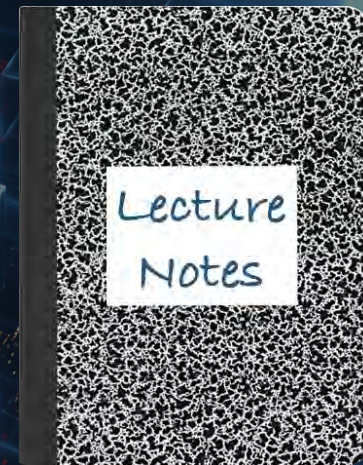


CS 417 – DISTRIBUTED SYSTEMS

Week 7: Parallel File Systems



Paul Krzyzanowski

© 2023 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

Client-server file systems

- Network Attached Storage is built on a **central server** architecture
 - Point of congestion, single point of failure
- **Alleviate performance somewhat with replication and client caching**
 - E.g., Coda, tokens (aka *leases*, *oplocks*)
 - Limited replication can lead to congestion
- **But file data is still centralized**
 - A file server stores all data from a file – not split across servers
 - Even if replication is in place, a client downloads all data for a file from one server
- **File sizes are limited to the capacity available on a server**
 - What if you need a 1,000 TB file?

What is a parallel file system?

- **Conventional file systems**

- Store data & metadata on the same storage device
- Example:
 - Linux directories are just files that contain lists of names & inodes
 - inodes are data structures placed in well-defined areas of the disk that contain information about the file

- **Parallel file systems**

- File data can span multiple servers
- Metadata can be on separate servers from the data
- **Metadata** = information about the file
 - Includes name, access permissions, timestamps, file size, & locations of data blocks
- **Data** = actual file contents

Google File System (GFS)

(\approx Apache Hadoop Distributed File System)

GFS Goals

- Scalable distributed file system
- Designed for large data-intensive applications
- Fault-tolerant; runs on commodity hardware
- Delivers high performance to a large number of clients

Design Assumptions

- Assumptions for conventional file systems don't work
 - E.g., *“most files are small”*, *“lots have short lifetimes”*
- Component failures are the norm, not an exception
 - File system = thousands of storage machines
 - Some % not working at any given time
- Files are huge. Multi-TB files are the norm
 - It doesn't make sense to work with billions of nKB-sized files
 - I/O operations and block size choices are also affected

Design Assumptions

- File access:
 - Most files are appended, not overwritten
 - Random writes within a file are almost never done
 - Once created, files are mostly read; often sequentially
 - Workload is mostly:
 - Reads: large streaming reads, small random reads – *these dominate*
 - Large appends
 - Hundreds of processes may append to a file concurrently
- GFS will store a modest number of files for its scale
 - approx. a few million
- Designing the GFS API together with the design of apps
 - Apps can handle a relaxed consistency model

Basic Design Principles

- **Use separate servers to store metadata**
 - Metadata includes lists of (*server*, *block_number*) sets that identify which blocks on which servers hold file data
 - We need more bandwidth for data access than metadata access
 - Metadata is small; file data can be huge
- **Use large logical blocks**
 - Most "normal" file systems are optimized for small files
 - A block size is typically 4KB
 - Expect huge files, so use *huge blocks* ... >1,000x larger
 - The list of blocks that makes up a file becomes easier to manage
- **Replicate data**
 - Expect some servers to be down
 - Store copies of data blocks on multiple servers

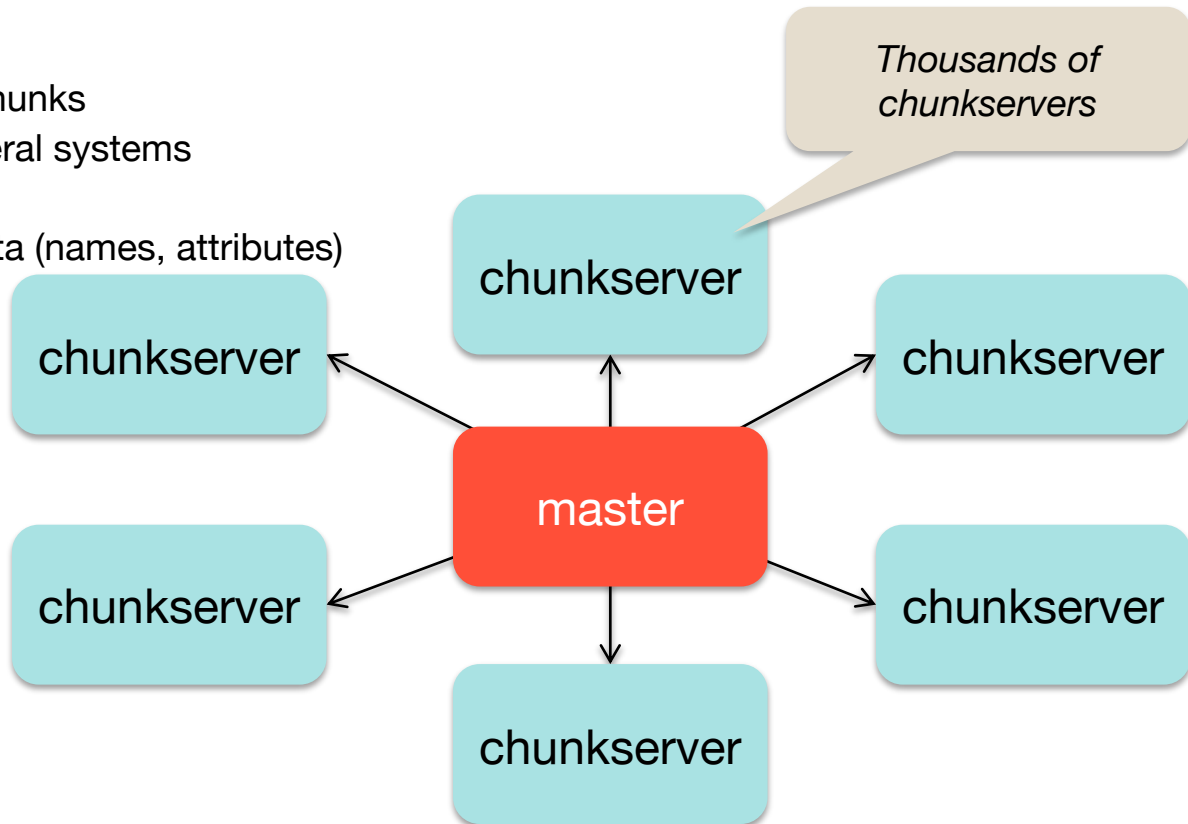
File System Interface

- GFS does *not* have a standard OS-level API
 - No POSIX system call level API – no kernel/VFS implementation
 - User-level API for accessing files
 - GFS servers are implemented in user space using native Linux FS
- Files organized hierarchically in directories
- Operations
 - Basic operations
 - *Create, delete, open, close, read, write*
 - Additional operations
 - *Snapshot*: create a copy of a file or directory tree at low cost
 - *Append*: allow multiple clients to append atomically without locking

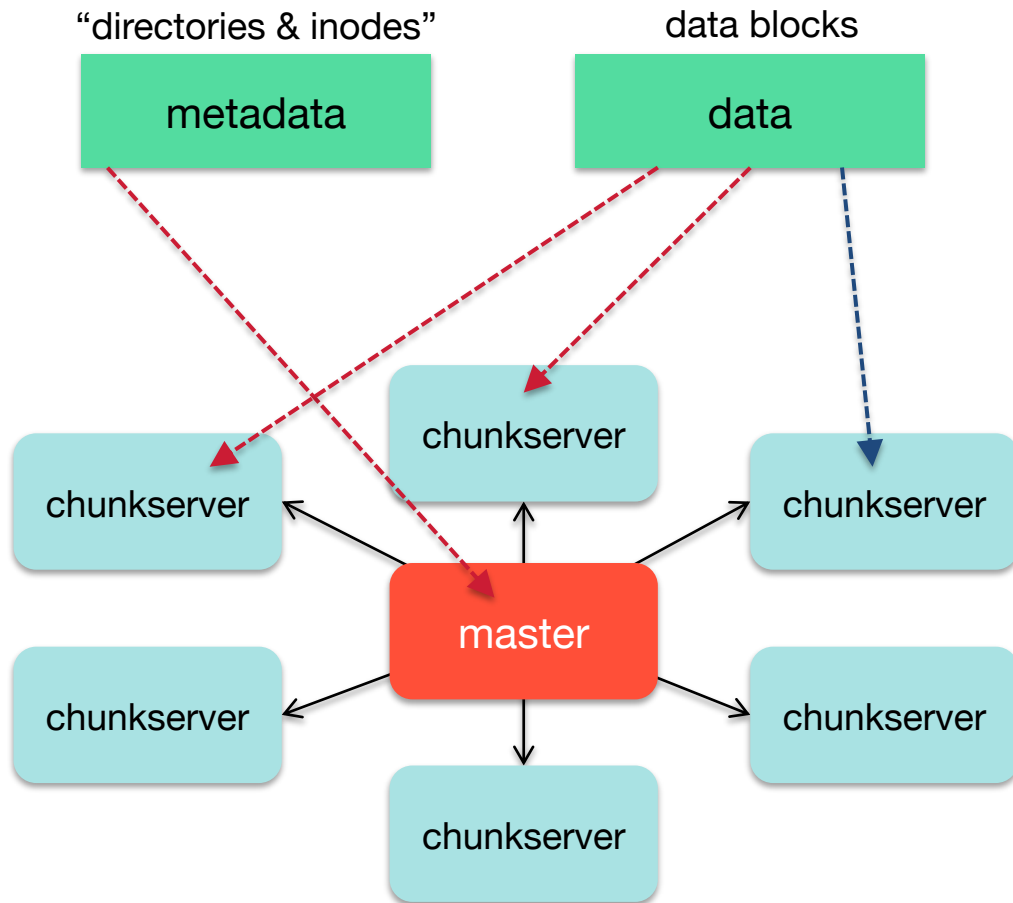
GFS Master & Chunkservers

GFS cluster

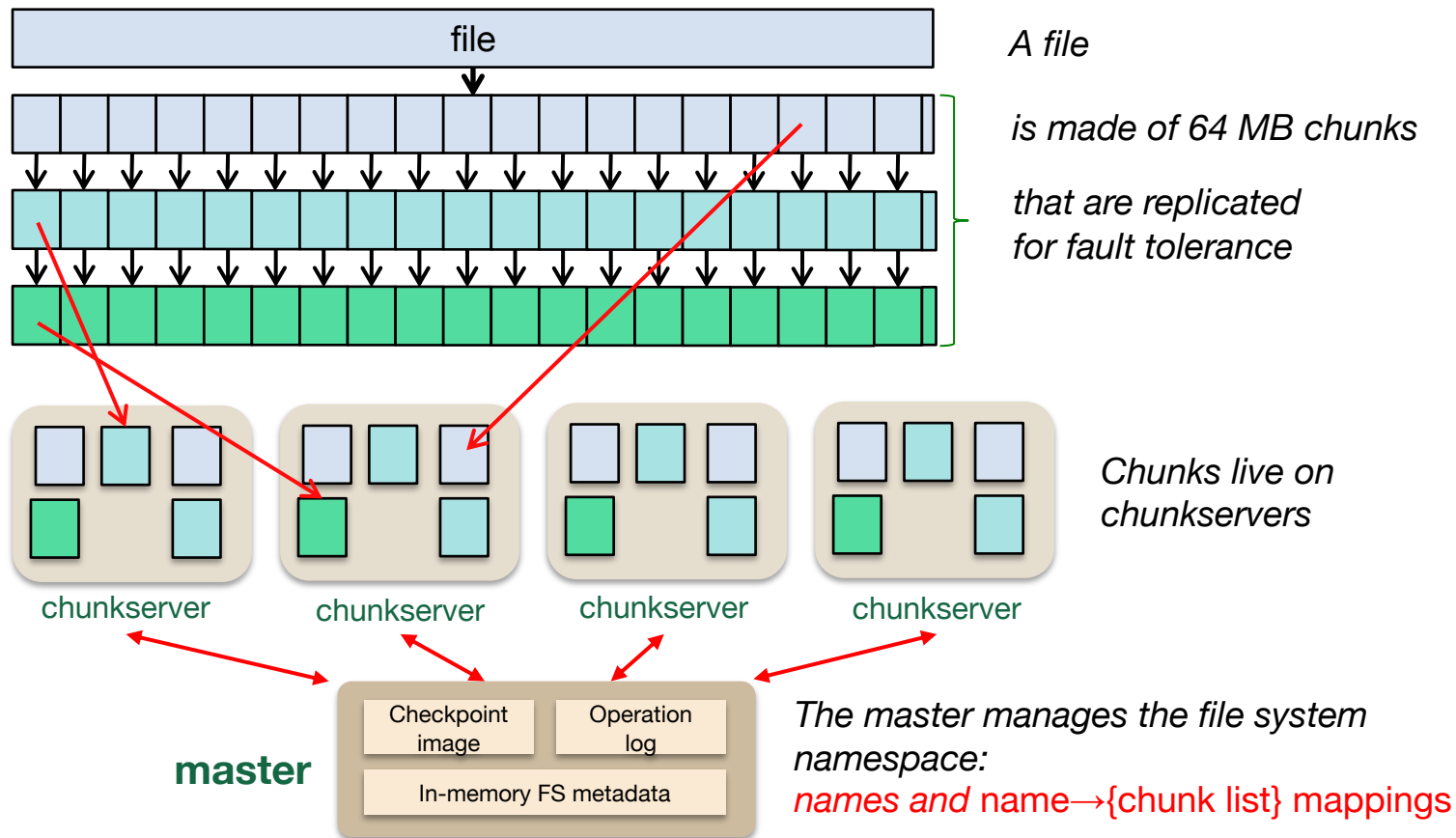
- Multiple **chunkservers**
 - Data storage: fixed-size chunks
 - Chunks replicated on several systems
- One **master**
 - Stores file system metadata (names, attributes)
 - Maps files to chunks



GFS Master & Chunkservers in a GFS cluster



GFS Files



Chunks and Chunkservers

- Chunk size = 64 MB (default)
 - Chunkserver stores a 32-bit checksum with each chunk
 - In memory & logged to disk: allows it to detect data corruption
- **Chunk Handle: identifies a chunk**
 - Globally unique 64-bit number
 - Assigned by the master when the chunk is created
- **Chunkservers** store chunks on local disks as Linux files
- Each chunk is replicated on multiple chunkservers
 - Three replicas (different levels can be specified)
 - Popular files may need more replicas to avoid hotspots

Master

- Maintains all file system metadata
 - Namespace
 - Access control info
 - Filename to chunks mappings
 - Current locations of chunks
- Manages
 - Chunk leases (locks)
 - Garbage collection (freeing unused chunks)
 - Chunk migration (copying/moving chunks)
- Fault tolerance
 - Operation log replicated on multiple machines
 - New master can be started if the master fails
- Periodically communicates with all chunkservers
 - Via heartbeat messages to get state and send commands

Client Interaction Model

- **GFS client code linked into each app**
 - No OS-level API – you have to use a library
 - Interacts with master for metadata-related operations
 - Interacts directly with chunkservers for file data
 - All reads & writes go directly to chunkservers
 - Master is not a point of congestion
- **Neither clients nor chunkservers cache data**
 - Except for the caching by the OS system buffer cache
 - Clients cache metadata – e.g., location of a file's chunks

One master = simplified design

- All metadata stored in master's memory
 - Super-fast access
- Namespaces and *name-to-chunk_list* maps
 - Stored in memory
 - Also persist in an *operation log* on the disk
 - Replicated onto remote machines for backup
- **Operation log**
 - Similar to a journal
 - All operations are logged
 - Periodic checkpoints (stored in a B-tree) to avoid playing back entire log
- Master does not store chunk locations persistently
 - This is queried from all the chunkservers: avoids consistency problems

Why Large Chunks?

- Default chunk size = 64MB
(Linux ext4 block sizes: typically, 4 KB and up to 1 MB)
- Reduces need for frequent communication with master to get chunk location info – *one query can give info on location of lots of bytes of data*
- Clients can easily cache info to refer to all data of large files
 - Cached data has timeouts to reduce possibility of reading stale data
- Large chunk makes it feasible to keep a TCP connection open to a chunkserver for an extended time
- Master stores <64 bytes of metadata for each 64MB chunk

Reading Files

1. Contact the master
2. Get file's metadata: list chunk handles
3. Get the location of each of the chunk handles
 - Multiple replicated chunkservers per chunk
4. Contact any available chunkserver for chunk data

Writing to files

- Less frequent than reading
- Master grants a **chunk lease** to one of the replicas
 - This replica will be the **primary replica** chunkserver
 - Primary can request lease extensions, if needed
 - Master increases the chunk version number and informs replicas

Writing to files: two phases

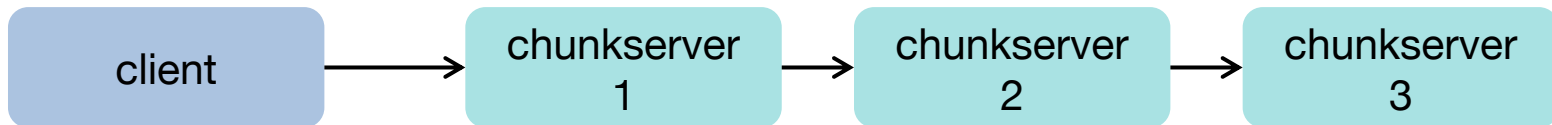
Phase 1: Send data

Deliver data but don't write to the file

- Client asks the master for a list of chunkservers with replicas: primary & secondaries
- Client writes to the closest replica chunkserver that has not received the data
 - Replica forwards the data to another replica chunkserver
 - That chunkserver forwards to another replica chunkserver ...
- Chunkservers store this data in a cache – *it's not part of the file yet*

Goal: Maximize bandwidth via pipelining

Minimize latency by forwarding data while it is being received



Writing to files: two phases

Phase 2: Write data

Add it to the file (commit)

- Client waits for replicas to acknowledge receiving the data
- Sends a *write* request to the primary, identifying the data that was sent
- The primary is responsible for serialization of writes
 - Assigns consecutive serial numbers to all writes that it received
 - Applies writes in serial-number order and forwards write requests in that order to secondaries
- Once all acknowledgments have been received, the primary acknowledges the client



Writing to files: separate data flow & control flow

Data Flow (*phase 1*) is different from **Control Flow** (*phase 2*)

- **Data Flow** (*upload*):
 - Client to chunkserver to chunkserver to chunkserver...
 - Order does not matter
- **Control Flow** (*write*):
 - Client to primary; primary to all secondaries
 - Locking used; Order maintained

Chunk version numbers are used to detect if any replica has stale data (was not updated because it was down)

Namespace

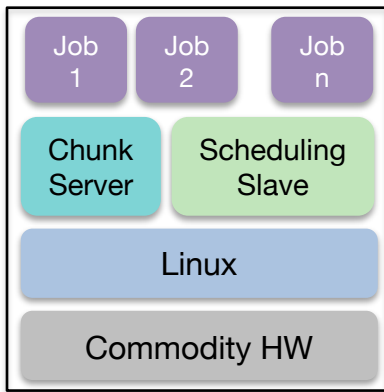
- No per-directory data structure like most file systems
 - E.g., directory file contains names of all files in the directory
- No aliases (hard or symbolic links)
- Namespace is a single lookup table
 - Maps pathnames to metadata

Core Part of Google Cluster Environment

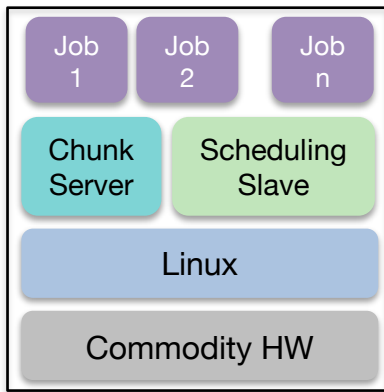
Google Cluster Environment

- Core services: GFS + cluster scheduling system
- Typically, 100s to 1000s of active jobs
- 200+ clusters, many with 1000s of machines
- Pools of 1000s of clients
- 4+ PB file systems, 40 GB/s read/write loads

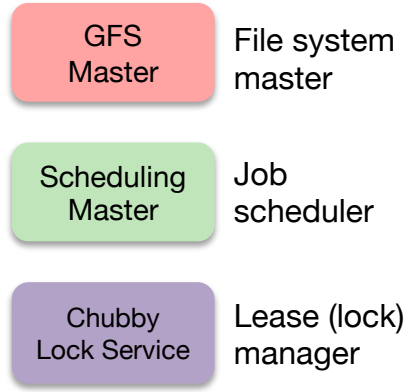
Bring the computation
close to the data



Machine 1



Machine n



HDFS: Hadoop Distributed File System

- Primary storage system for Hadoop applications
- Apache Hadoop
 - Framework for distributed processing of large data sets across clusters of computers
- Hadoop includes:
 - **MapReduce™**: software framework for distributed processing of large data sets on compute clusters.
 - **Avro™**: A data serialization system.
 - **Cassandra™**: A scalable multi-master database with no single points of failure.
 - **Chukwa™**: A data collection system for managing large distributed systems.
 - **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
 - **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
 - **Mahout™**: A Scalable machine learning and data mining library.
 - **Pig™**: A high-level data-flow language and execution framework for parallel computation.
 - **ZooKeeper™**: A high-performance coordination service for distributed applications
 - *and more ...*

HDFS Design Goals & Assumptions

- HDFS is an open source (Apache) implementation inspired by GFS design
- Similar goals and same basic design as GFS
 - Run on commodity hardware
 - Highly fault tolerant
 - High throughput – designed for large data sets
 - OK to relax some POSIX file access requirements
 - Large scale deployments
 - Instance of HDFS may comprise 1000s of servers
 - Each server stores part of the file system's data
- But
 - No support for concurrent appends

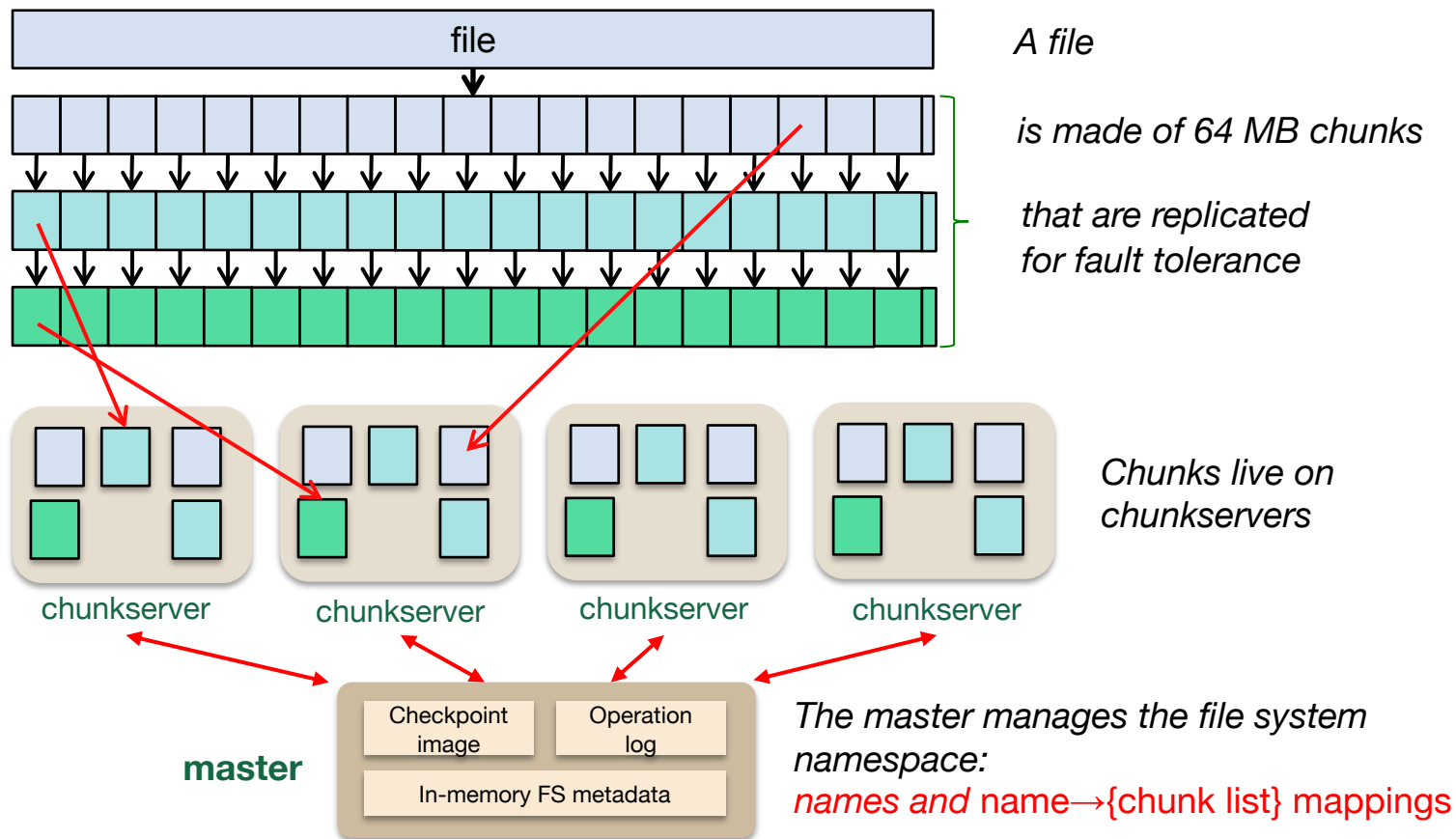
HDFS Design Goals & Assumptions

- Write-once, read-many-times file access model
 - Single writer, multiple readers
- A file's contents will not change
 - Simplifies data coherency
 - Suitable for web crawlers and big data analytics applications

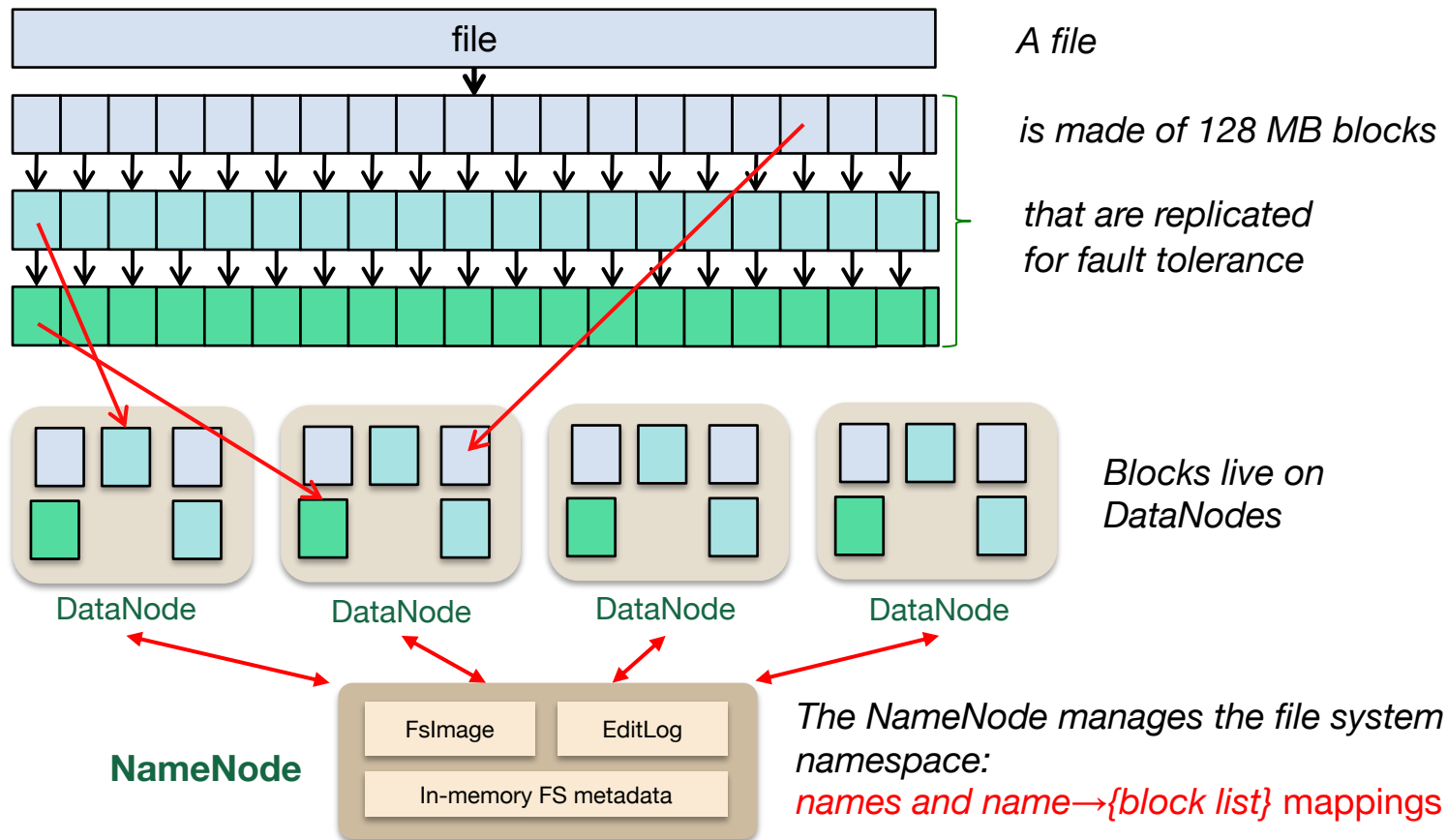
HDFS Architecture

- Written in Java
- Single **NameNode**
 - Master server responsible for the namespace & access control
- Multiple **DataNodes**
 - Responsible for managing storage attached to its node
- A file is split into one or more blocks
 - Typical block size = 128 MB (vs. 64 MB for GFS)
 - Blocks are stored in a set of DataNodes

GFS Files



HDFS: same stuff ... different names



NameNode (= GFS master)

- Executes metadata operations
 - *open, close, rename*
 - Maps file blocks to DataNodes
 - Maintains HDFS namespace
- Transaction log (**EditLog**) records every change that occurs to file system metadata
 - Entire file system namespace + file-block mappings is stored in memory
 - ... and stored in a file (**FsImage**) for persistence
- **NameNode** receives a periodic *Heartbeat* and *Blockreport* from each **DataNode**
 - Heartbeat = “*I am alive*” message
 - Blockreport = list of all blocks managed by a DataNode
 - Keep track of which DataNodes own which blocks & their replication count

DataNode (= GFS chunkserver)

- Responsible for serving read/write requests
- Blocks are replicated for fault tolerance
 - App can specify # replicas at creation time
 - Can be changed later
- Blocks are stored in the local file system at the DataNode

Rack-Aware Reads & Replica Selection

- Client sends request to NameNode
 - Receives list of blocks and replica DataNodes per block
- Client tries to read from the closest replica
 - Prefer same rack
 - Else same data center
 - Location awareness is configured by the admin

Writes

- Client caches file data into a temp file
- When temp file \geq one HDFS block size
 - Client contacts NameNode
 - NameNode inserts file name into file system hierarchy & allocates a data block
 - Responds to client with the destination data block
 - Client writes to the block at the corresponding DataNode
- When a file is closed, remaining data is transferred to a DataNode
 - NameNode is informed that the file is closed
 - NameNode commits file creation operation into a persistent store (log)
- Data writes are chained: pipelined
 - Client writes to the first (closest) DataNode
 - That DataNode writes the data stream to the second DataNode
 - And so on...

The End