Cloud and Cluster Data Management

CLOUD-SCALE FILE SYSTEMS

Google File System (GFS)

- Designing a file system for the Cloud
  - design assumptions
  - design choices
- Architecture
  - GFS Master
  - GFS Chunkservers
  - GFS Clients
- System operations and interactions
- Replication
  - fault tolerance
  - high availability

Key Application & Environment Observations

- Component failure is the norm
  - Application bugs, OS bugs, human errors, plus hw failures
  - => Need constant monitoring – error detection & recovery
- Files are huge
  - Files contain many small documents
  - Managing many small (kB) files is “unwieldy”
  - => I/O ops and block size need to be revisited
- Append-only Workload
  - Write-once, read-many (usually sequentially)
  - Such as: repositories for analysis, data streams, archive data, intermediate results
  - => Appending is the focus of performance optimization and atomicity guarantees, caching data blocks at client loses appeal

Kinds of Data Computations

- List of searches and their results
- Web crawl results
- Building an inverted index (list of pages where a word appears)
- Find all pages that link to a given page
- Intermediate results on page rank
- Spam pages for training

Design Assumptions

- System is built from many inexpensive commodity components
  - component failures happen on a routine basis
  - monitor itself to detect, tolerate, and recover from failures
- System stores a modest number of large files
  - a few million files, typically 100 MB or larger
  - multi-GB files are common and need to be managed efficiently
  - small files are to be supported but not optimized for
- System workload
  - **large streaming reads:** successive reads from one client read contiguous region, commonly 1 MB or more
  - **small random reads:** typically a few KB at some arbitrary offset (may be batched)
  - **large sequential writes:** append data to files; operation sizes similar to streaming reads; small arbitrary writes supported, but not efficiently
  - **small random writes:** supported, but not necessarily efficient

Design Assumption

- Support concurrent appends to the same file
  - efficient implementation
  - well-defined semantics
  - use case: producer-consumer queues or many-way merging, with hundreds of producers (one per machine) concurrently appending to a file
  - atomicity with minimal synchronization overhead is essential
  - file might be read later or simultaneously
- High sustained bandwidth is more important than low latency

Design Decisions: Interface

- GFS does not implement a standard API such as POSIX
- Supports standard file operations
  - **create/delete**
  - **open/close**
  - **read/write**
- Supports additional operations
  - **snapshot:** creates a copy of a file or a directory tree at low cost, using copy on write
  - **record append:** allows multiple clients to append data to the same file concurrently, while guaranteeing the atomicity of each individual client's append

Architecture

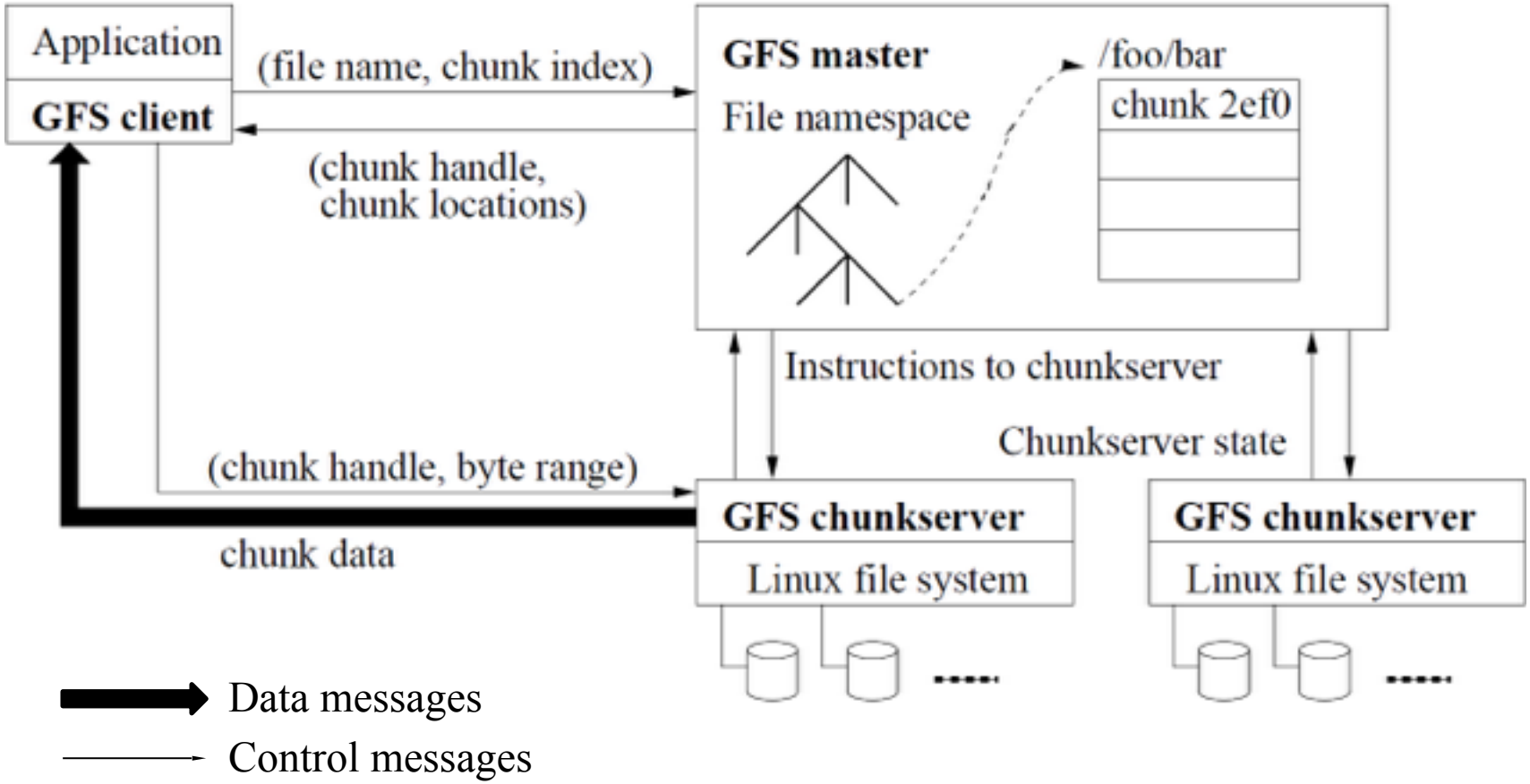
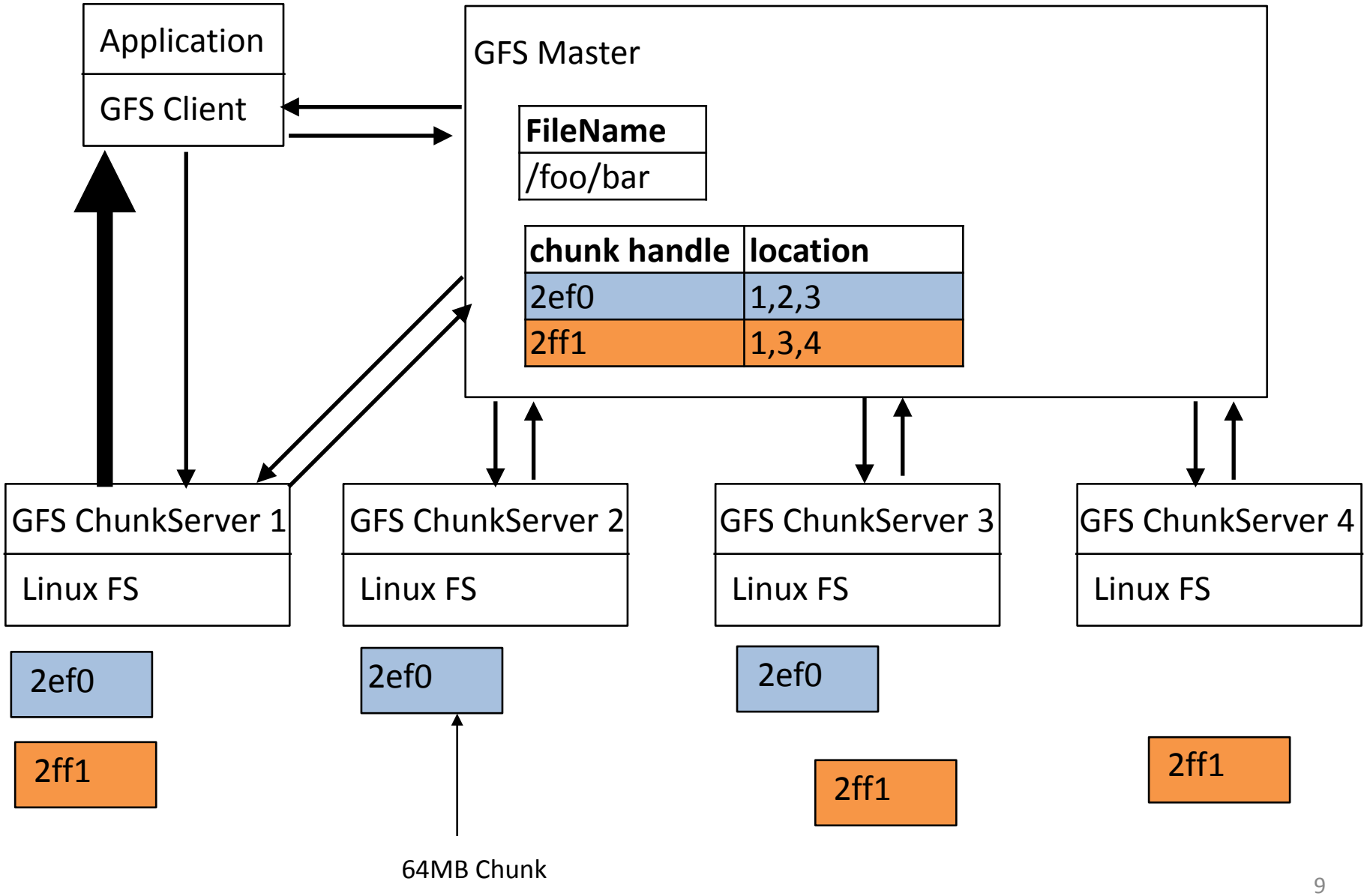


Figure Credit: "The Google File System" by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003


Application
GFS Client
GFS Master
FileName
/foo/bar
chunk handle
location
2ef0
1,2,3
2ff1
1,3,4
GFS ChunkServer 1
Linux FS
GFS ChunkServer 2
Linux FS
GFS ChunkServer 3
Linux FS
GFS ChunkServer 4
Linux FS
2ef0
2ff1
2ef0
2ff1
2ef0
2ff1
64MB Chunk

Design Decisions: Architecture

- GFS cluster
  - single master and multiple chunkservers
  - accessed by multiple clients
  - components are typically commodity Linux machines
  - chunkserver and client can run on same machine (lower reliability due to ‘flaky’ application code)
  - GFS server processes run in user mode
- Chunks
  - files are divided into fixed-size chunks
  - identified by globally unique chunk handle (64 bit), assigned by master
  - read/write based on chunk handle and byte range (no POSIX API)
  - chunks are replicated for reliability, typically the replication factor is 3

Design Decisions: Architecture

- Multiple chunkservers
  - store chunks on local disk as Linux files
  - accept and handle data requests
  - no special caching, relies on Linux's buffer cache
- Master
  - maintains all f/s met-data (namespace, acl, file->chunk mapping, …)
  - periodic *HeartBeat* messages to each chunkserver
  - clients communicate with master for metadata, but data requests go to chunkservers
- Caching
  - neither chunkserver nor client cache data
  - clients stream through files or have large working sets
  - eliminates complications of cache coherence

Design Decisions: Single Master

- Single master simplifies overall design
  - enables more sophisticated **chunk placement** and **replication**, but **single point of failure**
  - maintains **file system metadata**: namespace, access control information, file-to-chunk mapping, current chunk location
  - performs **management activities**: chunk leases, garbage collection, orphaned chunks, chunk migration
  - **heart beats**: periodic messages sent to chunkservers to give instructions or to collect state
- *Question: Issues with single master?*

Design Decisions: Single Master

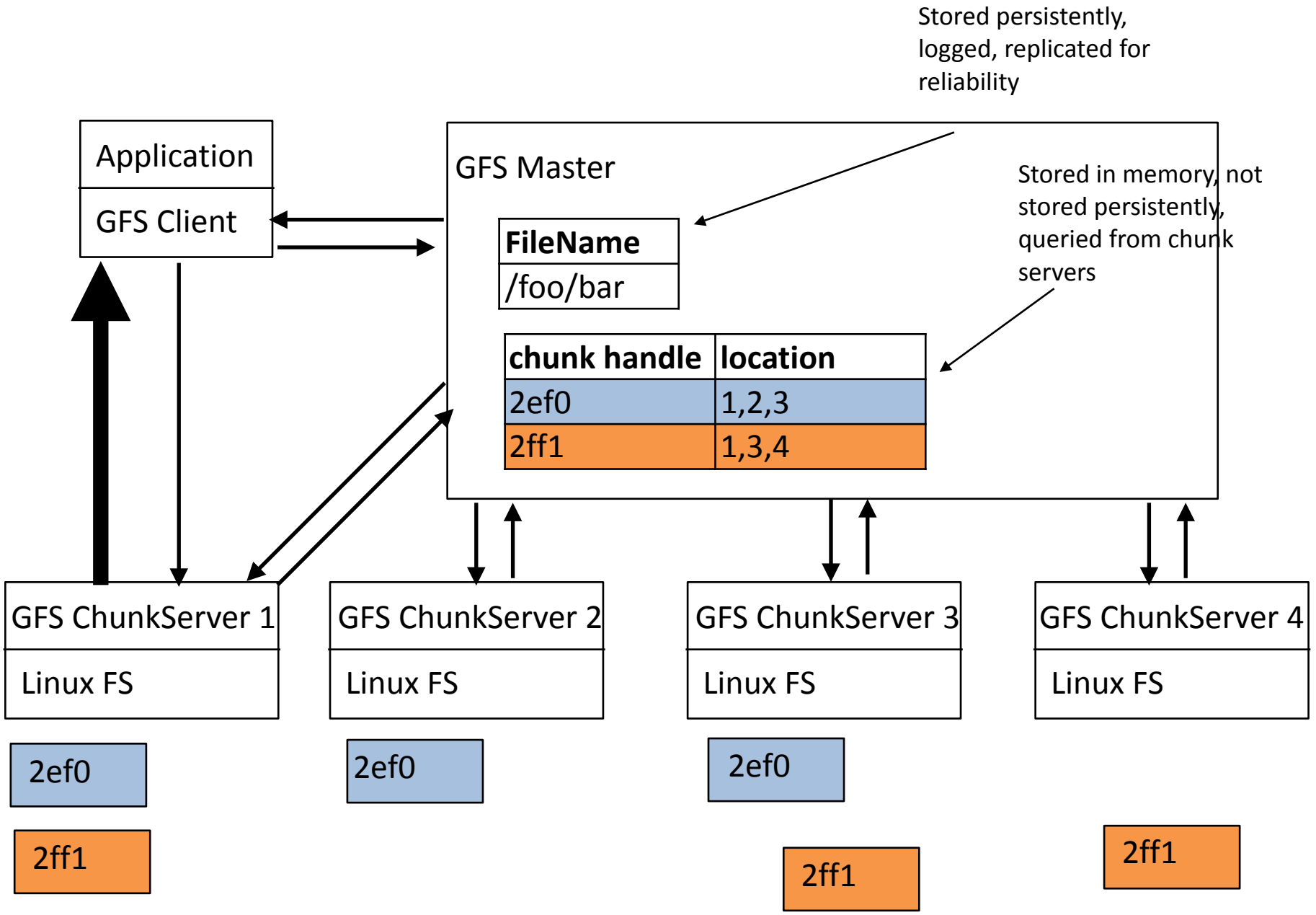
- Simple read example
  - client translates file/byte offsset -> chunk index (fixed chunk size)
  - master responds with handle/locations
  - data requests go to chunkservers (closest one, often)
  - multiple chunk requests at once - sidesteps client-master communication at minimal cost (round-trip is the overhead)

Design Decisions: Chunk Size

- One of the key parameters
  - set to a large value, i.e. 64 MB (larger than typical fs)
  - chunks stored as plain Linux files
  - to avoid fragmentation, chunkservers use lazy space allocation, i.e. files are only extended as needed
- Advantages
  - reduce interaction between client and master – one initial chunk info request (client can cache info for multi-TB working set)
  - reduce network overhead by using persistent TCP connection to do many operations on one chunk
  - reduce size of metadata stored on master
- Disadvantages
  - Internal fragmentation & small files consist of very few chunks
  - risk of hot spots → increase replication factor for small files, stagger start times

Design Decision: Metadata

- All metadata is kept in the master's main memory
  - **file and chunk namespaces:** lookup table with prefix compression
  - **file-to-chunk mapping**
  - **locations of chunk replicas:** not persistently stored, but queried from chunkservers
- Operation log
  - stored on master's local disc and replicated on remote machines
  - used to recover master in the event of a crash
- Discussion
  - size of master's main memory limits number of possible files
  - master maintains less than 64 bytes per chunk


Stored persistently, logged, replicated for reliability
Application
GFS Client
GFS Master
Stored in memory, not stored persistently, queried from chunk servers
FileName
/foo/bar
chunk handle
location
2ef0
1,2,3
2ff1
1,3,4
GFS ChunkServer 1
Linux FS
GFS ChunkServer 2
Linux FS
GFS ChunkServer 3
Linux FS
GFS ChunkServer 4
Linux FS
2ef0
2ef0
2ef0
2ff1
2ff1
2ff1

Design Decisions: Consistency Model

- Relaxed consistency model
  - tailored to Google's highly distributed applications
  - simple and efficient to implement
- File namespace mutations are **atomic**
  - handled exclusively by the master
  - namespace locking guarantees atomicity and correctness
  - master's operation log defines global total order of operations
- State of file region after data mutation
  - **consistent:** all clients always see the same data, regardless of the replica they read from
  - **defined:** consistent, plus all clients see the entire data mutation
  - **undefined but consistent:** result of concurrent successful mutations; all clients see the same data, but it may not reflect any one mutation
  - **inconsistent:** result of a failed mutation

Design Decisions: Consistency Model

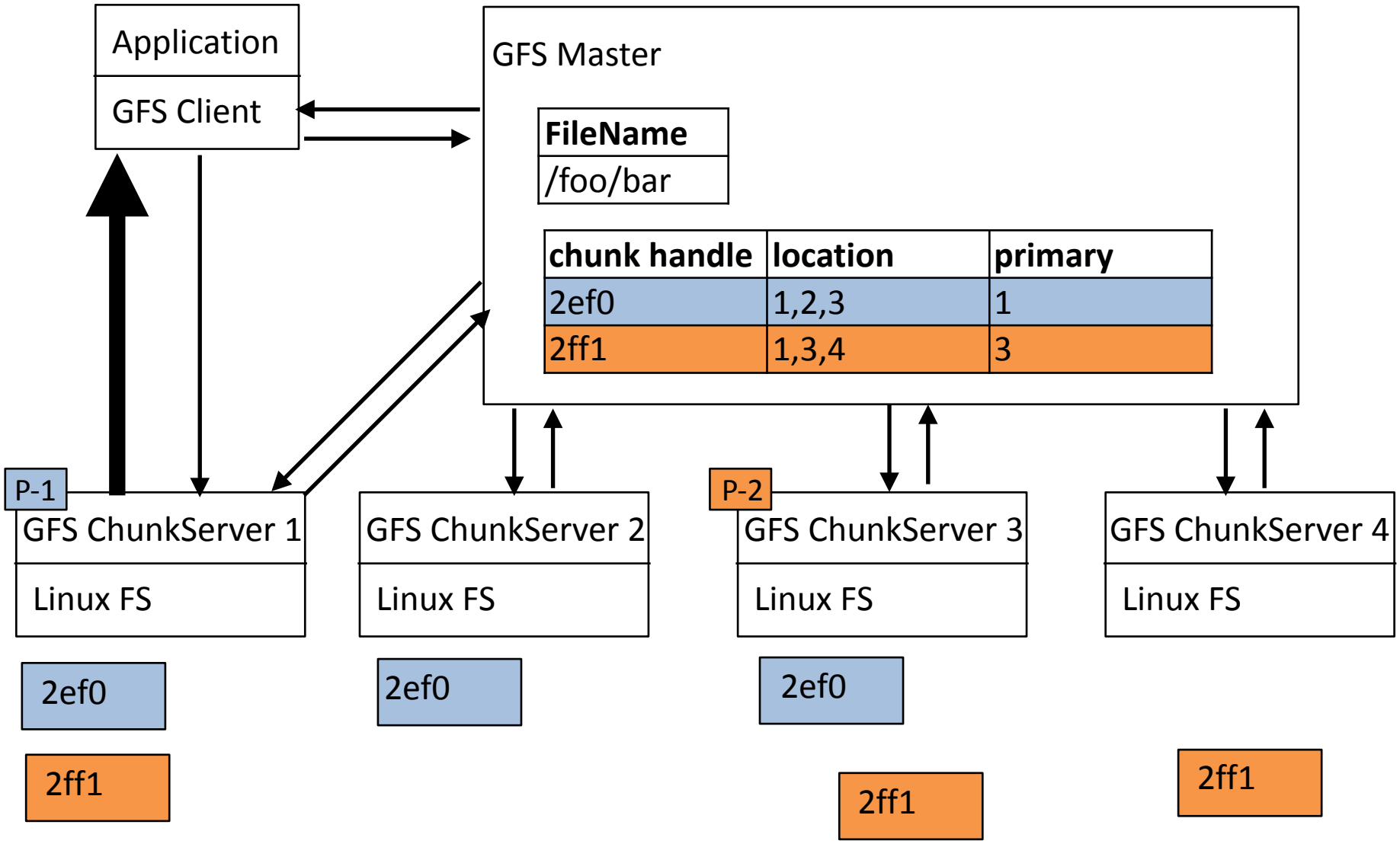
- Write data mutation
  - data is written at an application-specific file offset
- Record append data mutation
  - data ("the record") is appended *atomically at least once* even in the presence of concurrent mutations
  - GFS chooses the offset and returns it to the client
  - GFS may insert padding or record duplicates in between
- Stale replicas
  - not involved in mutations or given to clients requesting info
  - garbage collected as soon as possible
  - clients may read stale data – limited by cache entry timeout and next file open (purges chunk info from cache)
  - append only means typically premature not outdated data
  - also handles data corruptions due to hw/sw failure

Design Decisions: Concurrency Model

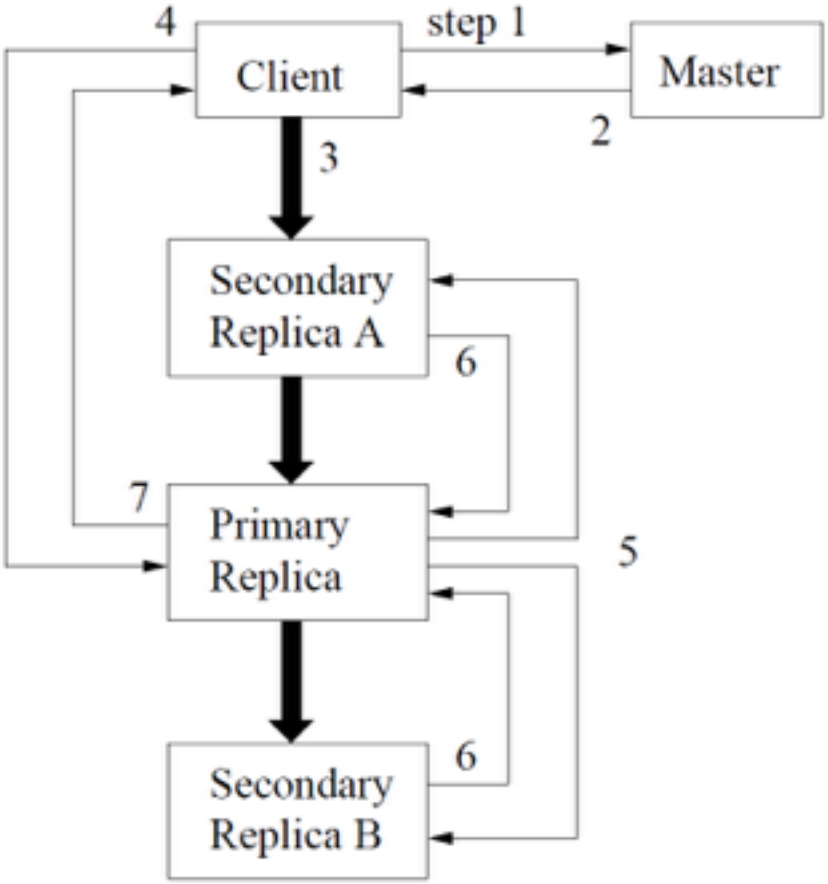
- Implications for applications
  - rely on appends rather than overwrites
  - checkpointing
  - application-level checksums
  - writing self-validating, self-identifying records
- Typical use cases (or “hacking around relaxed consistency”)
  - writer generates file from beginning to end and then atomically renames it to a permanent name under which it is accessed
  - writer inserts periodical checkpoints, readers only read up to checkpoint
  - many writers concurrently append to file to merge results, reader skip occasional padding and repetition using checksums
  - applications can use unique identifiers to deal with duplicates

Operations: Leases and Mutation Order

- Chunk lease granted to one replica – **primary**
  - primary picks serial order for mutations
  - replicas follow this order
  - global mutation order: lease grant order, serial order assigned by primary
- Designed to minimize management overhead
  - lease timeout – 60 seconds, but extensions can be requested and are typically approved
  - extension requests and grants piggybacked on *HeartBeat* messages
  - if master loses communication with primary, grant new lease to new replica after old lease expires


Application
GFS Client
GFS Master
FileName
/foo/bar
chunk handle
location
primary
2ef0
1,2,3
1
2ff1
1,3,4
3
P-1
P-2
GFS ChunkServer 1
Linux FS
GFS ChunkServer 2
Linux FS
GFS ChunkServer 3
Linux FS
GFS ChunkServer 4
Linux FS
2ef0
2ef0
2ef0
2ff1
2ff1
2ff1

Operations: Writing Files



- client ↔ master (1, 2)
  - chunkserver with chunk lease
  - chunkservers with replicas
- client ⟶ chunkservers
  - push data to chunkservers (3)
  - write request to primary (4)
- primary ⟶ secondary
  - forward write request (5)
- secondary ⟶ primary
  - operation status (6)
- primary ⟶ client
  - operation status

Figure Credit: "The Google File System" by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003

Operations: Data Flow

- Data flow decoupled from control flow
- Control flows client -> primary -> secondary
- Data is pushed(linearly) along a selected chain of chunkservers (pipelined)
- Utilize each machine's network bandwidth
  - Chain of chunk servers (not a tree), use outbound bandwidth
- Avoid network bottlenecks and high-latency links
  - each machine forwards to "closest" machine
- Minimize the latency to push the data
  - data pipelined over TCP connections (start forwarding as soon as you receive data)

Operations: Atomic Record Appends

- Atomic append – “Record Append”
- Traditional append: client specifies offset
  - concurrent writes to same region are not serializable
- Record append – client specifies only the data
  - GFS appends *at least once* atomically
  - GFS chooses offsets
- Avoids complicated & expensive synchronization
  - i.e. distributed lock manager
- Typical workload: multiple producer/single consumer

Operations: Atomic Record Appends II

- Atomic append – “Record Append”
  - Traditional append: client specifies offset – concurrent writes to same region are not serializable
- Follows previous control flow with only little extra logic
  - client pushes data to all replicas **of the last chunk of the file** (3’)
  - client sends its request to the primary replica (4)
- Additionally, primary checks if appending the record to the chunk exceeds the maximum chunk size (64 MB)
  - **yes:** primary and secondary pad the chunk to the maximum size (fragmentation) and append is tried on next chunk
    - record appends restricted to ¼ of maximum chunk size to keep worst-case fragmentation low
  - **no:** primary appends data to its replica and instructs secondaries to write data at the exact same offset

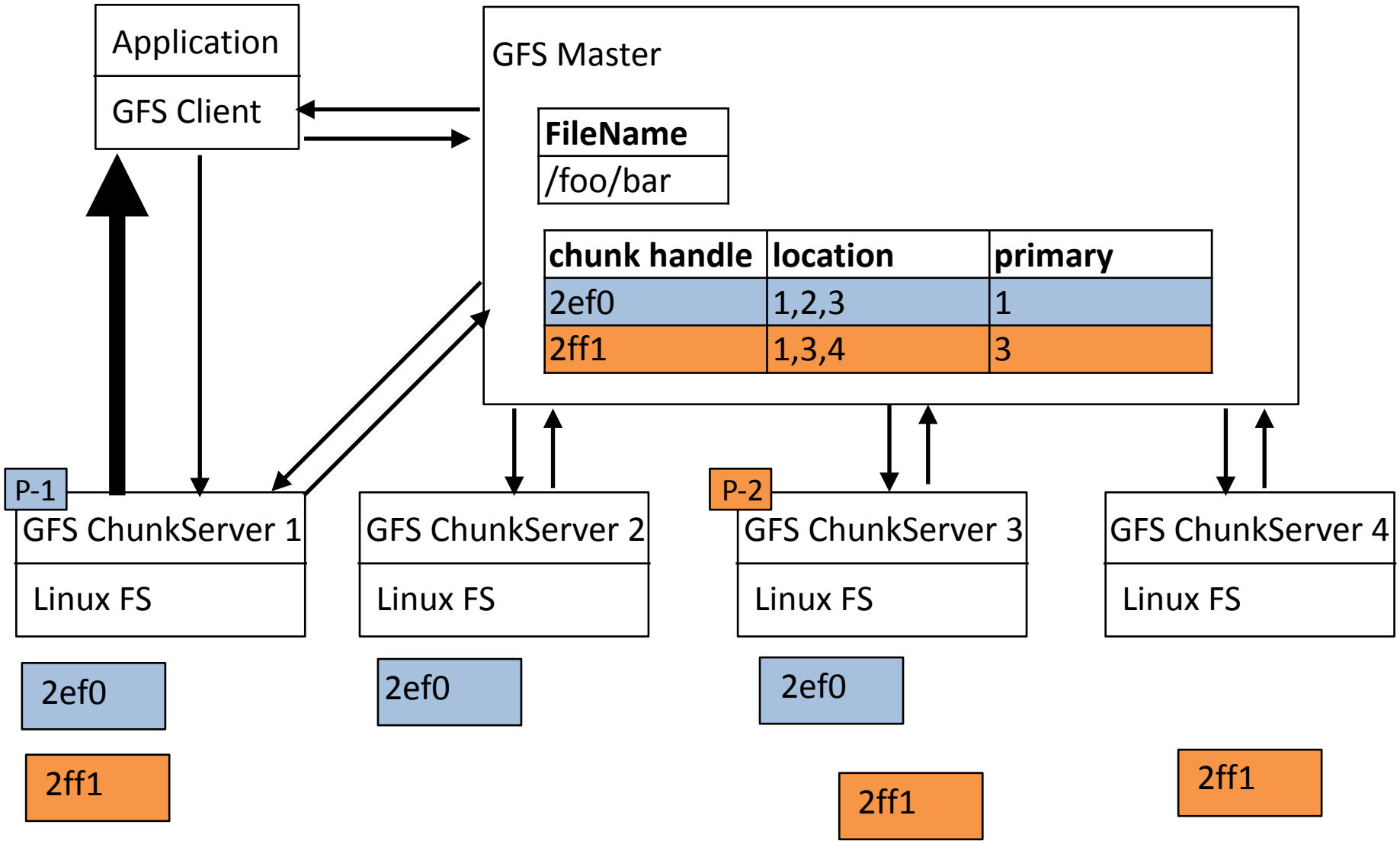
Operations: Append Failures

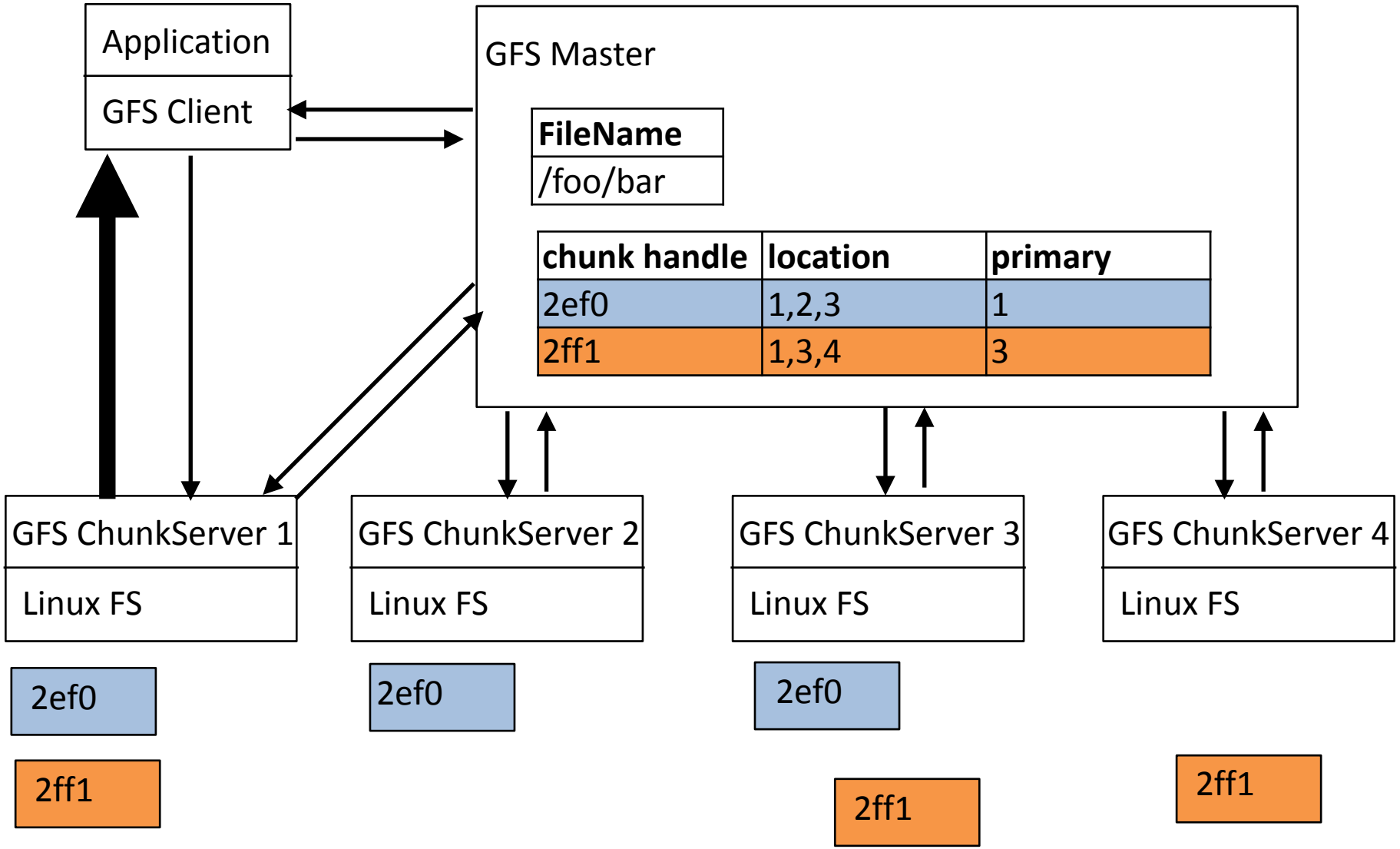
- Record append fails at any replica => client retries operation
  - => replicas of same chunk may contain different data, may include duplicates (whole or in part)
- GFS does not guarantee that all replicas are bytewise identical; guarantees data is written at least once as an atomic unit
- Operation reports success if data is written at same offset on all replicas
- Replicas are at least as long as the end of record, higher offsets for later appends

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

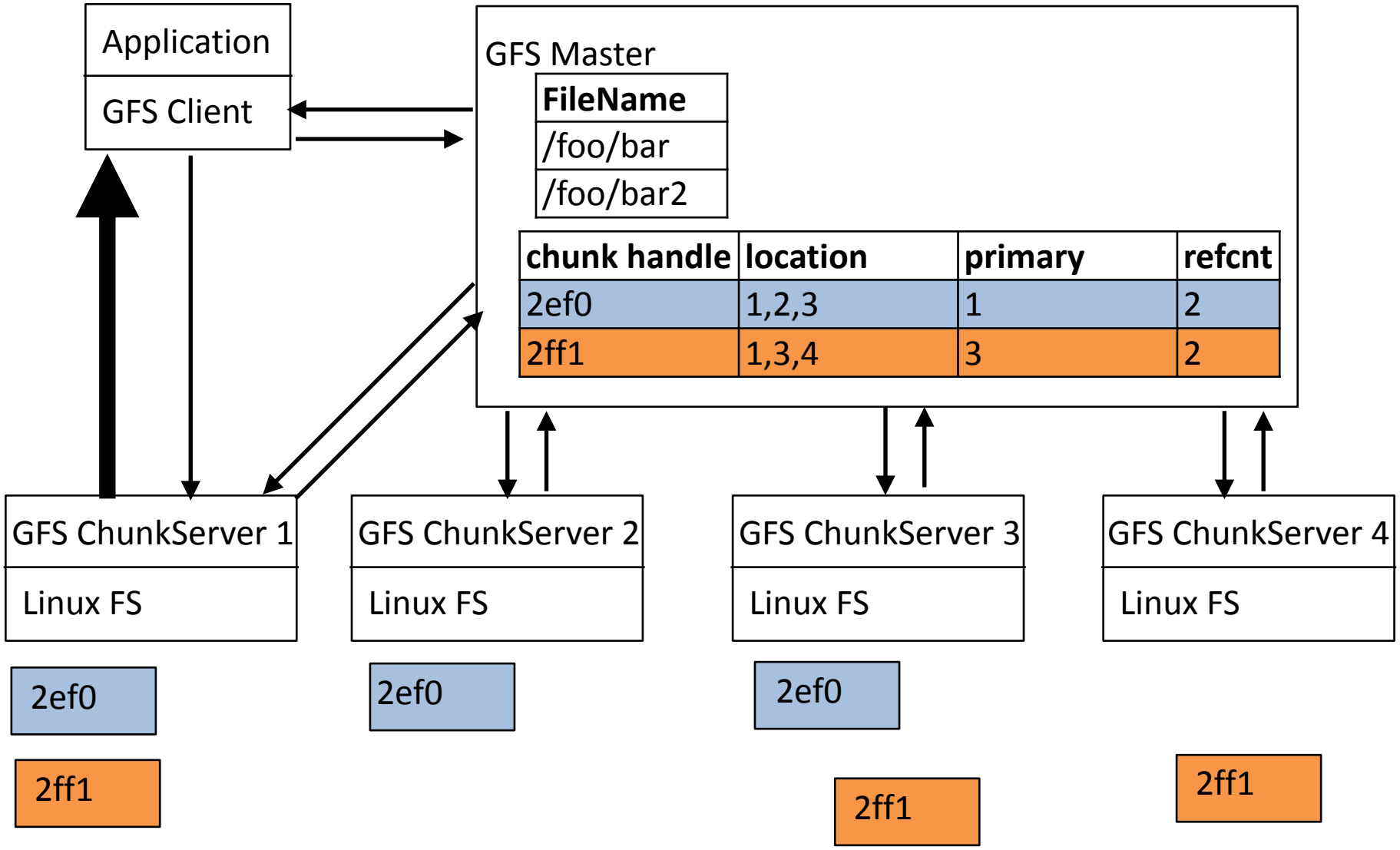
Operations: Snapshots

- Copies a file or directory (almost) instantaneously and with minimal interruption to ongoing mutations
  - quickly create branch copies of huge data sets
  - checkpointing the current state before experimenting
- Lazy copy-on-write approach
  - upon snapshot request, master first revokes any outstanding leases on the chunks of the files it is about to snapshot
  - subsequent writes will require interaction with master giving master chance to create new copy of chunk
  - after leases are revoked or have expired, operation is logged to disk
  - in-memory state is updated by duplicating metadata of source file or directory tree
  - reference counts of all chunks in the snapshot are incremented by one


Application
GFS Client
GFS Master
FileName
/foo/bar
chunk handle
location
primary
2ef0
1,2,3
1
2ff1
1,3,4
3
P-1
P-2
GFS ChunkServer 1
Linux FS
GFS ChunkServer 2
Linux FS
GFS ChunkServer 3
Linux FS
GFS ChunkServer 4
Linux FS
2ef0
2ef0
2ef0
2ff1
2ff1
2ff1



Snapshot Step 1: Revoke Leases…



Snapshot Step 2: Duplicate Meta-data and Increase Chunk Reference Count

Operations: Snapshots II

- Write request after snapshot (copy on write)
- Client sends request to master to write chunk C
- Master notices reference count on chunk C > 1 and defers reply to client
- Master picks new chunk handle; asks each chunkserver with a current replica of C to create a copy C’
- C’ created on chunkservers that have C – local data copying
- After creation of C’, proceed as normal

Master Operation

- Executes all namespace operations
- Manages chunk replicas
- Makes placement decisions
- Creates new chunks and replicas
- Coordination:
  - keep chunks replicated
  - balance load
  - reclaim unused storage

Namespace Management and Locking

- Differences to traditional file systems
  - no per-directory structures that list files in a directory
  - no support for file or directory aliases, e.g. soft and hard links in Unix
- Namespace implemented as a “flat” lookup table
  - full path name ⟶ metadata
  - prefix compression for efficient in-memory representation
  - each “node in the namespace tree” (absolute file or directory path) is associated with a read/write lock
- Each master operation needs to acquire locks before it can run
  - **read locks** on all “parent nodes”
  - **read** or **write lock** on “node” itself
  - file creation does not require a write lock on “parent directory” as there is no such structure
  - note metadata records have locks, whereas data chunks have leases

Replica Placement

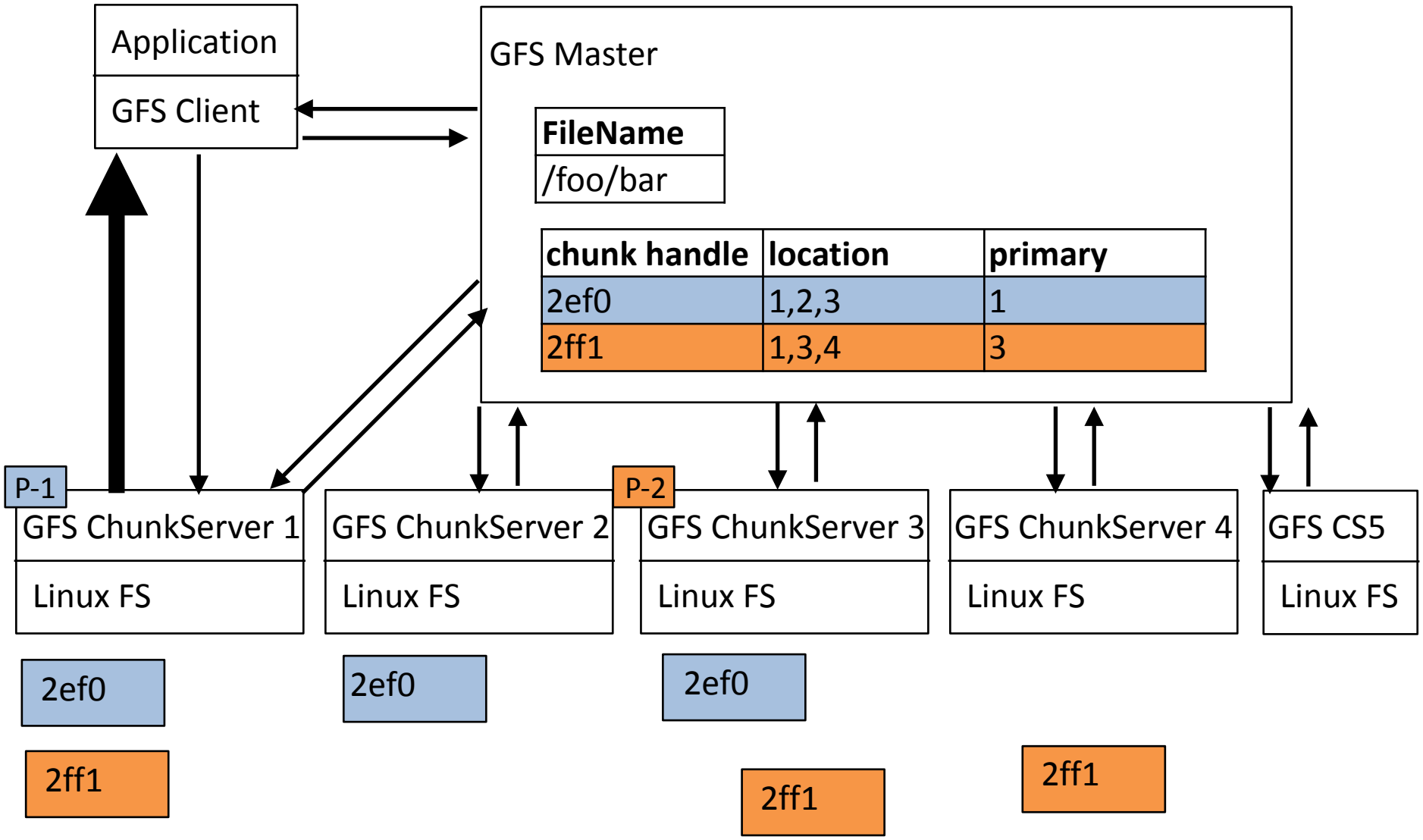
- Goals of placement policy
  - Maximize **data reliability** and **availability**
  - maximize **network bandwidth utilization**
- Background: GFS clusters are highly distributed
  - 100s of chunkservers across many racks
  - accessed from 100s of clients from the same or different racks
  - traffic between machines on different racks may cross many switches
  - in/out bandwidth of rack typically lower than within rack
- Possible solution: spread chunks across machines **and** racks
  - not enough to spread across machines, need to spread across racks also
  - failure or shared rack resource (network, power switch)
  - reads exploit aggregate bandwidth of multiple racks, writes flow through multiple racks

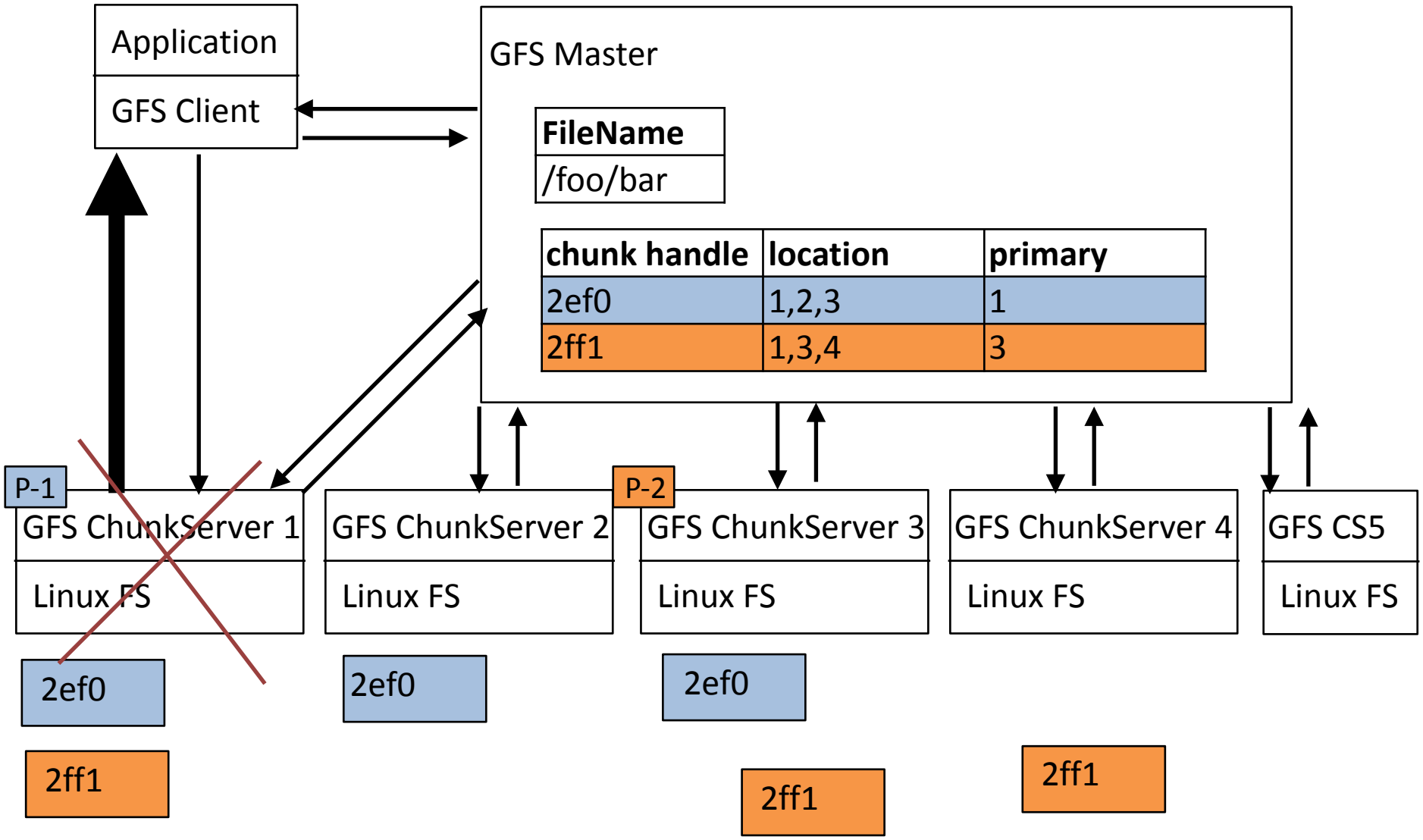
Replica Placement – Replica Creation

- Replicas created for three reasons
  - chunk creation
  - re-replication
  - rebalancing
- Chunk Creation: Selecting a chunkserver
  - place chunks on servers with below-average disk space utilization (goal is to equalize disk utilization)
  - place chunks on servers with low number of recent creations (creation => imminent heavy write traffic)
  - spread chunks across racks (see above)

Re-replication and Rebalancing

- Master triggers **re-replication** when replication factor drops below a user-specified goal
  - chunkservers becomes unavailable
  - replica is reported corrupted
  - a faulty disk is disabled
  - replication goal is increased
- Re-replication prioritizes chunks
  - how far chunk is from replication goal (i.e. lost two replicas vs. lost one)
  - prioritize chunks of live files
  - prioritize any chunk that is blocking client progress
- Limit number of active clone operations (cluster and chunkserver)
- Master **rebalances** replicas periodically
  - better disk space utilization
  - load balancing
  - gradually “fill up” new chunkservers (remove from below-avg free space)


Application
GFS Client
GFS Master
FileName
/foo/bar
chunk handle
location
primary
2ef0
1,2,3
1
2ff1
1,3,4
3
P-1
P-2
GFS ChunkServer 1
GFS ChunkServer 2
GFS ChunkServer 3
GFS ChunkServer 4
GFS CS5
Linux FS
Linux FS
Linux FS
Linux FS
Linux FS
2ef0
2ef0
2ef0
2ff1
2ff1
2ff1


Application
GFS Client
GFS Master
FileName
/foo/bar
chunk handle
location
primary
2ef0
1,2,3
1
2ff1
1,3,4
3
P-1
P-2
GFS ChunkServer 1
Linux FS
GFS ChunkServer 2
Linux FS
GFS ChunkServer 3
Linux FS
GFS ChunkServer 4
Linux FS
GFS CS5
Linux FS
2ef0
2ef0
2ef0
2ff1
2ff1
2ff1

Garbage Collection

- GFS does not immediately reclaim physical storage after a file is deleted
- Lazy garbage collection mechanism
  - master logs deletion immediately by renaming file to a “hidden name”
  - master removes any such hidden files during regular file system scan (3 day window to undelete)
- Orphaned chunks
  - chunks that are not reachable through any file
  - master identifies them in regular scan and deletes metadata
  - uses heartbeat messages to inform chunkservers about deletion (chunkservers send master list of chunks, master informs about deletes)

Garbage Collection II

- GC Discussion
  - easy to identify references (in file-chunk mappings on master)
  - easy to identify replicas (linux files in designated directories, not known to master -> garbage)
- GC Advantages vs. Eager Deletion
  - simple and reliable
  - ex. chunk creation succeeds on only some servers, can just let GC clean up replicas
  - ex. replica deletion message lost
  - Merges storage reclamation into master's normal background activities (scans of namespaces and handshakes with chunkservers)
  - Batches, amortized cost, done when master is relatively free
- Disadvantages
  - hinders user effort to fine tune usage when storage is tight
  - clients with lots of temp files (double delete)

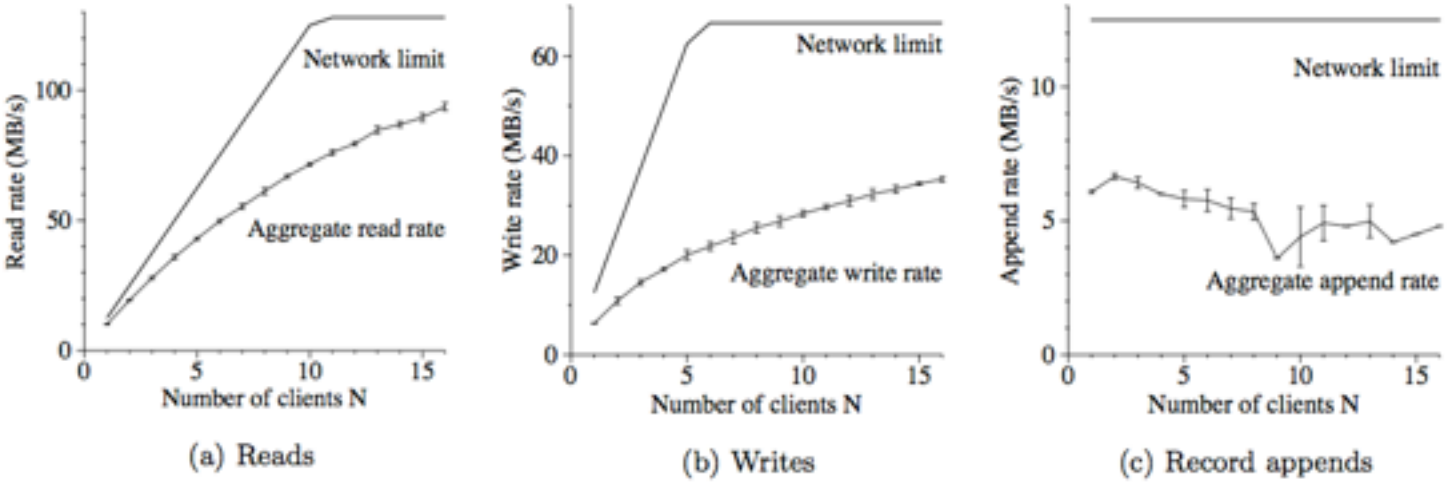
Stale Replicas

- Stale replicas
  - detected based on *chunk version number*
  - chunk version number is increased whenever master grants a lease
  - master and replicas record the new version number (in persistent state) before client is notified
  - recall leases required before writes
  - stale replicas detected on chunkserver restart – when sends its set of chunks to master
  - removed during regular garbage collection

Fault Tolerance

- High availability
  - **fast recovery:** master and chunkserver designed to recover in seconds; no distinction between normal and abnormal termination
  - **chunk replication:** different parts of the namespace can have different replication factors
  - **master replication:** operation log replicated for reliability; mutation is considered committed only once all replicas have written the update; “shadow masters” for read-only access
- Data integrity
  - chunkservers use checksums to detect data corruption
  - idle chunkservers scan and verify inactive chunks and report to master
  - each 64 KB block of a chunk has a corresponding 32 bit checksum
  - if a block does not match its check sum, client is instructed to read from different replica
  - checksums optimized for write appends, not overwrites

GFS Performance



Hadoop Distributed File System (HDFS)

- Open-source clone of GFS
  - similar assumptions
  - very similar design and architecture
- Differences
  - no support for random writes, append only
  - emphasizes platform independence (implemented in Java)
  - *possibly,* HDFS does not use a lookup table to manage namespace
  - terminology (see next bullet)
- "Grüezi, redet si Schwyzerdütsch?"
  - namenode → master
  - datanode → chunkserver
  - block → chunk
  - edit log → operation log

HDFS Architecture

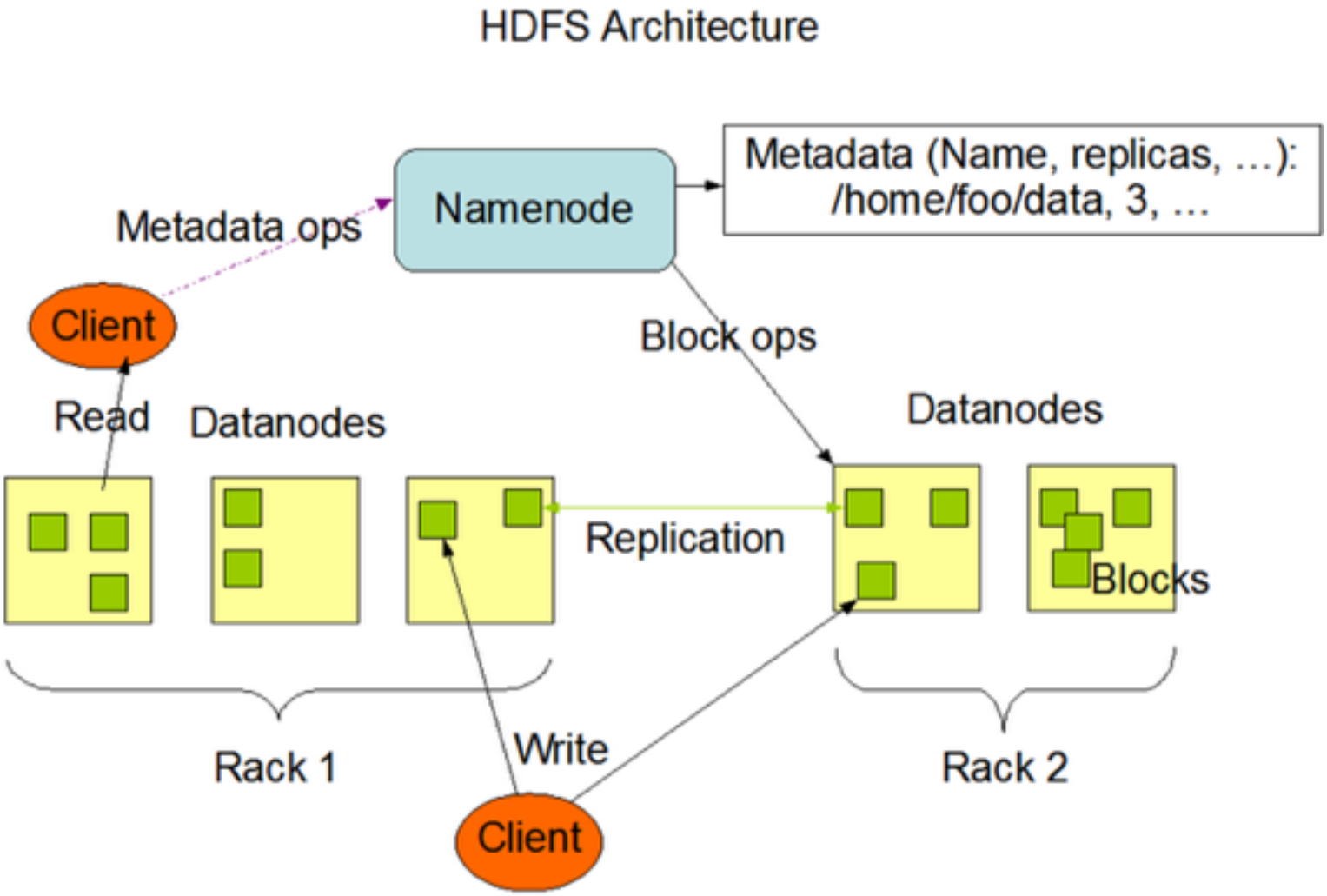


Figure Credit: "HDFS Architecture Guide" by D. Borthakur, 2008

Example Cluster Sizes

- GFS (2003)
  - 227 chunkservers
  - 180 TB available space, 155 TB used space
  - 737k files, 232k dead files, 1550k chunks
  - 60 MB metadata on master, 21 GB metadata on chunkservers
- HDFS (2010)
  - 3500 nodes
  - 60 million files, 63 million blocks, 2 million new files per day
  - 54k block replicas per datanode

  - all 25k nodes in HDFS clusters at Yahoo! provide 25 PB of storage

References

- S. Ghemawat, H. Gobioff, and S.-T. Leung: **The Google File System.** *Proc. Symp. on Operating Systems Principles (SOSP), pp. 29-43, 2003.*
- D. Borthakur: **HDFS Architecture Guide.** *2008.*
- K. Shvachko, H. Kuang, S. Radia, and R. Chansler: **The Hadoop Distributed File System.** *IEEE Symp. on Mass Storage Systems and Technologies, pp.1-10, 2010.*