MapReduce

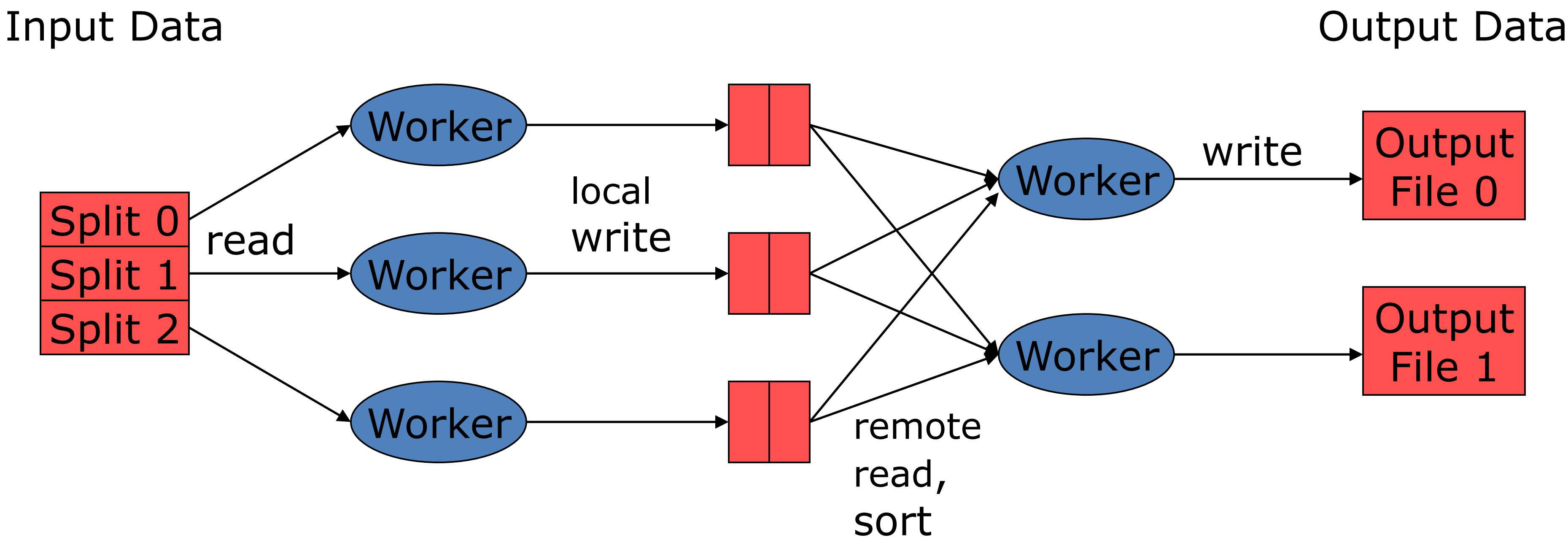
MapReduce

- MapReduce [OSDI'04] provides
  - Automatic parallelization, distribution
  - I/O scheduling
    - Load balancing
    - Network and data transfer optimization
  - Fault tolerance
    - Handling of machine failures

- **Need more power: Scale out, not up!**
    - Large number of **commodity servers** as opposed to some high end specialized servers

Apache Hadoop:
Open source implementation of MapReduce

MapReduce workflow



Map

extract something you care about from each record

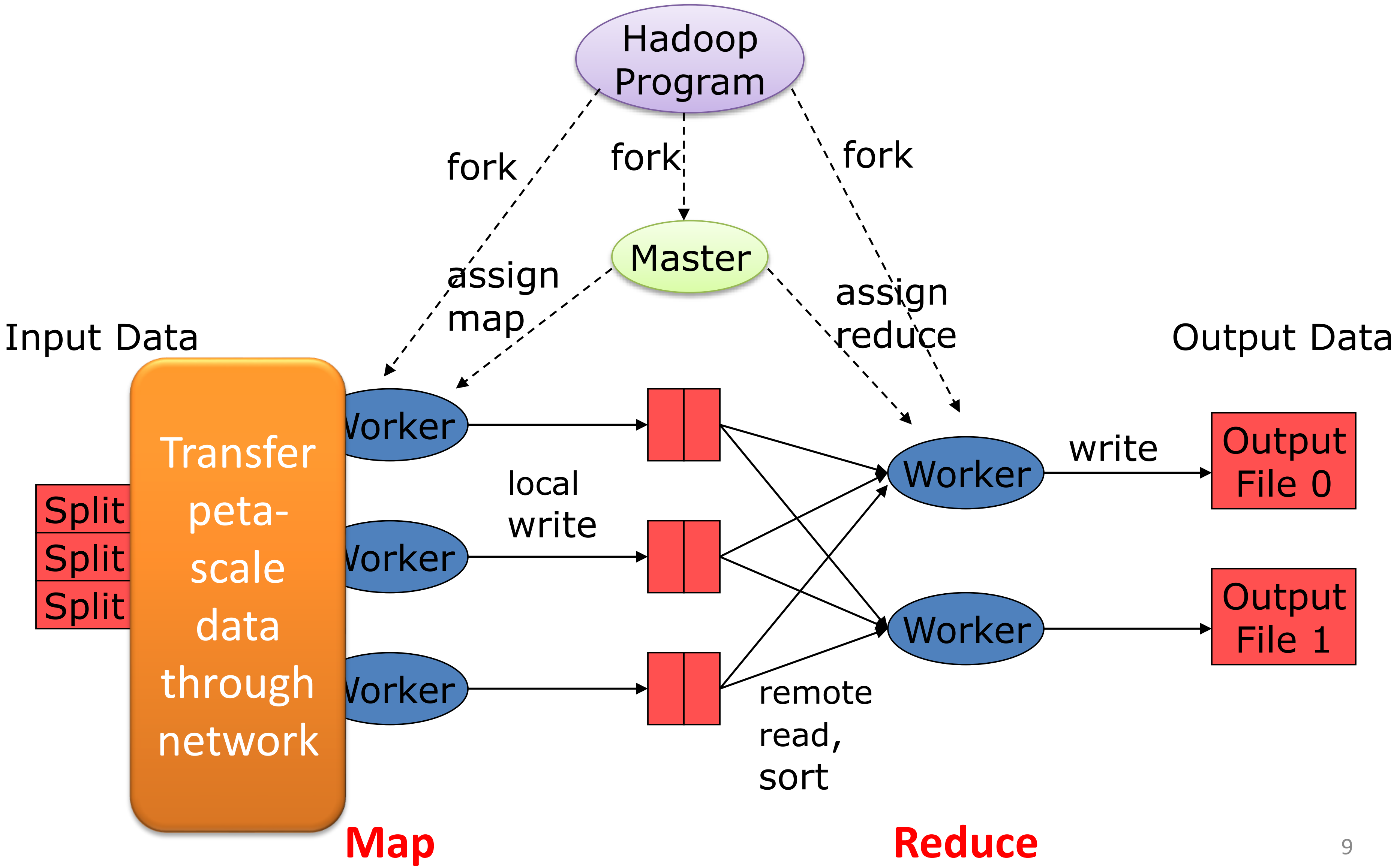
Reduce

aggregate, summarize, filter, or transform

Example: Word Count

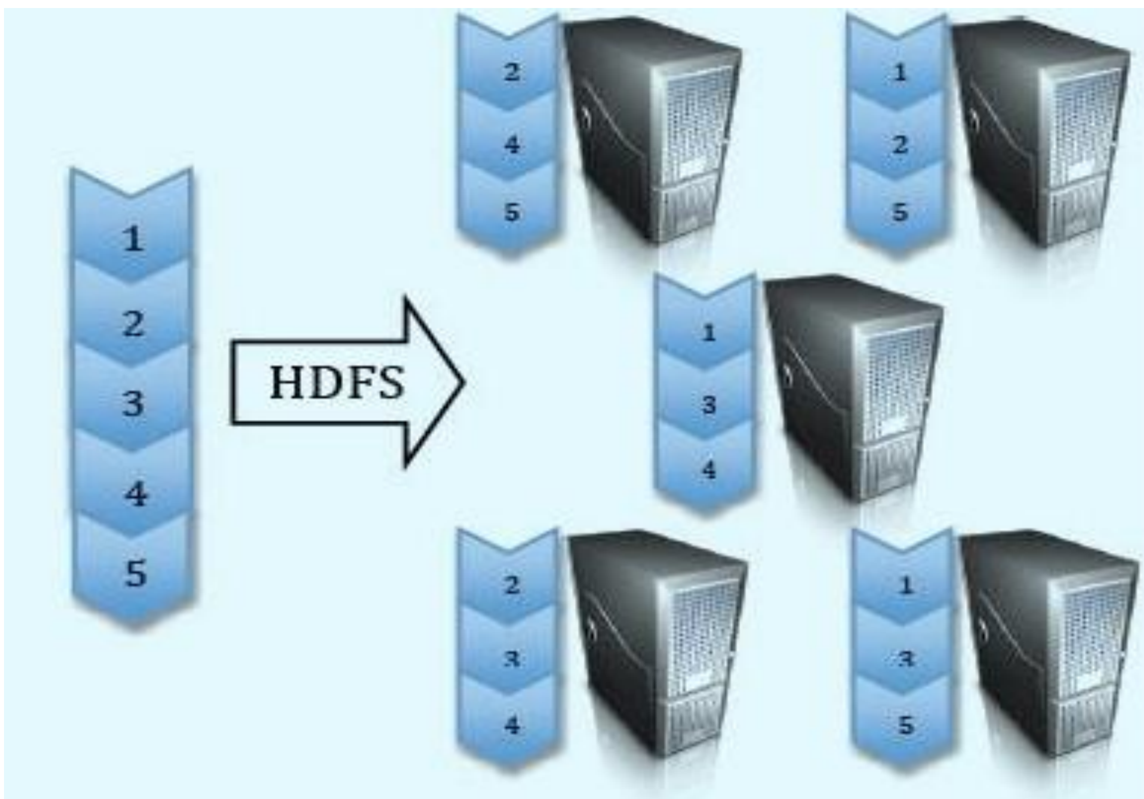
http://kickstarthadoop.blogspot.ca/2011/04/word-count-hadoop-map-reduce-example.html

MapReduce

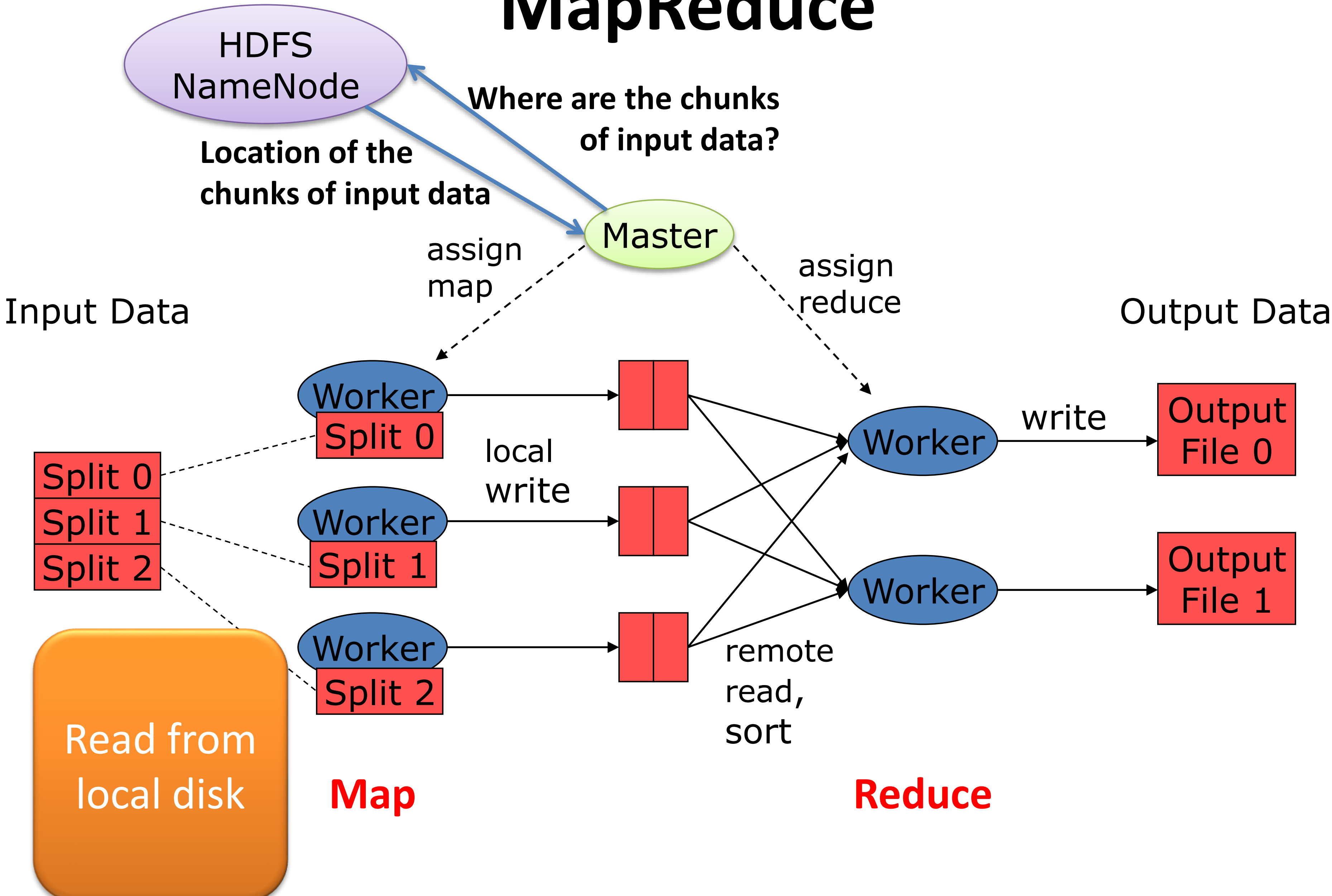

Hadoop Program
fork
fork
fork
Master
assign map
assign reduce
Input Data
Output Data
Split
Split
Split
Transfer peta-scale data through network
Worker
Worker
Worker
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Map
Reduce

Google File System (GFS)
Hadoop Distributed File System (HDFS)

- Split data and store 3 replica on commodity servers



MapReduce



HDFS NameNode
Where are the chunks of input data?
Location of the chunks of input data
Master
assign map
assign reduce
Input Data
Output Data
Split 0
Split 1
Split 2
Worker
Split 0
Worker
Split 1
Worker
Split 2
local write
remote read, sort
Worker
Worker
write
Output File 0
Output File 1
Read from local disk
Map
Reduce

Failure in MapReduce

- Failures are norm in commodity hardware
- **Worker** failure
  - Detect failure via periodic heartbeats
  - Re-execute in-progress map/reduce tasks
- **Master** failure
  - Single point of failure; Resume from Execution Log
- **Robust**
  - Google's experience: lost 1600 of 1800 machines once!, but finished fine

```java
public class WordCount {
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
 }}}
```

Mapper

```java
 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
 }}
```

Reducer

```java
 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}}
```

Run this program as a MapReduce job

Contents

- **Motivation**
- Design overview
  - Write Example
  - Record Append
- Fault Tolerance & Replica Management
- Conclusions

Motivation: Large Scale Data Storage

- Manipulate large (Peta Scale) sets of data
- Large number of machine with commodity hardware
- Component failure is the norm

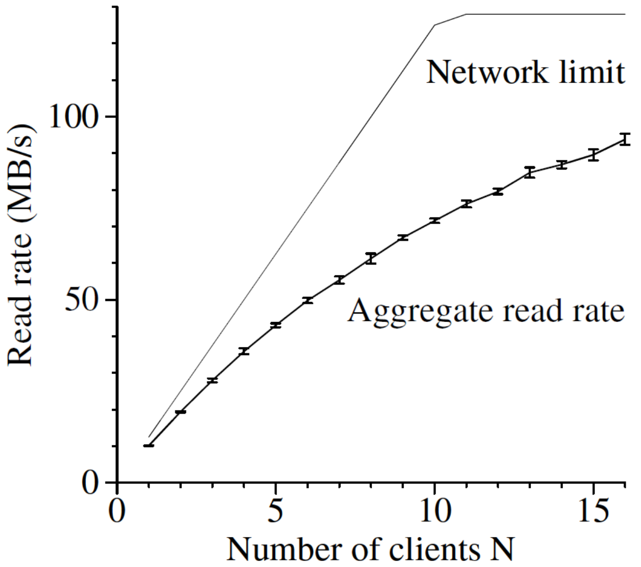
- Goal: **Scalable**, **high performance**, **fault tolerant** distributed file system

Why a new file system?

- None designed for their failure model
- Few scale as highly or dynamically and easily
- Lack of special primitives for large distributed computation

What should expect from GFS

- Designed for Google's application
  - Control of both file system and application
  - Applications use a few specific access patterns
    - Append to larges files
    - Large streaming reads
  - Not a good fit for
    - low-latency data access
    - lots of small files, multiple writers, arbitrary file modifications



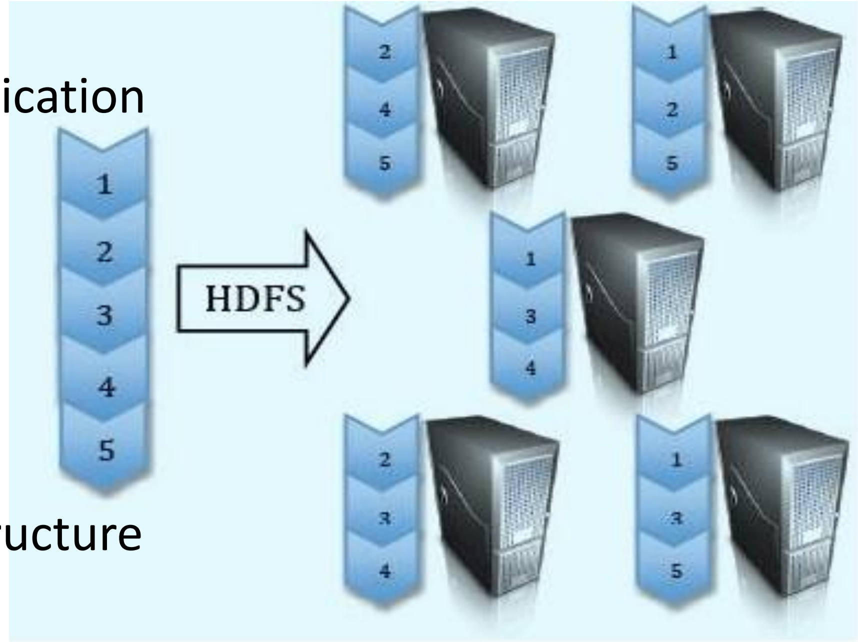
- Not POSIX, although mostly traditional
  - Specific operations: RecordAppend

Contents

- Motivation
- **Design overview**
  - Write Example
  - Record Append
- Fault Tolerance & Replica Management
- Conclusions

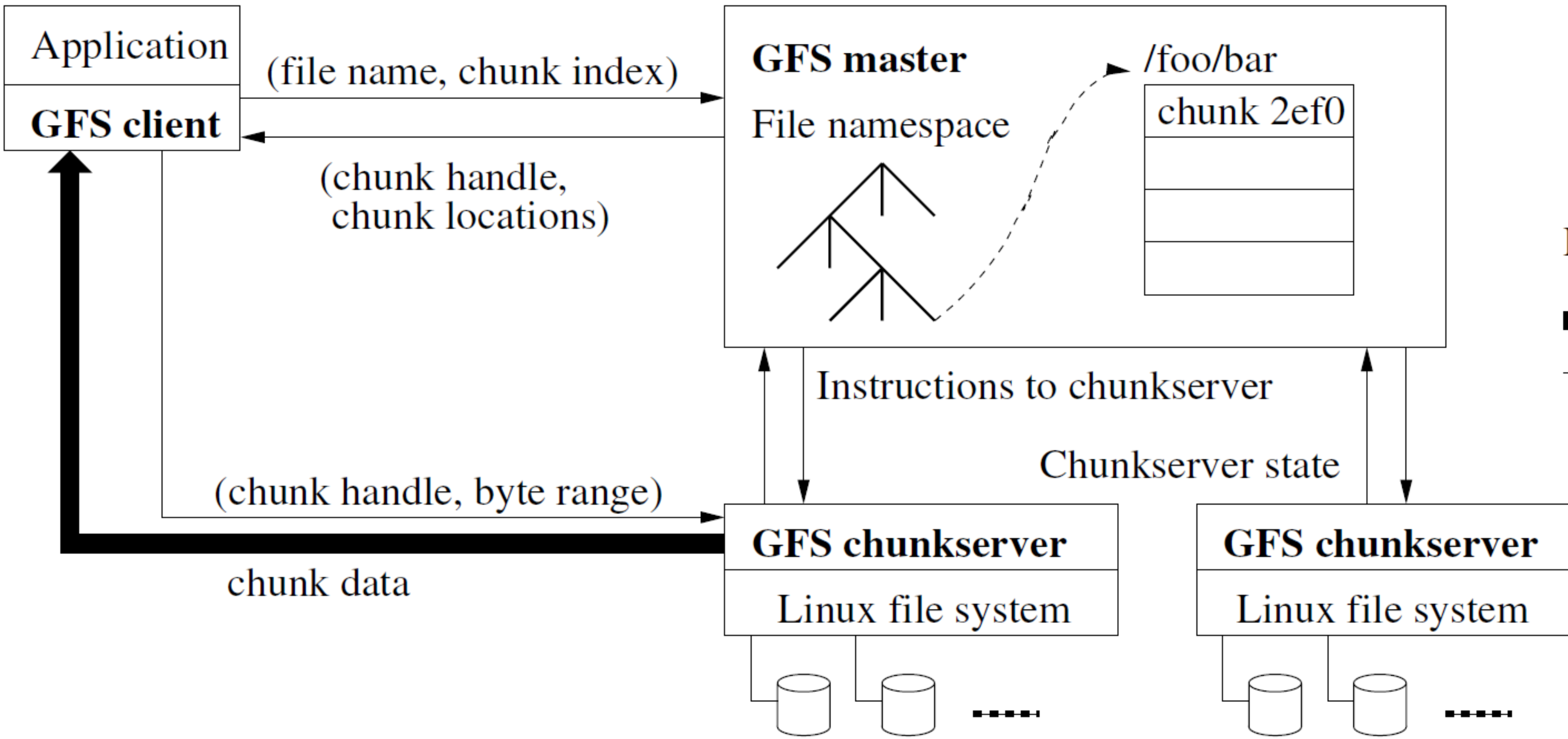
Components

- **Master (NameNode)**
  - Manages metadata (namespace)
  - Not involved in data transfer
  - Controls allocation, placement, replication
- **Chunkserver (DataNode)**
  - Stores chunks of data
  - No knowledge of GFS file system structure
  - Built on local linux file system

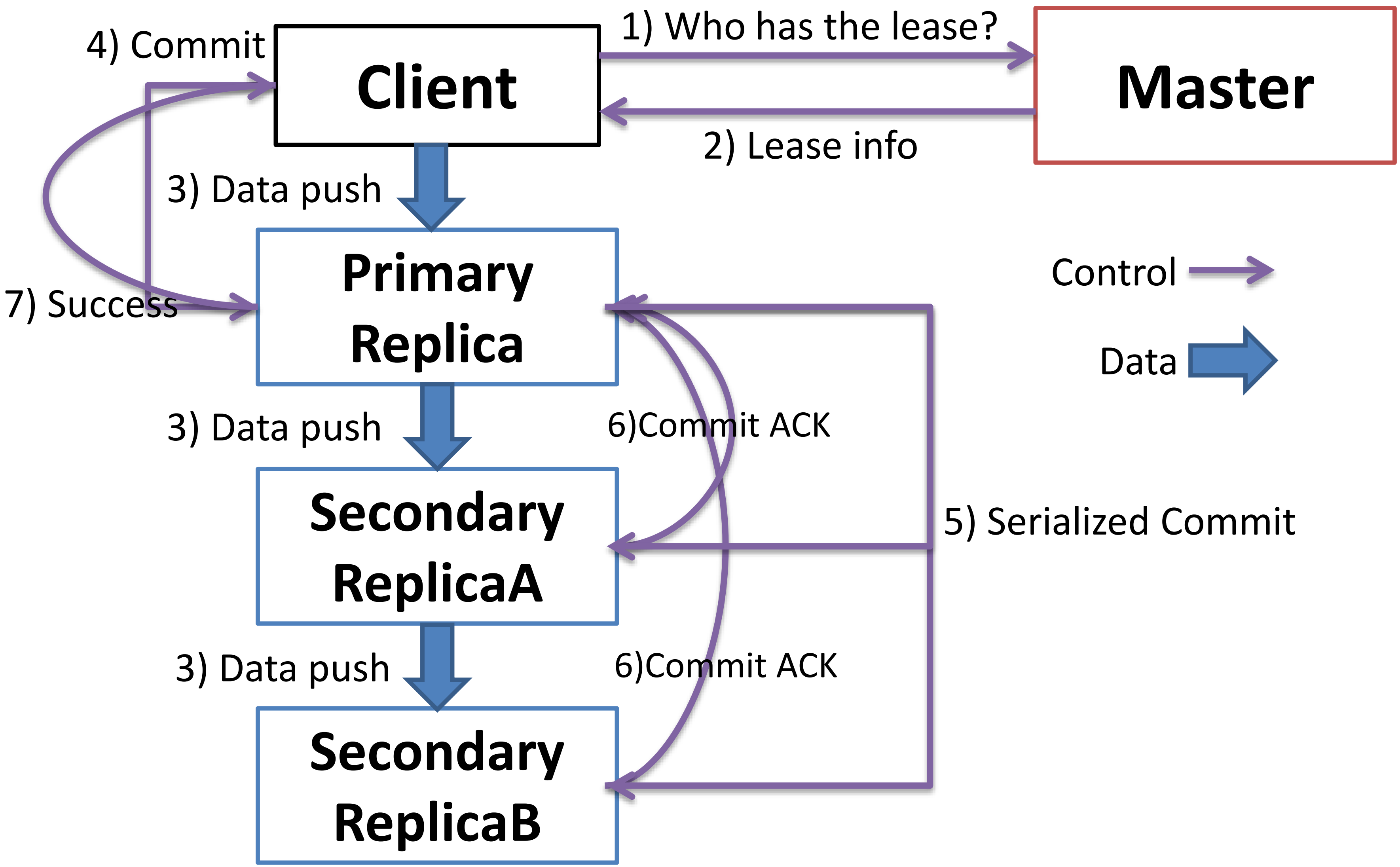


www.cse.buffalo.edu/~okennedy/courses/cse704fa2012/2.2-HDFS.pptx

GFS Architecture



Write(filename, offset, data)



RecordAppend(filename, data)

- Significant use in distributed apps. For example at Google production cluster:
  - 21% of bytes written
  - 28% of write operations
- Guaranteed: All data appended at least once as a single consecutive byte range
- Same basic structure as write
  - Client obtains information from master
  - Client sends data to data nodes (chunkservers)
  - Client sends “append-commit”
  - Lease holder serializes append
- **Advantage:** Large number of concurrent writers with minimal coordination

RecordAppend (2)

- Record size is limited by chunk size
- When a record does not fit into available space,
  - chunk is padded to end
  - and client retries request.

Contents

- Motivation
- Design overview
  - Write Example
  - Record Append
- **Fault Tolerance & Replica Management**
- Conclusions

Fault tolerance

- Replication
  - High availability for reads
  - User controllable, default 3 (non-RAID)
  - Provides read/seek bandwidth
  - Master is responsible for directing re-replication if a data node dies
- Online checksumming in data nodes
  - Verified on reads

Replica Management

- Bias towards topological spreading
  - Rack, data center
- Rebalancing
  - Move chunks around to balance disk fullness
  - Gently fixes imbalances due to:
    - Adding/removing data nodes

Replica Management (Cloning)

- Chunk replica lost or corrupt
- Goal: minimize app disruption and data loss
  - Approximately in priority order
    - More replica missing-> priority boost
    - Deleted file-> priority decrease
    - Client blocking on a write-> large priority boost
  - Master directs copying of data
- Performance on a production cluster
  - Single failure, full recovery (600GB): 23.2 min
  - Double failure, restored 2x replication: 2min

Garbage Collection

- Master does **not** need to have a strong knowledge of what is stored on each data node
  - Master regularly scans namespace
  - After GC interval, deleted files are removed from the namespace
  - Data node periodically polls Master about each chunk it knows of.
  - If a chunk is forgotten, the master tells data node to delete it.

Limitations

- Master is a central point of failure
- Master can be a scalability bottleneck
- Latency when opening/stating thousands of files
- Security model is weak

Conclusion

- Inexpensive commodity components can be the basis of a large scale reliable system
- Adjusting the API, e.g. RecordAppend, can enable large distributed apps
- Fault tolerant
- Useful for many similar apps