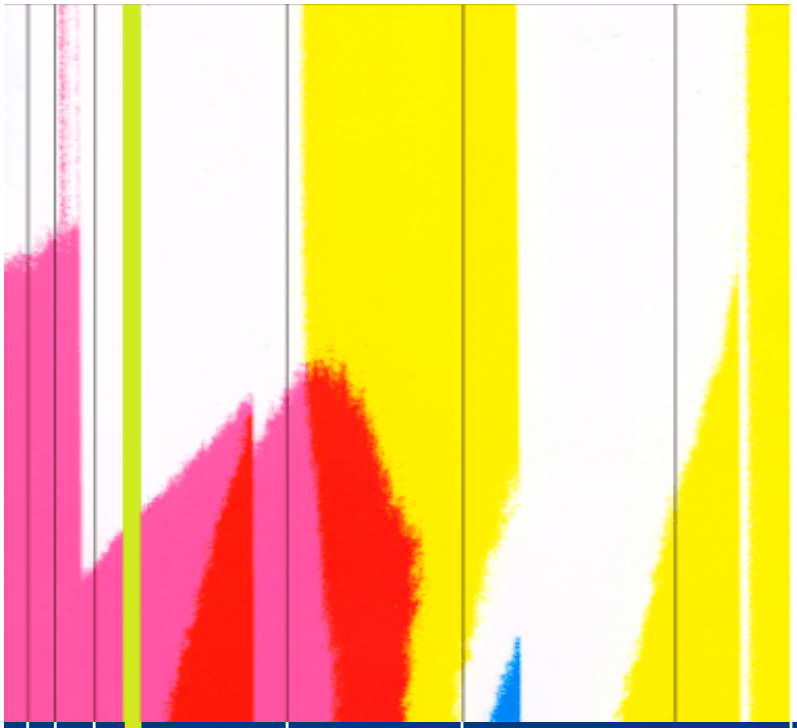


Data Management (2)

Transparent Data Management.
Case studies: NFS, Gfarm, GFS

Gabriel Antoniu

INRIA - Centre de Recherche de Rennes Bretagne Atlantique

February 2014

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Agenda

Introduction

- Grid data management: motivation and goals

Approaches to data management in grids

- Explicit grid data management: GridFTP, IBP, SRB
- Transparent grid data management
- Grid file systems. NFS, Gfarm, GFS

Convergence of Grid and P2P systems

- Basics of P2P file systems. CFS, Ivy
- Generic P2P services: JXTA
- RAM-based P2P-oriented grid data sharing: JuxMem

Data management in clouds

- Google MapReduce, Hadoop

Acknowledgements

Frédéric Desprez & Co

Adrien Lèbre

Paul Krzyzanowski

Osamu Tatebe

Sanjay Ghemawat

Howard Gobioff

Shun-Tak Leung

Accessing files

FTP, telnet:

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

We want more transparency

- Allow user to access remote resources just as local ones

Focus: file systems

File service types

Upload/Download model

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

Advantage

- **Simple**

Problems

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

File service types

Remote access model

File service provides functional interface:

- *create, delete, read bytes, write bytes, etc…*

Advantages:

- 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

File server

File Directory Service

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

File service

- Provides file access interface to clients

Client module (driver)

- Client side interface for file and directory service
- if done right, helps provide access transparency

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 problems if no cache
  - Obsolete data
- We can *write-through*
  - Must notify clients holding copies
  - Requires extra state, generates extra traffic

Session semantics

Relax the rules

Changes to an open file are initially visible only to the process (or machine) that modified it.

Last process to modify the file wins.

Other solutions

Make files **immutable**

- Aids in replication
- Does not help with detecting modification

Or...

Use **atomic transactions**

- Each file access is an atomic transaction
- If multiple transactions start concurrently
  - Resulting modification is serial

File usage patterns

We can’t have the best of all worlds

Where to compromise?

- Semantics vs. efficiency
- Efficiency = client performance, network traffic, server load

Need to understand how files are used

File usage

Most files are <10 Kbytes

- 2005: average size of 385,341 files on my Mac =197 KB
- 2007: average size of 440,519 files on my Mac =451 KB
- (files accessed within 30 days:
  147,398 files. average size=56.95 KB)
- Feasible to transfer entire files (simpler)
- Still have to support long files

Most files have short lifetimes

- Perhaps keep them local

Few files are shared

- Overstated problem
- Session semantics will cause no problem most of the time

System design issues

Where do you find the remote files?

Should all machines have the exact same view of the directory hierarchy?

e.g., global root directory?

```
//server/path
```

or forced “remote directories”:

```
/remote/server/path
```

or….

Should each machine have its own hierarchy with remote resources located as needed?

```
/usr/local/games
```

Naming and Transparency

Naming – mapping between logical and physical objects

Multilevel mapping – abstraction of a file that hides the details of how and where on the disk the file is actually stored

A **transparent** DFS hides the location where in the network the file is stored

For a file being replicated in several sites, the mapping returns a set of the locations of this file's replicas; both the existence of multiple copies and their location are hidden

Naming Structures

Location transparency – file name does not reveal the file's physical storage location

Location independence – file name does not need to be changed when the file's physical storage location changes

Naming Schemes — Three Main Approaches

Files named by combination of their host name and local name

- Guarantees a unique systemwide name

Attach remote directories to local directories

- Gives the appearance of a coherent directory tree
- Only previously mounted remote directories can be accessed transparently

Total integration of the component file systems

- A single global name structure spans all the files in the system
- If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable

Stateful or stateless design?

Stateful

- Server maintains client-specific state
- Shorter requests
- Better performance in processing requests
- Cache coherence is possiblen know who's accessing what
- File locking is possible

Stateful or stateless design?

Stateless

- Server maintains *no* information on client accesses
- Each request must identify file and offsets
- Server can crash and recover
- Client can crash and recover
- No open/close needed, they only establish state
- No server space used for state
- Don’t worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible

Caching

Hide latency to improve performance for repeated accesses

Four places

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

WARNING:
cache consistency
problems

Approaches to caching

Write-through

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

Delayed writes (write-behind)

- Data can be buffered locally (watch out for consistency – others won't see updates!)
- Remote files updated periodically
- One bulk wire 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.

Cache Location – Disk vs. Main Memory

Advantages of disk caches

- More reliable
- Cached data kept on disk are still there during recovery and don't need to be fetched again

Advantages of main-memory caches:

- Permit workstations to be diskless
- Data can be accessed more quickly
- Performance speedup in bigger memories
- Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users

Distributed File Systems
Case Studies

The “ancestor”: NFS Network File System

Sun Microsystems

c. 1985

NFS Design Goals

- Any machine can be a client or server
- Must support diskless workstations
- Heterogeneous systems must be supported
  - Different HW, OS, underlying file system
- Access transparency
  - Remote files accessed as local files through normal file system calls
- Recovery from failure
  - Stateless, UDP, client retries
- High Performance
  - Use caching and read-ahead

NFS Design Goals

No migration transparency

- If resource moves to another server, client must remount resource

No support for UNIX file access semantics

- Stateless design: file locking is a problem
- All UNIX file system controls may not be available

NFS Design Goals

Devices

Must support diskless workstations where *every* file is remote.

Remote devices refer back to local devices.

NFS Design Goals

Transport Protocol

Initially NFS ran over **UDP** using Sun RPC

Why UDP?

- Slightly faster than TCP
- No connection to maintain (or lose)
- NFS is designed for Ethernet LAN environment – relatively reliable
- Error detection but no correction.

NFS retries requests

NFS Protocols

Mounting protocol

Request access to exported directory tree

Directory & File access protocol

Access files and directories
(read, write, mkdir, readdir, …)

Problems with NFS

File consistency

Assumes clocks are synchronized

Open with append cannot be guaranteed to work

Locking cannot work

- Separate lock manager necessary (stateful)

Global UID space assumed

Problems with NFS

No reference counting of open files

- You can delete a file you (or others) have open!

Common practice

- Create temp file, delete it, continue access
- Sun's hack:
  - If same process with open file tries to delete it
  - Move to temp name
  - Delete on close

Problems with NFS

File permissions may change

- Invalidating access to file

No encryption

- Requests via unencrypted RPC
- Authentication methods available
  - Diffie-Hellman, Kerberos, Unix-style
- Rely on user-level software to encrypt

Improving NFS: version 2

User-level lock manager

- Monitored locks
  - Status monitor: monitors clients with locks
  - Informs lock manager if host inaccessible
  - If server crashes: status monitor reinstates locks on recovery
  - If client crashes: all locks from client are freed

NV RAM support

- Improves write performance
- Normally NFS must write to disk on server before responding to client *write* requests
- Relax this rule through the use of non-volatile RAM

Improving NFS: version 2

Adjust RPC retries dynamically

- Reduce network congestion from excess RPC retransmissions under load
- Based on performance

Client-side disk caching

- cacheFS
- Extend buffer cache to disk for NFS
  - Cache in memory first
  - Cache on disk in 64KB chunks

More improvements… NFS v3

Updated version of NFS protocol

Support 64-bit file sizes

TCP support and large-block transfers

- UDP caused more problems on WANs (errors)
- All traffic can be multiplexed on one connection
  - Minimizes connection setup
- No fixed limit on amount of data that can be transferred between client and server

Negotiate for optimal transfer size

Server checks access for entire path from client

More improvements… NFS v3

New *commit* operation

- Check with server after a *write* operation to see if data is committed
- If *commit* fails, client must **resend** data
- Reduce number of *write* requests to server
- Speeds up *write* requests
  - Don’t require server to write to disk immediately

Return file attributes with each request

- Saves extra RPCs

NFS version 4 enhancements

Compound RPC

- Group operations together
- Receive set of responses
- Reduce round-trip latency

Stateful server

Stateful open/close operations

- Supports exclusive creates
- Client can cache aggressively

NFS version 4 enhancements

create, link, open, remove, rename

- Inform client if the directory changed during the operation

Strong security

- Extensible authentication architecture

File system replication and migration

- To be defined

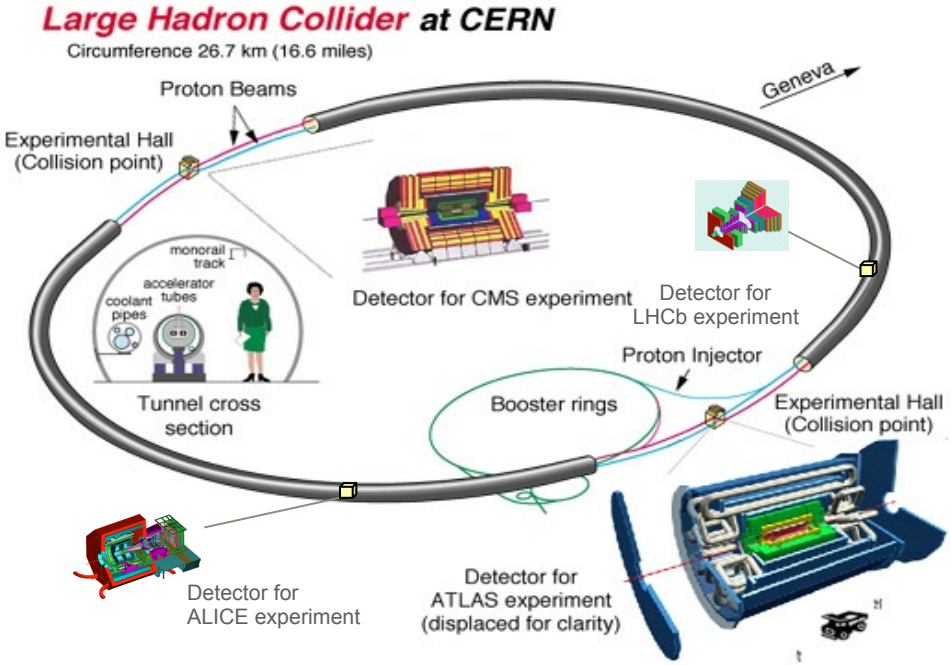
No concurrent write sharing or distributed cache coherence

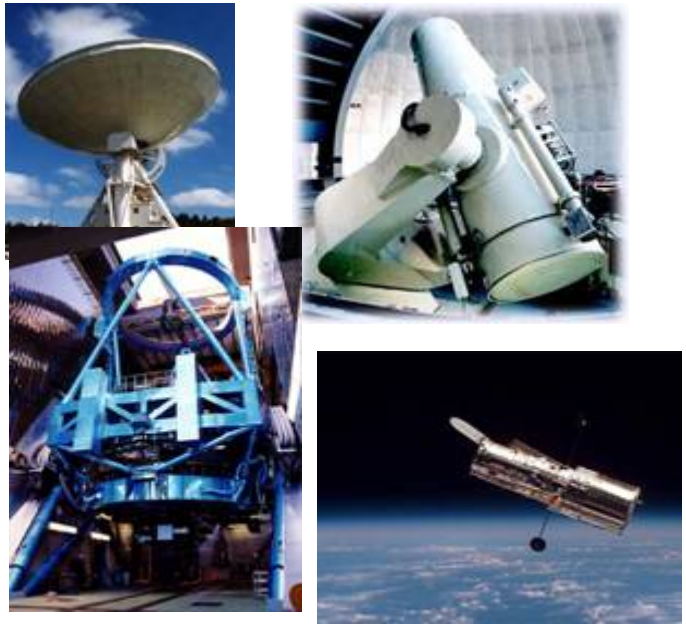
Grid file systems: towards grid-scale NFS
Case study: Gfarm (University of Tsukuba)

Target Applications: Petascale Data Intensive Computing

High Energy Physics

- CERN LHC, KEK-B Belle
  - ~MB/collision,
    100 collisions/sec
  - ~PB/year
  - 2000 physicists, 35 countries





Astronomical Data Analysis

- Sweep analysis of the whole data
- TB~PB/year/telescope

Petascale Data Intensive Computing Requirements

Storage Capacity
- Peta/Exabyte scale files, millions of millions of files

Computing Power
- > 1TFLOPS, hopefully > 10TFLOPS

I/O Bandwidth
- > 100GB/s, hopefully > 1TB/s within a system and between systems

Global Sharing
- group-oriented authentication and access control

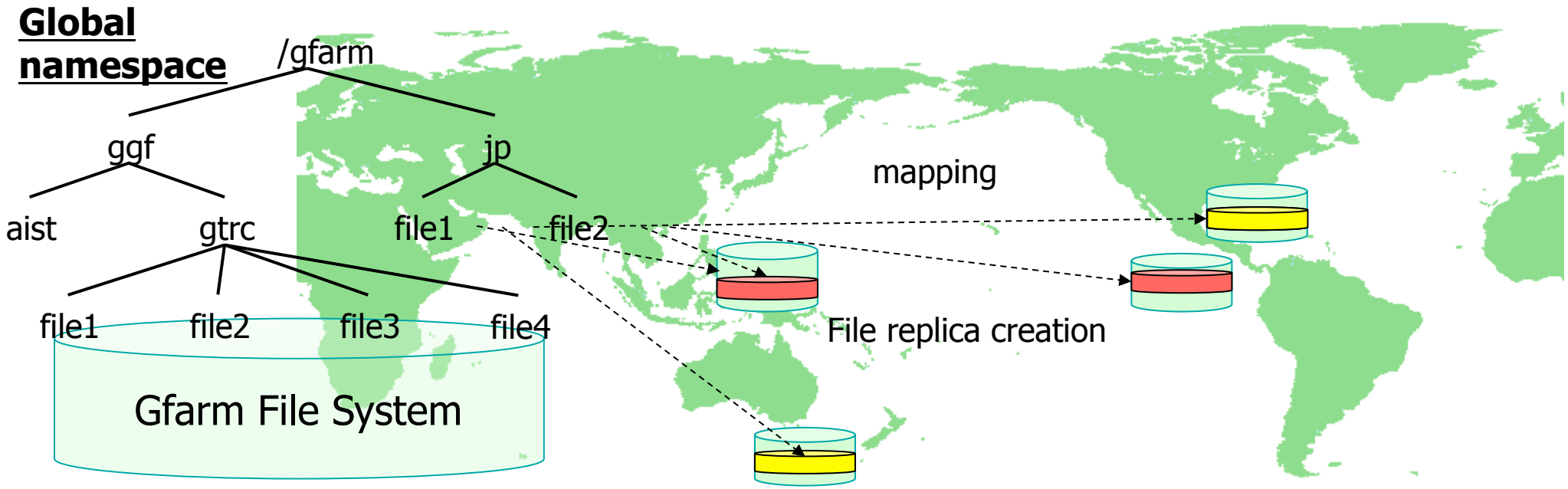
Gfarm Grid File System [CCGrid 2002]

Open Source wide area distributed file system

Global namespace to federate storages

It provides scalable I/O performance exploiting access locality

It supports fault tolerance and avoids access concentration by automatic file replica selection

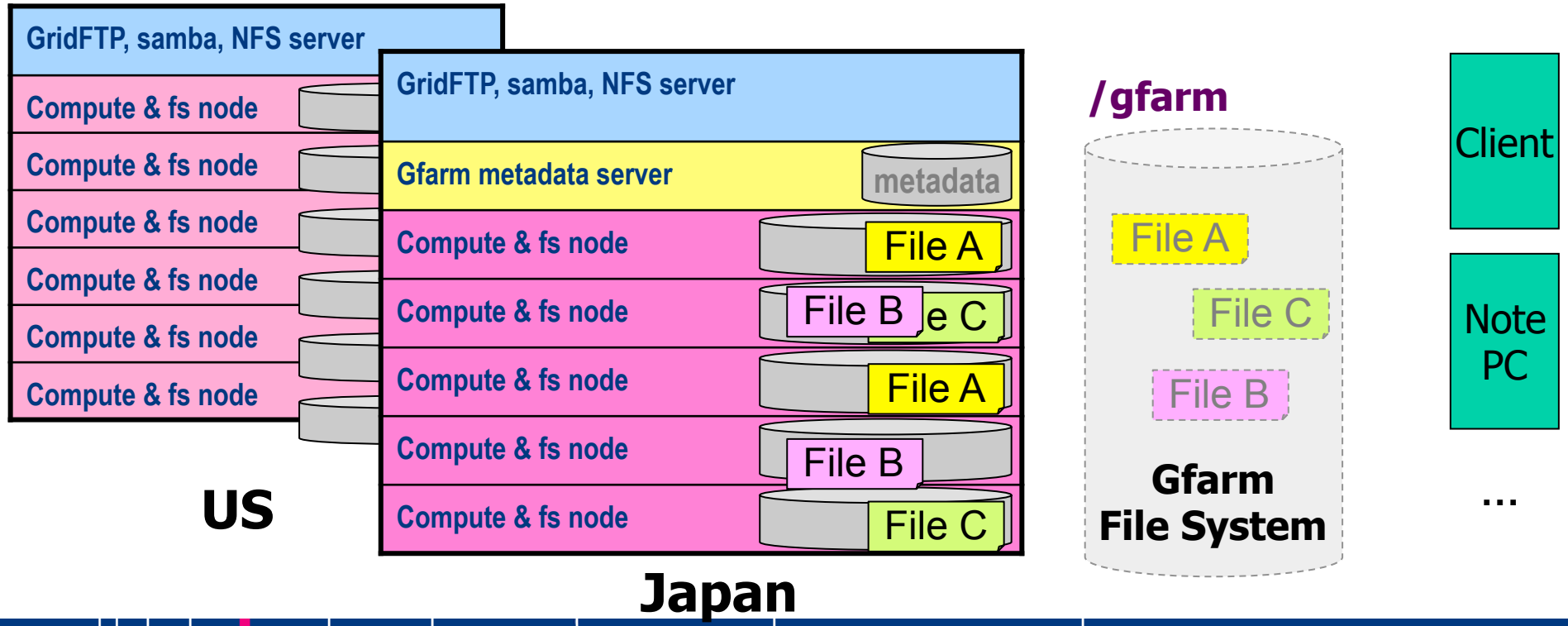


Gfarm Grid File System (2)

Physically, files may be replicated and stored in any file system node

Files can be shared transparently regardless of the location

File system nodes can be distributed



Scalable I/O Performance

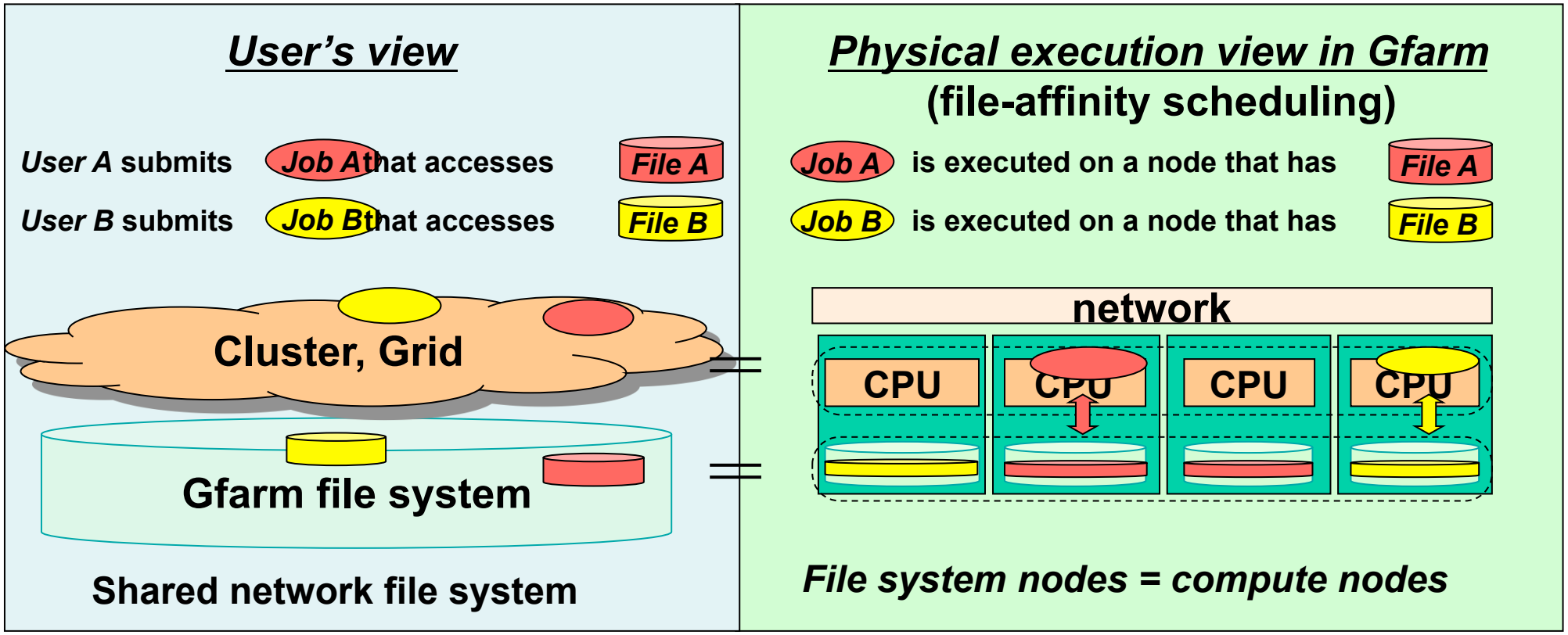
Decentralization of disk access putting priority to local disk

- When a new file is created,
  - Local disk is selected when there is enough space
  - Otherwise, near and the least busy node is selected
- When a file is accessed,
  - Local disk is selected if it has one of the file replicas
  - Otherwise, near and the least busy node having one of file replicas is selected

File affinity scheduling

- Schedule a process on a node having the specified file
  - Improve the opportunity to access local disk

Scalable I/O performance in distributed environment



Do not separate storage and CPU (SAN not necessary)

Move and execute program instead of moving large-scale data

exploiting local I/O is a key for scalable I/O performance

Particle Physics Data Analysis

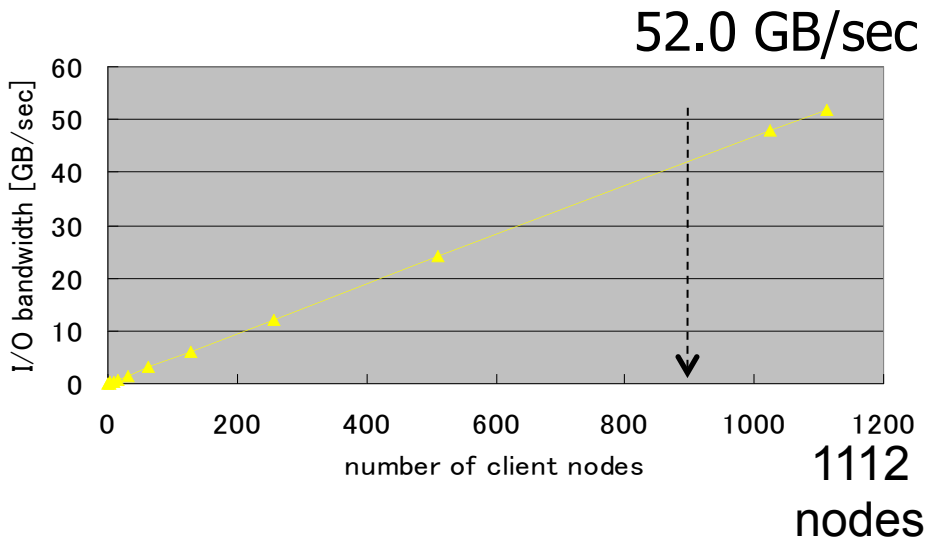
S. Nishida, N. Katayama, I. Adachi, O. Tatebe, M. Sato, T. Boku, A. Ukawa, "High Performance Data Analysis for Particle Physics using the Gfarm file system", Journal of Physics: Conference Series, 119, 062039, 2008

- Construct 26 TB of Gfarm FS using **1112** nodes
- Store all 24.6 TB of Belle experiment data
- 52.0 GB/s in parallel read

→ **3,024** times speedup

- 24.0 GB/s in skimming process for b → s γ decays using 704 nodes

→ **3 weeks to 30 minutes**



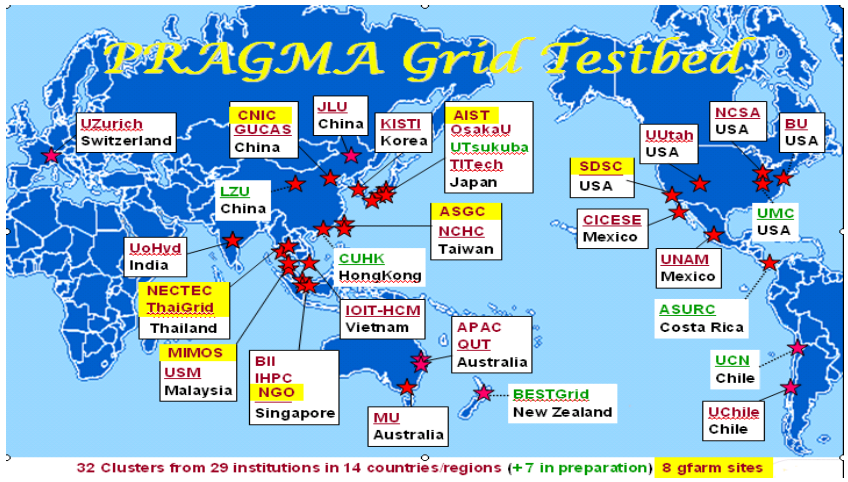
PRAGMA Grid

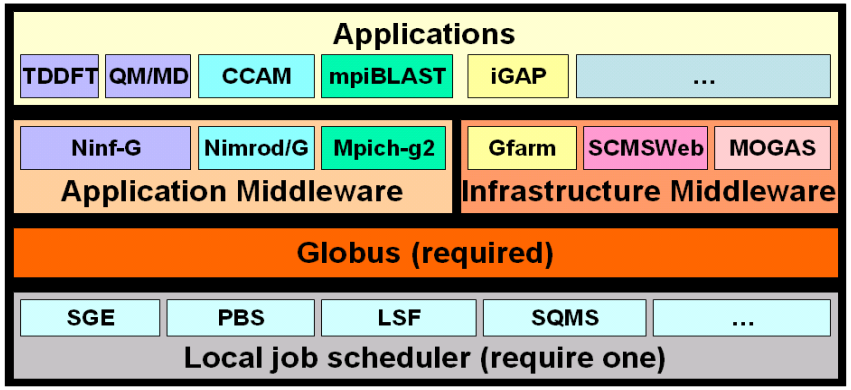
C. Zheng, O. Tatebe et al, "Lessons Learned Through Driving Science Applications in the PRAGMA Grid", Int. J. Web and Grid Services, Inderscience Enterprise Ltd., 2007

- Worldwide Grid testbed consisting of 14 countries, 29 institutes
- Gfarm file system is used for file sharing infrastructure

→ executable, input/output data sharing possible in Grid

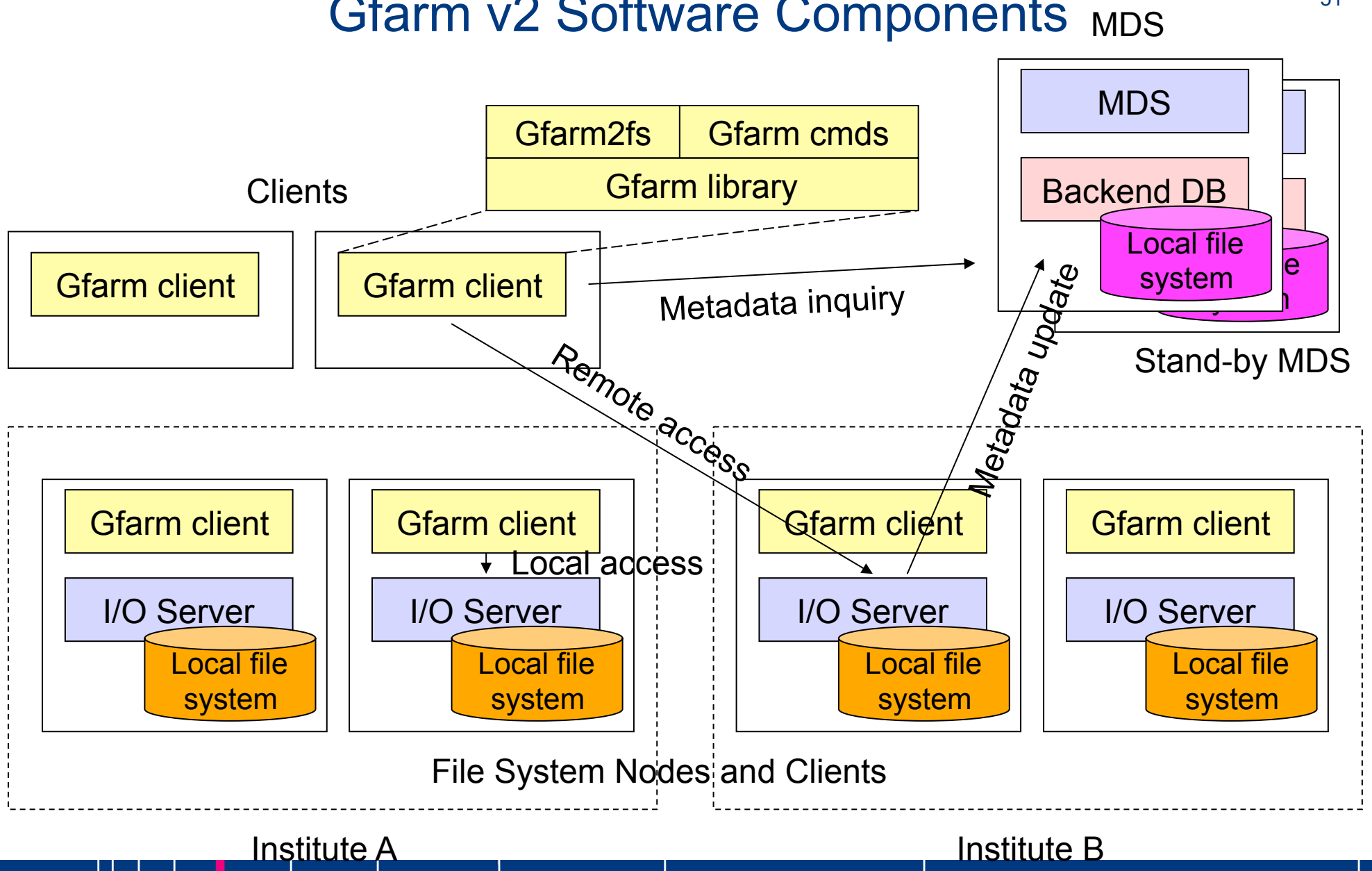
→ no explicit staging to a local cluster needed





Design and implementation of Gfarm v2

Gfarm v2 Software Components



Implementation features of Gfarm v2

File replica management
- Any number of file replicas, anywhere
- Close-to-open consistency semantics by a central metadata server

Keep consistency between metadata and the corresponding physical file
- Keep removed file replica information when the file system node is down in order to remove later
- Consistent metadata update even when an unexpected crash
- Prohibit invalid physical file access

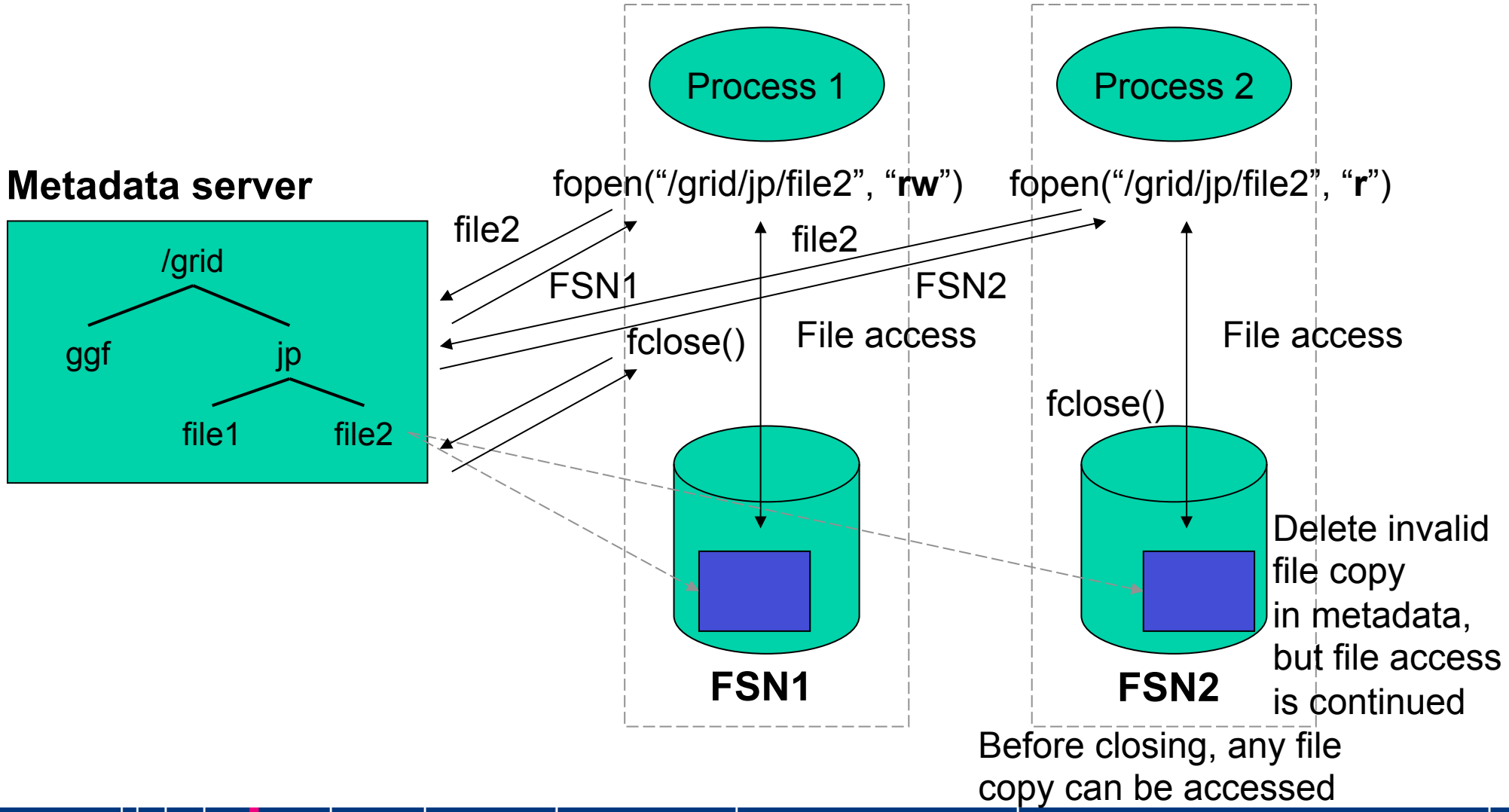
Improve metadata server performance
- Reduce # of RPCs by extending NFSv4 compound RPC
  - Reduce operation latency from a distant location
- Cache all metadata in memory and improve response time
  - No disk wait when responding
  - A writing thread to a backend database

Reduce scheduling cost of file system nodes
- Monitor file system node status by metadata server periodically

Manage global user names and group names
- Introduce a privileged user for file system management

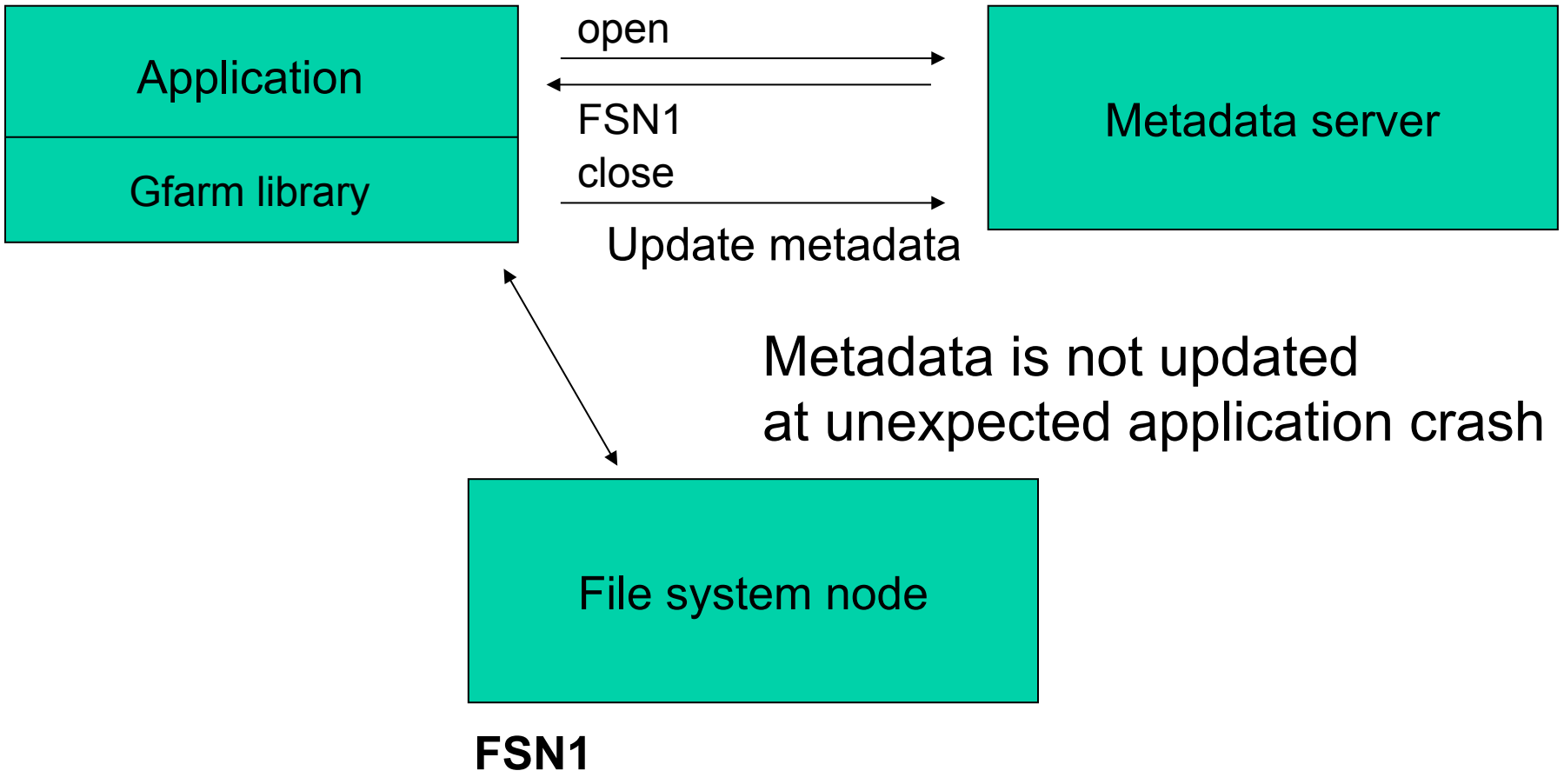
Close-to-open consistency

Remove obsolete file replicas when closing



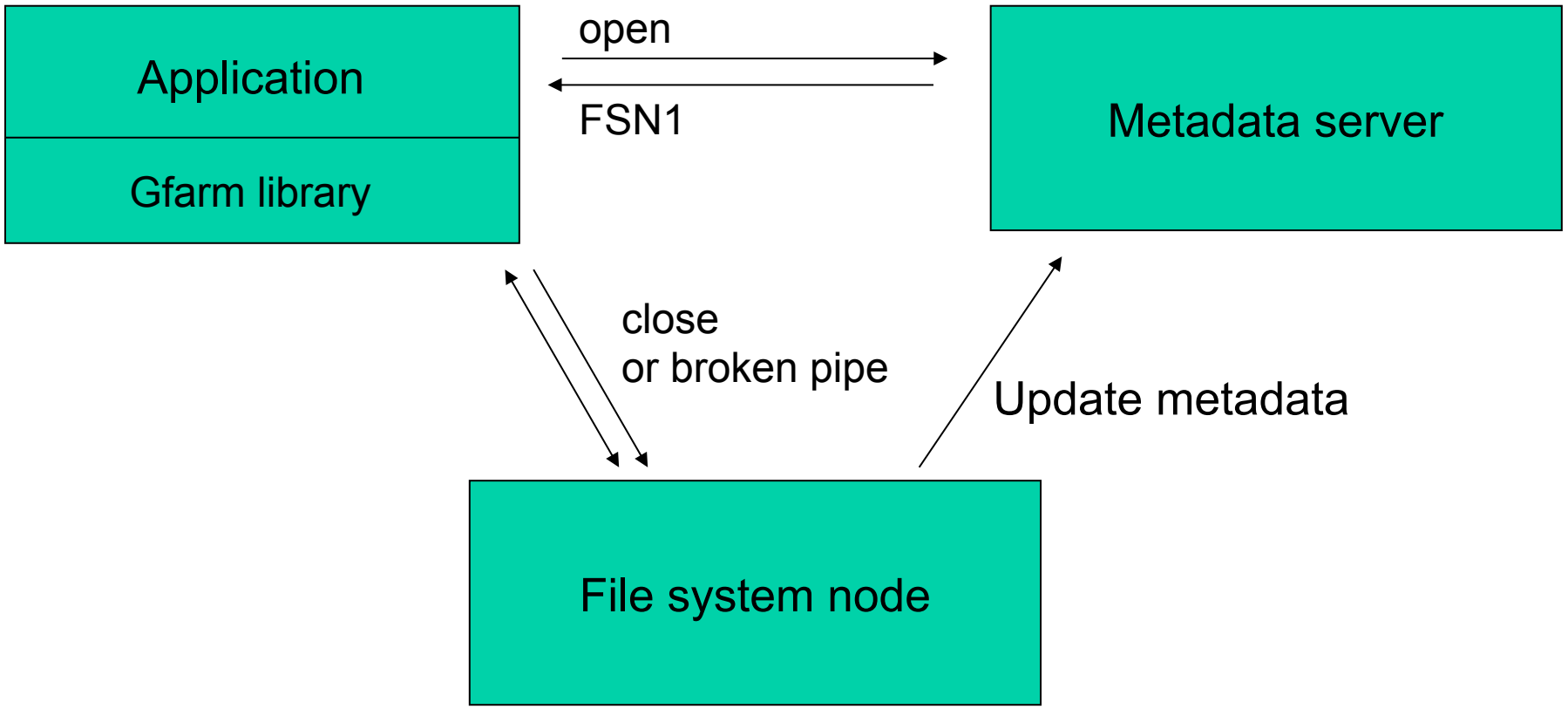
Consistent update of metadata (1)

Gfarm v1 – Gfarm library updates metadata



Consistent update of metadata (2)

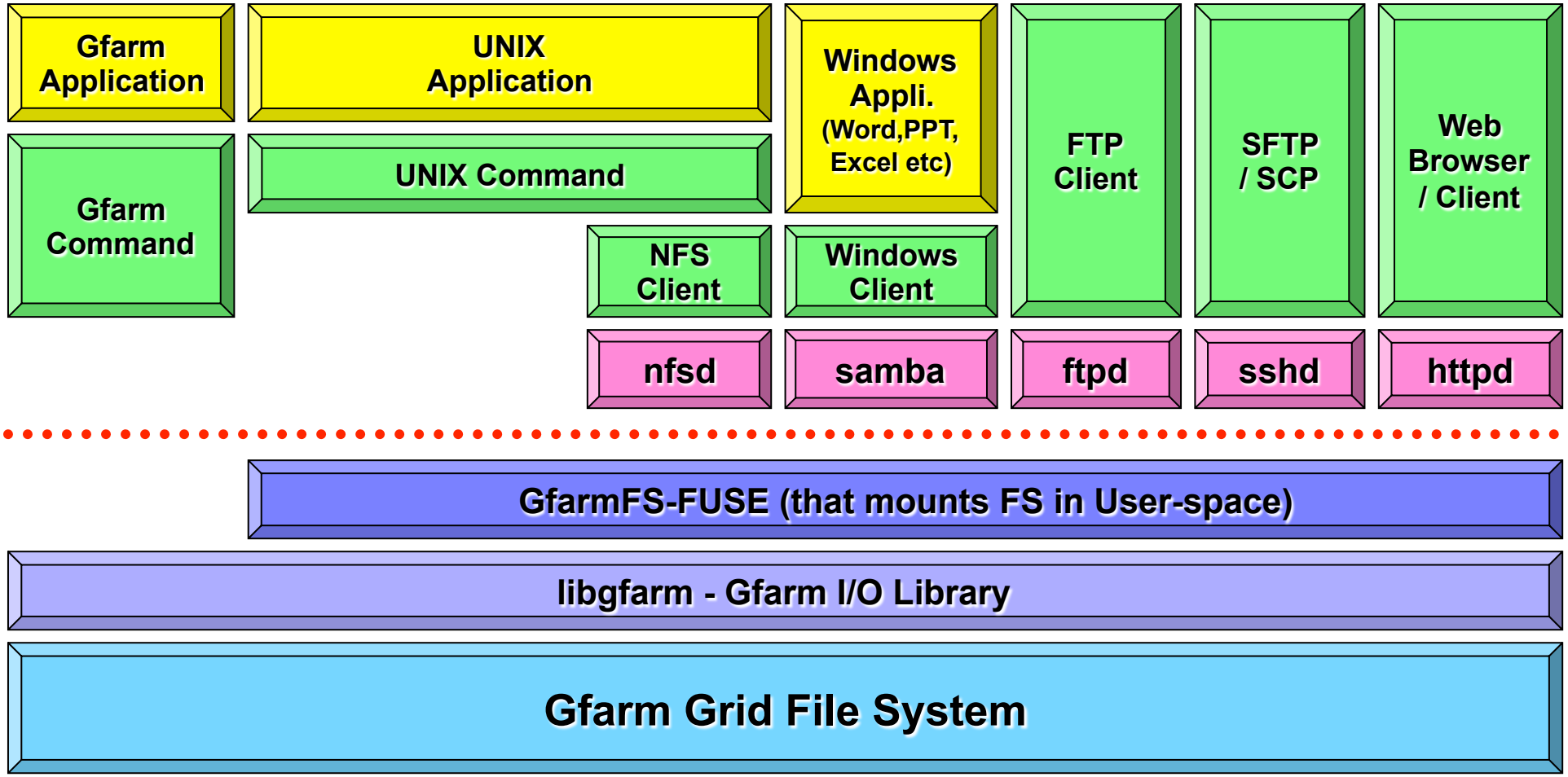
Gfarm v2 – file system node updates metadata



FSN1

Metadata is updated by file system node
even at unexpected application crash

Software Stack



Open Source Development

Sourceforge.net

- http://sourceforge.net/projects/gfarm

Latest Source code

- svn co **http://gfarm.svn.sourceforge.net/svnroot/gfarm/gfarm_v2/trunk** gfarm_v2
- svn co **http://gfarm.svn.sourceforge.net/svnroot/gfarm/gfarm2fs/trunk** gfarm2fs

Performance Evaluation

Environment

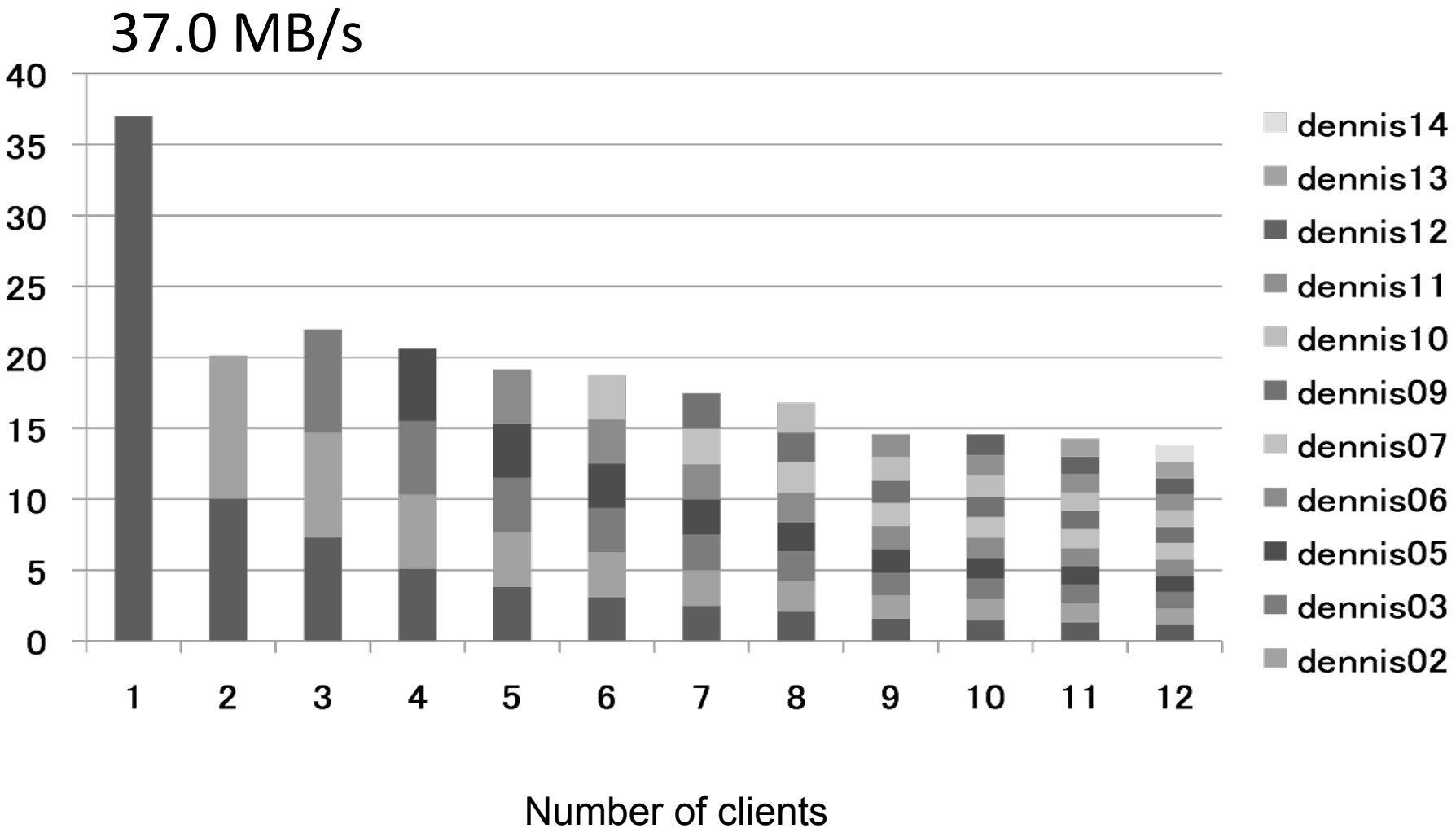
- Metadata server – Univ of Tsukuba
- File System Nodes

	Tsukuba	AIST	SDSC
#nodes	14	8	3
RTT [msec]	0.202	0.787	119

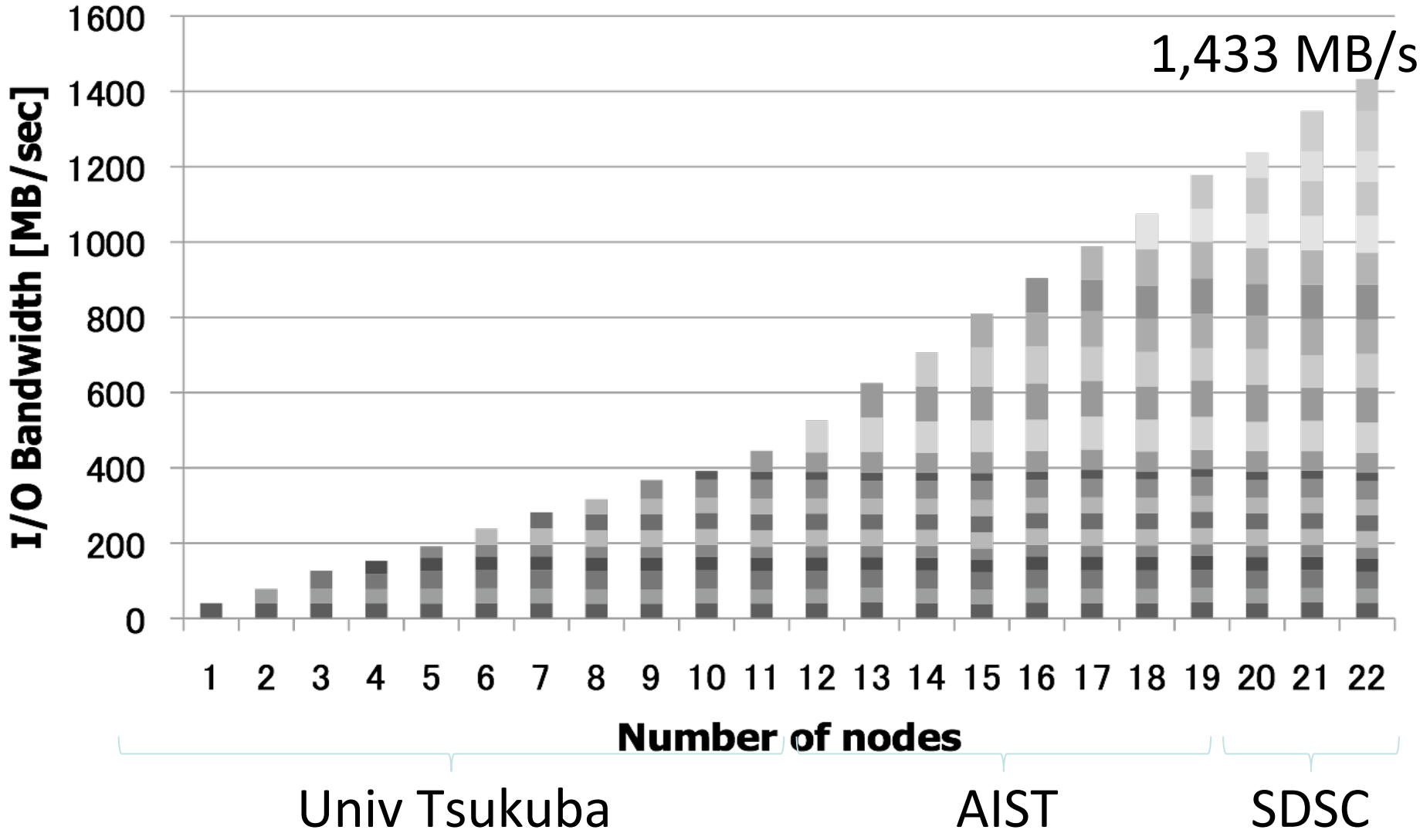
Evaluation items

- Parallel I/O Bandwidth
  - Each client reads a different 1GB of file
- Response time of file operations

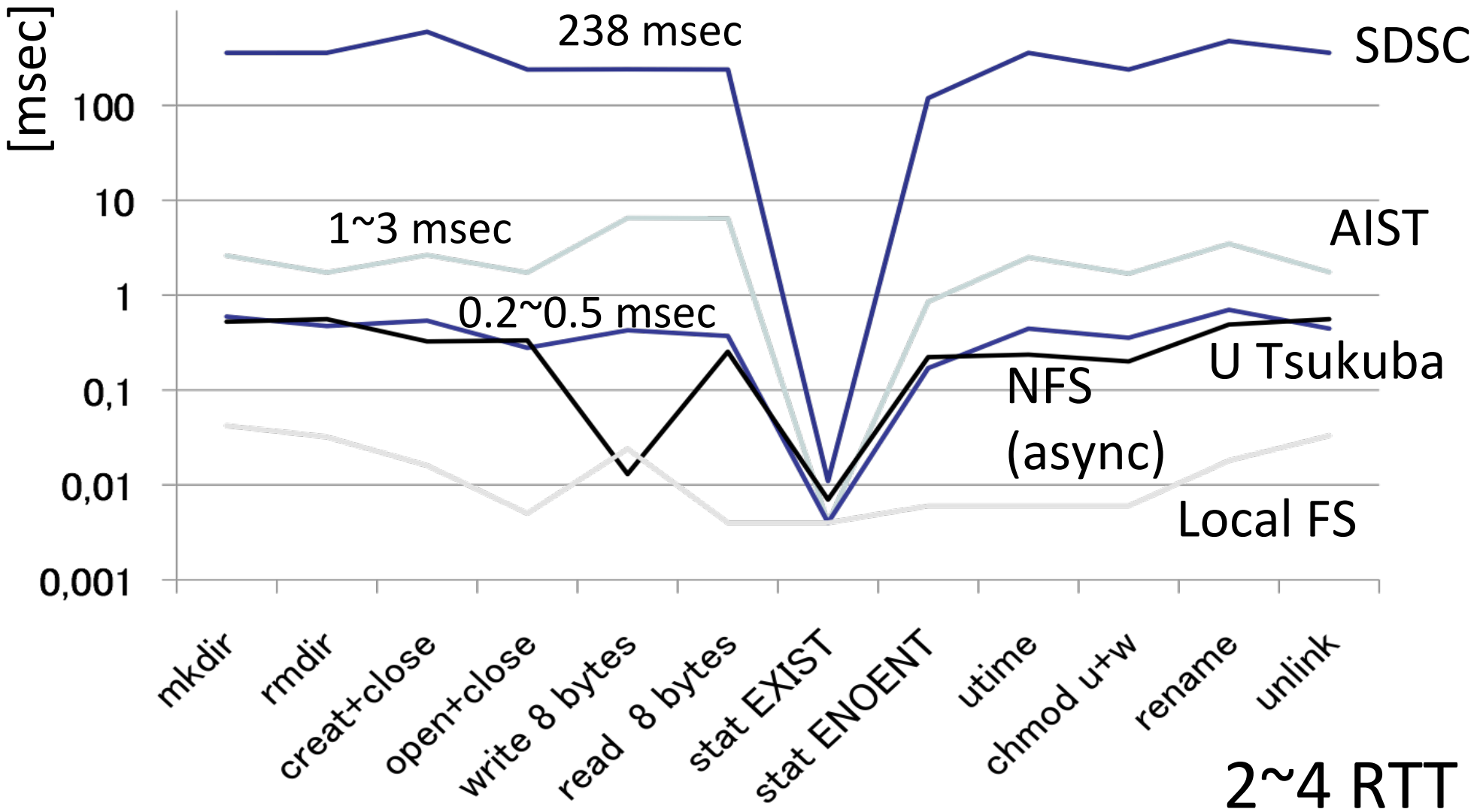
NFS I/O bandwidth (read 1G sep. data)



Gfarm v2 scalable I/O bandwidth

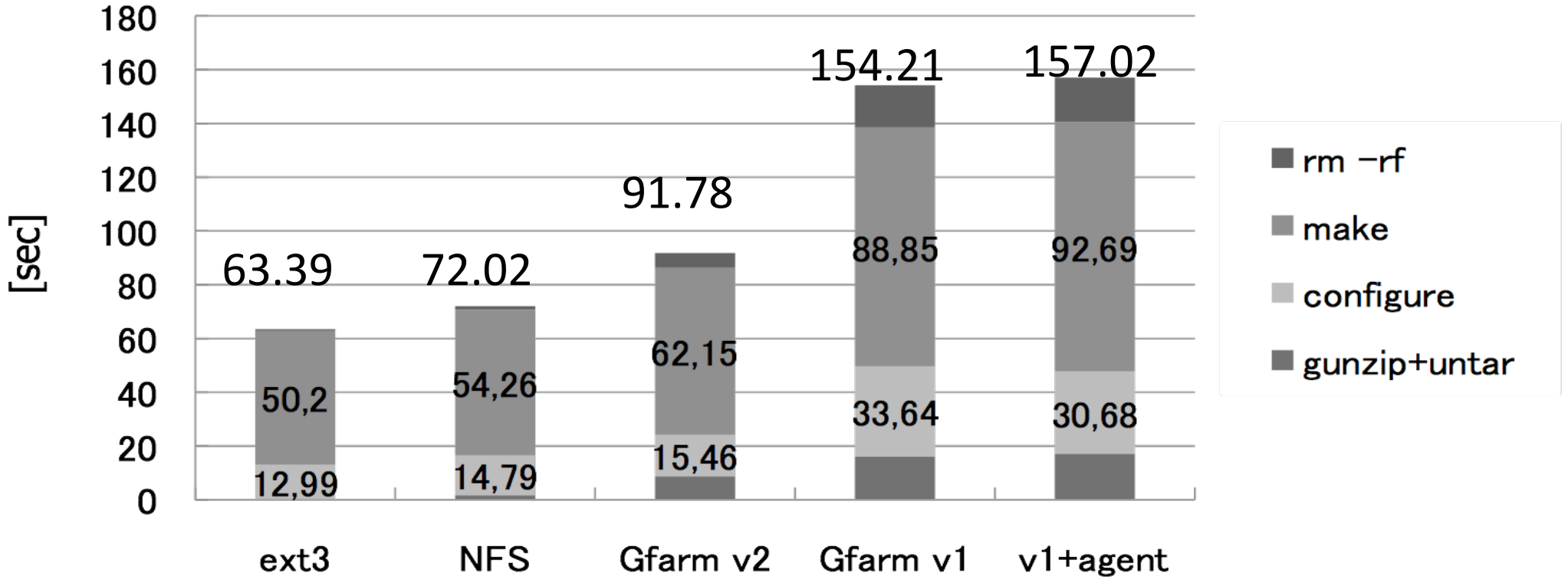


Operation latency



Gunzip+untar and build time of Gfarm v2

#files 1,424
Total file size 1.37 MByte



Summary

Gfarm v2 Grid file system

- Designed to improve consistency, security, performance
- Scalable I/O performance in geographically distributed environment
  - 1,433 MB/sec when accessing from 22 clients in US and Japan
- Operation latency is 2x to 4x RTT to metadata server
  - Almost same performance as NFS (async) in LAN

Open source development

- http://sf.net/projects/gfarm

The Google File System♪

(Presented at SOSP 2003)♪

Introduction♪

Google – search engine.

Applications process lots of data.

Need good file system.

Solution: Google File System (GFS).♪

Motivational Facts♪

More than 15,000 commodity-class PC's.

Multiple clusters distributed worldwide.

Thousands of queries served per second.

One query reads 100's of MB of data.

One query consumes 10's of billions of CPU cycles.

Google stores dozens of copies of the entire Web!

Conclusion: Need large, distributed, highly fault tolerant file system.♪

Design constraints

Component 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

Files are huge by traditional standards

- Multi-GB files are common
- Billions of objects

Design constraints

Most modifications are appends

- Random writes are practically nonexistent
- Many files are written once, and read sequentially

Two types of reads

- Large streaming reads
- Small random reads (in the forward direction)

Sustained bandwidth more important than latency

File system APIs are open to changes

Interface Design

Not POSIX compliant

Additional operations

- Snapshot
- Record append

Topics♪

Design Motivations

Architecture

Read/Write/Record Append

Fault-Tolerance

Performance Results♪

Design Motivations♪

1. Fault-tolerance and auto-recovery need to be built into the system.
2. Standard I/O assumptions (e.g. block size) have to be re-examined.
3. Record appends are the prevalent form of writing.
4. Google applications and GFS should be co-designed.

GFS Architecture (Analogy)♪

On a single-machine FS:

- An upper layer maintains the metadata.
- A lower layer (i.e. disk) stores the data in units called “blocks”.

In the GFS:

- A master maintains the metadata.
- A lower layer (i.e. a set of *chunkservers*) stores the data in units called “chunks”.♪

Architectural Design (1)

A GFS cluster

- A single *master*
- Multiple *chunkservers* per master
  - Accessed by multiple *clients*
- Running on commodity Linux machines

A file

- Represented as fixed-sized *chunks*
  - Labeled with 64-bit unique global IDs
  - Stored at chunkservers
  - 3-way mirrored across chunkservers

Architectural Design (2)

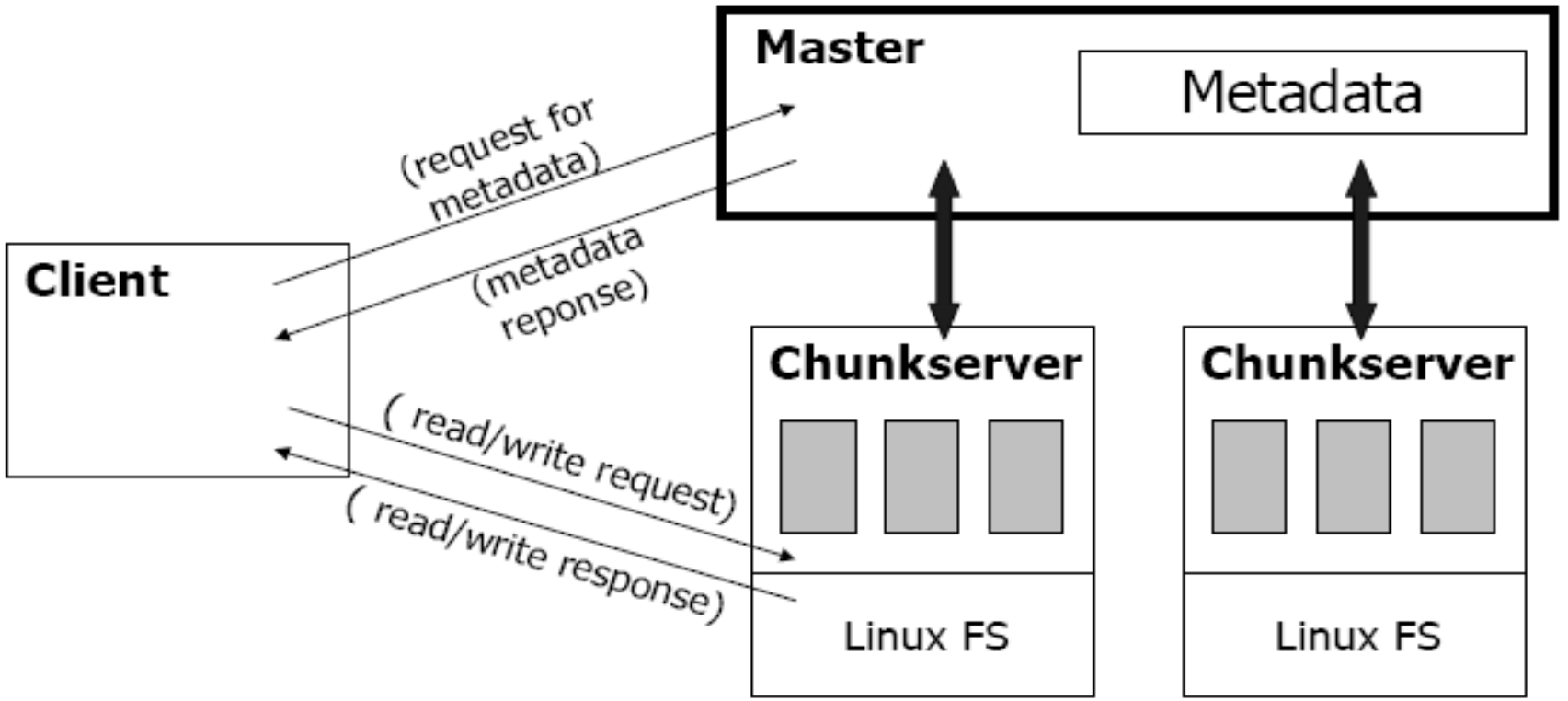
Master server

- Maintains all metadata
  - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration

GFS clients

- Consult master for metadata
- Access data from chunkservers
- Does not go through VFS
- No caching at clients and chunkservers due to the frequent case of streaming

GFS Architecture♪



GFS Architecture♪

What is a chunk?

- Analogous to block, except larger.
- Size: 64 MB!
- Stored on chunkserver as file
- Chunk handle (~ chunk file name) used to reference chunk.
- Chunk replicated across multiple chunkservers
- Note: There are hundreds of chunkservers in a GFS cluster distributed over multiple racks.♪

GFS Architecture♪

What is a master?

- A single process running on a separate machine.
- Stores all metadata:
  - File namespace
  - File to chunk mappings
  - Chunk location information
  - Access control information
  - Chunk version numbers
  - Etc.♪

GFS Architecture♪

Master <-> Chunkserver Communication:

- Master and chunkserver communicate regularly to obtain state:
  - Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunk replicas does chunkserver store?
- Master sends instructions to chunkserver:
  - Delete existing chunk.
  - Create new chunk.♪

Single-Master Design

Simple

Master answers only chunk locations

A client typically asks for multiple chunk locations in a single request

The master also predicatively provide chunk locations immediately following those requested

Chunk Size

64 MB

Fewer chunk location requests to the master

Reduce network overhead

Fewer metadata entries

- Kept in memory

- Some potential problems with fragmentation

Metadata

Three major types

- File and chunk namespaces
- File-to-chunk mappings
- Locations of a chunk's replicas

Metadata

All kept in memory

- Fast!
- Quick global scans
  - Chunk garbage collections
  - Reorganizations
- 64 bytes per 64 MB of data

Chunk Locations

No persistent states

- Polls chunkservers at startup
- Use heartbeat messages to monitor servers
- Simplicity
- On-demand approach vs. coordination
  - On-demand wins when changes (failures) are often

Operation Logs

Metadata updates are logged

- e.g., <old value, new value> pairs
- Log replicated on remote machines

Take global snapshots (checkpoints) to truncate logs

- B-tree like form and can be mapped into memory
- Checkpoints can be created while updates arrive

Recovery

- Latest checkpoint + subsequent log files

GFS Architecture♪

Serving Requests:

- Client retrieves metadata for operation from master.
- Read/Write data flows between client and chunkserver.
- Single master is not bottleneck, because its involvement with read/write operations is minimized.

♪

Overview♪

Design Motivations

Architecture

- Master
- Chunkservers
- Clients

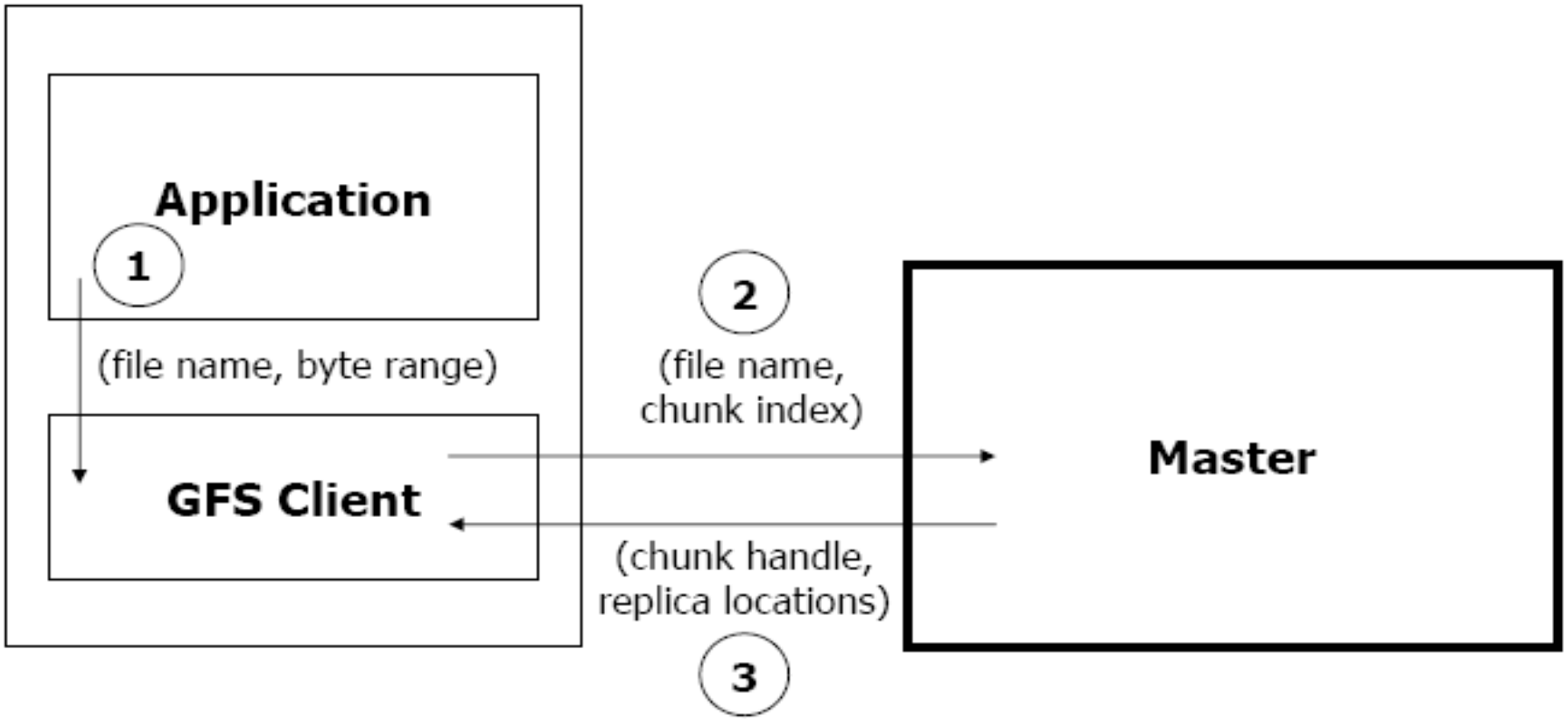
Read/Write/Record Append

Fault-Tolerance

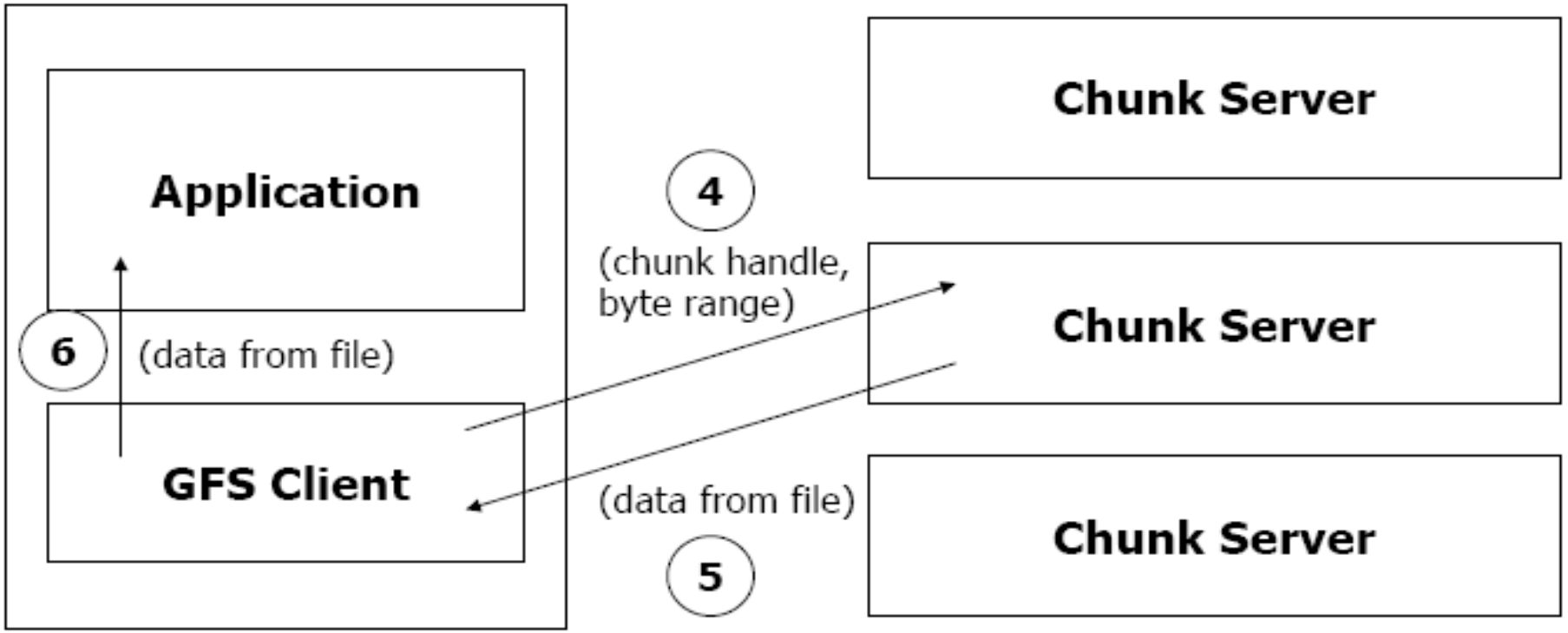
Performance Results♪

And Now for the Meat…♪

Read Algorithm♪



Read Algorithm♪



Read Algorithm♪

1. Application originates the read request.

2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.

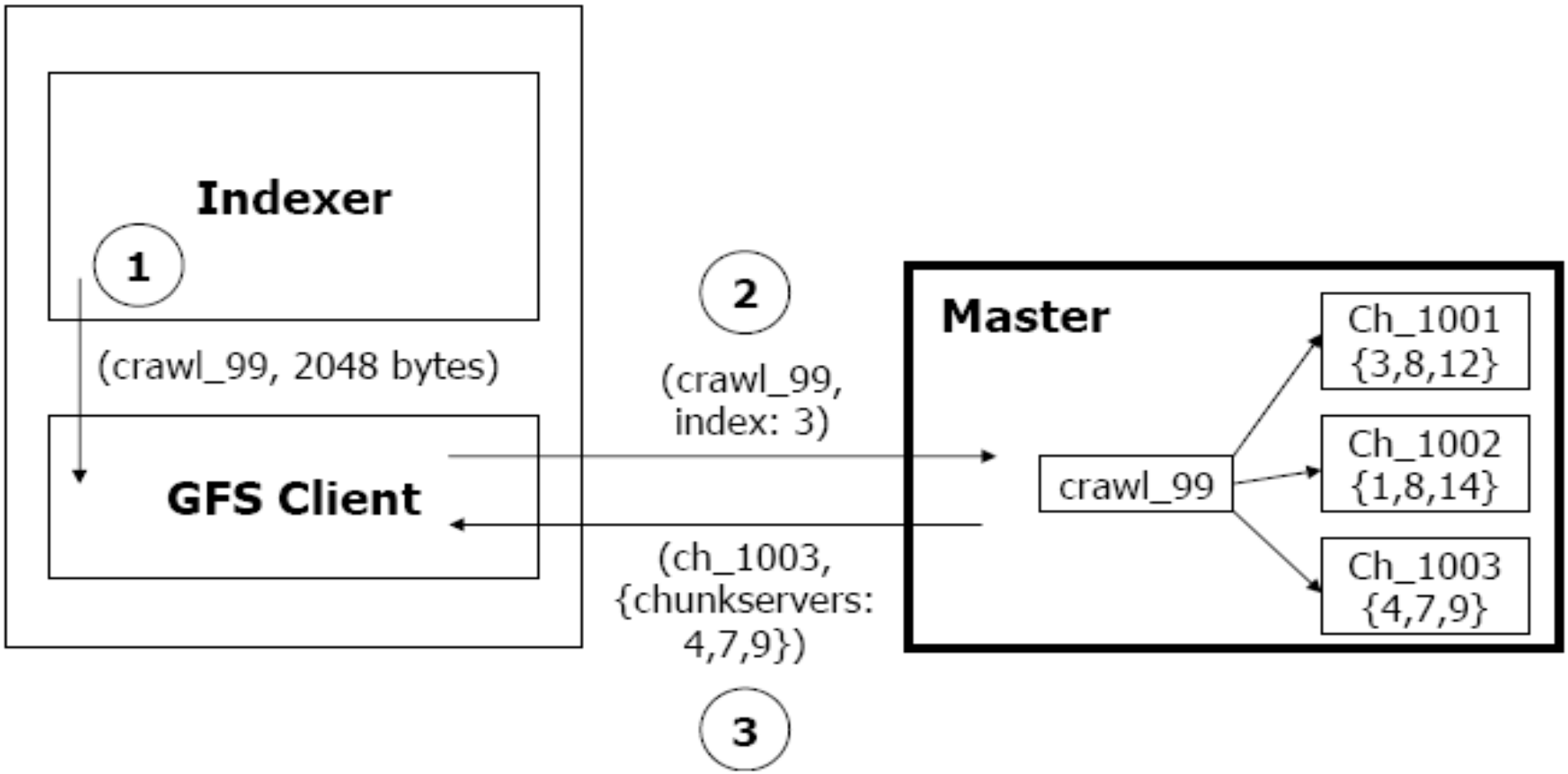
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored).

4. Client picks a location and sends the (chunk handle, byte range) request to that location.

5. Chunkserver sends requested data to the client.

6. Client forwards the data to the application.♪

Read Algorithm (Example)♪



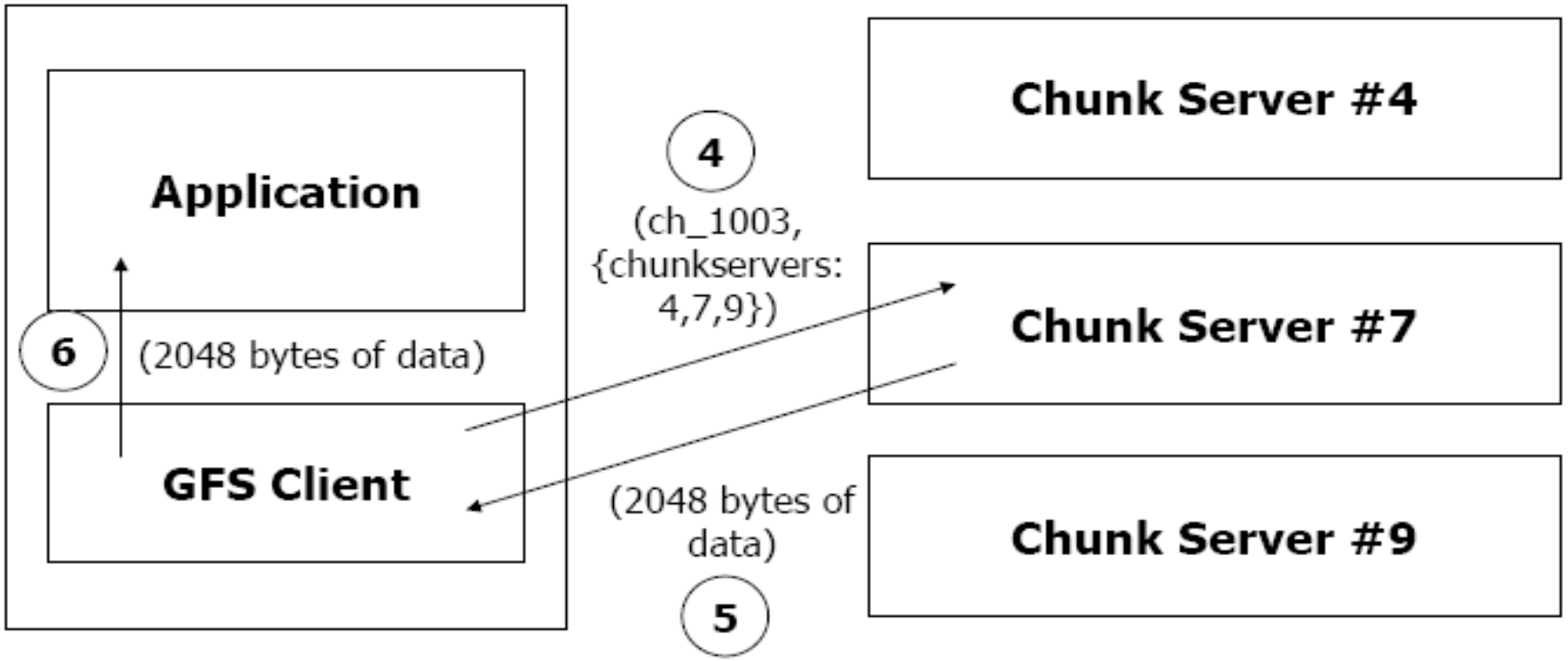
Read Algorithm (Example)♪

Calculating chunk index from byte range:

(Assumption: File position is 201,359,161 bytes)

- Chunk size = 64 MB.
- 64 MB = 1024 *1024 * 64 bytes = 67,108,864 bytes.
- 201,359,161 bytes = 67,108,864 * 2 + 32,569 bytes.
- So, client translates 2048 byte range -> chunk index 3.♪

Read Algorithm (Example)♪



Write Algorithm♪

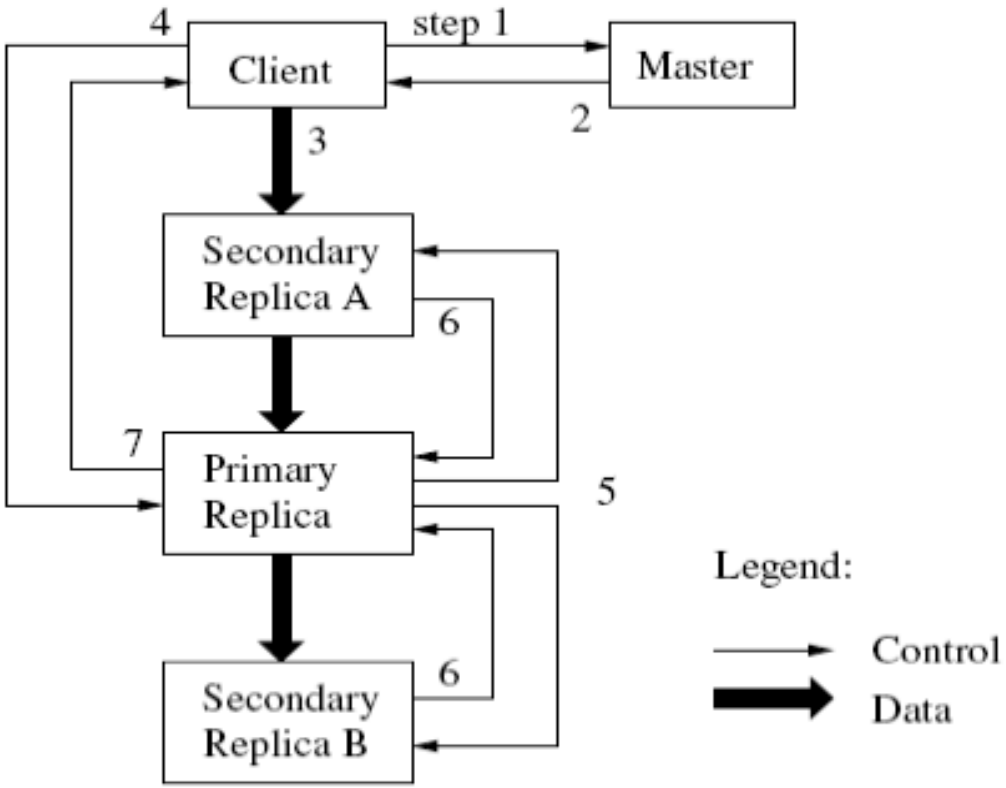
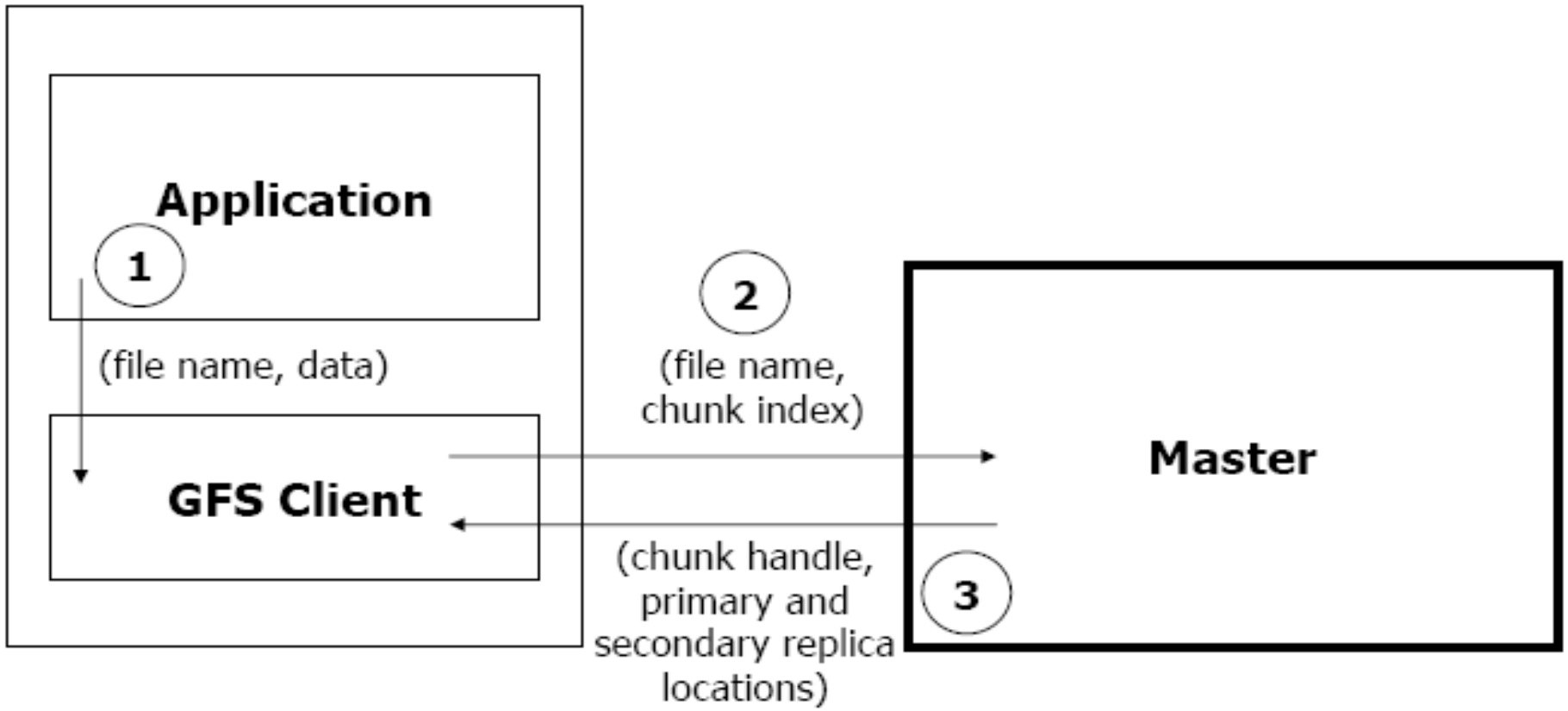
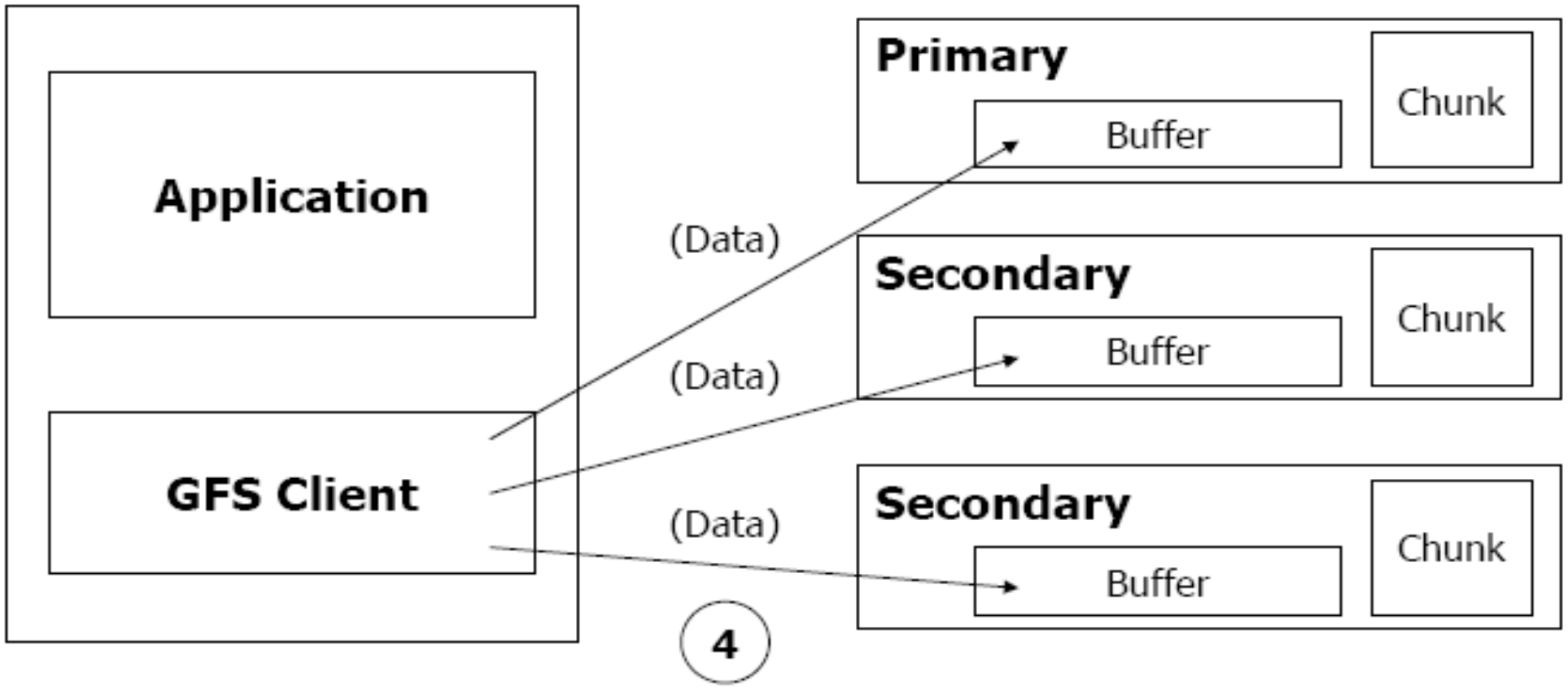


Figure 2: Write Control and Data Flow

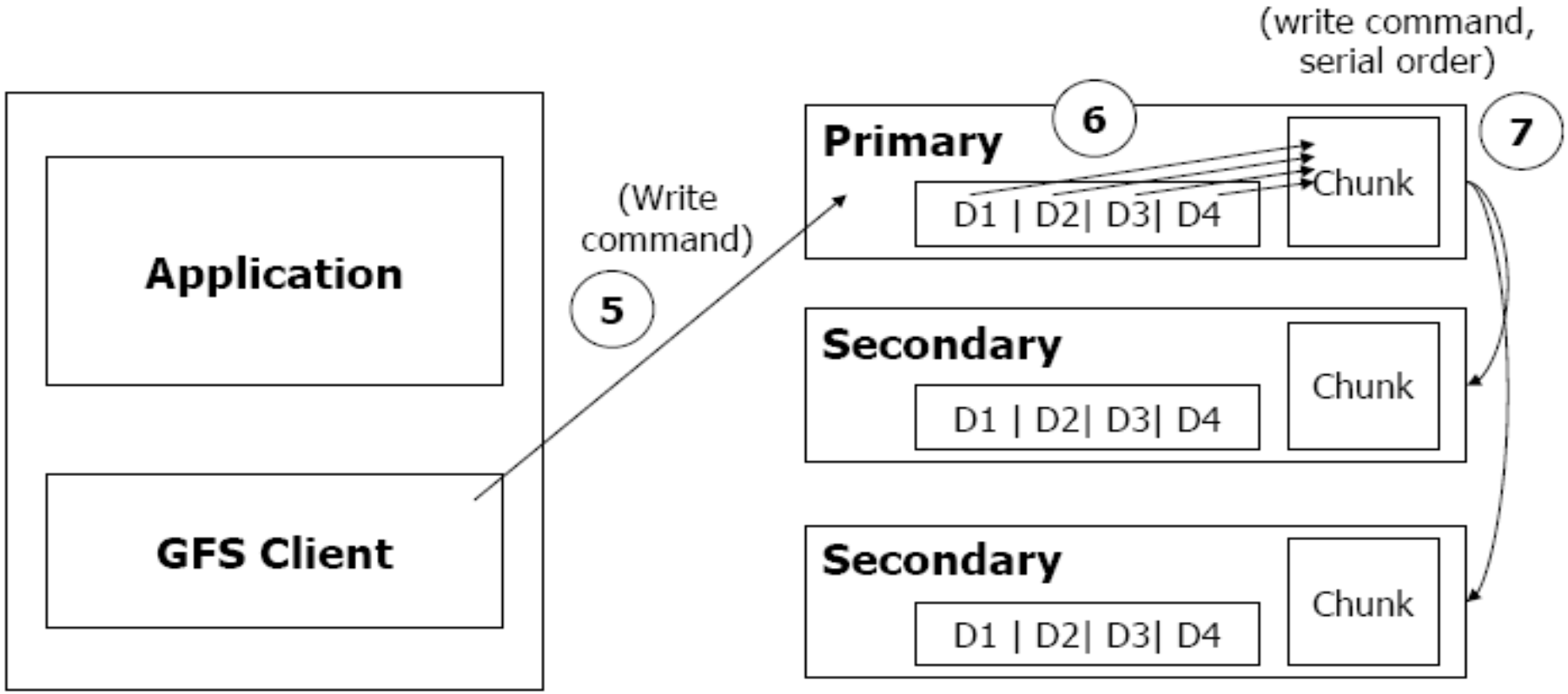
Write Algorithm♪



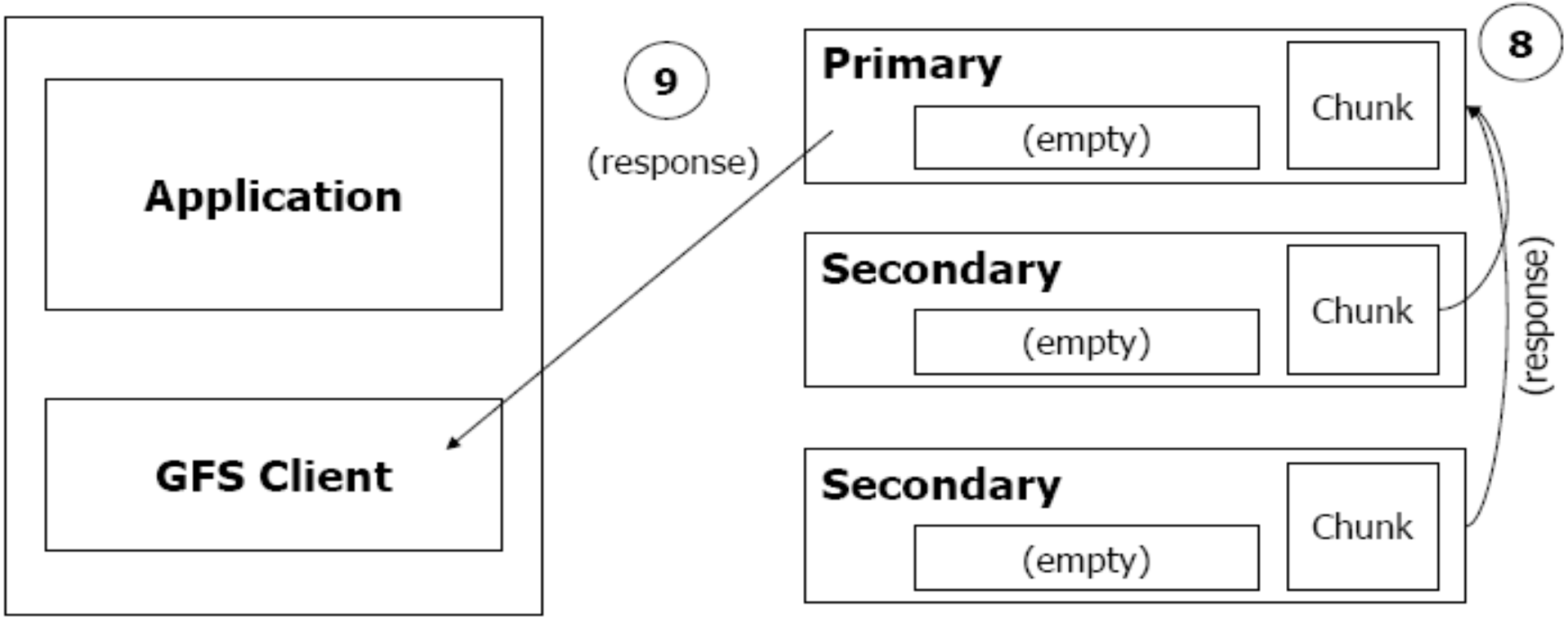
Write Algorithm♪



Write Algorithm♪



Write Algorithm♪



Write Algorithm♪

1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers.
5. Client sends write command to primary.♪

Write Algorithm♪

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

Note: If write fails at one of chunkservers, client is informed and retries the write.♪

Record Append Algorithm♪

Important operation at Google:

Merging results from multiple machines in one file.

Using file as producer - consumer queue.

1. Application originates record append request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations.♪

Record Append Algorithm♪

5. Primary checks if record fits in specified chunk.
6. If record does not fit, then the primary:
   - pads the chunk,
   - tells secondaries to do the same,
   - and informs the client.
   - Client then retries the append with the next chunk.
7. If record fits, then the primary:
   - appends the record,
   - tells secondaries to do the same,
   - receives responses from secondaries,
   - and sends final response to the client.♪

Observations♪

Clients can read in parallel.

Clients can write in parallel.

Clients can append records in parallel.♪

Consistency Model

Relaxed consistency

- Concurrent changes are consistent but undefined
  - All clients see same data but it may not reflect what any one mutation has written.
- An append is atomically committed at least once

  - Occasional duplications

All changes to a chunk are applied in the same order to all replicas

Use version number to detect missed updates

System Interactions

The master grants a chunk lease to a replica(becomes a primary)

The replica holding the lease determines the order of updates to all replicas

Lease

- 60 second timeouts
- Can be extended indefinitely
- Extension request are piggybacked on heartbeat messages
- After a timeout expires, the master can grant new leases

Data Flow

Separation of control and data flows

- Avoid network bottleneck

Updates are pushed linearly among replicas

Pipelined transfers

13 MB/second with 100 Mbps network

Snapshot

Copy-on-write approach

- Revoke outstanding leases
- New updates are logged while taking the snapshot
- Commit the log to disk
- Apply the log to a copy of metadata
- A chunk is not copied until the next update

Master Operation

No directories

No hard links and symbolic links

Full path name to metadata mapping

- With prefix compression

Locking Operations

A lock per path

- To access /d1/d2/leaf
- Need to lock /d1, /d1/d2, and /d1/d2/leaf
- Can modify a directory concurrently
  - Each thread acquires
    - A read lock on a directory
    - A write lock on a file
- Totally ordered locking to prevent deadlocks (first by level, then lexicographically)

Replica Placement

Several replicas for each chunk

- Default: 3 replicas

Goals:

- Maximize data reliability and availability
- Maximize network bandwidth

Need to spread chunk replicas across machines and racks

Replication

Chunks that need to be re-replicated: prioritized

- Higher priority to replica chunks with lower replication factors

Max Network Bandwidth Utilization

- Pipelined data writes

Primary replica:

- The master grants a chunk lease
- Manages the mutation order to the chunk

Garbage Collection

When a file is deleted, master logs the deletion and doesn't reclaim immediately

Simpler than eager deletion due to

- Unfinished replica creation
- Lost deletion messages

Garbage Collection

Deleted files are hidden for three days

Then they are garbage collected

Combined with other background operations (taking snapshots)

Safety net against accidental deletion

Fault Tolerance and Diagnosis

Fast recovery

- Master and chunkserver are designed to restore their states and start in seconds regardless of termination conditions

Chunk replication : across multiple machines, across multiple racks.

Fault Tolerance and Diagnosis

Master Mechanisms:

- Log of all changes made to metadata.
- Periodic checkpoints of the log.
- Log and checkpoints replicated on multiple machines.
- Master state is replicated on multiple machines.
- “Shadow” masters for reading data if “real” master is down.

Fault Tolerance and Diagnosis

Data integrity

- A chunk is divided into 64-KB blocks
- Each with its 32 bit checksum
- Verified at read and write times
- Also background scans for rarely used data

Performance (Test Cluster)♪

Performance measured on cluster with:

- 1 master, 2 master replicas
- 16 chunkservers
- 16 clients

Server machines connected to central switch by 100 Mbps Ethernet.

Same for client machines.

Switches connected with 1 Gbps link.♪

Measurements

Chunkserver workload

- Bimodal distribution of small and large files
- Ratio of write to append operations: 3:1 to 8:1
- Virtually no overwrites

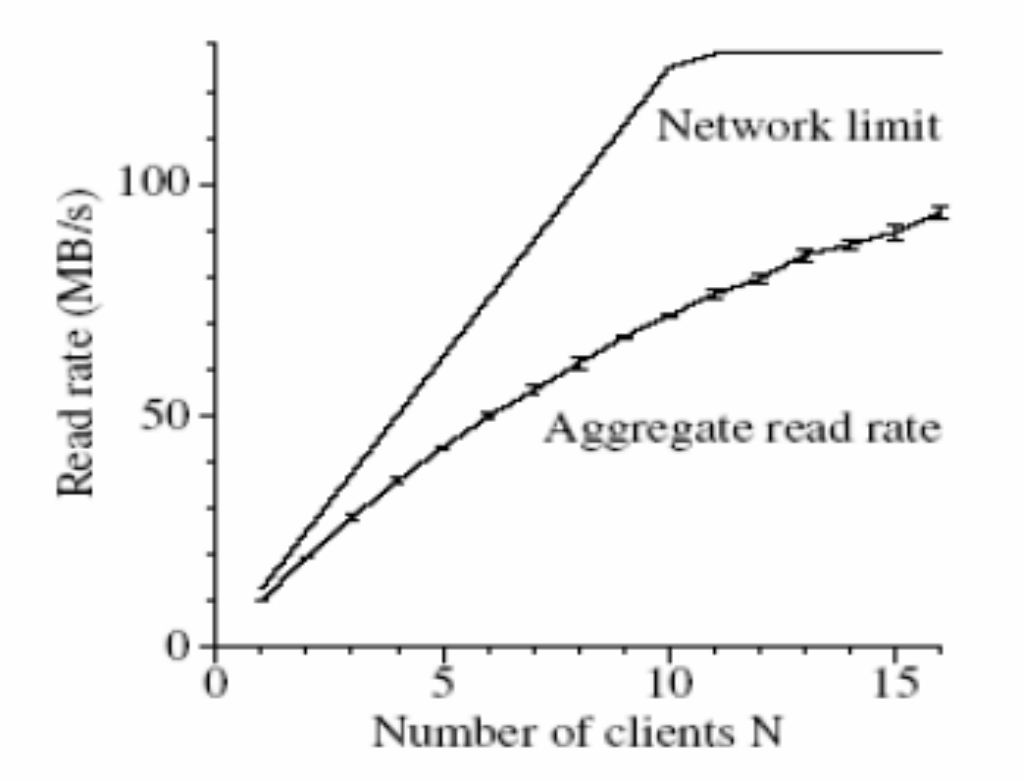
Master workload

- Most request for chunk locations and open files

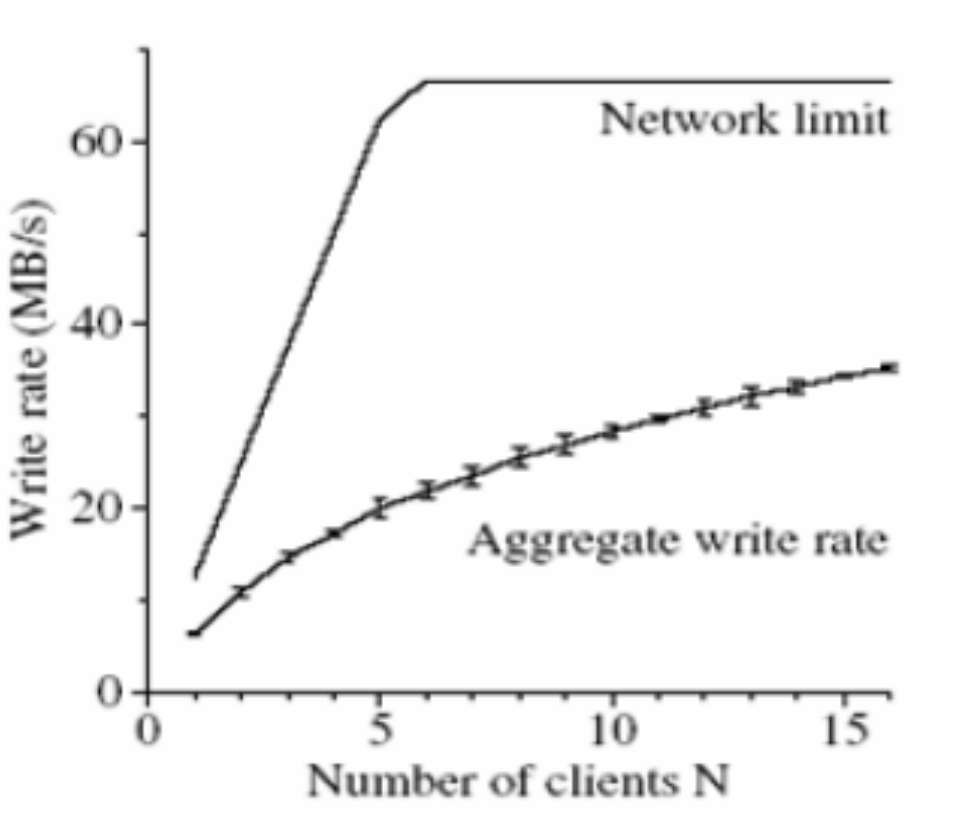
Reads achieve 75% of the network limit

Writes achieve 50% of the network limit

Performance (Test Cluster)♪



Performance (Test Cluster)♪



Performance (Real-world Cluster)♪

Cluster A:

- Used for research and development.
- Used by over a hundred engineers.
- Typical task initiated by user and runs for a few hours.
- Task reads MB's-TB's of data, transforms/analyzes the data, and writes results back.

Cluster B:

- Used for production data processing.
- Typical task runs much longer than a Cluster A task.
- Continuously generates and processes multi-TB data sets.
- Human users rarely involved.

Clusters had been running for about a week when measurements were taken.♪

Performance (Real-World Cluster)♪

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Performance (Real-world Cluster)♪

Many computers at each cluster (227, 342!)

On average, cluster B file size is triple cluster A file size.

Metadata at chunkservers:

- Chunk checksums.
- Chunk Version numbers.

Metadata at master is small (48, 60 MB)

-> master recovers from crash within seconds.♪

Performance (Real-world Cluster)♪

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Performance (Real-world Cluster)♪

Many more reads than writes.

Both clusters were in the middle of heavy read activity.

Cluster B was in the middle of a burst of write activity.

In both clusters, master was receiving 200-500 operations per second -> master is not a bottleneck.♪

Performance (Real-world Cluster)♪

Experiment in recovery time:

- One chunkserver in Cluster B killed.
- Chunkserver has 15,000 chunks containing 600 GB of data.
- Limits imposed:
  - Cluster can only perform 91 concurrent clonings.
  - Each clone operation can consume at most 6.25 MB/s.

Took 23.2 minutes to restore all the chunks.

This is 440 MB/s.

GFS: Summary

Optimized for large file reads and sequential appends

- large chunk size

File system API tailored to stylized workload

Single-master design to simplify coordination

Implemented on top of commodity hardware

GFS: Summary

Metadata fit in memory

Flat namespace

Dedicated Care for Component Failure

- Hard disk failure, data corruption, network disconnection, etc.

High-throughput

- Minimized the master involvement
  - Chunk servers themselves send and receive the client data
  - The master leases authority to mutate chunks