Distributed Computation Models

SWE 622, Spring 2017
Distributed Software Engineering

Some slides ack: Jeff Dean

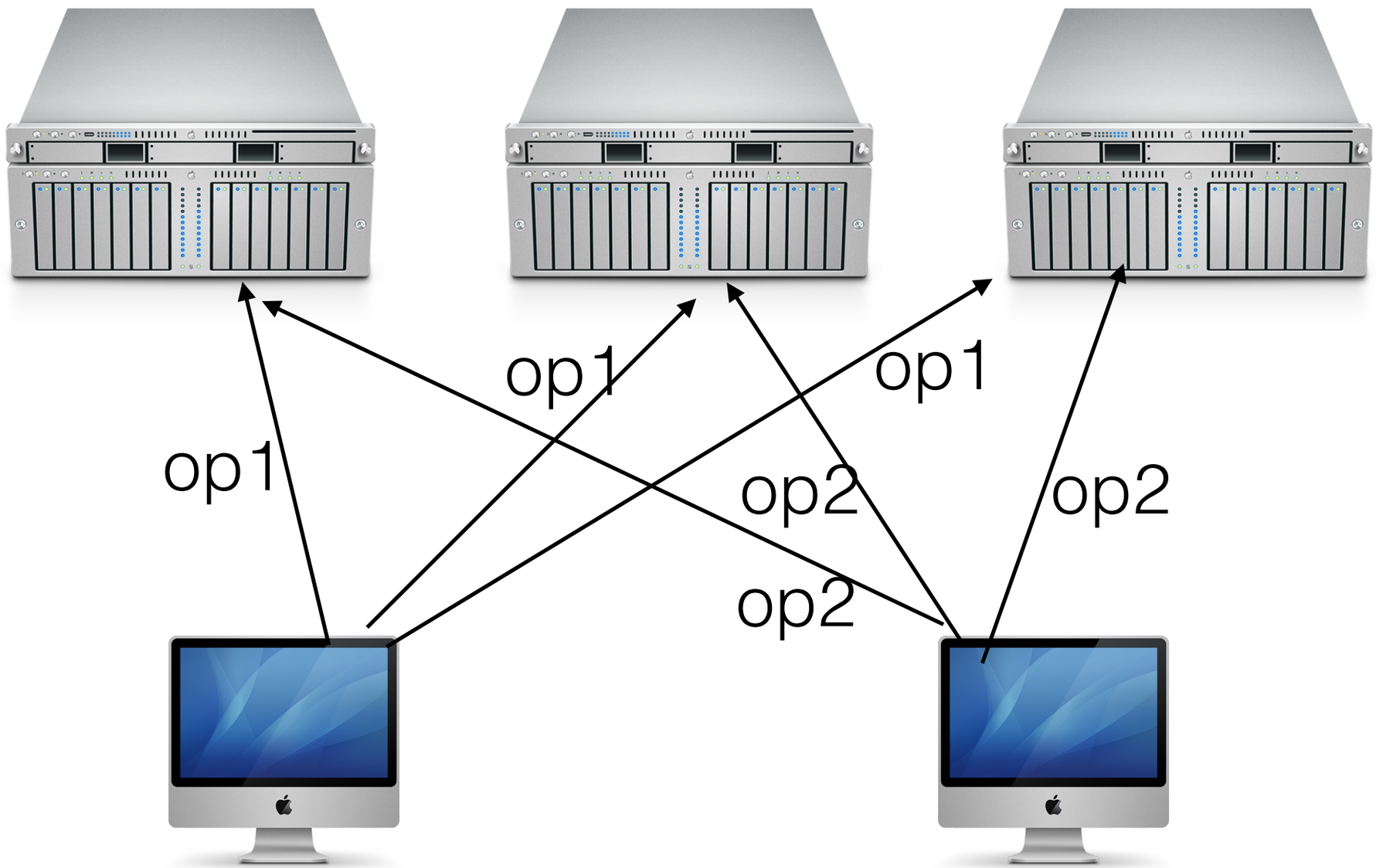
HW4 Recap

- https://b.socrative.com/
- Class: SWE622

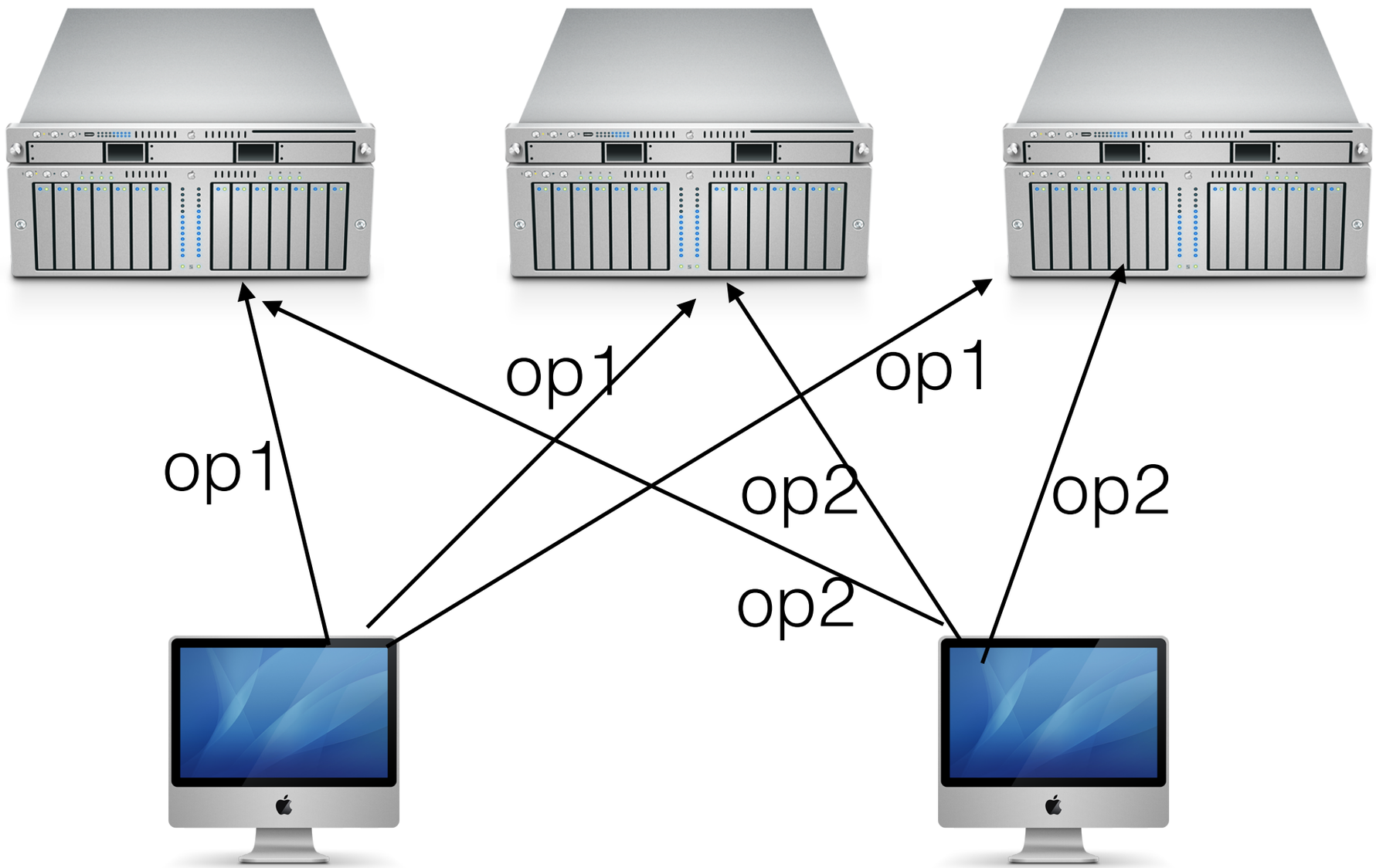
Review

- Replicating state machines
- Case studies - Hypervisor, GFS, MagicPocket

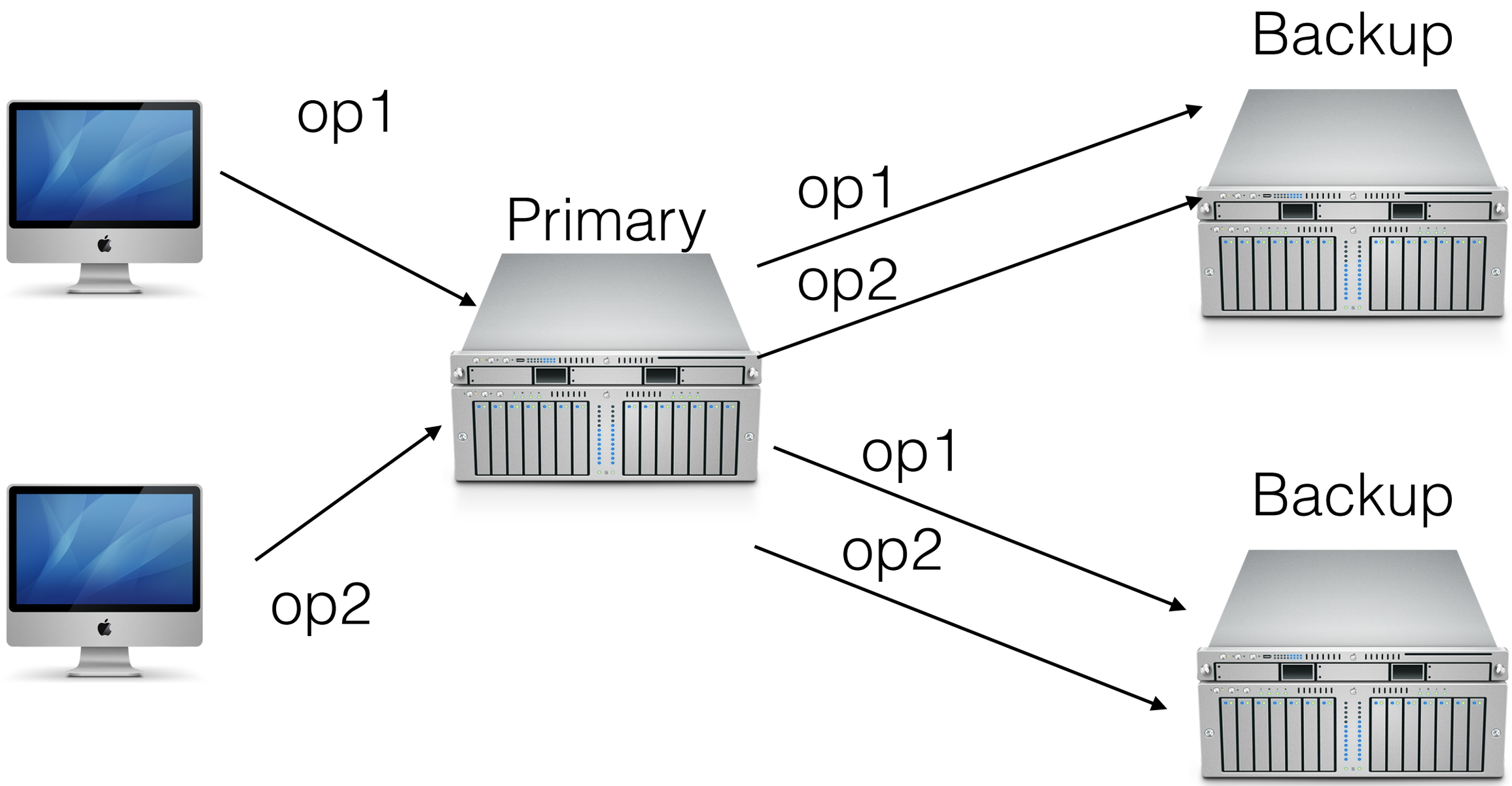
Strawman RSM implementation



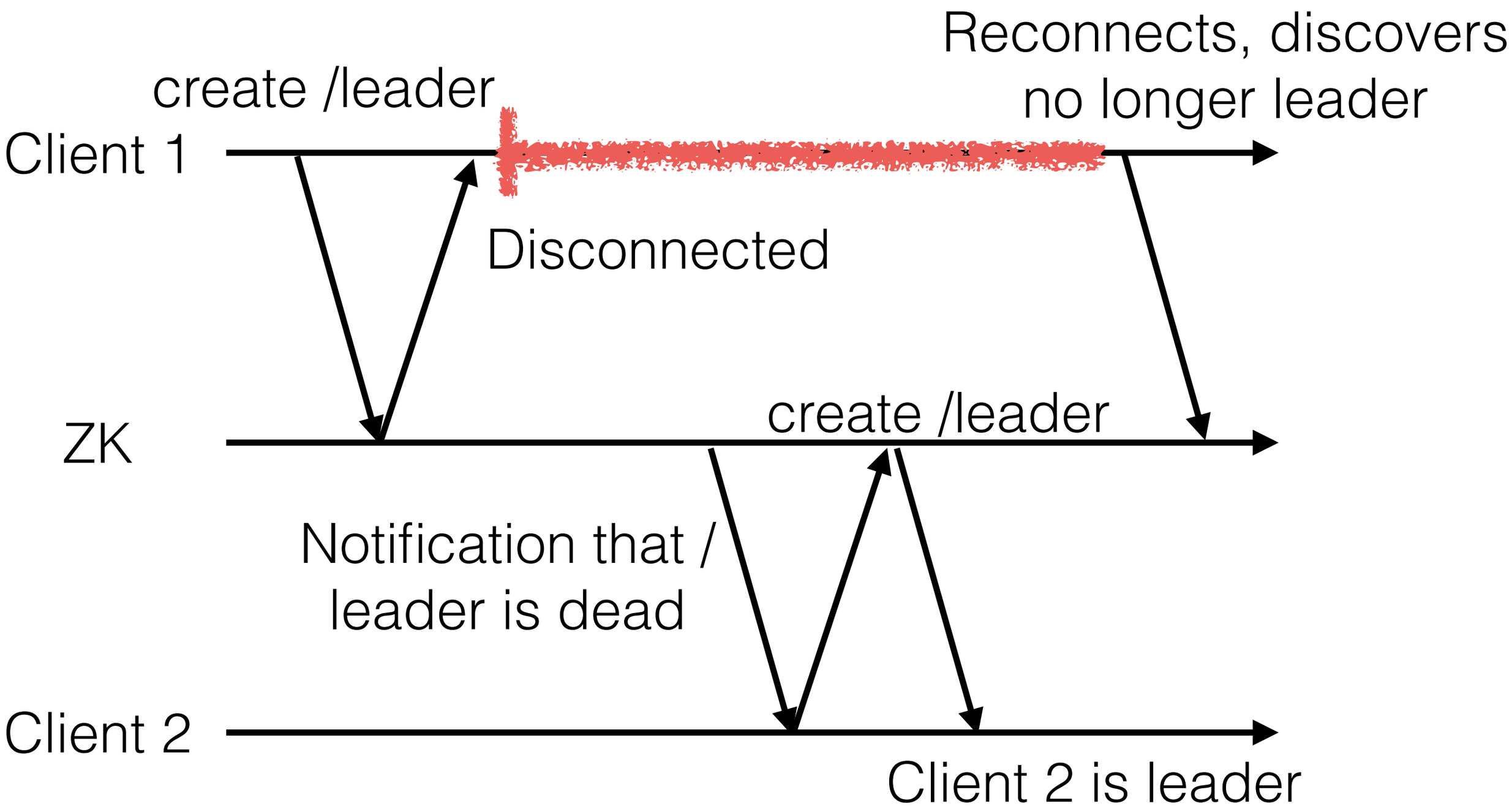
Strawman RSM implementation



Primary/backup RSM



Dangers of ignorance



Dangers of ignorance

- Each client needs to be aware of whether or not its connected: when disconnected, can not assume that it is still included in any way in operations
- By default, ZK client will NOT close the client session because it's disconnected!
  - Assumption that eventually things will reconnect
  - Up to you to decide to close it or not

Hypervisor (Case Study)

Hypervisor-Based Fault-Tolerance

THOMAS C. BRESSOUD
Isis Distributed Systems
and
FRED B. SCHNEIDER
Cornell University

Protocols to implement a fault-tolerant computing system are described. These protocols augment the hypervisor of a virtual-machine manager and coordinate a primary virtual machine with its backup. No modifications to the hardware, operating system, or application programs are required. A prototype system was constructed for HP's PA-RISC instruction-set architecture. Even though the prototype was not carefully tuned, it ran programs about a factor of 2 slower than a bare machine would.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Fault-tolerant computing system, primary/backup approach, virtual-machine manager

1. INTRODUCTION

One popular scheme for implementing fault tolerance involves replicating a computation on processors that fail independently. Replicas are coordinated so that they perform the same sequence of state transitions and, therefore, produce the same results. This article describes a novel implementation of that scheme. We interpose a software layer between the hardware and the operating system. The result is a fault-tolerant computing system whose implementation does not require modifications to the hardware, operating system, or any application software.

http://www.cs.cornell.edu/ken/book/2005sp/vft.tocs.pdf

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of ap-

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

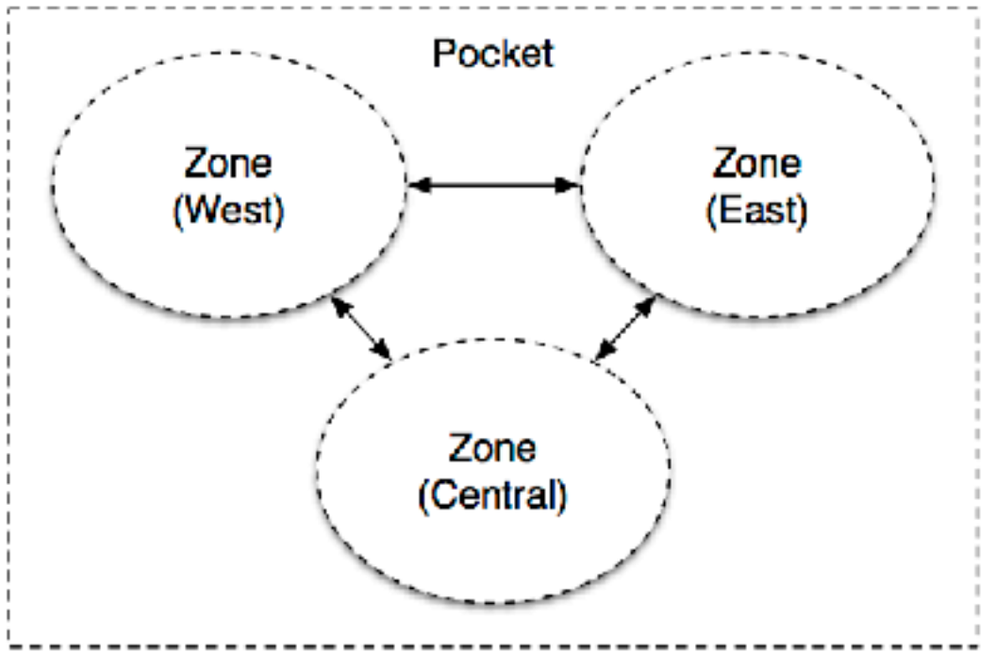
1 Introduction

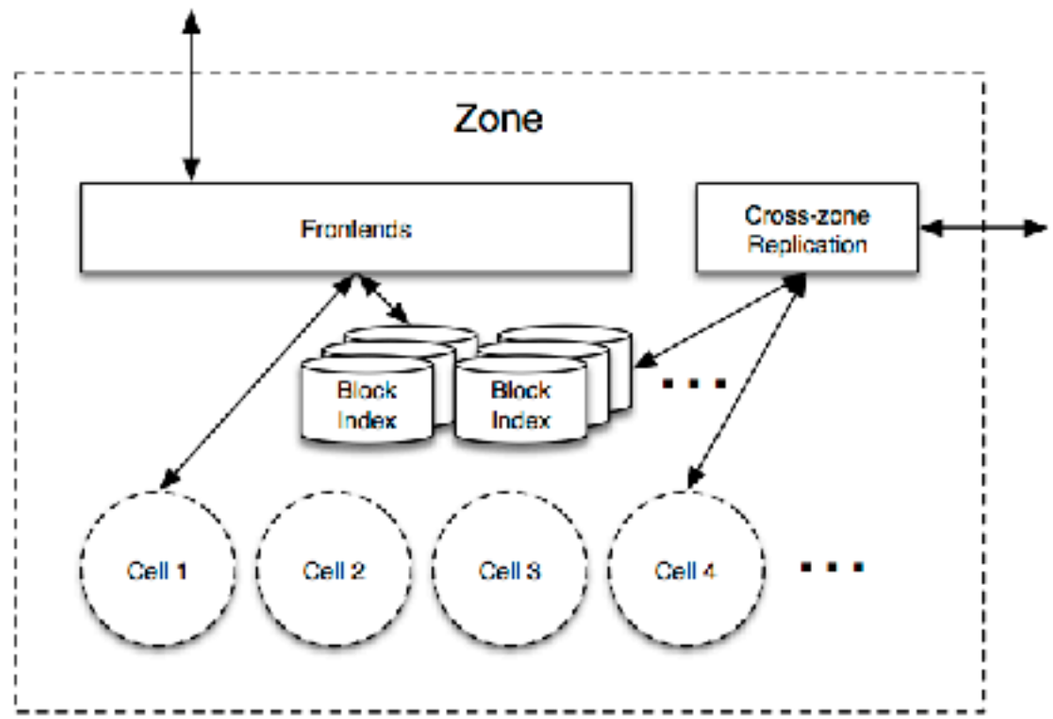
Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically

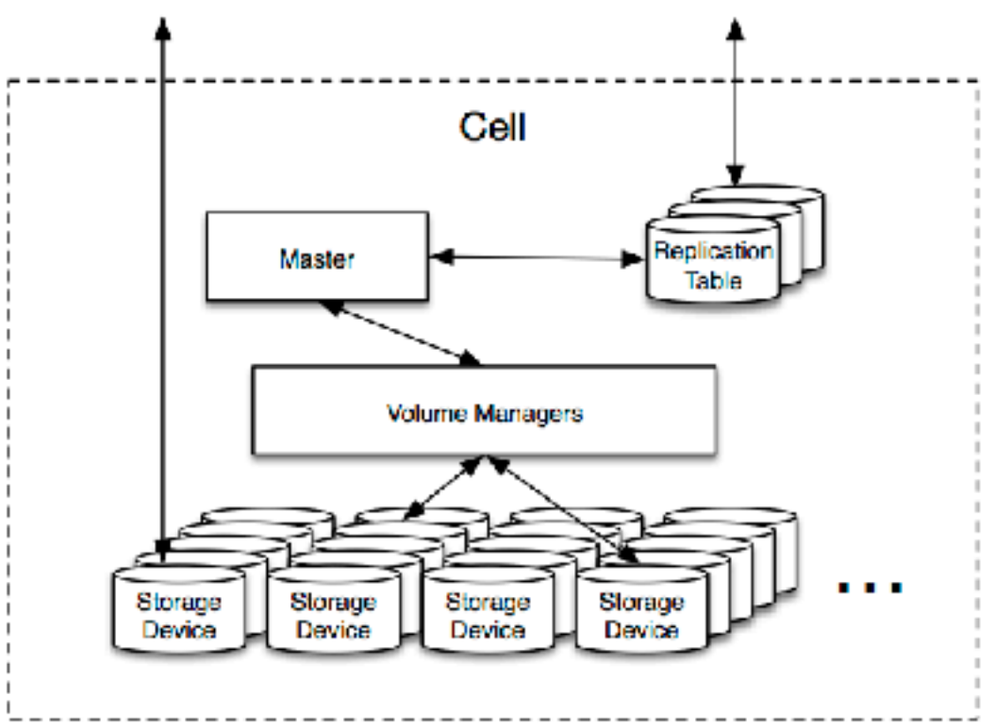
tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

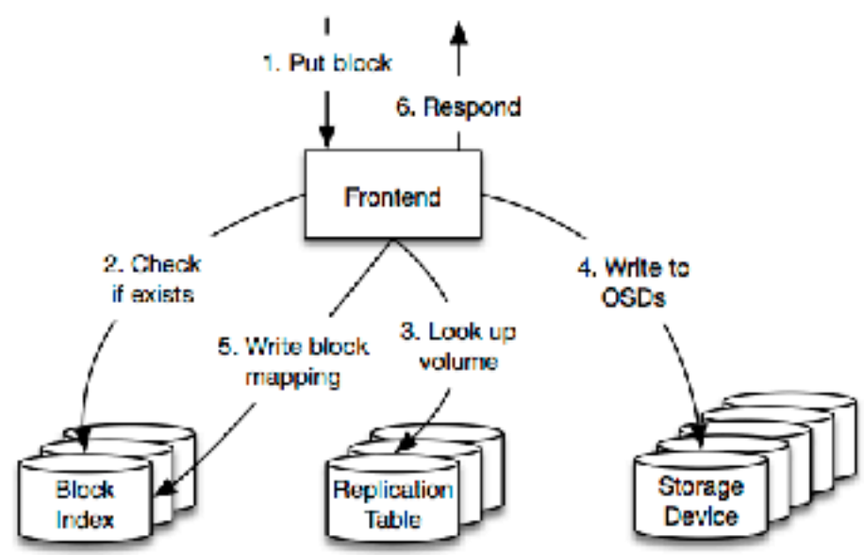
Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps.

MagicPocket

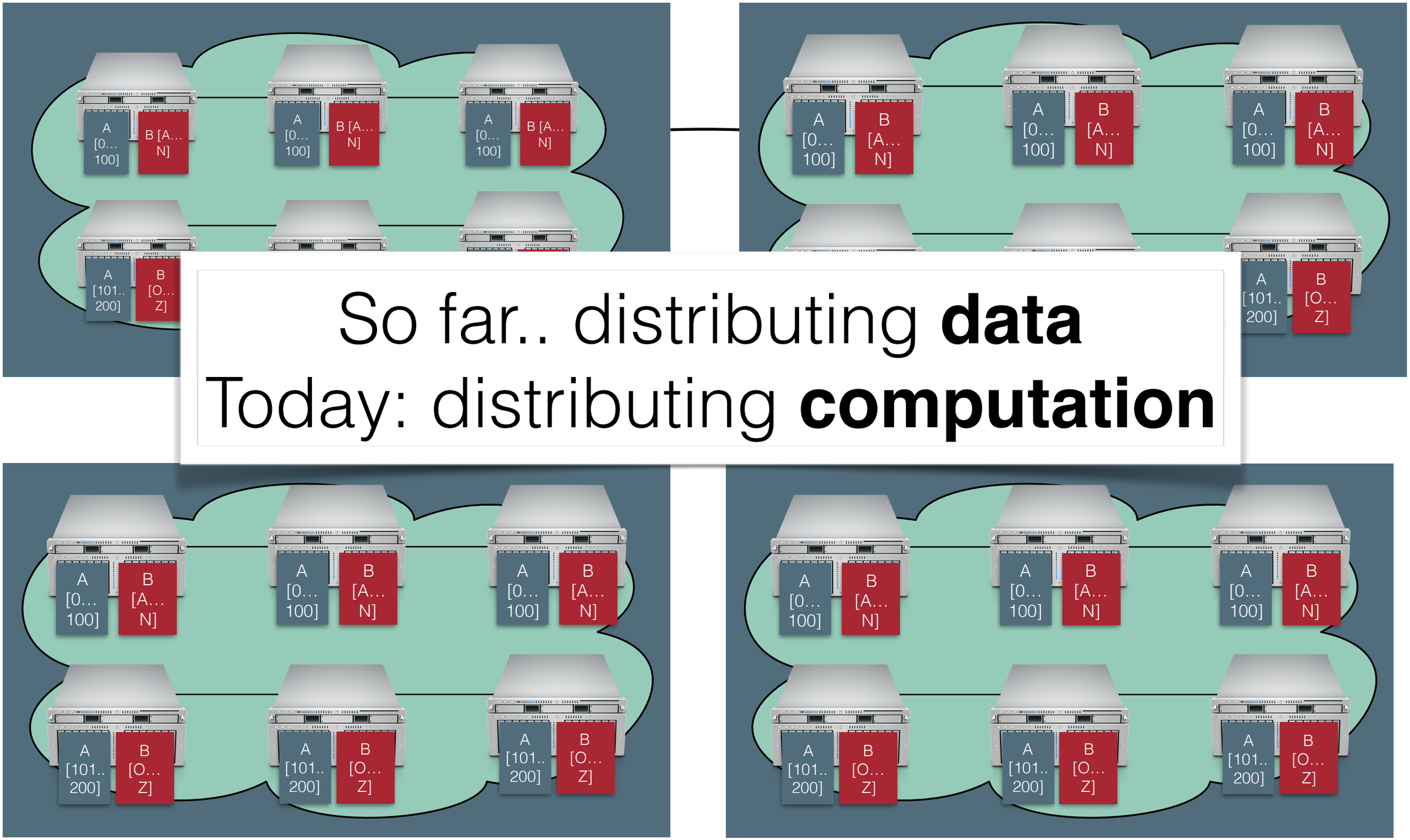








Review: Why Distributed?



So far.. distributing **data**

Today: distributing **computation**

Today

- Distributed Computation Models
- Has anyone heard of this "big data" thing?
- Exploiting parallelism
- MapReduce
- Spark

More data, more problems

- I have a 1TB file
- I need to sort it
- …My computer can only read 60MB/sec
- …
- …
- …
- 1 day later, it’s done

More data, more problems

- Think about scale:
  - Google indexes ~20 petabytes of web pages per **day** (as of 2008!)
  - Facebook has 2.5 petabytes of user data, increases by 15 terabytes/day (as of 2009!)

Distributing Computation

Distributing Computation

- Can't I just add 100 nodes and sort my file 100 times faster?
- Not so easy:
  - Sending data to/from nodes
  - Coordinating among nodes
  - Recovering when one node fails
  - Optimizing for locality
  - Debugging

Distributing Computation

- Lots of these challenges re-appear, regardless of our specific problem
  - How to split up the task
  - How to put the results back together
  - How to store the data
- Enter, MapReduce

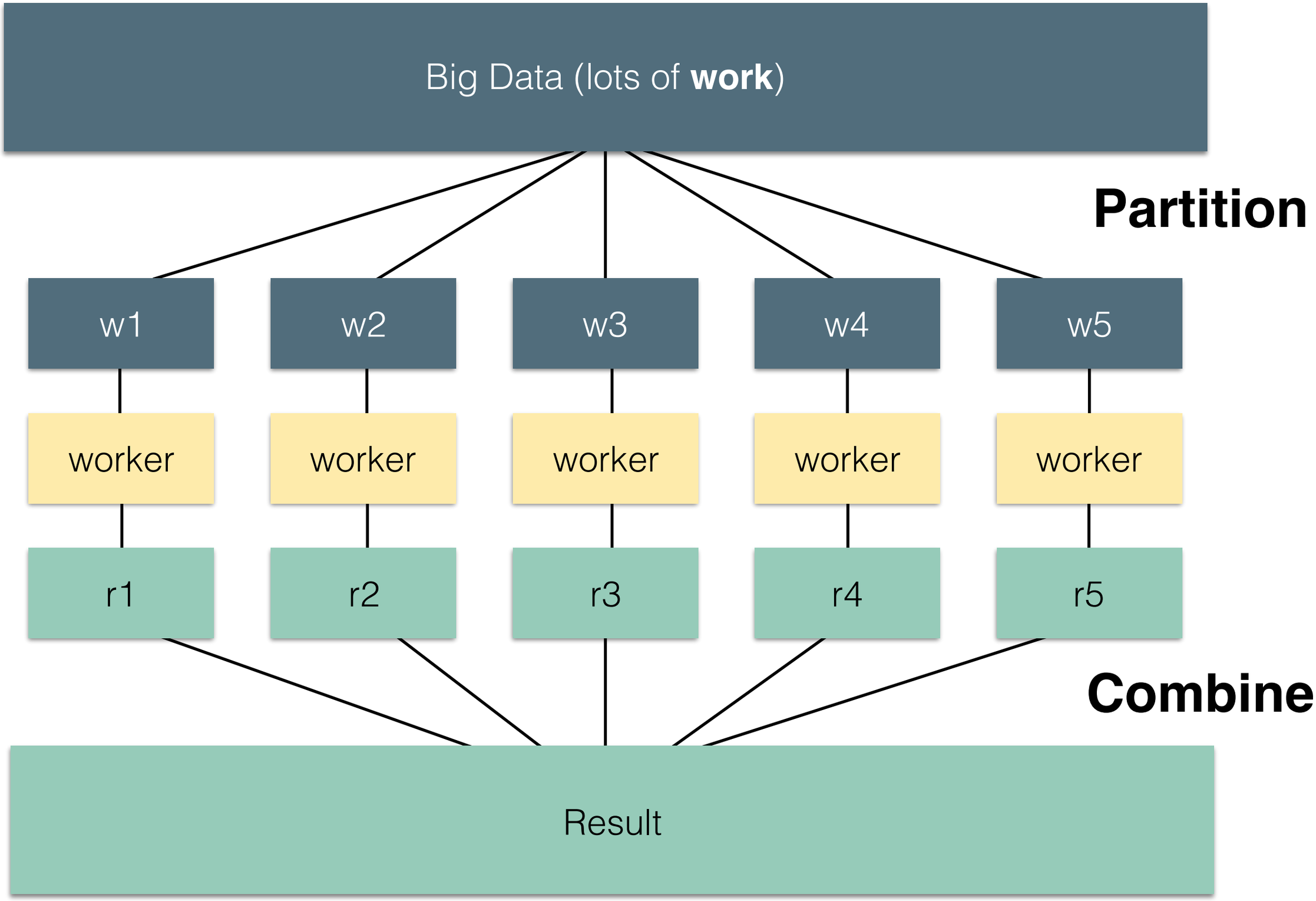
MapReduce

- A programming model for large-scale computations
  - Takes large inputs, produces output
  - No side-effects or persistent state other than that input and output
- Runtime library
  - Automatic parallelization
  - Load balancing
  - Locality optimization
  - Fault tolerance

MapReduce

- Partition data into splits (**map**)
- Aggregate, summarize, filter or transform that data (**reduce**)
- Programmer provides these two methods

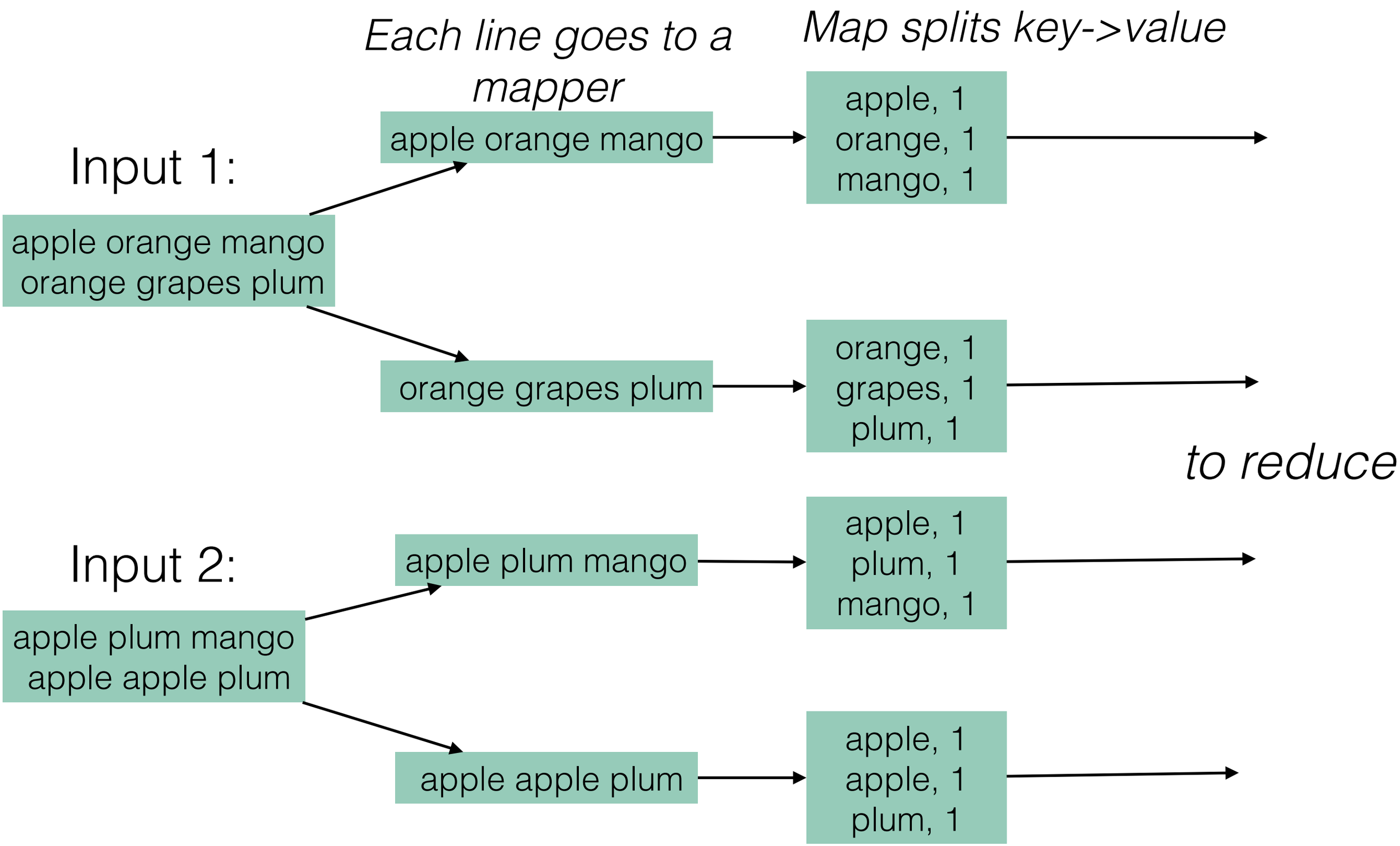
MapReduce: Divide & Conquer



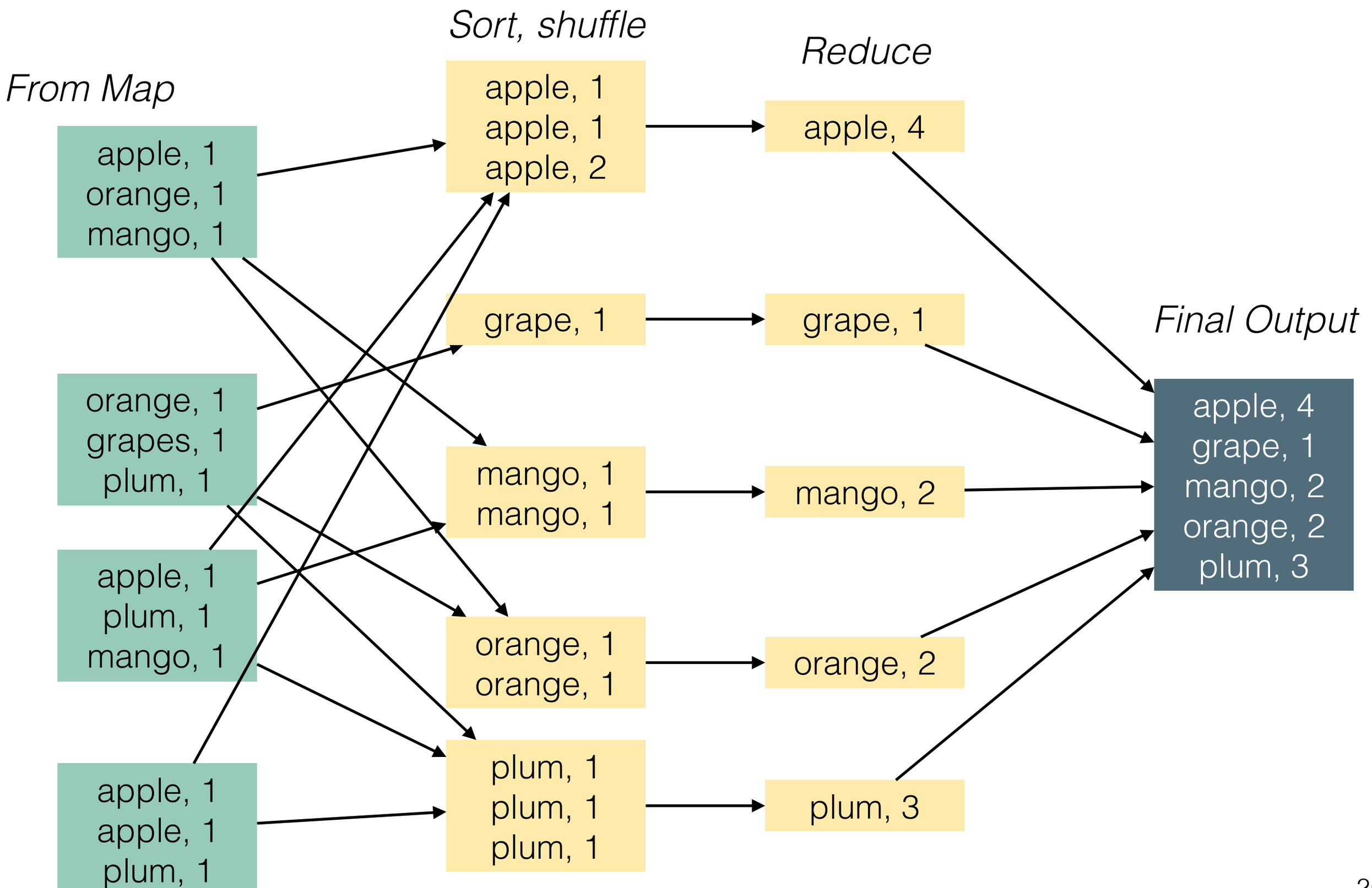
MapReduce: Example

- Calculate word frequencies in documents
- Input: files, one document per record
- **Map** parses documents into words
  - Key - Word
  - Value - Frequency of word
- **Reduce**: compute sum for each key

MapReduce: Example



MapReduce: Example



MapReduce Applications

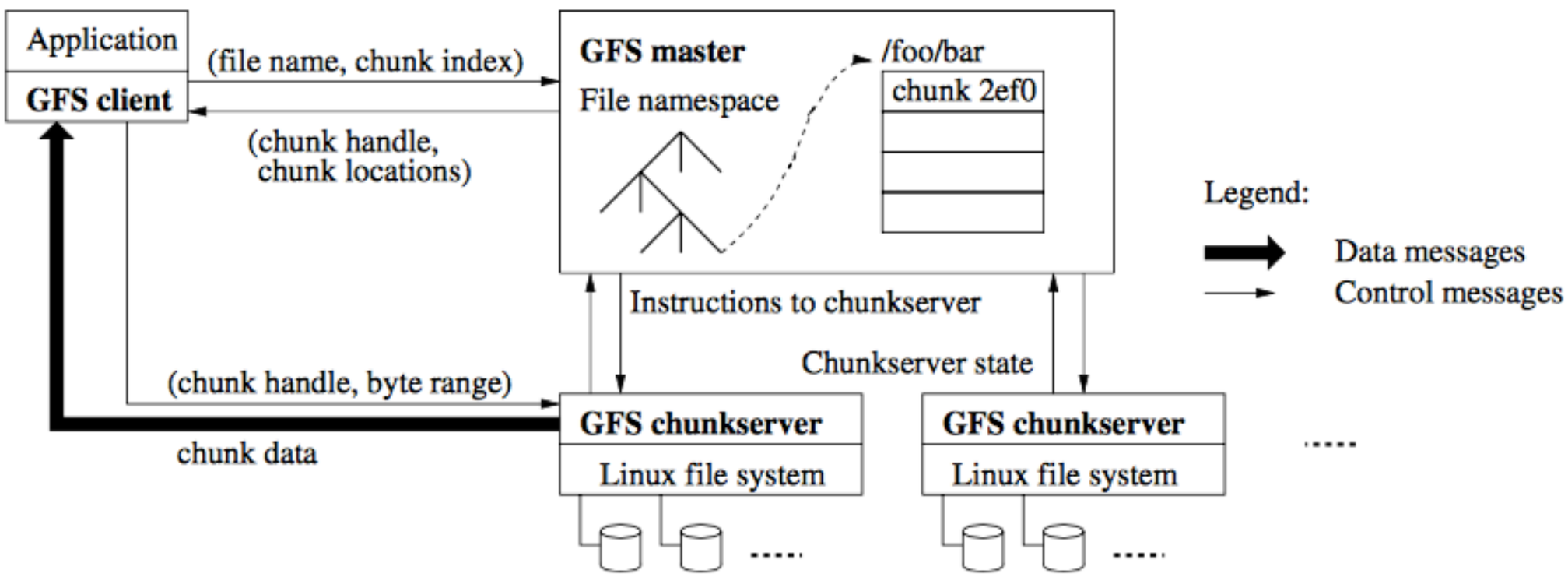
- Distributed grep
- Distributed clustering
- Web link graph traversal
- Detecting duplicate web pages

MapReduce: Implementation

- Input data is partitioned into *M* splits
- Map - extract information about each split
- Each map produces *R* partitions
- Shuffle and sort
  - Bring the right partitions to the right reducer
- Output from R

MapReduce: Implementation

- Each worker node is **also** a GFS chunk server!



MapReduce: Scheduling

- One master, many workers
- Input data split into *M* map tasks (typically 64MB ea)
- *R* reduce tasks
- Tasks assigned to works dynamically; stateless and idempotent -> easy fault tolerance for workers
- Typical numbers:
  - 200,000 map tasks, 4,000 reduce tasks across 2,000 workers

MapReduce: Scheduling

- Master assigns map task to a free worker
  - Prefer "close-by" workers for each task (based on data locality)
  - Worker reads task input, produces intermediate output, stores locally (K/V pairs)
- Master assigns reduce task to a free worker
  - Reads intermediate K/V pairs from map workers
  - Reduce worker sorts and applies some *reduce* operation to get the output

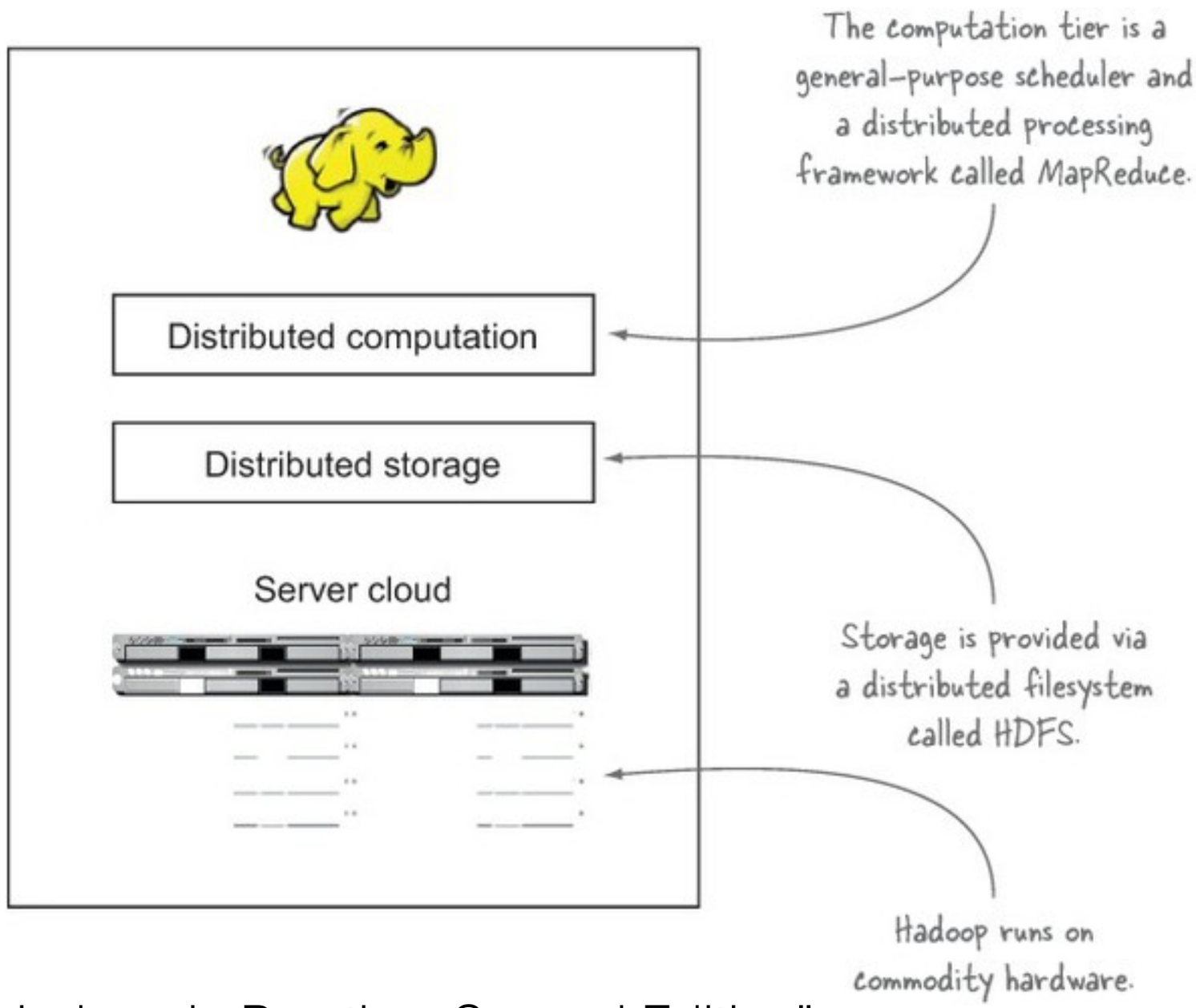
Fault tolerance via re-execution

- Ideally, fine granularity tasks (more taks than machines)
- On worker-failure:
  - Re-execute completed and in-progress map tasks
  - Re-executes in-progress reduce tasks
  - Commit completion to master
- On master-failure:
  - Recover state (master checkpoints in a primary-backup mechanism)

MapReduce in Practice

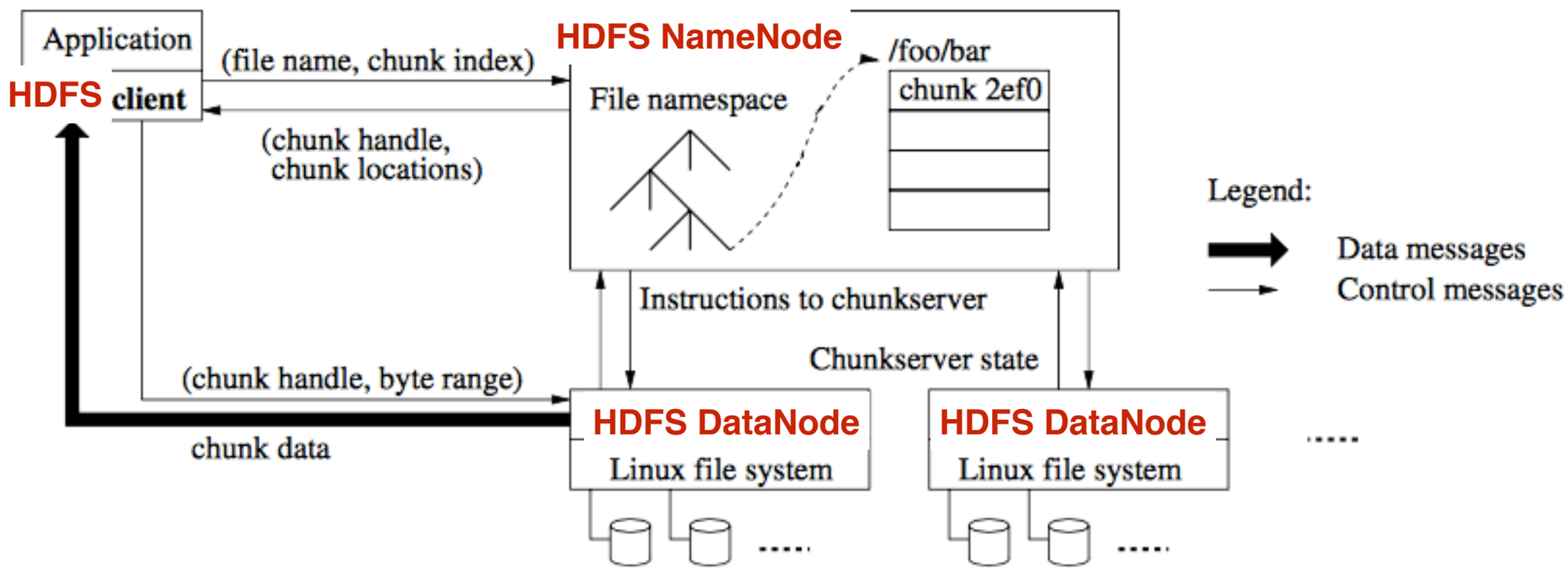
- Originally presented by Google in 2003
- Widely used today (**Hadoop** is an open source implementation)
- Many systems designed to have easier programming models that compile into MapReduce code (Pig, Hive)

Hadoop



"Hadoop in Practice, Second Edition"

Hadoop: HDFS


Application
HDFS client
(file name, chunk index)
(chunk handle,
chunk locations)
HDFS NameNode
File namespace
/foo/bar
chunk 2ef0
Legend:
Data messages
Control messages
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
HDFS DataNode
Linux file system
HDFS DataNode
Linux file system

HDFS (GFS Review)

- Files are split into blocks (128MB)
- Each block is replicated (default 3 block servers)
- If a host crashes, all blocks are re-replicated somewhere else
- If a host is added, blocks are rebalanced
- Can get awesome locality by pushing the map tasks to the nodes with the blocks (just like MapReduce)

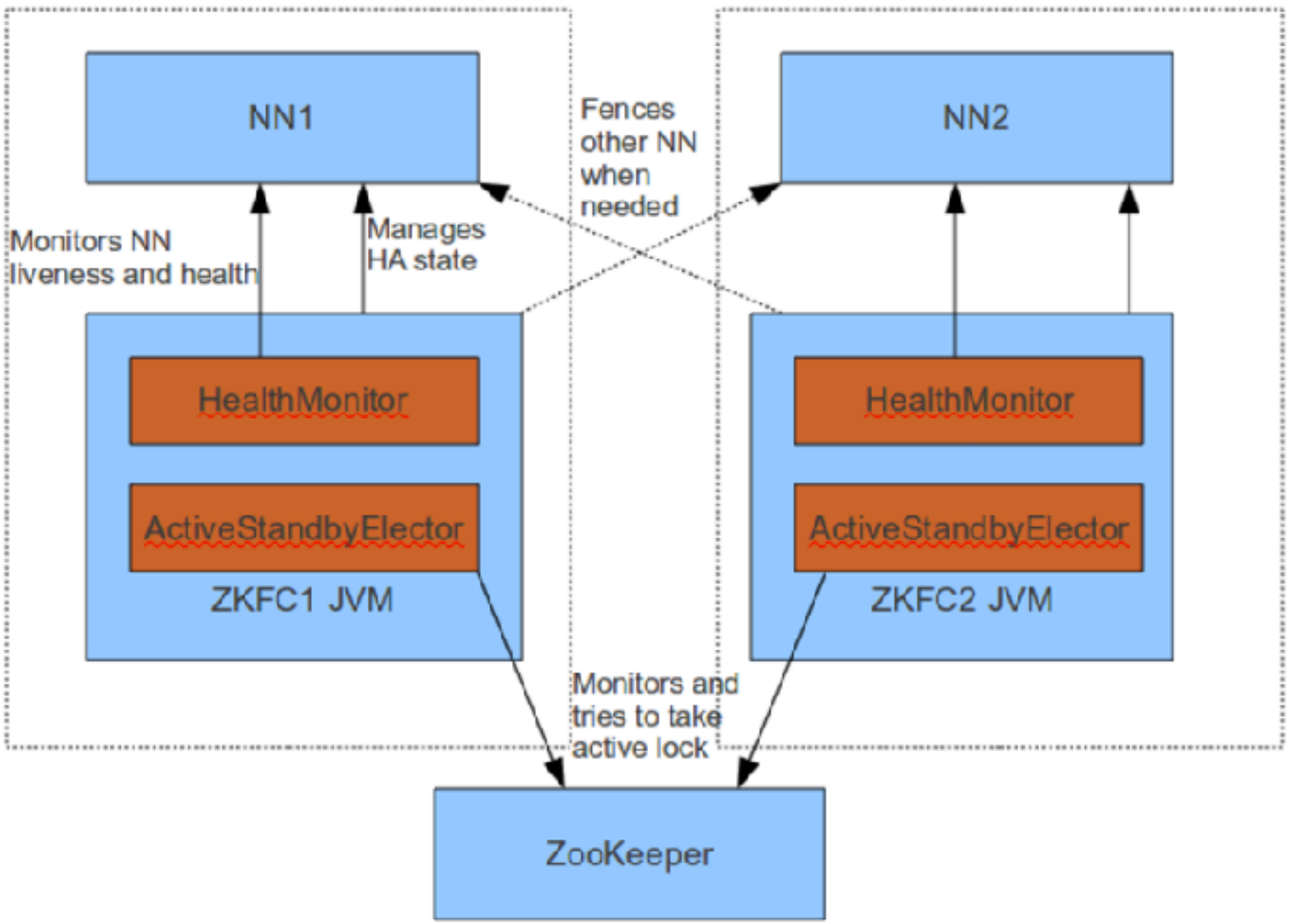
YARN

- Hadoop's distributed resource scheduler
- Helps with scaling to very large deployments (>4,000 nodes)
- General purpose system for scheduling executable code to run on workers, managing those jobs, collecting results

Hadoop + ZooKeeper

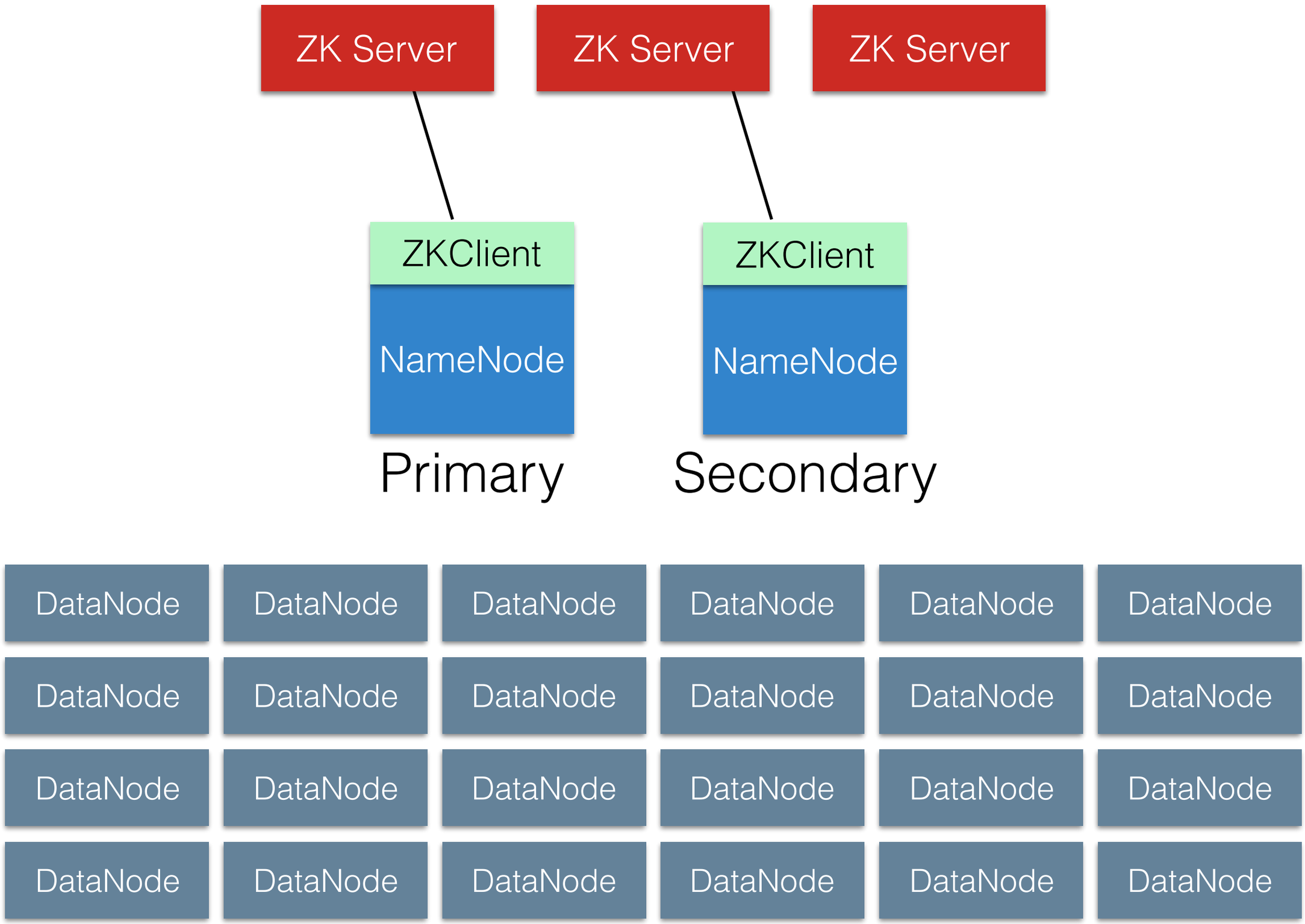
- Hadoop uses ZooKeeper for automatic failover for HDFS
- Run a ZooKeeper client on each NameNode (master)
- Primary NameNode and standbys all maintain session in ZK, primary holds an ephemeral lock
- If primary doesn't maintain contact it session expires, triggering a failure (handled by the client)
- Similar mechanism to failover a YARN controller

Hadoop + ZooKeeper

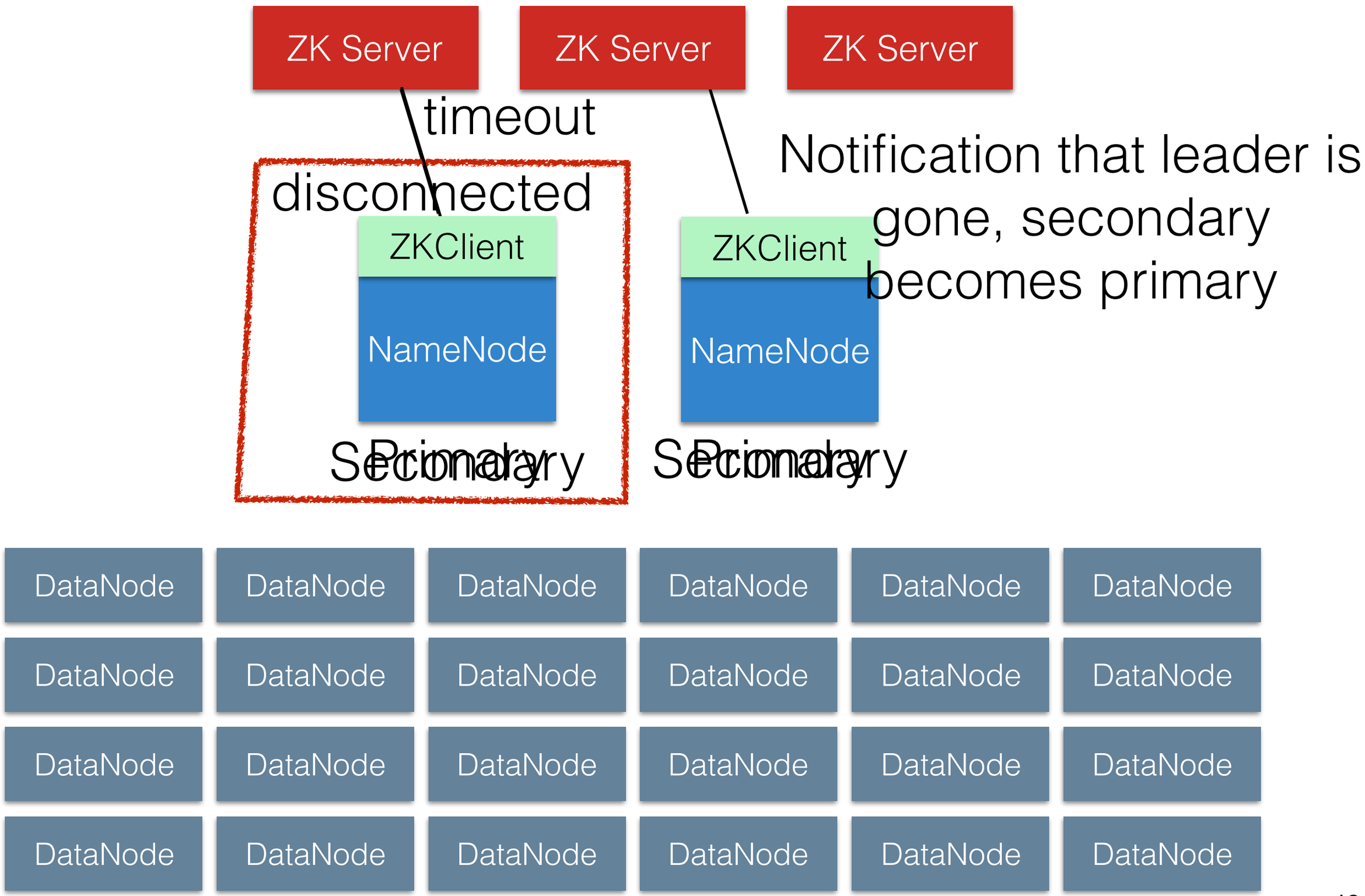


https://issues.apache.org/jira/secure/attachment/12519914/zkfc-design.pdf

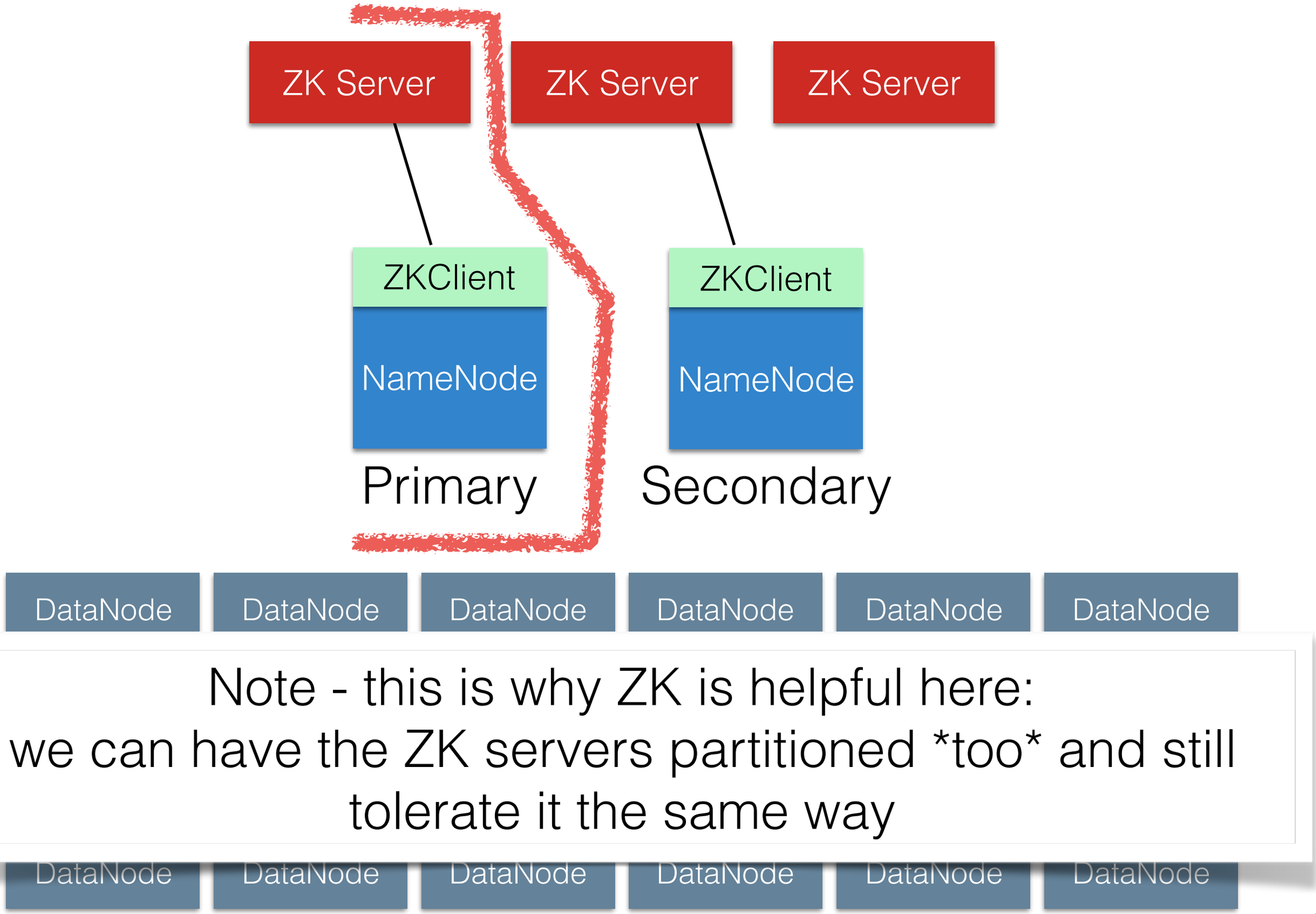
Hadoop + ZooKeeper



Hadoop + ZooKeeper



Hadoop + ZooKeeper

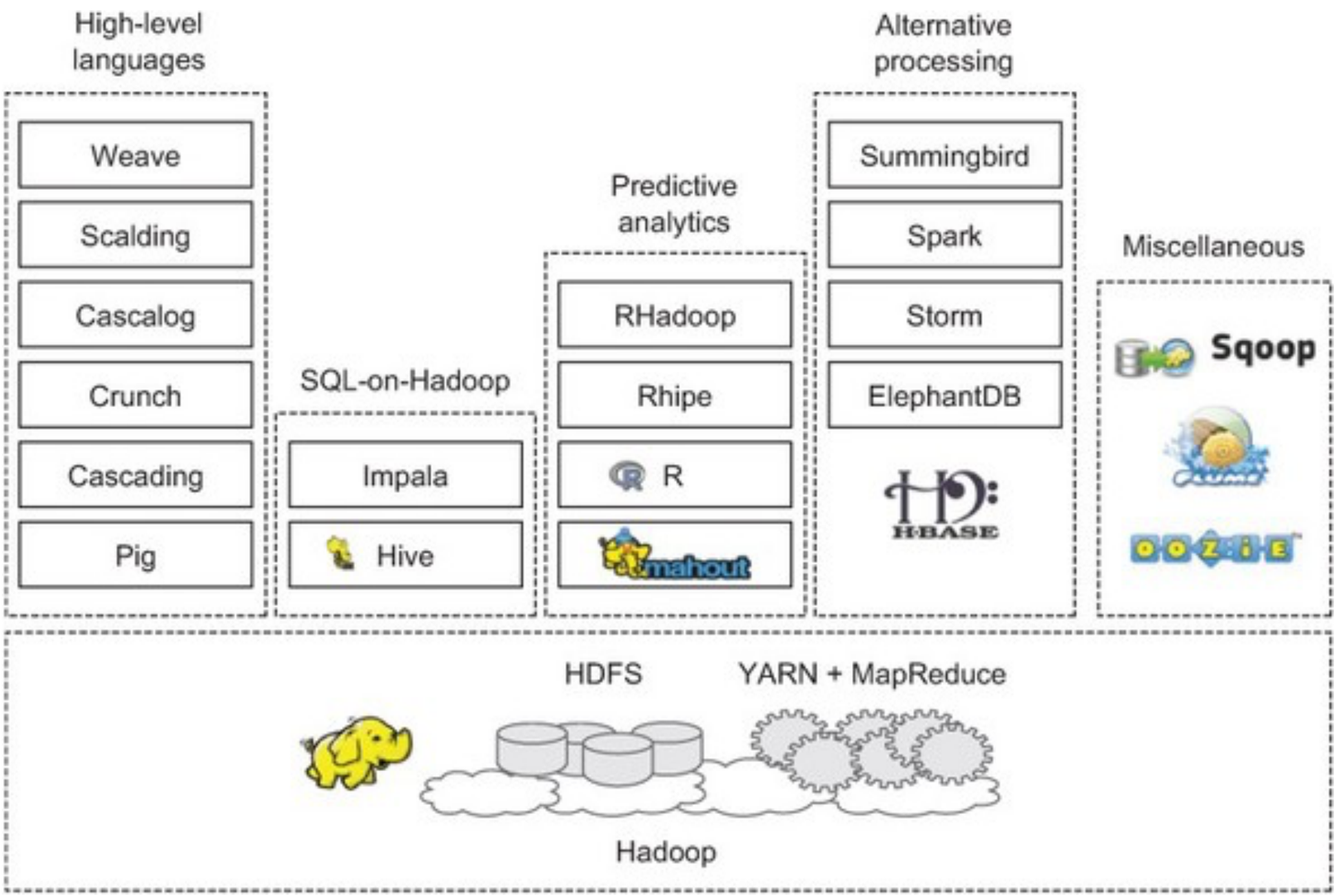


Note - this is why ZK is helpful here:
we can have the ZK servers partitioned *too* and still tolerate it the same way

Hadoop + ZooKeeper

- Why run ZK client in a different process?
- Why run ZK client on the same machine?
- Can this config still lead to unavailability?
- Can this config lead to inconsistency?

Hadoop Ecosystem



Beyond MapReduce

- MapReduce is optimized for I/O bound problems
  - E.g. sorting big files
- Primitive API
  - Developers have to build on the simple map/reduce abstraction
- Typically result in many small files that need to be combined (reduce results)

Apache Spark

- Developed in 2009 at Berkeley
- Revolves around resilient distributed data sets (RDD)
- Fault-tolerant read-only collections of elements that can be operated on in parallel

Spark vs MapReduce

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

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

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

RDD Operations

- Generally expressive:
- Transform an existing RDD dataset into a new one
  - Map, filter, distinct, union, sample, join, reduce, etc
- Make some actions on RDDs after a computation
  - collect, count, first, foreach, etc.

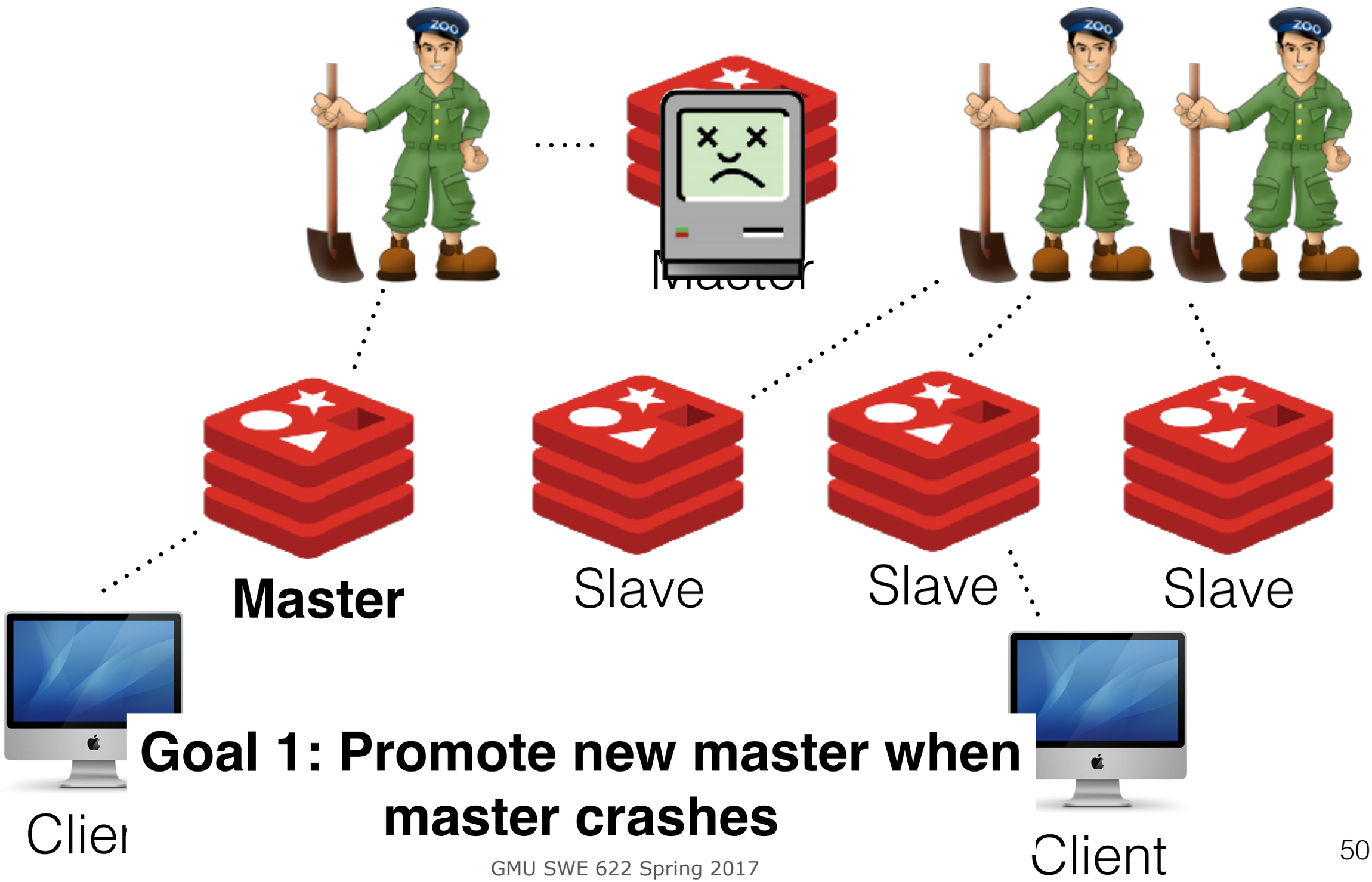
Spark

- Aggressively caches data in memory
- *AND* is fault tolerant
- How? MapReduce got tolerance through its disk replication
- RDDs are *resilient* but they are also **restricted**
  - Immutable, partitioned records
  - Can only be built through coarse-grained and **determistic** transformations (e.g. map, filter, join)

Spark - Fault Recovery

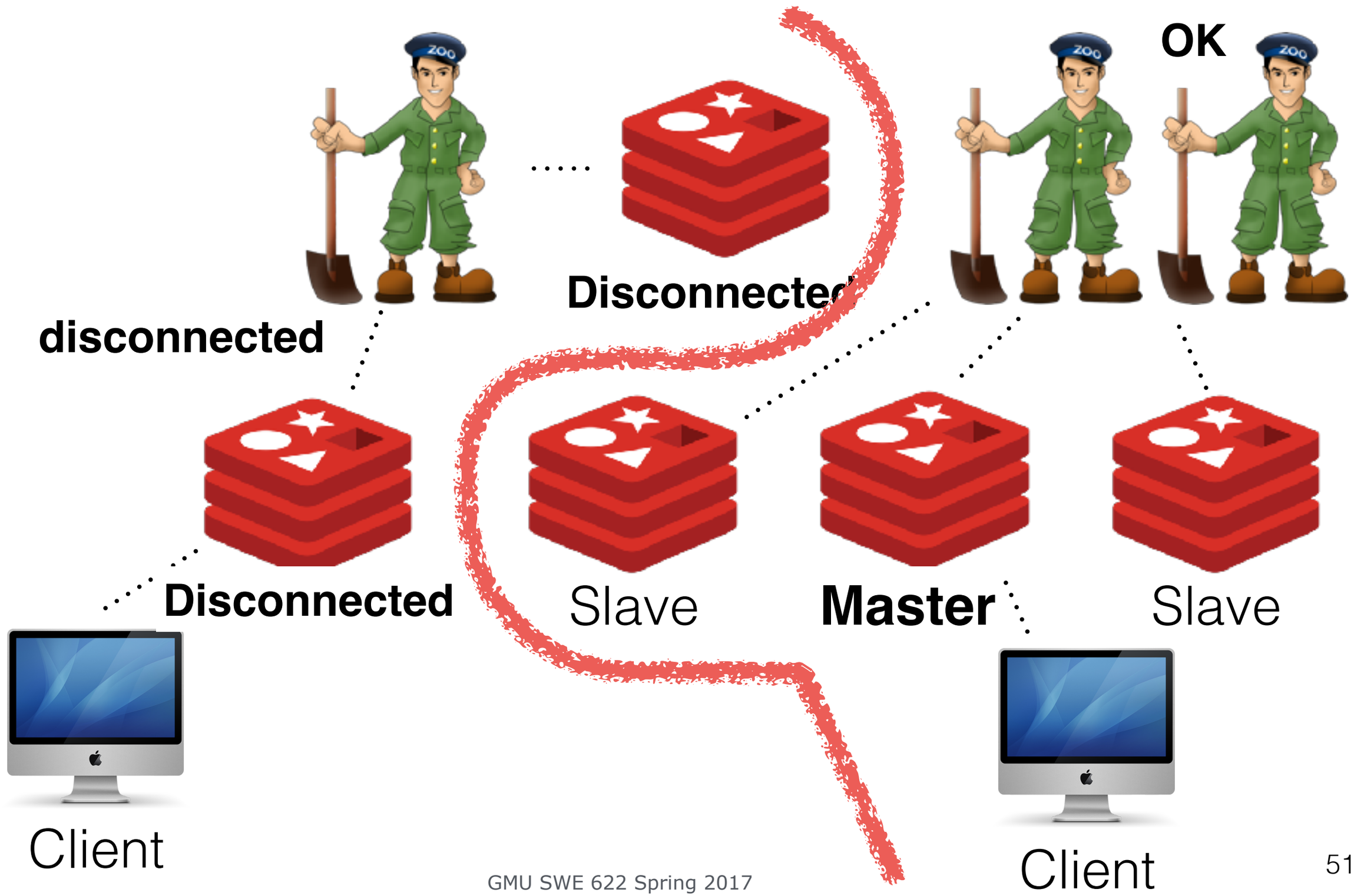
- Can efficiently recover from faults using lineage
- Just like write-ahead-logging:
  - Log which RDD was the input, and the **high level operation** to generate the output
  - Only need to log that the operation applied to **the entire** dataset, not per-element
  - RDDs track the graph of transformations that built them all the way to their origin

How fault tolerant is CFS?

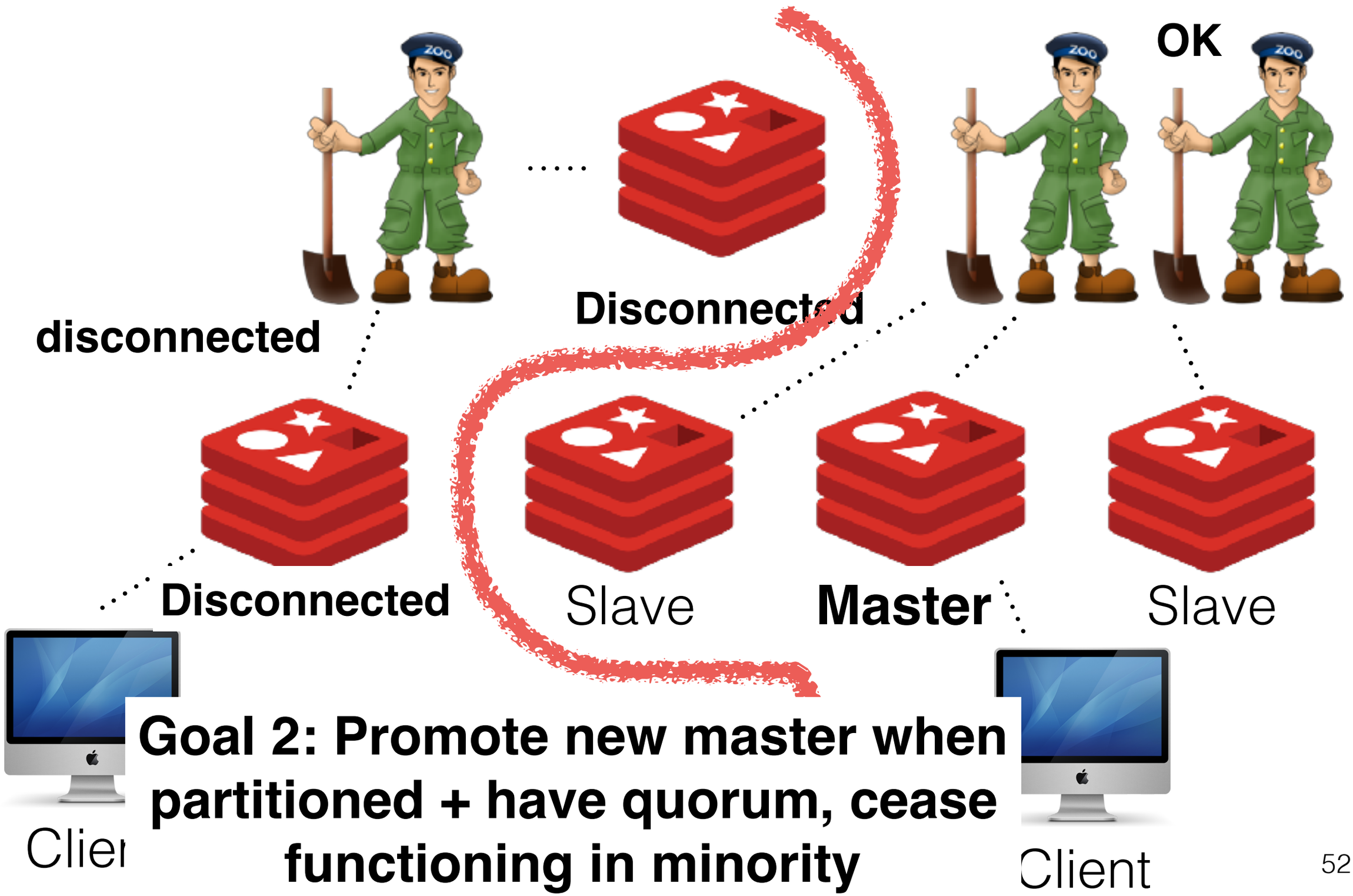


Goal 1: Promote new master when master crashes

How fault tolerant is CFS?



How fault tolerant is CFS?



Goal 2: Promote new master when partitioned + have quorum, cease functioning in minority

Return to failover & CFS

- WAIT returns the number of servers that acknowledged the update
- How many do we really need to know it though?
  - So far, for N servers, need N ack's
  - N/2+1?

Role of ZK in CFS for fault tolerance

- We will track who the Redis master is with ZK
- If something is wrong (WAIT fails) or can't contact master, then you need to challenge the master's leadership
  - Maybe master is partitioned from slaves but ZK isn't
  - Maybe master is crashed
- If you promote a master, then you need to track that in ZK
- If you were disconnected and reconnect from ZK, you need to validate who the master is

HW 5: Fault Tolerance

- We're going all-in on ZooKeeper here
- Use ZK similar to how HDFS does: each Redis slave will have a dedicated ZK client to determine who the master Redis server is
- Redis master holds a lease that can be renewed perpetually
- When client notices a problem (e.g. WAIT doesn't work right or can't talk to master) it proposes becoming the master
- As long as a client can talk to a quorum of ZK nodes, then they can decide who the leader is
- Clients don't need to vote - just matters that there is exactly one of them

HW 5: Fault Tolerance

- Does this make Redis a fault-tolerant CP system?
- No.
- Argument:
  - Master might have gotten writes that succeeded there but not been replicated, then loses leadership
  - Then later, master comes back online. Master accepted writes that nobody else knew about (and kept functioning without)

HW 5: Fault Tolerance

- Is this a fundamental limitation, is there a theorem that says this is impossible?
- No! It's just how Redis is implemented.
- It would be easier if we could do SET and WAIT in one step, failing SET if WAIT failed
  - That's literally what synchronous replication, and Redis is an asynchronous system
- https://aphyr.com/posts/283-jepsen-redis
- https://aphyr.com/posts/307-call-me-maybe-redis-redux

Lab: ZooKeeper Leadership

- We're going to simulate failures in our system, and work to maintain a single leader