TECHNISCHE UNIVERSITÄT DRESDEN

**Department of Computer Science** Institute of System Architecture, Operating Systems Group

# DISTRIBUTED FILE SYSTEMS

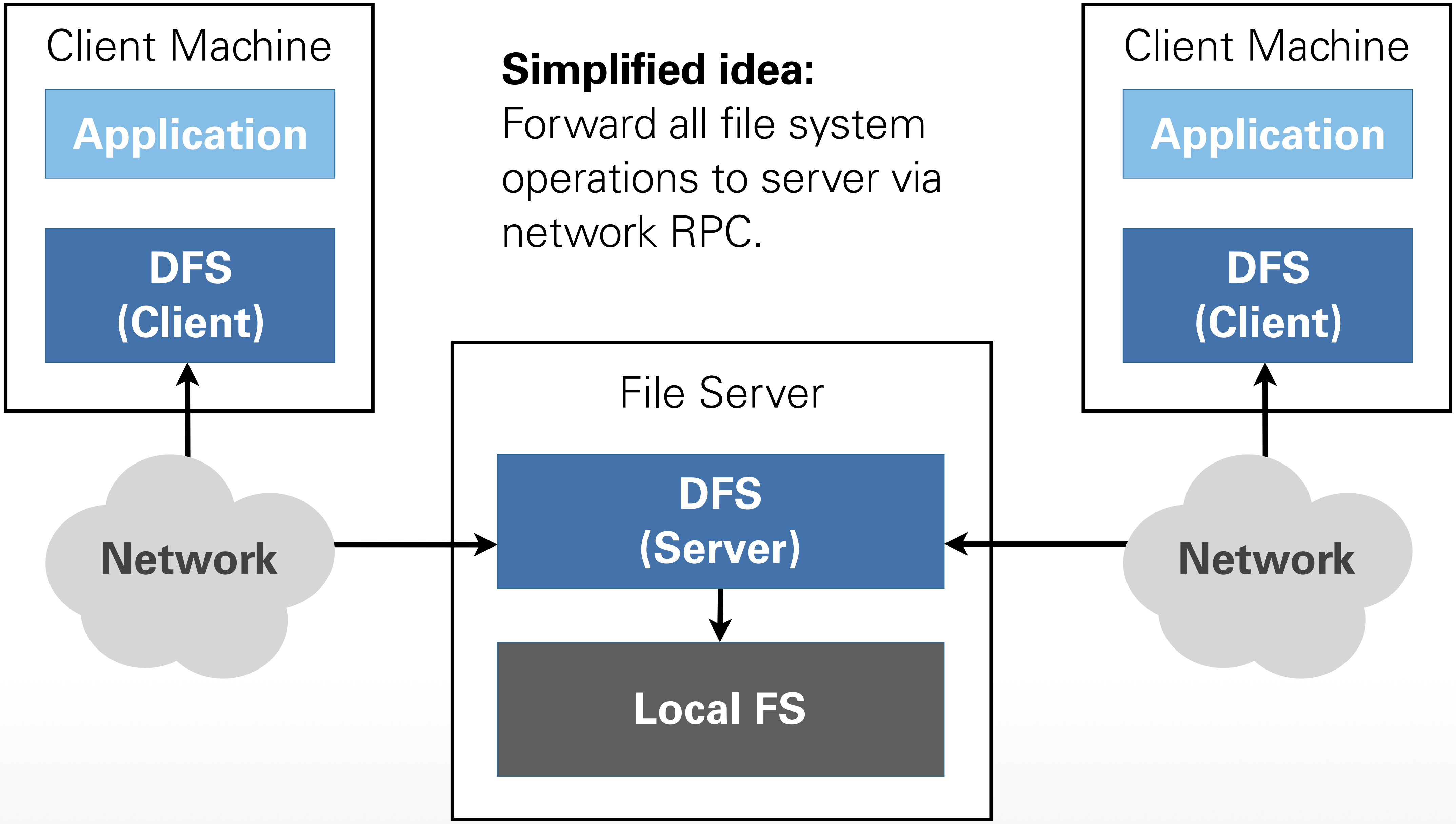
CARSTEN WEINHOLD

# OUTLINE

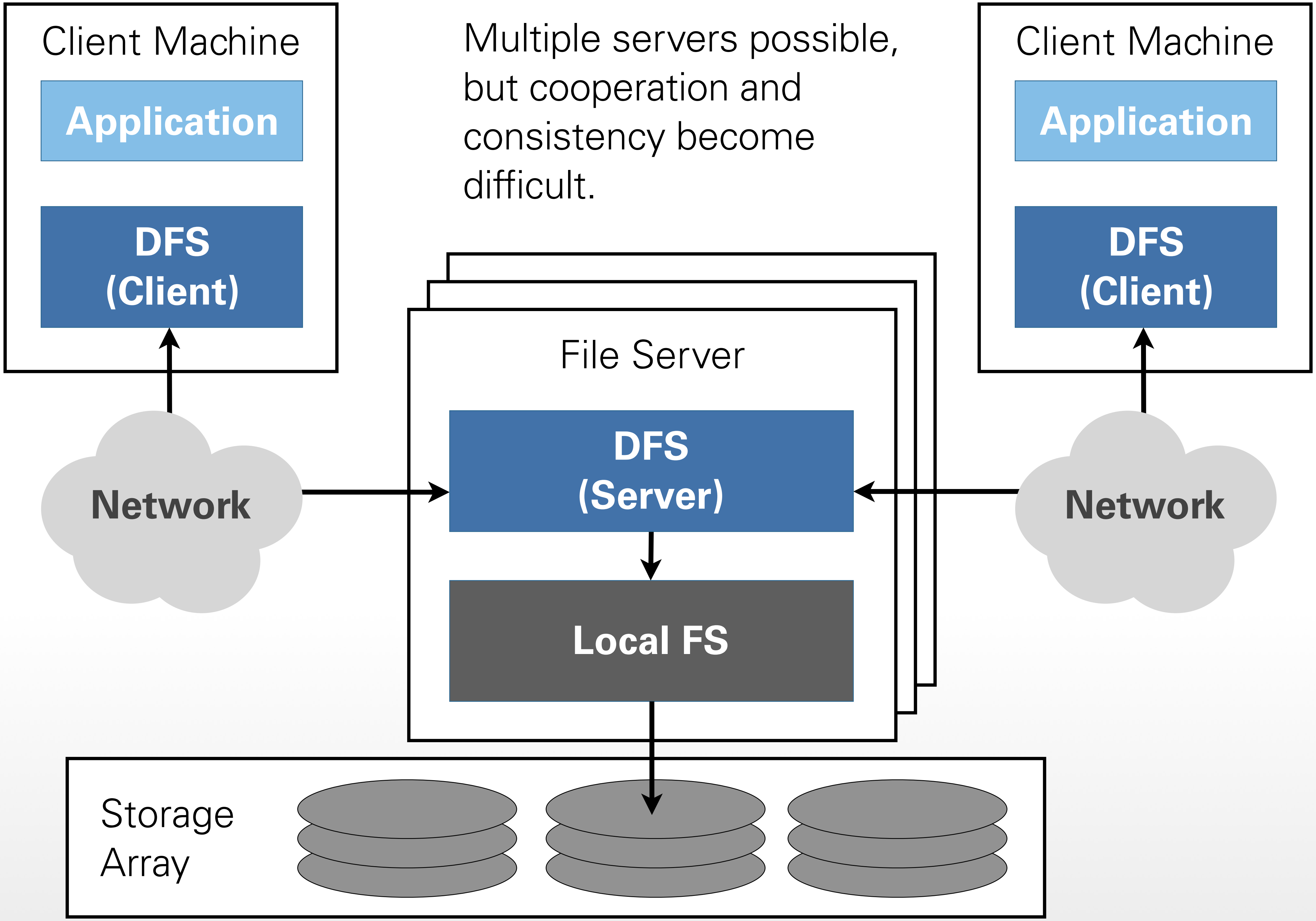
- Classical distributed file systems
  - NFS: Sun Network File System
  - AFS: Andrew File System
- Parallel distributed file systems
- Case study: The Google File System
  - Scalability
  - Fault tolerance
- Other approaches

# DFS ARCHITECTURE

**Simplified idea:**
Forward all file system operations to server via network RPC.



# DFS ARCHITECTURE



# SUN NFS (V2, V3)

| | |
|---|---|
| API | As close to UNIX as possible |
| Names/Lookup | Message to file server for each path element |
| Open/Close | Unique NFS handle per file, no state on server |
| Read/Write | Messages to read/write blocks, small block size |
| Caching (client) | Metadata (e.g., NFS handle), data blocks |
| Consistency | Consistency messages exchanged regularly, clients might see stale data/metadata |
| Replication | Multiple read-only servers (if synced manually) |
| Fault Handling | Write through on server (v2), idempotent client writes, clients block if server crashed |

# ANDREW FILE SYSTEM

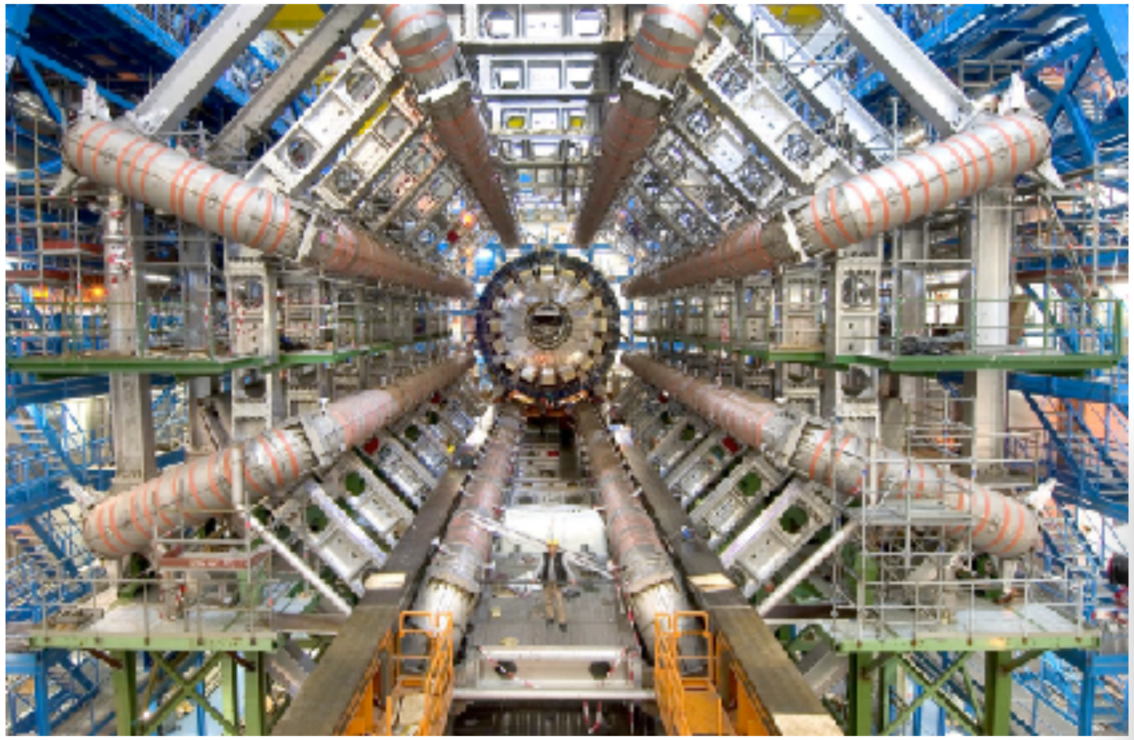
| | |
|---|---|
| API | As close to UNIX as possible |
| Names/Lookup | Name resolution on client, uses dir caches |
| Open/Close | Local file, might need to transmit from/to server |
| Read/Write | Local file, but some work in open/close phase |
| Caching (client) | Complete files, LRU replacement if needed |
| Consistency | Callback promises: server informs client, if another client wants to modify a cached file |
| Replication | Pool of servers, may improve performance |
| Fault Handling | Some (e.g., client can still access files in local cache if network or servers fail) |

TECHNISCHE UNIVERSITÄT DRESDEN

# DFS SCALABILITY

- Work well for home directories (e.g., AFS)
- POSIX consistency causes complexity:
  - Cache coherency traffic (e.g., AFS callbacks)
  - Write semantics (e.g., may need distributed locks for concurrent writes to same file)
- One-to-one mapping:
  - File in DFS is file on server (higher load?)
- Servers must cache both metadata+data

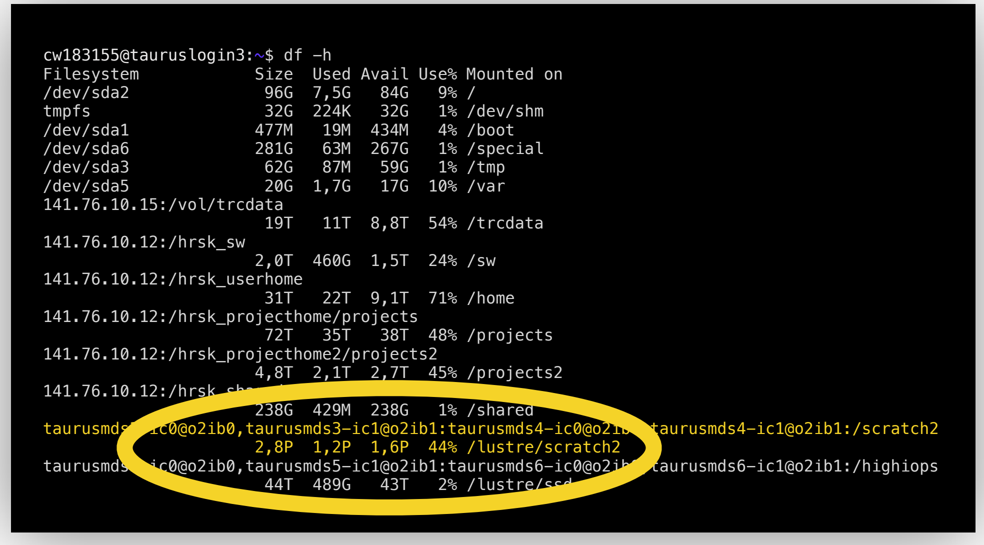
# BIG DATA DEMANDS



ATLAS Experiment © 2012 CERN, Image source: http://www.atlas.ch/photos/full-detector-photos.html

**Scientific Computing:**

Approximately **1 GB/s** of data generated at the Worldwide LHC Computing Grid. This is after two filtering stages ... [3]




**Social Media:**

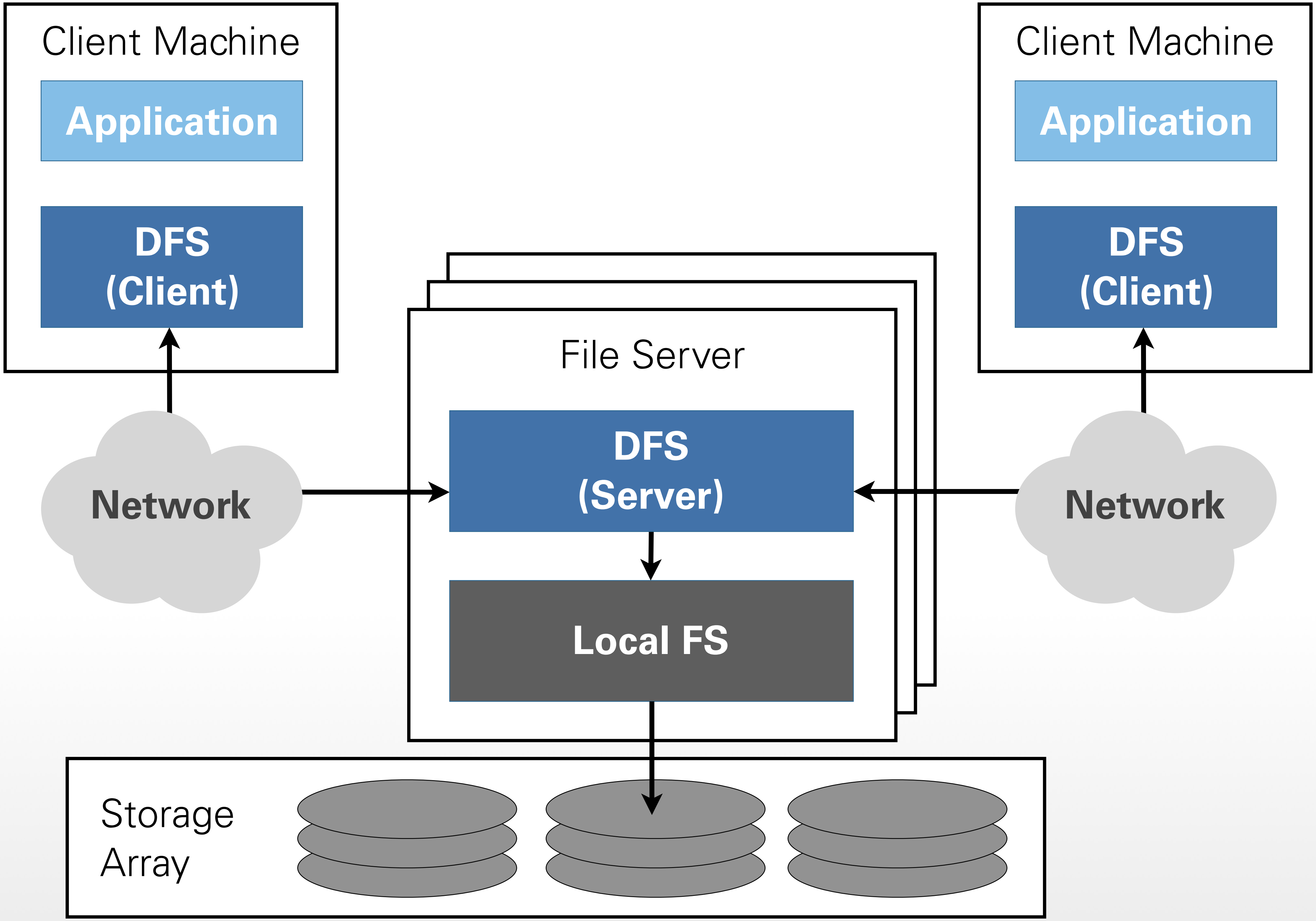
„Facebook serves over **one million images per second** at peak. [...] our previous approach [...] leveraged network attached storage appliances over **NFS**. Our key observation is that **this traditional design** incurs an **excessive number of disk operations** because of metadata lookups.“ [4]

Image source: http://facebook.com

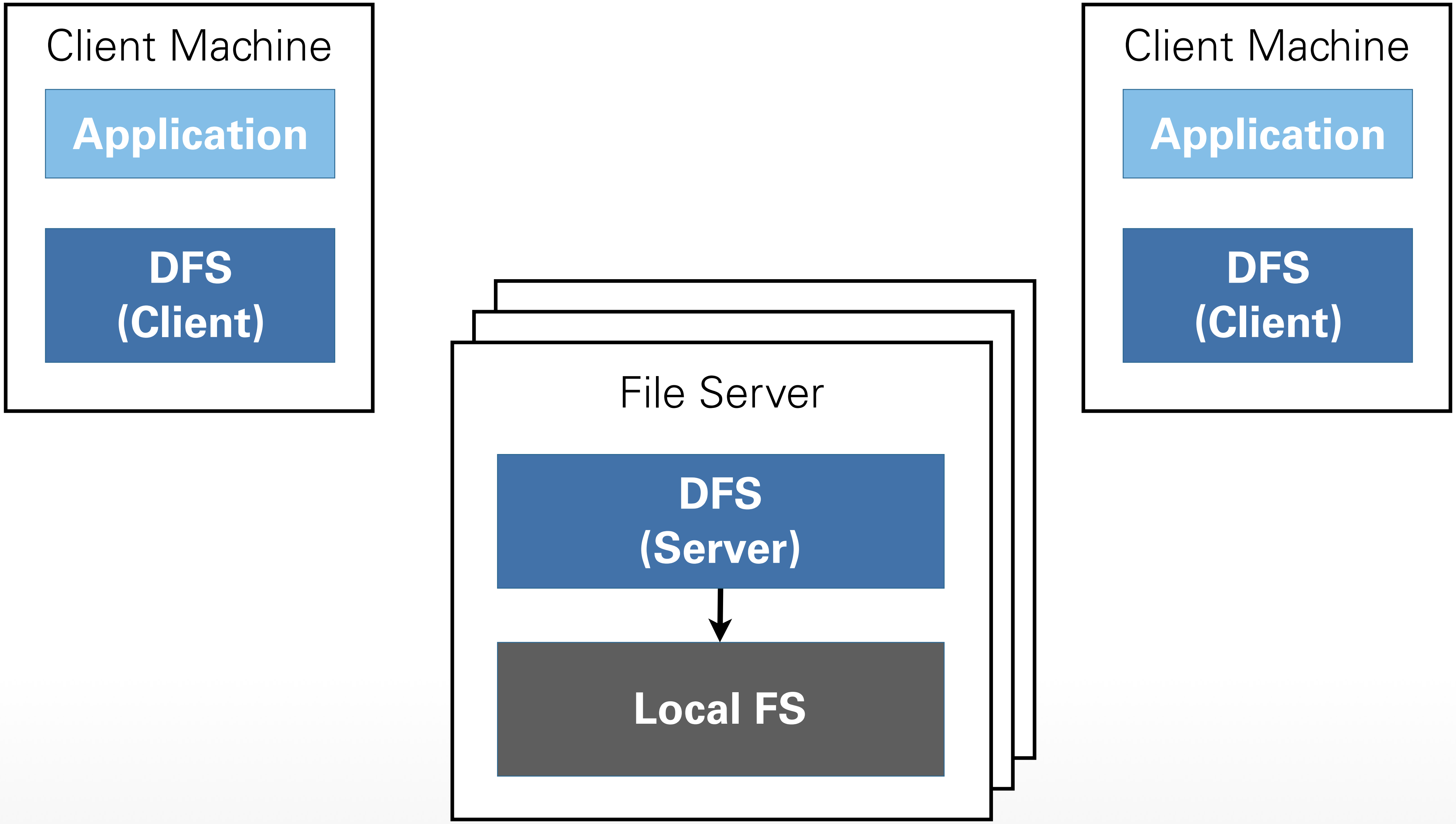
# OUTLINE

- Classical distributed file systems
  - NFS: Sun Network File System
  - AFS: Andrew File System
- Parallel distributed file systems
- Case study: The Google File System
  - Scalability
  - Fault tolerance
- Other approaches

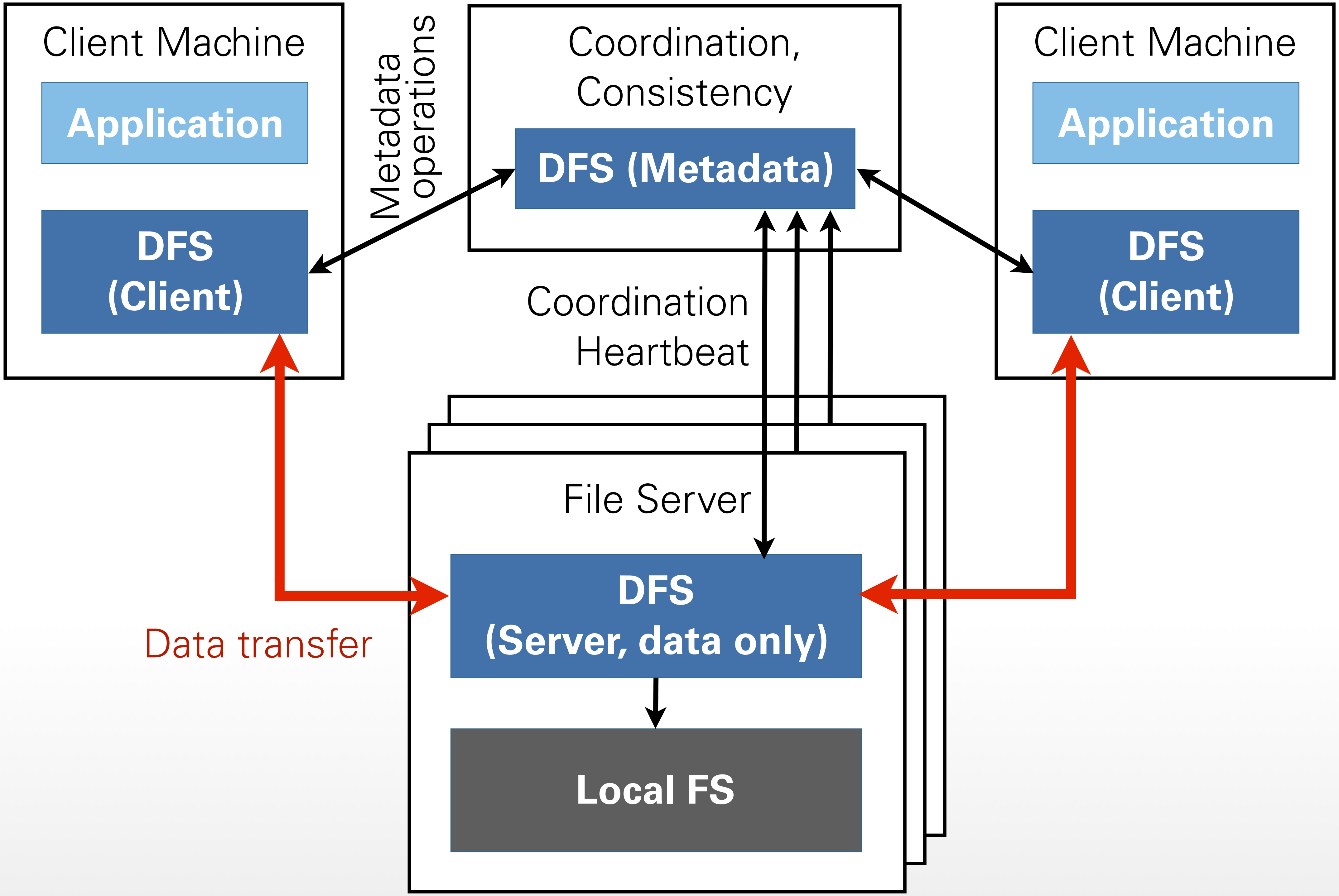
# DFS ARCHITECTURE



# DFS ARCHITECTURE



# PARALLEL FILE SYSTEMS



# LARGER DESIGN SPACE

- **Better load balancing:**
  - Few servers handle metadata only
  - Many servers serve (their) data
- **More flexibility, more options:**
  - Replication, fault tolerance built in
  - Specialized APIs for different workloads
  - Lower hardware requirements per machine
  - Client and data server on same machine

# PARALLEL FILE SYSTEMS

- Lustre
- GFS = Google File System
- GPFS
- PVFS
- HadoopFS
- TidyFS
- ...

TECHNISCHE UNIVERSITÄT DRESDEN

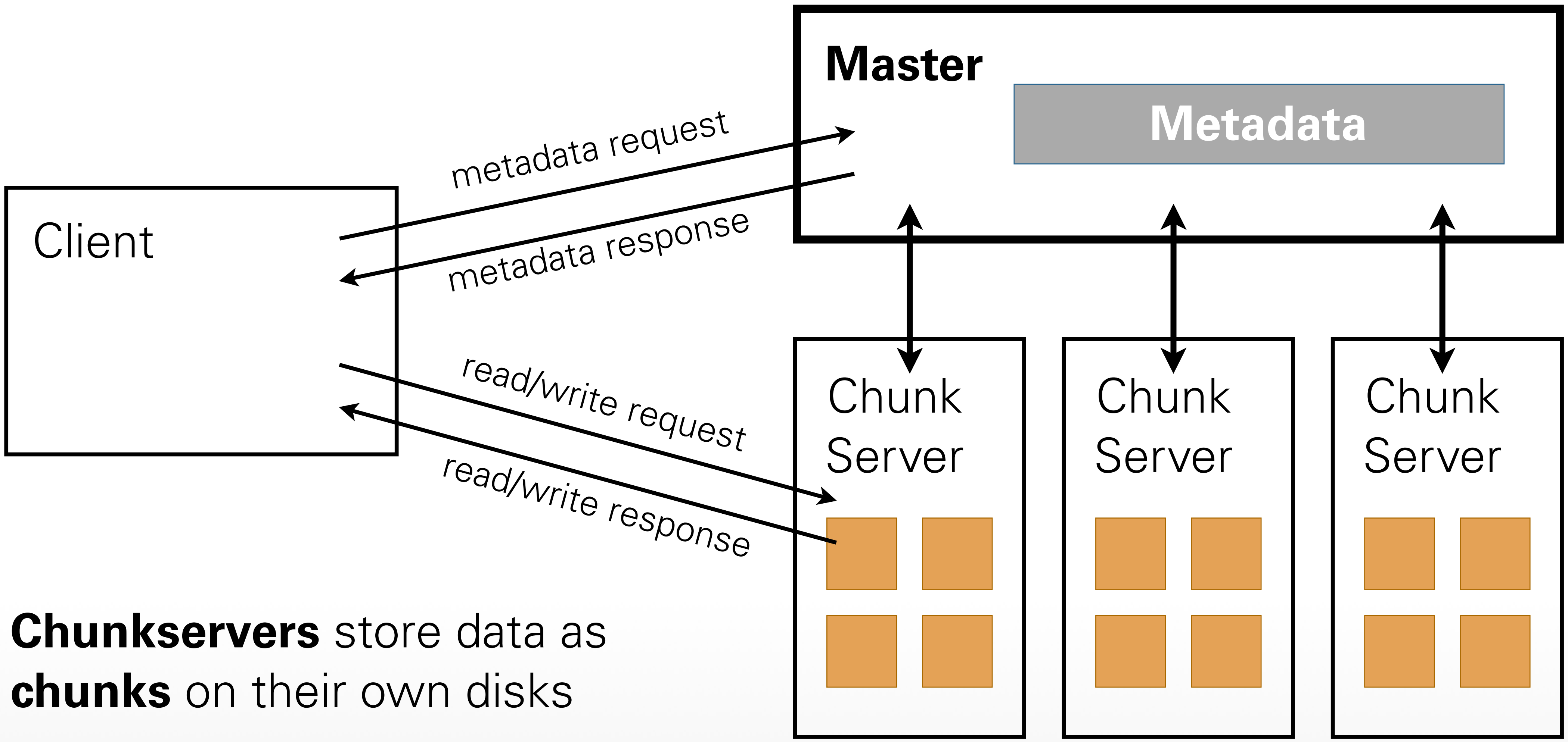
# OUTLINE

- Classical distributed file systems
  - NFS: Sun Network File System
  - AFS: Andrew File System
- Parallel distributed file systems
- Case study: The Google File System
  - Scalability
  - Fault tolerance
- Other approaches

# GFS KEY DESIGN GOALS

- **Scalability:**
  - High throughput, parallel reads/writes
- **Fault tolerance** built in:
  - Commodity components might fail often
  - Network partitions can happen
- **Re-examine** standard I/O semantics:
  - Complicated POSIX semantics *vs* scalable primitives *vs* common workloads
  - Co-design file system and applications

# GFS ARCHITECTURE




- **Chunkservers** store data as **chunks** on their own disks
- **Master** manages **metadata** (e.g., which chunks belong to which file, etc.)

Source [2]

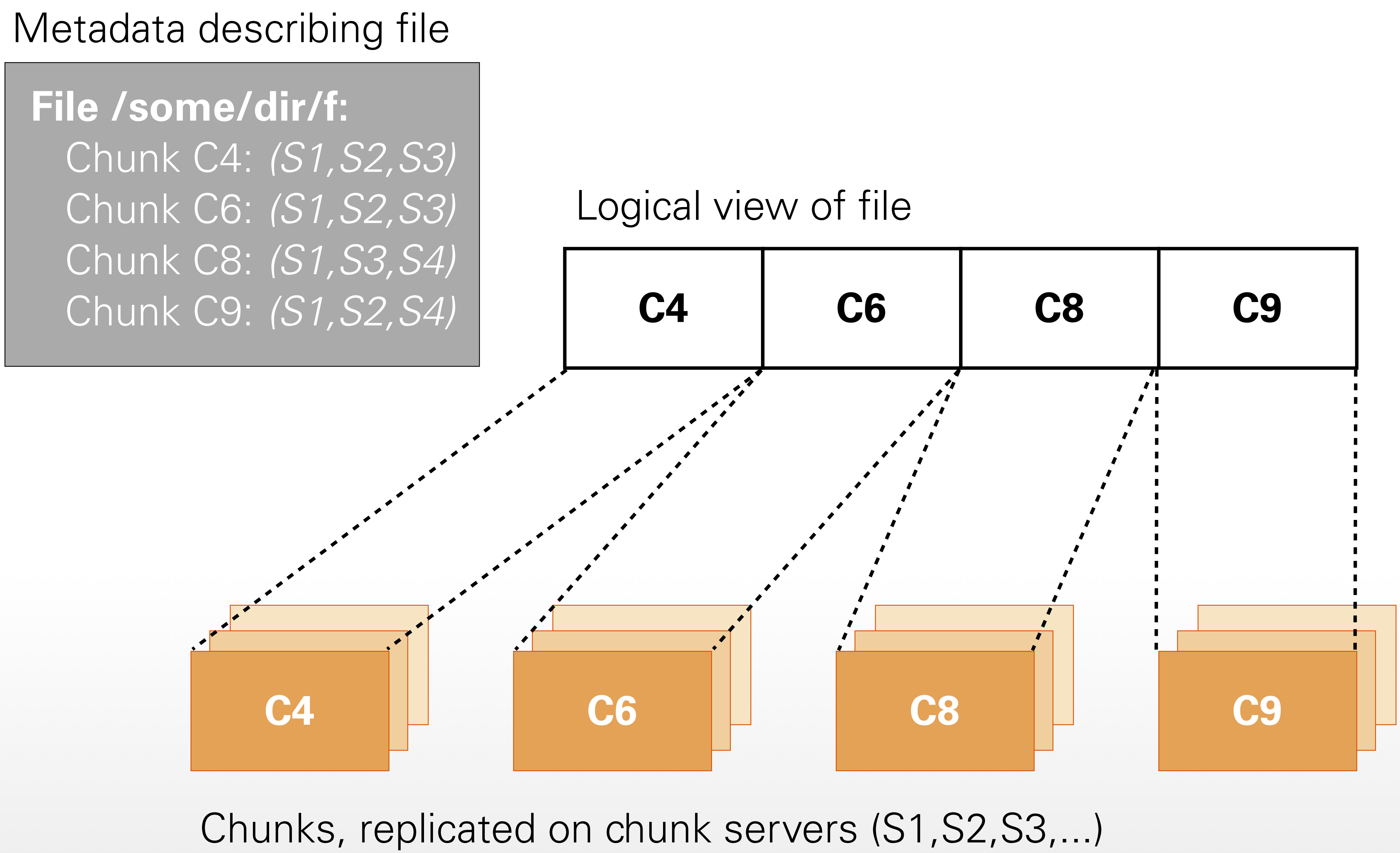
# MASTER & METADATA

- **Master** is process on separate machine
- Manages all **metadata:**
  - File namespace
  - File-to-chunk mappings
  - Chunk location information
  - Chunk version information
  - Access control information
- Does <u>not</u> store/read/write any file data!

# FILES & CHUNKS

- **Files** are made of (multiple) **chunks:**
  - Chunk size: 64 MiB
  - Stored on **chunkserver**, in Linux file system
  - Referenced by **chunk handle** (i.e., filename in Linux file system)
  - **Replicated** across multiple chunkservers
  - Chunkservers located in different racks

# FILES & CHUNKS



# ACCESSING FILES

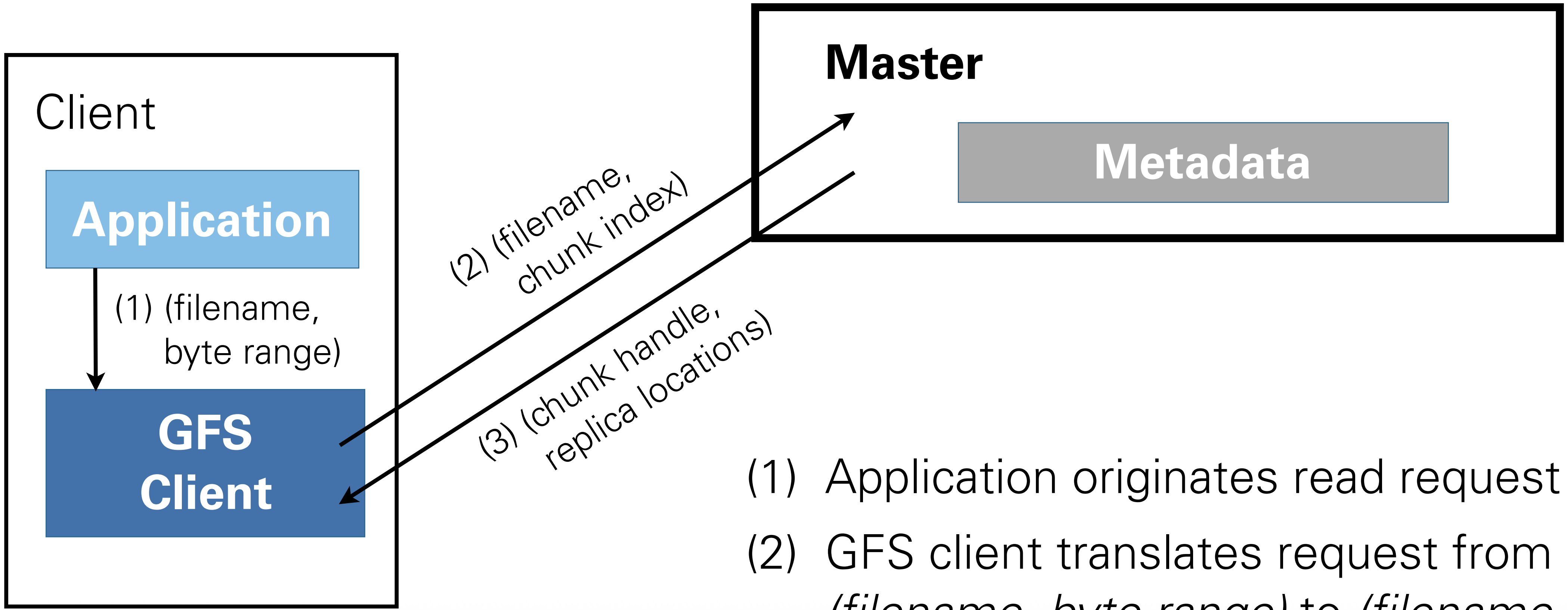
- **Client accesses file in two steps:**

  (1) Contact Master to retrieve metadata

  (2) Talk to chunkservers directly

- **Benefits:**
  - Metadata is small (one master can handle it)
  - Metadata can be cached at client
  - Master not involved in data operations
- **Note:** clients cache metadata, but <u>not</u> data

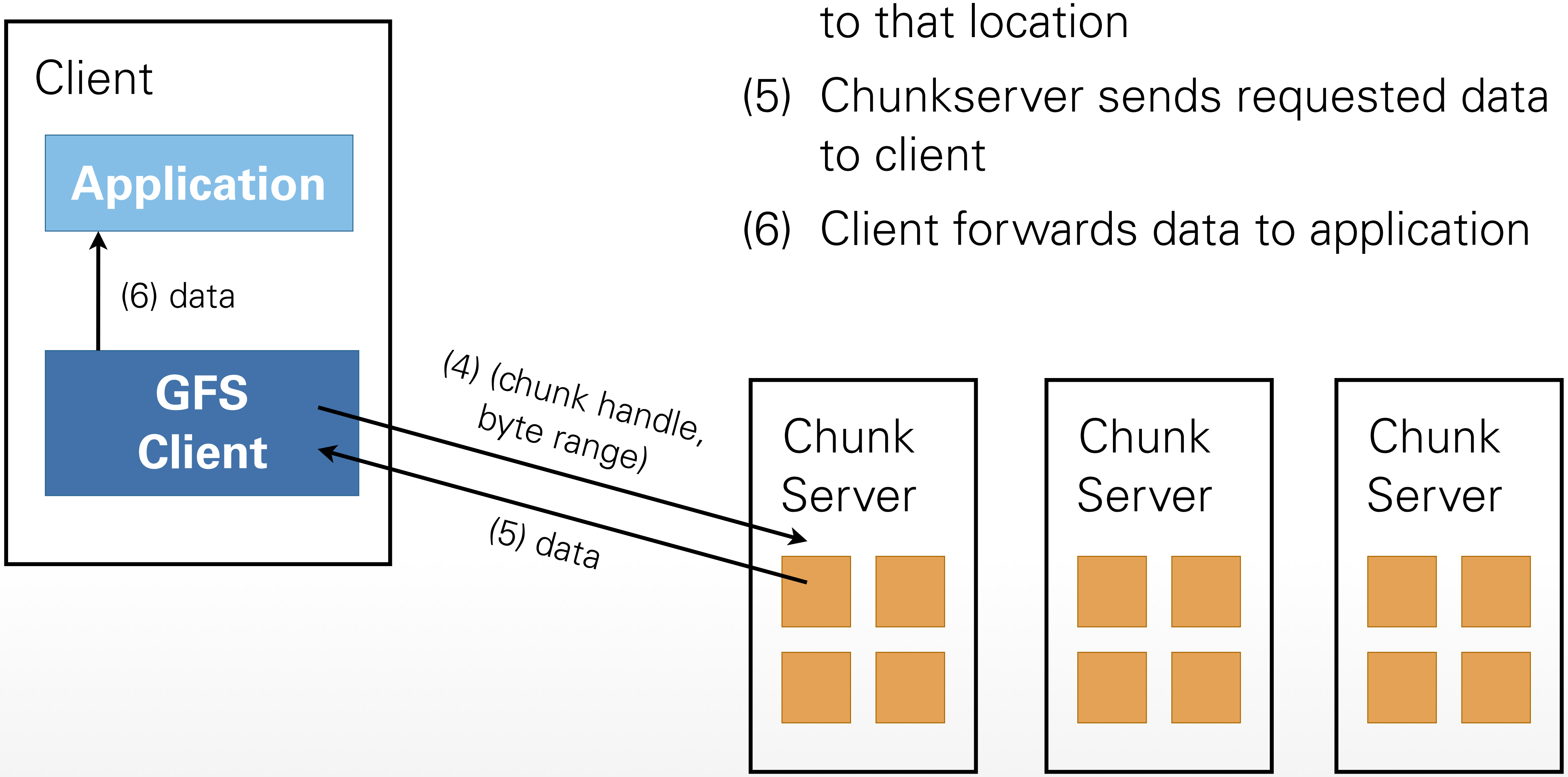
# READ ALGORITHM




(1) Application originates read request
(2) GFS client translates request from *(filename, byte range)* to *(filename, chunk index)* and sends it to master
(3) Master responds with *(chunk handle, replica locations)*

Source [6]

# READ ALGORITHM




(4) GFS client picks location and sends *(chunk handle, byte range)* request to that location

(5) Chunkserver sends requested data to client

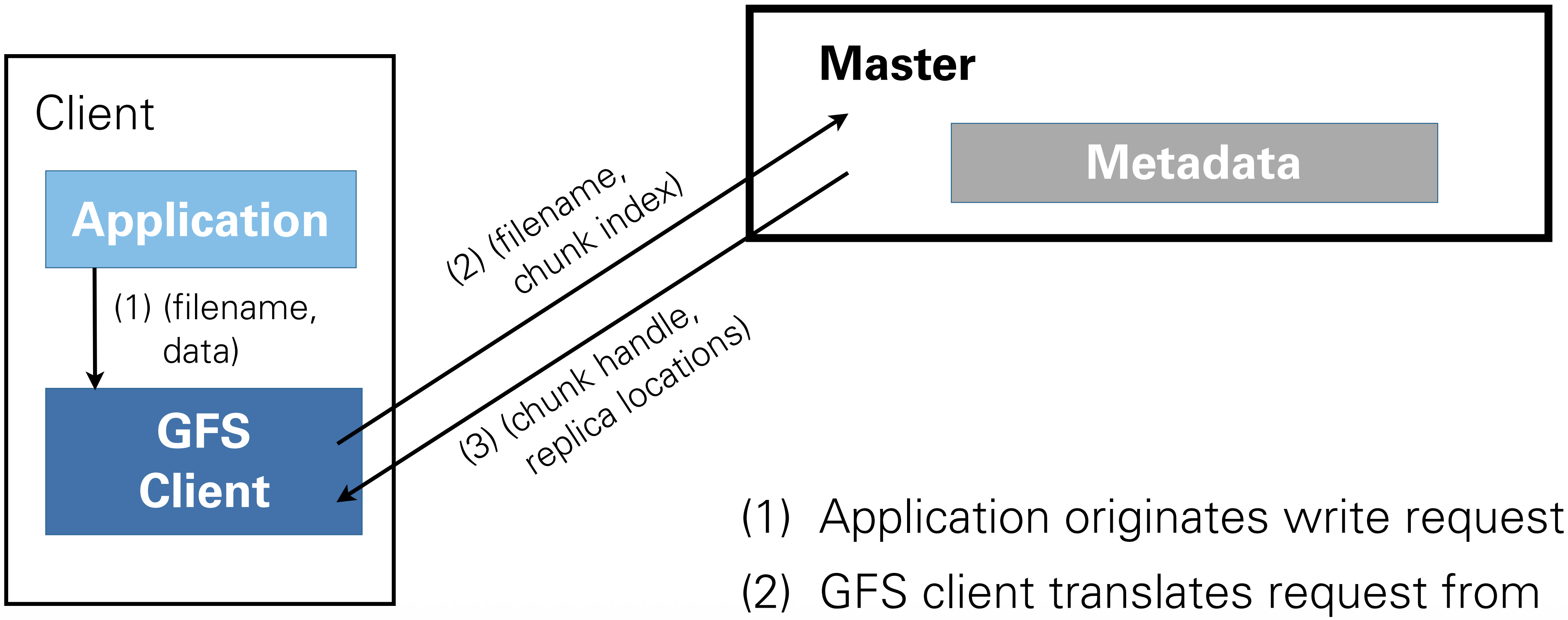
(6) Client forwards data to application

Source [6]

# READ PERFORMANCE

- Division of work reduces load:
  - Master provides metadata quickly (in RAM)
  - Multiple chunkservers available
  - One chunkserver (e.g., the closest one) is selected for delivering requested data
  - Chunk replicas equally distributed across chunkservers for load balancing
- Can we do this for writes, too?

# WRITE ALGORITHM




(1) Application originates write request

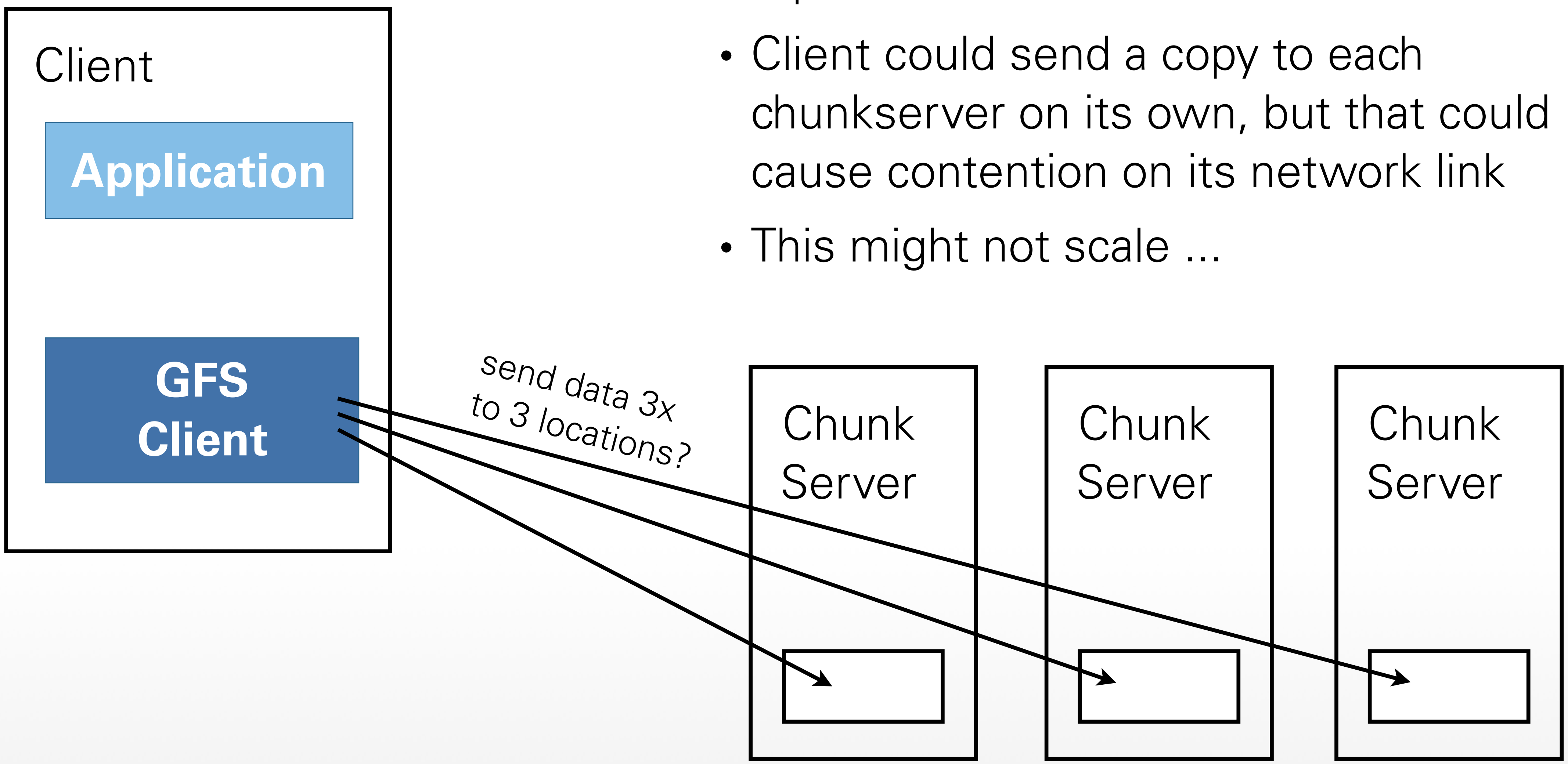
(2) GFS client translates request from *(filename, data)* to *(filename, chunk index)* and sends it to master

(3) Master responds with *(chunk handle, replica locations)*

Source [6]

# HOW TO WRITE DATA?

- Data needs to be pushed to all chunkservers to reach required replication count
- Client could send a copy to each chunkserver on its own, but that could cause contention on its network link
- This might not scale ...



# PIPELINING WRITES

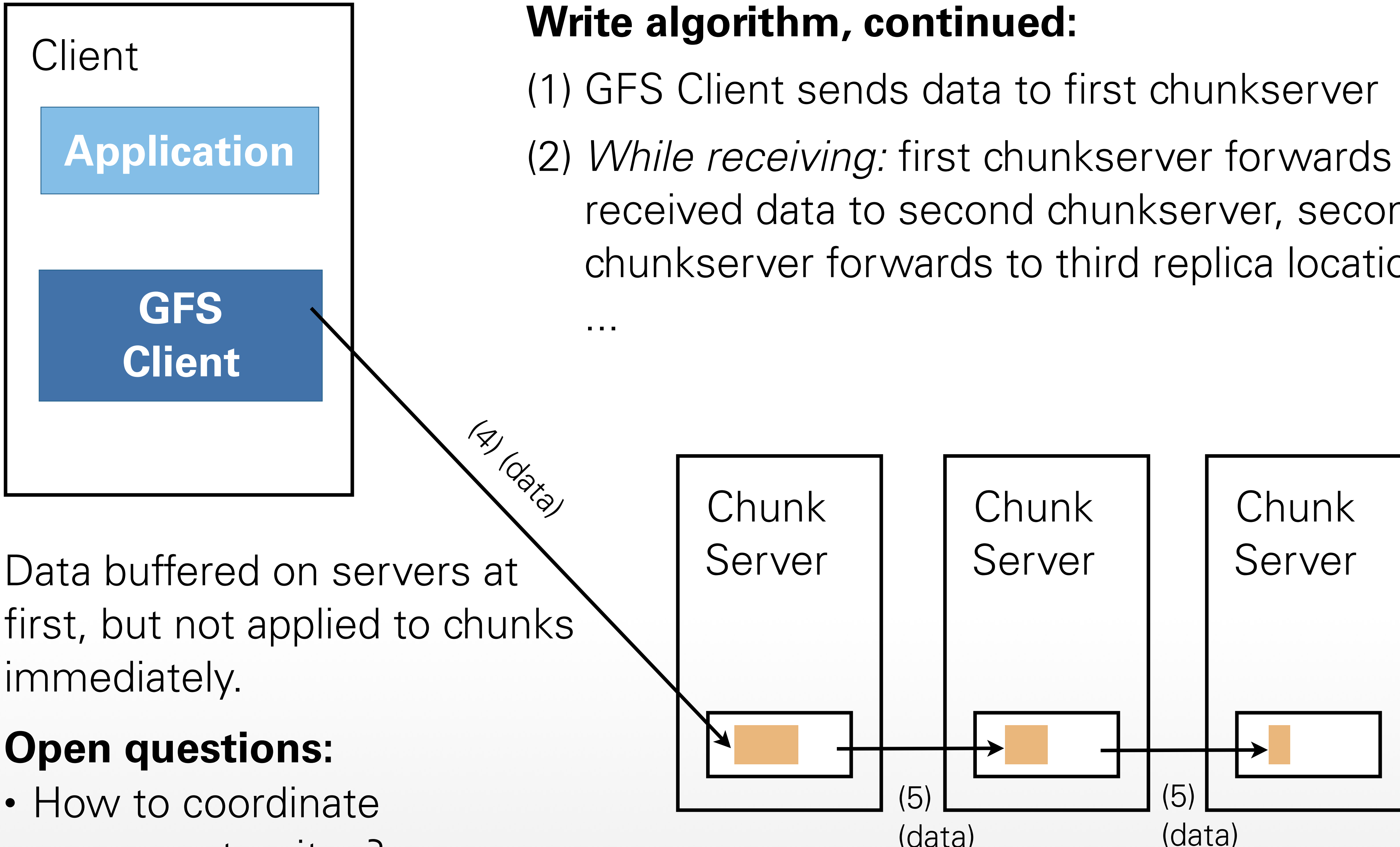
- Sending data over client's network card multiple times is inefficient, but ...
- ... network cards can send <u>and</u> receive at the same time at full speed (full duplex)
- **Idea:** Pipeline data writes
  - Client sends data to just one chunkserver
  - Chunkserver starts forwarding data to next chunkserver, while still receiving more data
  - Multiple links utilized, lower latency

# WRITE ALGORITHM

**Write algorithm, continued:**

(1) GFS Client sends data to first chunkserver

(2) *While receiving:* first chunkserver forwards received data to second chunkserver, second chunkserver forwards to third replica location, ...




Data buffered on servers at first, but not applied to chunks immediately.

**Open questions:**

- How to coordinate concurrent writes?
- What if a write fails?

Source [6]

# REPLICA TYPES

- **Primary:**
  - Determines serial order of pending writes
  - Forwards write command + serial order
- **Secondary:**
  - Execute writes as ordered by primary
  - Replies to primary (success or failure)
- **Replica roles determined by Master:**
  - Tells client in step (2) of write algorithm
  - Decided <u>per chunk</u>, not per chunkserver

# WRITE ALGORITHM

**Write algorithm, continued:**

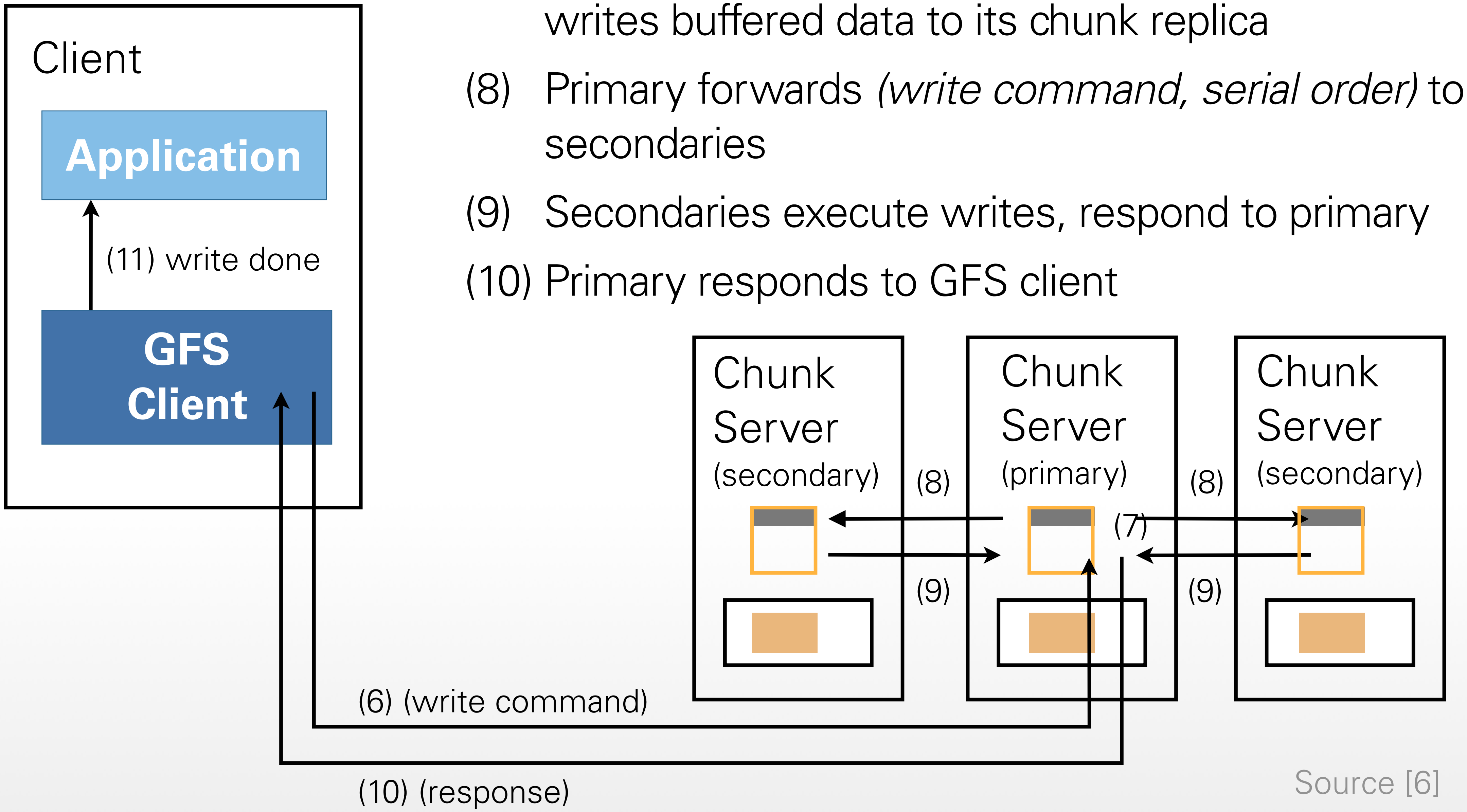
(6) GFS Client sends write command to primary

(7) Primary determines serial order of writes, then writes buffered data to its chunk replica

(8) Primary forwards *(write command, serial order)* to secondaries

(9) Secondaries execute writes, respond to primary

(10) Primary responds to GFS client




Source [6]

# WRITE SEMANTICS

- Multiple clients can write concurrently
- Things to consider:
  - Clients determine offset in chunk
  - Concurrent writes to overlapping byte ranges possible, may cause overwrites
  - Last writer wins, as determined by primary
- **Problem:** what if multiple clients want to write to a file and no write must be lost?

# ATOMIC RECORD APPEND

- Append is common workload (at Google):
  - Multiple clients merge results in single file
  - Must not overwrite other's **records**, but specific order not important
  - ➡ *Use file as consumer-producer queue!*
- Primary + secondary chunkservers agree on common order of records
- Client library provides record abstraction

# APPEND ALGORITHM

(1) Application makes append request

(2) GFS client translates, sends it to master

(3) Master responds with *(chunk handle, primary+secondary replica locations)*

(4) Client pushes data to locations (pipelined)

(5) Primary check, if record fits into chunk

**Case (A):** It fits, primary does:

(6) Appends record to end of chunk

(7) Tells secondaries to do the same

(8) Receives responses from all secondaries

(9) Sends final response to client

**Case (B):** Does not fit, primary does:

(6) Pads chunk

(7) Tells secondaries to do the same

(8) Informs client about padding

(9) Client retries with next chunk

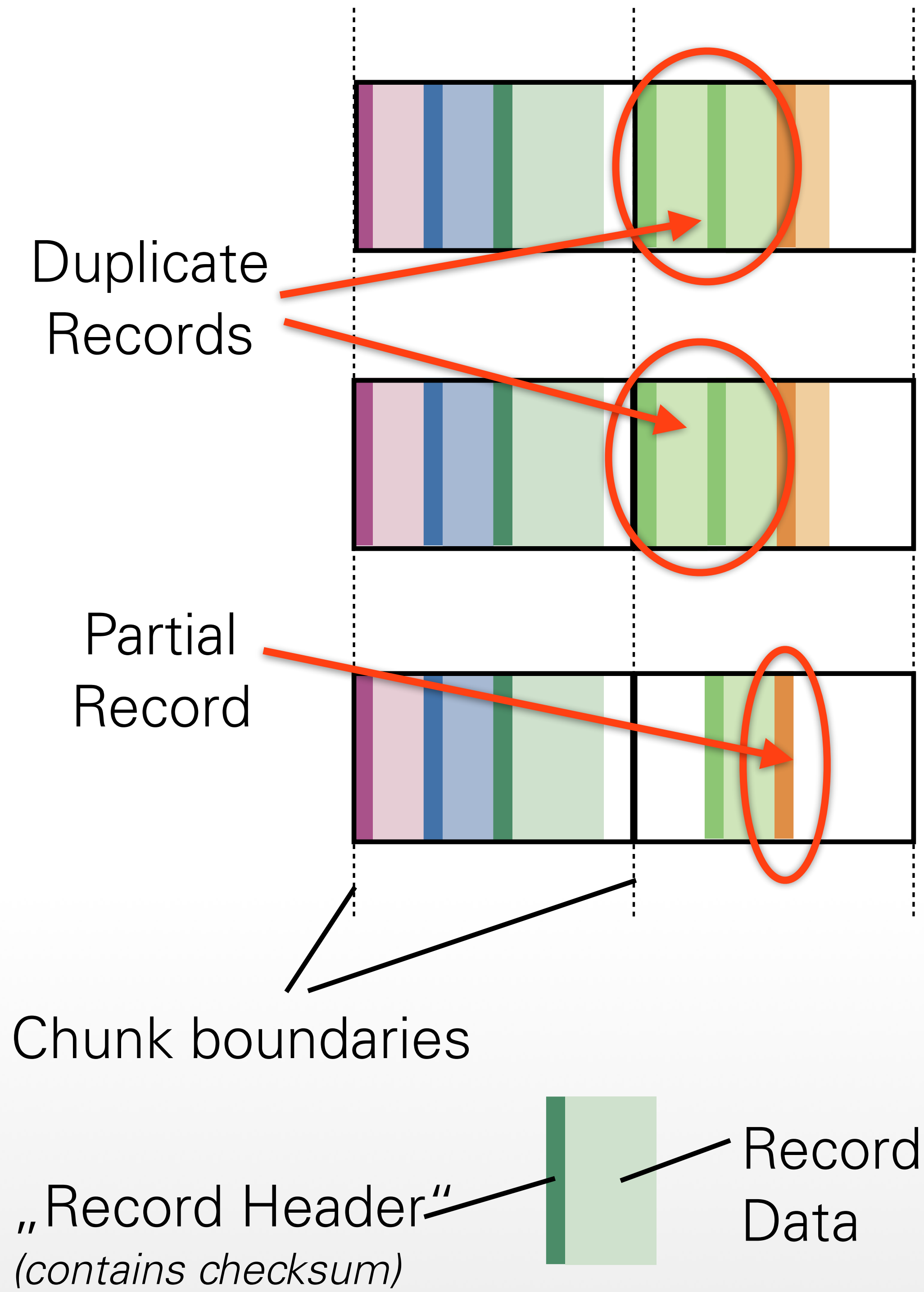
Source [6]

# HANDLING RECORDS

- **GFS guarantees:**
  - Records are appended atomically (not fragmented, not partially overwritten)
  - Each record is appended at least once
- **Failed append:** may lead to „*undefined regions*“ (partial records, no data)
- **Retries:** may lead to duplicate records in some chunks
- **Client:** handles broken/duplicate records

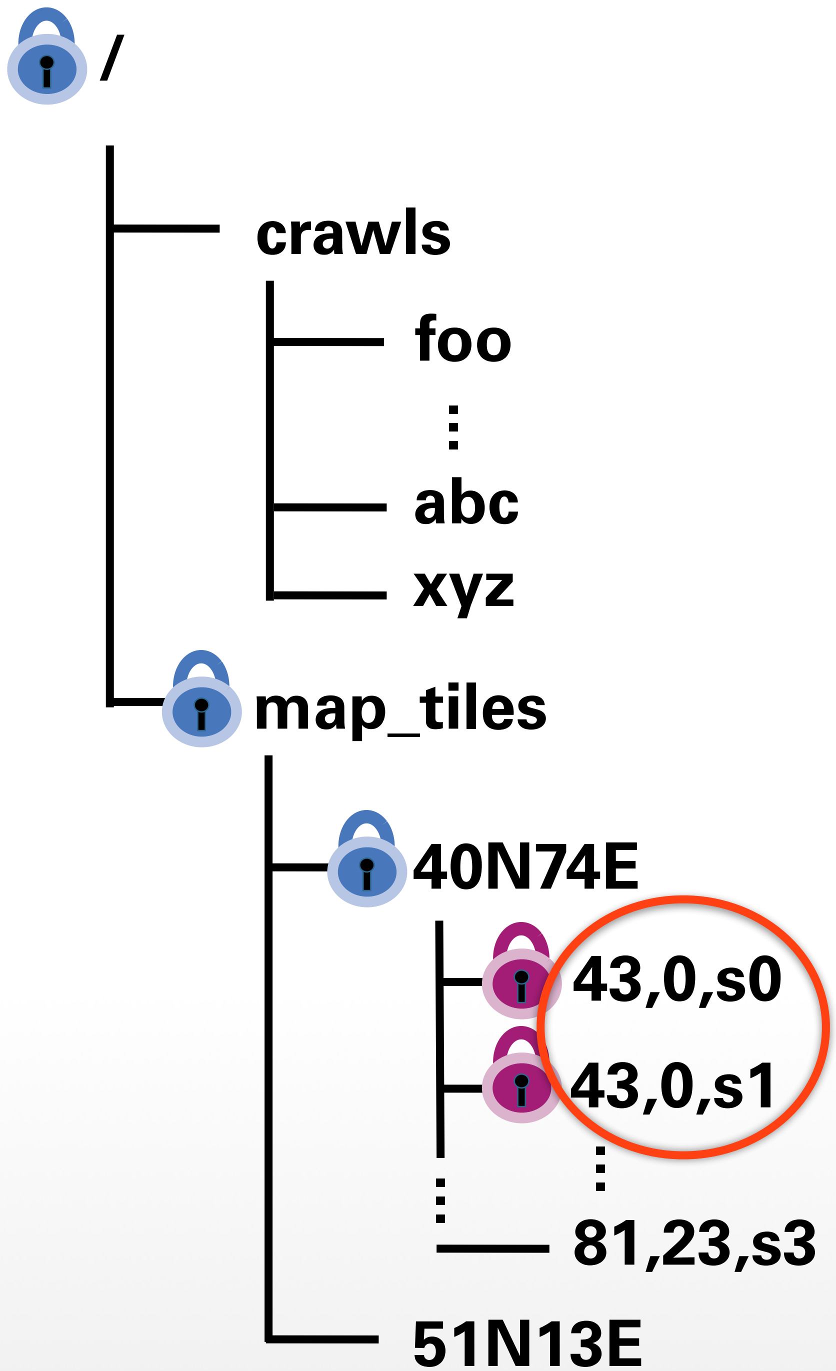
# EXAMPLE: RECORDS

- **Client library:**
  Generic support for per-record checksums
- **Application:**
  May add unique IDs to record to help detect duplicates




Note: the example offers a conceptual view, as the paper [5] does not have details on the real data layout for records

# FILE NAMESPACE




- Hierarchical namespace
- In Master’s memory
- Master is multi-threaded: concurrent access possible (read/writer lock)
- No „real“ directories:
  - (+) Read-lock parent dirs, write-lock file’s name
  - (-) No readdir()

# EFFICIENT SNAPSHOTS

- **Copy-on-write snapshots** are cheap:
  - Master revokes leases on chunks to be snapshotted to temporarily block writes
  - Master acquires write locks on all directories / files be snapshotted
  - Master creates new metadata structures pointing to original chunks
  - Upon write access to chunks, master delays client reply until chunkservers duplicated respective chunks

# DELETING FILES

- **Deleting a file:**
  - Renamed to hidden filename + timestamp
  - Can still be accessed under hidden name
  - Undelete possible via rename
  - Chunkservers not involved (yet)
- **Background scan of namespace:**
  - Find deleted file based on special filename
  - Erase metadata if timestamp is older than grace period

# GARBAGE COLLECTION

- Garbage collection is background activity
- **Master:**
  - Scans chunk namespace regularly
  - Chunks not linked from any file are obsolete
- **Chunkservers:**
  - Send heartbeat messages to master
  - Receive list of obsolete chunks in reply
  - Delete obsolete chunks when convenient

# OUTLINE

- Classical distributed file systems
  - NFS: Sun Network File System
  - AFS: Andrew File System
- Parallel distributed file systems
- Case study: The Google File System
  - Scalability
  - Fault tolerance
- Other approaches

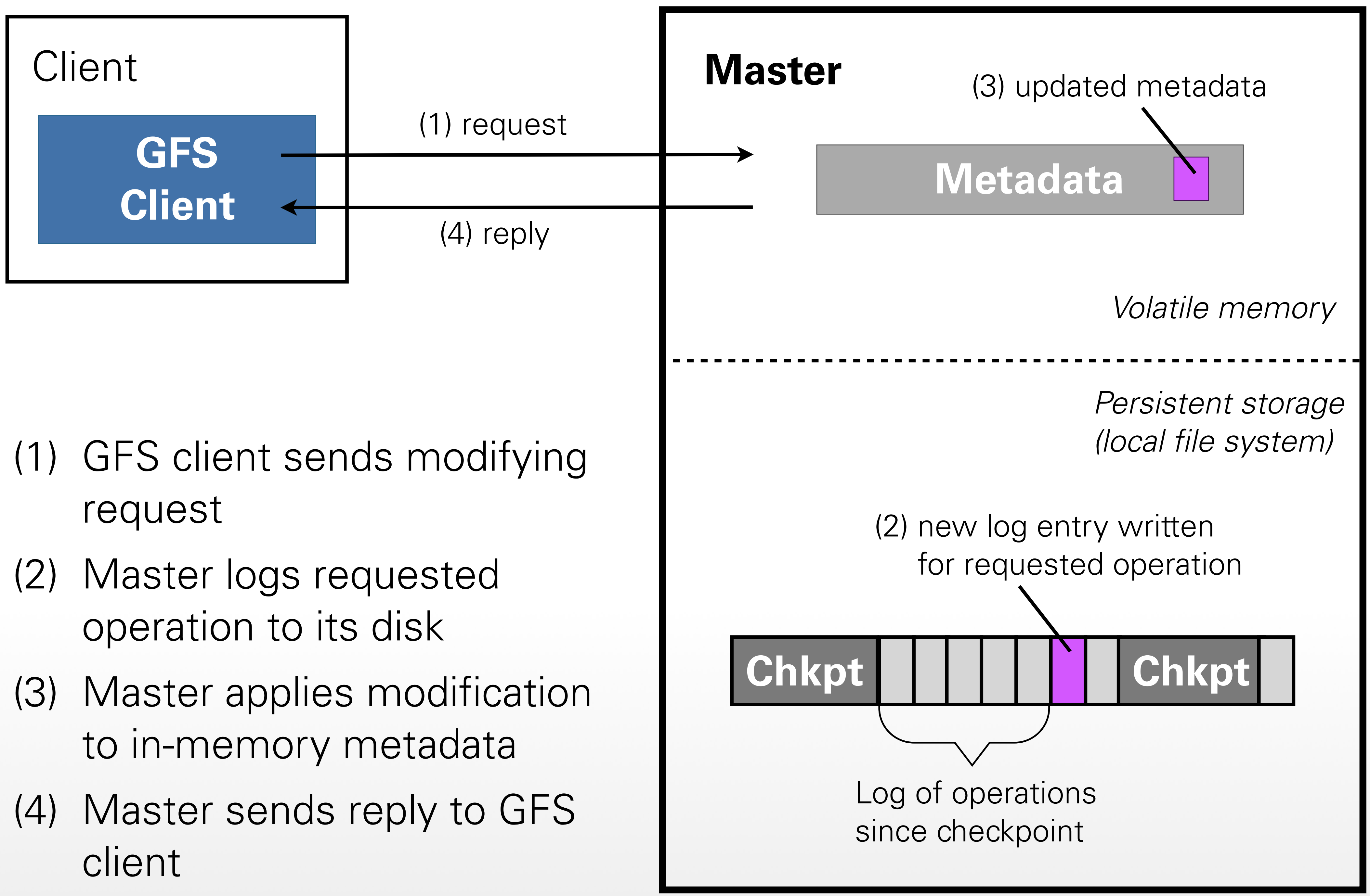
# MASTER & METADATA

- **Master** is process on separate machine
- Manages all **metadata:**
  - File namespace
  - File-to-chunk mappings
  - Chunk location information
  - Access control information
  - Chunk version information
- Does <u>not</u> store/read/write any file data!

Cached in RAM

Persistently stored on disk

# LOG + CHECKPOINTS




(1) GFS client sends modifying request
(2) Master logs requested operation to its disk
(3) Master applies modification to in-memory metadata
(4) Master sends reply to GFS client

# MASTER FAILURES

- **Fast restart** from checkpoint+log, if Master process dies, but ...
- ... the Master's *machine* might still fail!
- **Master replication:**
  - Log+checkpoints **replicated** on multiple machines
  - Changes considered committed after being logged both *locally* and *remotely*
  - Clients are sent reply only after full commit

# SHADOW MASTERS

- Only one (real) master is in charge, performs background jobs (e.g., garbage collection)
- For better read availability: **Shadow Masters**
  - Read replicated logs, apply observed changes to their own in-memory metadata
  - Receive heartbeat messages from all chunkservers, like real master
  - Can serve read-only requests, if real master is down

# GFS: KEY TECHNIQUES

- **Scalability:** metadata + data separated
  - Large chunk size, less coordination overhead
  - Simple, in-memory metadata (namespace, ...)
- **Fault tolerant:**
  - Replication: Master + chunks
  - More in paper [5]: checksums for chunks, chunk replica recovery, ...
- **Non-POSIX:** applications use primitives that suit their workload (e.g., record append)

# OUTLINE

- Classical distributed file systems
  - NFS: Sun Network File System
  - AFS: Andrew File System
- Parallel distributed file systems
- Case study: The Google File System
  - Scalability
  - Fault tolerance
- **Other approaches**

# OTHER APPROACHES

- **Distributed metadata servers:**
  - Replicated state machine handles metadata
  - TidyFS, GPFS, …
- **Distributed key–value stores:**
  - Data stored as binary objects (blobs)
  - Read / write access via get() / put()
  - Multiple nodes store replicas of blobs
  - Consistent hashing determines location

# REFERENCES


**Classical distributed file systems**

[1] Text book: „Distributed Systems - Concepts and Design", Couloris, Dollimore, Kindberg

[2] Basic lecture on distributed file systems from „Operating Systems and Security" (in German)

**Large-scale distributed file systems and applications**

[3] Data processing at Worldwide LHC Computing Grid: http://lcg.web.cern.ch/LCG/public/data-processing.htm

[4] *„Finding a Needle in Haystack: Facebook's Photo Storage"*, Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel, OSDI'10, Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010

[5] *„The Google File System"*, Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, SOSP'03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2003

[6] *„The Google File System"*, Slides presented at SOSP '03, copy provided by the authors mirrored on DOS lecture website: http://os.inf.tu-dresden.de/Studium/DOS/SS2012/GFS-SOSP03.pdf

[7] *„Dynamo: Amazon's Highly Available Key-value Store"*, Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, SOSP'09 Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009