered at any of the replicas are reported to the client.

GFS Data Write Operation



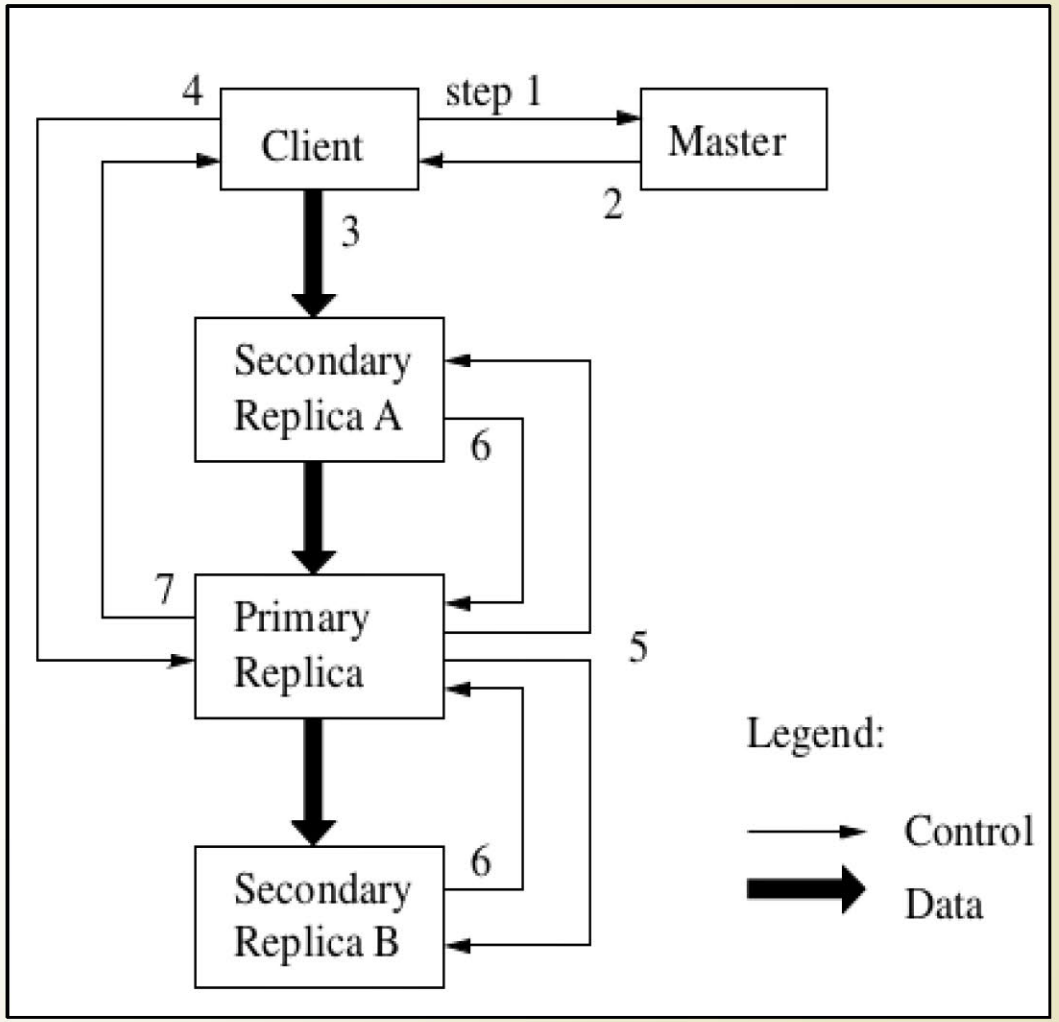


GFS Record Append Operation

- GFS provides ***an atomic append operation*** called ***record append.***
- ***In a traditional write***, the client ***specifies the offset*** at which data is to be written.
- ***Concurrent writes to the same region are not serializable***: the region may end up containing data fragments from multiple clients.
- ***In a record append***, however, the client specifies only the data.
  - ***GFS appends it to the file at least once atomically*** (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client.
- The ***client pushes the data to all replicas of the last chunk of the file.*** Then, it sends its request to the primary.

GFS Record Append Operation

- The ***primary checks to see*** if appending the record to the ***current chunk would cause the chunk to exceed*** the maximum size (64 MB).
- ***If so,***
  - It ***pads the chunk to the maximum size***,
  - ***Tells secondaries*** to do the same,
  - And ***replies to the client*** indicating that the operation should be retried on the next chunk.



GFS Record Append Operation

- If the **record fits within the maximum size**, which is the common case,
  - The **primary appends the data to its replica**,
  - Tells **the secondaries to write the data at the exact offset** where it has,
  - And **finally replies success to the client.**
- ***If a record append fails at any replica***, the client retries the operation.
- As a result, ***replicas of the same chunk may contain different data possibly*** including duplicates of the same record in whole or in part.
- ***GFS does not guarantee that all replicas are bytewise identical.*** It only ***guarantees that the data is written*** at least once as an atomic unit.
- ***For the operation to report success,*** the data ***must have*** been written ***at the same offset on all replicas*** of some chunk.

GFS Master Operation

- The ***master executes all namespace operations.***
- It ***manages chunk replicas*** throughout the system:
  - Makes placement decisions
  - Creates new chunks and hence replicas
  - Coordinates various system-wide activities
  - ***To keep chunks fully replicated***
    - To balance load across all the chunkservers
    - To reclaim unused storage

GFS Master Operation

- ***Namespace Management and Locking***
  - GFS logically represents its namespace as a lookup table mapping full pathnames to metadata.
  - Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.
  - Each master operation acquires a set of locks before it runs.
  - Locks are used over namespaces to ensure proper serialization
  - Read/Write locks
  - GFS logically represents its namespace as a lookup table mapping full pathnames to metadata
  - If a Master operation involves /d1/d2/.../dn/leaf,
    - It will acquire read-locks on the directory names /d1, /d1/d2, ..., /d1/d2/.../dn,
    - And either a read lock or a write lock on the full pathname /d1/d2/.../dn/leaf.
  - Leaf may be a file or directory depending on the operation

GFS Master Operation

- ***Replica Placement***
  - A GFS cluster typically has hundreds of chunkservers spread across many machine racks.
  - The chunk replica placement policy serves two purposes:
    - Maximize data reliability and availability, and
    - Maximize network bandwidth utilization.
  - For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth.
  - We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline.

GFS Master Operation

- ***Creation, Re-replication, Rebalancing***
  - Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.
  - When the master creates a chunk, it chooses where to place the initially empty replicas.
  - The master re-replicates a chunk as soon as the number of available replicas falls below a user-specified goal.
  - ***Factors considered for creating replicas***
    - Place new replicas on chunkserver with below average disk utilization
    - Limit the number of “recent” creation on chunkservers
    - Spread replicas of chunk across racks

GFS Garbage Collection

- Deletion logged by master
- file renamed to a hidden file, deletion timestamp kept
- Periodic scan of the master's file system namespace
  - hidden file older than 3 days are deleted from master's memory (no further connection between file and its chunk)
- Periodic scan of the master's chunk namespace
  - Orphaned chunks (not reachable from any file) are identified, their metadata deleted
- Heartbeat messages used to synchronize deletion between master/chunkserver

GFS Fault tolerance

- ***Fast Recovery:*** Master and chunkservers are designed to restart and restore state in few seconds
- ***Chunk Replication:*** across multiple machines, across multiple racks
- ***Master Mechanisms:***
  - Keep look of ***all changes made to metadata***
  - Periodic ***checkpoints of the log***
  - ***Log and checkpoints replicated*** on multiple machines
  - ***Master state is replicated*** on multiple machines
  - ***Shadow master for reading data*** if real master is down

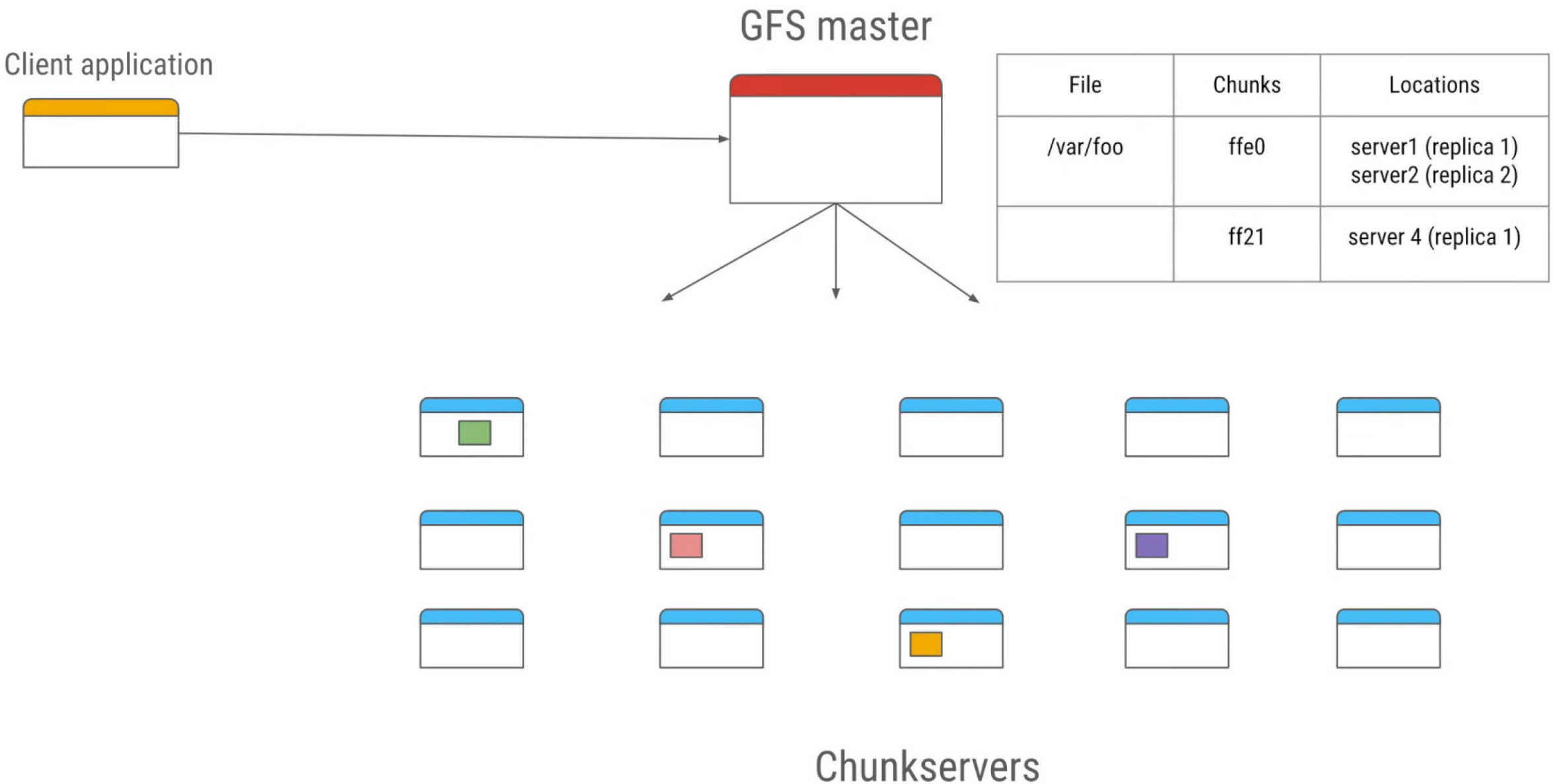
Conclusion

- ***GFS is a distributed file system*** that supports large-scale data processing workloads on commodity hardware
- ***GFS has different point in the design space***
  - Component failures as the norm
  - Optimize for huge files
- ***GFS provides fault tolerance***
  - Replicating data
  - Fast and automatic recovery
  - Chunk replication
- ***GFS has the simple, centralized master that does not become a bottleneck.***

Thank You
???

- **Reference**

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung "The Google File System", 2003


GFS master
Client application
File
Chunks
Locations
/var/foo
ffe0
server1 (replica 1)
server2 (replica 2)
ff21
server 4 (replica 1)
Chunkservers