Google File System

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

2017 fall DIP

Heerak lim, Donghun Koo

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions

Introduction

Design choices :

1. Component failures are the norm rather than the exception.
2. Files are huge by traditional standards. Multi-GB files are comm on.
3. Most files are mutated by appending new data rather than over writing existing data.
4. Co-designing the applications and the file system API benefits t he overall system by increasing our flexibility.

Agenda

- Introduction
- Design overview
  - Assumptions
  - Interface
  - Architecture
  - Single master
  - Chunk size
  - Metadata
- Systems interactions
- Master operation
- Fault tolerance
- Measurements
- Conclusions

Design overview

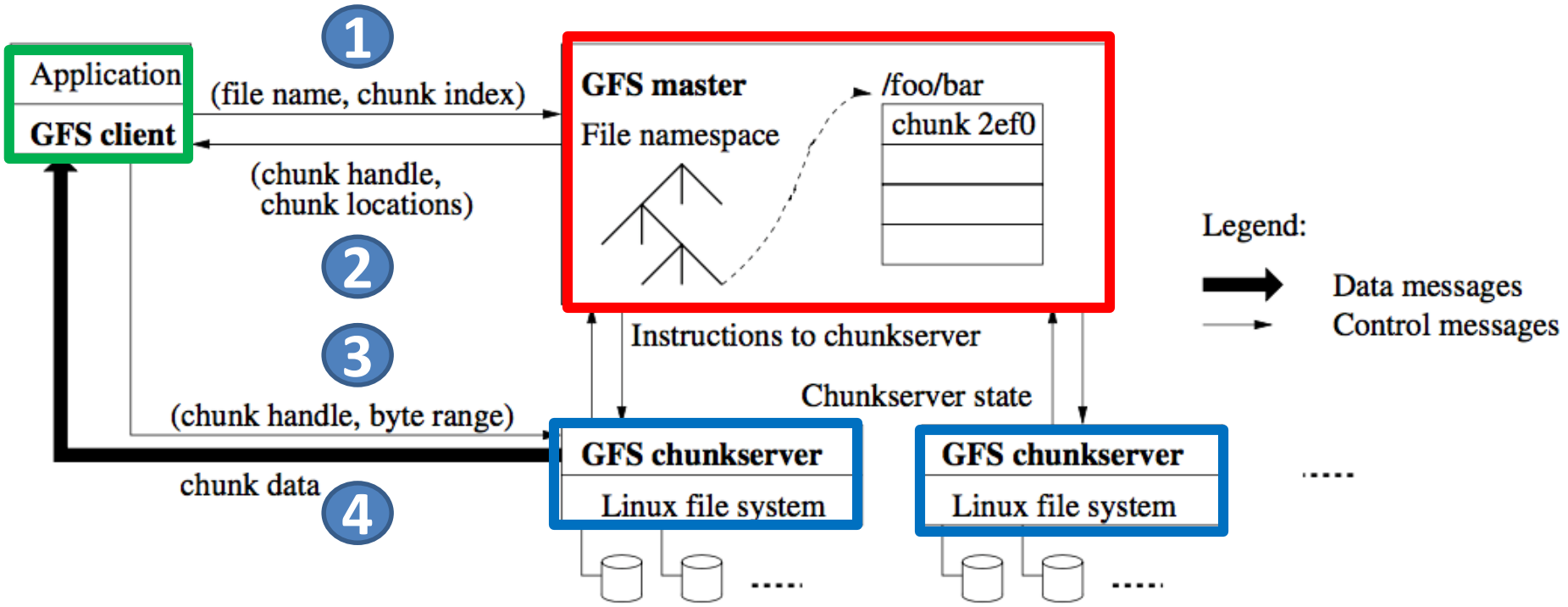
- Assumptions
  - inexpensive commodity components that often fail
  - Few million files, typically 100MB or larger
  - Large streaming reads, small random reads
  - Sequential writes, append data
  - Multiple concurrent clients that append to the same file
  - High sustained bandwidth more important than low latency

Design overview

- Interface
  - Familiar file system interface API but not POSIX compatible
  - Usual files operations:
    - create
    - delete
    - open
    - close
    - read
    - write
  - Enhanced files operations:
    - snapshot (copy--on--write)
    - record append (concurrency atomic append support)

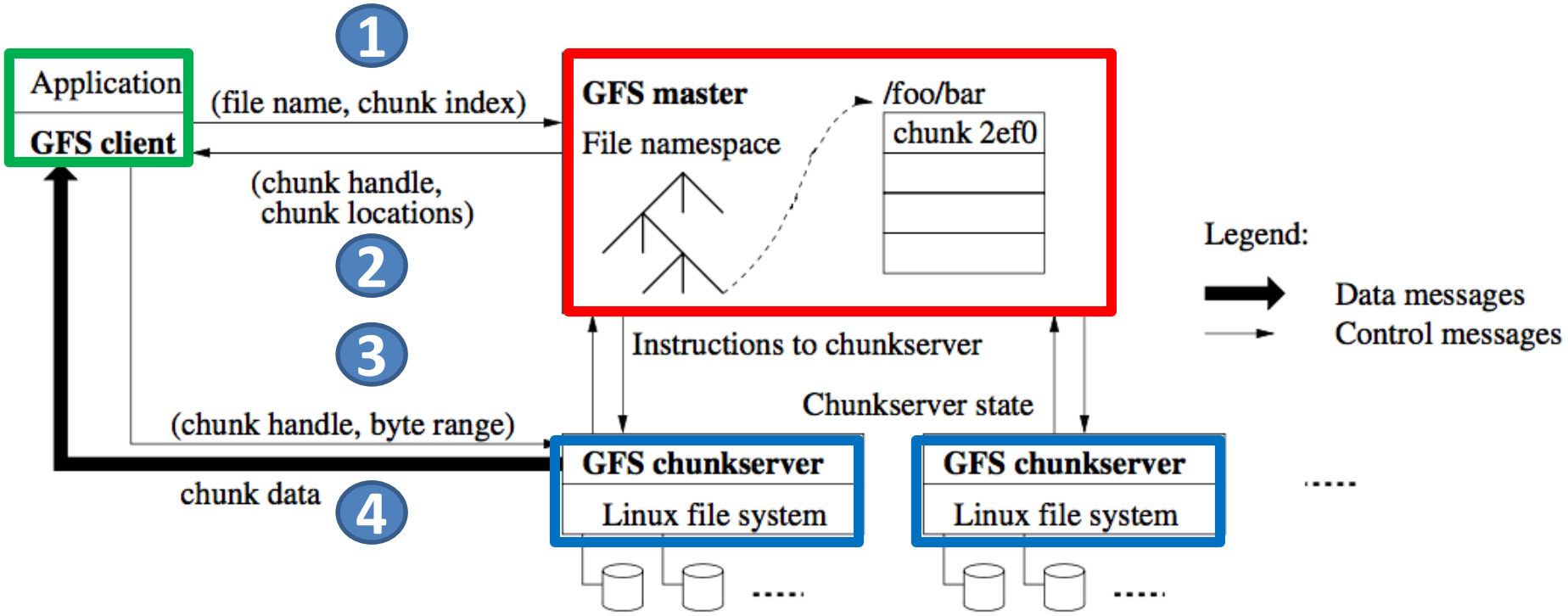
Design overview

- **Architecture**
  - Single ***GFS master*** ! and multiple ***GFS chunkservers*** accessed by multiple ***GFS clients***
  - GFS Files are divided into fixed–size chunks (64MB)
  - Each chunk is identified by a globally unique “chunk handle” (64 bits)
  - Chunkservers store chunks on local disks as Linux file
  - For reliability each chunk is replicated on multiple chunkservers (default = 3)



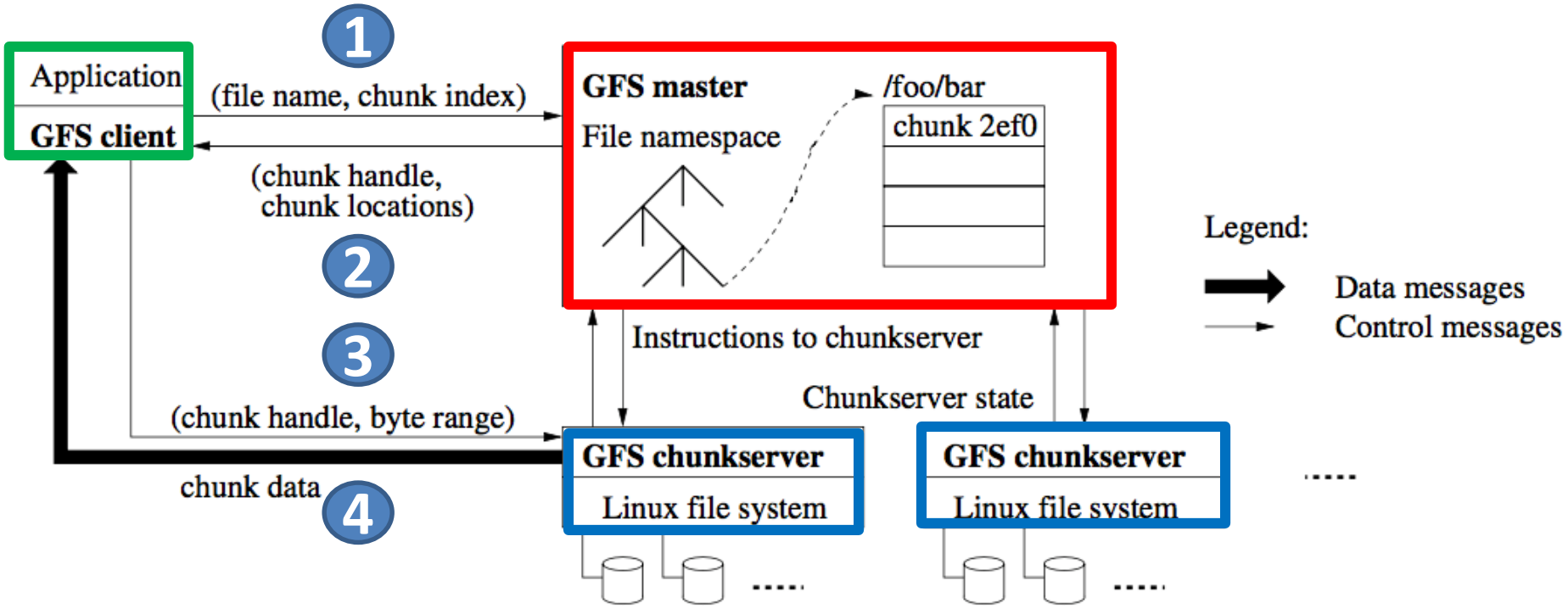
Design overview

- **Architecture**
  - GFS master
    - maintains all file system metadata
      - namespace, access control, chunk mapping & locations (files→→chunks→→replicas)
    - send periodically heartbeat messages with chunkservers
      - instructions + state monitoring



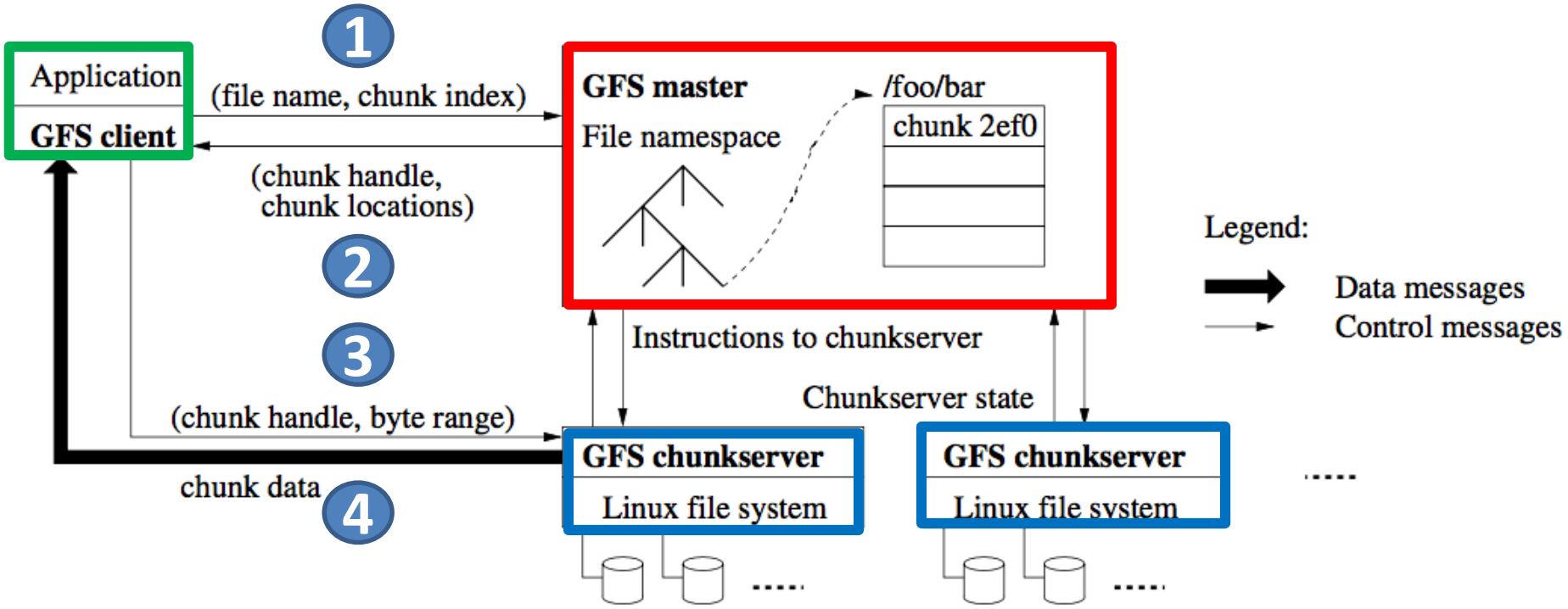
Design overview

- **Architecture**
  - GFS client
    - Library code linked into each applications
    - Communicates with GFS master for metadata operations (control plane)
    - Communicates with chunkservers for read/write operations (data plane)



Design overview

- **Single master**
  - Design simplification
  - Master makes chunk placement and replication decisions



Design overview

- **Chunk size**
  - 64MB
  - Stored as a plain Linux file on chunkserver
  - Why large chunk size ?
    - Advantages
      - Reduces client interaction with GFS master for chunk location information
      - Reduces size of metadata stored on master (full in--memory)
      - Reduces network overhead by keeping persistent TCP connections to the chunkserver over an extended period of time
    - Disadvantages
      - Small files can create hot spots on chunkservers if many clients accessing the same file

Design overview

- **Metadata**
  - **3 types of metadata**
    - File and chunk namespaces *(in--memory + operation log)*
    - Mapping from files to chunks *(in--memory + operation log)*
    - Locations of each chunks'replicas *(in--memory only)*
  - **All metadata is kept in GFS master's memory (RAM)**
    - Periodic scanning to implement chunk garbage collection, re--replication (when chunks erver failure) and chunk migration to balance load and disk space usage across chunk servers
  - **Operation log file**
    - The logical time line that defines the order of concurrent operations
    - Contains only metadata Namespaces + Chunk mapping
    - kept on GFS master's local disk and replicated on remote machines
    - No persistent record for chunk locations (master polls chunkservers at startup)
    - GFS master checkpoint its state (B--tree) whenever the log grows beyond a certain size

Agenda

- Introduction
- Design overview
- **Systems interactions**
  - Leases and mutation order
  - Data flow
  - Atomic record appends
  - Snapshot
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions

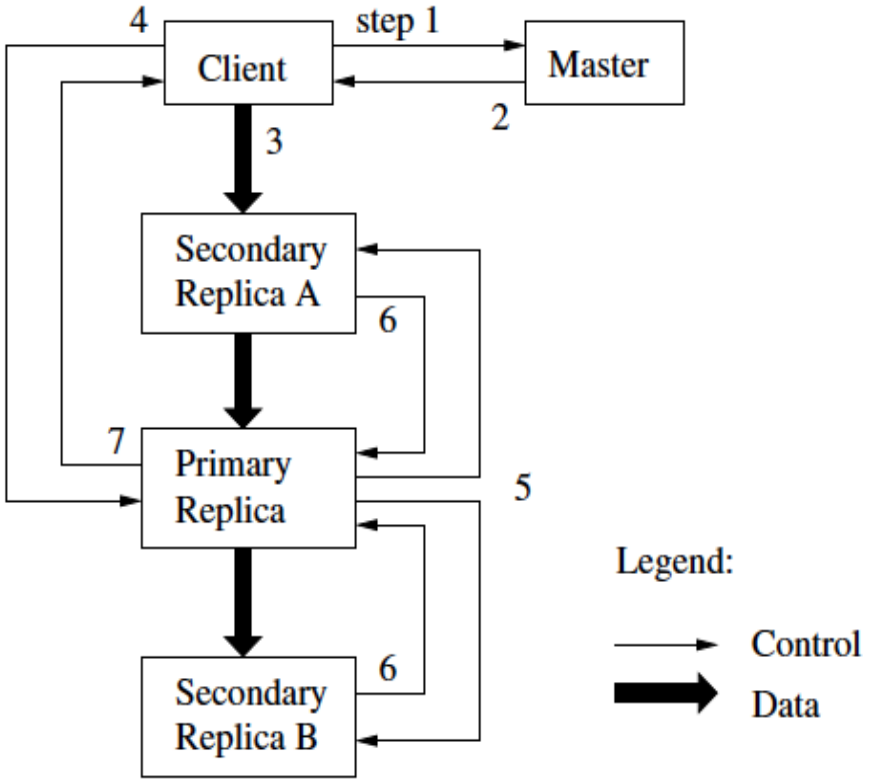
Systems interactions

- **Leases and mutation order**
  - Each mutation *(write or append)* is performed at all the chunk's replicas
  - Leases used to maintain a consistent mutation order across replicas
    - Master grants a chunk lease to one of the replicas:(the primary replica)
    - The primary replica picks a serial order for all mutations to the chunk
    - All replicas follow this order when applying mutations
  - Master may sometimes try to revoke a lease before it expires (when master wants to disable mutations on a file that is being renamed)

Systems interactions

Write control and Data flow

1. Client ask master for primary and secondaries replicas info for a chunk
1. Master replies with replicas identity (client put info in cache with timeout)
1. Client pushes Data to all replicas (pipelined fashion) Each chunkserver store the Data in an internal LRU buffer cache until Data is used or aged out
2. One all replicas have acknowledged receiving the data, client send a write request to the primary replica
3. Primary replica forward the write request to all secondary replicas that applies mutations in the same serial number order assigned by the primary
4. Secondary replicas acknowledge the primary that they have completed the operation
5. The primary replica replies to the client. Any errors encountered at any of the replicas are reported to the client.



Systems interactions

- **Data flow**
  - **To utilize high bandwidth**
  - **To avoid network bandwidth and high-latency links**

  – GFS decouples the Data flow from Control flow
    -> Looks linear

  – Each machine forwards the data to the closest machine in the network topology that has not received it
    closest: Our network topology is simple enough that “distances” can be accurately estimated from IP addresses.

  – GFS minimize the latency by pipelining the data transfer over TCP connections

Systems interactions

- **Atomic record appends(called *record append*)**
  - In a record append, client specify only the data, and GFS choose the offset when appending the data to the file (concurrent atomic record appends are serializable)

Algorithm of atomic record append

- Client pushes the data to all replicas of the last chunk of the file
- Client sends the append request to the primary
- Primary checks to see if data causes chunk to exceed the maximize size

  Yes-> pads chunk to the maximum size and tells secondaries to do same then tells client to retry on the next chunk.

  No-> the primary appends data to its replicas, tells secondaries to write at the exact offset where is has.

Systems interactions

- **Snapshot operation**
  - Make a instantaneously copy of a file or a directory tree by using standard copy–on–write techniques
    - Revoke any outstanding leases on the chunk to snapshot
    - Logs the snapshot operation to disk and duplicates the metadata
    - If the client wants to write, it creates a new chunk and tell secondary to do same

Agenda

- Introduction
- Design overview
- Systems interactions
- **Master operation**
  - Namespace management and locking
  - Replica placement
  - Creation, re--replication, rebalancing
  - Garbage collection
- Fault tolerance
- Measurements
- Conclusions

Namespace management and locking

- **GFS allows multiple operations to be active and uses locks to ensure serialization**
- **Namespace is represented as a lookup table mapping full pathnames to metadata**
- **Each node has an associated read-write lock and each master operation acquires locks before it runs**
- **Locks are acquired in a total order to prevent deadlock**

Replicas

- **GFS spreads chunk replicas across racks**
  - This ensures that some replicas will survive even if an entire rack is damaged
  - This can exploit the aggregate bandwidth of multiple racks
- **Replicas are created for chunk creation, re-replication, and rebalancing**
  - Places new replicas on chunkservers with below-average disk space utilization
  - Re-replicates a chunk when the number of available replicas falls below a user-specified goal
  - Rebalances replicas periodically for better disk space and load balancing

Garbage Collection

- **GFS reclaims the storage for deleted files lazily via garbage collection at the file and chunk levels**
  - **File Level**
    - Master logs the deletion and renames the file to a hidden name that includes the deletion timestamp
    - Hidden files are removed if they have existed for more than 3 days
  - **Chunk Level**
    - Each chunkserver reports a subset of chunks it has and master identi fies and replies with orphaned chunks
    - Each chunkserver deletes orphaned chunks

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- **Fault tolerance**
  - High Availability
  - Data Integrity
- Measurements
- Conclusions

High availability

- **Fast recovery**
  - Master and chunkserver are designed to restore their state and start in seconds
- **Chunk replication**
  - Each chunk is replicated on multiple chunkservers on different racks
  - Users can specify different replication levels (default is 3)
- **Master replication**
  - Operation logs and checkpoints are replicated on multiple machines
  - One master process is in charge of all mutations and background activities
  - Shadow masters provide read-only access to the file system when the primary master is down

Data integrity

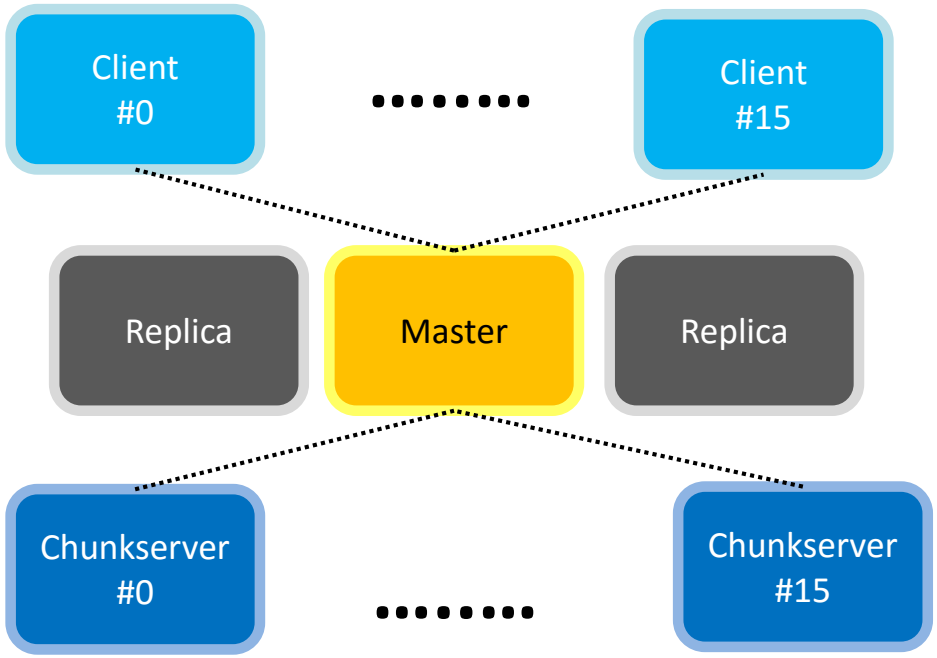
- **Each chunkserver uses checksumming to detect data corruption**
  - A chunk is broken up into 64KB blocks
  - Each block has a corresponding 32bit checksum
- **Chunkserver verifies the checksum of data blocks for reads**
- **During idle period, chunkservers scan and verify the inactive chunks**

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- Fault tolerance
- **Measurements**
  - Micro--benchmarks
  - Real world clusters
- Conclusions

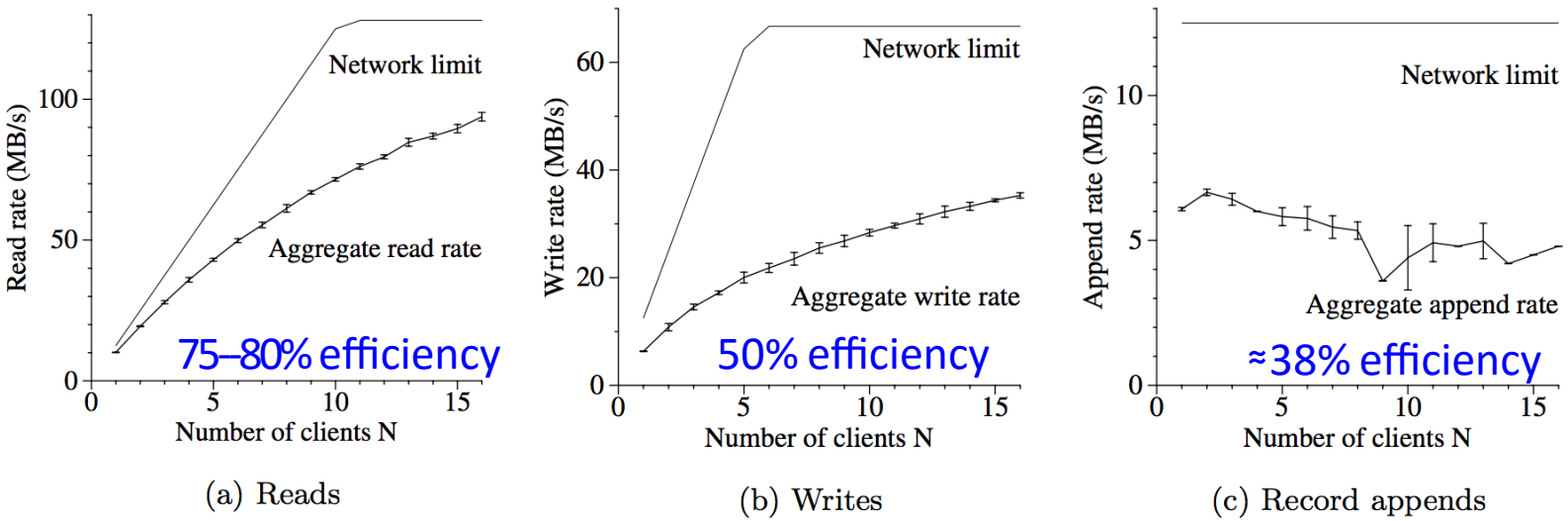
Micro-benchmarks

- **Experiment Environment**
  - 1 **master**, 2 **master replicas**, 16 **chunkservers** with 16 **clients**
  - **Dual 1.4 GHz PIII** processors, **2GB RAM**, **2x80GB** 5400 rpm disks, 100Mbps full-deplex Ethernet



Micro-benchmarks

- **READS**
  - Each client read a randomly selected 4MB region 256 times (= 1 GB of data) from a 320MB file
- **Writes**
  - Each client writes 1GB data to a new file in a series of 1MB writes
  - Each write involves 3 different replicas
- **Record appends**
  - All clients append simultaneously to a single file



(a) Reads (b) Writes (c) Record appends

Real world clusters

- **Experiment Environment**
  - Cluster A
    - Used for research and development
    - Reads MBs ~ TBs data and writes the results back
  - Cluster B
    - Used for research and development
    - Reads MBs ~ TBs data and writes the results back

	R&D	Production
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

Real world clusters

- **Performance metrics for two GFS clusters**
  - Cluster B was in the middle of a burst write activity
  - The read rates were much higher than the write rates

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

- **Recovery time**
  - Killed a single chunckserver of cluster B
    - Which had 15K chunks containing 600GB data
  - All chunks were restored in 23.3 minutes

Conclusions

- **GFS**
  - GFS treats Component failures as the norm rather than the exception
  - GFS optimizes for huge files mostly append to and then read sequentially
  - GFS provides fault tolerance by constant monitoring, replicating crucial data and fast and automatic recovery (+ checksum to detect data corruption)
  - GFS delivers high aggregate throughput to many concurrent readers and writers (by separating file system control from data transfer)