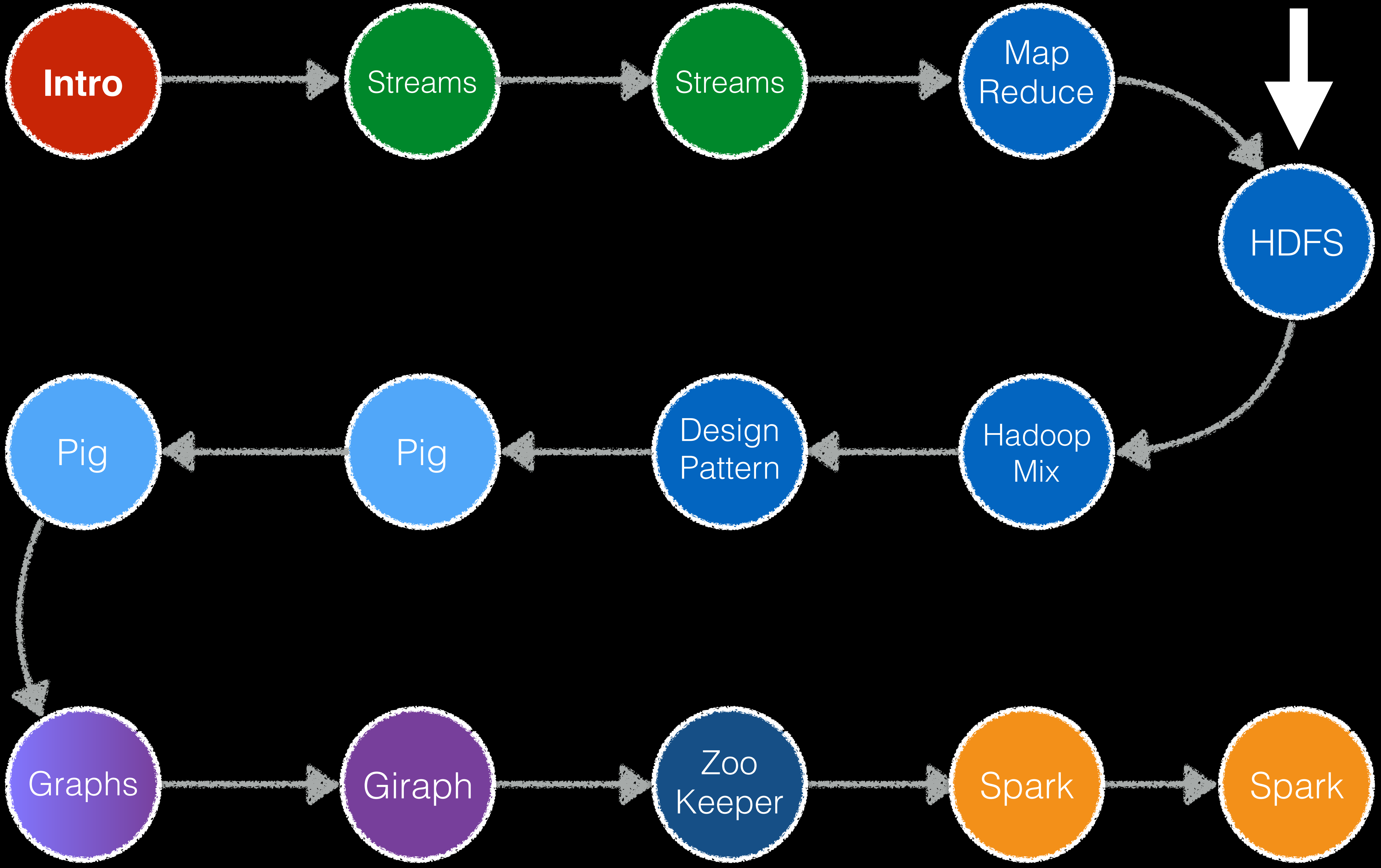
TI2736-B

Big Data Processing

Claudia Hauff

ti2736b-ewi@tudelft.nl



Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Mix
Design Pattern
Pig
Pig
Graphs
Giraph
Zoo Keeper
Spark
Spark

But first …

Partitioner & Combiner

Reminder: map & reduce

Key/value pairs form the basic data structure.

- Apply a map operation to **each record** in the input to compute a set of intermediate key/value pairs

$$map\!: \ (k_i, v_i) \rightarrow [(k_j, v_j)]$$
$$map\!: \ (k_i, v_i) \rightarrow [\boxed{(k_j, v_x),}(k_m, v_y), \boxed{(k_j, v_n),}\ldots]$$

There are **no limits** on the number of key/value pairs.

Combiner overview

- Combiner: **local aggregation** of key/value pairs after `map()` and before the shuffle & sort phase (occurs on the same machine as `map()`)
- Also called "**mini-reducer**"

Sometimes the reducer code can be used.

- Instead of emitting 100 times `(the,1)`, the combiner emits `(the,100)`
- Can lead to great **speed-ups**
- Needs to be **employed with care**

There is more: the combiner

Setup: a mapper which outputs `(term,termFreqInDoc)` and a combiner which is simply a copy of the reducer.

Task 1: **total term frequency** of a term in the corpus

without combiner

`(the,2),(the,2),(the,1)`

reduce

`(the,5)`

with combiner (reducer copy)

`(the,2),(the,3)`

reduce

`(the,5)`

correct!

Task 2: **average frequency** of a term in the documents

without combiner

`(the,2),(the,2),(the,1)`

reduce

`(the,(2+2+1)/3=1.66)`

with combiner (reducer copy)

`(the,2),(the,1.5)` (2+1)/2=1.5

reduce

`(the,(2+1.5)/2=1.75)`

wrong!

There is more: the combiner

- Each combiner **operates in isolation**, has no access to other mapper's key/value pairs
- A combiner **cannot** be assumed to process all values associated with the same key (may not run at all! Hadoop's decision)
- Emitted key/value pairs **must be the same** as those emitted by the mapper
- Most often, **combiner code != reducer code**
    - Exception: associative & commutative reduce operations

Hadoop in practice

Specified by the user:

- Mapper
- Reducer
- Combiner (optional)
- Partitioner (optional)
- Driver/job configuration

90% of the code comes from given templates

Hadoop in practice

Mapper: counting inlinks

input key/value: (`sourceUrl, content`)
output key/value: (`targetUrl, 1`)

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class InlinkCount extends Mapper<Object,Text,Text,IntWritable>
{
	IntWritable one = new IntWritable(1);
	Pattern linkPattern = Pattern.compile("\\[\\[.+?\\]\\]");

	public void map(Object key, Text value, Context con) throws Exception
	{
		String page = value.toString();
		Matcher m = linkPattern.matcher(page);
		while(m.find())
		{
			String match = m.group();
			con.write(new Text(match),one);
		}
	}
}
```

template differs slightly in diff. Hadoop versions

Hadoop in practice

Reducer: counting inlinks

input key/value: `(targetUrl, 1)`
output key/value: `(targetUrl, count)`

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class SumReducer extends Reducer<Text,IntWritable,Text,IntWritable>
{
      public void reduce(Text key,Iterable<IntWritable> values,Context con)
            throws Exception
      {
            int sum = 0;
            for(IntWritable iw : values)
                  sum += iw.get();

            con.write(key, new IntWritable(sum));
      }
}
```

template differs slightly in diff. Hadoop versions

Hadoop in practice

Driver: counting inlinks

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
…
public class InlinkCountDriver
{
      public static void main(String[] args) throws Exception
      {
            Configuration conf = new Configuration();
            String[] otherArgs = new GenericOptionsParser
                                         (conf,args).getRemainingArgs();
            Job job = new Job(conf, “InlinkAccumulator");
            job.setMapperClass(InlinkCountMapper.class);
            job.setCombinerClass(SumUpReducer.class);
            job.setReducerClass(SumUpReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

            FileInputFormat.addInputPath(job,new Path("/user/in/"));
            FileOutputFormat.setOutputPath(job,new Path("/user/out/"));
            job.waitForCompletion(true);
      }
}
```

Hadoop in practice

Partitioner: two URLs that are the same apart from their #fragment should be sent to the same reducer.

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
…
public class CustomPartitioner extends Partitioner
{
     public int getPartition(Object key, Object value,
                                                  int numPartitions)
     {
          String s = ((Text)key).toString();
          String newKey = s.substring(0,s.lastIndexOf('#'));

          return newKey.hashCode() % numPartitions;
     }

}
```

GFS / HDFS

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
- **Implement** strategies for handling small files

GFS introduction

Hadoop is *heavily* inspired by it.

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

History of MapReduce & GFS

- Developed by engineers at **Google** around **2003**
  - Built on principles in parallel and distributed processing
- **Seminal Google 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)

- **Yahoo! paper**:

  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** (Google) and **HDFS** (Hadoop) are distributed file systems
- **HDFS inspired by GFS**

GFS Assumptions

based on Google's main **use cases** (at the time)

- 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

Disclaimer

- GFS/HDFS are **not** a good fit for:
  - **Low latency data access** (in the ms range)
    - Solutions: HBase, Hive, …
  - **Many small files**
    - Solution: stuffing of binary files
  - **Constantly changing data**

- Not all details of GFS are public knowledge (HDFS developers “filled in” the details)

GFS architecture

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

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

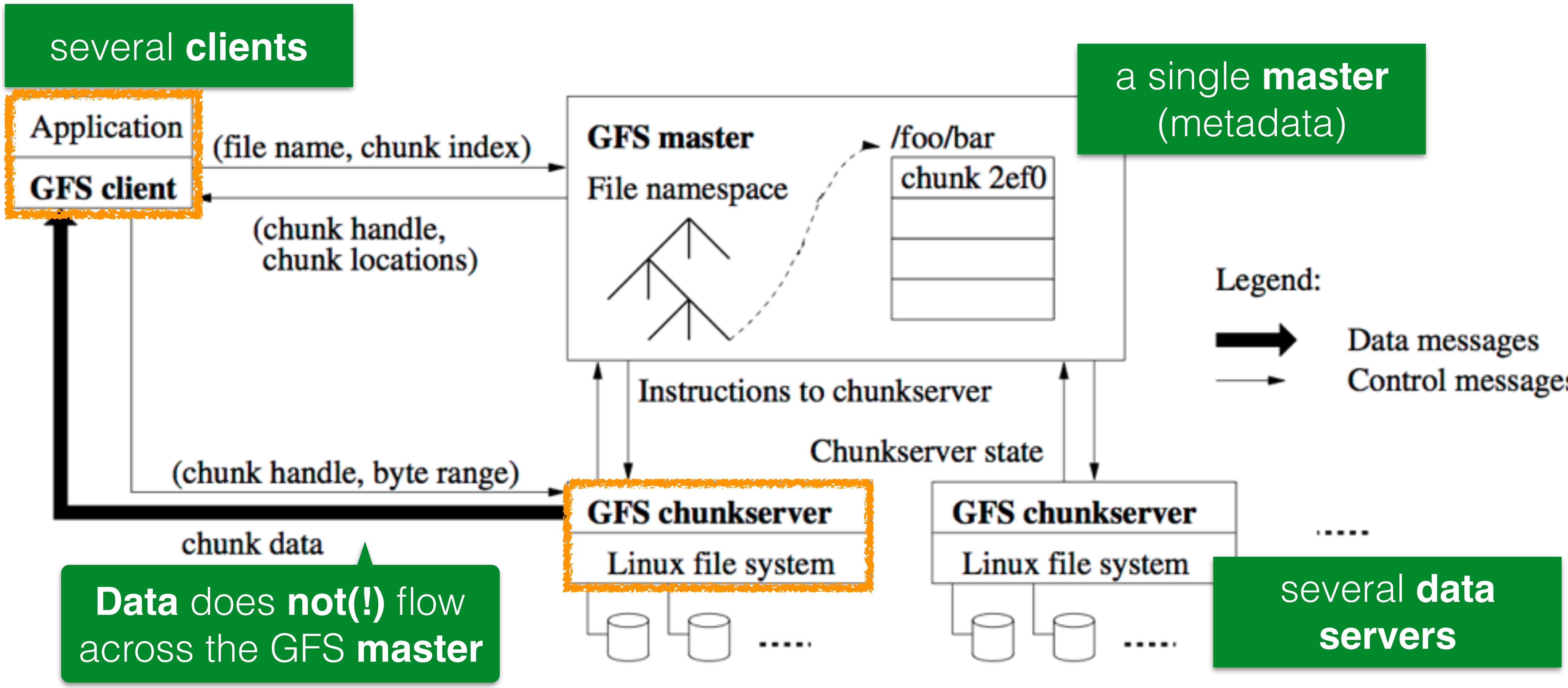


Image source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

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
  - 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)`

File information at Master level

- 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 (0.01s)
- 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**

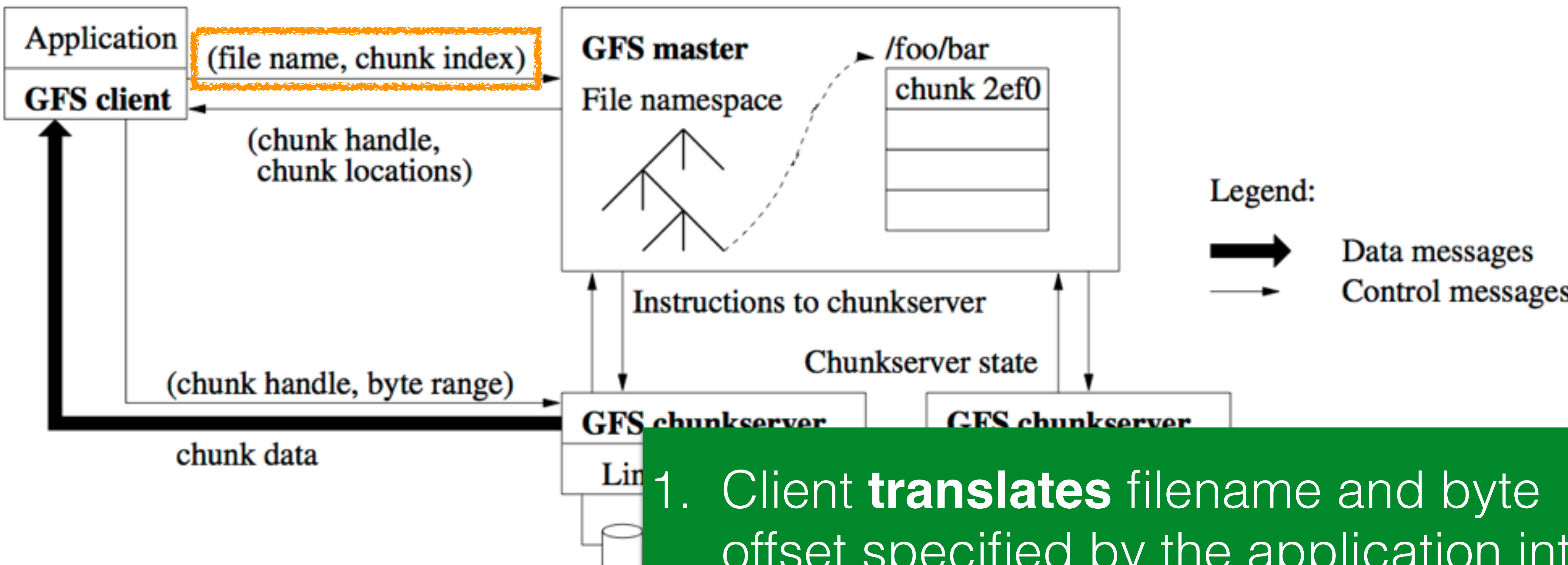
GFS: Master

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)

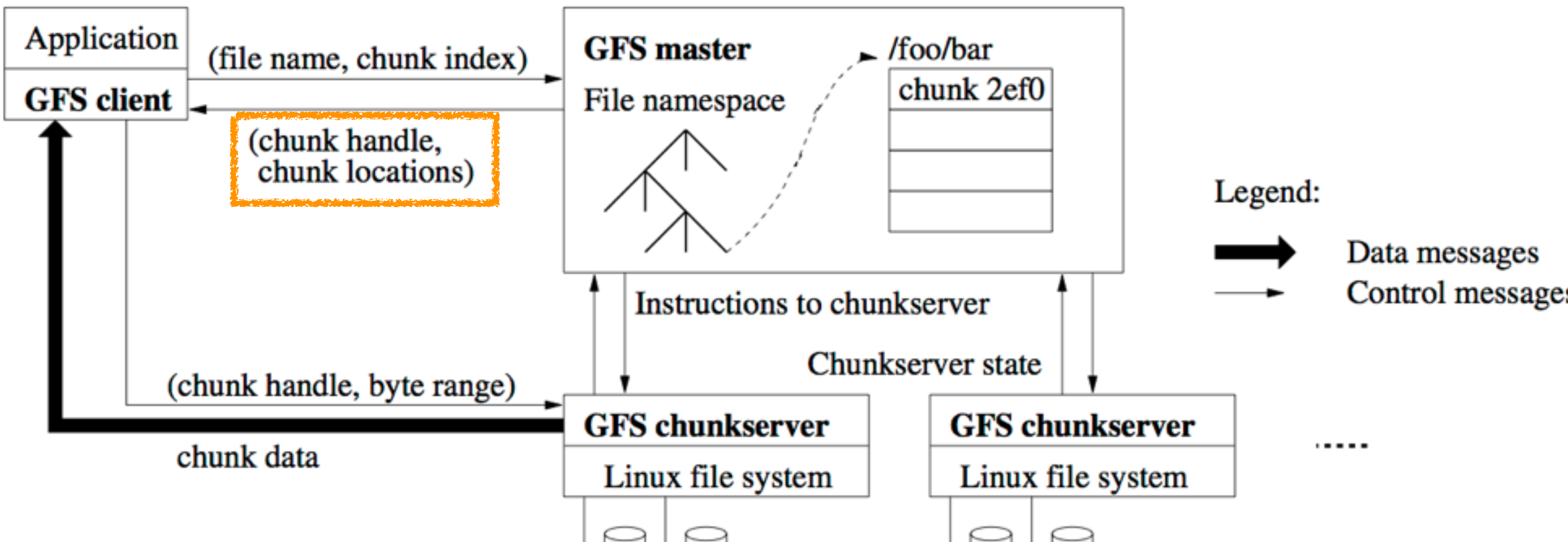
A read operation (in detail)



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

Image 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**.

Image 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.

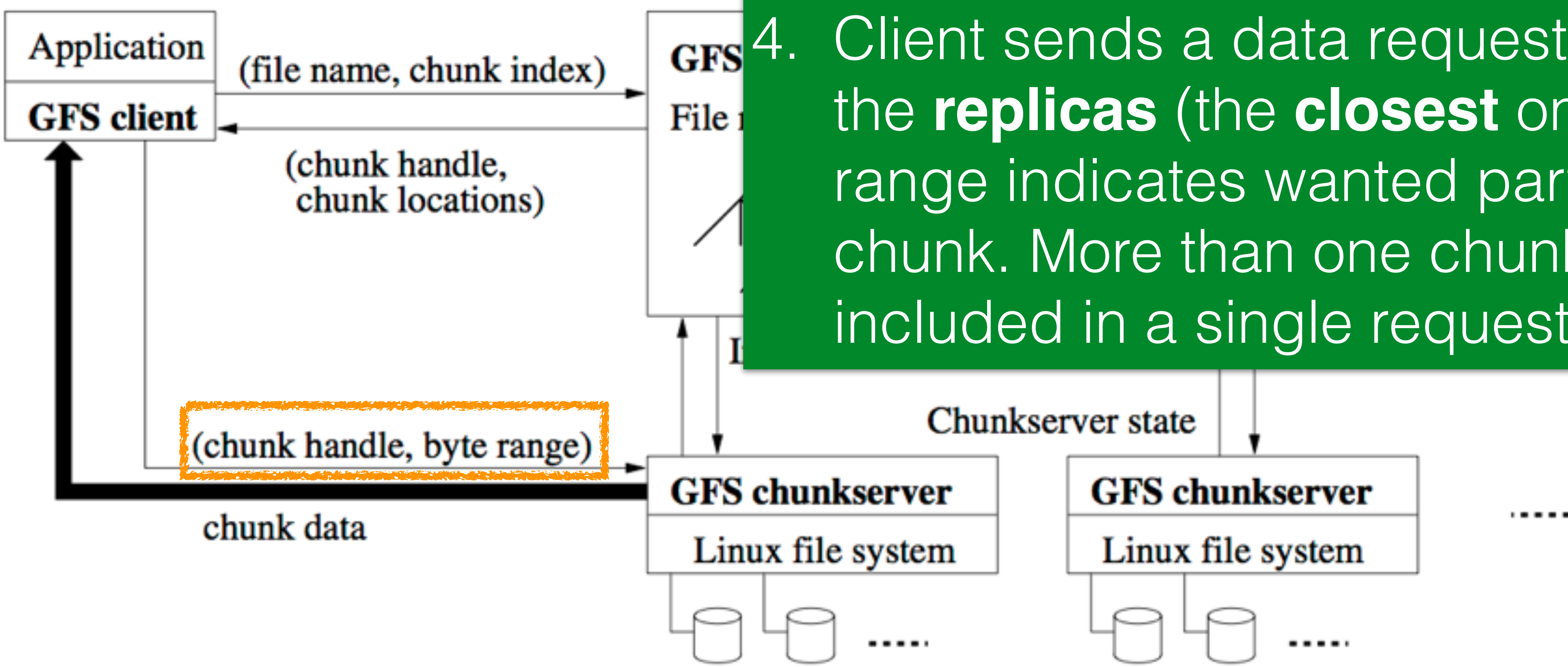
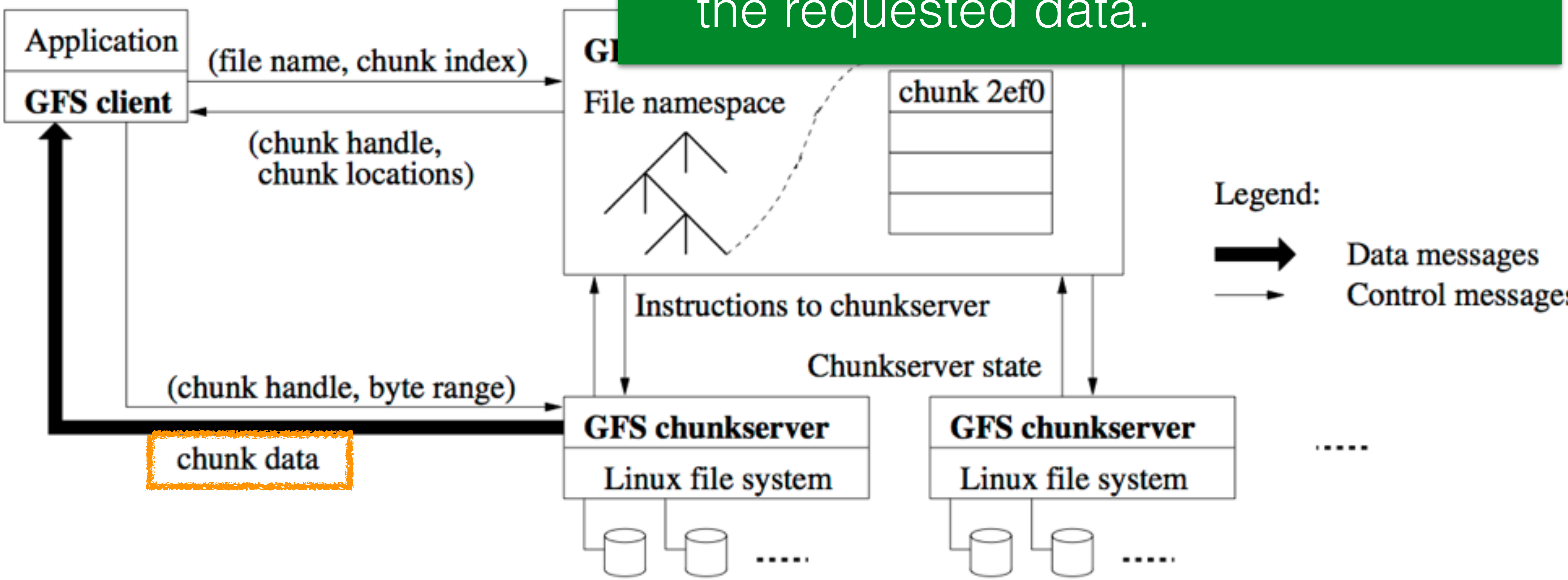


Image source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

A read operation (in detail)



5. Contacted chunk server replies with the requested data.

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

Metadata on the master

- **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

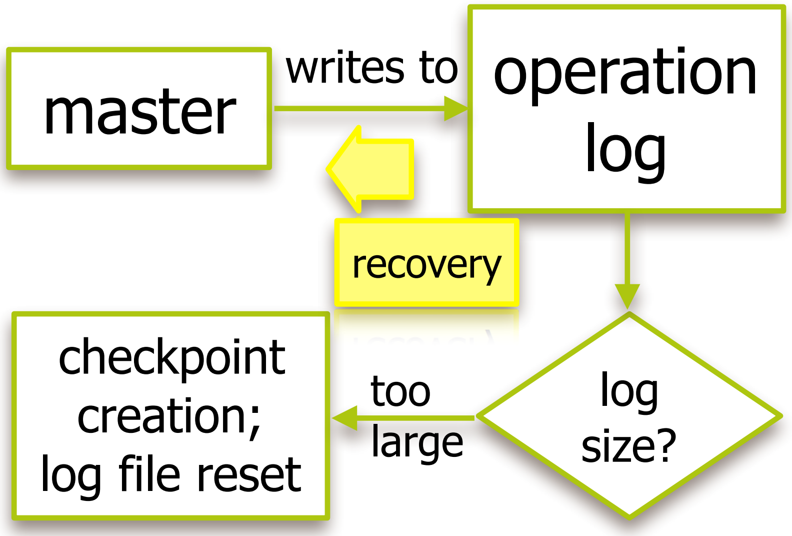
- 1 chunk = 64MB or 128MB (can be changed); chunk stored as a **plain Linux file** on a chunk server
- Advantages of large (but not too large) chunk size
  - **Reduced** need for client/master **interaction**
  - 1 request per chunk suits the target workloads
  - Client can **cache all the chunk locations** 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

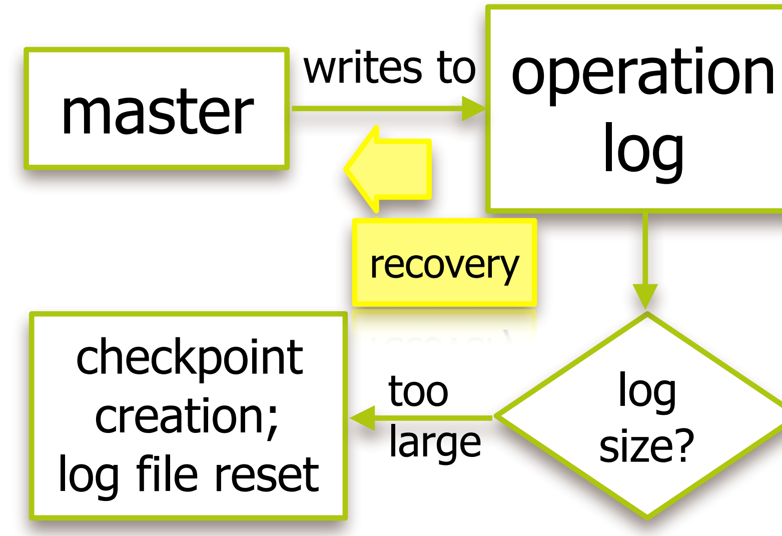


- Persistent record of critical metadata changes
- Critical to the recovery of the system
- Changes to metadata are only made visible to clients **after** they have been written to the operation log

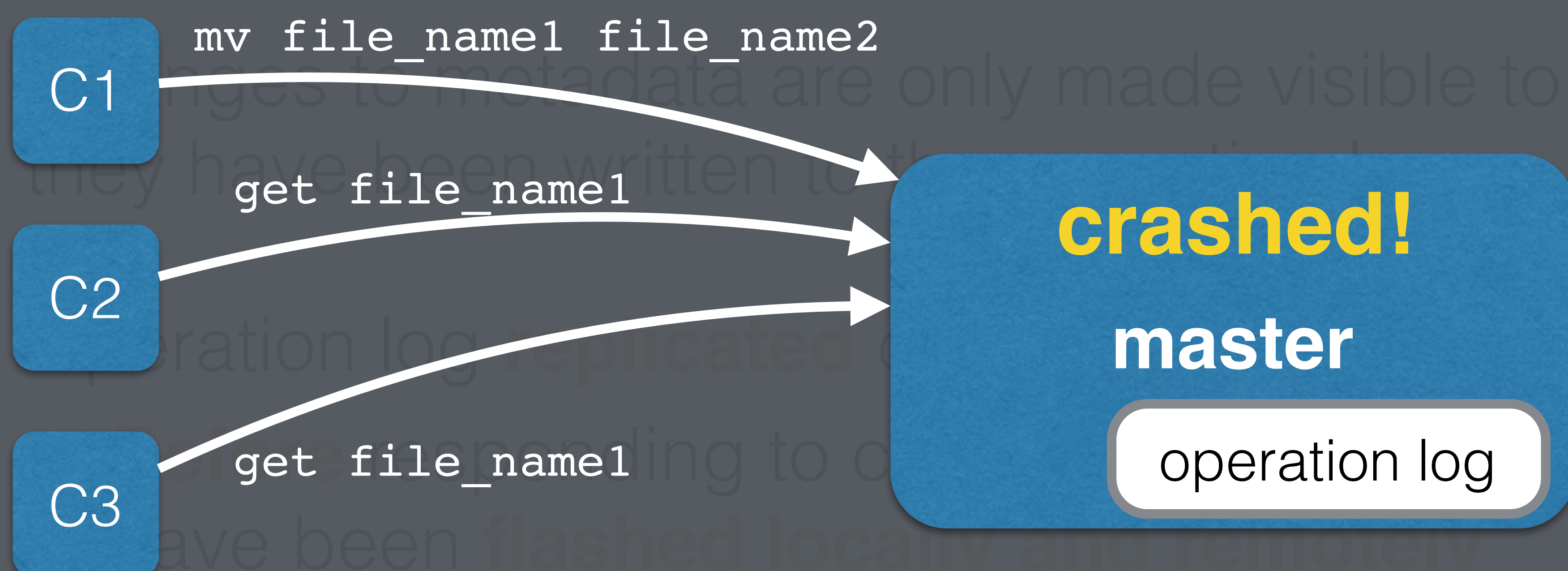
crashed!

- Operation log **replicated** on multiple remote machines
  - **Before** responding to client operation, log record must have been **flashed locally and remotely**
- Master recovers its file system from checkpoint + operation

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?

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

Question: how can chunk servers detect whether or not their stored data is corrupt?

- 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

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

“MapReduce 1”

Hadoop’s architecture 0.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

“MapReduce 1”

JobTracker and TaskTracker

- JobTracker (job scheduling + task progress monitoring)
  - **One** JobTracker **per Hadoop cluster**
  - **Middleman** between your application and Hadoop (single point of contact)
  - Determines the **execution plan** for the application (files to process, assignment of nodes to tasks, task monitoring)
  - Takes care of (supposed) **task failures**

- TaskTracker
  - **One** TaskTracker **per DataNode**
  - Manages individual tasks
  - **Keeps in touch** with the JobTracker (via HeartBeats) - sends progress report & signals empty task slots

JobTracker and TaskTracker

“MapReduce 1”

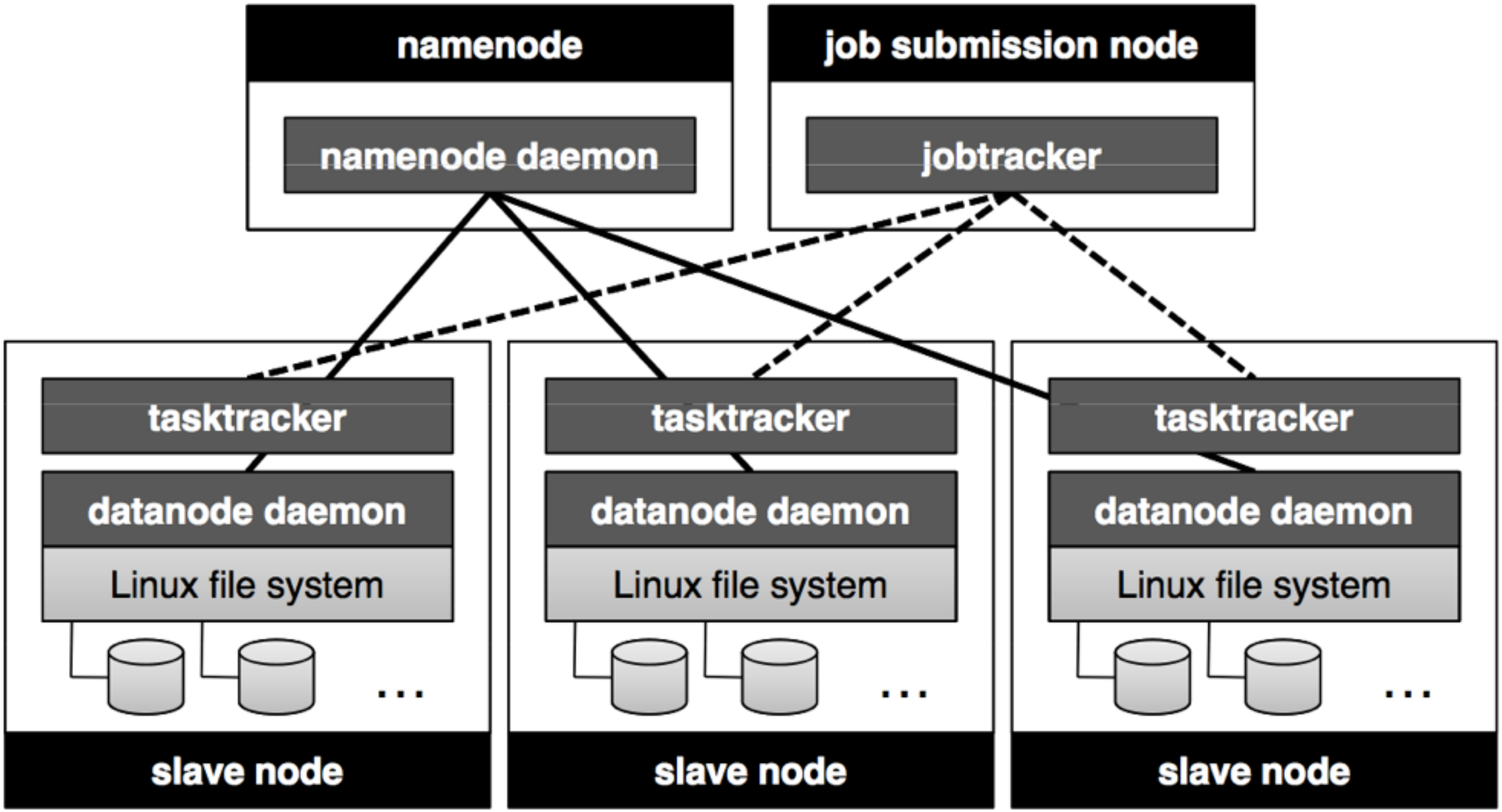


Image source: http://lintool.github.io/MapReduceAlgorithms/

"MapReduce 1"

What about the jobs?

- "**Hadoop job**": unit of work to be performed (by a client)
  - Input data
  - MapReduce program
  - Configuration information
- Hadoop **divides job into tasks** (two types: map, reduce)
- Hadoop divides input data into fixed size input splits
  - **One map task per split**
  - One map function call for each record in the split
  - Splits are processed in **parallel** (if enough DataNodes exist)
  - Job execution controlled by JobTracker and TaskTrackers

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 3,500 nodes

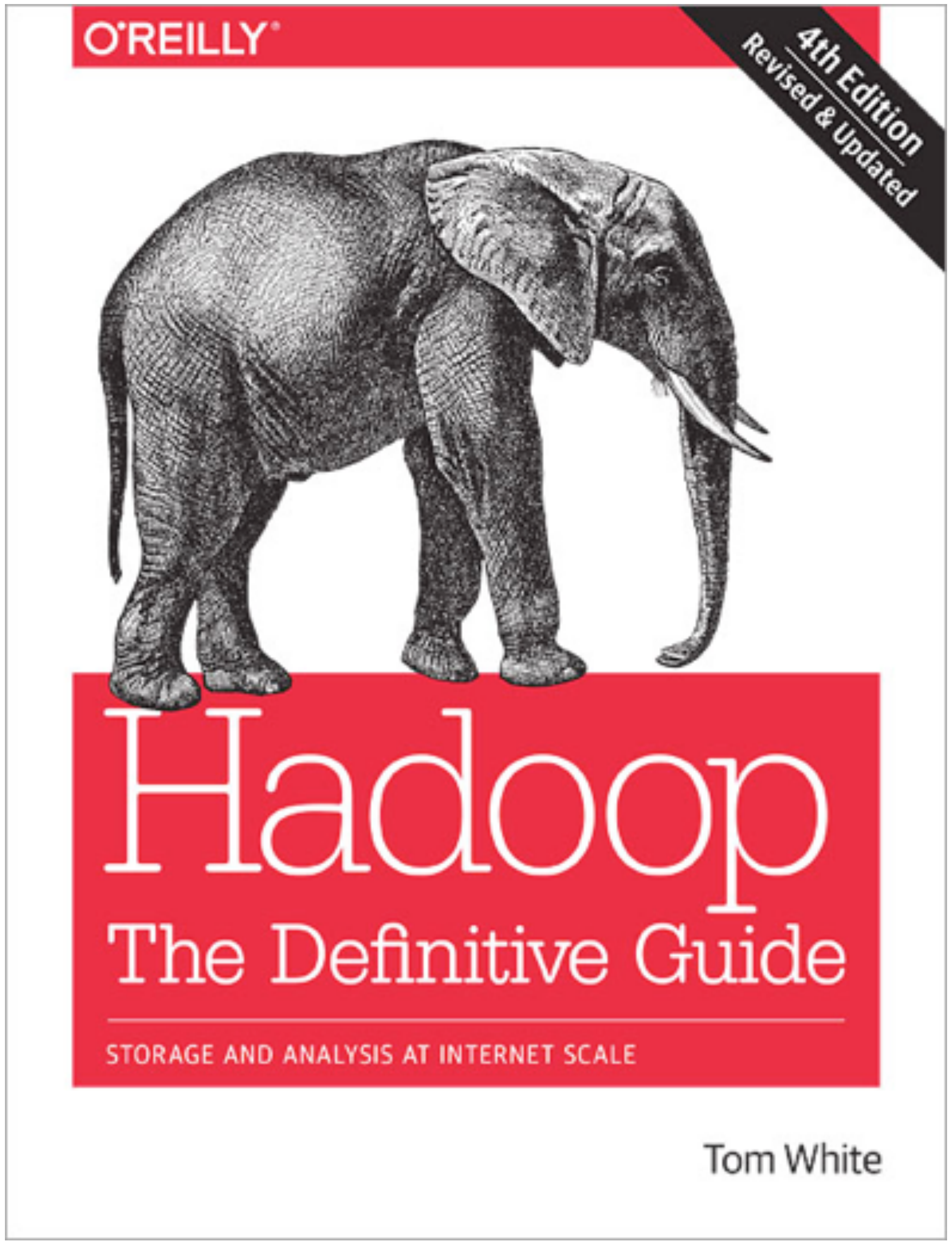
YARN (MapReduce 2)

- JobTracker/TaskTrackers setup becomes a **bottleneck** in clusters with thousands of nodes
- As answer YARN has been developed (**Y**et **A**nother **R**esource **N**egotiator)
- YARN splits the JobTracker's tasks (job scheduling and task progress monitoring) into two daemons:
  - **Resource manager** (RM)
  - **Application master** (negotiates with RM for cluster resources; each Hadoop job has a dedicated master)

YARN Advantages

- **Scalability:** larger clusters are supported
- **Availability:** high availability (high uptime) supported
- **Utilization:** more fine-grained use of resources
- **Multitenancy:** MapRedue is just one application among many

Recommended reading



Chapter 3

THE END