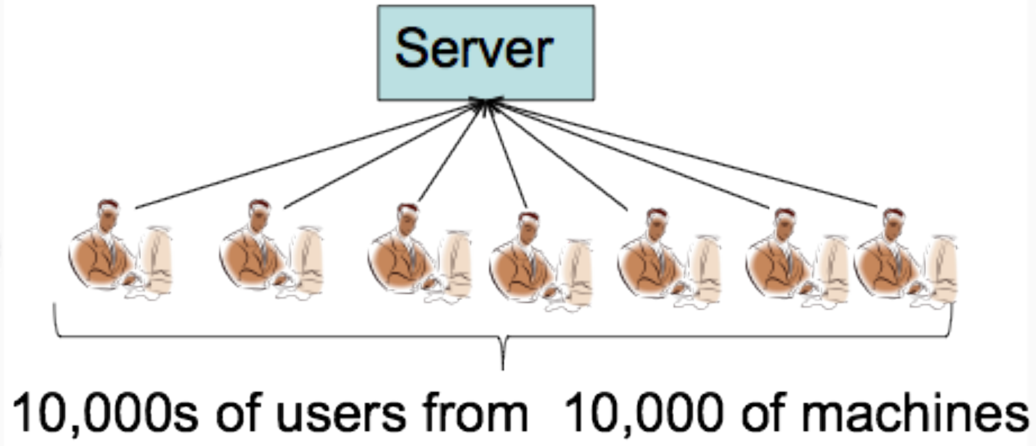
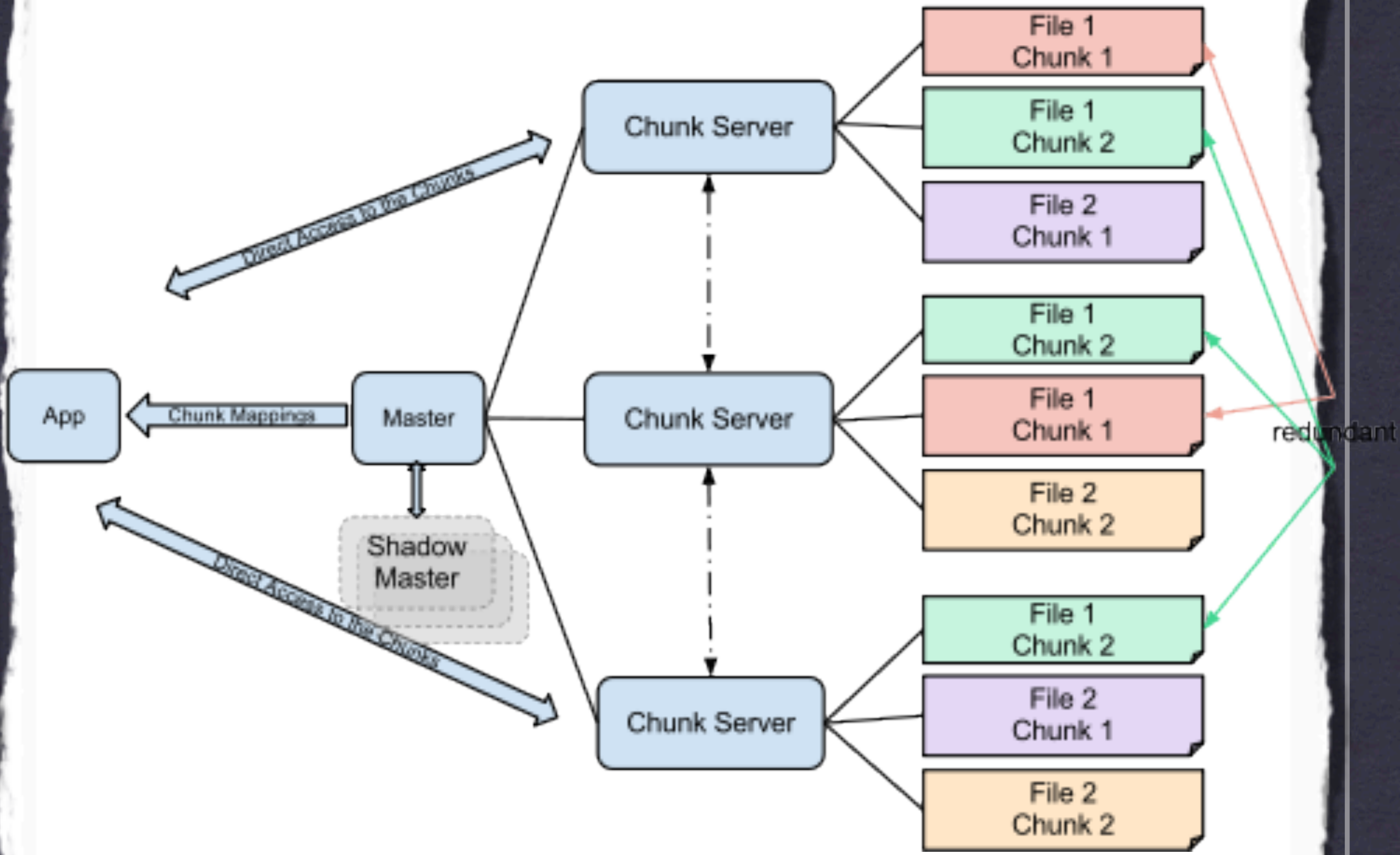




distributed systems

SEMESTER 2, 2020

DISTRIBUTED FILESYSTEMS

This lecture ...

- Distributed file systems - one of the first uses of distributed computing
  - challenges
  - general design considerations
- NFS (Networked File System)
- GFS (Google File System)
- HDFS (Hadoop Distributed File System)
- PFS (Parallel File System)

Definition of a DFS

- DFS: File system spread over multiple sites, and (possibly) distributed storage of files.
- Benefits
  - File sharing
  - Uniform view of system from different clients
  - Centralized administration
- Goals of a distributed file system
  - Network Transparency (access transparency)
  - Availability

Goals

- **Network (Access)Transparency**
  - Users should be able to access files over a network as easily as if the files were stored locally.
  - Users should not have to know the physical location of a file to access it.
- Transparency can be addressed through naming and file mounting mechanisms

Components of Access Transparency

- Location Transparency: file name doesn’t specify physical location
- Location Independence: files can be moved to new physical location, no need to change references to them. (A name is independent of its addresses.)
- Location independence $\rightarrow$ location transparency, but the reverse is not necessarily true.

Goals

- **Availability:**

    files should be easily and quickly accessible.

- The number of users, system failures, or other consequences of distribution shouldn’t compromise the availability.
- Addressed mainly through replication.

Challenges of DFS

- Heterogeneity (lots of different computers & users)
- Scalability
- Security
- Failures
- Concurrency
- Geographic distribution
- High latency

HOW CAN WE BUILD THIS?

How to start?

- Prioritized list of goals
  - performance, scale, consistency - which one do you care more about?
- We are scientists! Therefore we measure and design, and revise
- Workload-oriented design:
  - Measure characteristics of target workloads to inform the design

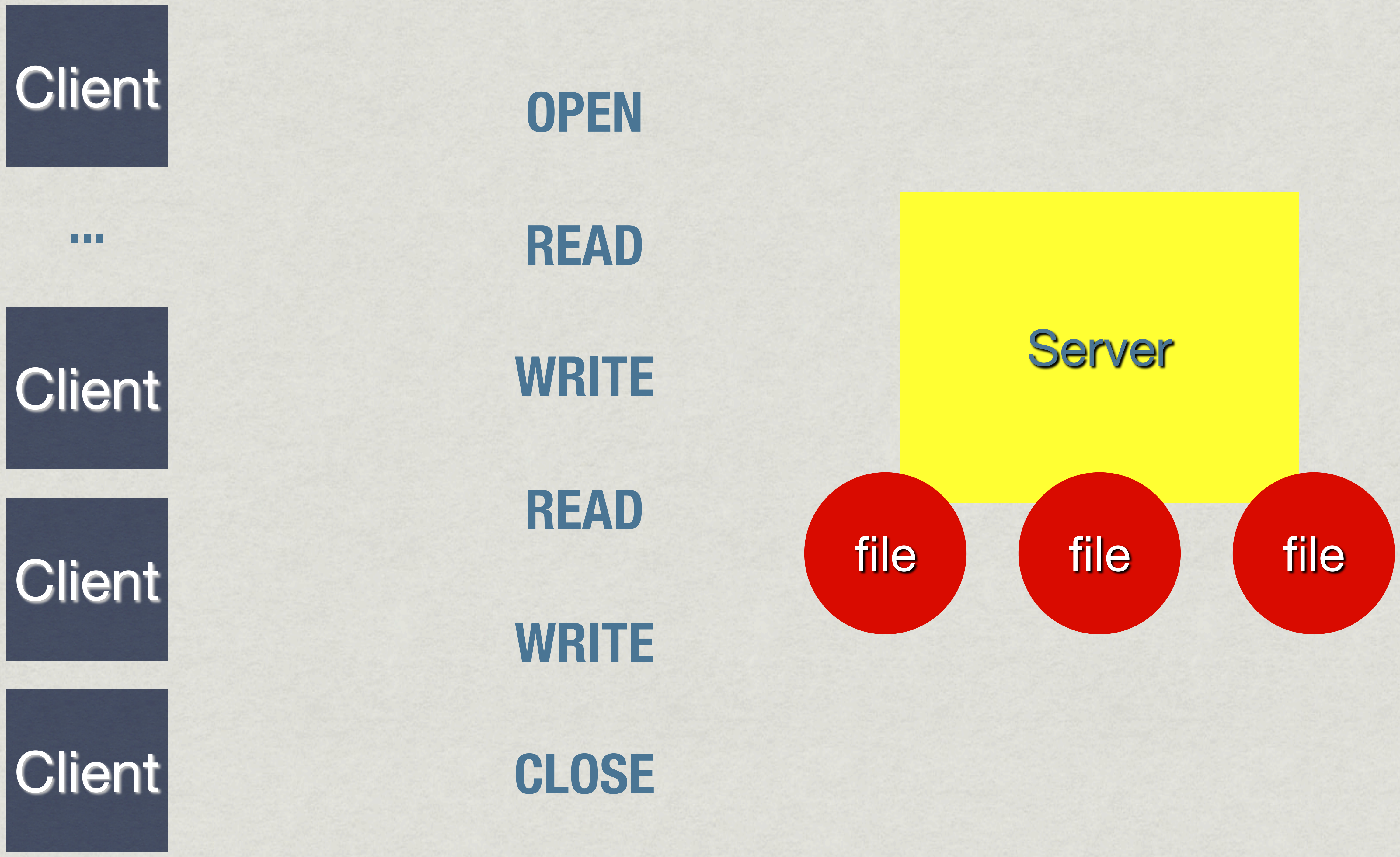
Workload Oriented Design

- User-oriented (NFS, AFS)
  - optimize how users use files
  - files are privately owned
  - not too much concurrent access
  - Sequential is common; reads more common than writes
- Program-oriented (GFS)
  - Focus on big-data workload: files are very very large
  - Failures are normal occurrences
  - Most files mutate by appending new data, not overwriting

Architectures

- Client-Server
  - Traditional; e.g. Sun Microsystem Network File System (NFS)
  - Cluster-Based Client-Server; e.g., Google File System (GFS)
- Symmetric
  - Fully decentralized; based on peer-to-peer technology
  - e.g., Ivy (uses a Chord DHT approach)

FS Interface



Directory operations

- Create file
- create directory
- rename file
- delete file
- delete directory

Naive DFS Design

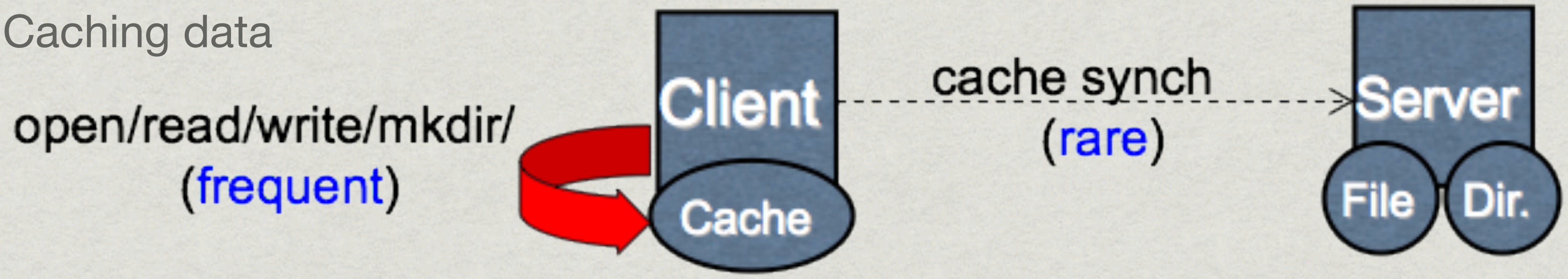
- Use RPC to forward *every* filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- **Good**: Same behavior as if both programs were running on the same local filesystem!
- **Bad**: Performance will stink. Latency of access to remote server often much higher than to local memory.
- **Ugly**: server gets hammered! No scalability.
- Example: WebDAV = filesystem over HTTP, minor extensions.

LESSON 1: NEEDING TO HIT THE SERVER FOR EVERY DETAIL IMPAIRS PERFORMANCE AND SCALABILITY.

QUESTION: HOW CAN WE AVOID GOING TO THE SERVER FOR EVERYTHING? *What* CAN WE AVOID THIS FOR? WHAT DO WE LOSE IN THE PROCESS?

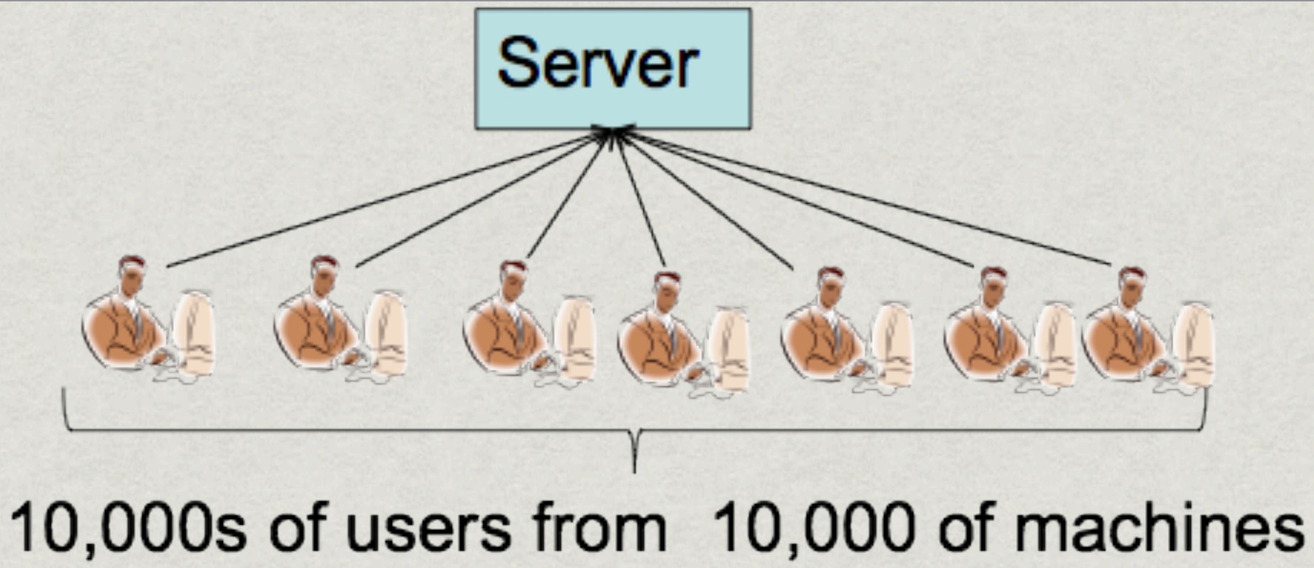
Solution: Caching

- Lots of systems problems can be solved in two ways:
    - Adding a level of indirection
        - *"All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem."* -- D. Wheeler
    - Caching data



- Questions:
    - What do we cache?
    - If we cache... doesn't that risk making things inconsistent?

Sun NFS

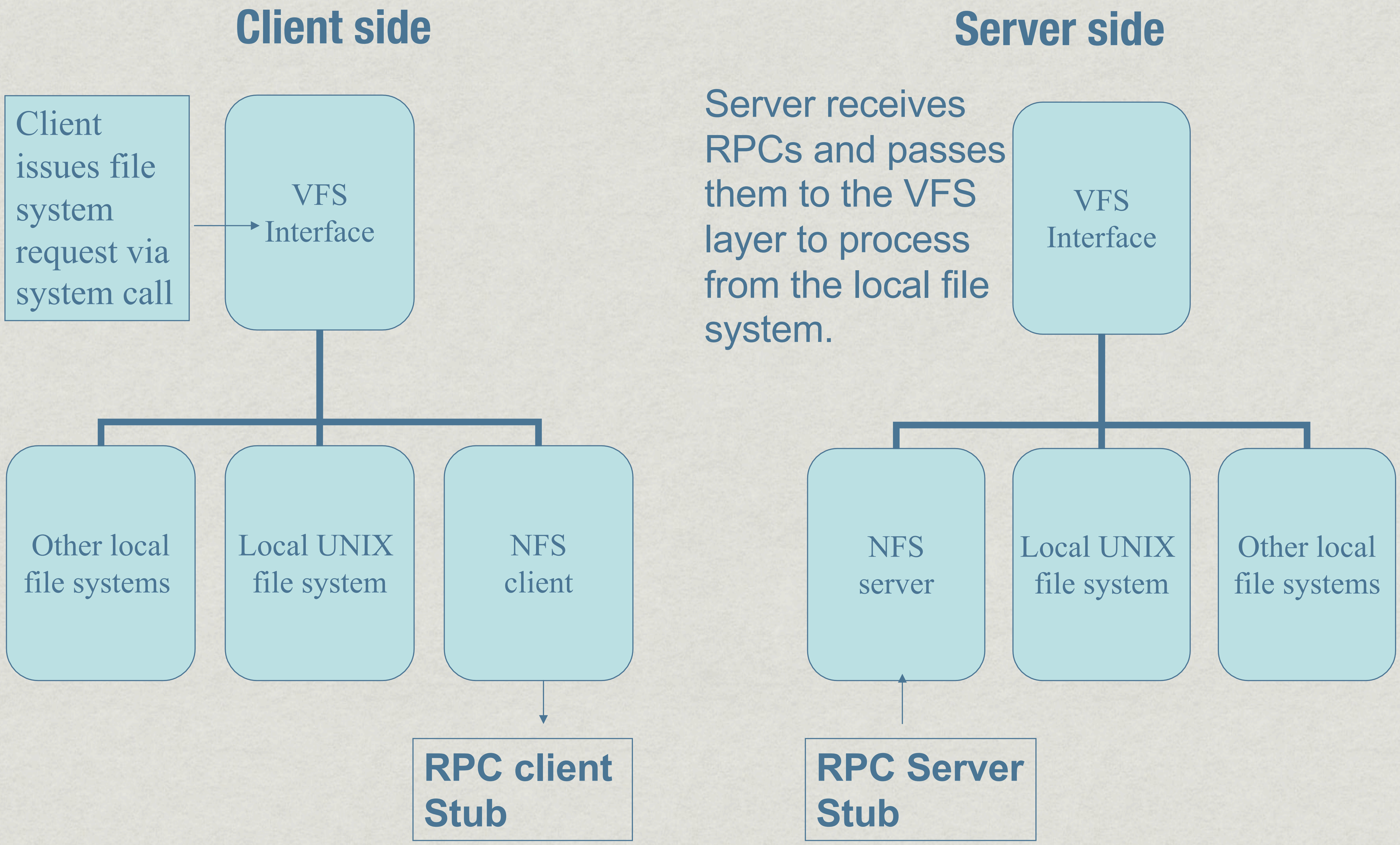


- Networked file system with a single-server architecture
- Goals
  - **consistent namespace** for files across computers
  - allow **authorized** users to access their files from any computer
  - location transparency
- Very 1990's client-server approach (NFSv3). Huge # of server, 10^1-10^4 "small" clients.
- NFSv4 introduced in 2003 made significant changes

Access Model

- Clients access the server transparently through Virtual File System (VFS)
    - Client-side caching may be used to save time and network traffic
- The NFS client communicates with the server using RPCs (with parameters “marshelled”)
- At the server: an RPC server stub receives the request, “un-marshalls” the parameters & passes them to the NFS server, which creates a request to the server’s VFS layer.
- The VFS layer performs the operation on the local file system and the results are passed back to the client.

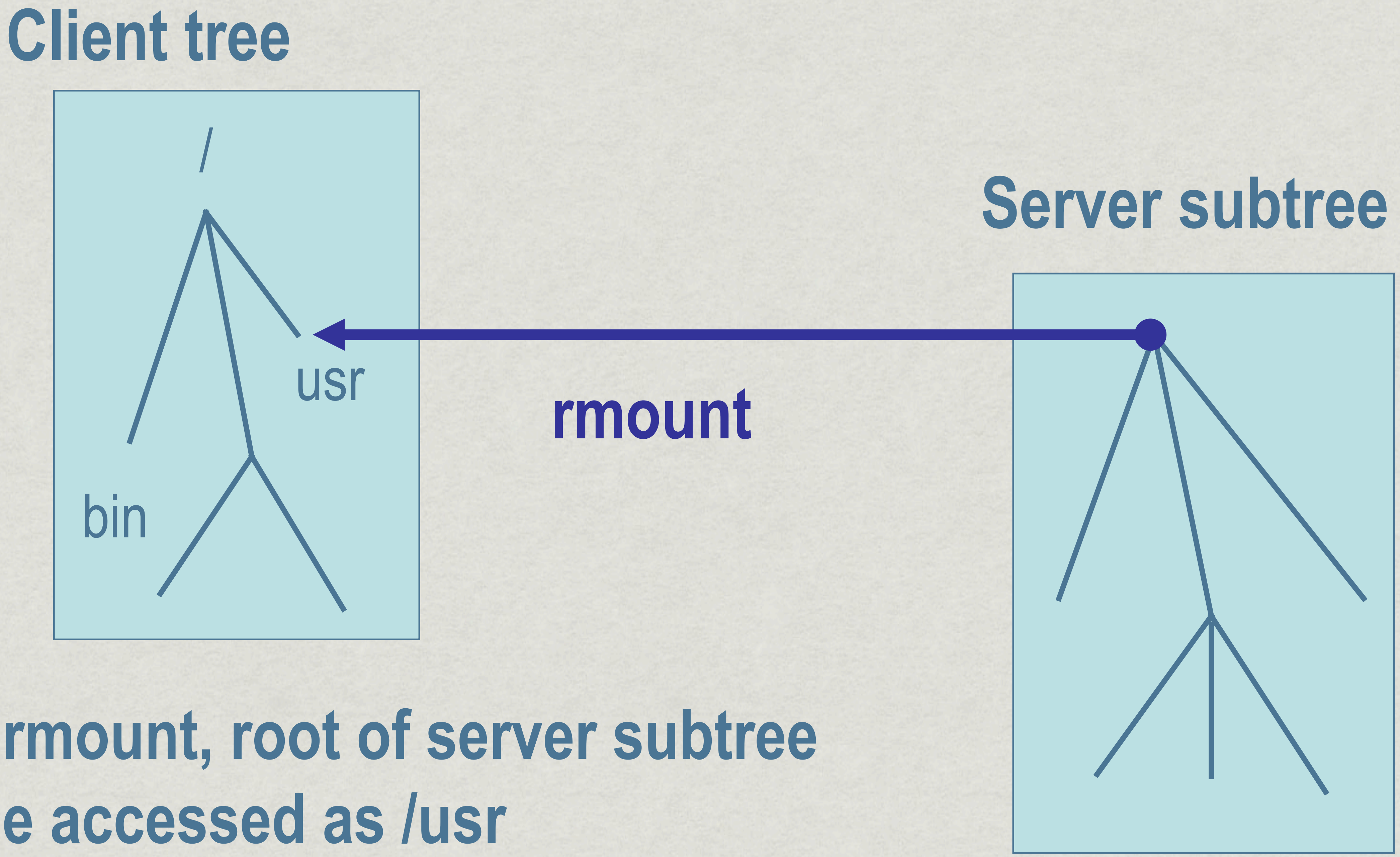
Client to NFS Server Access



NFS as a Stateless Server

- NFS servers historically did not retain any information about past requests.
- Consequence: crashes weren’t too painful
  - If server crashed, it had no tables to rebuild – just reboot and go
- Disadvantage: client has to maintain all state information; messages are longer than they would be otherwise.
- NFSv4 is state*ful*

Remote mount



After rmount, root of server subtree can be accessed as /usr

Advantages/Disadvantages

- Stateless Servers
  - Fault tolerant
  - No open/close RPC required
  - No need for server to waste time or space maintaining tables of state information
  - Quick recovery from server crashes
- Stateful Servers
  - Messages to server are shorter (no need to transmit state information)
  - Supports file locking
  - Supports idempotency (don’t repeat actions if they have been done)

Caching

- Cache file blocks, file headers, etc., at both clients and servers.
- Advantage: No network traffic if open/read/write/close can be done locally.
- But: failures and cache consistency.
  - NFS trades some consistency for increased performance

Caching problem #1: Failures

- **Server crashes**
    - So... what if client does `seek() ;  /* SERVER CRASH */; read()`
        - If server maintains file position, this will fail.
        - same for open(), read()
- **Lost messages**: what if we lose acknowledgement for delete("foo")
    - And in the meantime, another client created foo again?
- **Client crashes**
    - Might lose data in client cache

NFS's Solutions

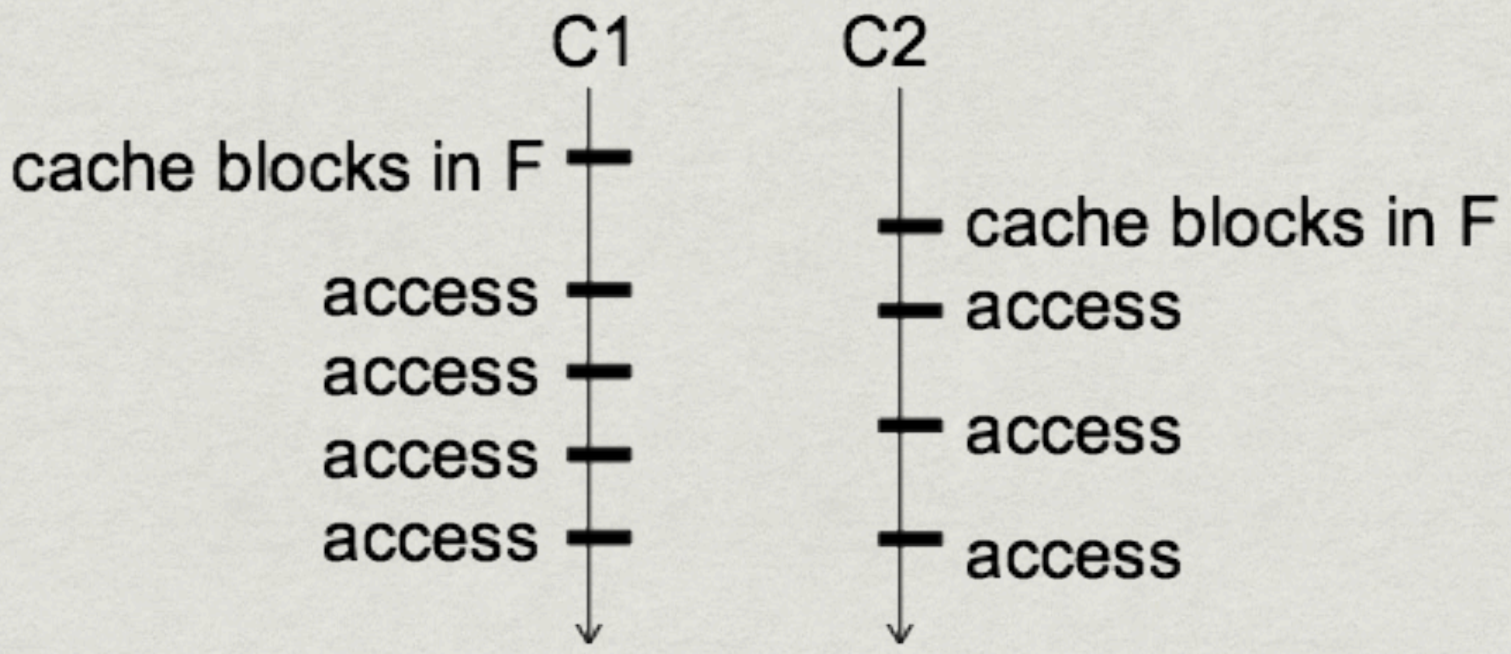
- **Stateless design**
  - Write-back caching: When file is closed, all modified blocks sent to server. write() is mostly local but close() does not return until bytes safely stored.
  - Stateless protocol: requests specify exact state. read() -> read([position]). no seek on server.
- Operations are ***idempotent***
  - Have the same effect no matter how many times they are repeated
  - How can we ensure this? Unique IDs on files/directories. It's not `delete("foo")`, it's `delete(1337f00f)`, where that ID won't be reused.

Design tip

- *Idempotency* is a useful property when building reliable systems.
- When an operation can be issued more than once, it is much easier to handle failure of the operation
    - you can just retry it.
- If an operation is not idempotent, life becomes much more difficult!

Caching problem #2: Consistency

- If we allow client to cache parts of files, file headers, etc.
  - What happens if another client modifies them?



Solution: Weak Consistency

- NFS flushes updates on close()
- How does other client find out?
- NFS's answer: It checks periodically.
  - This means the system can be inconsistent for a few seconds: two clients doing a read() at the same time for the same file could see different results if one had old data cached and the other didn't.

Design choice

- Clients can choose a stronger consistency model: *close-to-open* consistency: any changes made by clients are flushed to the server on closing the file, and a cache revalidation occurs when the file is re-opened.
  - How? Always ask server for cache validation before open()
  - Trades a bit of scalability for better consistency

What about multiple writes?

- NFS provides no guarantees at all!
- Might get one client’s writes, other client’s writes, or a mix of both!

NFS and Failures

- You can choose -
  - retry until things get through to the server
  - return failure to client
- Most client apps can’t handle failure of close() call. NFS tries to be a transparent distributed filesystem -- so how can a write to local disk fail? And what should the application do, anyway?
- Usual option: hang for a long time trying to contact server

Summary (1)

- NFS is a distributed file system for multiple clients to access files on a single server
- Provides transparent, remote file access
- Simple, portable, *really popular*
  - (it's gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

Cluster-Based DFS

- Some cluster-based systems organize the clusters in an application specific manner
- For file systems used primarily for parallel applications involving a large volume of data, the data in a file might be striped across several servers so it can be read in parallel.
- Or, it might make more sense to partition the file system itself – some portion of the total number of files are stored on each server.
- For systems that process huge numbers of requests; e.g., large data centers, reliability and management issues take precedence.
  - *e.g.*, Google File System

Google File System

- GFS uses a cluster-based approach implemented on ordinary commodity Linux boxes (not high-end servers).
- There are about 30 clusters world-wide http://www.slideshare.net/ultradvorka/google-cluster-innards , referenced 4/17/12
- Slides based on the Google SOSP 2003 paper:
  - *The Google File System*, S. Ghemawat, H. Gobioff, S-T Leung
  - some details might be out of date
  - Colossus: Successor to the Google File System (GFS) (2010)

Design Constraints

- Machine failures are the norm
  - 1000s of components
  - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
  - Monitoring, error detection, fault tolerance, automatic recovery must be integral parts of a design
  - Jeff Dean (2008): In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; **50% chance that the cluster will overheat**
- Big-data workloads
  - Search, ads, web analytics, Map/Reduce
    Now days we do (streaming videos, real-time document editing, etc)

Workload Characteristics

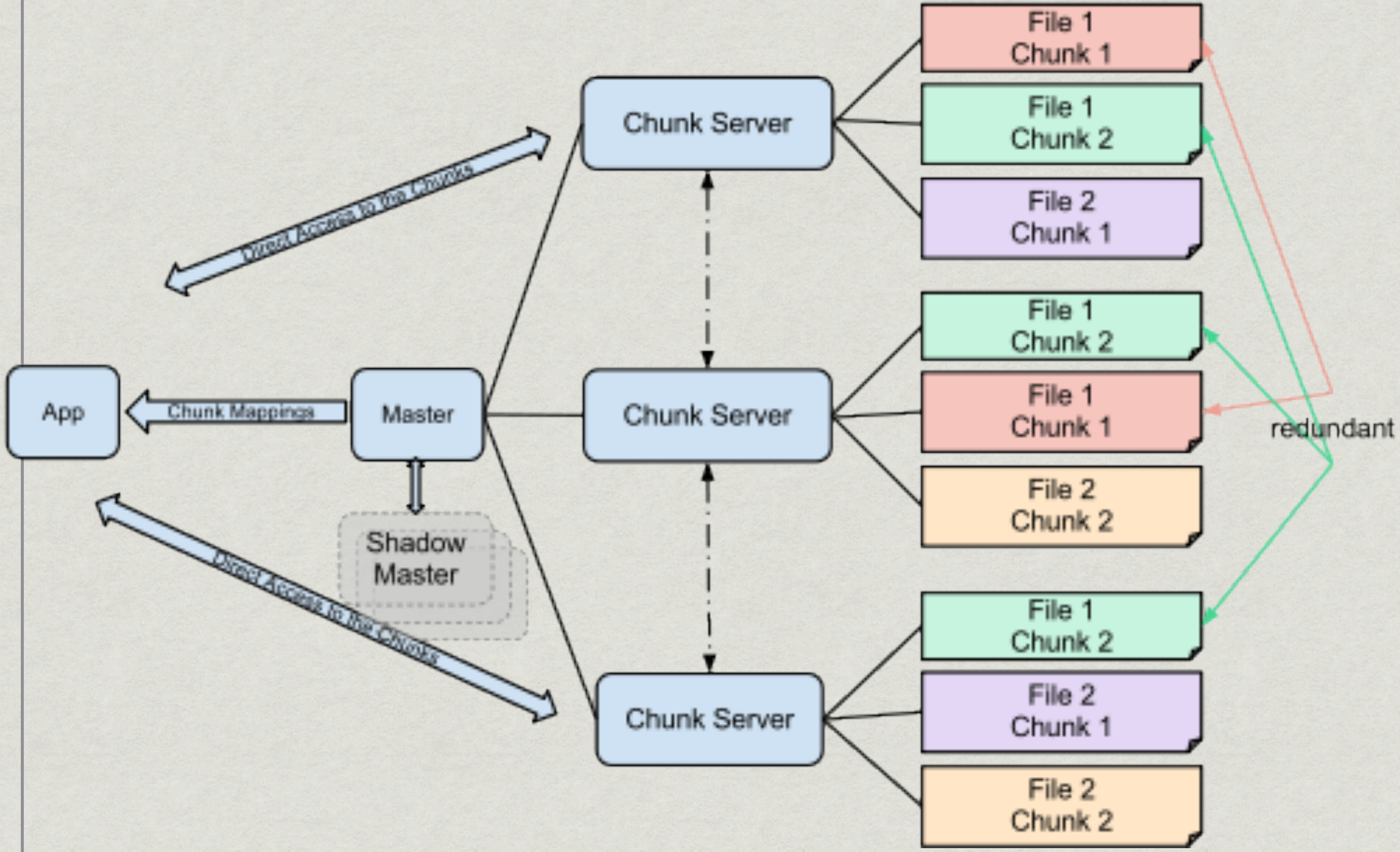
- Files are huge by traditional standards
    - Multi-GB files are common
- Most file updates are appends
    - Random writes are practically nonexistent
    - Many files are written once, and read sequentially
- High bandwidth (throughput with concurrency) is more important than latency
    - Lots of concurrent data accesses, e.g. multiple crawler workers updating the index file
- GFS' design is geared toward apps' characteristics
    - Stream-like data connections between processing phases; think Map-Reduce.
    - and Google apps have been geared toward GFS

Additional operation

- Record append
  - Frequent operation at Google
  - Merging results from multiple machines in one file (Map/Reduce)
  - Using file as producer - consumer queue
  - Logging user activity, site traffic
  - Order doesn’t matter for appends, but atomicity and concurrency matter

GFS Organization

- A GFS cluster
  - One master: Maintains a mapping from file name to chunks & chunks to chunk servers
  - Many chunkservers - accessed by many clients
- A file
  - Divided into fixed-sized chunks
  - Labeled with 64-bit unique global IDs (called handles)
  - Stored at chunkservers
  - 3-way replicated across chunkservers
  - Master keeps track of metadata: which chunks belong to which files



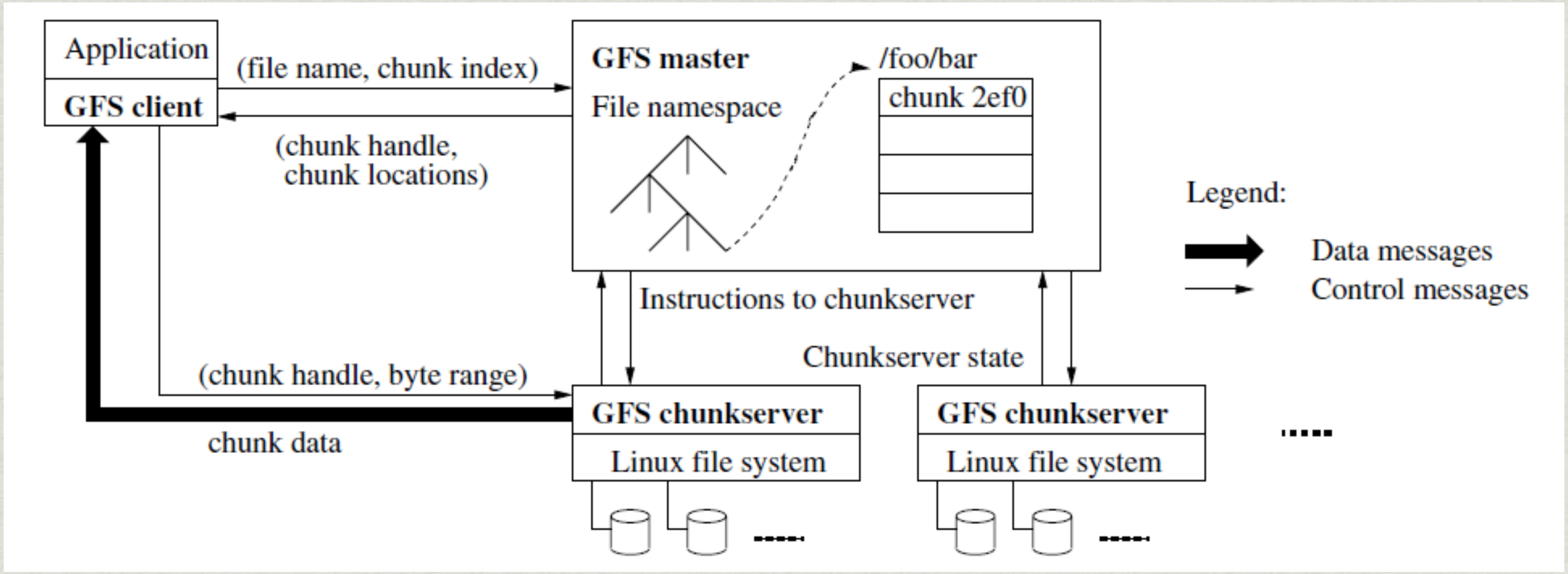
App
Chunk Mappings
Master
Shadow Master
Direct Access to the Chunks
Direct Access to the Chunks
Chunk Server
Chunk Server
Chunk Server
File 1 Chunk 1
File 1 Chunk 2
File 2 Chunk 1
File 1 Chunk 2
File 1 Chunk 1
File 2 Chunk 2
File 1 Chunk 2
File 2 Chunk 1
File 2 Chunk 2
redundant

Basic Operations

1. Client retrieves metadata from master
2. Read/write data from client to chunkserver

- Master really not that involved: not usually a bottleneck

In detail ...


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

Chunks

- Similar to FS blocks, but bigger!
- 64 MB instead of 512B – 64kB
- **Advantages** of a large chunk size?
- **Disadvantage**?

Chunks Pros vs Cons

- **Advantages** of a large chunk size?
  - Less load on the server, especially metadata
  - suitable for big data
  - sustains large bandwidth
- **Disadvantage**?
  - What if small files are more frequent than initially believed?
    - E.g. downloading the latest email
  - Support for random writes?

GFS Master

- The client’s first point of contact
- One process running on a separate machine
  - at a later stage shadow masters were added for fault-tolerance
- Stores metadata in memory (fast!)
  - Using a RAM Disk, 64 bytes for 64MB of data
- Metadata types
  - File and chunk namespaces (hierarchical and flat respectively)
  - File-to-chunk mappings
  - Location of chunk’s replicas

Master <-> Chunkserver communication

- Heartbeat messages (regular communication)
  - Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunks does chunkserver store?
- Master sends instructions:
  - Delete chunk, create chunk
  - Replicate and start serving a particular chunk

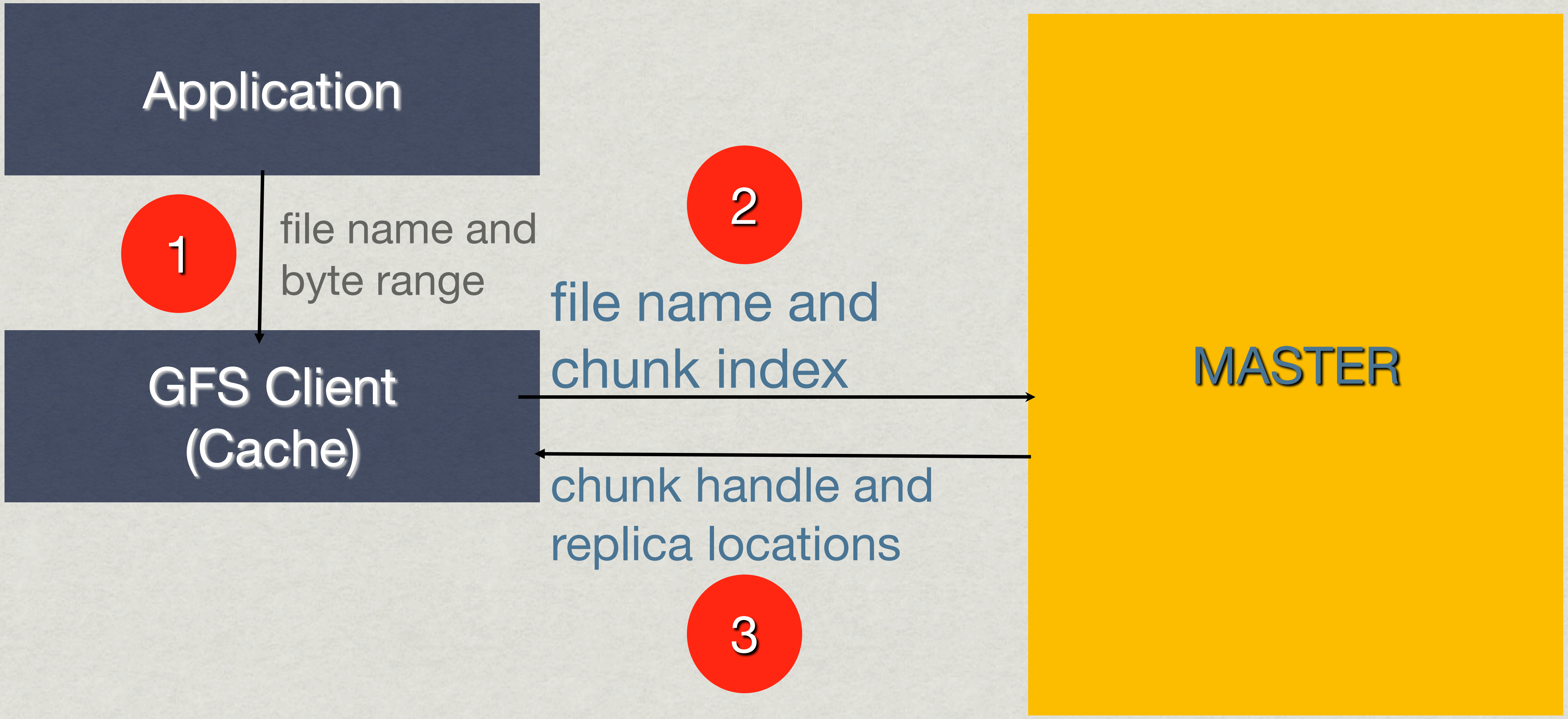
Chunk Locations

- No persistent location
  - Master polls chunkservers at startup
  - Use heartbeat messages to monitor servers
- **Advantages? (what if master dies?)**
- **Disadvantages? (what if master dies?)**

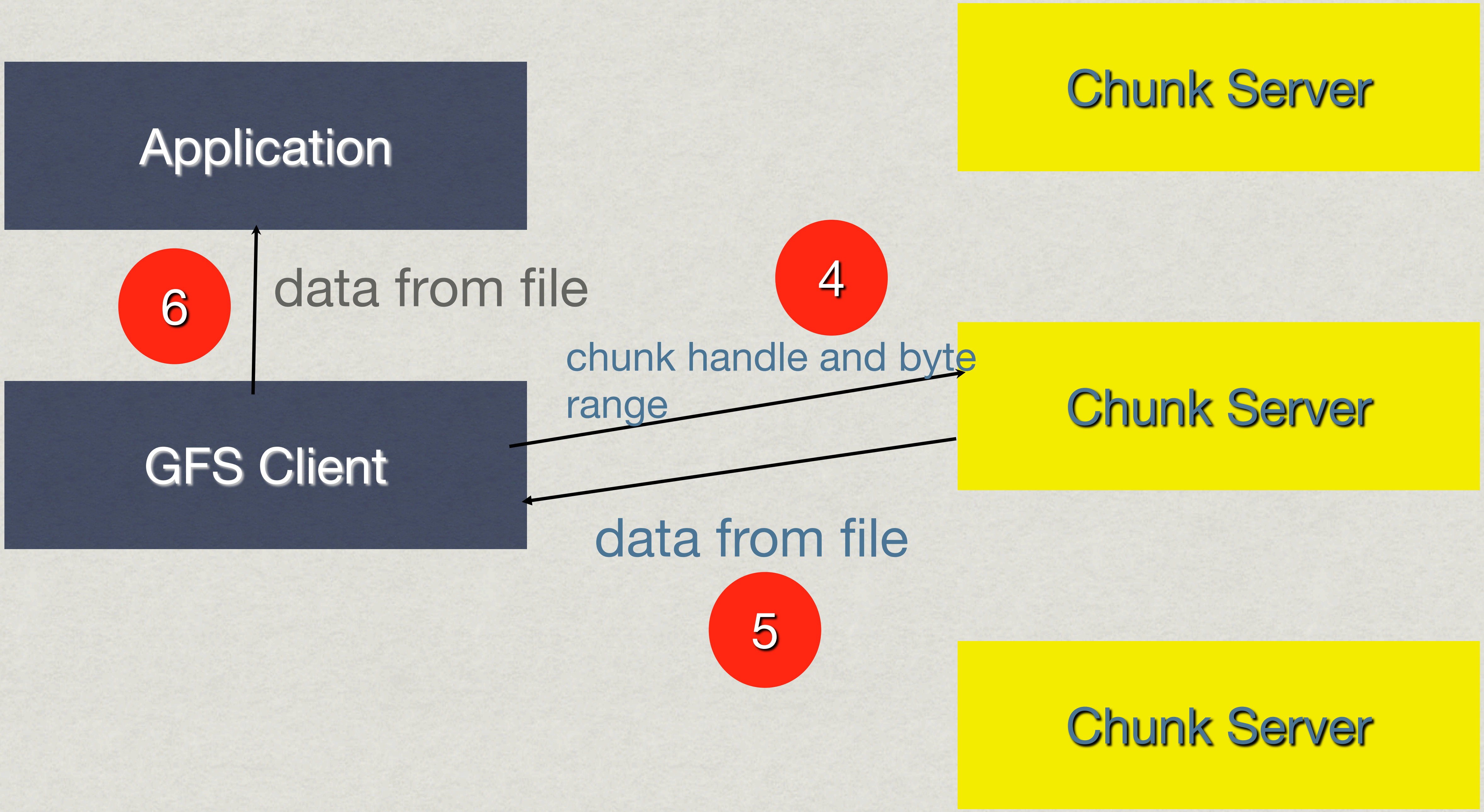
Chunk Locations

- No persistent location
  - Master polls chunkservers at startup
  - Use heartbeat messages to monitor servers
- **Advantages?**
  - **if master dies, can recover chunks from chunkservers**
- **Disadvantages?**
  - **if master dies, it takes a loooong time to restart**

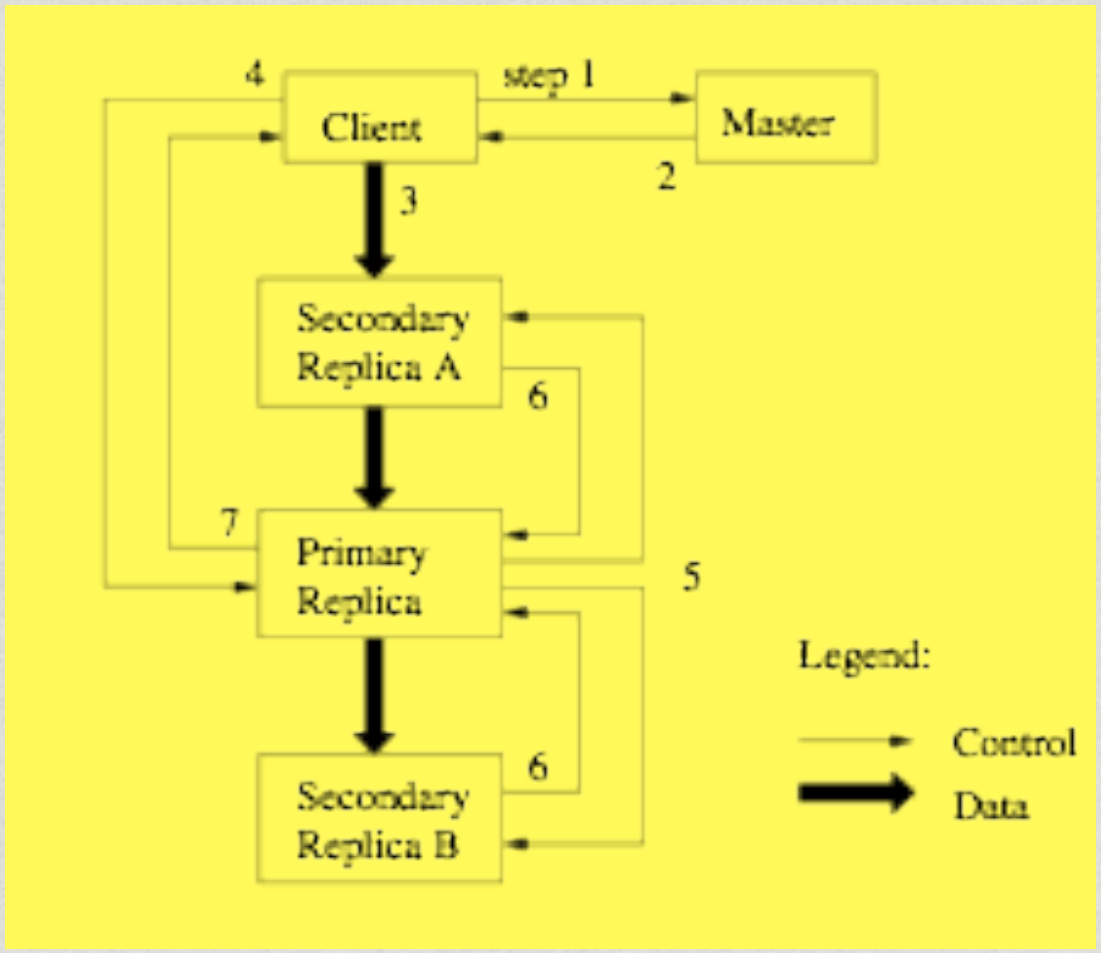
Read Protocol - Part 1



Read Protocol - Part 2



Write Protocol



1. Client queries master which chunkserver holds the current lease for the chunk + the locations of the other replicas.

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 dies

3. The client pushes the data to all the replicas.

4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary.

5. Primary forwards write request to all secondaries

6. Secondaries all reply to primary

7. Primary replies to client. Any errors are reported to the client

Replica placement

- Hundreds of chunkservers distributed across many many machine racks
- Chunk replica placement needs to
  - maximize data reliability and availability
  - maximize network bandwidth utilization
- Not sufficient to spread replicas across machines
  - need to spread across racks to ensure replicas survive even if an entire rack is down
- Chunks are re-replicated as soon as the number of replicas falls below a goal
  - need to place replicas on chunkservers with below average disk utilization

Aside – Policy vs Mechanism

- Mechanism: code that implements concrete actions
- Policy: code that specifies where/how/why something is to occur
- Separation is good: cleaner designs, easier to change and experiment/test
- Examples
    - Chunk replica placement
    - Network retries
    - Cache replacement

Summary (2)

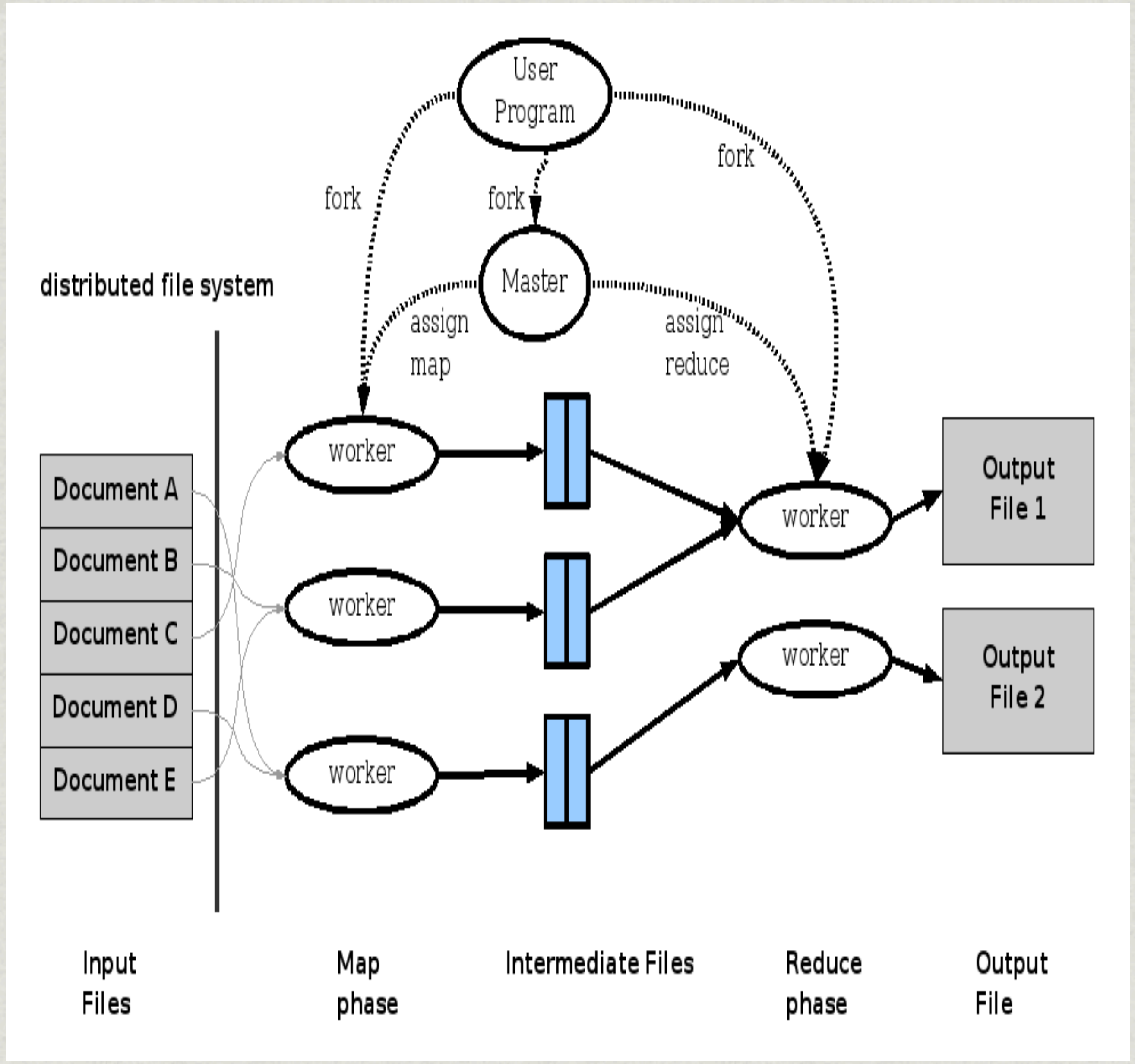
- GFS is a distributed file system that supports search service and other large data-intensive applications
- High throughput by decoupling control and data
- Supports massive data sets and concurrent appends
- Design overview
  - Single Master - centralized management
  - Files stored as chunks with a fixed size of 64MB each.
  - Reliability through replication - each chunk is replicated across 3 (or more) chunk servers
  - No client side caching - due to large size of data sets
  - Interface - create, delete, open, close, read, write, snapshot (copy-on-write), record append (concurrent atomic append support)

Hadoop Distributed File System (HDFS)

Motivation

- Leverage GFS to schedule a map task on a machine that contains a replica of the corresponding input data.
- Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

MapReduce: Execution overview

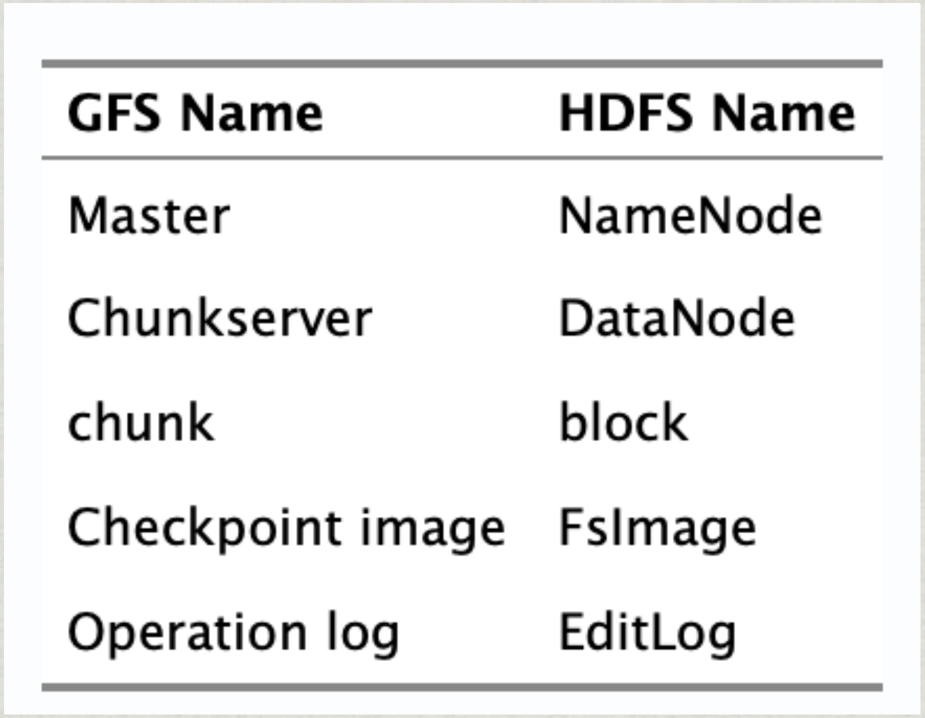


http://www.cs.cmu.edu/~srini/15-440-all/2015.Fall/lectures/18-hdfs-gfs.ppt

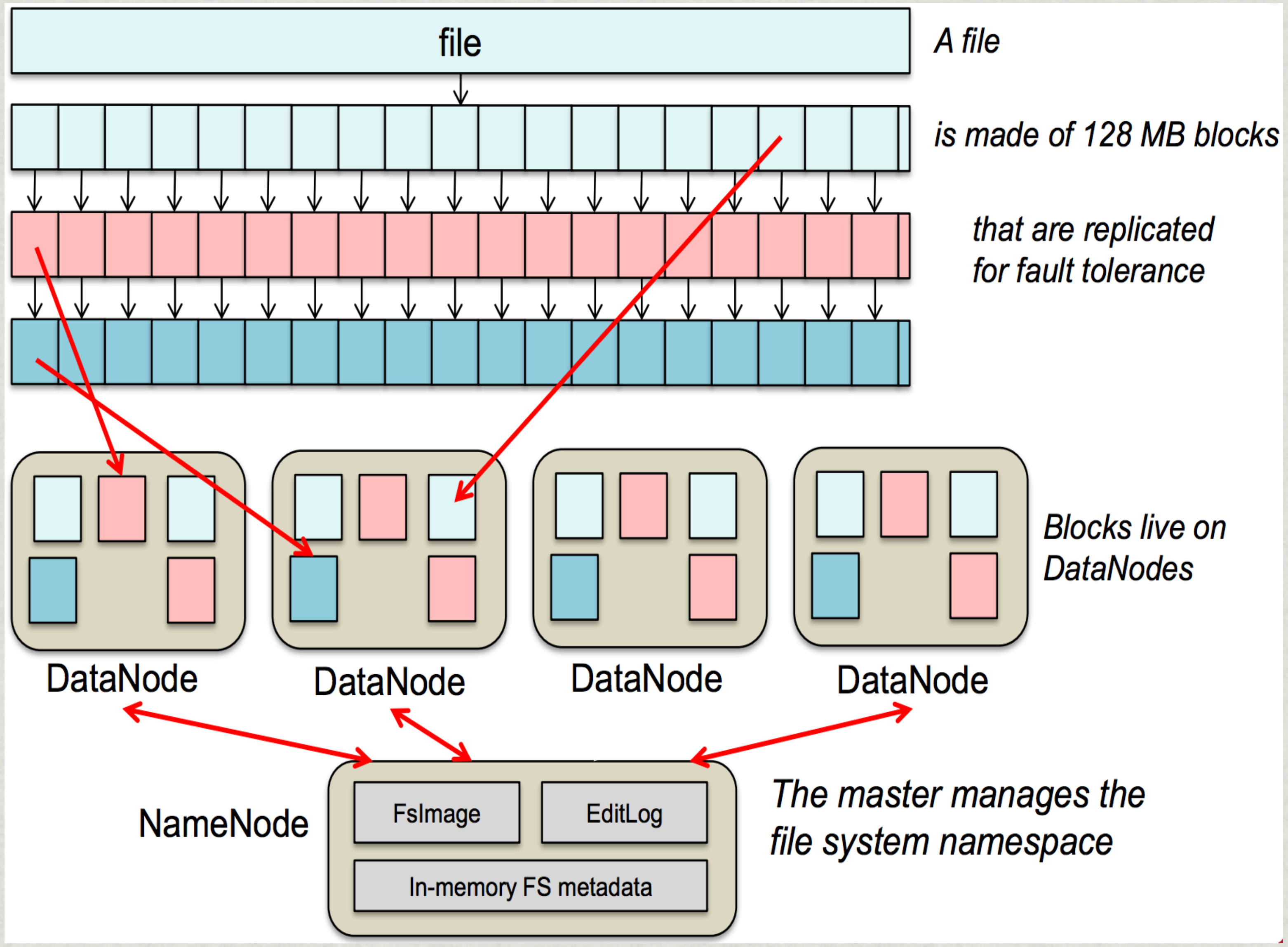
Hadoop Distributed File System (HDFS)

Architecture

- Same overall architecture as GFS, with different terminology



GFS Name	HDFS Name
Master	NameNode
Chunkserver	DataNode
chunk	block
Checkpoint image	FsImage
Operation log	EditLog



- File system as user-level libraries rather than a kernel module under VFS -> Applications have to be compiled to incorporate these libraries

https://www.cs.rutgers.edu/~pxk/417/notes/16-dfs.html

Hadoop Distributed File System (HDFS)

- DataNodes:
  - store blocks
  - handle read/write(=append) requests,
  - allocate and delete blocks
  - accept commands to replicate blocks on another DataNode.
- NameNode
  - store file system info and updates
  - handle file/directory open, close, rename, move
  - choose DataNodes to host replicas (rack & datacenter aware)
- DataNode $\xrightarrow[\text{block report}]{\text{heartbeat msg}}$ NameNode

Fault Tolerance in Distributed File Systems

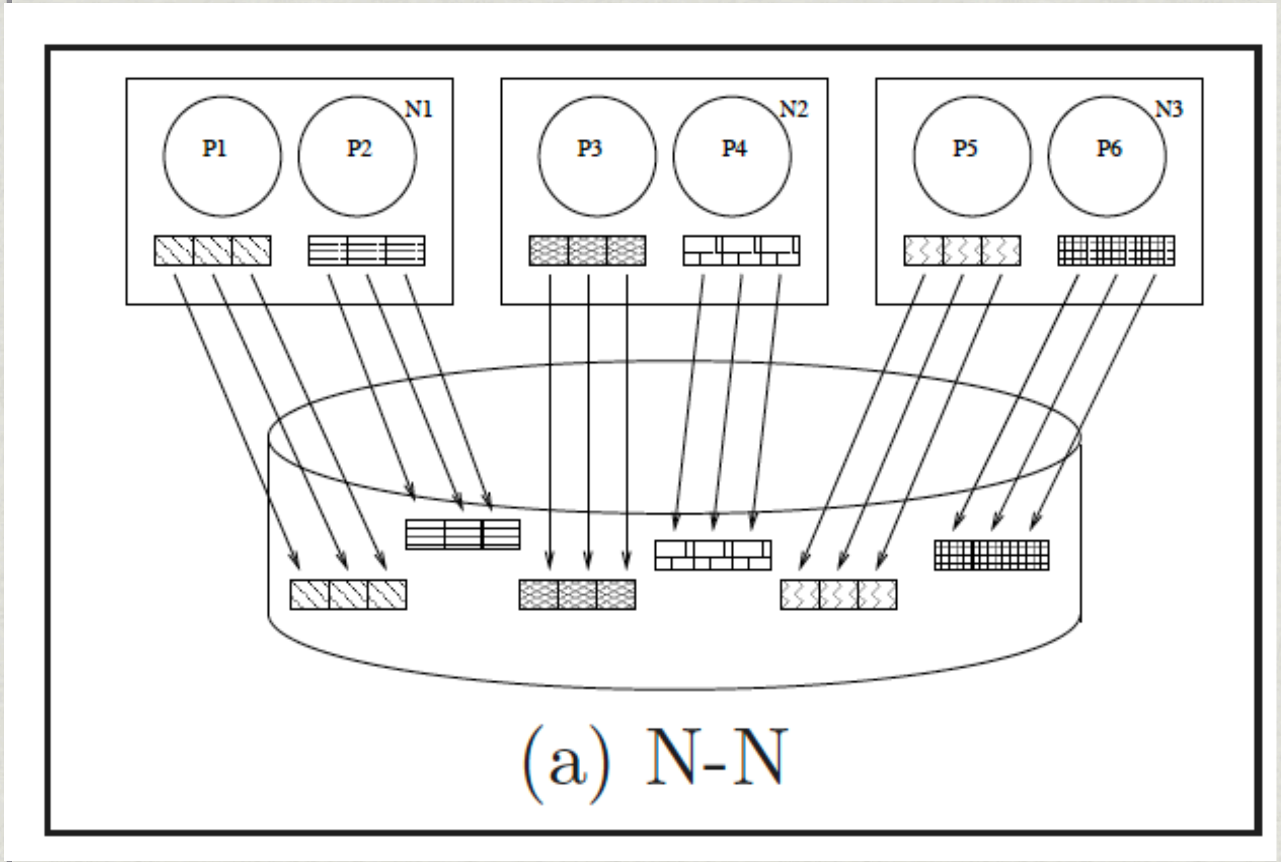
- Many options...
  - Do nothing, e.g. NFS
  - Hot, consistent replicas (every change affects multiple servers in case one dies)
    - e.g. GFS, HDFS
  - Consistent snapshots - filesystem backup (beautiful but costly to make)

Parallel File Systems

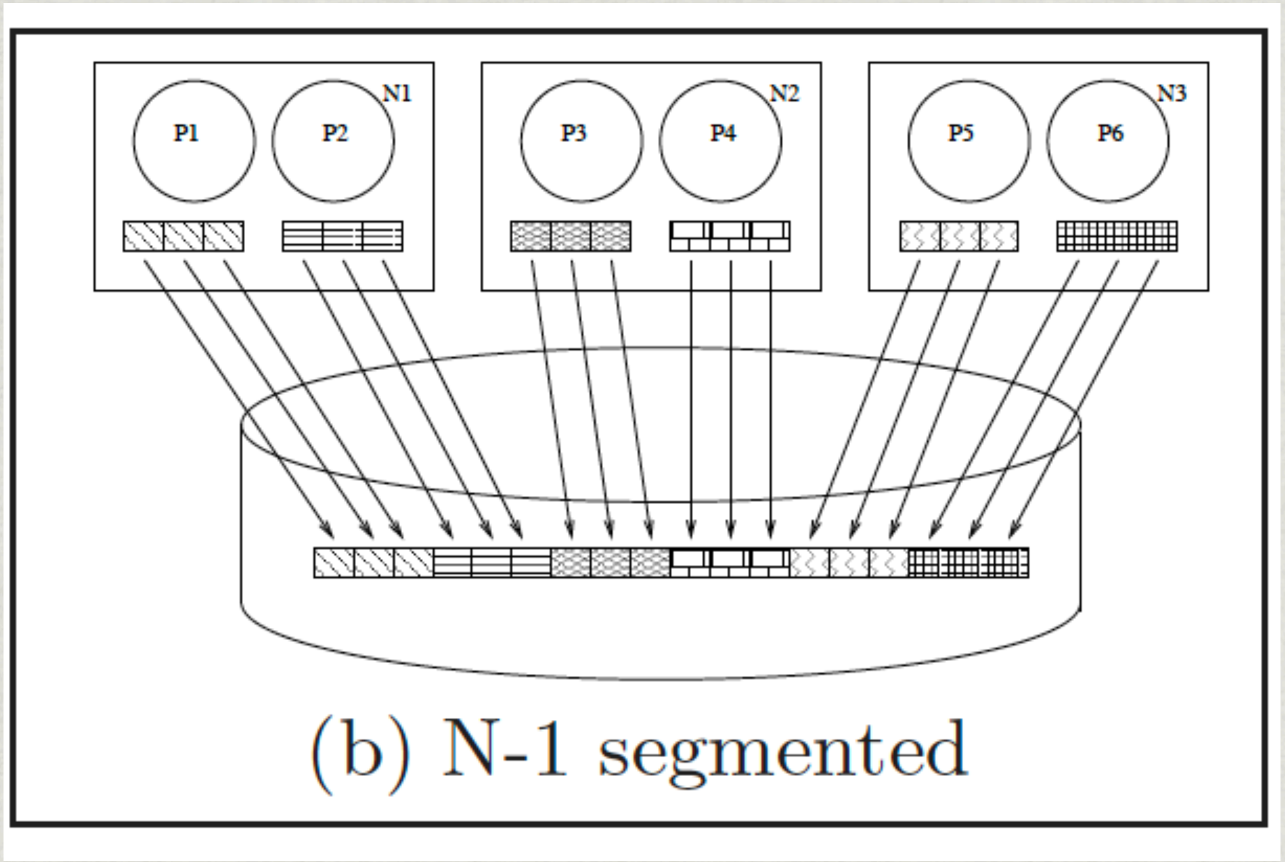
- Parallel I/O is commonly used by scientific applications for parallel processing to
    - Store numerical output from simulations for later analysis
    - Process more data than can fit in system memory and must page data in from disk
    - Checkpoint to files that save the state of an application from time to time (for recovery in case of system failure)
- Most scientific applications write large amounts of data in a \`checkpointing' or sequential 'append-only' way that does not overwrite previously written data or require random seeks throughout the file
    - though there are seeky workloads like graph traversal and bioinformatics problems
- Most HPC systems are equipped with a **Parallel File System** such as Lustre or GPFS that abstracts away spinning disks, RAID arrays, and I/O subservers to present the user with a simplified view of a single address space for reading and writing to files

https://www.nersc.gov/users/training/online-tutorials/introduction-to-scientific-i-o/?show_all=1

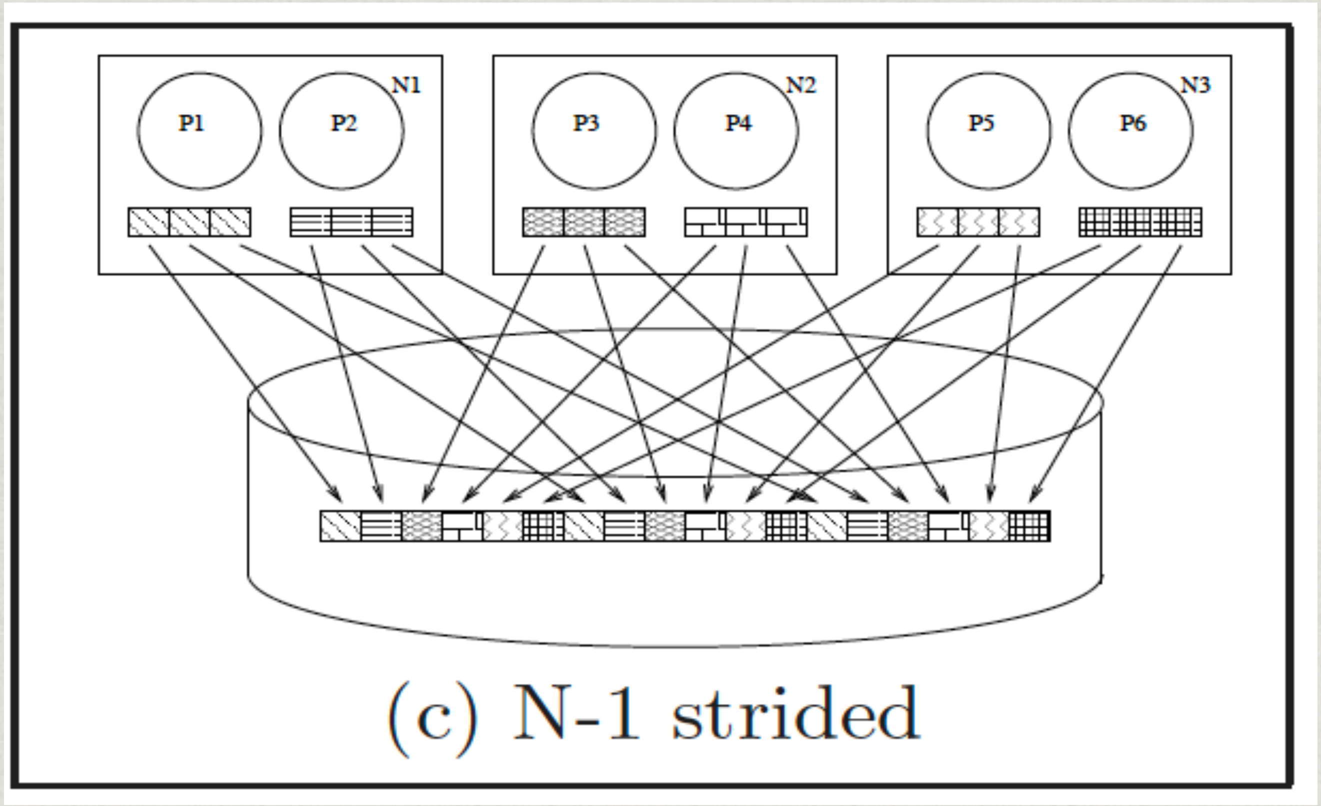
Common Checkpointing Patterns



(a) N-N



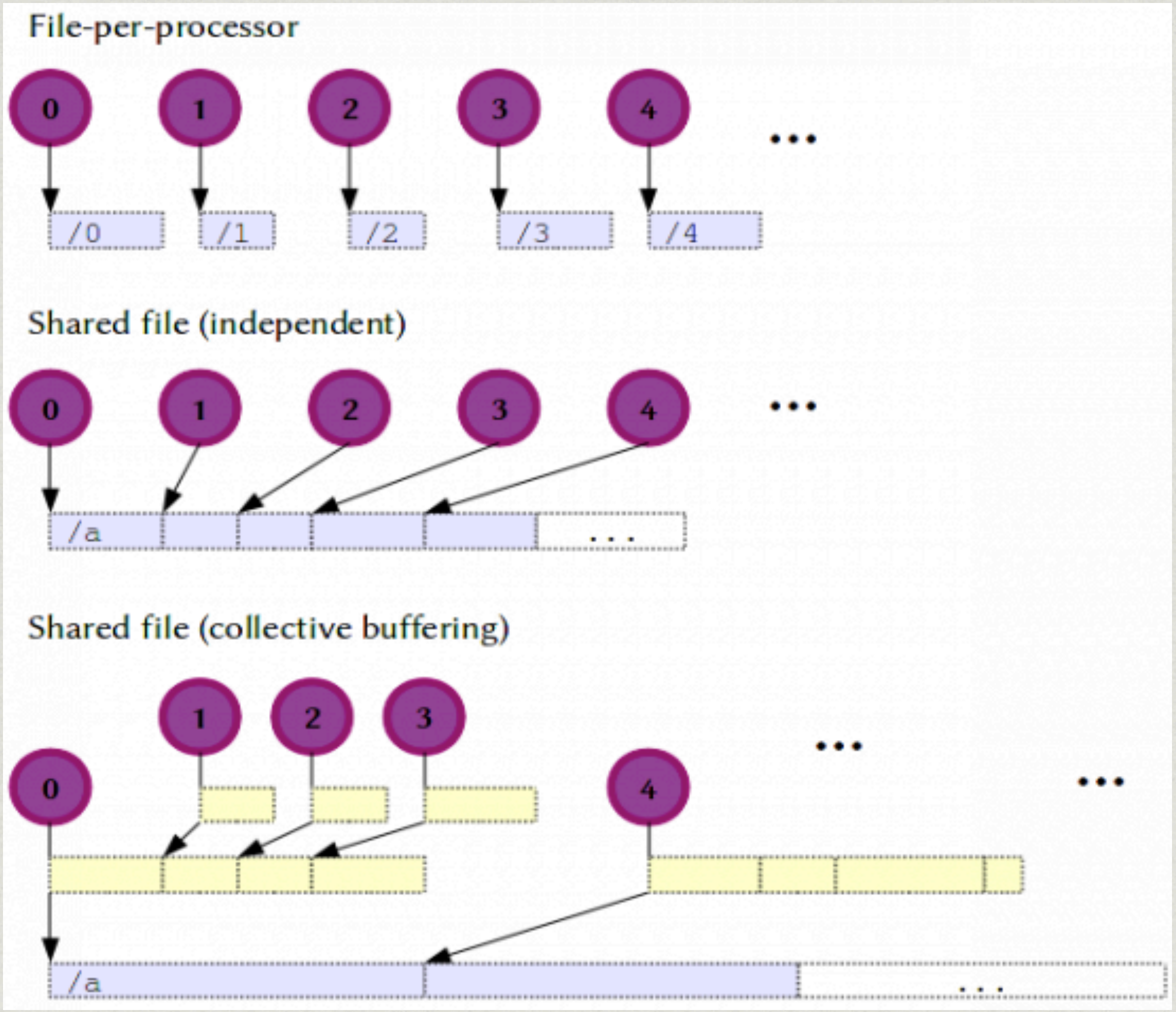
(b) N-1 segmented



(c) N-1 strided

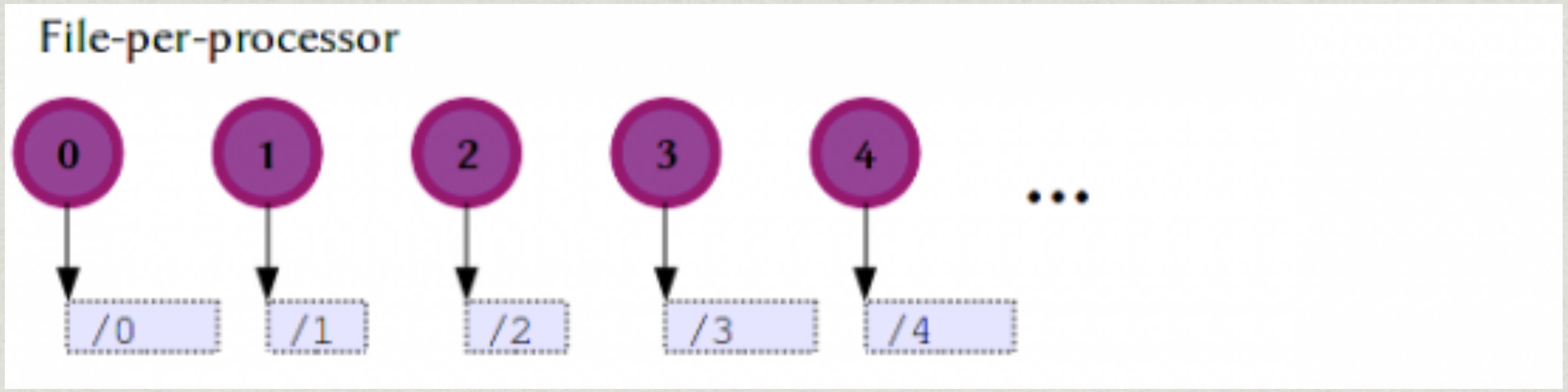
https://ieeexplore.ieee.org/document/6375580 (SC'09)

Common Methods for Access Data in Parallel



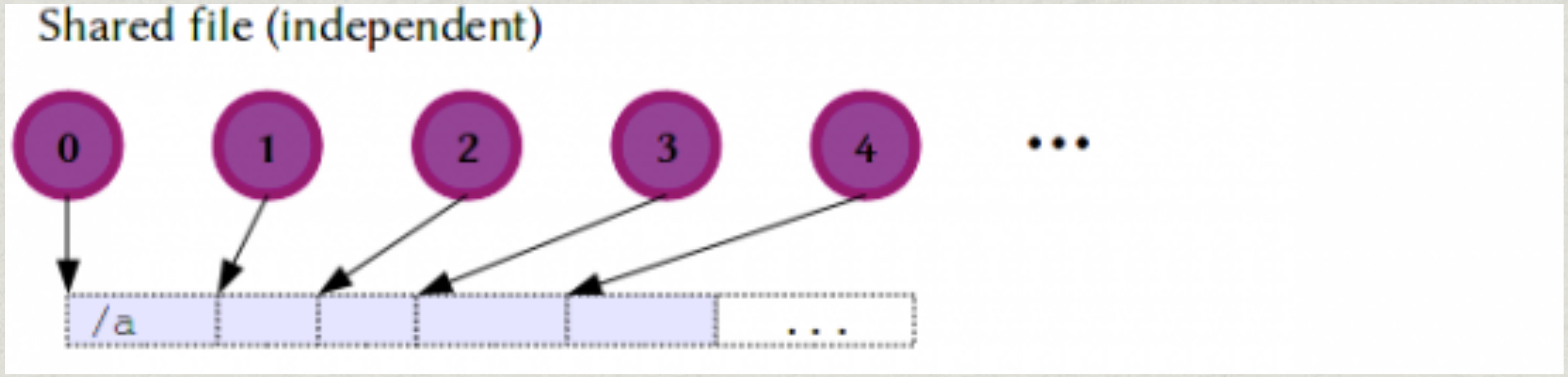
https://www.nersc.gov/users/training/online-tutorials/introduction-to-scientific-i-o/?show_all=1

File-per-Processor Access



- Simplest to implement – each processor has its own file-handle and works independently of other nodes
- PFS perform well on this type of IO, but this creates a bottleneck of managing metadata on large collection of files
- Another problem is that program restarts are now dependent on getting the same processor-file layout

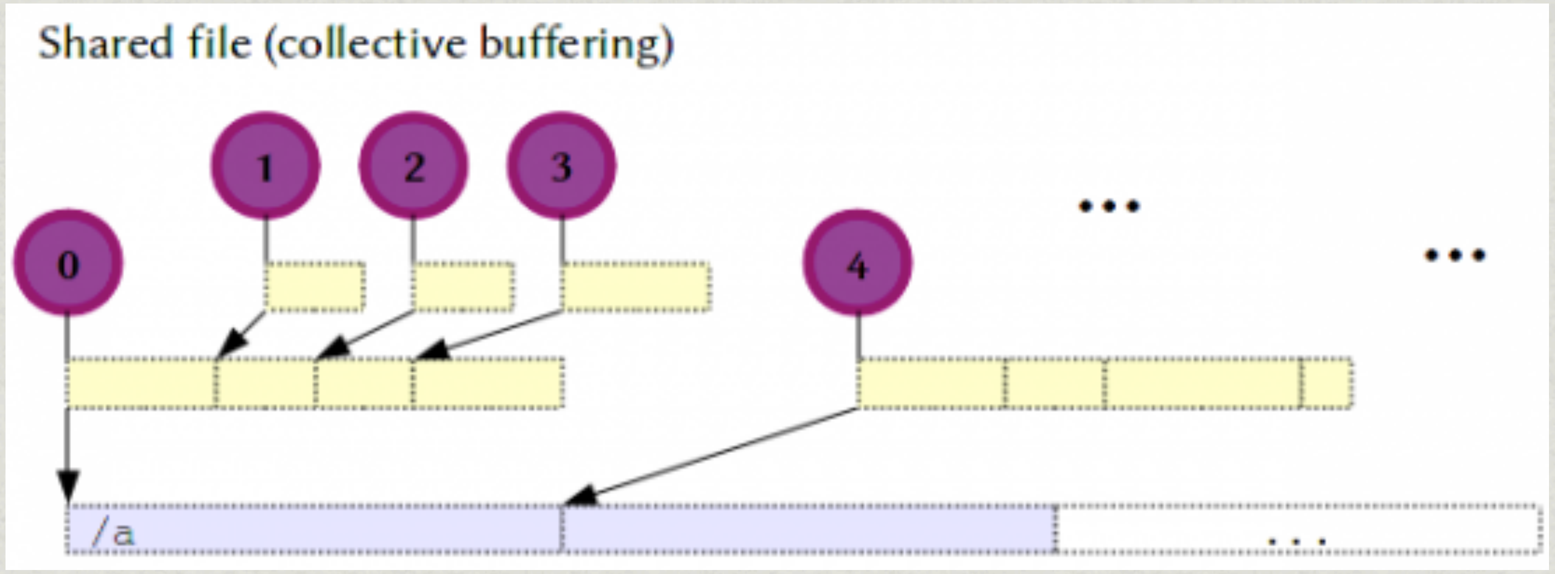
Shared-File (Independent) Access



- Many processors share the same file-handle, but write to their own distinct sections of a shared file
- If there are shared regions of files, then a locking manager is used to serialize access
  - For large O(N), the locking is an impediment to performance
  - Even in ideal cases where the file system is guaranteed that processors are writing to exclusive regions, shared file performance can be lower compared to file-per-processor

https://www.nersc.gov/users/training/online-tutorials/introduction-to-scientific-i-o/?show_all=1

Shared-File (Collective Buffering) Access



Collective buffering: a technique used to improve the performance of shared-file access by offloading some of the coordination work from the file system to the application. (Originally developed to reduce the number of small, noncontiguous writes.)

- A subset of the processors is chosen to be the 'aggregators'
- These collect data from other processors and pack it into contiguous buffers in memory that are then written to the file system
- Reducing the number of processors that interact with the I/O subservers reduces PFS contention
- Another benefit that is important for file systems such as Lustre is that the buffer size can be set to a multiple of the ideal transfer size preferred by the file system

https://www.nersc.gov/users/training/online-tutorials/introduction-to-scientific-i-o/?show_all=1

Summary (3)

- HDFS has an almost identical architecture to GFS to support MapReduce execution
  - 128MB block (HDFS) vs 64MB chunk (GFS)
  - append-only write vs random write
  - single-write multiple-read vs multiple-write multiple-read
- Parallel File Systems (PFS) aim to support parallel I/O from multiple processes (nodes)
- PFS performance depends on
  - File checkpointing pattern: N-N; N-1 segmented; N-1 strided.
  - Data access pattern:
    - File-per-processor
    - Shared-file (independent)
    - Shared-file (collective buffering)

Open Question

- Control (management of the Maste/NameNode) in GFS/HDFS is centralized.
  - What need to be done in order to make the control decentralized?
  - Can you think of a possible scheme of decentralized control?

Next Week:

- **DHT (Distributed Hash Tables)**
- **DTD (Distributed Termination Detection)**