The Google File System

大数据分析 | 何铁科

http: // hetieke.cn

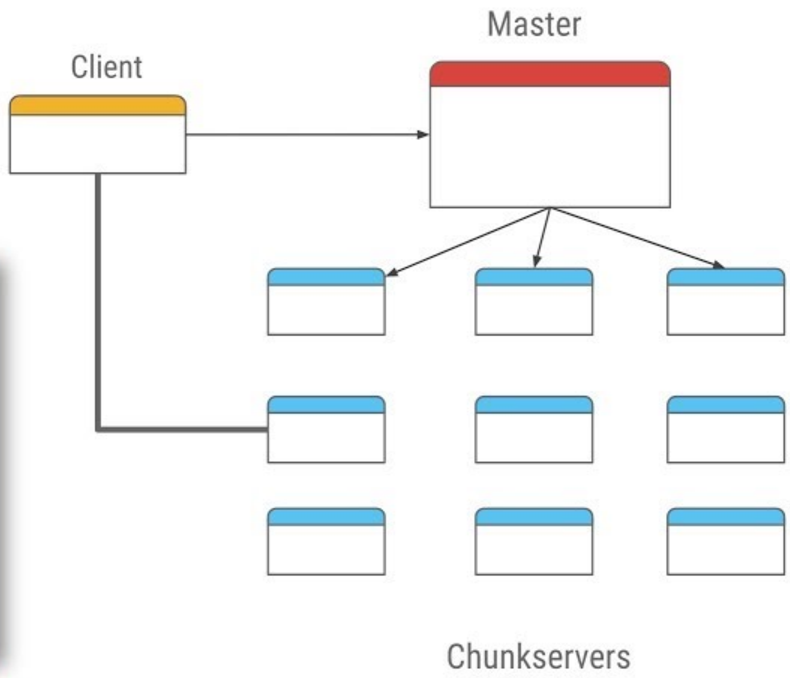
南京大學
NANJING UNIVERSITY

Goals of DFS

1. **Performance**
2. **Scalability**
3. **Reliability**
4. **Availability**

Google File System

Paper that inspired Hadoop



Motivation

- ❑ **Need for a scalable DFS**
- ❑ **Large distributed data-intensive applications**
- ❑ **High data processing needs**
- ❑ **Performance, Reliability, Scalability and Availability**
- ❑ **More than traditional DFS**

1st Component failure

1. norm rather than exception
2. hundreds or even thousands of storage machines
3. not functional
4. problems seen
5. monitoring, error checking, auto recovery

2nd Huge files

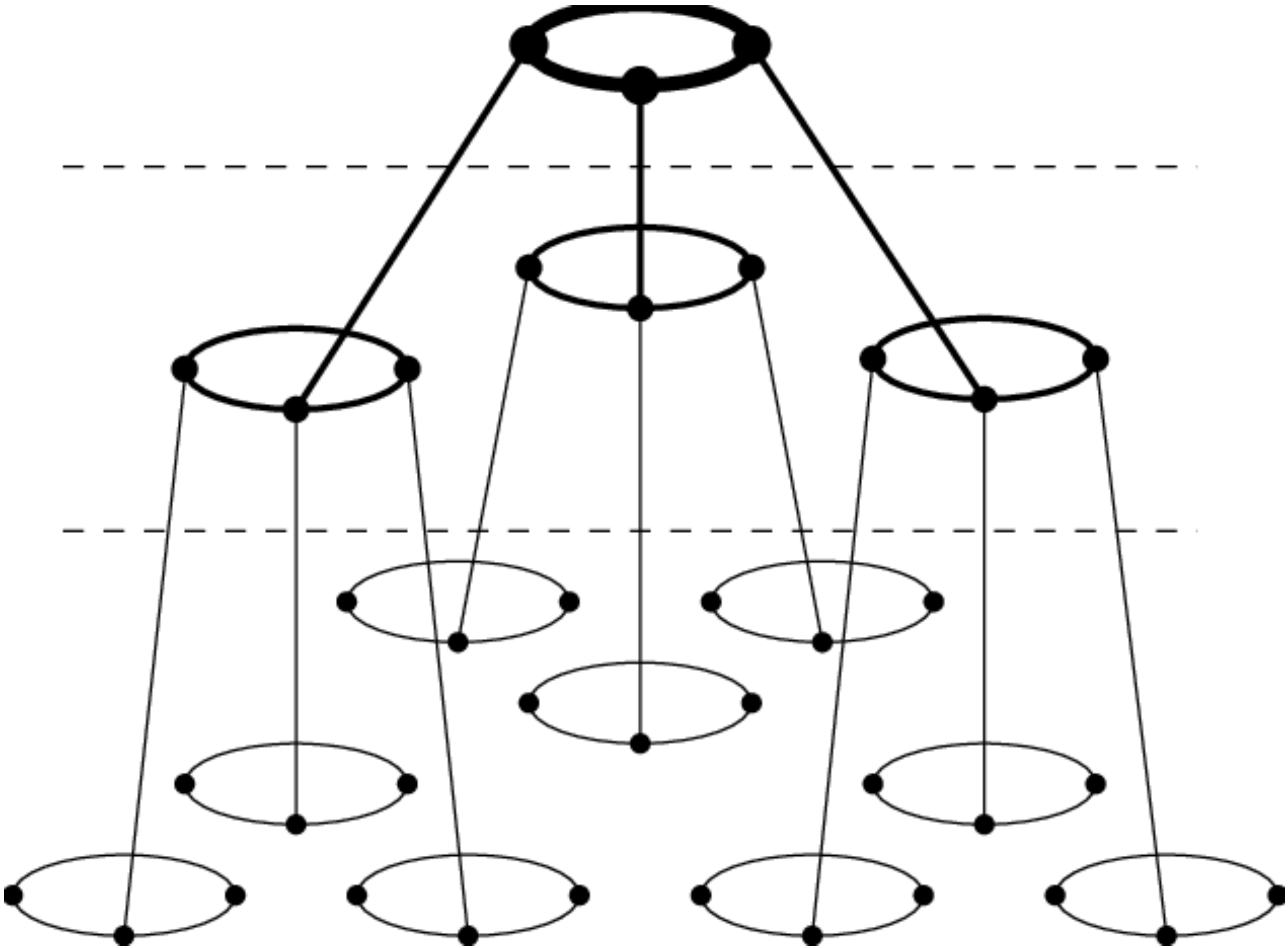
1. files are huge
2. many app objects
3. fast growing datasets
4. I/O and block sized revisited

3rd New data appended

1. files are mutated by appending new data
2. random writes? no
3. once written, only read
4. many data share these above characteristics
5. appending K.O the caching

4th New data appended

1. co-design the apps and file system API
2. relaxed consistency
3. atomic append operation



The Whole Design

1. **Assumptions**
2. **Interface**
3. **Architecture**
4. **Single Master**
5. **Chunk Size**
6. **Metadata**
7. **Consistency Model**

1. Assumptions

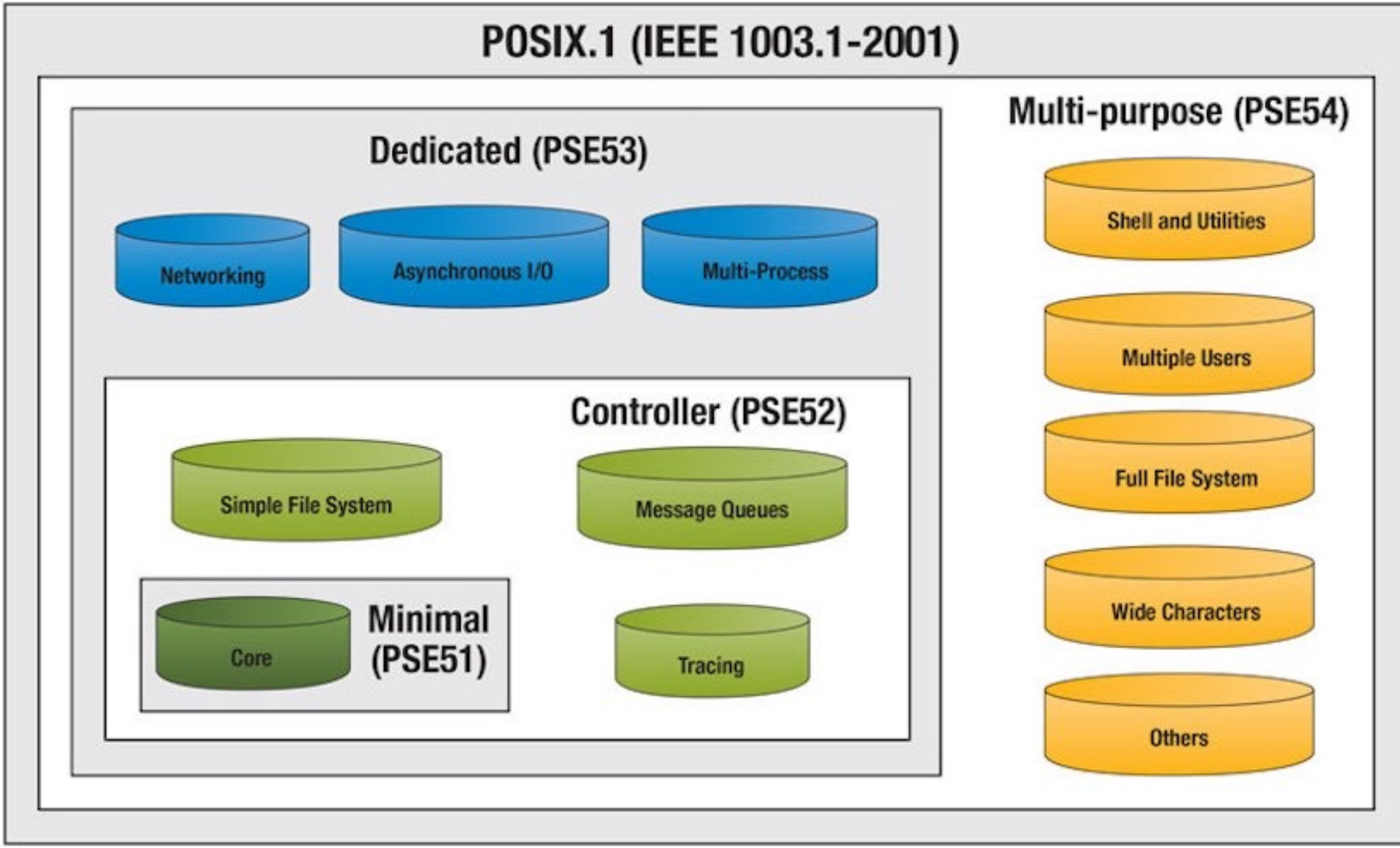
- inexpensive commodity components
- large files
- reads: large streaming & small random
- large, sequential writes
- well-defined semantics, co-designed with apps
- bandwidth more important than low latency

A note

- GFS was designed to be used by ***mostly co-designed applications***
  - Not by regular users
- Explains many of its features

2. Interface (I)

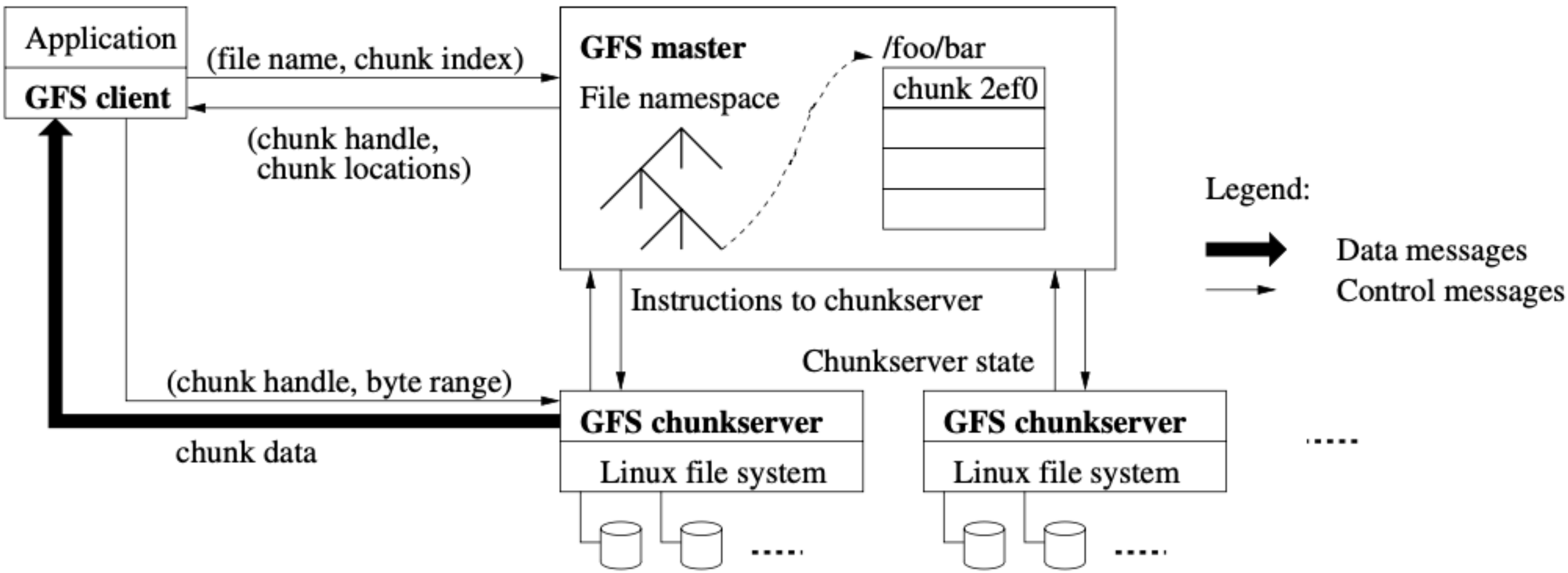
- Quite familiar but non-POSIX
  - Files organized in directories
  - Usual primitives for
    - Creating, deleting, opening, closing files
    - Writing to and reading from files


POSIX.1 (IEEE 1003.1-2001)
Dedicated (PSE53)
Networking
Asynchronous I/O
Multi-Process
Controller (PSE52)
Simple File System
Message Queues
Core
Minimal (PSE51)
Tracing
Multi-purpose (PSE54)
Shell and Utilities
Multiple Users
Full File System
Wide Characters
Others

2. Interface(II)

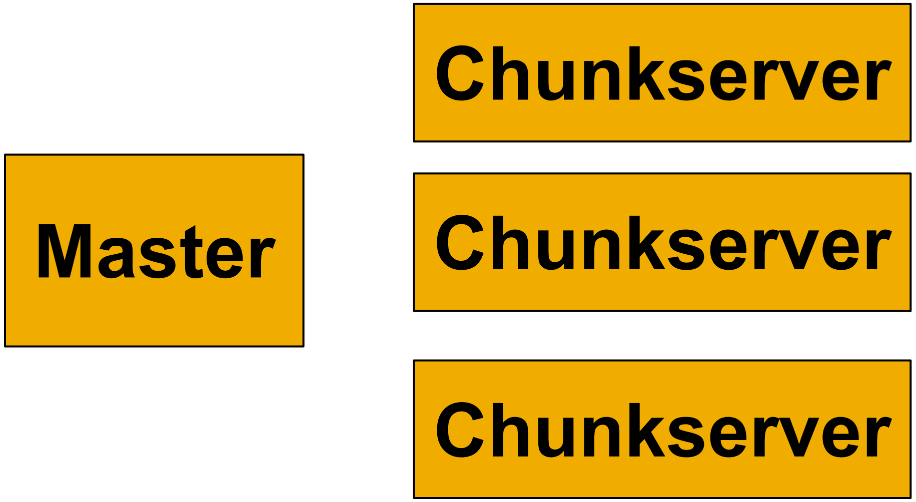
- Two new operations
  - ***Snapshots***
    - Create copies of files and directories
  - ***Record appends***
    - Allow multiple clients to concurrently append data to the same file
    - Useful for implementing
      - Multi-way merge results
      - Producer-consumer queues

3. Architecture



GFS clusters

- GFS Cluster
  - A ***master***
  - Multiple ***chunkservers***
  - Concurrently accessed by many clients



The files

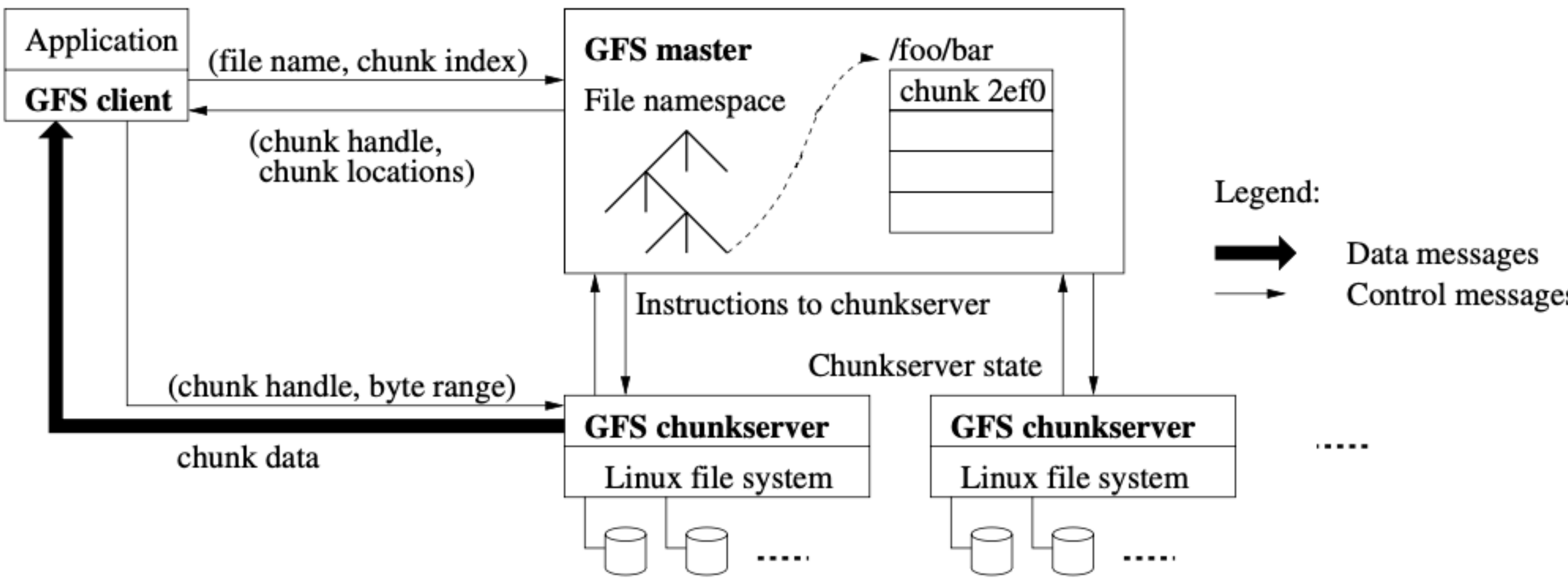
- Files are divided into fixed-size ***chunks*** of 64MB
  - Similar to clusters or sectors in other file systems
- Each chunk has a ***unique 64-bit label***
  - Assigned by the master node at time of creation
- GFS maintain ***logical mappings*** of files to constituent chunks
- Chunks are replicated
  - At least ***three times***
  - More for critical or heavily used files

4. Single Master (I)

- Single master server
- Stores chunk-related metadata
  - Tables mapping the 64-bit labels to chunk locations
  - The files they make up
  - Locations of chunk replicas
  - What processes are reading or writing to a particular chunk, or taking a snapshot of it

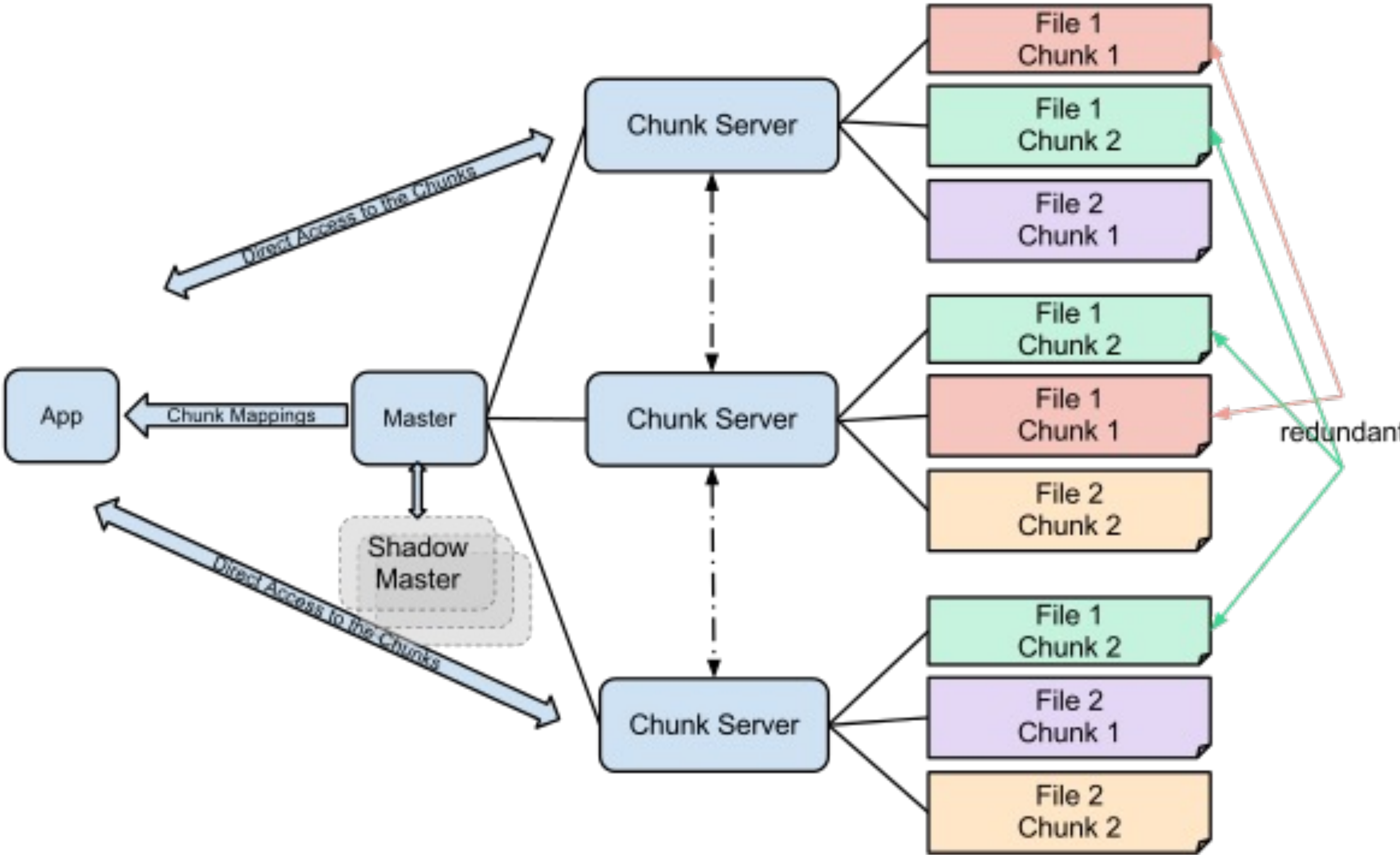
4. Single Master (II)

- Communicates with its chunkservers through heartbeat messages
- Also controls
  - Lease management
  - Garbage collection of orphaned chunks
  - Chunk migration between chunk servers
- The ***metadata server***


Application
GFS client
(file name, chunk index)
(chunk handle,
chunk locations)
GFS master
File namespace
/foo/bar
chunk 2ef0
Legend:
Data messages
Control messages
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
GFS chunkserver
Linux file system
GFS chunkserver
Linux file system

The chunk servers

- Store chunks as Linux files
- Transfer data ***directly*** to/from clients
- Neither the clients nor the chunk servers cache files
  - Little benefits in a streaming environment
  - Omitting it results in a simpler design
  - Linux I/O buffers already keep in RAM frequently accessed chunks



App
Chunk Mappings
Master
Shadow Master
Direct Access to the Chunks
Direct Access to the Chunks
Chunk Server
Chunk Server
Chunk Server
File 1 Chunk 1
File 1 Chunk 2
File 2 Chunk 1
File 1 Chunk 2
File 1 Chunk 1
File 2 Chunk 2
File 1 Chunk 2
File 2 Chunk 1
File 2 Chunk 2
redundant

Accessing a file

1. Client converts (*file name, file offset*) into (*file name, chunk index*)
2. Sends (*file name, chunk index*) to master
3. Master replies with *chunk handle* and *replica locations*
4. Client caches this information
5. Client selects a chunk server and sends (*chunk handle*, *byte range* within the chunk)

Optimization

- Clients typically send requests for multiple chunks to the master
- Master can add to their reply information about chunks immediately following the requested chunks
- Avoid many client requests to the master
  - At almost no cost!

5. Chunk Size

- Large chunk sizes
  - Reduce the number of interactions between clients and master
  - As clients are more likely to perform many operations on the same chunk, they reduce the number of TCP connection requests
  - Reduce the size of the metadata stored on the master
  - Also increase the likelihood of observing ***hot spots***.
    - Not a real problem and replication helps

6. Metadata

- Master stores ***in memory***
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replica
- First two types of metadata are kept persistent by logging mutations to an ***operation log*** stored on the master's HD
- Not true for the locations of chunk replicas
  - Obtained from the chunkservers themselves

Chunk locations

- Obtained from chunkservers
  - At startup time
- Maintained up to date because master
  - Controls all chunk placement
  - Monitors chunkserver status though heartbeats
- Simplest solution

Operation log

- Contains historical record of critical metadata changes
- Acts a logical time line for the order of all concurrent operations
- Replicated on multiple remote machines
  - Using ***blocking writes***, both locally and remotely

7. Consistence Model

- All file namespace mutations are ***atomic***
  - Handled exclusively by the master
- Status of a file region can be
  - ***Consistent:*** all clients see the same data
  - ***Defined:*** all clients see the same data, which include the entirety of the last mutation
  - ***Undefined but consistent***: all clients see then same data but it may not reflect what any one mutation has written
  - ***Inconsistent***

Guarantees by GFS

<table>
<tr><th></th><th>Write</th><th>Record Append</th></tr>
<tr><td>Serial Success</td><td>defined</td><td rowspan="2">Defined interspersed with inconsistent</td></tr>
<tr><td>Concurrent Successes</td><td>consistent but undefined</td></tr>
<tr><td>Failure</td><td colspan="2">inconsistent</td></tr>
</table>

- File namespace mutations (e.g., file creation) are atomic
  - Namespace management and locking
  - The master's operation log
- After a sequence of successful mutations, the mutated file is guaranteed to be defined and contain the data written by the last mutation. This is obtained by
  - Applying the same mutation order to all replicas
  - Using chunk version numbers to detect stale replica

Implications for apps

- **Relying on appends rather on overwrites**
- **Checkpointing**
  - to verify how much data has been successfully written
- **Writing self-validating records**
  - Checksums to detect and remove *padding*
- **Self-identifying records**
  - Unique Identifiers to identify and discard *duplicates*

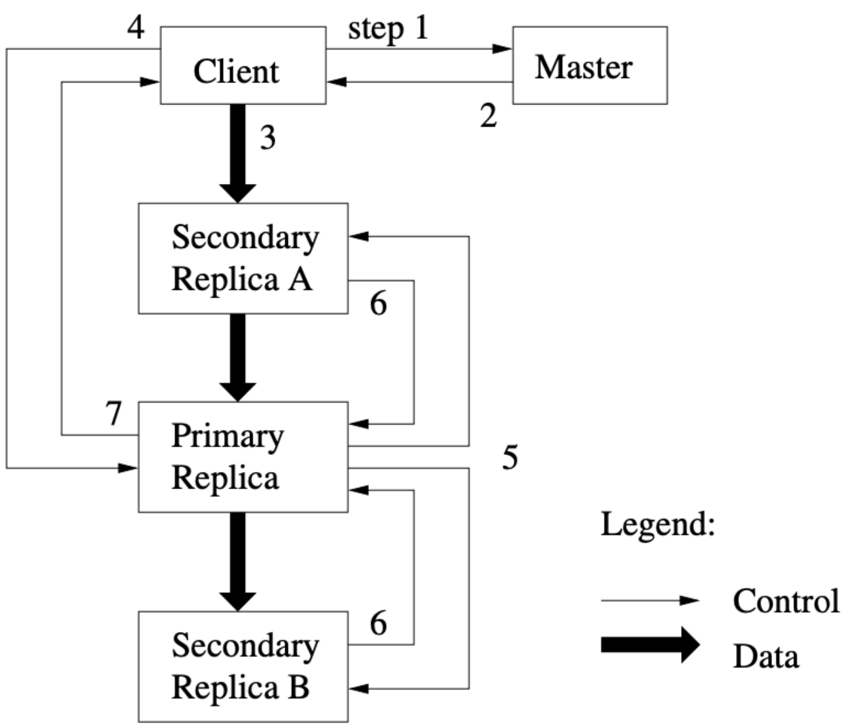
System Interactions

1. **Data Mutations**
2. **Atomic Record Append**
3. **Snapshot**

Leases and Mutation Order

- Master uses *leases* to maintain a consistent mutation order among replicas
- *Primary* is the chunkserver who is granted a chunk lease
- All others containing replicas are *secondaries*
- Primary defines a mutation order between mutations
- All *secondaries* follows this order

Writes



Mutation Order

- → *identical replicas*
- → File region may end up containing *mingled fragments* from different clients (*consistent* but *undefined*)

Data flow

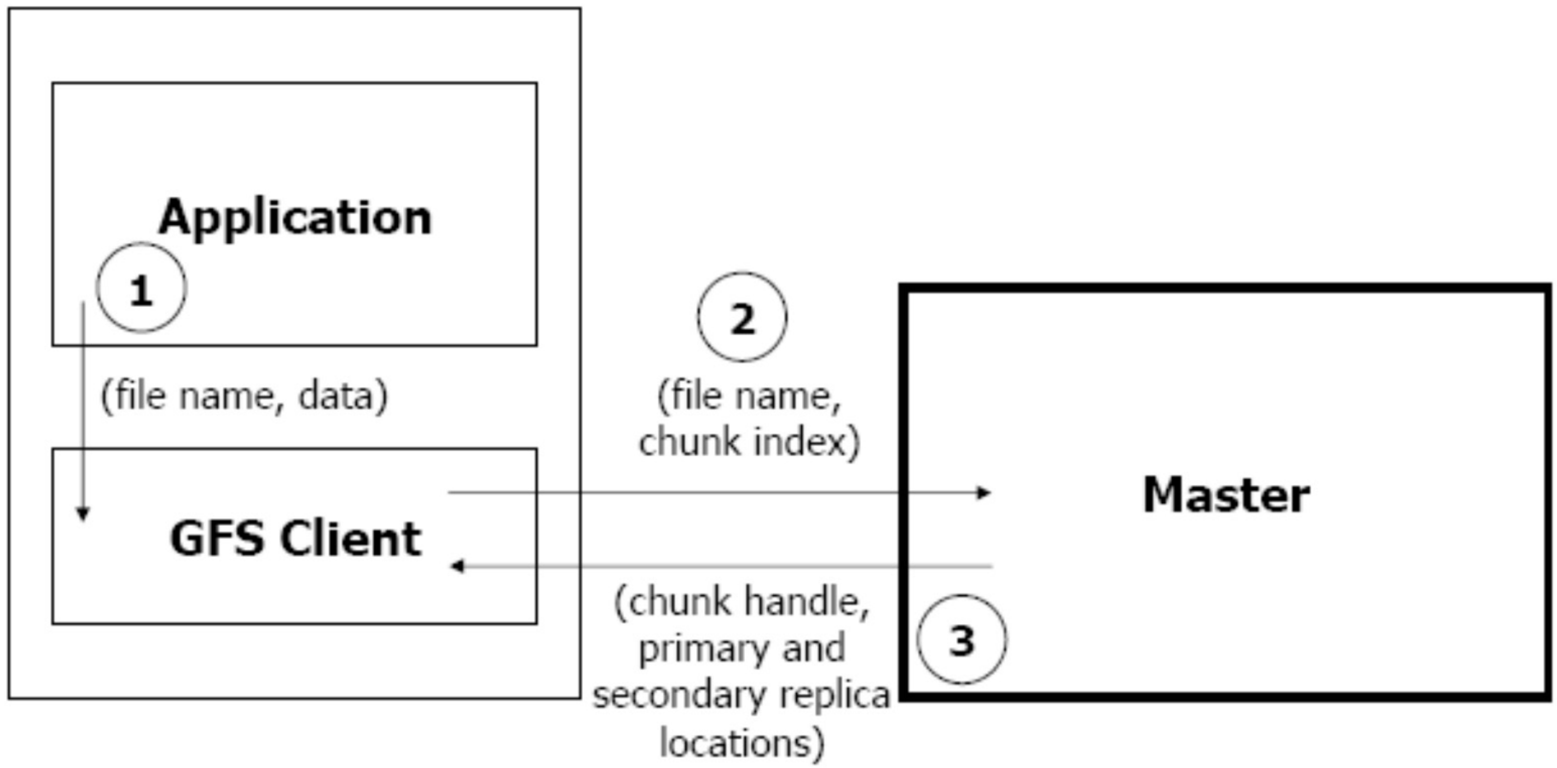
- Decoupled from control flow
  - to use the network efficiently
- Pipelined fashion
    - Data transfer is pipelined over TCP connections
    - Each machine forwards the data to the "closest" machine
- Benefits
  - Avoid bottle necks and minimize latency

Atomic Appends

- **The client specifies only the data**
- **Similar to writes**
  - Mutation order is determined by the *primary*
  - All *secondaries* use the same mutation order
- **GFS appends data to the file *at least once atomically***
  - The chunk is padded if appending the record exceeds the maximum size → *padding*
  - If a record append fails at any replica, the client retries the operation → *record duplicates*
  - File region may be *defined* but interspersed with *inconsistent*

Snapshot

- Goals
  - To quickly create branch copies of huge data sets
  - To easily checkpoint the current state
- Copy-on-write technique
  - Metadata for the source file or directory tree is duplicated
  - Reference count for chunks are incremented
  - Chunks are copied later at the first write


Application
1
(file name, data)
GFS Client
2
(file name,
chunk index)
Master
(chunk handle,
primary and
secondary replica
locations)
3

Namespace Management and Locking

- Namespaces are represented as a lookup table mapping full pathnames to metadata
- Use locks over regions of the namespace to ensure proper serialization
- Each master operation acquires a set of locks before it runs

Example of Locking Mechanism

- Preventing /home/user/foo from being created while /home/user is being snapshotted to /save/user
  - Snapshot operation
    - Read locks on /home and /save
    - Write locks on /home/user and /save/user
  - File creation
    - read locks on /home and /home/user
    - write locks on /home/user/foo
  - Conflict locks on /home/user

Replica Operations

- Creation
  - Disk space utilization
  - Number of recent creations on each chunkserver
  - Spread across many racks
- Re-replication
  - Prioritized: How far it is from its replication goal…
  - The highest priority chunk is cloned first by copying the chunk data directly from an existing replica
- Rebalancing
  - Periodically

Garbage Collection

- Deleted files
  - Deletion operation is logged
  - File is renamed to a hidden name, then may be removed later or get recovered
- Orphaned chunks (unreachable chunks)
  - Identified and removed during a regular scan of the chunk namespace
- Stale replicas
    - *Chunk version numbering*

Fault Tolerance and Diagnosis

- Frequent component failures
- Trust machines?
- Trust disks?

High Availability

- Fast Recovery
  - Operation log
  - Checkpointing
- Chunk replication
  - Each chunk is replicated on multiple chunkservers on different racks
- Master replication
  - Operation log and check points are replicated on multiple machines

Data Integrity

- Data integrity
  - Checksumming to detect corruption of stored data
  - Each chunkserver independently verifies the integrity

Diagnostic tools

- Diagnostic logs
  - Chunkservers going up and down
  - RPC requests and replies

Summary

The Google File System