Distributed File Systems

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Credits: some slides from Paul Krzyzanowski (Rutgers)

Accessing Remote Files

FTP, telnet, …

- Explicit access
- User-directed connection to access remote resource

We want more **transparency**

- Allow user to access remote resource just as local ones

NAS: **N**etwork **A**ttached **S**torage

File Service Types

Upload/Download model

- Read file: copy file from server to client
- Write file: copy file from client to server

Advantage

- Simple

Problem

- Wasteful "*what if client needs small piece?*"
- Problematic "*what if client doesn't have enough space?*"
- Consistency "*what if others modify the same file?*"

File Service Types

Remote access model

- File service provide functional interface (**create**, **delete**, **read**, **write**, etc …)

Advantage

- Client gets only what's needed
- Server can manage coherent view of file system

Problem

- Possible server and network congestion
  - Servers are accessed for duration of file access
  - Same data may be requested repeatedly

Remote File Service

File Service

- Provides file access interface to clients

Directory Service

- Maps textual names for file to internal locations that can be used by file service

Client module (driver)

- Client side interface for file & directory service
- If done right, helps provide access transparency
  - e.g. implement the **FS** under the **VFS** layer

Semantics of File Sharing

Sequential semantics

- Read returns result of last write

Easily achieved if

- Only one server
- Clients do not cache data

BUT

- Performance problem if no cache
- We can write-through
  - Must notify clients holding copies
  - Requires extra state, generates extra traffic

Semantics of File Sharing

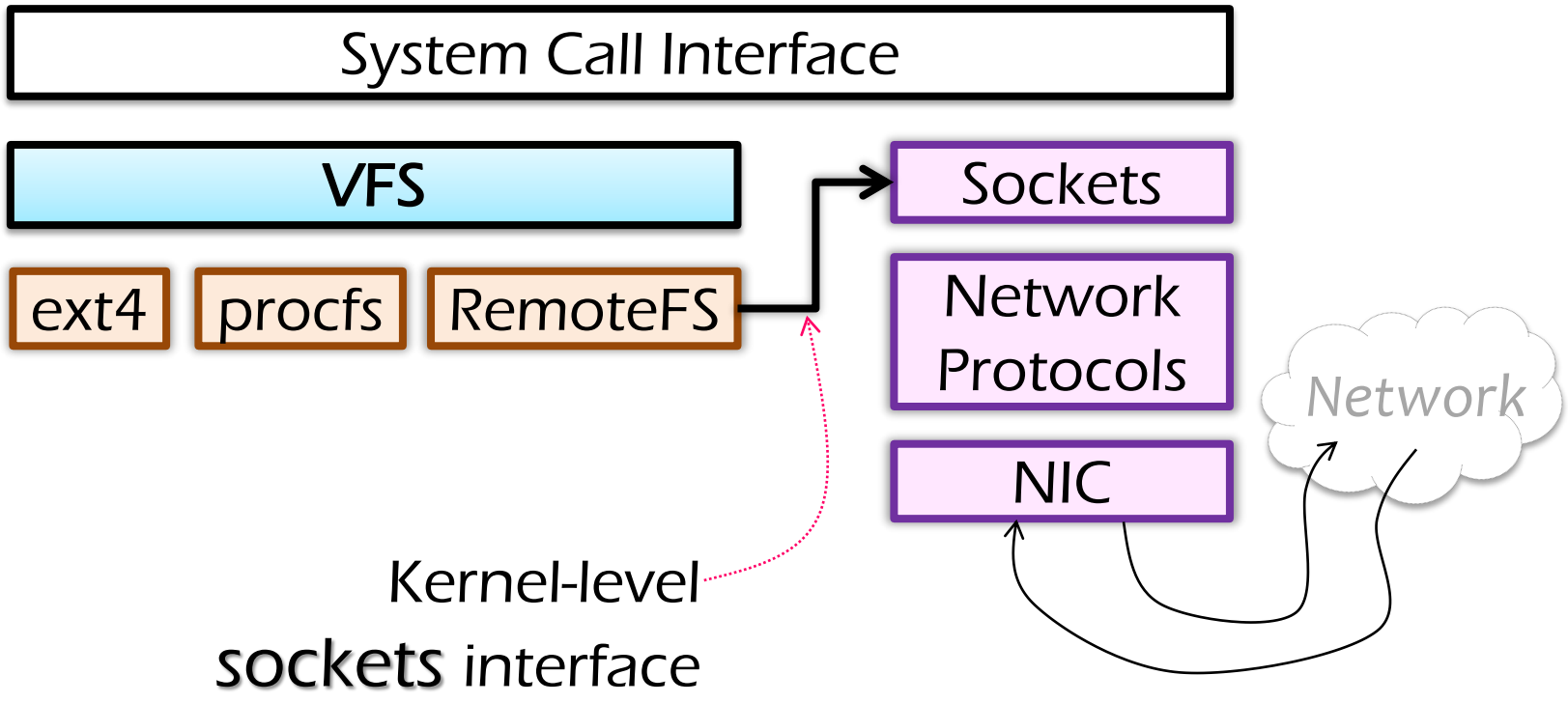
Session semantics

Relax the rules

- Changes to an open file are initially visible only to the process that modified it
- Last process to modify the file wins

Accessing Remote Files

Implement the client module as a FS under VFS



Stateful or Stateless

Stateful → Server maintains client-specific state

Shorter requests

Better performance in processing requests

Cache coherence is possible

- Server can know who's accessing what

File locking is possible

Stateful or Stateless

Stateless → Server maintains no info for client

Each request must identify file and offsets

Server/Client can crash and recovery

- No state to lose

No open/close needed

No server space used for state (scalable)

Problems if file is deleted on server

File locking not possible

Caching

Hide latency to improve performance for repeated accesses by bringing the data closer to where it's needed

Three places to replicate data

- Server's buffer cache
- Client's buffer cache
- Client's disk

WARNING:
Potential cache consistency problem

Approaches to Caching

Write-through

- Keep data cached at the client
  but send modifications to the server

- What if another client reads its own (out-of-data) cache copy?
- All accesses will require checking with server
- Or … server maintains state of clients and sends invalidations

Approaches to Caching

Delayed writes (write-behind)

- Data is cached locally (watch out for consistency – others won't see updates)
- Remote files updated ~~(immediately)~~ periodically
- One bulk write is more efficient than lots of little writes

- Problem: semantics become ambiguous

Approaches to Caching

Read-ahead (prefetch)

- Request chunks of data **before** it is needed
- Minimize **wait** when it actually is needed

Write on close

- Admit that we have **session** semantics

Centralized control

- Keep track of who has what **open** and **cached** on each node
- Stateful file system with signaling traffic

NFS: Network File System

Design Goals

- Any machine can be a client or server
- Must support diskless workstations
- Heterogeneous system must be supported
  - Different HW, OS, underlying file system
- Access transparency
- Recovery from failure
  - Stateless, UDP, client retries
- High performance
  - Use caching and read-ahead

NFS Protocols: Mount

Mounting Protocol:

Request access to exported directory tree

- Client sends pathname to server
  - Requests **permission** to access contents

> **Client**: parses **pathname**
> contacts server for file **handle**

- Server returns file handle

> **Client**: create in-memory VFS **inode** at mount point
> internally points to **rnode** for remote files
> (client keeps state, not the server)

NFS Protocols: Mount

Static mounting

- *mount* request contacts server

Server: add list of shared directories to /etc/exports

Client: mount r900:/users/paul /home/paul

NFS Protocols: Access

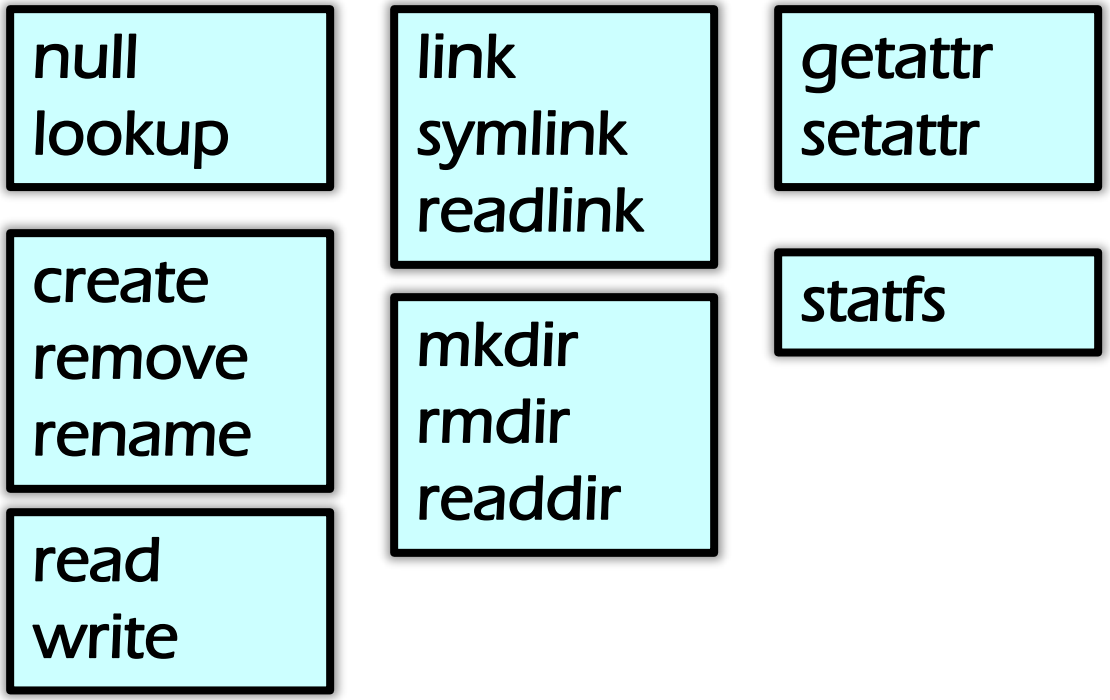
Directory and File Access Protocol:

Access files and directories (read, mkdir, …)

- First, perform a **lookup** RPC
  - Returns file handle and attributes
- "**lookup**" is not like "**open**"
  - Establish state on the client only (no information on server)
  - Call the **NFS** lookup function
- The handle passed as a parameter for other file access function
  - e.g., read(handle, offset, count)

NFS Protocols: Access

NFS has 16 functions (version 2)



NFS Performance

Usually slower than local

Improve by caching at client

- Goal: reduce number of remote ops
- Caching: *read, readlink, getattr, lookup, readdir*
  1. Cache file data at client (buffer cache)
  2. Cache file attribute information at client
  3. Cache pathname bindings for faster lookup

Server side

- Caching is "automatic" via buffer cache
- All NFS writes are write-through to disk

Validation

Inconsistencies may arise in NFS

Try to resolve by validation

- Save **timestamp** of file
- When file opened or server contacted for new
  1. Compare last modification time
  2. If remote is more recent, invalidate cached data

Always **invalidate** data after some time

- open files (3 sec), directories (30 sec)

If data block is modified, it is:

- Marked dirty, then flushed on file close

Improving Read Performance

Transfer data in large chunks

- 8KB default

Read-ahead

- Optimize for sequential file access
- Send requests to read disk blocks before they are requested by the applications

Problem with NFS

File consistency

Assumes clocks are synchronized

Open with append can't be guaranteed to work

Locking cannot work

- Separate lock manager added (stateful)

No reference counting of open files

- You can delete a file opened by yourself/others

Global UID space assumed

…

Improving NFS

Version 3

User-level lock manager

NVRAM support

Adjust RPC retries dynamically

Client-side disk caching

Support 64-bit file sizes

TCP support and large-block transfers

Commit operation

...

Version 4

More state: control of caching, notify of file changes

Server export a single name space (pseudo file system)

Compound RPC support

Extended attribute and ACL

Negotiate security mechanism on mount

...

GFS: The Google File System

Sanjay Ghemawat, "*The Google File System*". SOSP, **2003**

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", "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. *n*-GB/TB files are the norm

- I/O ops and block size choices are affected

Design Assumptions

File access

- Most files are appended, not ~~overwritten~~
  - Random writes within a file are rare
  - Once created, files are mostly read; often seq.
- Workload is mostly
  - Mostly reads: large streaming reads
  - Large appends
  - Hundreds of *proc*s append to a file concurrently

Designing the **FS** API with the design of **apps** benefits the system

File System Interface

GFS does not have a standard OS-level API

- No POSIX API
- No kernel/VFS implementation
- User-level API

Files organized hierarchically in directories

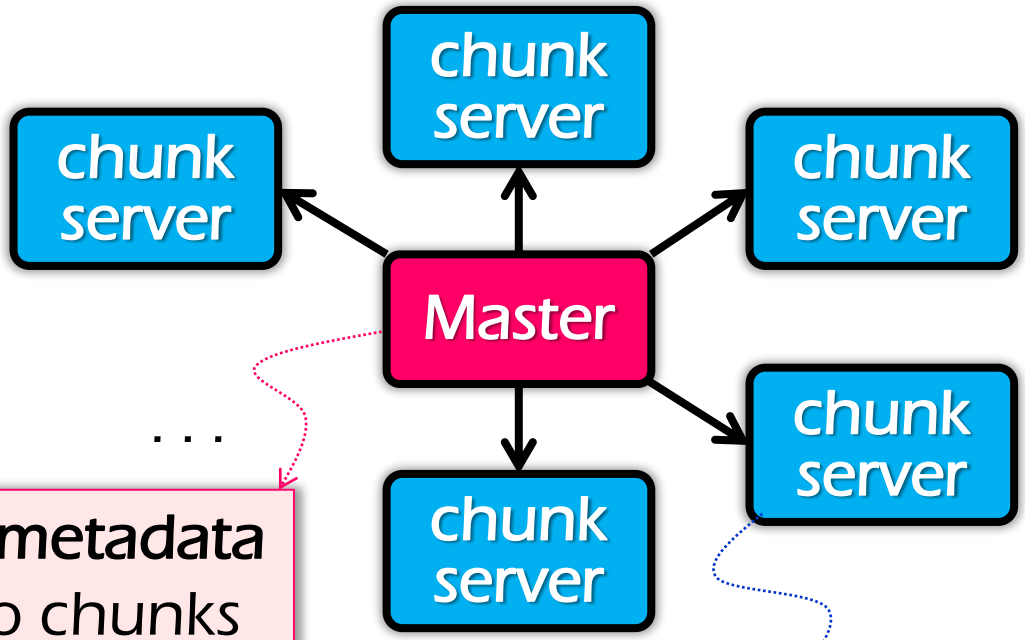
Operations

- Basic ops: **create/delete/open/close/read/write**
- Additional ops: **snapshot/append**

Allow multi-clients to append atomically without locking

GFS Servers

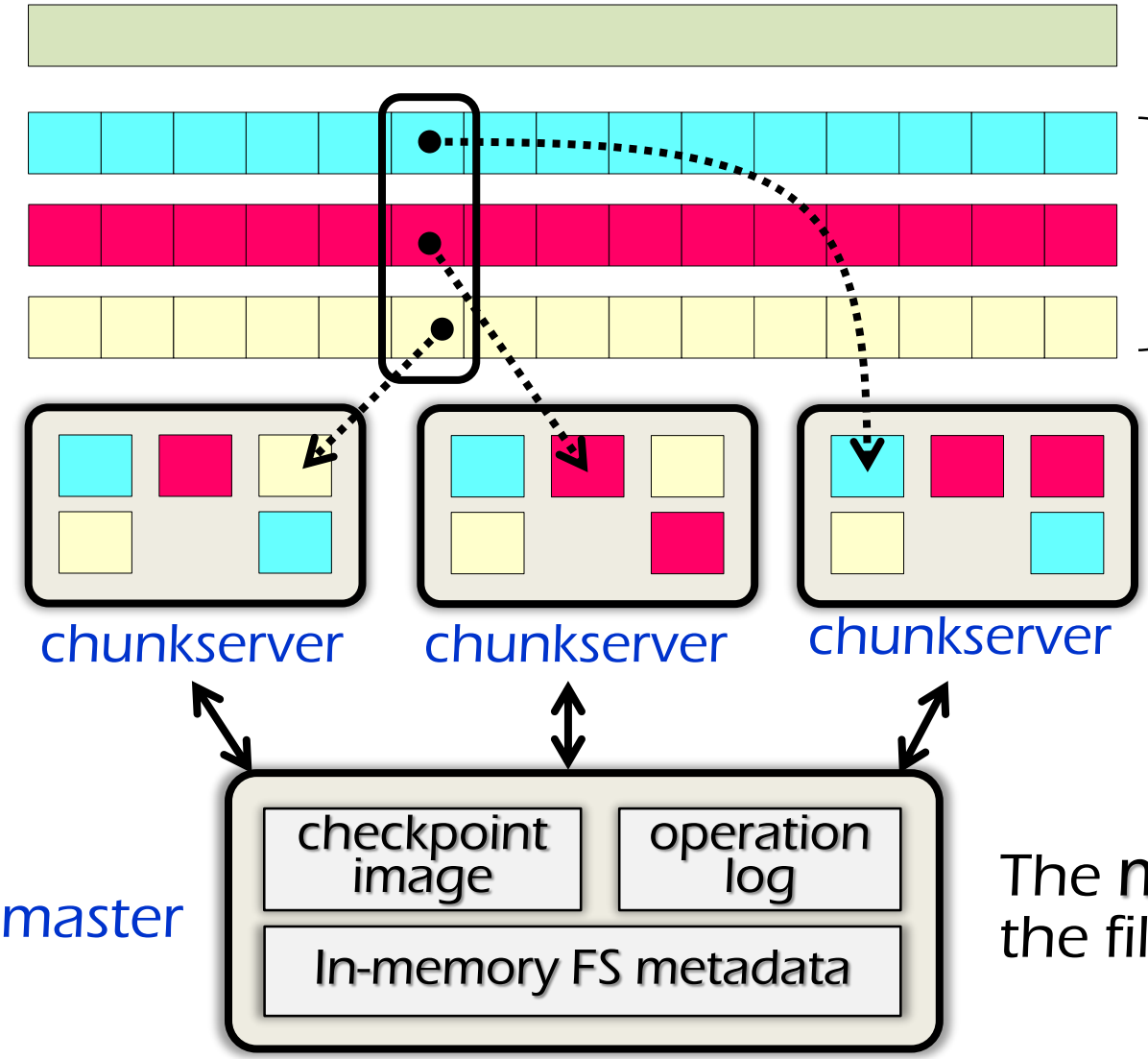
GFS cluster: *N* **Chunkservers** + *1* **Master**



File system **metadata**
Maps files to chunks

Data Storage: fixed-size chunks
Chunks replicated on several system

GFS Files



A File is made of 64MB chunks

That are replicated for fault-tolerance

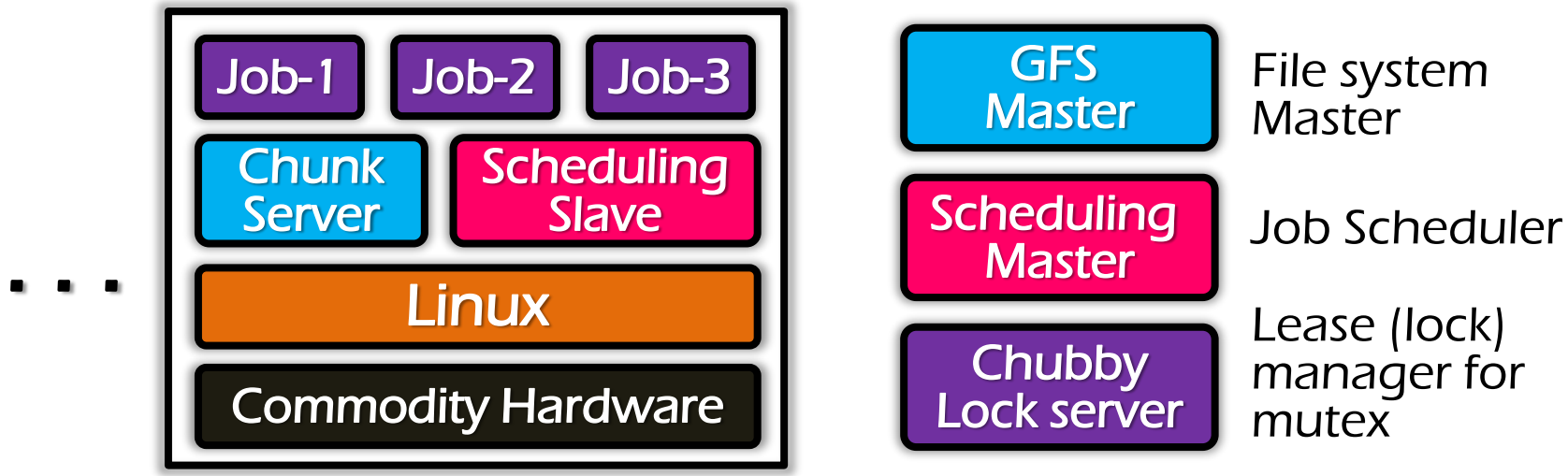
Chunks live on chunkservers

The master manages the file system namespace

Core Part of Google Cluster

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 filesystems, 40GB/s read/write loads



Chunks and Chunkservers

Chunk size = 64 MB (default)

- 32-bit checksum with each chunk

Chunk **Handle**

- Globally unique 64-bit number
- Assigned by the master when creation

Chunks are stored on local disk as Linux files

Each chunk is replicated on multiple nodes

- Three replicas (default)
- more replicas for popular files 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, chunk migration

Master replicates its data for fault-tolerance

Periodically communicates with all nodes

- Via heartbeat messages
- To get state and send commands

Client Interaction Model

GFS client code linked into each app

- No OS-level API
- Interacts with master for metadata-related ops
- Interacts directly with chunkservers for data
  - Master is not a point of congestion

Neither clients nor chunkservers cache data

- Except for the system buffer cache

Clients cache metadata

- e.g., location of a file's chunks

One Master = Simple Design

All **metadata** stored in master's **memory**

- Super-fast access

Namespaces and **name-to-chunk** maps

- Stored in **memory**
- Also persist in an **operation log** on the disk

Similar to a journal

All ops are logged

Periodic checkpoints (B-tree) to avoid playing back entire log

Don't store chunk location persistently

- This is queried from all the chunkservers: avoid consistency problems

Why Large Chunks?

Default chunks size = 64MB
(compare to Linux ext4 block size: 4KB~1MB)

Reduces need for frequent communication with master to get chunk location info

Large chunk makes it feasible to keep a TCP connection open for an extended time

Master stores all metadata in memory

Reading Files

Contact the master

Get file's metadata: chunk handles

Get the location of each of the chunk handles

- Multiple replicated chunkservers per chunk

Contact any available chunkserver for chunk

Writing Files

Less frequent than reading

Master grants a **chunk lease** to one of the replicas

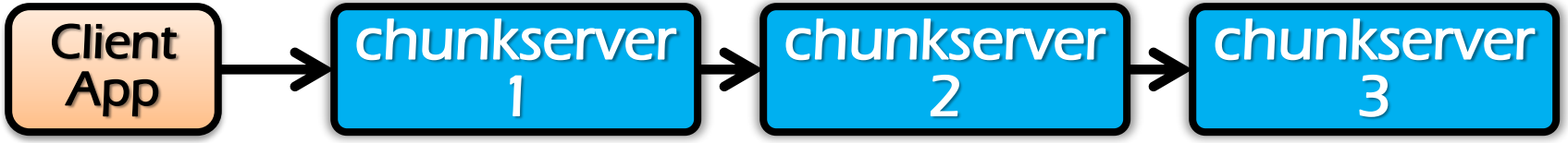
- This replica will be the **primary** chunkserver
- Primary can request extensions, if needed
- Master increases the chunk version number and informs replicas

Writing Files: Two Phases

Phase 1: send data

Deliver data but don't write to the file

- A client is given a list of replicas
  - Identifying the **primary** and **secondaries**
- Client writes to the closest replica
  - Pipeline forwarding
- Chunkservers store this data in a cache

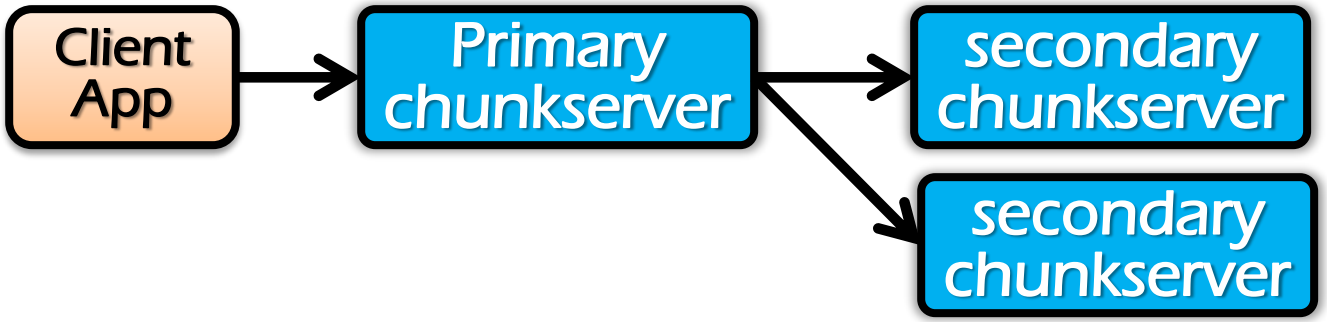


Writing Files: Two Phases

Phase 2: write data

Add it to the file (commit)

- Client waits for replicas to ack. receiving data
- Send a **write** request to the **primary**
- The **primary** is responsible for serialization of writes (*applying* then *forwarding*)
- Once all ack. have been received
  → the **primary** ack. the client



Writing Files

Data flow (phase 1) is different from control flow (phase 2)

Data flow

- Client → chunkserver → chunkserver → ...
- Order does not matter

Control flow

- Client → primary → all secondaries
- Order maintained

Chunk version numbers are used to detect if any replica has stale data

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

HDFS: Hadoop Distributed FS

Primary storage system for Hadoop apps

Hadoop A framework that allows for the distributed processing of large data sets across clusters of computers

Apache Hadoop Ecosystem




Design Goals & Assumptions

HDFS is an open source (Apache) implementation inspired by GFS design

Similar goals as GFS

- Run on commodity hardware
- Highly fault tolerant
- High throughput & large-scale deployments
- …

Write-once, read-many file access model

A file's contents will not change

HDFS Architecture

Written in Java

Master/Slave architecture

Single NameNode

- Master server responsible for the namespace & access control

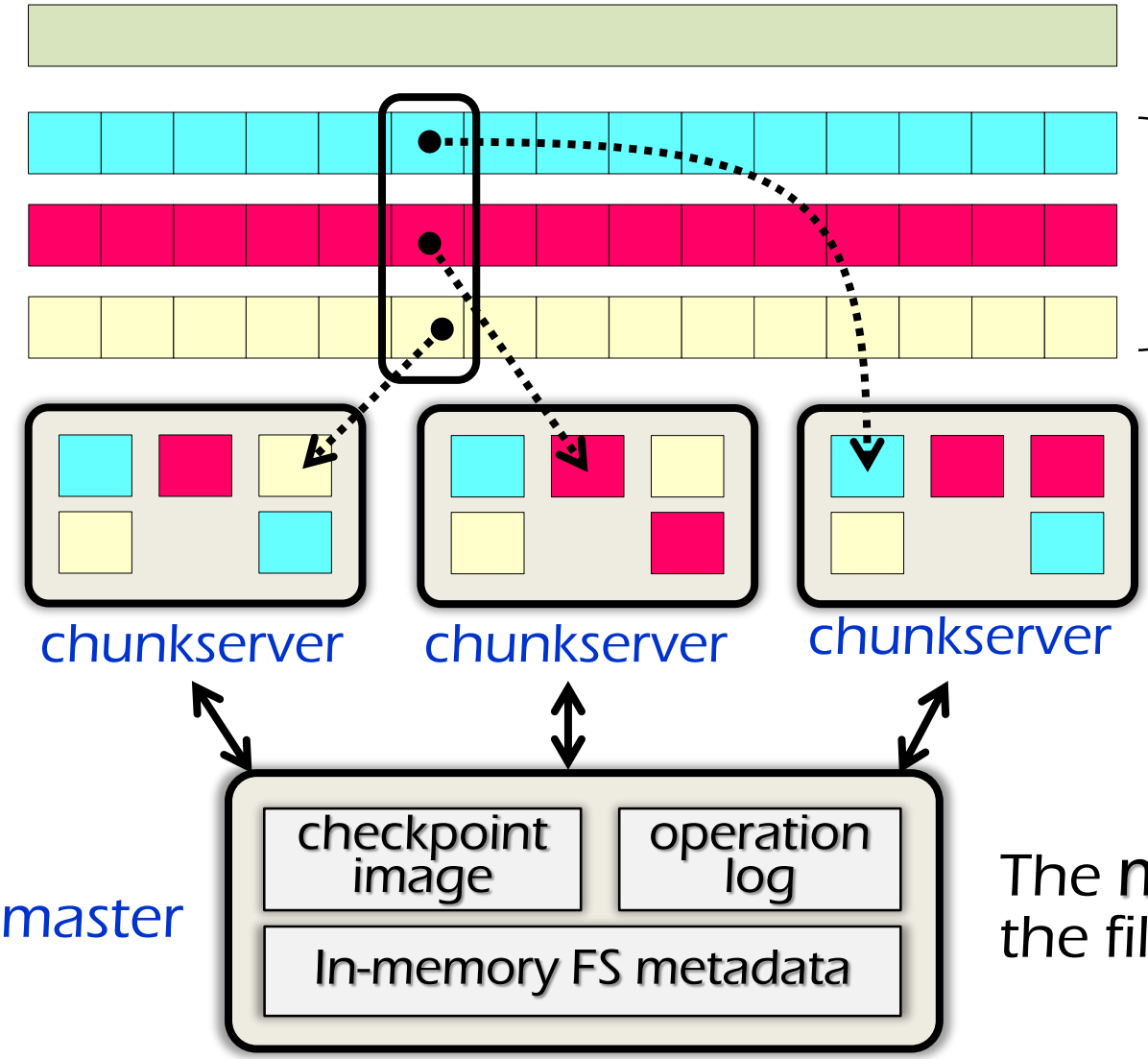
Multiple DataNode

- Responsible for managing storage attached

A file is split into one or more blocks

- Typical block size = 64MB

GFS Files



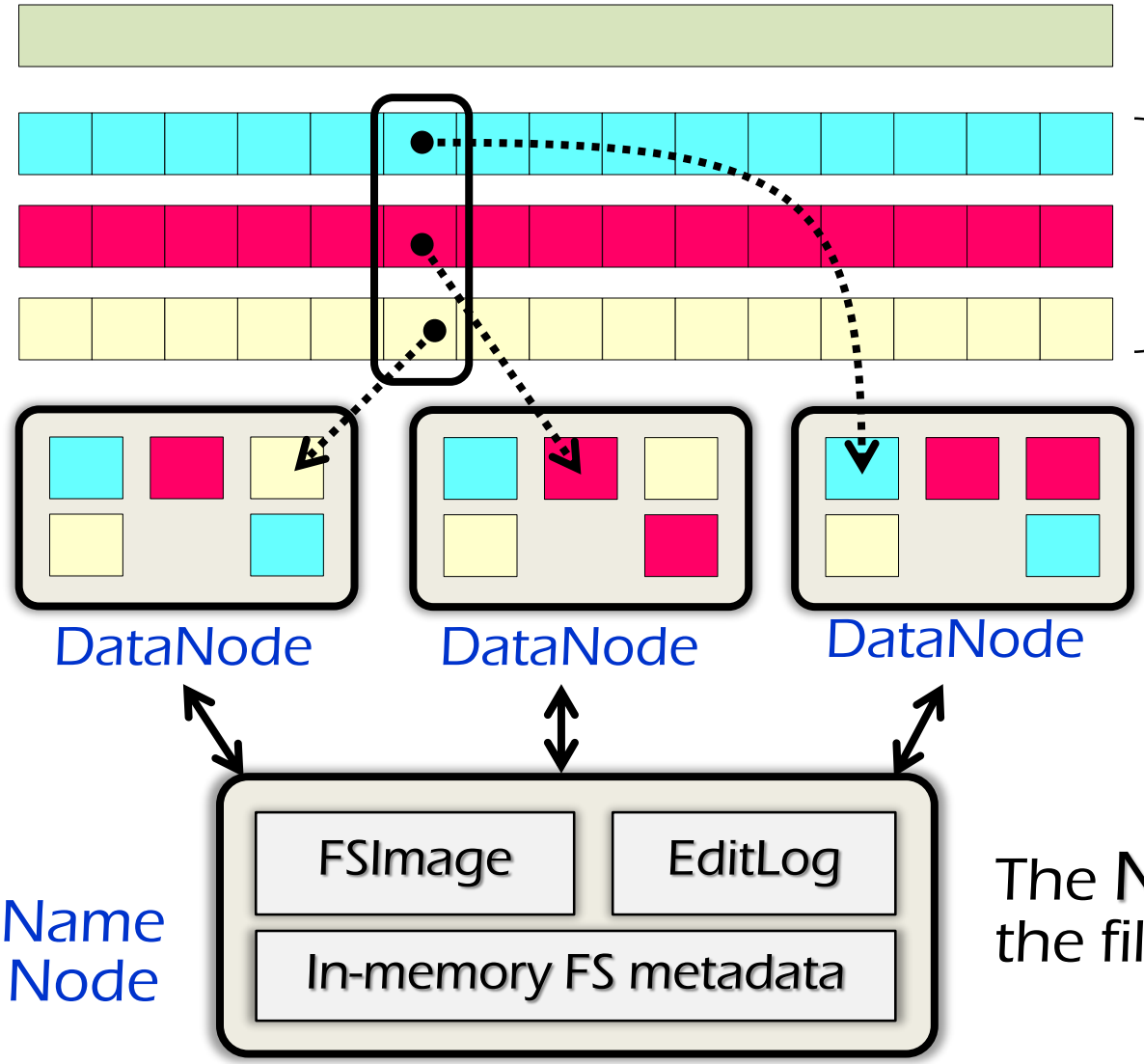
A File is made of 64MB chunks

That are replicated for fault-tolerance

Chunks live on chunkservers

The master manages the file system namespace

HDFS Files



A File is made of 64MB blocks

That are replicated for fault-tolerance

Blocks live on DataNodes

The Namenode manages the file system namespace

THANKS