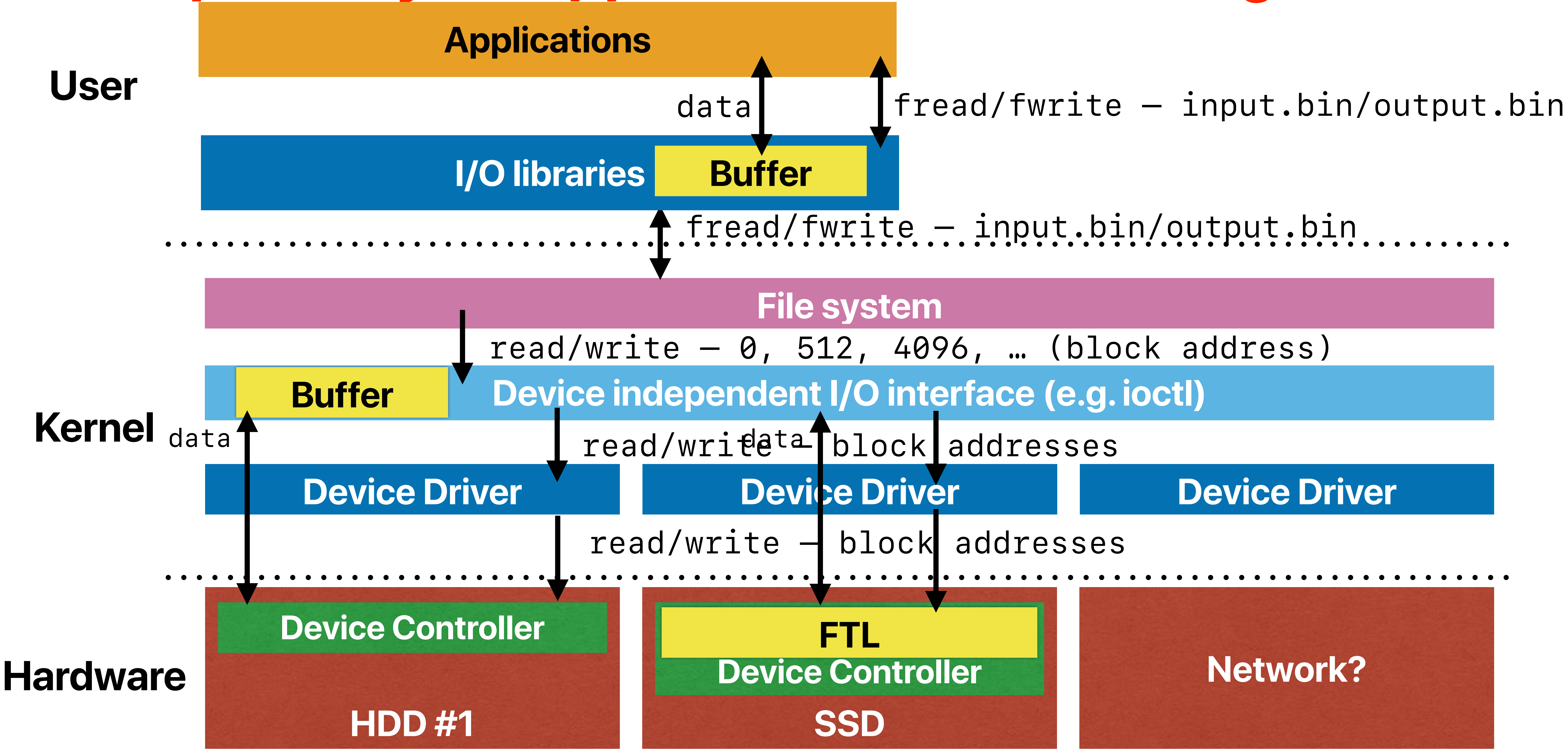
File systems over the network

Hung-Wei Tseng

Recap: How your application reaches storage device



Recap: File systems on a computer

- Unix File System
  - Hierarchical directory structure
  - File — metadata (inode) + data
  - Everything is files
- BSD Fast File System — optimize for reads
  - Cylinder group — Layout data carefully with device characteristics, replicated metadata
  - Larger block size & fragments to fix the drawback
  - A few other new features
- Sprite Log-structured File System — optimize for small random writes
  - Computers cache a lot — reads are no more the dominating traffic
  - Aggregates small writes into large sequential writes to the disk
  - Invalidate older copies to support recovery

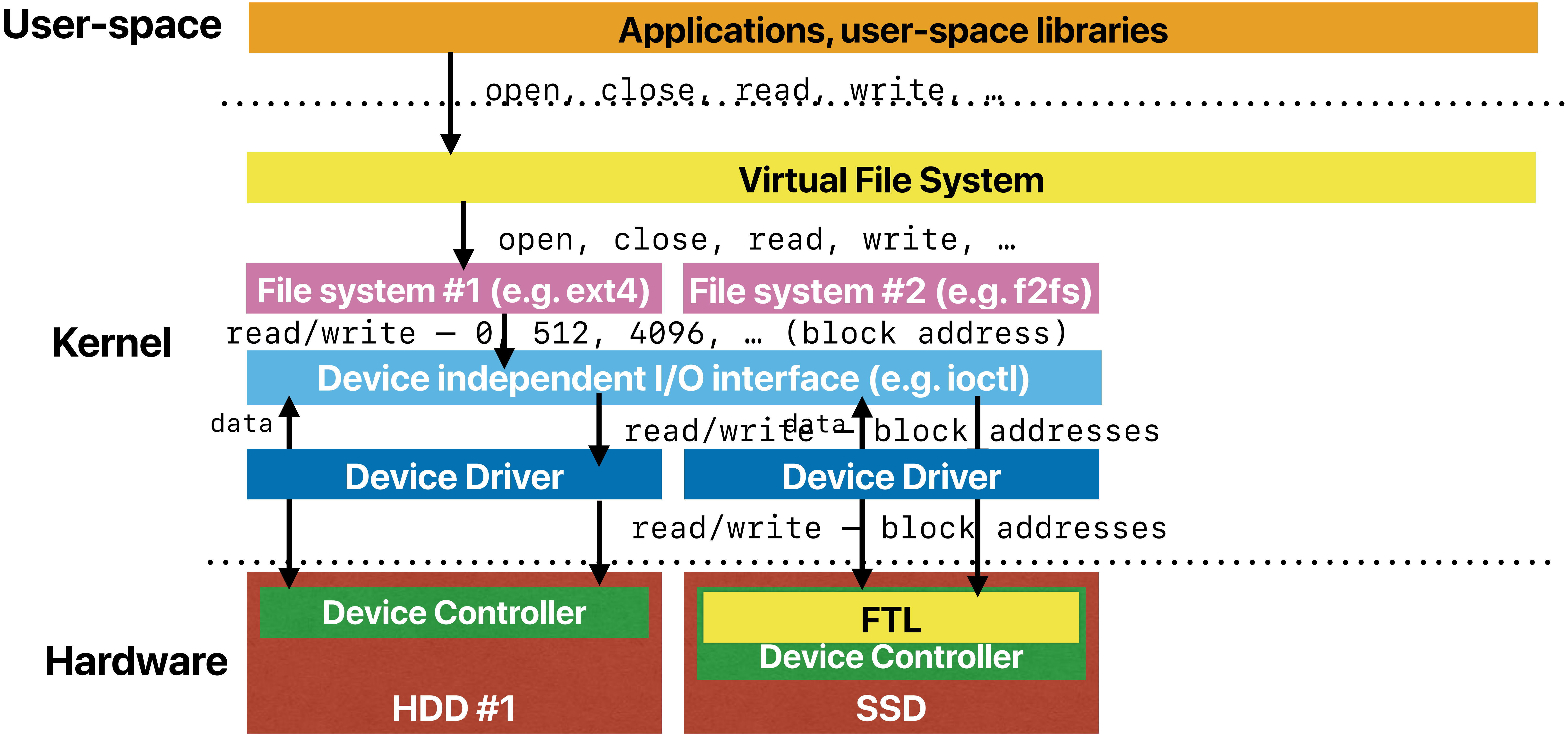
Recap: Extent file systems — ext2, ext3, ext4

- Basically optimizations over FFS + Extent + Journaling (write-ahead logs)
- Extent — consecutive disk blocks
- A file in ext file systems — a list of extents
- Journal
  - Write-ahead logs — performs writes as in LFS
  - Apply the log to the target location when appropriate
- Block group
  - Modern H.D.Ds do not have the concept of "cylinders"
  - They label neighboring sectors with consecutive block addresses
  - Does not work for SSDs given the internal log-structured management of block addresses

Recap: flash SSDs, NVM-based SSDs

- Asymmetric read/write behavior/performance
- Wear-out faster than traditional magnetic disks
- Another layer of indirection is introduced
    - Intensify log-on-log issues
    - We need to revise the file system design

The introduction of virtual file system interface

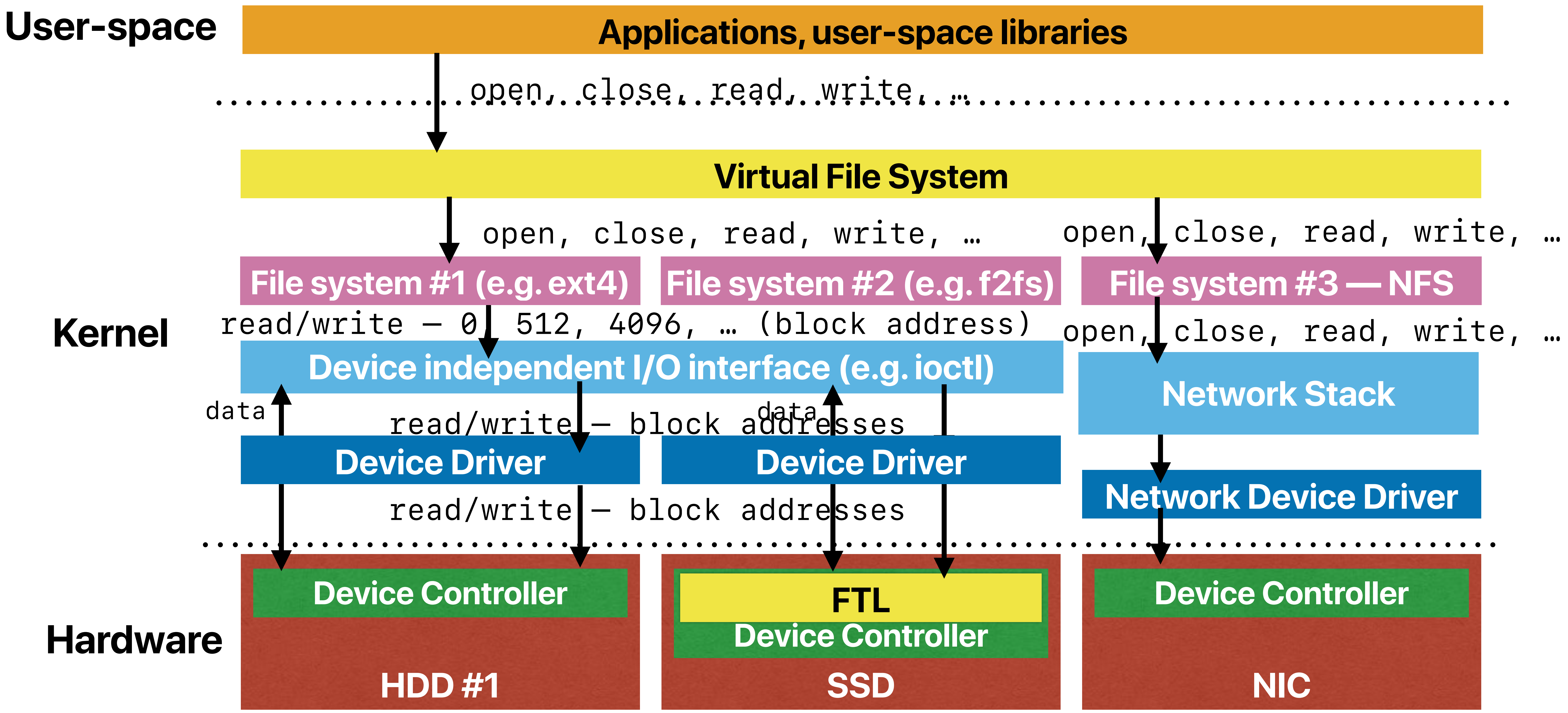


Outline

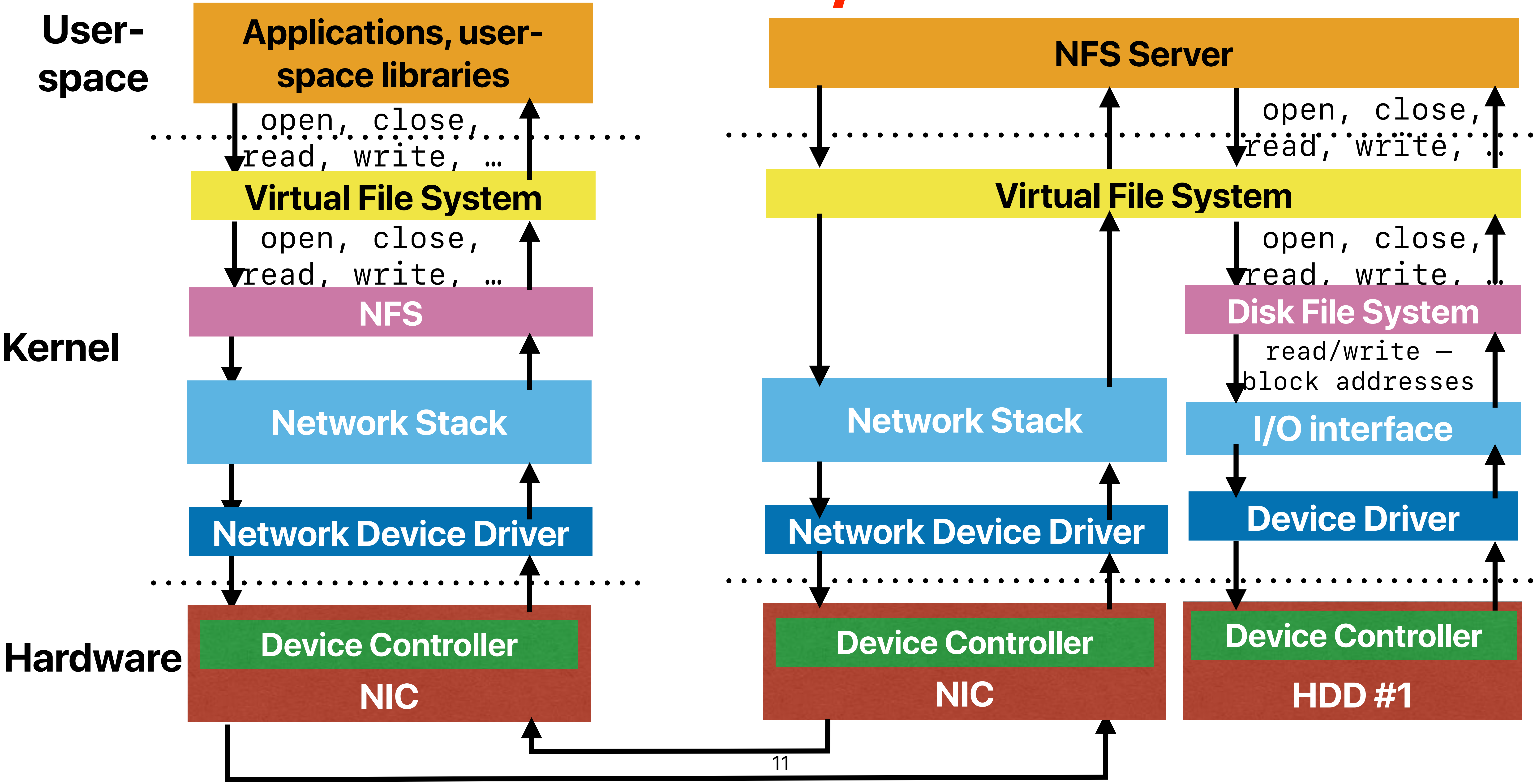
- NFS
- Google file system

Network File System

The introduction of virtual file system interface



NFS Client/Server



How does NFS handle a file?

- The client gives it’s file system a tuple to describe data
  - Volume: Identify which server contains the file — represented by the mount point in UNIX
  - inode: Where in the server
  - generation numer: version number of the file
- The local file system forwards the requests to the server
- The server response the client with file system attributes as local disks

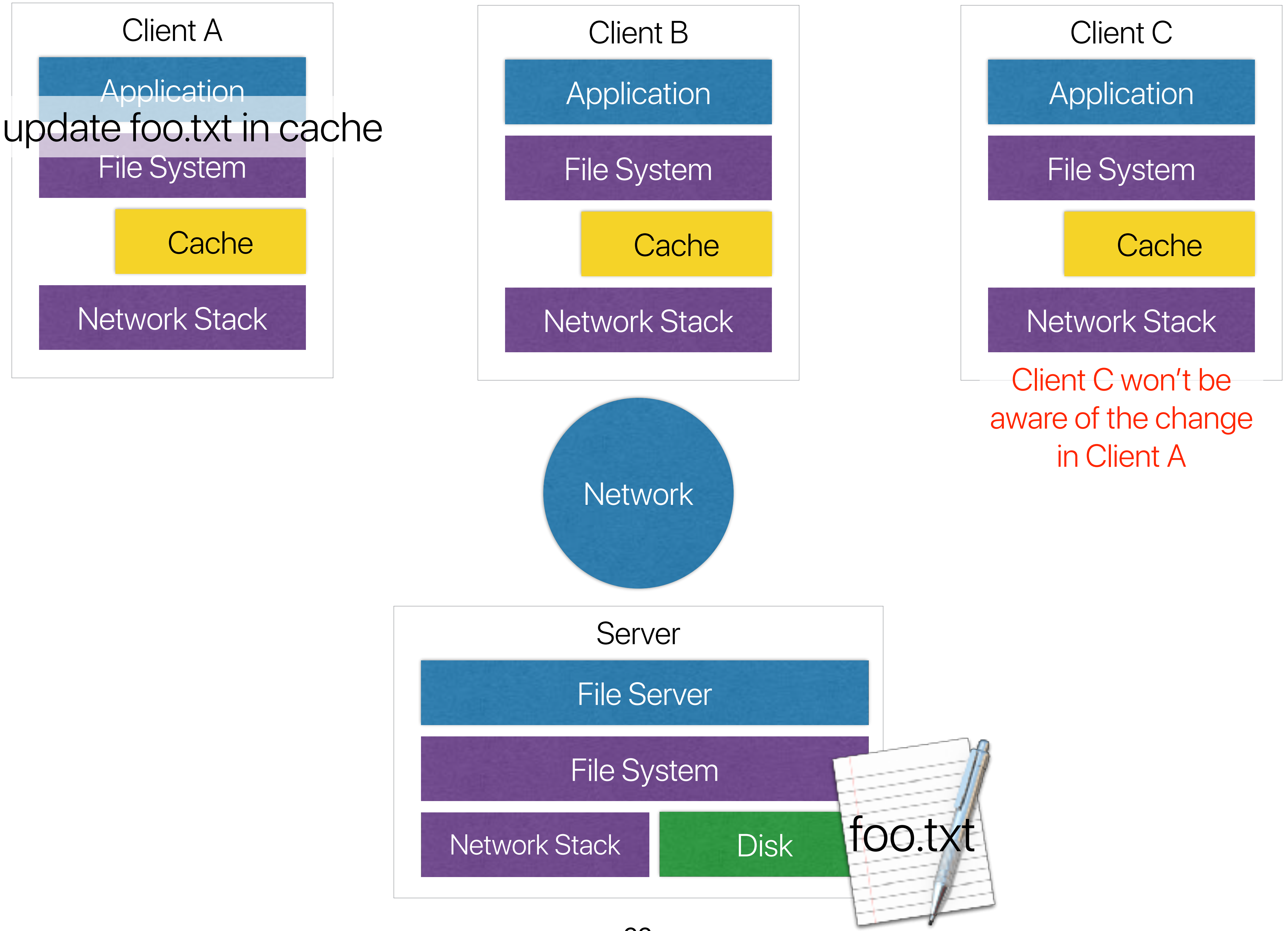
Caching

- NFS operations are expensive
  - Lots of network round-trips
  - NFS server is a user-space daemon
- With caching on the clients
  - Only the first reference needs network communication
  - Later requests can be satisfied in local memory

Idempotent operations

- Given the same input, always give the same output regardless how many times the operation is employed
- You only need to retry the same operation if it failed

Think about this



Solution

- Flush-on-close: flush all write buffer contents when close the file
  - Later open operations will get the latest content
- Force-getattr:
  - Open a file requires getattr from server to check timestamps
  - attribute cache to remedy the performance

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google

Why we care about GFS

- Conventional file systems do not fit the demand of data centers
- Workloads in data centers are different from conventional computers
  - Storage based on inexpensive disks that fail frequently
  - Many large files in contrast to small files for personal data
  - Primarily reading streams of data
  - Sequential writes appending to the end of existing files
  - Must support multiple concurrent operations
  - Bandwidth is more critical than latency

Data-center workloads for GFS

- Google Search (Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, vol. 23, 2003)
- MapReduce (MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004)
  - Large-scale machine learning problems
  - Extraction of user data for popular queries
  - Extraction of properties of web pages for new experiments and products
  - Large-scale graph computations
- BigTable (Bigtable: A Distributed Storage System for Structured Data, OSDI 2006)
  - Google analytics
  - Google earth
  - Personalized search

What GFS proposes?

- Maintaining the same interface
  - The same function calls
  - The same hierarchical directory/files
- Files are decomposed into large chunks (e.g. 64MB) with replicas
- Hierarchical namespace implemented with flat structure
- Master/chunkservers/clients

Latency Numbers Every Programmer Should Know

Operations	Latency (ns)	Latency (us)	Latency (ms)	
L1 cache reference	0.5 ns			~ 1 CPU cycle
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	3 us		
Send 1K bytes over 1 Gbps network	10,000 ns	10 us		
Read 4K randomly from SSD*	150,000 ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Read 512B from disk	10,000,000 ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA-Netherlands-CA	150,000,000 ns	150,000 us	150 ms	

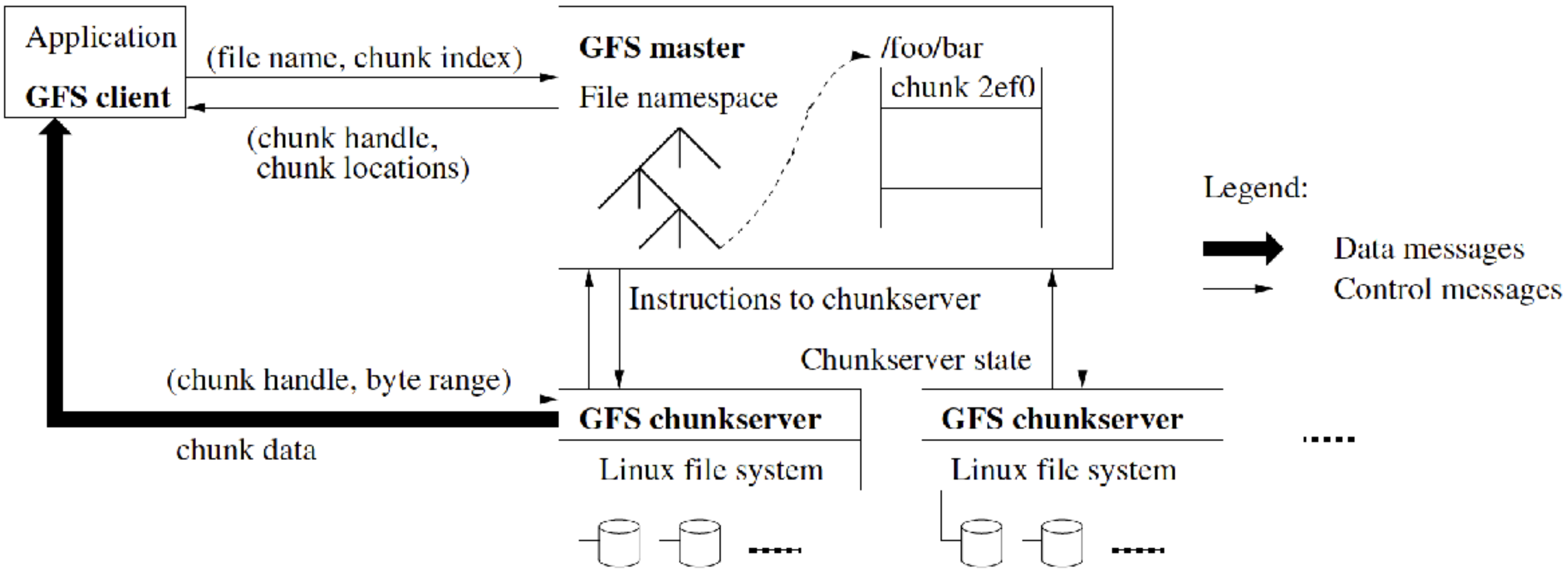
Flat file system structure

- Directories are illusions
- Namespace maintained like a hash table

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this

Distributed architecture

decoupled data and control paths —
only control path goes through master

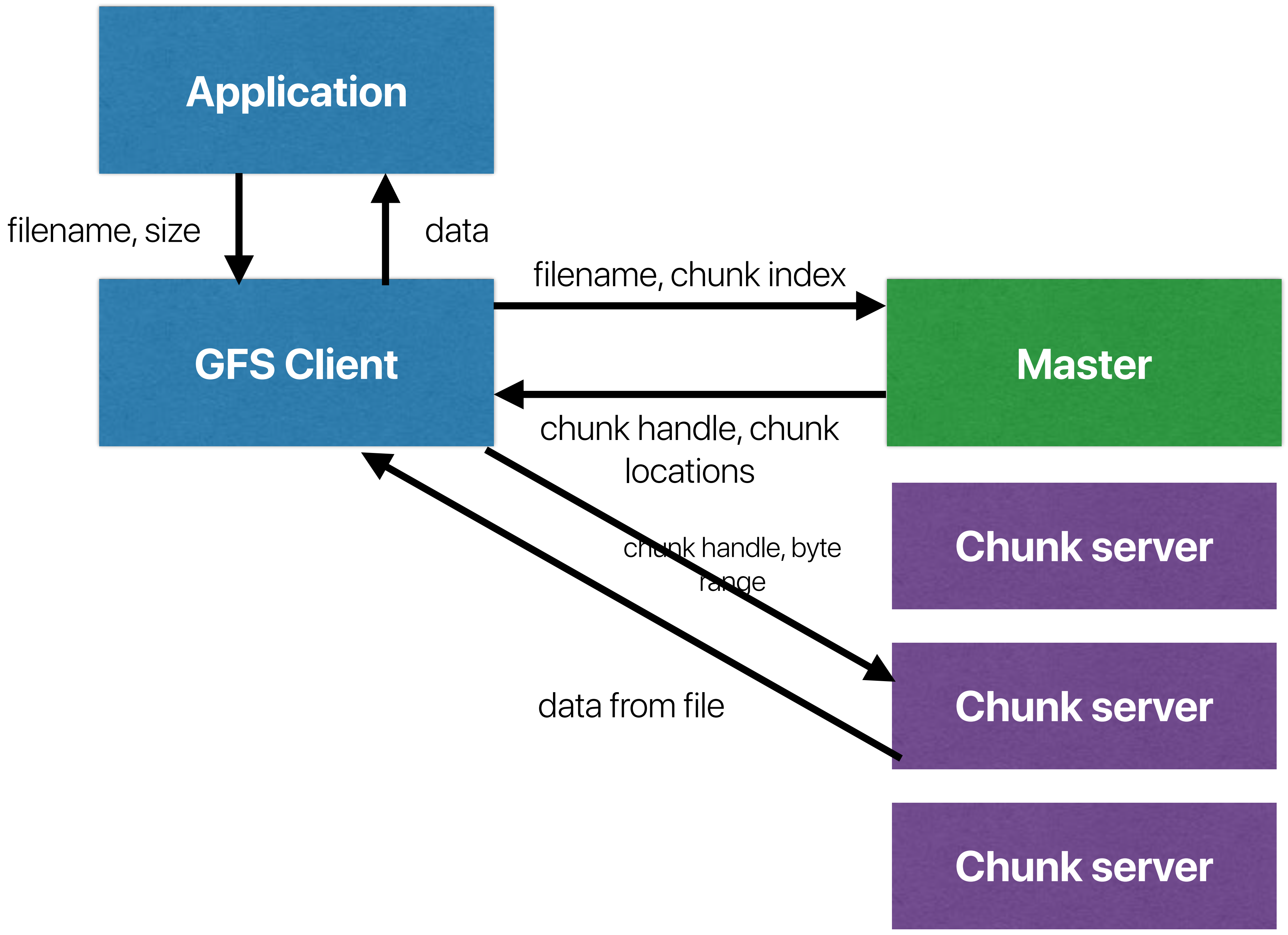


load balancing, replicas among chunkservers

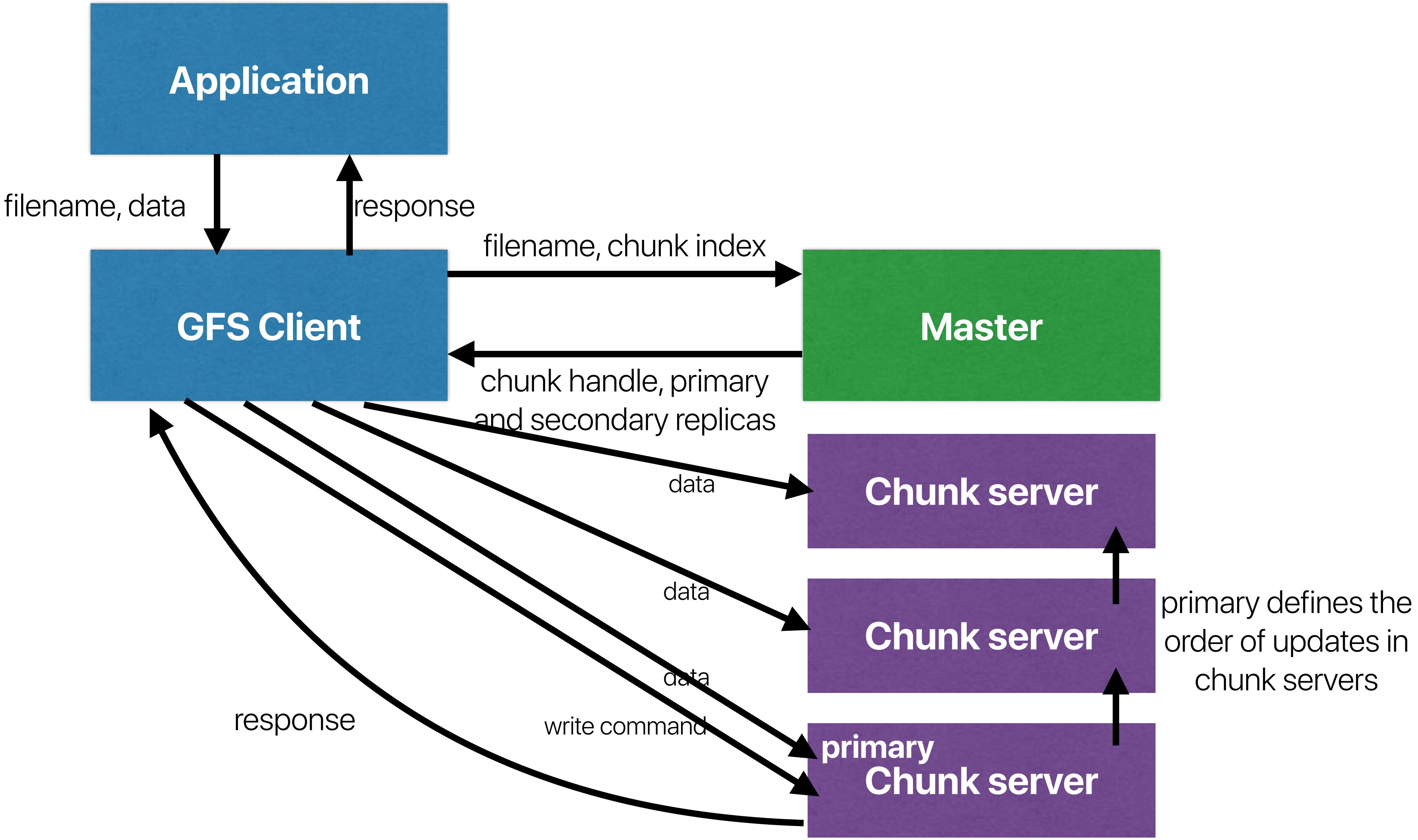
Distributed architecture

- Single master
  - maintains file system metadata including namespace, mapping, access control and chunk locations.
  - controls system wide activities including garbage collection and chunk migration.
- Chunkserver
  - stores data chunks
  - chunks are replicated to improve reliability (3 replicas)
- Client
  - APIs to interact with applications
  - interacts with masters for control operations
  - interacts with chunkservers for accessing data
  - Can run on chunkservers

Reading data in GFS



Writing data in GFS



Real world, industry experience

- Linux problems (section 7)
  - Linux driver issues — disks do not report their capabilities honestly
  - The cost of fsync — proportion to file size rather than updated chunk size
  - Single reader-writer lock for mmap
  - Due to the open-source nature of Linux, they can fix it and contribute to the rest of the community
- **GFS is not open-sourced**

> system behavior. When appropriate, we improve the kernel
> and share the changes with the open source community.

Single master design

- GFS claims this will not be a bottleneck
- In-memory data structure for fast access
- Only involved in metadata operations — decoupled data/control paths
- Client cache
- What if the master server fails?

The evolution of GFS

- Mentioned in "Spanner: Google's Globally-Distributed Database", OSDI 2012 — "tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System)"
- Single master

acmqueue Case Study
GFS: Evolution on Fast-forward

A discussion between Kirk McKusick and Sean Quinlan about the origin and evolution of the Google File System.

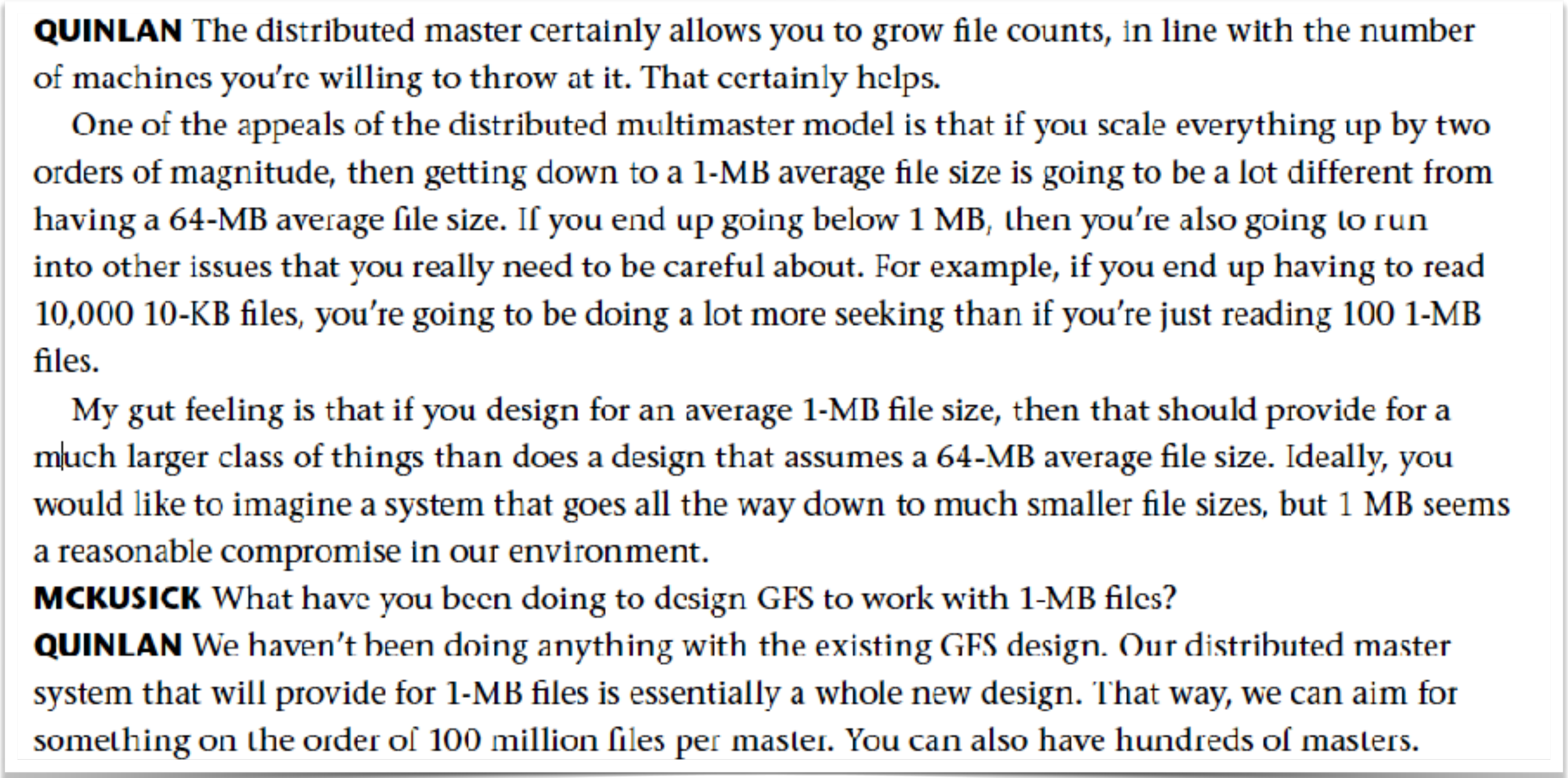
proportionate increase in the amount of metadata the master had to maintain. Also, operations such as scanning the metadata to look for recoveries all scaled linearly with the volume of data. So the amount of work required of the master grew substantially. The amount of storage needed to retain all that information grew as well.

In addition, this proved to be a bottleneck for the clients, even though the clients issue few metadata operations themselves—for example, a client talks to the master whenever it does an open. When you have thousands of clients all talking to the master at the same time, given that the master is capable of doing only a few thousand operations a second, the average client isn't able to command all that many operations per second. Also bear in mind that there are applications such as MapReduce, where you might suddenly have a thousand tasks, each wanting to open a number of files. Obviously, it would take a long time to handle all those requests, and the master would be under a fair amount of duress.

MCKUSICK And historically you've had one cell per data center, right?
QUINLAN That was initially the goal, but it didn't work out like that to a large extent—partly because of the limitations of the single-master design and partly because isolation proved to be difficult. As a consequence, people generally ended up with more than one cell per data center. We also ended up doing what we call a "multi-cell" approach, which basically made it possible to put multiple GFS masters on top of a pool of chunkservers. That way, the chunkservers could be configured to have, say, eight GFS masters assigned to them, and that would give you at least one pool of underlying storage—with multiple master heads on it, if you will. Then the application was responsible for partitioning data across those different cells.

The evolution of GFS

- Support for smaller chunk size — gmail



QUINLAN The distributed master certainly allows you to grow file counts, in line with the number of machines you're willing to throw at it. That certainly helps.

One of the appeals of the distributed multimaster model is that if you scale everything up by two orders of magnitude, then getting down to a 1-MB average file size is going to be a lot different from having a 64-MB average file size. If you end up going below 1 MB, then you're also going to run into other issues that you really need to be careful about. For example, if you end up having to read 10,000 10-KB files, you're going to be doing a lot more seeking than if you're just reading 100 1-MB files.

My gut feeling is that if you design for an average 1-MB file size, then that should provide for a much larger class of things than does a design that assumes a 64-MB average file size. Ideally, you would like to imagine a system that goes all the way down to much smaller file sizes, but 1 MB seems a reasonable compromise in our environment.

MCKUSICK What have you been doing to design GFS to work with 1-MB files?

QUINLAN We haven't been doing anything with the existing GFS design. Our distributed master system that will provide for 1-MB files is essentially a whole new design. That way, we can aim for something on the order of 100 million files per master. You can also have hundreds of masters.

Lots of other interesting topics

- snapshots
- namespace locking
- replica placement
- create, re-replication, re-balancing
- garbage collection
- stable replica detection
- data integrity
- diagnostic tools: logs are your friends

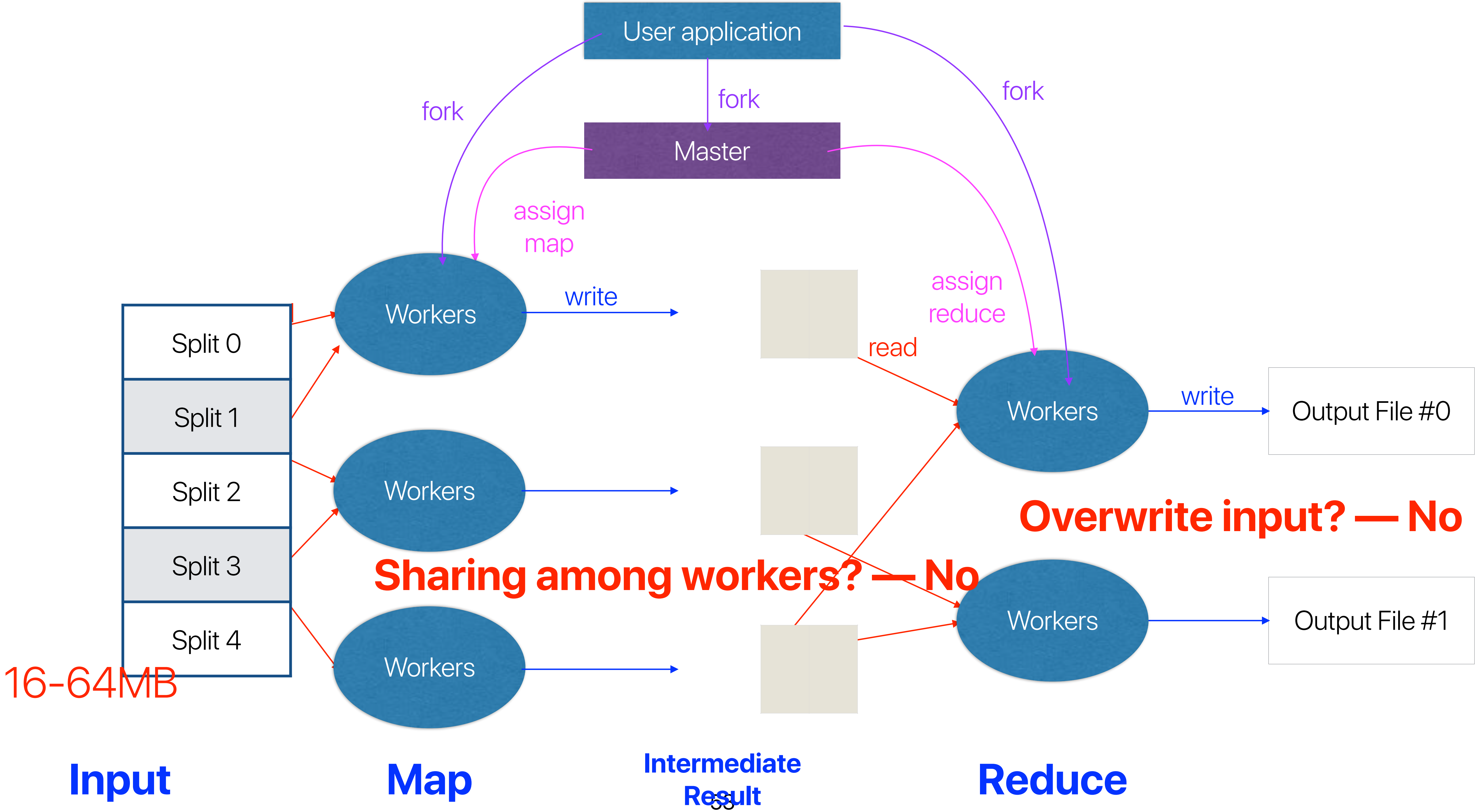
GFS: Relaxed Consistency model

- Distributed, simple, efficient
- Filename/metadata updates/creates are atomic
- Consistency modes

	Write — write to a specific offset	Append — write to the end of a file
Serial success	Defined	Defined with interspersed with inconsistent
Concurrent success	Consistent but undefined	
Failure	inconsistent	

  - Consistent: all replicas have the same value
  - Defined: replica reflects the mutation, consistent
- Applications need to deal with inconsistent cases themselves

MapReduce


User application
fork
fork
fork
Master
assign
map
assign
reduce
Workers
Workers
Workers
write
read
Workers
Workers
write
Output File #0
Output File #1
Split 0
Split 1
Split 2
Split 3
Split 4
16-64MB
Overwrite input? — No
Sharing among workers? — No
Input
Map
Intermediate
Result
Reduce

Why we care about GFS

- Conventional file systems do not fit the demand of data centers
- Workloads in data centers are different from conventional computers
  - Storage based on inexpensive disks that fail frequently
    — MapReduce is fault tolerant
  - Many large files in contrast to small files for personal data
    — MapReduce aims at processing large amount of data once
  - Primarily reading streams of data — MapReduce reads chunks of large files
  - Sequential writes appending to the end of existing files
    — Output file keep growing as workers keep writing
  - Must support multiple concurrent operations
    —MapReduce has thousands of workers simultaneously
  - Bandwidth is more critical than latency
    —MapReduce only wants to finish tasks within "reasonable" amount of time