Big Data Processing, 2014/15

Lecture 5: GFS & HDFS

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- **Looking behind the scenes of MapReduce: HDFS** & Scheduling
- Algorithm design for MapReduce
- A high-level language for MapReduce: Pig 1 & 2
- MapReduce is not a database, but HBase nearly is
- Lets iterate a bit: Graph algorithms & Giraph
- How does all of this work together? ZooKeeper/Yarn

Learning objectives

- **Explain** the design considerations behind GFS/HDFS
- **Explain** the basic procedures for data replication, recovery from failure, reading and writing
- **Design** alternative strategies to handle the issues GFS/HDFS was created for
- **Decide** whether GFS/HDFS is a good fit given a usage scenario

The Google File System

Hadoop is *heavily* inspired by it.

One way (not *the* way) to design a distributed file system.

there are always alternatives

MapReduce & Hadoop

"MapReduce is a programming model for expressing **distributed** computations on **massive amounts of data** and an execution framework for large-scale data processing on clusters of **commodity servers**."

-Jimmy Lin

GFS

Hadoop is an **open-source implementation** of the MapReduce framework.

HDFS

History of MapReduce (and GFS)

- Developed by researchers at **Google** around **2003**
  - Built on principles in parallel and distributed processing
- Seminal papers:
  - *The Google file system* by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (2003)
  - *MapReduce: Simplified Data Processing on Large Clusters.* by Jeffrey Dean and Sanjay Ghemawat (2004)
- *The Hadoop distributed file system* by Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler (2010)

What is a file system?

- File systems determine **how** data is stored and retrieved

- **Distributed file systems** manage the storage across a network of machines
  - Added complexity due to the network

- GFS/HDFS are distributed file systems

Question: which file systems do you know?

GFS Assumptions

based on Google's main use cases

- Hardware **failures are common** (commodity hardware)
- **Files are large** (GB/TB) and their number is limited (millions, not billions)
- Two main types of reads: **large streaming reads** and **small random reads**
- Workloads with **sequential writes** that **append** data to files
- Once written, files are **seldom modified** (!=append) again
  - Random modification in files possible, but not efficient in GFS
- High sustained bandwidth trumps low latency

Question: which of the following scenarios fulfil the brief?

- Global company dealing with the data of its 100 million employees (salary, bonuses, age, performance, etc.)
- A search engine’s query log (a record of what kind of search requests people make)
- A hospital’s medical imaging data generated from an MRI scan
- Data sent by the Hubble telescope
- A search engine’s index (used to serve search results to users)

Disclaimer

- GFS/HDFS are not a good fit for:
  - **Low latency data access** (in the milliseconds range)
    - Solution: use HBase instead [later in the course]
  - **Many small files**
    - Solution: *.har *.warc **[later in this lecture]**
  - **Constantly changing data**

- Not all details of GFS are public knowledge

Question: how would you design a distributed file system?

Application

How to write data to the cluster?

How to read data from the cluster?

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

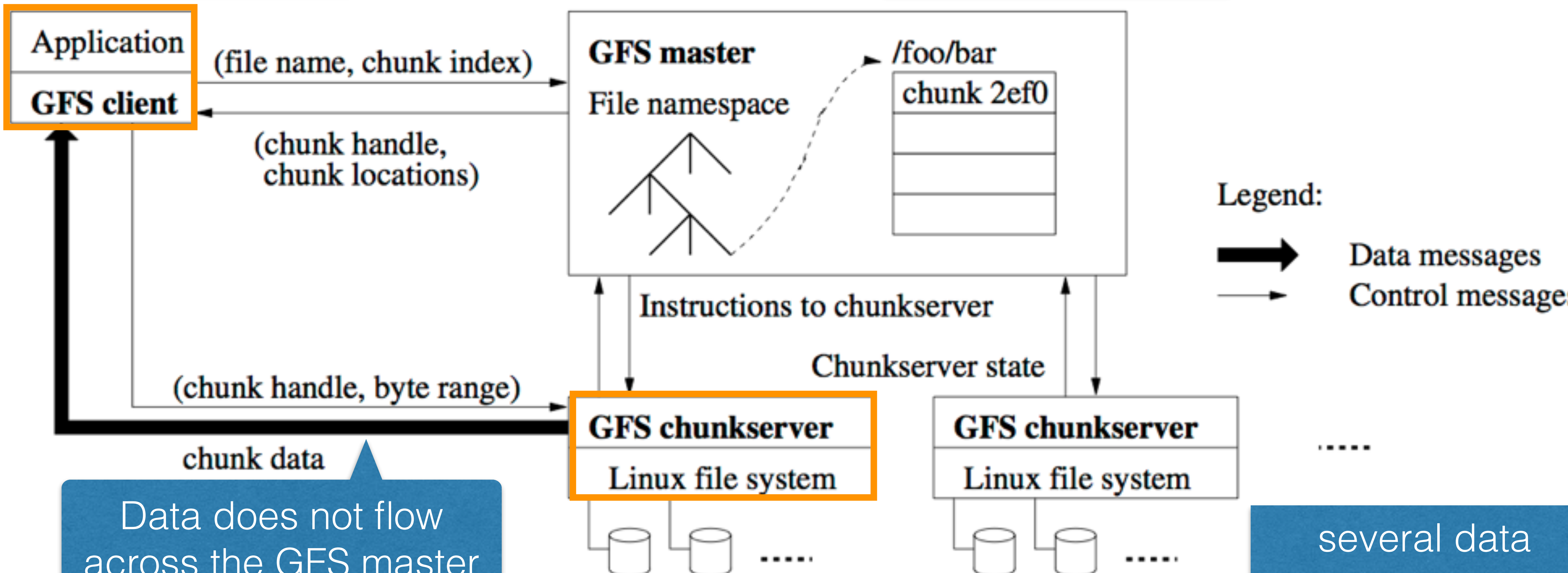
GFS architecture

user level processes: they can run on the same physical machine

Remember: **one way, not the way**.

several clients

a single master (metadata)



Data does not flow across the GFS master

several data servers

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

GFS: Files

Files on GFS

- A **single file** can contain **many objects** (e.g. Web documents)

- Files are divided into **fixed size chunks** (64MB) with unique 64 bit identifiers

Question: what does this ↑ mean for the maximum allowed file size in a cluster?

Linux files

  - Reading & writing of data specified by the **tuple** `(chunk_handle, byte_range)`

Question: what is the purpose of `byte_range`?

Files on GFS

- A **single file** can contain **many objects** (e.g. Web documents)
- Files are divided into **fixed size chunks** (64MB) with unique 64 bit identifiers
  - IDs assigned by GFS master at chunk creation time
- **chunkservers** store chunks on local disk as “normal” Linux files
  - Reading & writing of data specified by the **tuple** `(chunk_handle, byte_range)`

Question: what is the purpose of `byte_range`?

Master

- Files are **replicated** (by default 3 times) across all chunk servers
- **master** maintains all **file system metadata**
    - Namespace, access control information, **mapping from file to chunks**, **chunk locations**, garbage collection of orphaned chunks, chunk migration, …

distributed systems are complex!

- **Heartbeat** messages between master and chunk servers
    - Is the chunk server still **alive**? **What chunks are stored at the chunkserver?**
- **To read/write data**: client communicates with master (metadata operations) and chunk servers (data)

Files on GFS

- seek time: 10ms
- transfer rate: 100MB/s
- What is the chunk size to make the seek time 1% of the transfer rate?

- Clients **cache metadata**
- Clients do **not** cache file data

Question: why don't clients store file data?

Question: why not increase the chunk size to more than 128MB? (Hint: Map tasks operate on one chunk at a time**)**

- A single file can be **larger** than a node's disk space
- Fixed size makes **allocation computations easy**

Files on GFS

- seek time: 10ms
- transfer rate: 100MB/s
- What is the chunk size to make the seek time 1% of the transfer rate?

- Clients **cache metadata**
- Clients do **not** cache file data
- Chunkservers do **not** cache file data (responsibility of the **underlying file system**: Linux's buffer cache)
- Advantages of (large) fixed-size chunks:
  - **Disk seek time small compared to transfer time**
  - A single file can be **larger** than a node's disk space
  - Fixed size makes **allocation computations easy**

GFS: Master

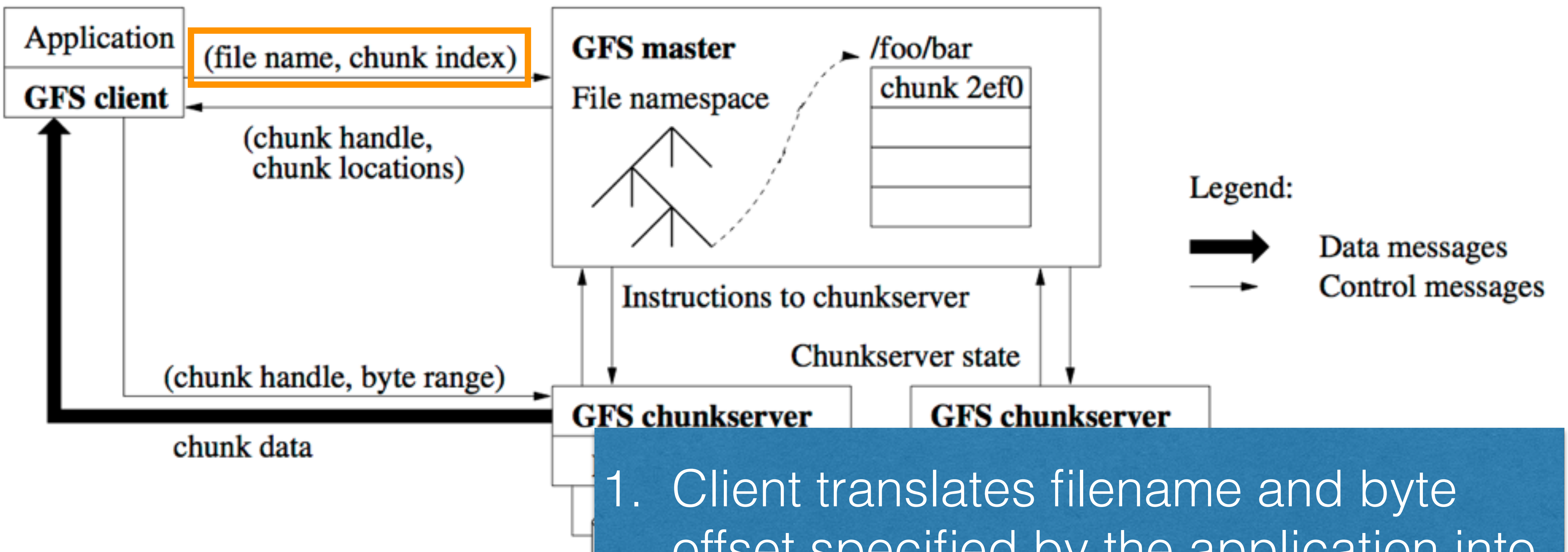
One master

- Single master **simplifies the design** tremendously
  - Chunk placement and replication with **global knowledge**

- Single master in a large cluster can become a **bottleneck**
  - Goal: **minimize the number of reads and writes** (thus metadata vs. data)

Question: as the cluster grows, can the master become a bottleneck?

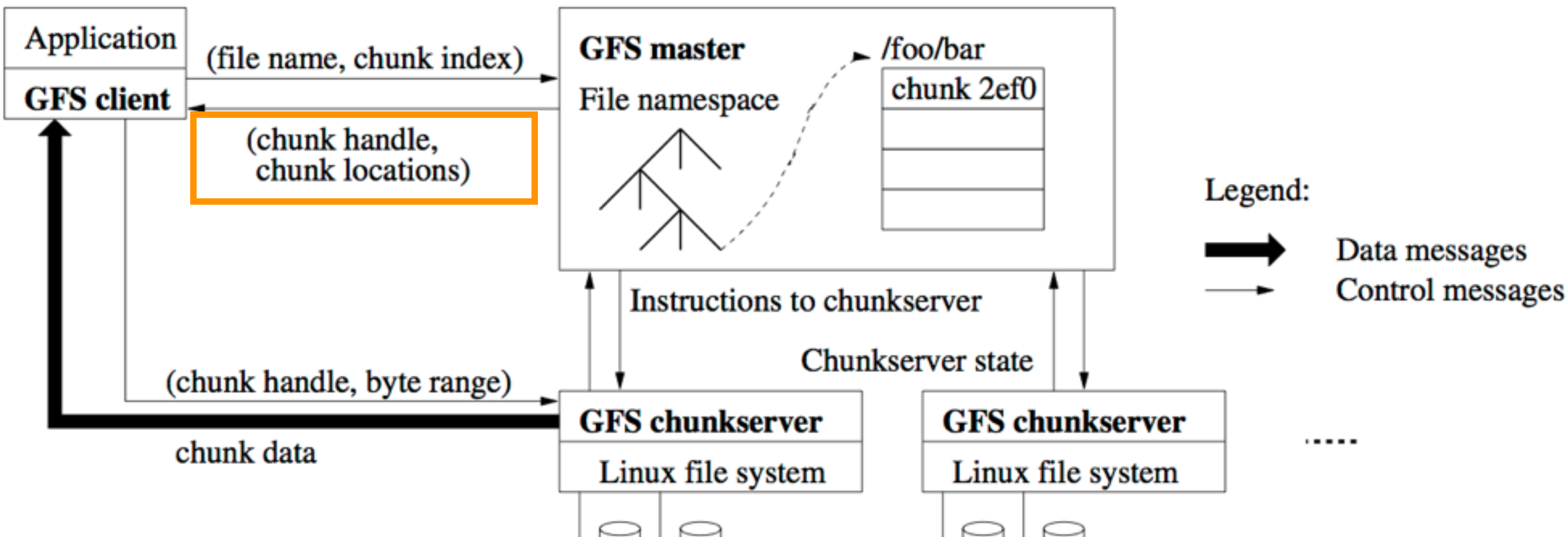
A read operation (in detail)



1. Client translates filename and byte offset specified by the application into a chunk index within the file. Sends request to master.

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

A read operation (in detail)



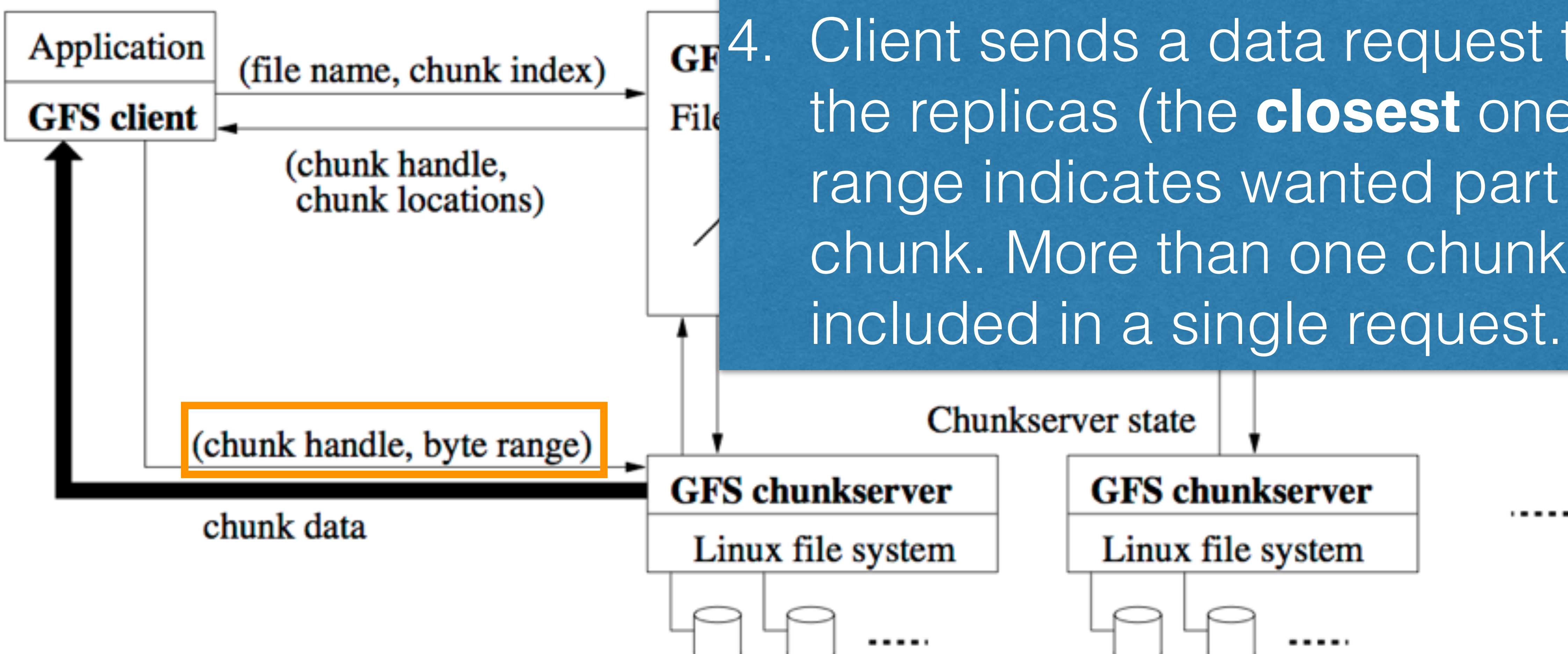
2. Master replies with chunk handle and locations.

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

A read operation (in detail)

3. Client caches the metadata.

4. Client sends a data request to one of the replicas (the **closest** one). Byte range indicates wanted part of the chunk. More than one chunk can be included in a single request.

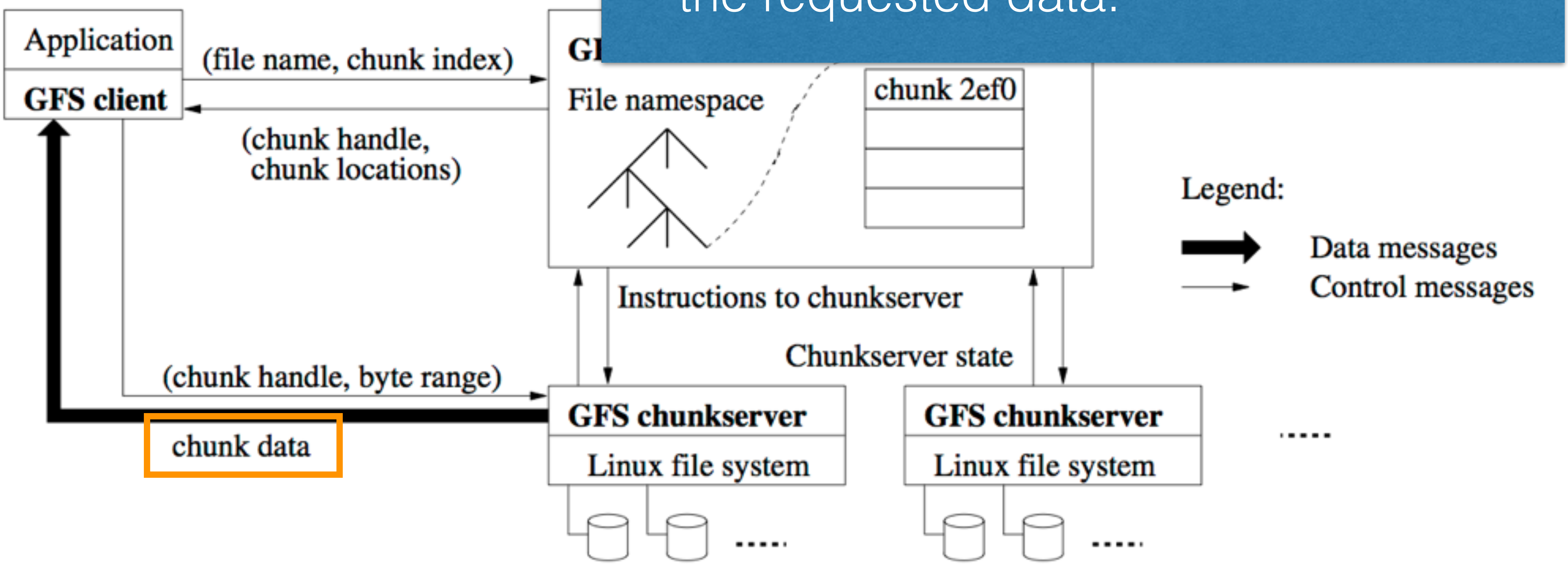


Question: how can the cluster topology be discovered automatically?

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

A read operation (in detail)

5. Contacted chunk server replies with the requested data.



Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Metadata on the master

Question: what happens to the cluster if the master crashes?

- **3 types of metadata**
  - Files and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All metadata is kept in master's **memory** (fast random access)
  - Sets limits on the entire system's capacity
- **Operation log** is kept on master's local disk: in case of the master's crash, master state can be recovered
  - Namespaces and mappings are logged
  - Chunk locations are **not** logged

GFS: Chunks

Chunks

- 1 chunk = 64MB or 128MB (can be changed); chunk stored as a **plain Linux file** on a chunk server
- Advantages of large (but not too large) chunk size
  - **Reduced** need for client/master **interaction**
  - 1 request per chunk suits the target workloads
  - Client can **cache all the chunk locations** for a multi-TB working set
  - **Reduced size of metadata** on the master (kept in memory)
- Disadvantage: chunkserver can become **hotspot** for popular file(s)

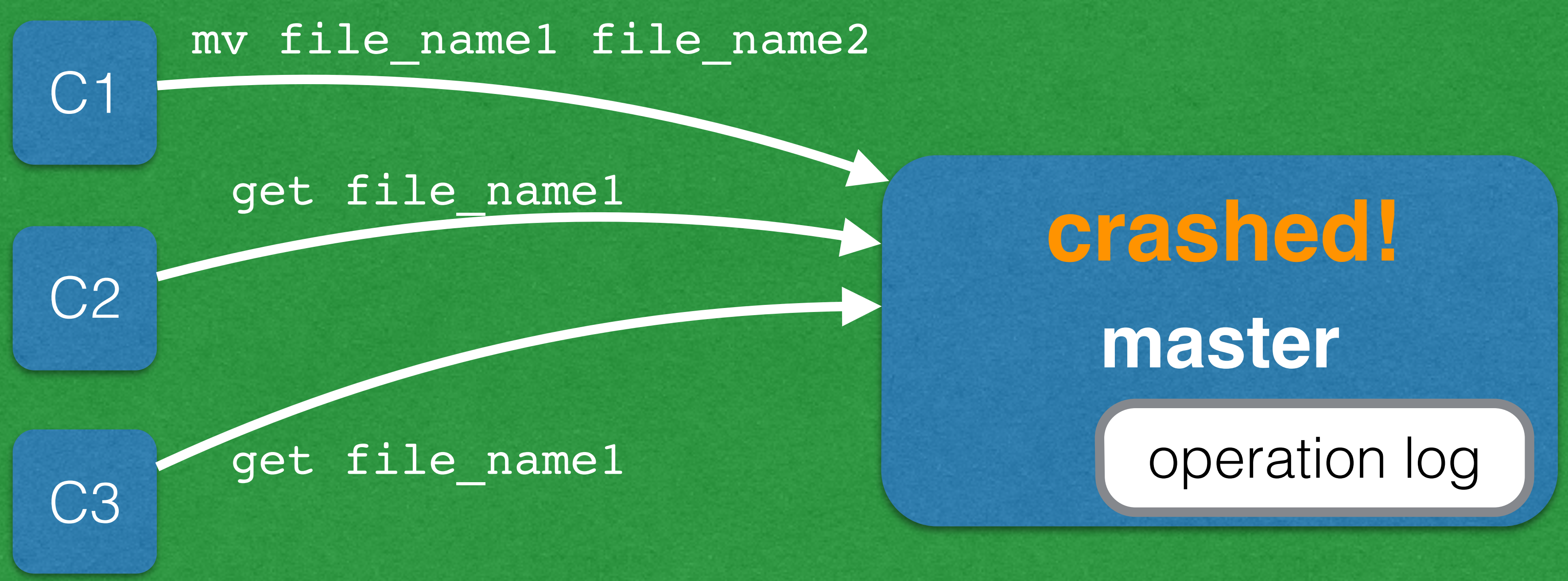
Question: how could the hotspot issue be solved?

Chunk locations

- Master does **not** keep a **persistent record** of chunk replica locations
- Master **polls** chunkservers about their chunks at **startup**
- Master keeps up to date through **periodic HeartBeat messages**
  - Master/chunkservers easily kept in sync when chunk servers leave/join/fail/restart [regular event]
  - Chunkserver has the final word over what chunks it has

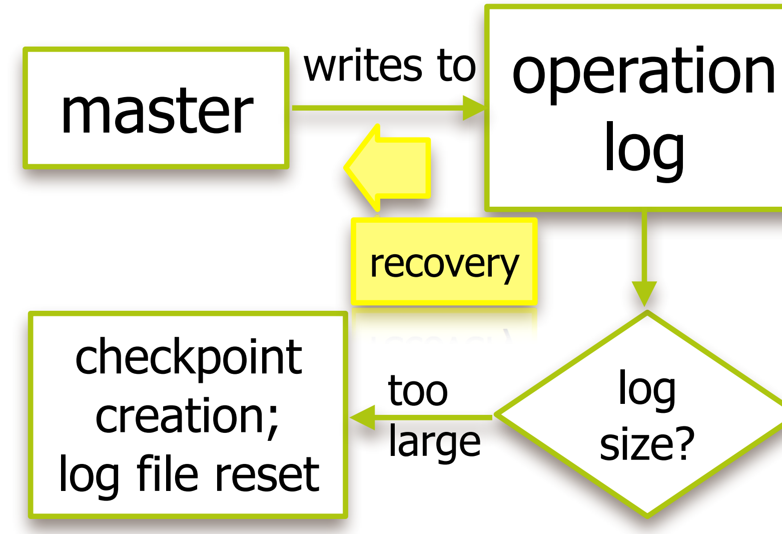
Operation log

- Persistent record of critical metadata changes
- Critical to the recovery of the system



Question: when does the master relay the new information to the clients? Before or after having written it to the op. log?

Operation log



- Persistent record of critical metadata changes
- Critical to the recovery of the system
- Changes to metadata are only made visible to clients **after** they have been written to the operation log
- Operation log **replicated** on multiple remote machines
  - **Before** responding to client operation, log record must have been **flashed locally and remotely**
- **M**aster recovers its file system from checkpoint + operation

Chunk replica placement

- **Creation** of (initially empty) chunks
    - **Use under-utilised** chunk servers; spread across racks
    - Limit number of recent creations on each chunk server
- **Re-replication**
    - Started once the available replicas fall below setting
    - Master instructs chunkserver to copy chunk data directly from existing valid replica
    - Number of active clone operations/bandwidth is limited
- **Re-balancing**
    - Changes in replica distribution for better load balancing; gradual filling of new chunk servers

GFS: Data integrity

Garbage collection

Question: how can a file be deleted from the cluster?

- **Deletion logged** by master
- **File renamed** to a hidden file, deletion timestamp kept
- **Periodic scan** of the master's file system namespace
  - Hidden files older than 3 days are deleted from master's memory (no further connection between file and its chunk)
- **Periodic scan** of the master's chunk namespace
  - Orphaned chunks (not reachable from any file) are identified, their metadata deleted
- **HeartBeat** messages used to **synchronise** deletion between master/chunkserver

Stale replica detection

Scenario: a chunkserver misses a change (“mutation”) applied to a chunk, e.g. a chunk was appended

- Master maintains a **chunk version number** to distinguish up-to-date and stale replicas
- Before an operation on a chunk, master ensures that version number is advanced
- Stale replicas are removed in the regular garbage collection cycle

Data corruption

- Data corruption or loss can occur at the read and write stage
- Chunkservers use **checksums** to detect corruption of stored data
  - Alternative: compare replicas across chunk servers
- Chunk is broken into **64KB blocks**, each has a 32 bit checksum
  - Kept in **memory** and stored persistently
- Read requests: **chunkserver verifies checksum** of data blocks that overlap read range (i.e. corruptions not send to clients)

HDFS: Hadoop Distributed File System

HDFS is still under active development (many changes between Hadoop 1.X/0.X and 2.X)

GFS vs. HDFS

GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
random file writes possible	**only append is possible**
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 64KB data and 32bit checksum pieces	per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp)
default block size: 64MB	default block size: 128MB

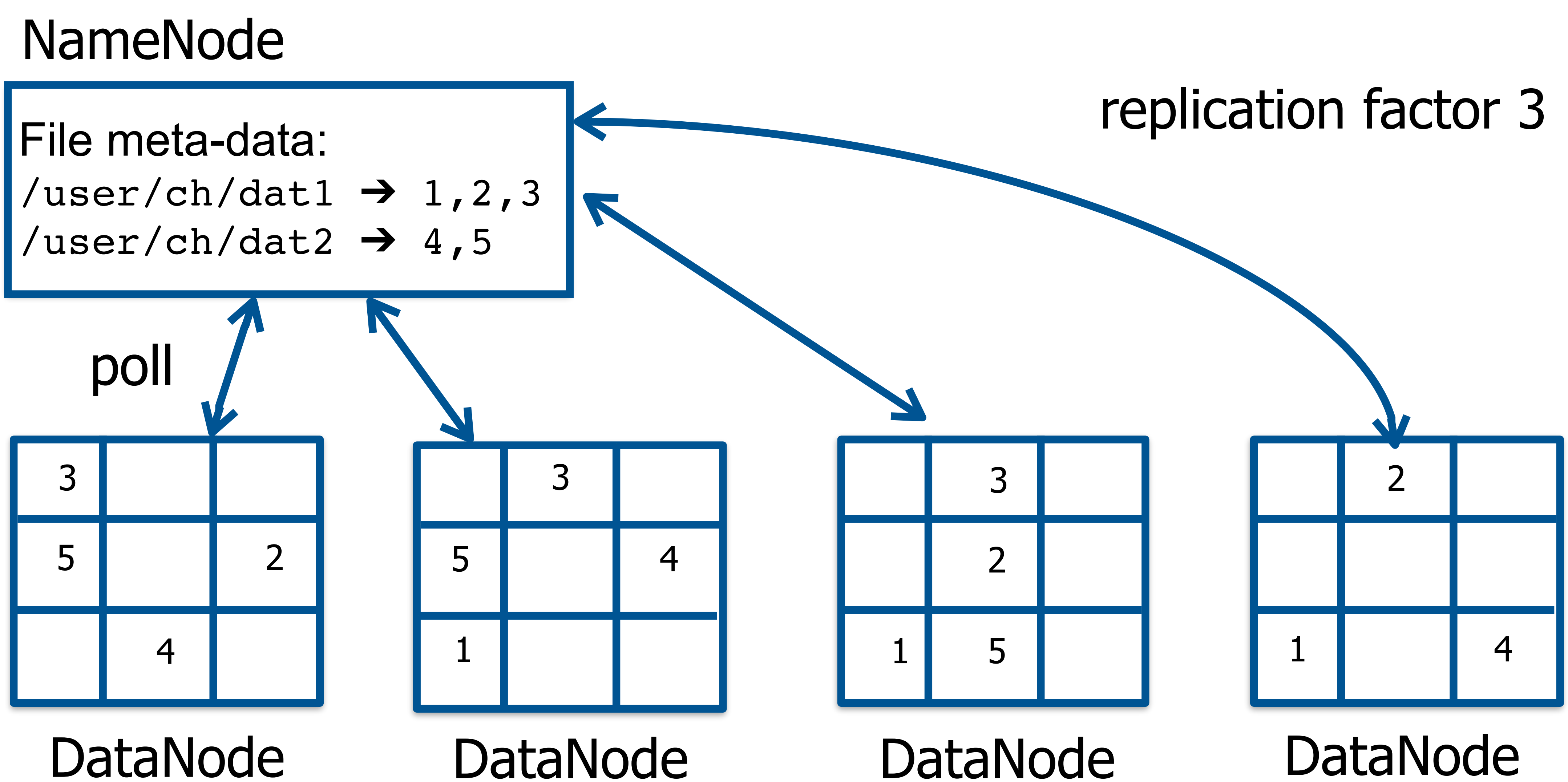
Hadoop's architecture O.X and 1.X

- **NameNode**
  - **Master** of HDFS, directs the slave DataNode daemons to perform low-level I/O tasks
  - Keeps track of file splitting into blocks, replication, block location, etc.

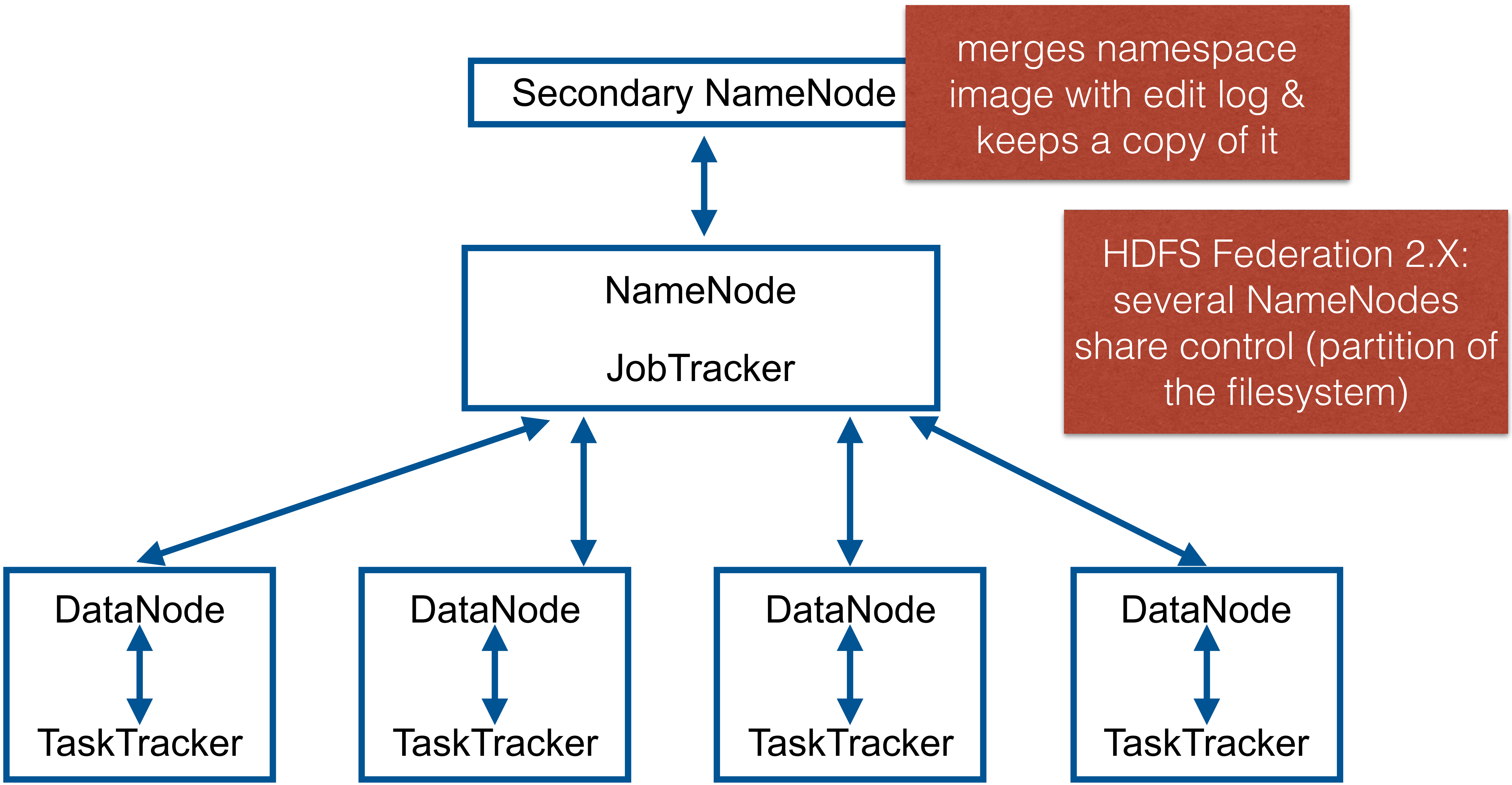
- **Secondary NameNode**: takes snapshots of the NameNode

- **DataNode**: each slave machine hosts a DataNode daemon

NameNodes and DataNodes



Hadoop cluster topology



Hadoop in practice: your system's health

```
[cloudera@localhost ~]$ hdfs fsck /user/cloudera -files -blocks
```

http://localhost.localdomain:50070

Hadoop in practice: Yahoo! (2010)

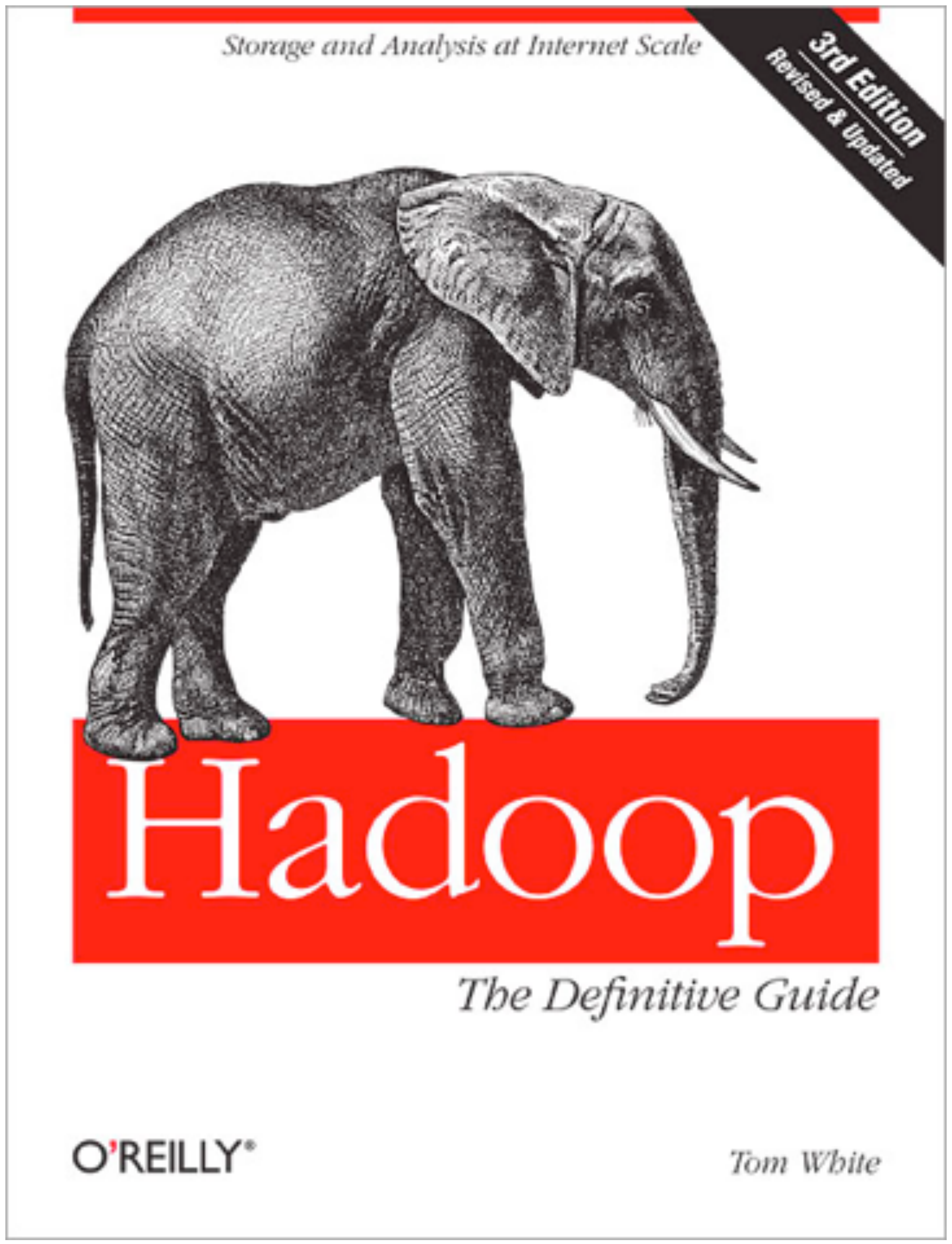
- **40 nodes/rack** sharing one IP switch
- **16GB RAM** per cluster node, 1-gigabit Ethernet
- **70% of disk space allocated to HDFS**
  - Remainder: operating system, data emitted by Mappers (not in HDFS)
- **NameNode**: up to **64GB RAM**
- **Total storage**: 9.8PB -> **3.3PB** net storage (replication: 3)
- **60 million files**, 63 million blocks
- 54,000 blocks hosted per DataNode
- **1-2 nodes lost per day**
- **Time for cluster to re-replicate lost blocks: 2 minutes**

HDFS cluster with 3500 nodes

Summary

- Ides behind the Google File System
- Basic procedures for replication, recovery, reading and writing of data
- Differences between GFS and HDFS

Recommended reading



Chapter 3.

A warning: coding takes time. More time than usual.
MapReduce is not difficult to understand, but different templates, different advice on different sites (of widely different quality).
Small errors are disastrous.

THE END