香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

Lecture 05:
Distributed File Systems

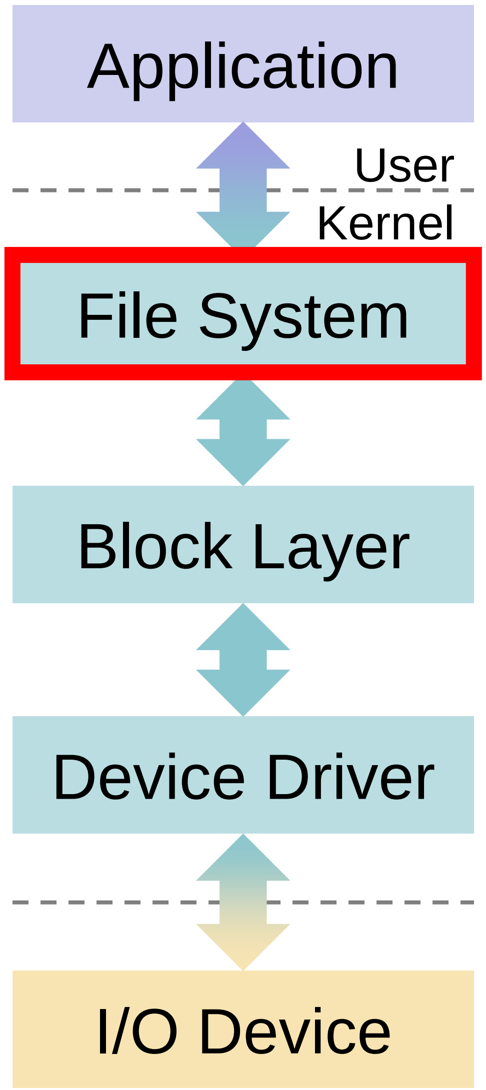
Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency

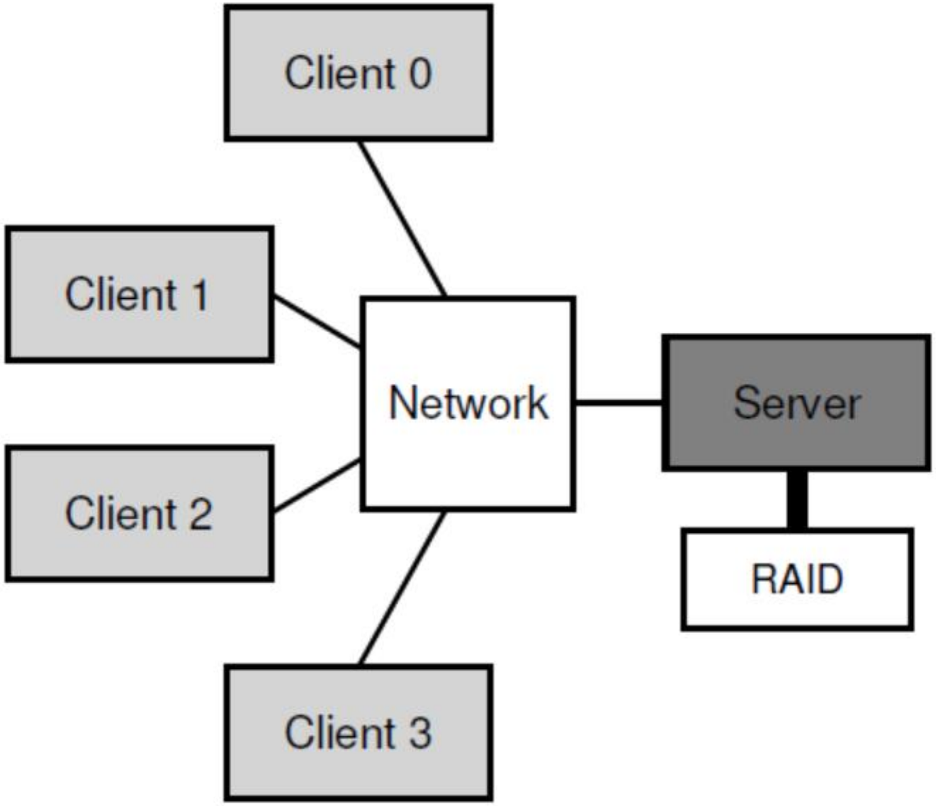
I/O Stack



Client-Server Model (1/2)

- Generic Client-Server Model:
  - **One (or a few) server** stores the data on its disks;
  - **Multiple clients** request data through protocol messages.

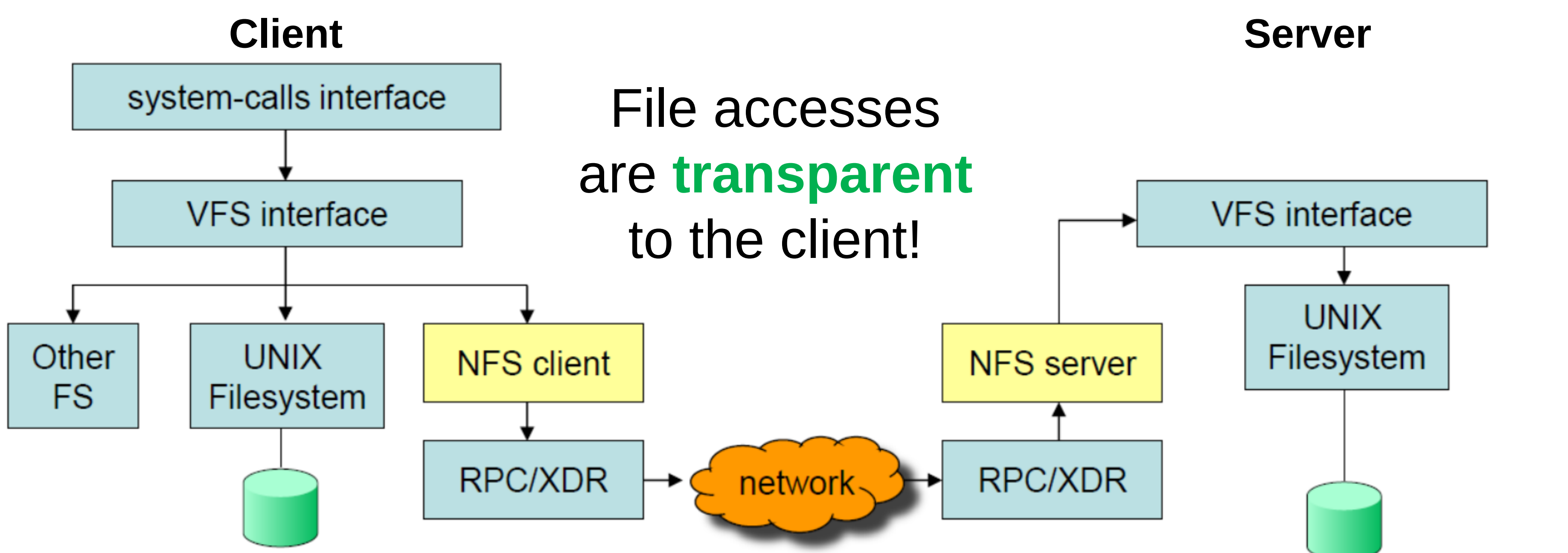
Easy sharing of data across multi-clients



Centralized administration for data backup and security

Client-Server Model (2/2)

- **Client**
  - Issues system calls to the **client-side file system** to access files on the server.
  - Caches retrieved blocks in memory for future use.
- **Server**
  - Accesses data blocks in the **server-side file system** (i.e., **file server**).
  - Caches and buffers reads/writes in memory.

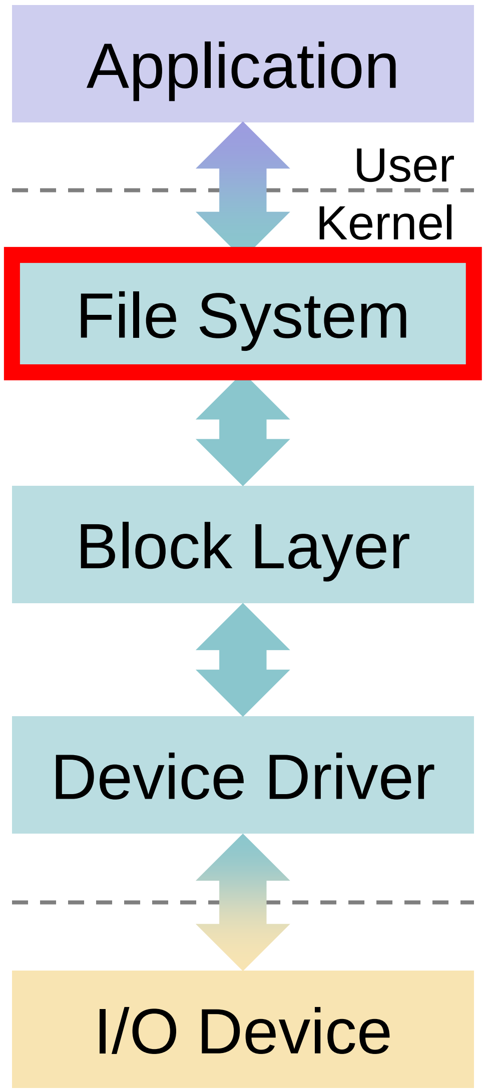


File accesses are **transparent** to the client!

Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency

I/O Stack



Network File System (NFS)

- **Sun Network File System (NFS)**
  - Developed by **Sun Microsystems** in 1980's.
  - An **open protocol** that specifies the exact message formats for client-server communication.
    - Rather than a proprietary and **closed system**.
  - **It worked**: Many big companies sell NFS servers, including Oracle/Sun, NetApp, EMC, IBM, etc.
  - Current Standard: NFSv4 supports larger-scale protocol.

- We focus on the NFS protocol version 2 (**NFSv2**):
  - **Goals of NFSv2**: **simplicity** and **fast crash recovery**
    - Crashes are common in distributed systems, due to power outages, software bugs, network disconnections, etc.

Fast Crash Recovery: Statelessness (1/2)

- NFS is **stateless**: The file server doesn't keep track of anything about the actions of clients.
  - Each client includes all information in the protocol request;
  - The server processes and then "forgets" the request.
- CounterEx: **Shared state** complicates crash recovery.
  - The client-side file system **`opens`** the file.
  - The file server opens the file and returns the descriptor (**`fd`**).
  - The client-side file system uses **`fd`** for subsequent reads.

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor from server
read(fd, buffer, N);             // read N bytes from foo via fd
...
close(fd);                       // close file
```

Fast Crash Recovery: Statelessness (2/2)

- **Server Crashes**
  - Imagine the server crashes between two consecutive reads.
  - After the server is up again, the client re-issues the read.
  - The server has no idea to which file **`fd`** is referring.
    - **`fd`** was keeping in server memory and lost when server crashed.
- **Client Crashes**
  - Imagine a client opens a file and then crashes.
    - The **`open()`** uses up a file descriptor on the server.
  - However, the server never receives a **`close()`**.
- For above reasons, NFS adopts a **stateless design**.
  - No fancy crash recovery is needed:
    - The server just starts running again;
    - A client, at worst, might have to retry a request.

NFSv2: A Stateless File Protocol (1/2)

Key to NFSv2 Protocol: The **File Handle**

- A **file handle** uniquely identifies a file or a directory with three components:
  ① **Volume Identifier**: specifies a file system;
  ② **Inode Number**: specifies a file/directory in a file system;
  ③ **Generation Number**: is needed when reusing an inode.
    - By incrementing it whenever an inode number is reused.
    - The server ensures that a client with an old file handle cannot accidentally access the newly-allocated file.
- A file handle is encoded into some forms of strings.

NFSv2: A Stateless File Protocol (2/2)

- **NFSPROC_LOOKUP**
  - Obtain a file handle for a file or directory from the file server.
- **NFSPROC_READ**
  - Pass the file handle, offset, and the number of bytes to read;
  - Obtain the retrieved data.
- **NFSPROC_WRITE**
  - Pass the file handle, offset, the number of bytes, along with the data to write.
- **NFSPROC_GETATTR/NFSPROC_SETATTR**
  - Get/Set metadata (*e.g.*, last modified time) with a file handle.
- Others: **NFSPROC_CREATE**, **NFSPROC_REMOVE**, **NFSPROC_MKDIR**, **NFSPROC_RMDIR**, **NFSPROC_READDIR**

Protocol Messages

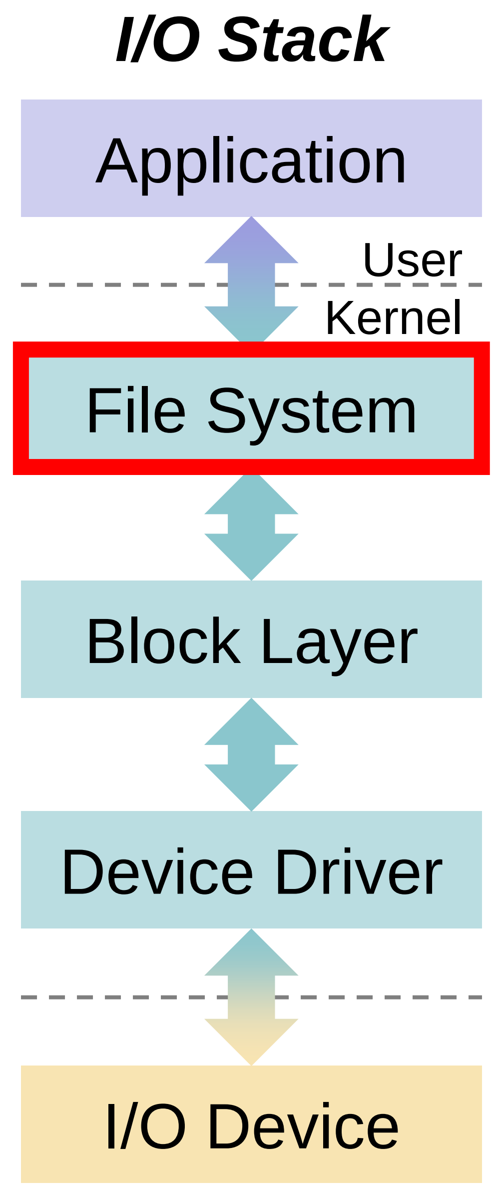
- The client-side file system tracks open files, and translates file system calls into **protocol messages**.
- The server responds to **protocol messages**, which contains all information needed to complete a request.

- Example: Reading a File

Client	Protocol Messages
`fd = open("/foo", …);`	`NFSPROC_LOOKUP(rootdir FH, "foo")`
`read(fd, buffer, N);`	`NFSPROC_READ(FH, offset=0, cnt=N)`
`read(fd, buffer, N);`	`NFSPROC_READ(FH, offset=N, cnt=N)`
`read(fd, buffer, N);`	`NFSPROC_READ(FH, offset=2*N, cnt=N)`
`close(fd);`	*(do nothing)*

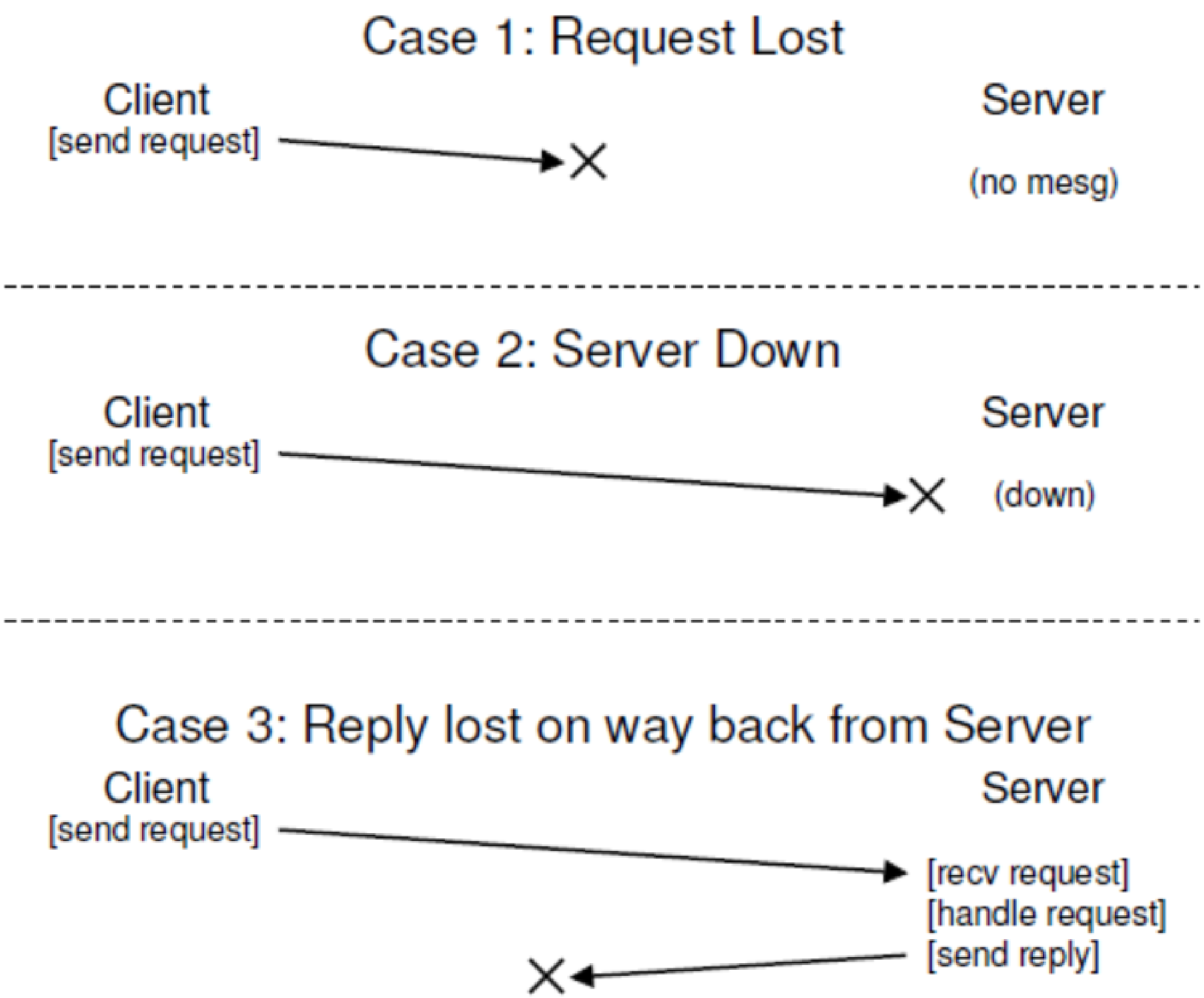
Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency



Handling Server Failures (1/3)

- Three types of protocol message losses:



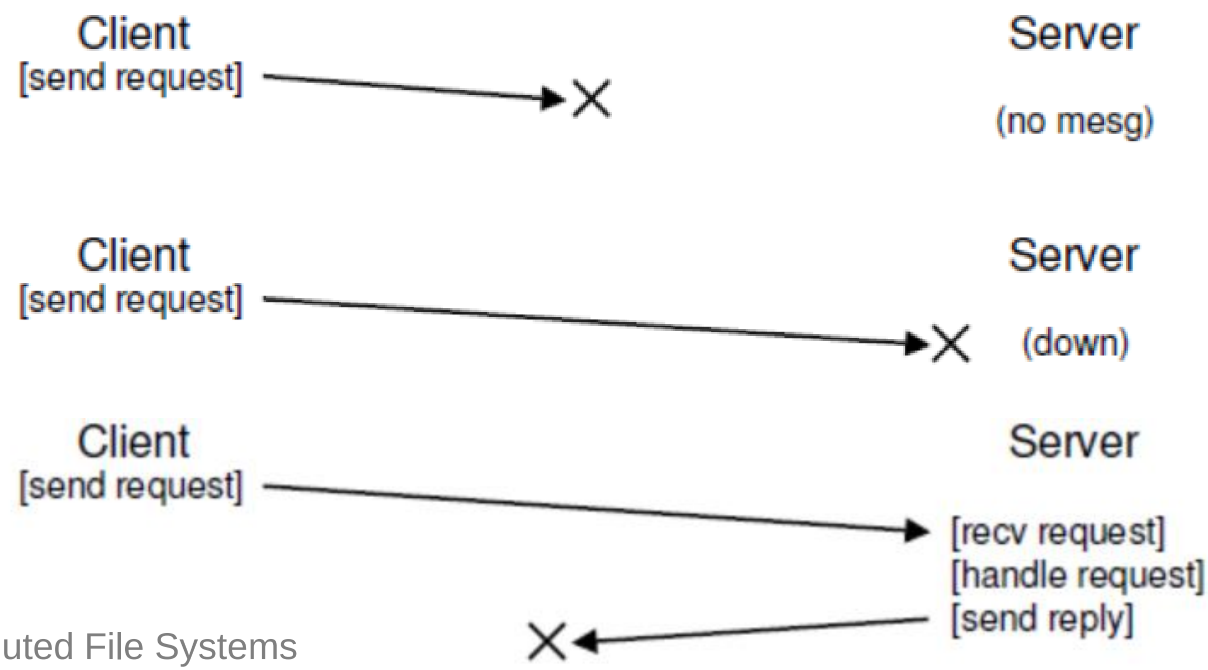
Handling Server Failures (2/3)

- In NFSv2, a client detects the response **timeout** and simply **retries** the request.
- **Reason**: Most NFS requests are **idempotent.**
  - The effect of performing the request **multiple** times is **equivalent** to that of performing the request **a single** time.
  - E.g., **`LOOKUP`**, **`READ`**, and **`WRITE`** requests are idempotent.

Case 1
Request Lost

Case 2
Server Down

Case 3
Reply Lost



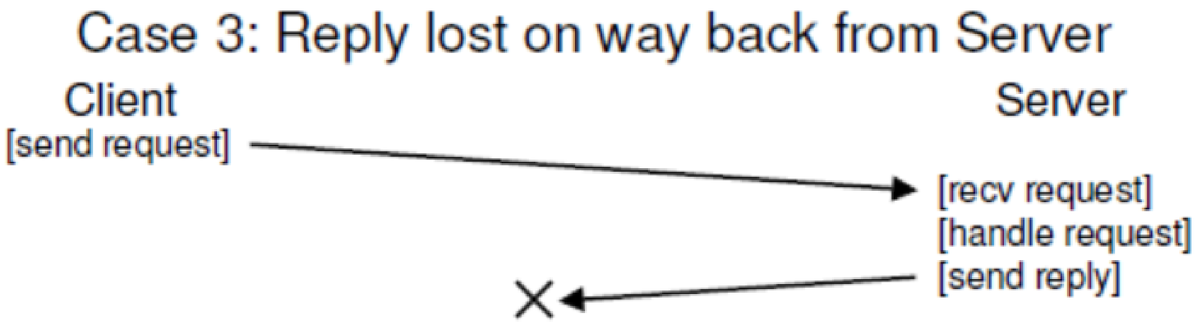
Retry!

Retry!

Do It Again!

Handling Server Failures (3/3)

- Some requests are **hard** to make idempotent.
  – For example, if the file server receives a **`MKDIR`** protocol message and executes it successfully;
  – But the reply is lost and the client may **retry** it (as Case 3).

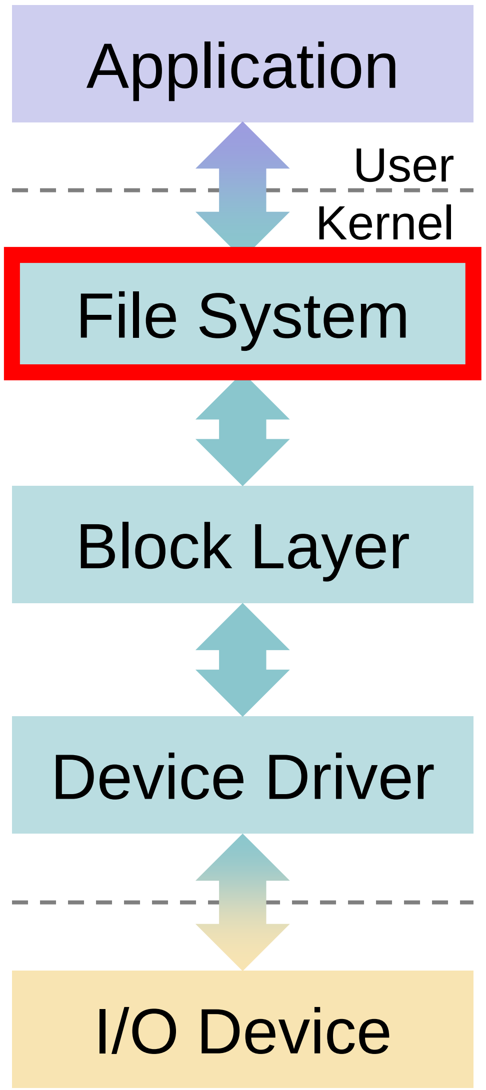


  – The server must **fail the retry** (rather than re-do it).
    - Why? The effect of creating a directory twice is not equivalent to the effect of creating a directory once (i.e., **not idempotent**).

Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency

I/O Stack



Client-side Caching / Buffering

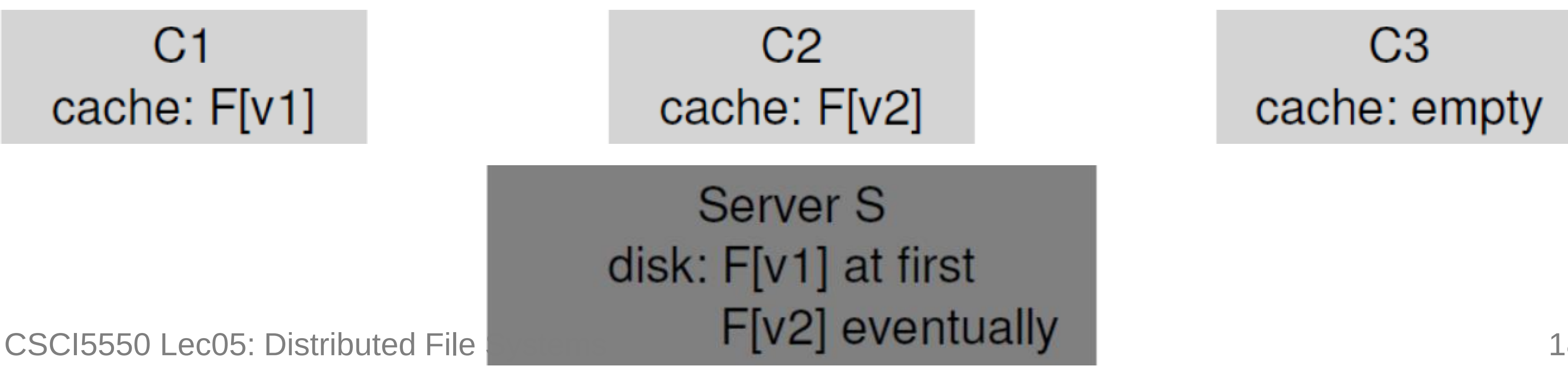
- Sending all read and write requests **across the network** can lead to a big performance problem.
- Intuitive Solution: Client-side Caching / Buffering

- The NFS client **caches** file data and metadata read from server in its local memory.
  - The first access is still expensive (via network communication);
  - Subsequent accesses are serviced quite quickly in memory.
- The NFS client **buffers** data in its local memory before writing them out to server.
  - The **`write()`** system call succeeds immediately.

Cache Consistency Problem (1/2)

- Consider a NFS with three clients and one server:
  - Client C1 reads a file F[v1], and keeps a copy in its cache.
  - Client C2 overwrites file F, but buffers F[v2] in its cache.
  - Client C3 has not yet accessed the file F.
- **Cache Consistency/Coherence Problems**:
  ① **Stale Cache** (from read perspective)
    - The cache still holds not-yet-updated data.

  ② **Update Visibility** (from write perspective)
    - Updates are buffered in memory and not seen by others.



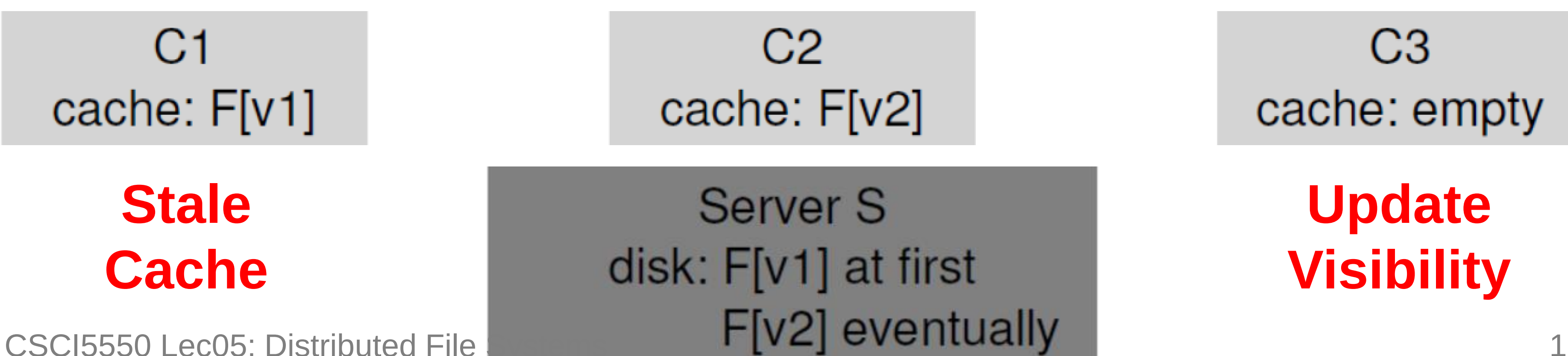
Cache Consistency Problem (2/2)

① **Stale Cache**: C1 has the stale F[v1] in its cache.

- **Solution**: NFS clients first check whether a file has changed before using its cached contents.
- **How?** Issuing a **`GETATTR`** request to server to know when the file was last modified (but raise flooding of **`GETATTR`**).

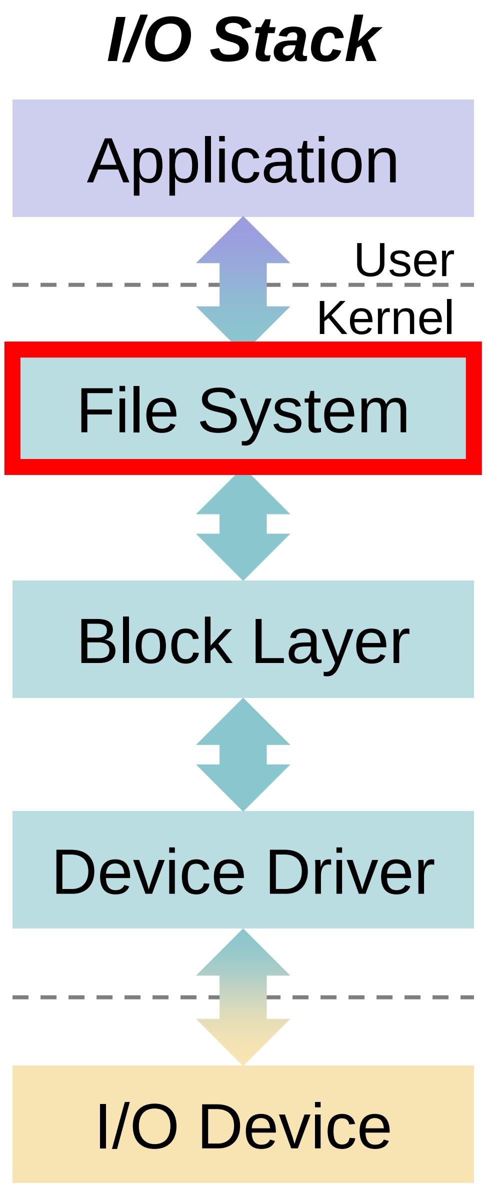
② **Update Visibility**: The update from C2 is not visible to C3: C3 only gets old copy F[v1] from the server.

- **Solution**: NFS clients (C2) implement flush-on-close to ensure that a subsequent open will get the latest version.



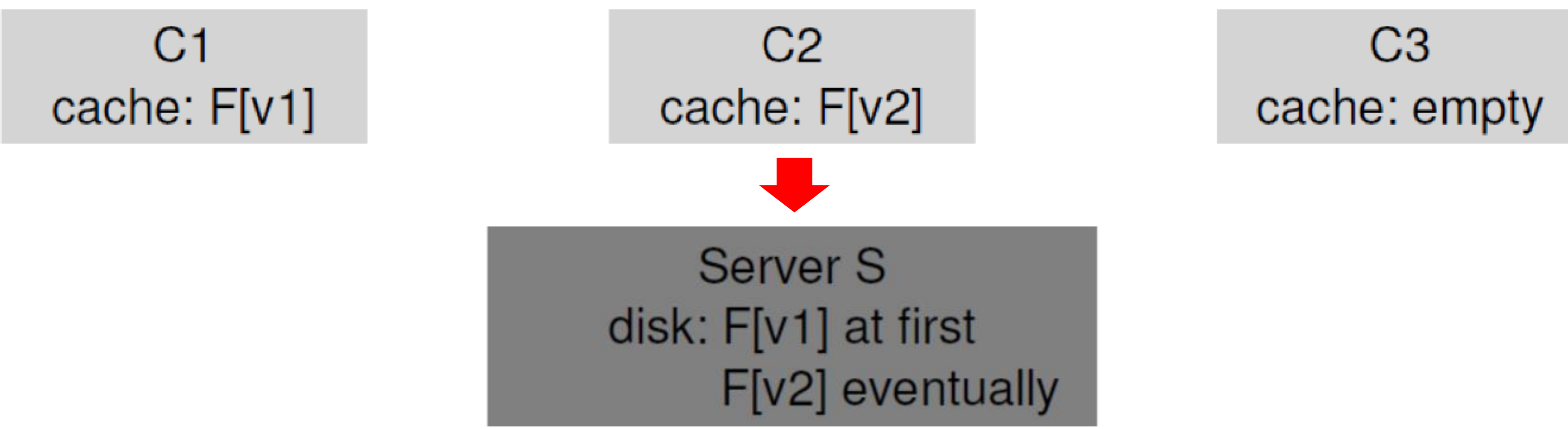
Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency



Server-side Caching / Buffering

- The file server can also cache read/write requests.
- Write buffering needs to be carefully implemented:
  - The server **must commit** each write before informing the client of success.
- To avoid write becoming the performance bottleneck:
  - The server may use battery-backed memory or the log-structured approach to improve write performance.

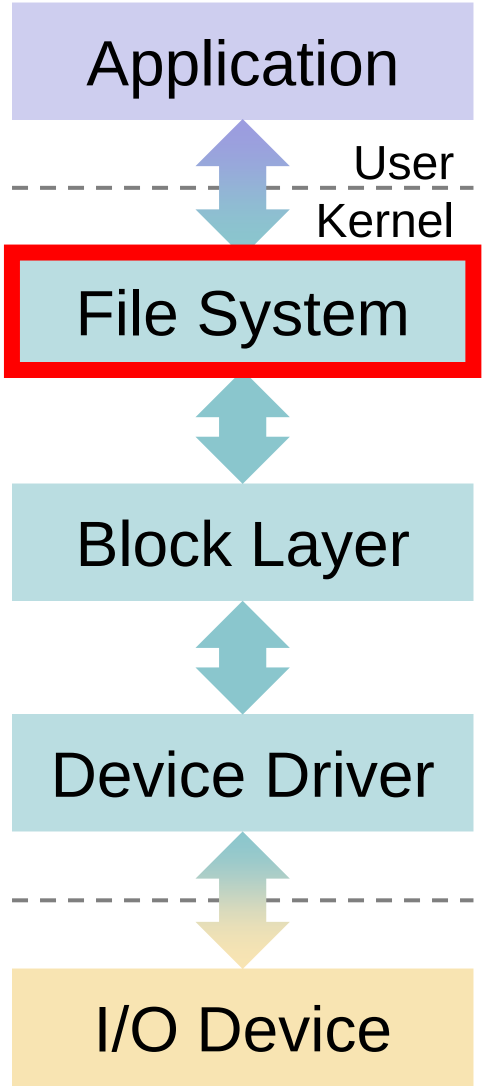


Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering

- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency

I/O Stack



Google File System (GFS)

- GFS is a scalable distributed file system for large distributed data-intensive applications.

- GFS is driven by Google's specific application workloads and technological environment.

- As of 2003, multiple GFS clusters are deployed:
  - Over 1000 storage nodes;
  - Over 300TB disk storage;
  - Heavily accessed by hundreds of clients.

Considerations and Assumptions

① Component failures are the norm, not the exception.
   – The system is of inexpensive components that often fail.

② Files are huge: Multi-GB files are common.

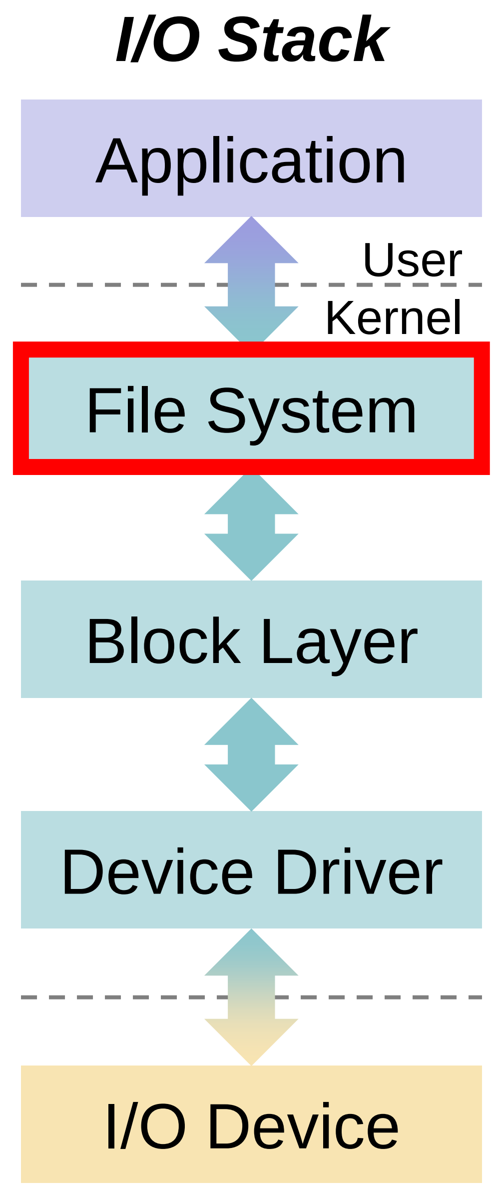
③ Appending new data is much more common than overwriting existing data.
   – Random writes are uncommon; instead, clients may concurrently append large, sequential writes to files.
   – GFS fulfils `record append` and `snapshot` operations.

④ The read workloads consist of large streaming reads and small random reads.

⑤ It is more critical to sustain high bandwidth rather than low latency.

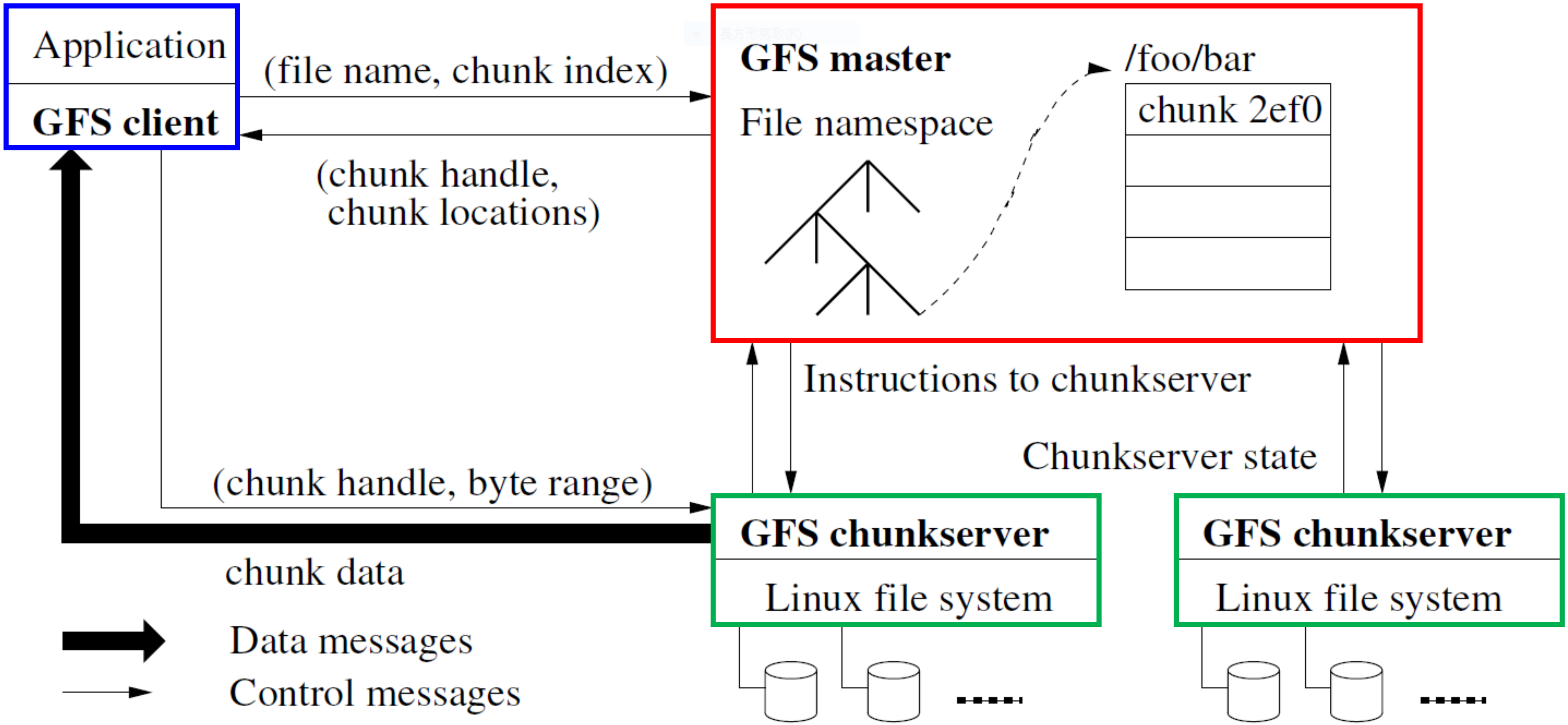
Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency



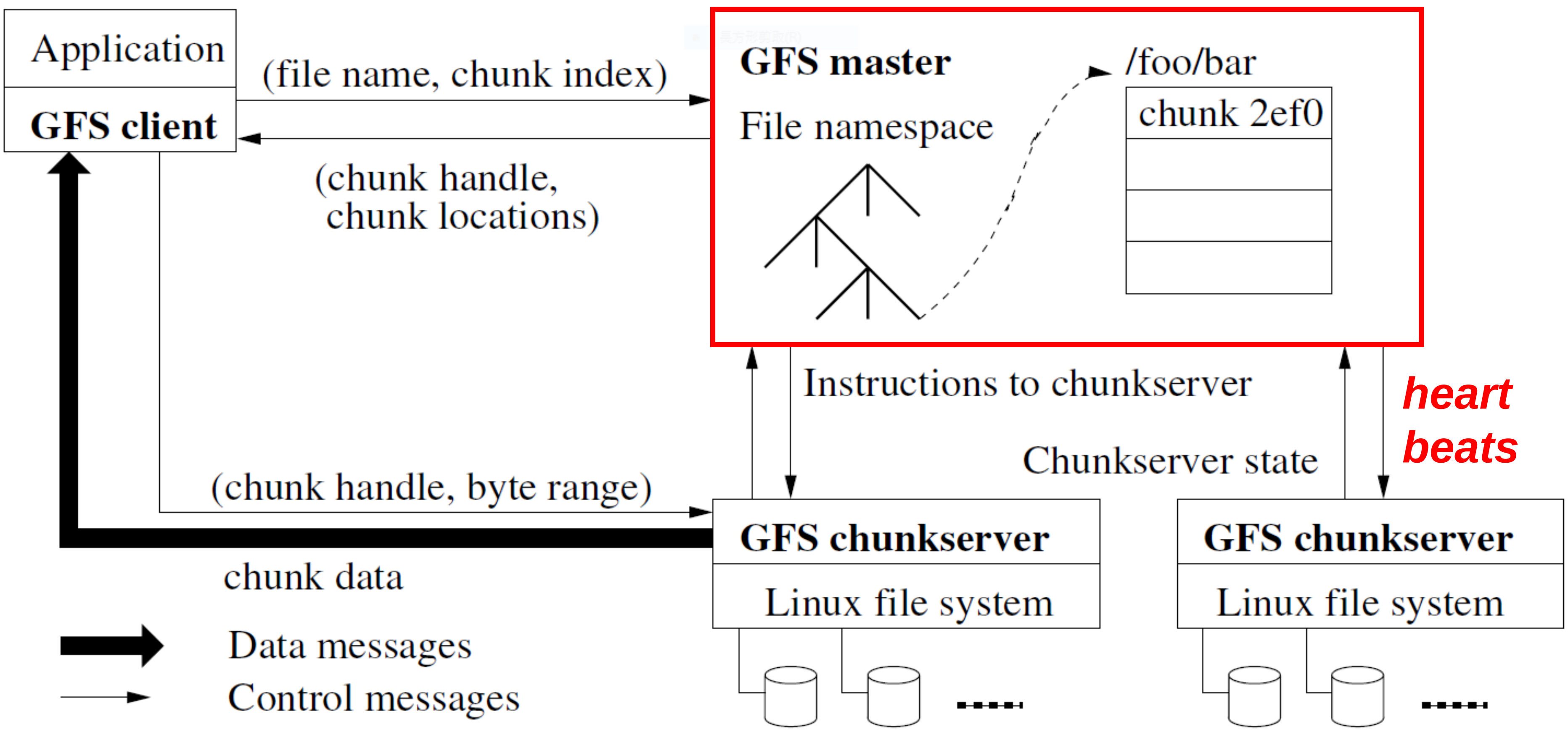
Overview: GFS Architecture

- A GFS cluster consists ① a single master, ② multiple chunkservers, and is accessed by ③ multiple clients.
  - Each of these is typically a commodity Linux machine running a user-level server process.



Single Master (1/2)

- A GFS cluster consists ① a single master, ② multiple chunkservers, and is accessed by ③ multiple clients.
  - Each of these is typically a commodity Linux machine running a user-level server process.



Single Master (2/2)

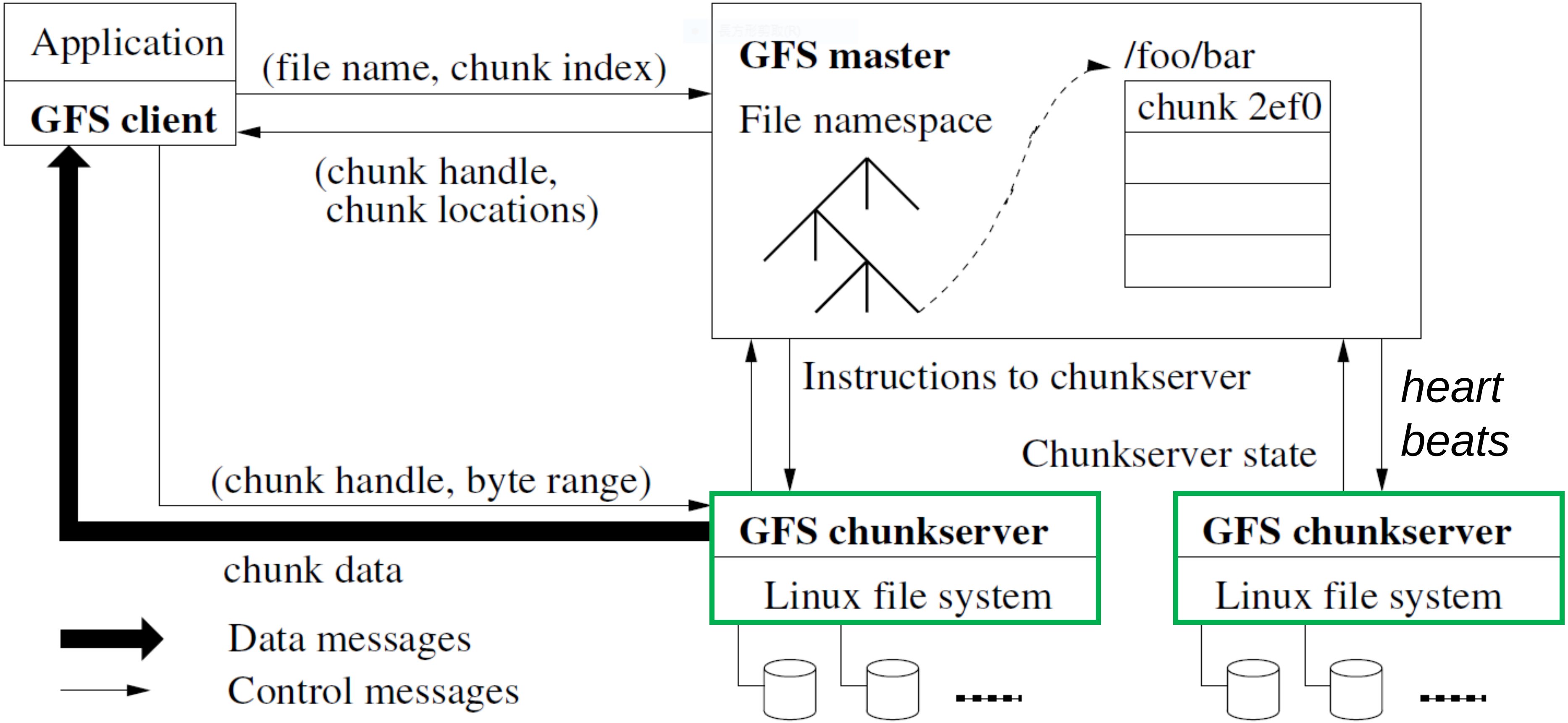
- Maintain all file system metadata:
  - Including namespace, access control information, the mapping from files to chunks, and the locations of chunks.

- Control system-wide activities:
  - Chunk replica placement
  - Chunk release management
  - Chunk migration between chunkservers (i.e., rebalancing)
  - Garbage collection of orphaned chunks

- Communicate periodically with each chunkserver in ***heartbeats*** to give it instructions and collect its state.

Multiple Chunkservers (1/2)

- A GFS cluster consists ① a single master, ② multiple chunkservers, and is accessed by ③ multiple clients.
  – Each of these is typically a commodity Linux machine running a user-level server process.

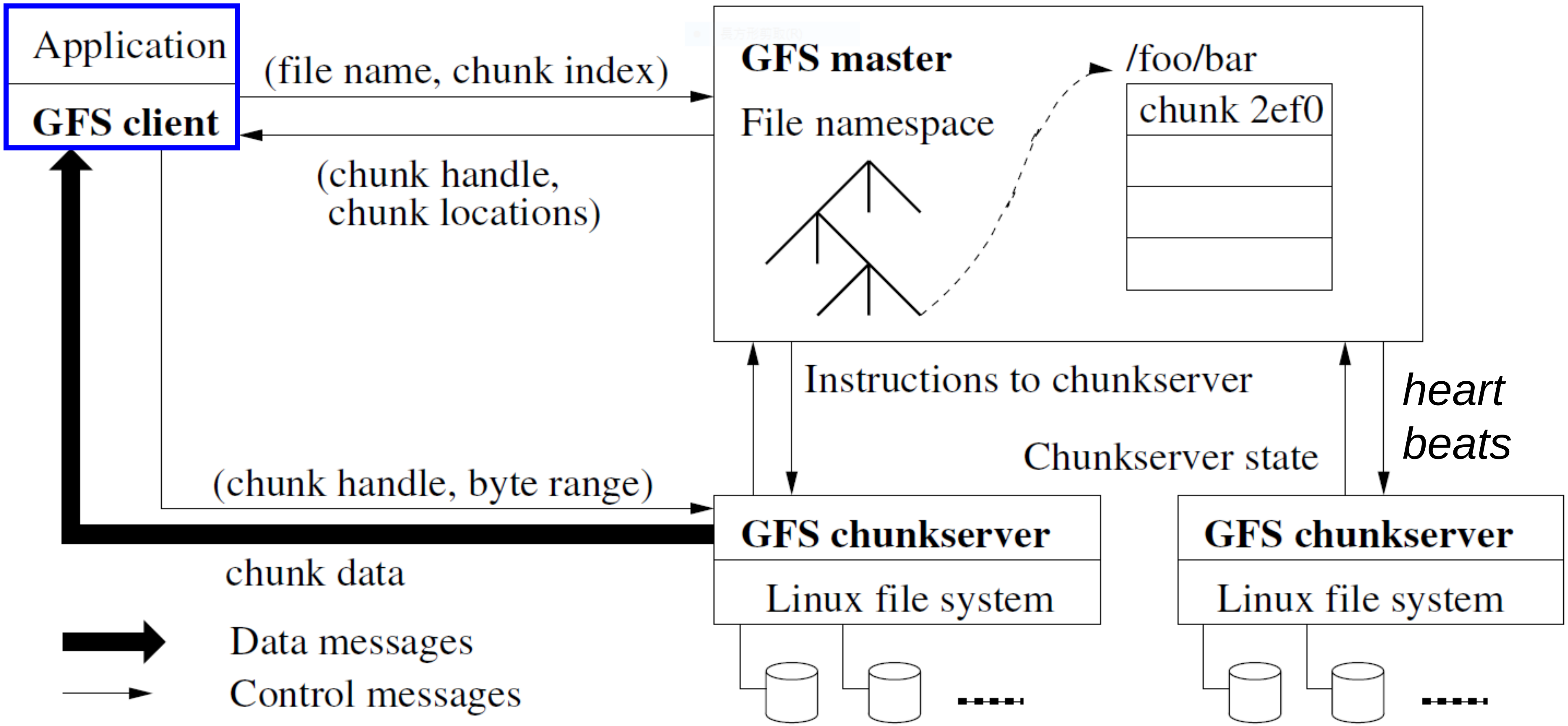


Multiple Chunkservers (2/2)

- Files are divided into fixed-size chunks, which can be identified by a unique chunk handle (*like FH in NFS*).
  - Chunks are stored on local disks of chunkservers as files.
  - Chunks are accessed by the chunk handle and byte range.
  - The chunk size (chosen 64 MB) is much larger than typical file system block sizes (e.g., 4 KB).
    - Reduce clients' need to **interact** with the master;
    - Reduce the size of **metadata** stored on the master;
    - Reduce the **network overhead** for consecutive workloads (e.g., search) by keeping a stable TCP connection.
  - Chunks are replicated across chunkservers (by default, three copies) for reliability concerns.
- Chunkservers need not cache file data.
  - Linux's buffer cache keeps frequently-accessed data.

Multiple Clients (1/2)

- A GFS cluster consists ① a single master, ② multiple chunkservers, and is accessed by ③ multiple clients.
  - Each of these is typically a commodity Linux machine running a user-level server process.



Multiple Clients (2/2)

- GFS client code, linked into the upper application, offers the file system API to communicate with master and chunkservers.

- Interact with the master for **metadata operations**
  - Clients can cache metadata to reduce the need to interact with the master.

- Interact with chunkservers for **direct data-bearing communications**
  - Clients cache **no** file data in its local memory.
    - Avoidance of cache coherence/consistency issues (existed in NFS!)
    - Limited benefits with streaming of large files and large working sets

File System Metadata

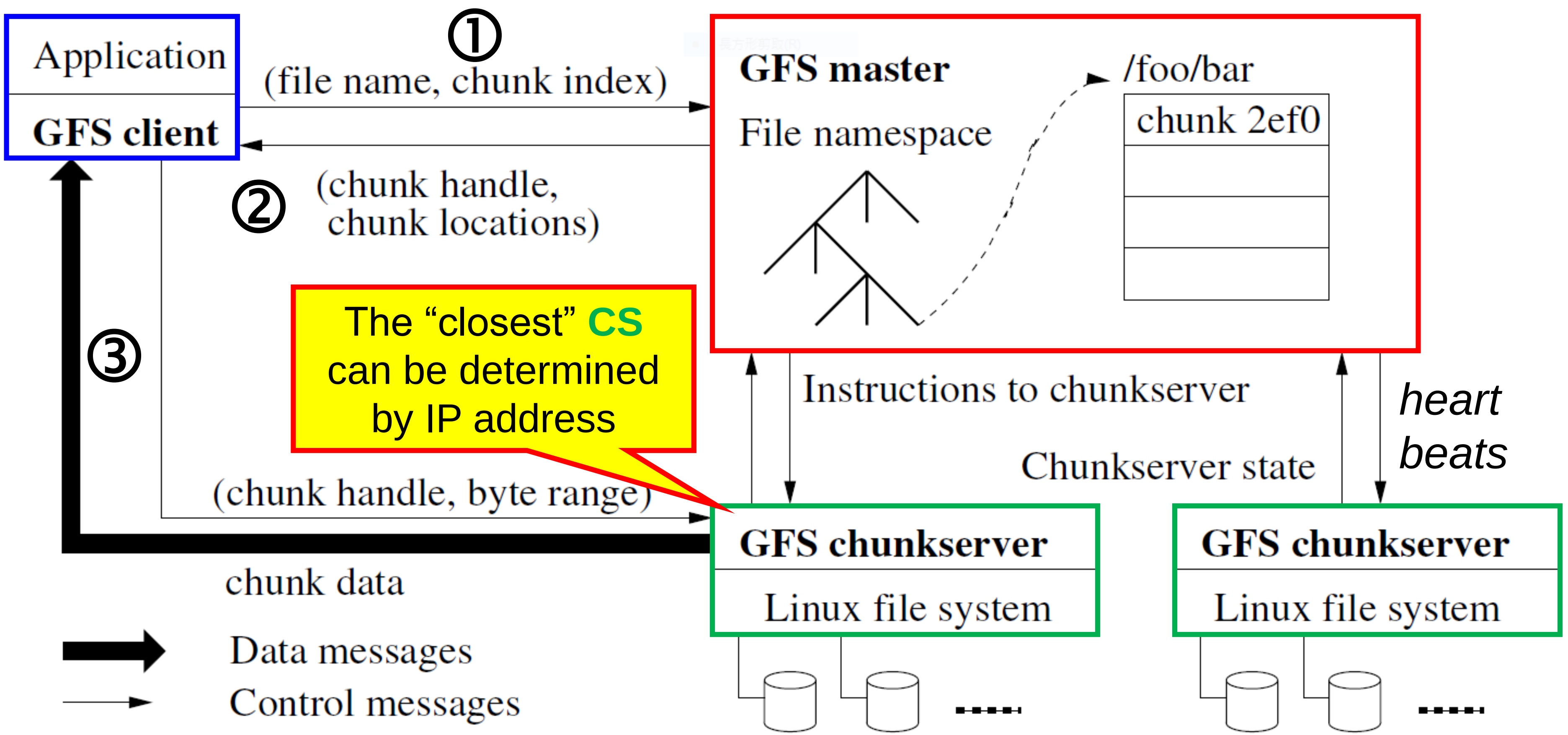
- The master maintains three types of FS metadata:
  ① The file and chunk **namespaces** (i.e., directory hierarchy) ;
  ② The **mapping** from files to chunks,
  ③ The locations of each chunk's **replicas**.
- The master keeps all three types of metadata in its **memory** for fast access.
  - Less than 64 bytes of metadata for each 64 MB chunk.
  - Less than 64 bytes per file if **prefix compression** is used.
- The master **persists** ① namespaces and ② file-to-chunk mapping in its local disks as an operation log.
  - But the master does not **persist** the chunk locations.
    - It can be pulled from chunkservers at startup via heartbeats.

Working Example: Client Reads

① **C** translates (file name and offset) into a chunk index, and sends a request to **M**

② **M** replies **C** the chunk handle and chunk locations

③ **C** requests for chunk **directly** from the "closest" **CS**



Working Example: Client Writes (1/2)

- A write must perform at all the chunk's replicas.
- A **mutation** is an operation that changes the contents or metadata of a **single chunk** over all replicas.
  - If a write exceeds the chunk boundary, the client must break it down into **multiple mutation operations**.
- The master uses leases to maintain a consistent mutation order across replicas.
  - The master grants a lease to one of replicas called *primary*.
    - The lease is designed to minimize management overhead at master.
    - A lease has an initial **timeout** of 60 seconds.
    - A lease can be **renewed** through heartbeats; the master can also **revoke** a lease before it expires.
  - The primary picks a serial order for all mutations on other replicas called *secondary*.

Working Example: Client Writes (2/2)

① **C** asks master for the **CSs** holding the *primary* and *secondary* replicas.

② **M** replies **C.**

③ **C** pushes the data to all the replicas in any order.

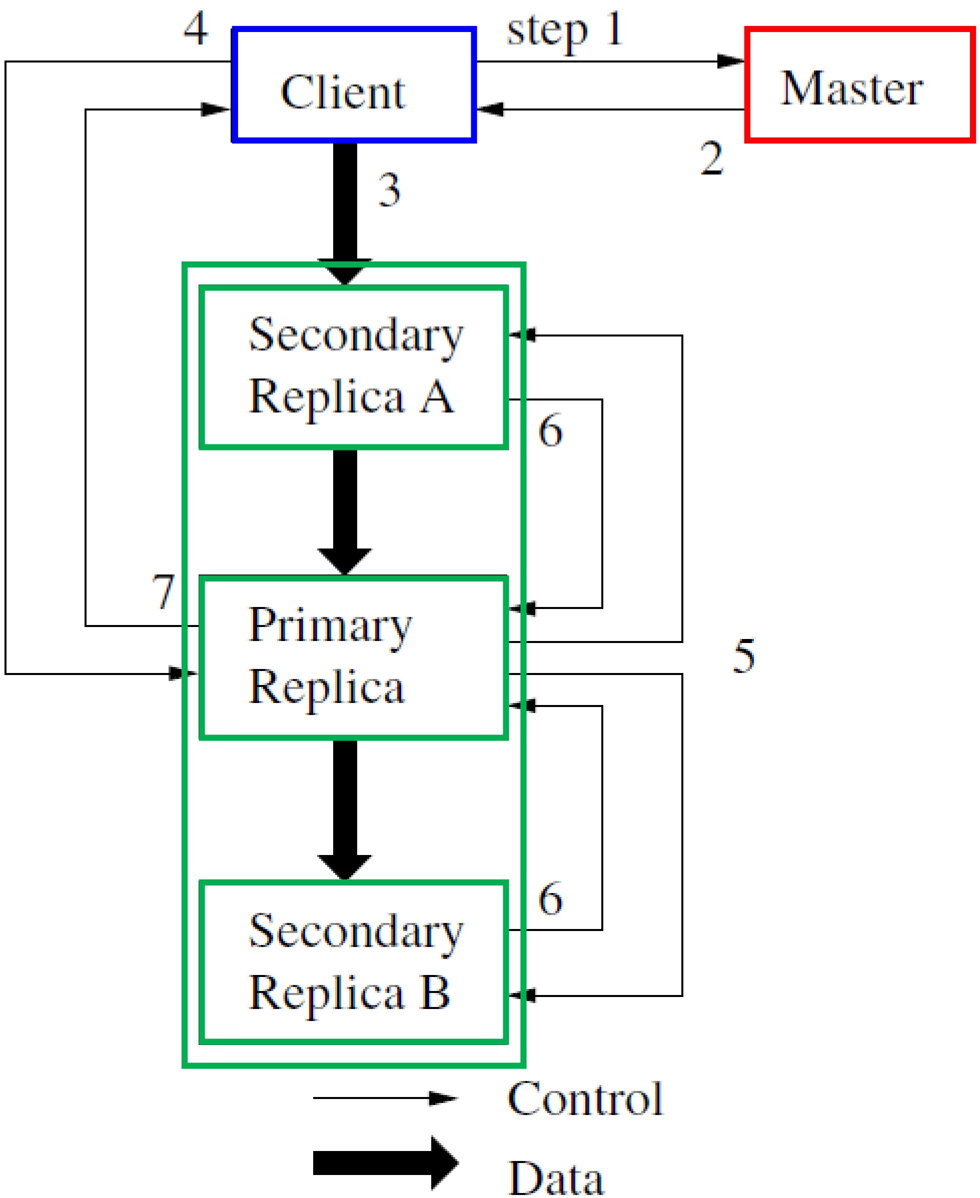
④ Once all acknowledged, **C** sends a write to the *primary.*

⑤ The *primary* forwards the write to all *secondary(s).*

⑥ The *secondary(s)* all reply to *primary* upon completed.

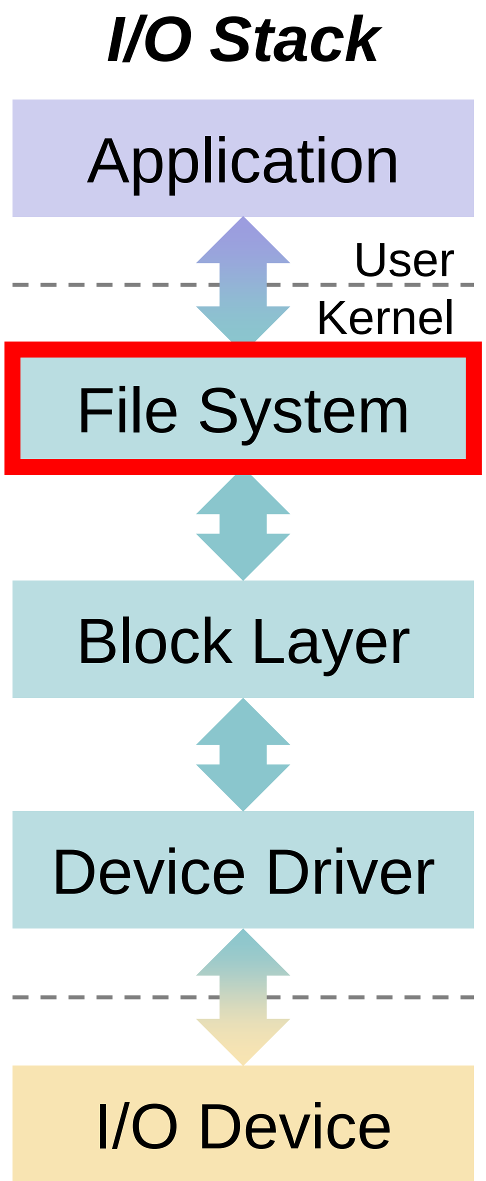
⑦ The *primary* replies to **C.**

- If some fail, retry ③~⑦ or all.



Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency



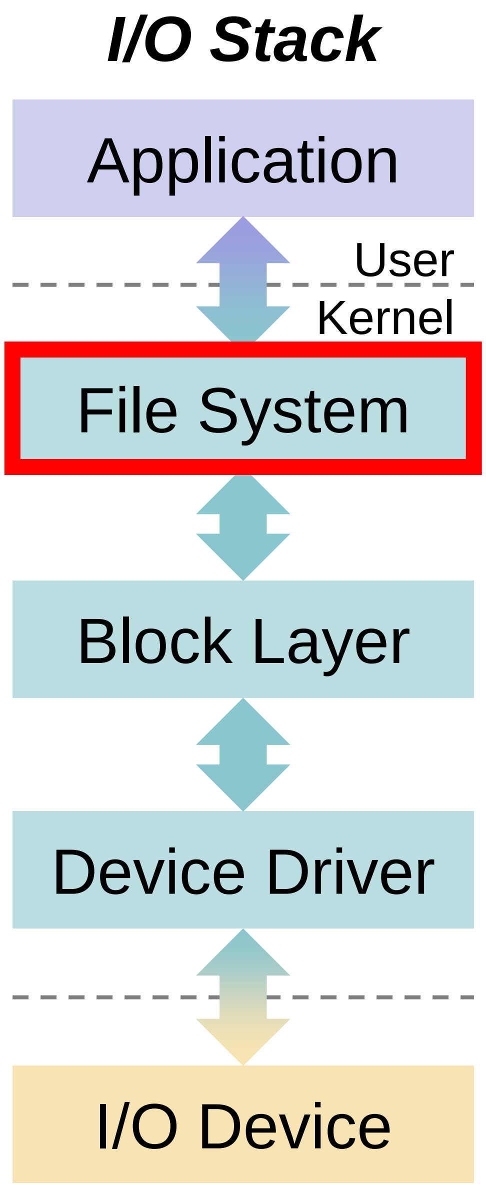
Record Appends

- **Workload Observation**: Clients may concurrently append large, sequential writes to files.
  – Concurrent writes to the same region are not serializable.
- GFS offers an atomic operation called **`record append`**.
  ① **C** pushes data to all replicas of the last chunk of the file.
  ② **C** sends the `record append` request to the *primary*.
  ③ If record fits within a chunk, the *primary* appends data to its replica and asks *secondary(s)* to write at the exact offset; otherwise, the *primary* **pads** the chunk to the maximum size, and asks **C** to retry the operation on the "next" chunk.
  ④ If a record append fails at any replica, **C** must **retry** but may result in **inconsistency**: The GFS application must cope with it.

Note: The record append is restricted to be at most one-fourth (i.e., 16 MB) of the maximum chunk size (64 MB).

Outline

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency



Relaxed Consistency (1/4)

- GFS guarantees a "relaxed consistency" model.
  - **File namespace operations are atomic**: They are handled by the master exclusively.
  - The states of a **file region** depend on ① the operation type (i.e., `write` or `record append`), ② whether the operation succeeds or fails, and ③ whether there're concurrent ones.
    - **"Relaxed" Consistent**: all clients see the same data in all replicas
    - **Defined**: ① a region is consistent after an operation, and ② clients see what the mutation has written in entirety

	Write	**Record Append**
Serial Success	*defined*	*defined* interspersed with *inconsistent*
Concurrent Successes	*consistent but undefined*	
Failure	*inconsistent*	

Relaxed Consistency (2/4)

- **Consistent**: all clients see the same data in all replicas
- **Defined**: ① a region is consistent after an operation, and ② clients see what the mutation has written in entirety

	Write	Record Append
Serial Success	*defined*	*defined* interspersed with *inconsistent*
Concurrent Successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Case: Write – Serial Success

defined

write(“Hello”, 9)
write(“World”, 10)

Chunk 1
9: Hello
10: World

→

Chunk 1’
9: Hello
10: World

Case: Write – Failure

inconsistent

write(“Hello”, 9)
write(“World”, 10)

Chunk 1
9: Hello
10: World

→

Chunk 1’
9: Hello

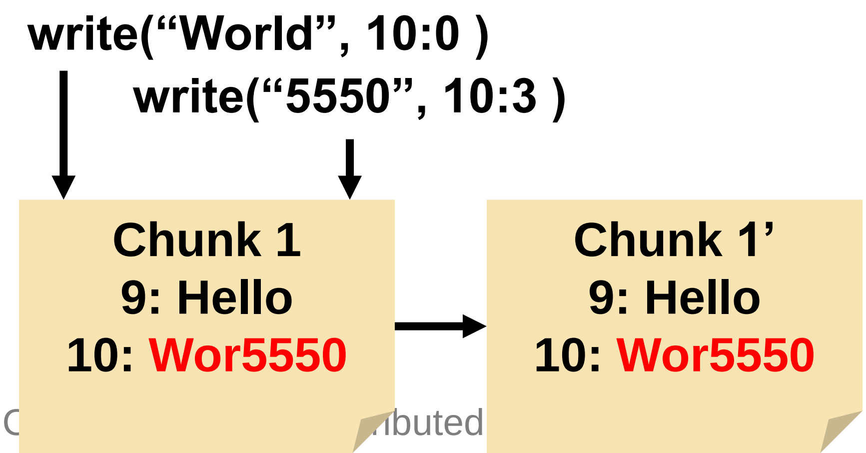
Relaxed Consistency (3/4)

- **Consistent**: all clients see the same data in all replicas
- **Defined**: ① a region is consistent <u>after an operation</u>, and ② clients see what the mutation has written in entirety

	Write	Record Append
Serial Success	*defined*	*defined* interspersed with *inconsistent*
Concurrent Successes	*consistent* but *undefined*	
Failure	*inconsistent*	

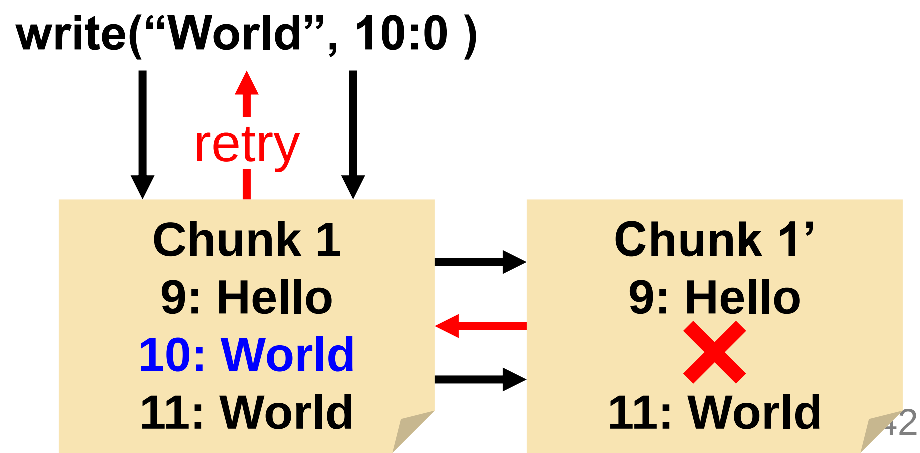
Case: Write – Concurrent Successes

consistent but *undefined*



Case: Record Append

defined but *inconsistent*



Relaxed Consistency (4/4)

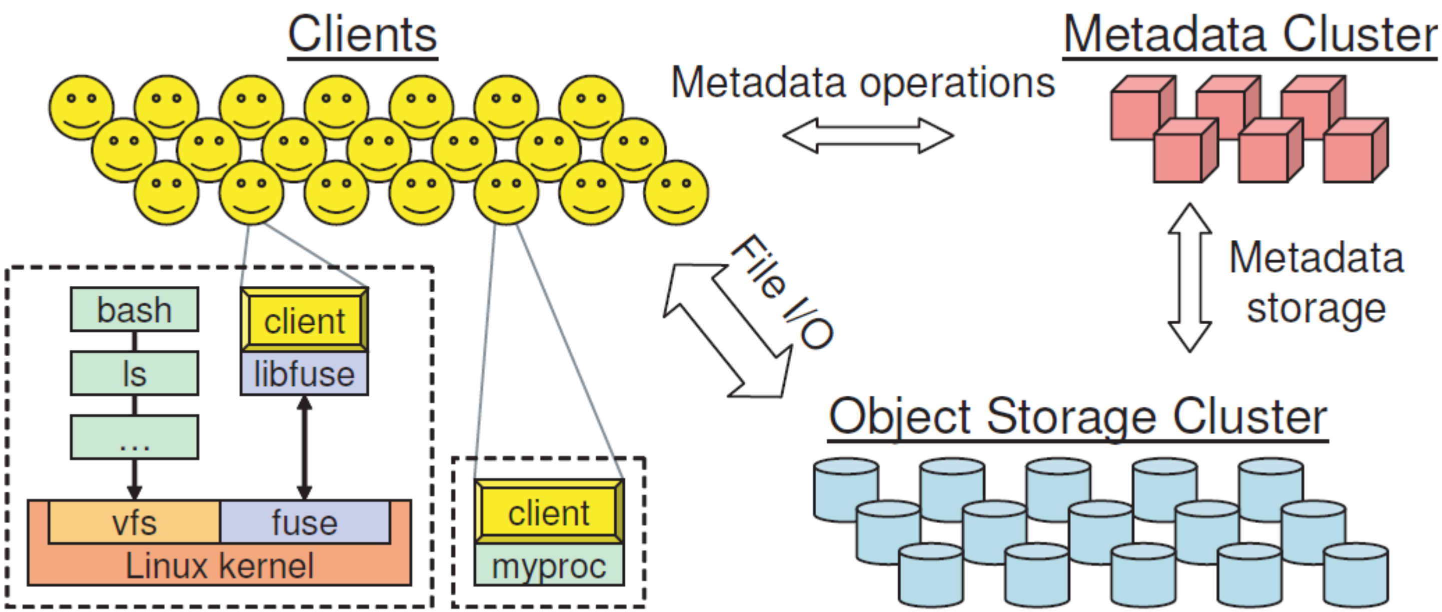
- Concurrent **`writes`** may result in consistent but undefined:
  - All clients see the same data, but it may not reflect what any mutation has written.
  - The order is not guaranteed; a region may contain fragments from multiple clients.

- **`Record append`** ensures a record is appended **atomically at least once**, but at an offset chosen by the primary.
  - Applications need to deal with possible duplicates.

GFS Limitations

① Single master simplifies the coordination, but it may become the single point of failure.

– *Ceph: A Scalable, High-Performance Distributed File System (OSDI'06)*



② Relaxed consistency burdens the GFS applications.

Other Distributed File Systems

- Ceph: A Scalable, High-Performance Distributed File System (OSDI'06)
- Hadoop Distributed File System (by Yahoo!)
- GlusterFS

WIKIPEDIA
The Free Encyclopedia

Summary

- Network File System (NFS)
  - Client-Server Model
  - NFSv2: A Stateless File Protocol
  - Handling Server Failures
  - Client-side Caching / Buffering
  - Server-side Caching / Buffering
- The Google File System (GFS)
  - Design Considerations and Assumptions
  - GFS Architecture
  - Record Appends
  - Relaxed Consistency