Systems@Google

Vamsi Thummala
Slides by Prof. Cox

DeFiler FAQ

- Multiple writes to a dFile?
  - Only one writer at a time is allowed
    - Mutex()/ReaderWriterLock() at a dFile
- read()/write() always start at beginning of the dFile (no seeking).
- Size of a inode
  - Okay to assume fixed size but may not be a good idea to assume the size of a inode == block size
  - 256 bytes can hold 64 pointers => at least 50 blocks after metadata (satisfies the requirement)
  - Simple to implement as a linked list
    - Always the last pointer is reserved for indirect block pointer

DeFiler FAQ

- Valid status?

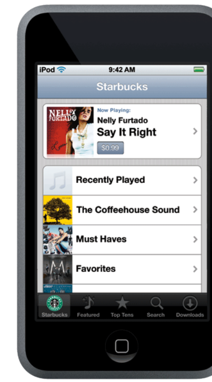
```
ReadBlock() {
    getBlock(); // returns DBuffer for the block
  /* check the contents, the buffer may be associated with
  other block earlier and the contents are invalid */
    if (checkValid())
        return buffer;
    else startFetch();
    wait for ioComplete();
    return buffer;
    }
```

DeFiler FAQ

- You may not use any memory space other than the DBufferCache
  - FreeMap + Inode region + Data blocks all should reside in DBufferCache space
  - You can keep the FreeMap + Inode region in memory all the time
    - Just have an additional variable called “isPinned” inside DBuffer.
- Synchronization: Mainly in DBufferCache, i.e, getBlock() and releaseBlock()
  - You need a CV or a semaphore to wakeup the waiters
- Only a mutex need at a DFS level
- No synchronization at the VirtualDisk level
  - A queue is enough to maintain the sequence of requests

A brief history of Google

=



BackRub:
1996

4 disk drives
24 GB total storage

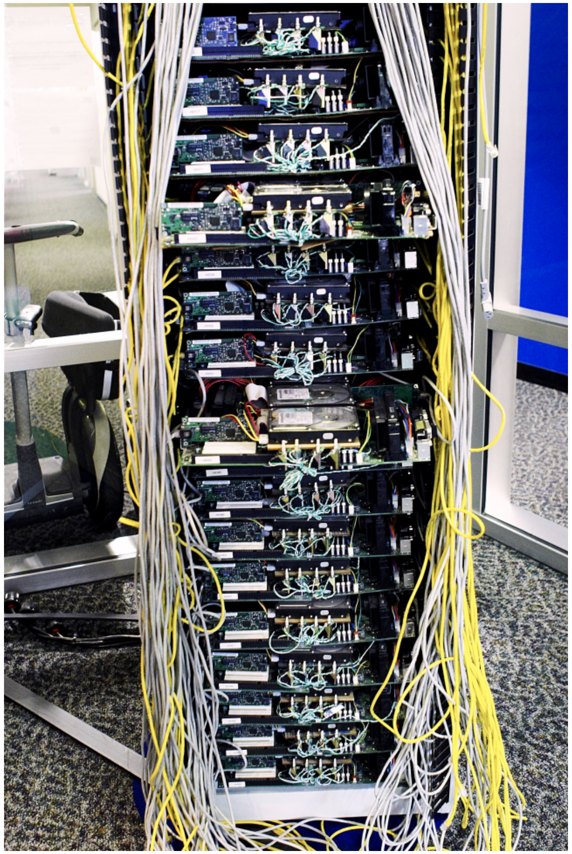
A brief history of Google

=

Google:
1998

44 disk drives
366 GB total storage

A brief history of Google

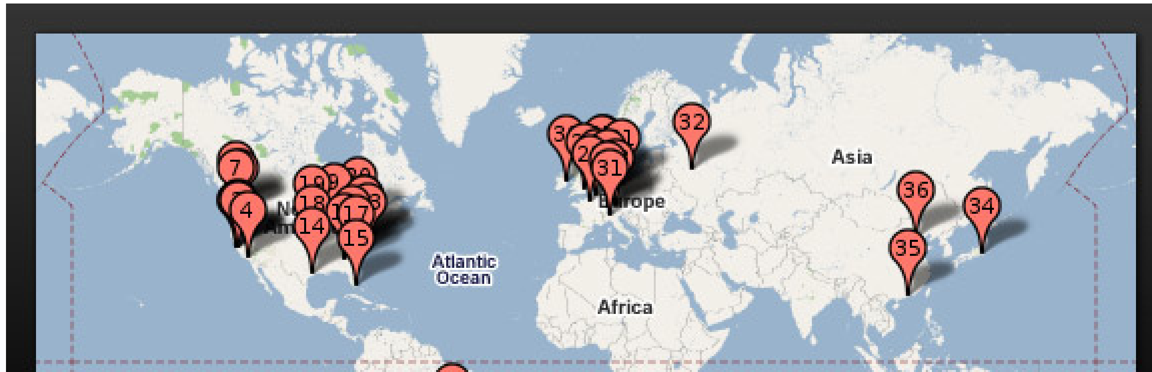


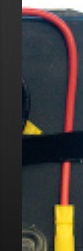
Google: 2003

15,000 machines

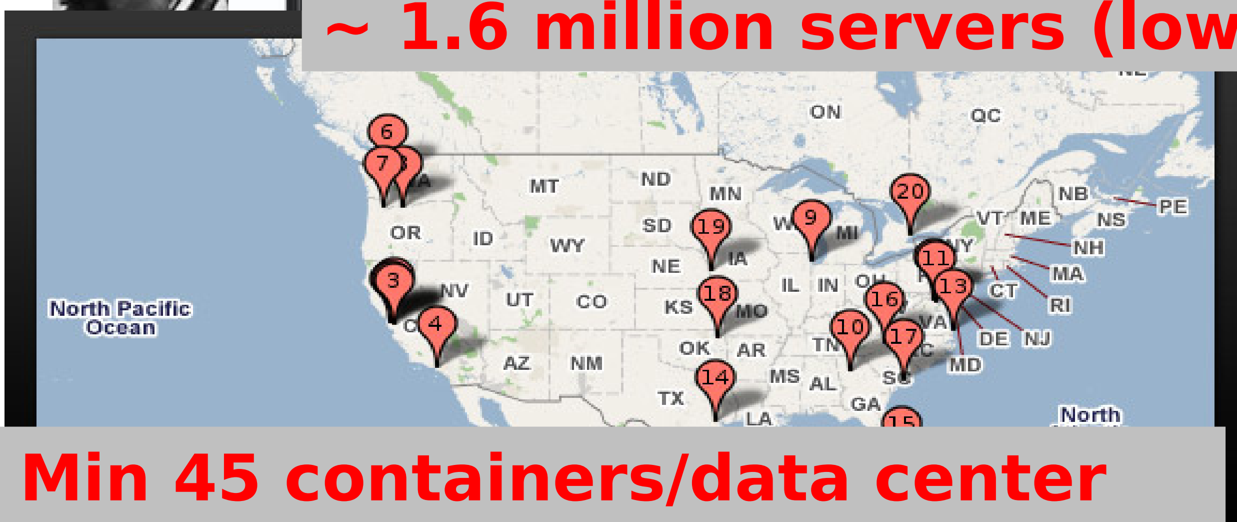
? PB total storage

A brief history of Google





45 containers x 1000 servers x 36 sites =
~ 1.6 million servers (lower bound)



per shipping

Min 45 containers/data center

Google design principles

- **Workload: easy to parallelize**
  - Want to take advantage of many processors, disks
- **Why not buy a bunch of supercomputers?**
  - Leverage parallelism of lots of (slower) cheap machines
  - Supercomputer price/performance ratio is poor
- **What is the downside of cheap hardware?**

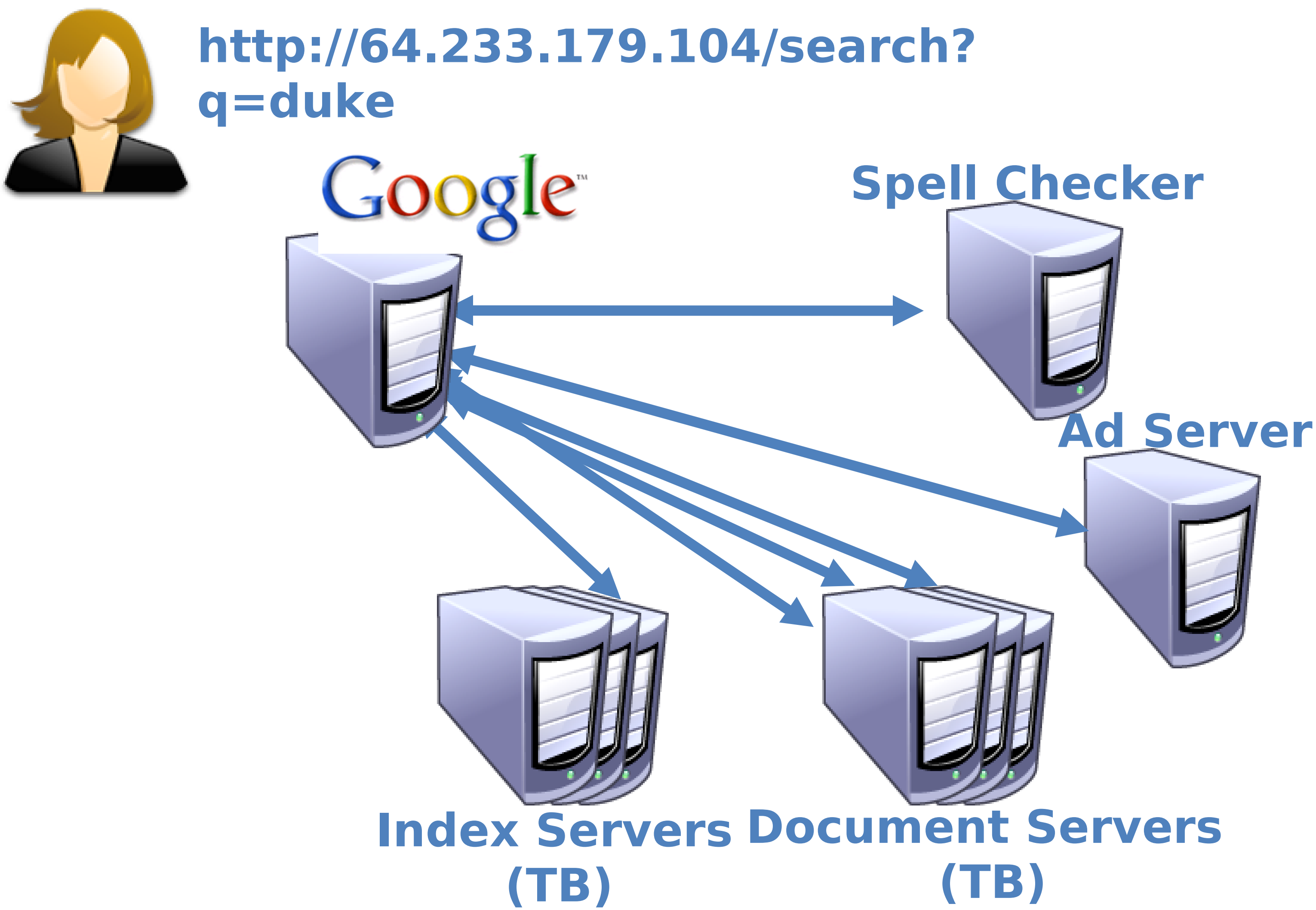
What happens on a query?

http://www.google.com/search?q=duke

http://64.233.179.104/search?q=duke

DNS

What happens on a query?



Google hardware model

- **Google machines are cheap and likely to fail**
- **What must they do to keep things up and running?**
  - Store data in several places (replication)
  - When one machine fails, shift load onto ones still around
- **Does replication get you anything else?**
  - Enables more parallel reads

Fault tolerance and performance

- **Google machines are cheap and likely to fail**
- **Does it matter how fast an individual machine is?**
  - Somewhat, but not that much
  - Parallelism enabled by replication has a bigger impact
- **Any downside to having a ton of machines?**
  - Space

Fault tolerance and performance

- **Google machines are cheap and likely to fail**
- **Any workloads where this wouldn’t work?**
  - Lots of writes to the same data
  - Web examples? (web is mostly read)

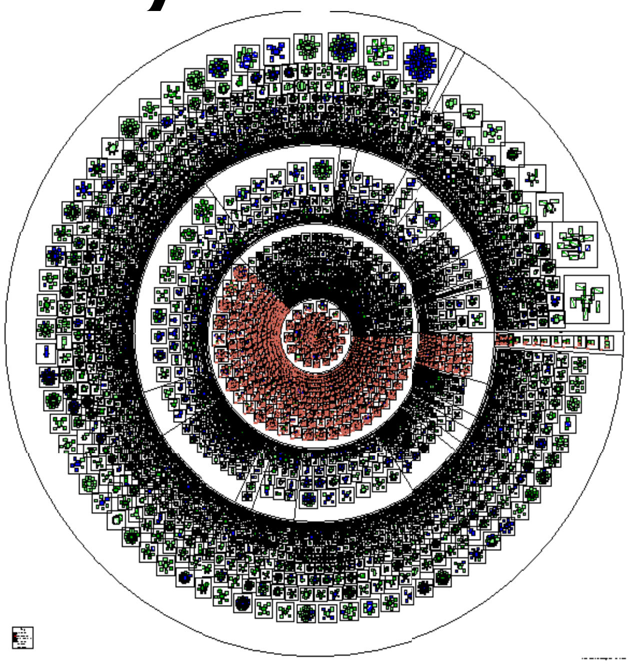
Google power consumption

- **A circa 2003 mid-range server**
  - Draws 90 W of DC power under load
  - 55 W for two CPUs
  - 10 W for disk drive
  - 25 W for DRAM and motherboard
- **Assume 75% efficient ATX power supply**
  - 120 W of AC power per server
  - 10 kW per rack

Google power consumption

- **A server rack fits comfortably in 25 ft2**
  - Power density of 400 W/ ft2
  - Higher-end server density = 700 W/ ft2
- **Typical data centers provide 70-150 W/ ft2**
  - Google needs to bring down the power density
  - Requires extra cooling or space
- **Lower power servers?**
  - Slower, but must not harm performance

OS Complexity

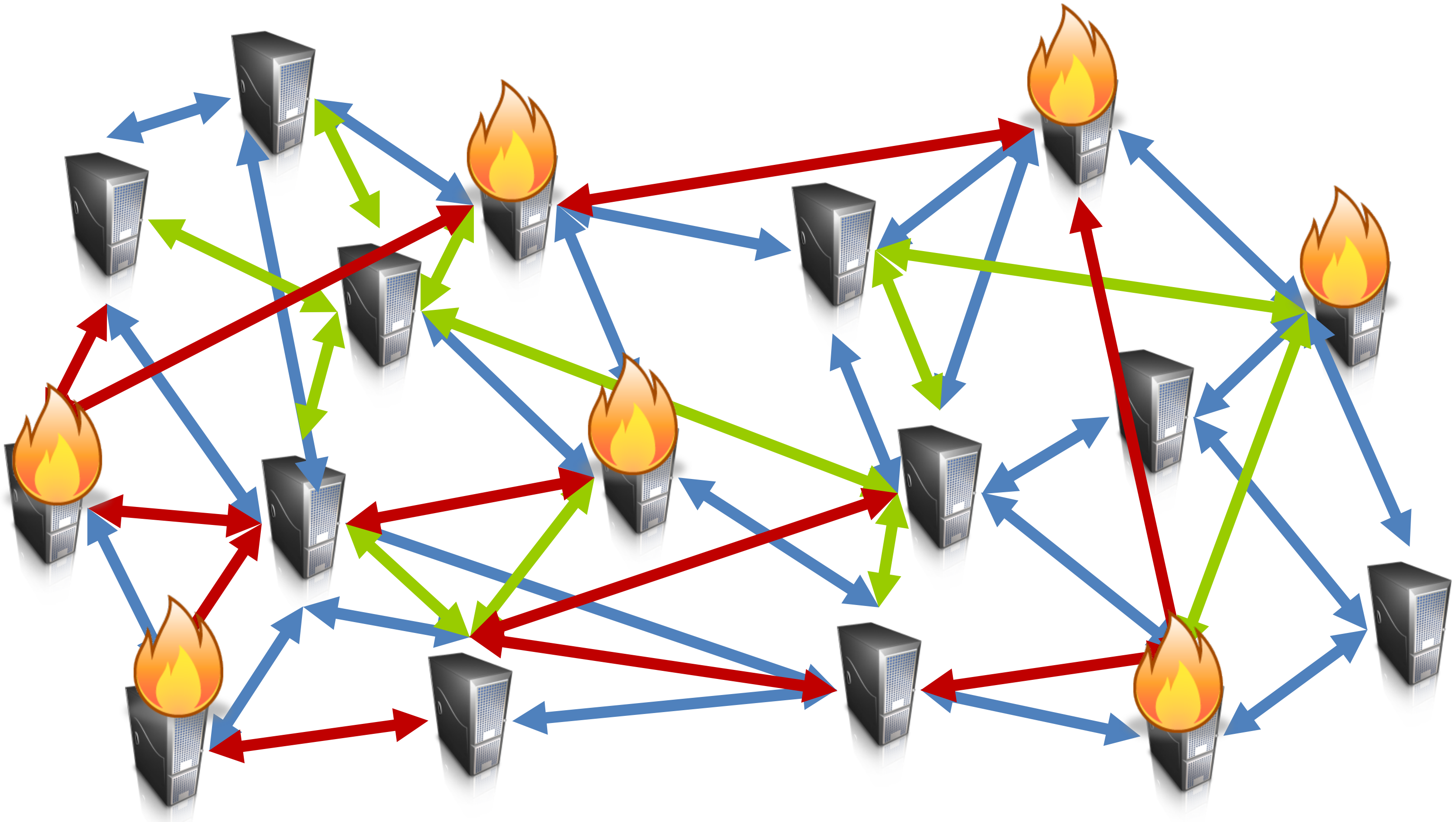


- **Lines of code**
  - XP: 40 million
  - Linux 2.6: 6 million
  - (mostly driver code)
- **Sources of complexity**
  - Multiple instruction streams (processes)
  - Multiple interrupt sources (I/O, timers, faults)

Complexity in Google

- **Consider the Google hardware model**
  - Thousands of cheap, commodity machines
- **Why is this a hard programming environment?**
  - Speed through parallelism (concurrency)
  - Constant node failure (fault tolerance)

Complexity in Google



Google provides abstractions to make programming easier.

Abstractions in Google

- **Google File System**
  - Provides data-sharing and durability
- **Map-Reduce**
  - Makes parallel programming easier
- **BigTable**
  - Manages large relational data sets
- **Chubby**
  - Distributed locking service

Problem: lots of data

- **Example:**
  - 20+ billion web pages x 20KB = 400+ terabytes
- **One computer can read 30-35 MB/sec from disk**
- **~four months to read the web**
- **~1,000 hard drives just to store the web**
- **Even more to *do something with the data***

Solution: spread the load

- **Good news**
  - Same problem with 1,000 machines, < 3 hours
- **Bad news: programming work**
  - Communication and coordination
  - Recovering from machine failures
  - Status reporting
  - Debugging and optimizing
  - Workload placement
- **Bad news II: repeat for every problem**

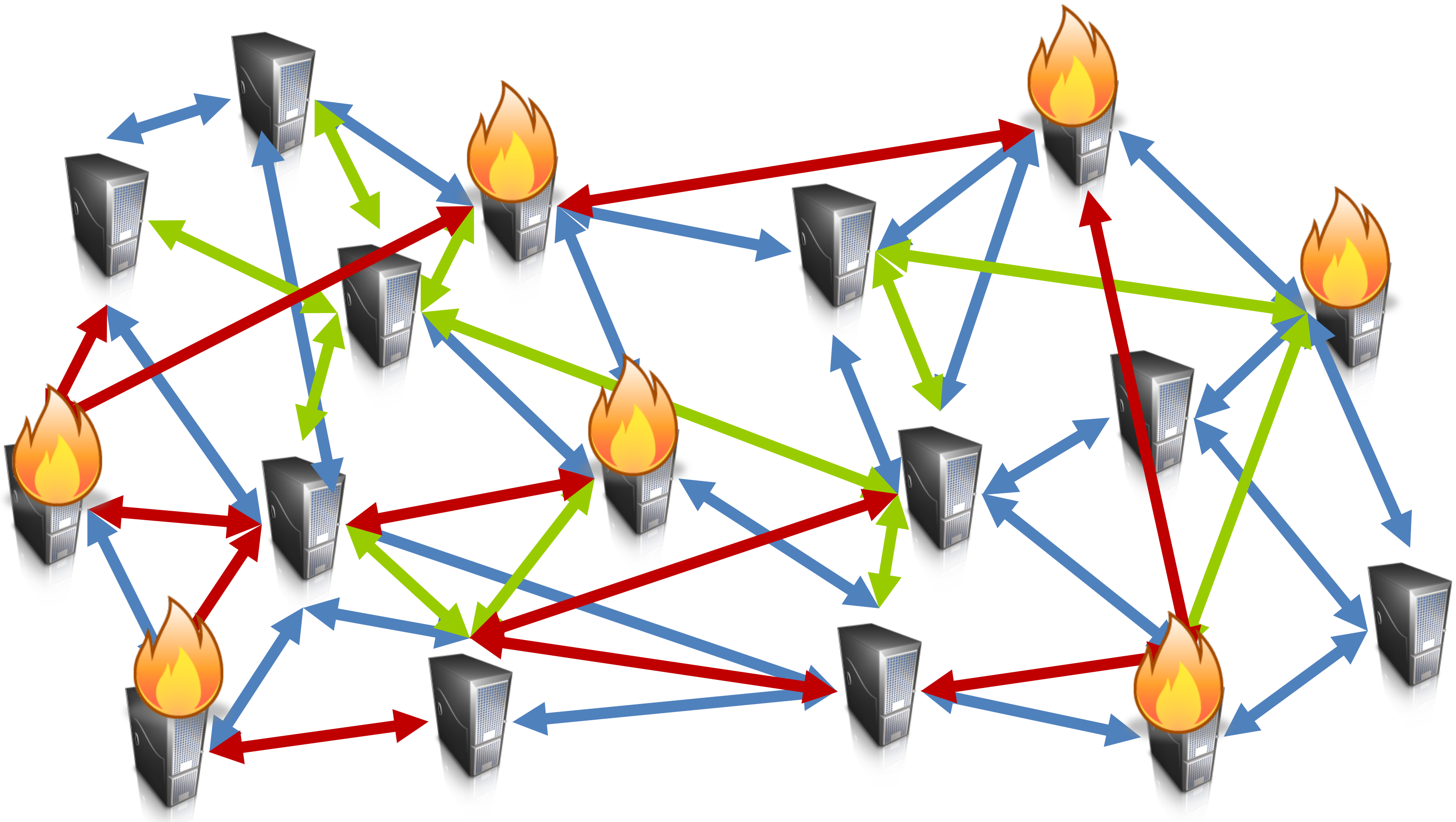
Machine hardware reality

- **Multiple cores**
- **2-6 locally-attached disks**
  - 2TB to ~12 TB of disk
- **Typical machine runs**
  - GFS chunkserver
  - Scheduler daemon for user tasks
  - One or many tasks

Machine hardware reality

- **Single-thread performance doesn't matter**
  - Total throughput/$ more important than peak perf.
- **Stuff breaks**
  - One server may stay up for three years (1,000 days)
  - If you have 10,000 servers, expect to lose 10/day
  - If you have 1,000,000 servers, expect to lose 1,000/day

Google hardware reality



Google storage

- **“The Google File System”**
  - Award paper at SOSP in 2003
- **“Spanner: Google's Globally distributed datastore”**
  - Award paper at OSDI in 2012
- **If you enjoy reading the paper**
  - Sign up for COMPSCI 510 (you’ll read lots of papers like it!)

Google design principles

- **Use lots of cheap, commodity hardware**
- **Provide reliability in software**
- **Scale ensures a constant stream of failures**
  - 2003: > 15,000 machines
  - 2007: > 1,000,000 machines
  - 2012: > 10,000,000?
- **GFS exemplifies how they manage failure**

Sources of failure

- **Software**
  - Application bugs, OS bugs
  - Human errors
- **Hardware**
  - Disks, memory
  - Connectors, networking
  - Power supplies

Design considerations

1. **Component failures**

2. **Files are huge (multi-GB files)**

- Recall that PC files are mostly small
- **How did this influence PC FS design?**
- Relatively small block size (~KB)

Design considerations

1. **Component failures**
2. **Files are huge (multi-GB files)**
3. **Most writes are large, sequential appends**
    - Old data is rarely over-written

Design considerations

1. **Component failures**
2. **Files are huge (multi-GB files)**
3. **Most writes are large, sequential appends**
4. **Reads are large and streamed or small and random**
    - Once written, files are only read, often sequentially
    - **Is this like or unlike PC file systems?**
    - PC reads are mostly sequential reads of small files
    - **How do sequential reads of large files affect client caching?**
    - Caching is pretty much useless

Design considerations

1. **Component failures**
2. **Files are huge (multi-GB files)**
3. **Most writes are large, sequential appends**
4. **Reads are large and streamed or small and random**
5. **Design file system for apps that use it**
    - Files are often used as producer-consumer queues
    - 100s of producers trying to append concurrently
    - Want atomicity of append with minimal synchronization
    - Want support for atomic append

Design considerations

1. **Component failures**
2. **Files are huge (multi-GB files)**
3. **Most writes are large, sequential appends**
4. **Reads are large and streamed or small and random**
5. **Design file system for apps that use it**
6. **High sustained bandwidth better than low latency**
   - **What is the difference between BW and latency?**
   - Network as road (BW = # lanes, latency = speed limit)

Google File System (GFS)

- **Similar API to POSIX**
  - Create/delete, open/close, read/write
- **GFS-specific calls**
  - Snapshot (low-cost copy)
  - Record_append
  - (allows concurrent appends, ensures atomicity of each append)
- **What does this description of record_append mean?**
  - Individual appends may be interleaved arbitrarily
  - Each append's data will not be interleaved with another's

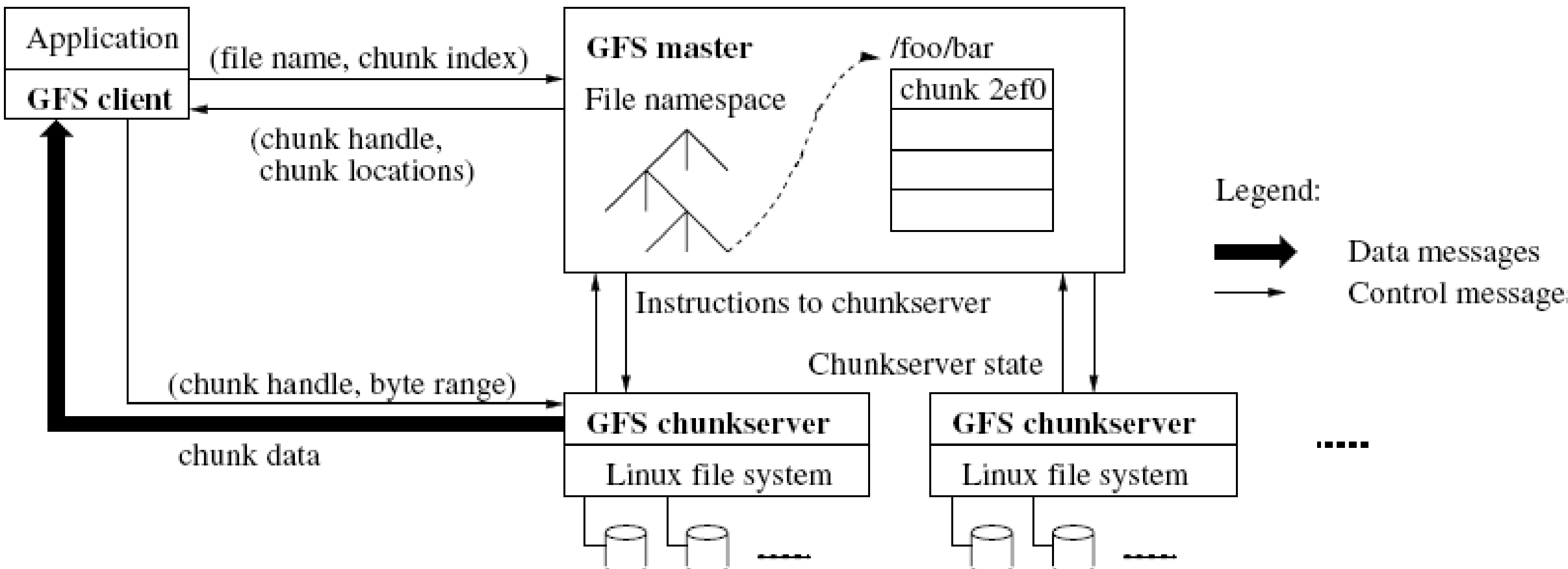
GFS architecture

- **Key features:**
  - Must ensure atomicity of appends
  - Must be fault tolerant
  - Must provide high throughput through parallelism

GFS architecture

- **Cluster-based**
  - Single logical **master**
  - Multiple **chunkservers**
- **Clusters are accessed by multiple clients**
  - Clients are commodity Linux machines
  - Machines can be both clients and servers

GFS architecture



File data storage

- **Files are broken into fixed-size chunks**
- **Chunks are named by a globally unique ID**
  - ID is chosen by the master
  - ID is called a **chunk handle**
- **Servers store chunks as normal Linux files**
- **Servers accept reads/writes with handle + byte range**

File data storage

- **Chunks are replicated at 3 servers**
- **What are the advantages of replication?**
  - Better availability (if one fails, two left)
  - Better read performance (parallel reads)

File data storage

- **Chunks are replicated at 3 servers**
  - Using more than three would waste resources
- **If 4 machines try to be replicas**
  - First 3 should be allowed, 4th should be denied
- **How does this look like a synchronization problem?**
  - Can think of “acting as a chunk’s replica” as critical section
  - Only want three servers in that critical section
- **How did we solve this kind of problem previously?**
  - Semaphores or locks/CVs
  - Ensure that max of 3 threads in critical section

```
Server () {

}
```

```
Lock l;
int num_replicas=0;

Server () {
  l.lock ();
  if (num_replicas < 3) {
    num_replicas++;
    l.unlock ();

    while (1) {
      // do server things
    }

    l.lock ();
    num_replicas--;
  }
  l.unlock ();
  // do something else
}
```

File data storage

- **Chunks are replicated at 3 servers**
  - Using more than three would waste resources
- **Why wouldn't distributed locking be a good idea?**
  - Machines can fail holding a lock
  - Responsibility for chunk cannot be re-assigned

What happens if a thread fails in here?

```
Lock l;
int num_replicas=0;

Server () {
  l.lock ();
  if (num_replicas < 3) {
    num_replicas++;
    l.unlock ();

    while (1) {
      // do server things
    }

    l.lock ();
    num_replicas--;
  }
  l.unlock ();
  // do something else
}
```

File data storage

- **Chunks are replicated at 3 servers**
- **Instead: servers *lease* right to serve a chunk**
  - Responsible for a chunk for a period of time
  - Must renew lease before it expires
- **How does this make failure easier to handle?**
  - If a node fails, its leases will expire
  - When it comes back up, just renew leases
- **What has to be synchronized now between replicas/master?**
  - Time: need to agree on when leases expire
- **How do we ensure that time is synchronized between machines?**
  - Only need a rough consensus (order of seconds)
  - Can use protocol like NTP
  - Spanner is clever: Uses GPS for atomic timestamps

File meta-data storage

- **Master maintains all meta-data**
  - Namespace info
  - Access control info
  - Mapping from files to chunks
  - Current chunk locations

Other master responsibilities

- **Chunk lease management**
- **Garbage collection of orphaned chunks**
  - **How might a chunk become orphaned?**
  - If a chunk is no longer in any file
- **Chunk migration between servers**
- **HeartBeat messages to chunkservers**

Client details

- **Client code is just a library**
  - Similar to File class in java
- **Caching**
  - No in-memory data caching at the client or servers
  - Clients still cache meta-data

Master design issues

- **Single (logical) master per cluster**
  - Master's state is actually replicated elsewhere
  - Logically single because client speaks to one name
  - **Where else have we seen this?**
  - Client communication with Google
  - Request sent to google.com
  - Use DNS tricks to direct request to nearby machine

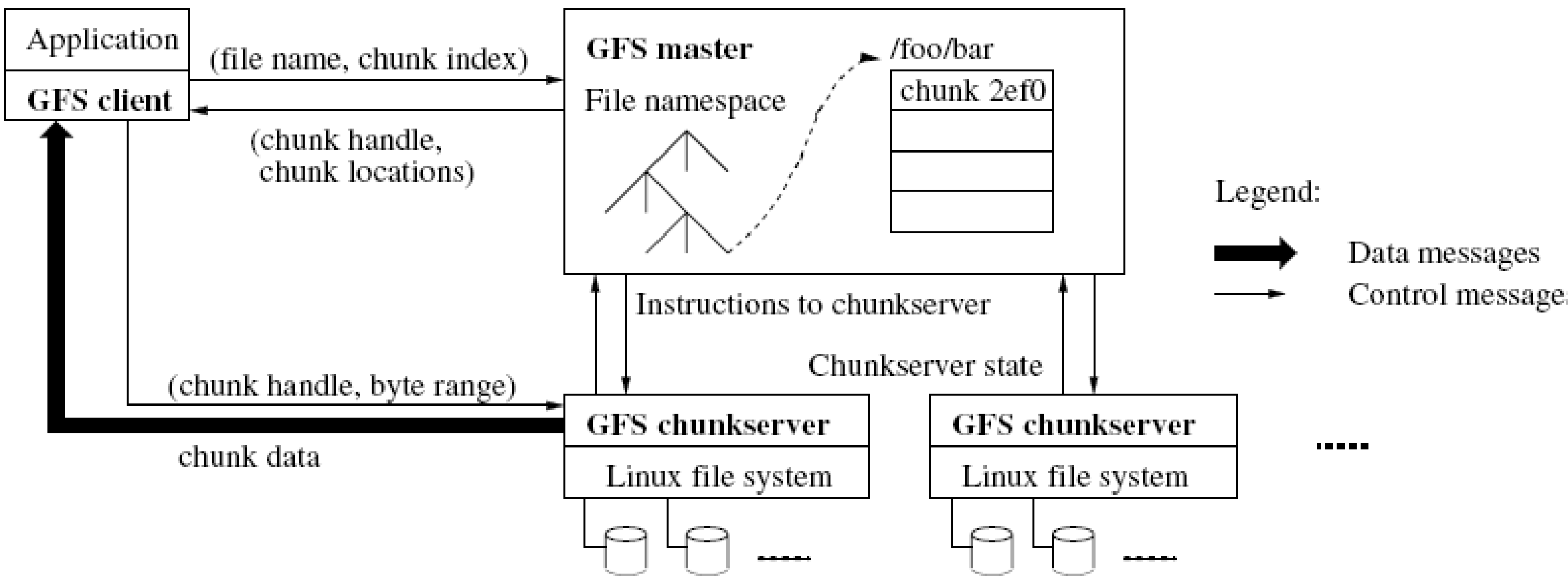
Master design issues

- **Single (logical) master per cluster**
  - Master’s state is actually replicated elsewhere
  - Logically single because client speaks to one name
  - Use DNS tricks to locate/talk to a master
- **Pros**
  - Simplifies design
  - Master endowed with global knowledge
  - (makes good placement, replication decisions)

Master design issues

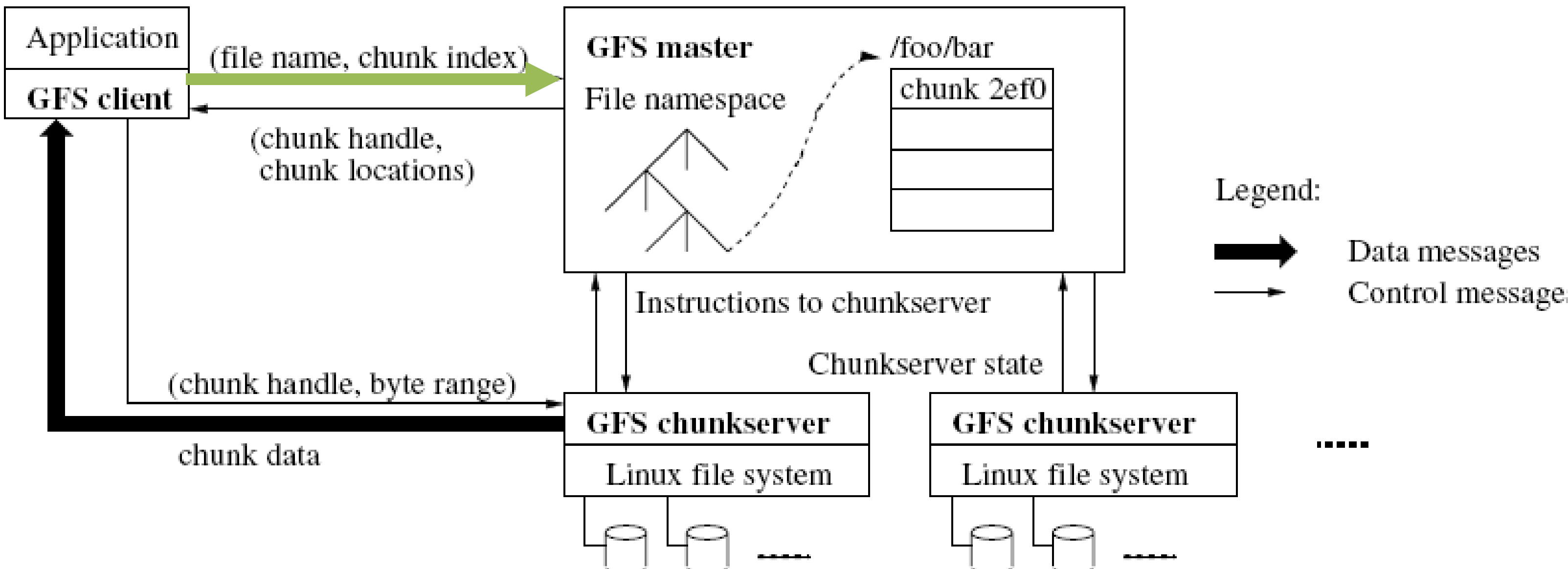
- **Single (logical) master per cluster**
  - Master’s state is actually replicated elsewhere
  - Logically single because client speak to one name
- **Cons?**
  - Could become a bottleneck
  - (recall how replication can improve performance)
  - **How to keep from becoming a bottleneck?**
  - Minimize its involvement in reads/writes
  - Clients talk to master very briefly
  - Most communication is with chunkservers

Example read



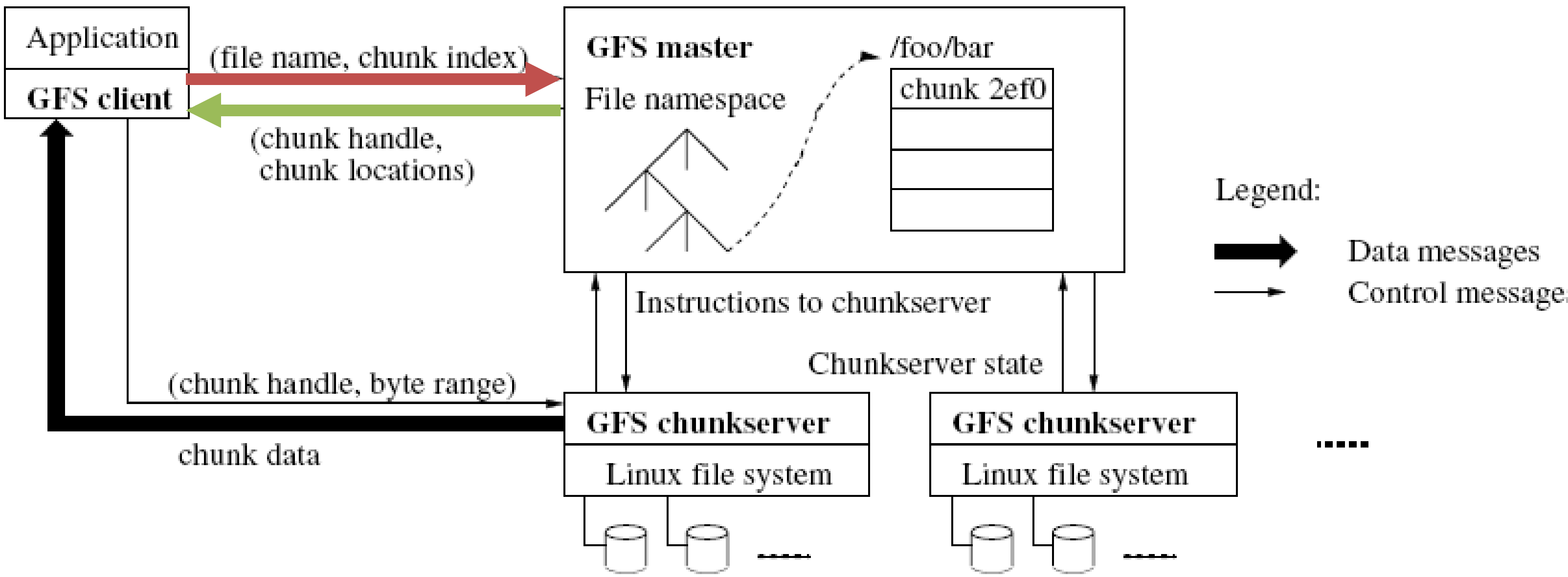
Client uses fixed size chunks to compute chunk index within a file

Example read



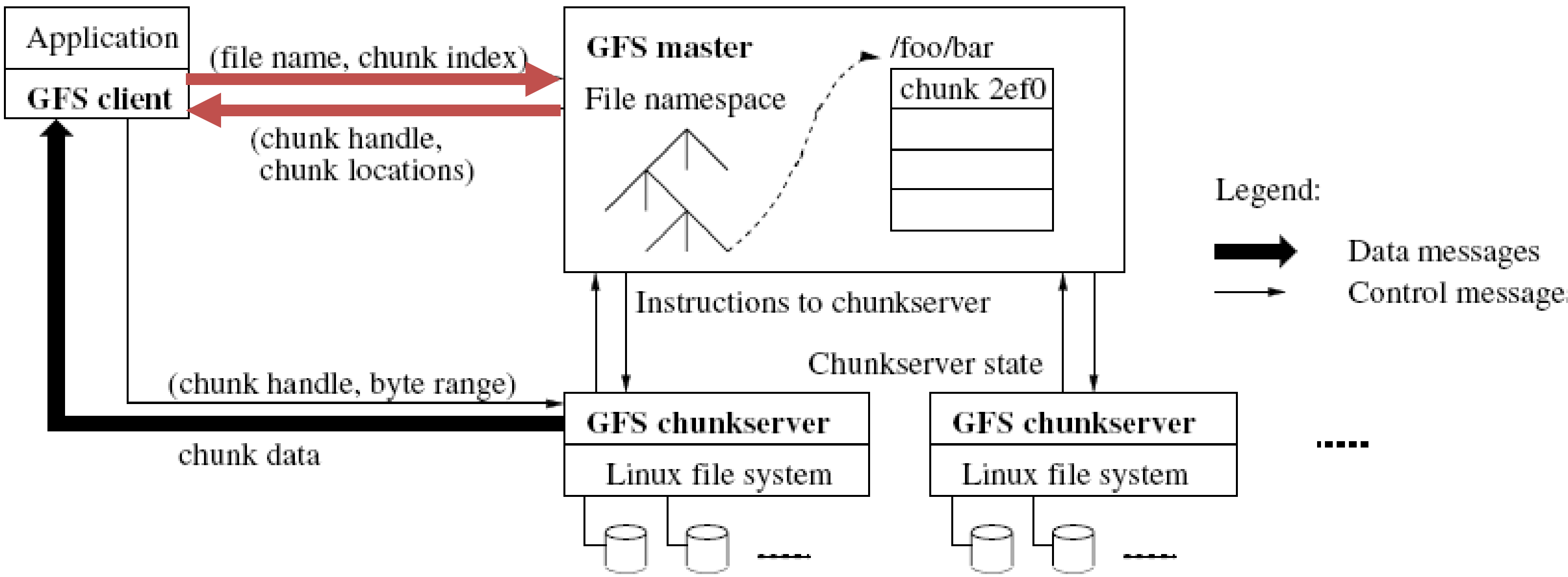
- **Client asks master for the chunk handle at index i of the file**

Example read



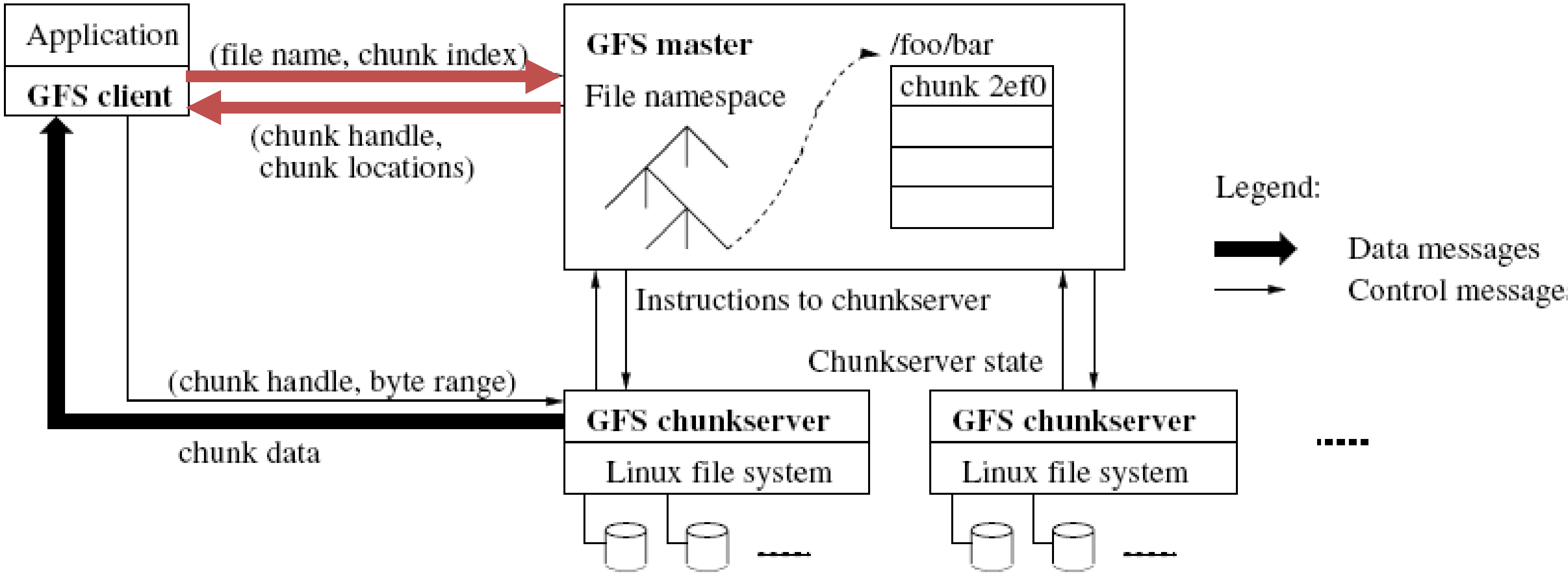
- **Master replies with the chunk handle and list of replicas**

Example read



- **Client caches handle and replica list**
- **(maps filename + chunk index → chunk handle + replica list)**

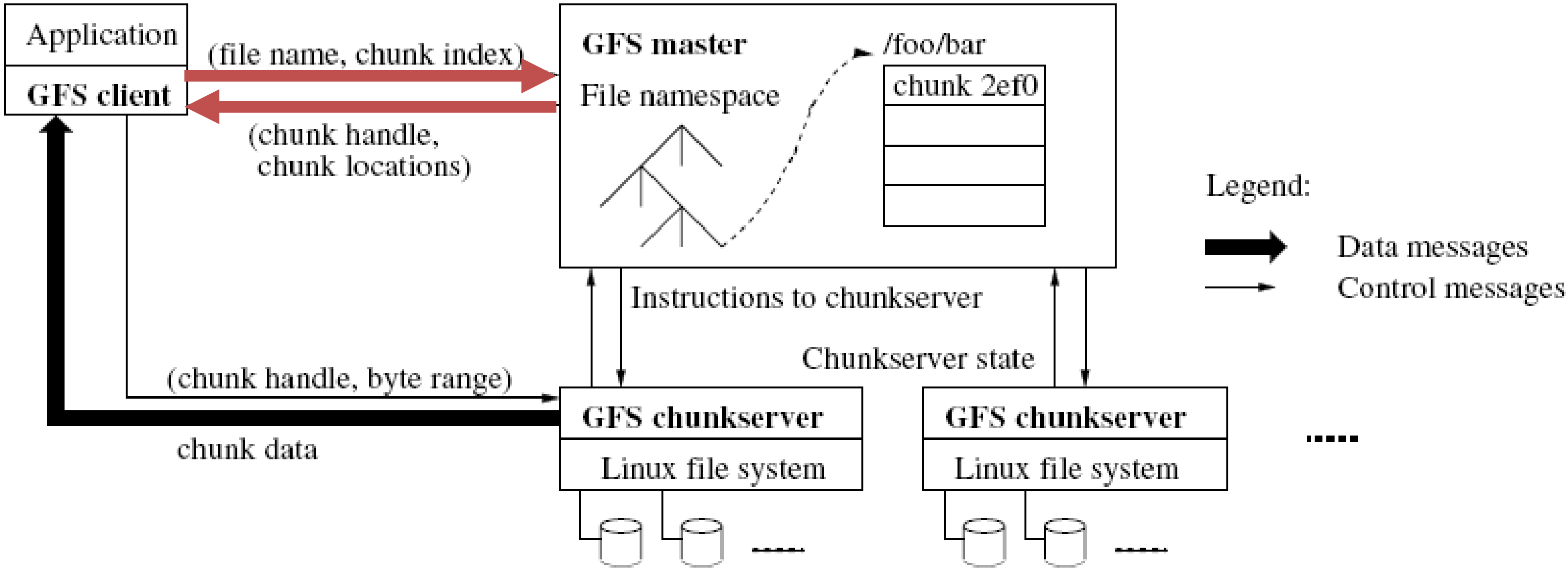
Example read



Client sends a request to the closest chunk server

Server returns data to client

Example read



- **Can you think of any possible optimizations?**
  - Could ask for multiple chunk handles at once (batching)
  - Server could return handles for subsequent indices (pre-fetching)

Chunk size

- **Recall how we chose block/page size?**
- **What are the disadvantages of small/big chunks?**
    - If too small, too much storage used for meta-data
    - If too large, too much internal fragmentation
- **Impact of chunk size on client's meat-data caching?**
    - Data chunks are not cached (so no impact there)
    - Large chunks → less meta-data/chunk
    - Clients can cache more meta-data at clients
    - Masters can fit all meta-data in memory
    - Much faster than retrieving from disk

Chunk size

Recall how we chose block/page sizes

What are the disadvantages of small/big chunks?

If too small, too much storage used for meta-data

If too large, too much internal fragmentation

What is a reasonable chunk size then?

Big?

They chose 64 MB

Reasonable when most files are many GB

Master’s state

1. **File and chunk namespaces**
2. **Mapping from files to chunks**
3. **Chunk replica locations**
4. **All are kept in-memory**
    - **1. and 2. are kept persistent**
    - Use an **operation log**

Operation log

- **Historical record of all meta-data updates**
- **Only persistent record of meta-data updates**
- **Replicated at multiple machines**
    - Appending to log is transactional
    - Log records are synchronously flushed at all replicas
    - To recover, the master replays the operation log
- **What this means for master performance**
    - State updates will be slow (order of 10s of ms)
- **Why is this OK?**
    - Updates to namespaces and chunk mappings are relatively infrequent
    - Log writes not in critical path of data updates

Atomic record_append

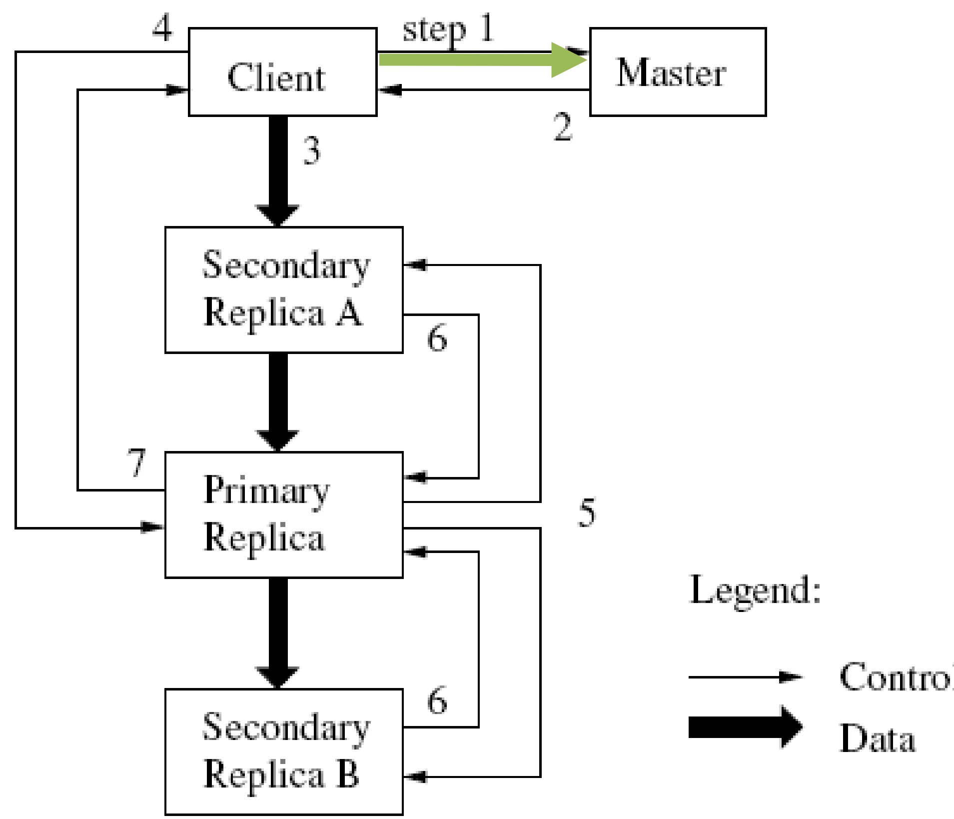
- **How are concurrent file writes conventionally treated?**
  - Concurrent writes to same file region are not serialized
  - Region can end up containing fragments from many clients
- **Record_append**
  - Client only specifies the data to append
  - GFS appends it to the file at least once atomically
  - GFS chooses the offset
- **Why is this simpler than forcing clients to synchronize?**
  - Clients would need a distributed locking scheme
  - GFS provides an abstraction, hides concurrency issues from clients
- **Where else have we seen Google hide synchronization?**
  - Map-Reduce programs

Mutation order

- **Mutations are performed at each chunk’s replica**
- **Master chooses a primary for each chunk**
  - Others are called **secondary** replicas
- **Primary chooses an order for all mutations**
  - Called “serializing”
- **All replicas follow this “serial” order**

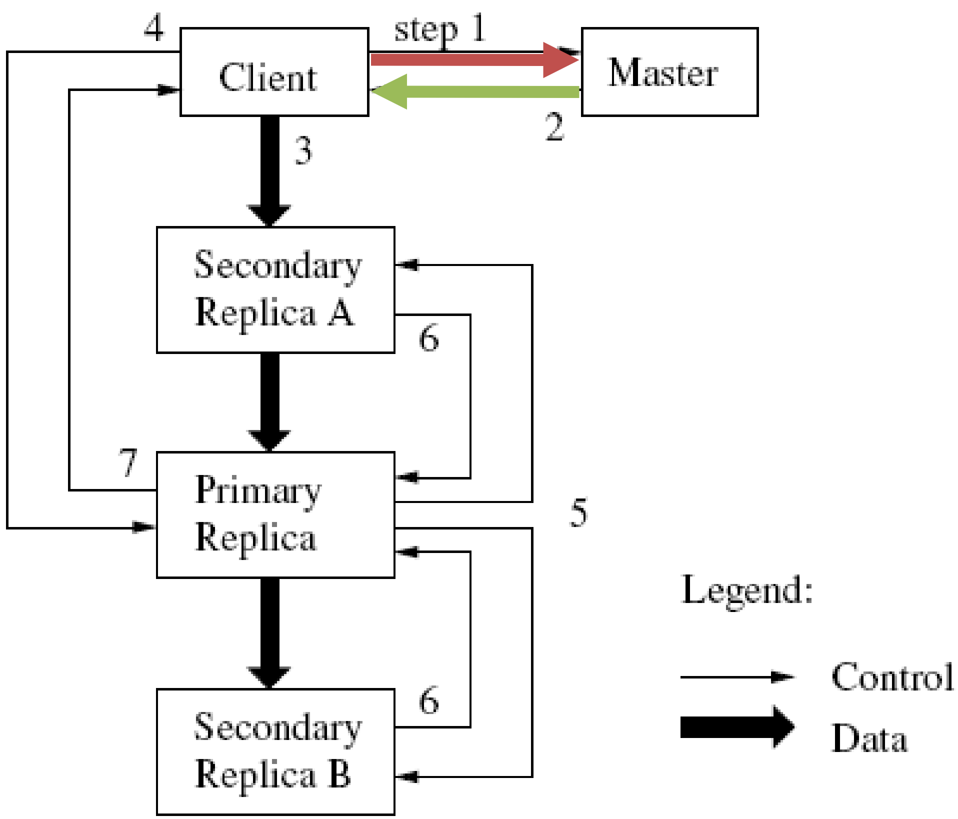
Example mutation

- **Client asks master**
  - Primary replica
  - Secondary replicas



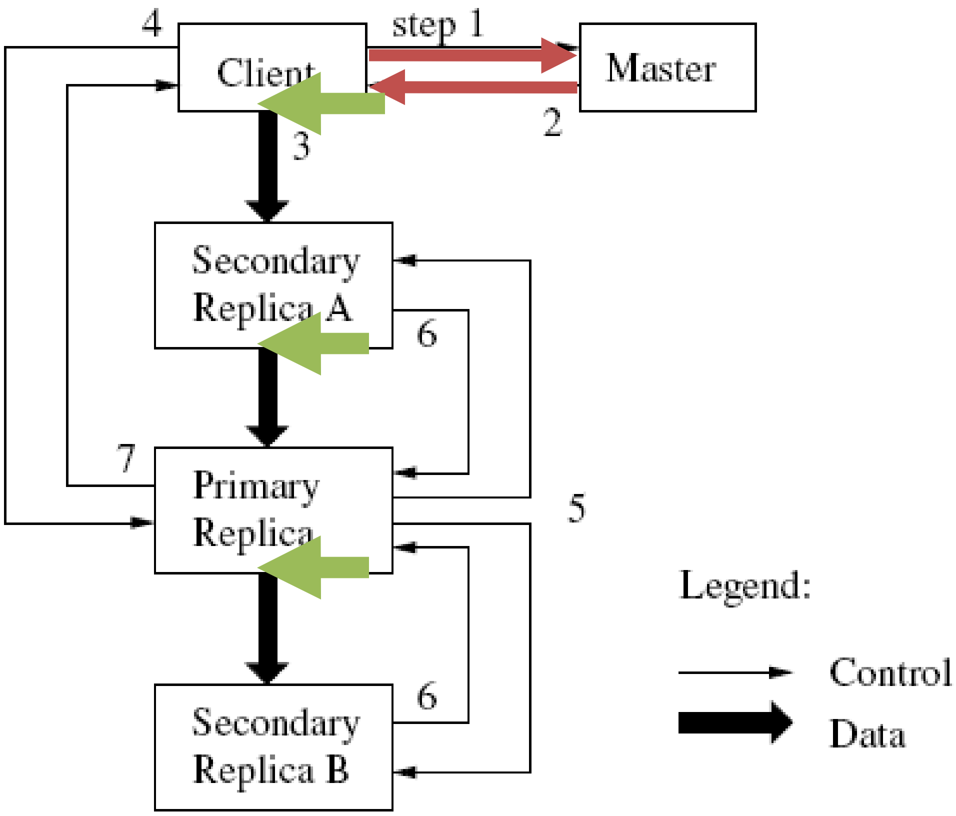
Example mutation

- **Master returns**
  - Primary replica
  - Secondary replicas



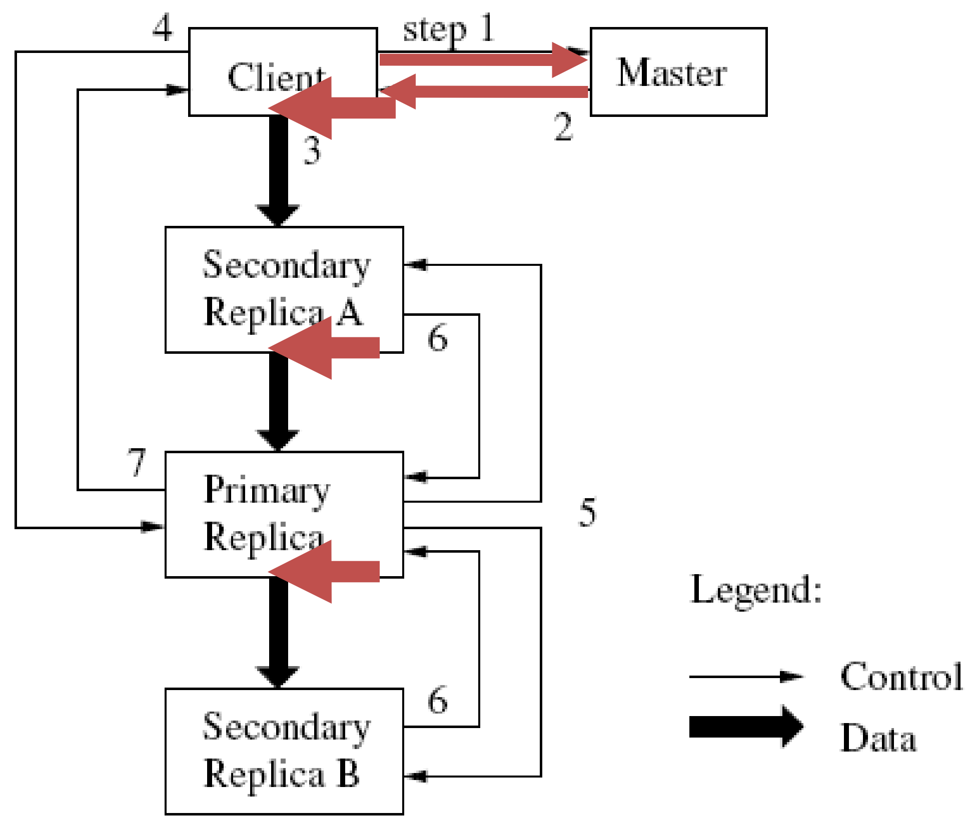
Example mutation

- **Client sends data**
  - To all replicas
- **Replicas**
  - Only buffer data
  - Do not apply
  - Ack client



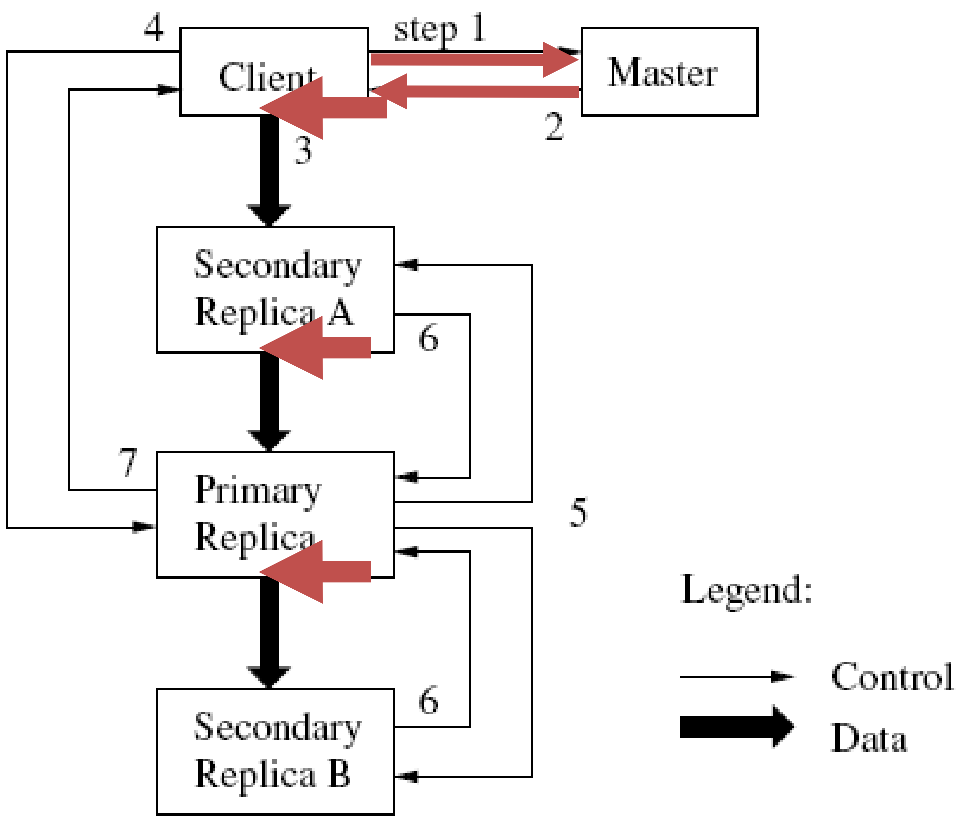
Example mutation

- **Client tells primary**
  - Write request
  - Identifies sent data
- **Primary replica**
  - Assigns serial #s
  - Writes data locally
  - (in serial order)



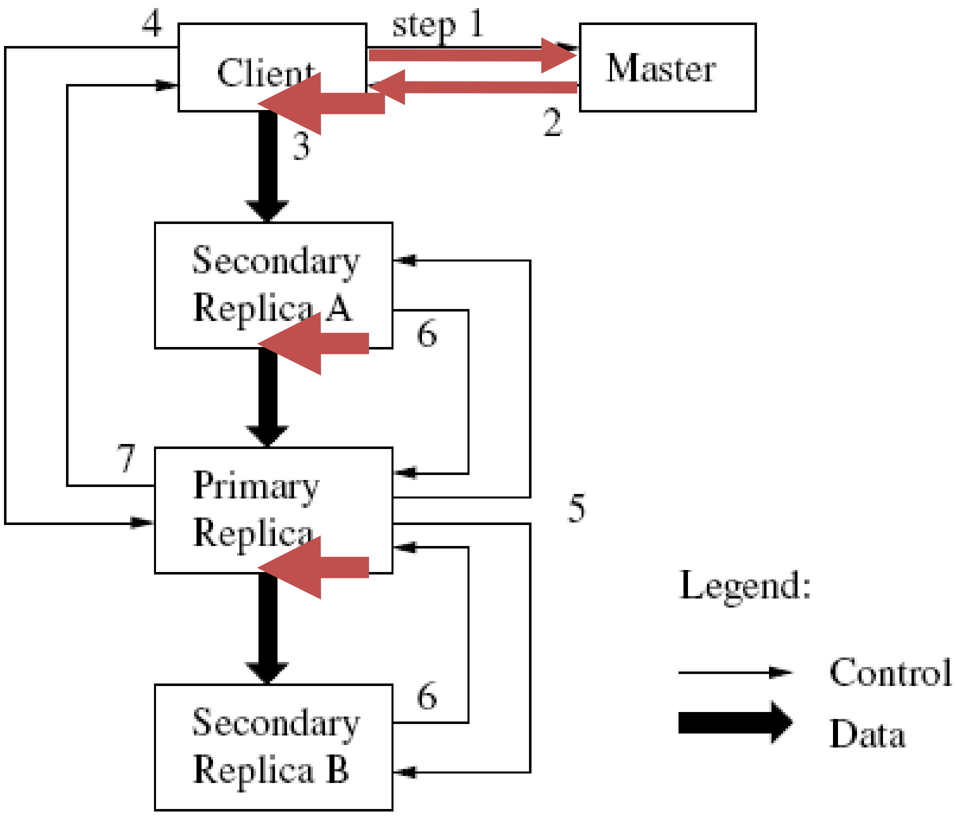
Example mutation

- **Primary replica**
  - Forwards request
  - to secondaries
- **Secondary replicas**
  - Write data locally
  - (in serial order)



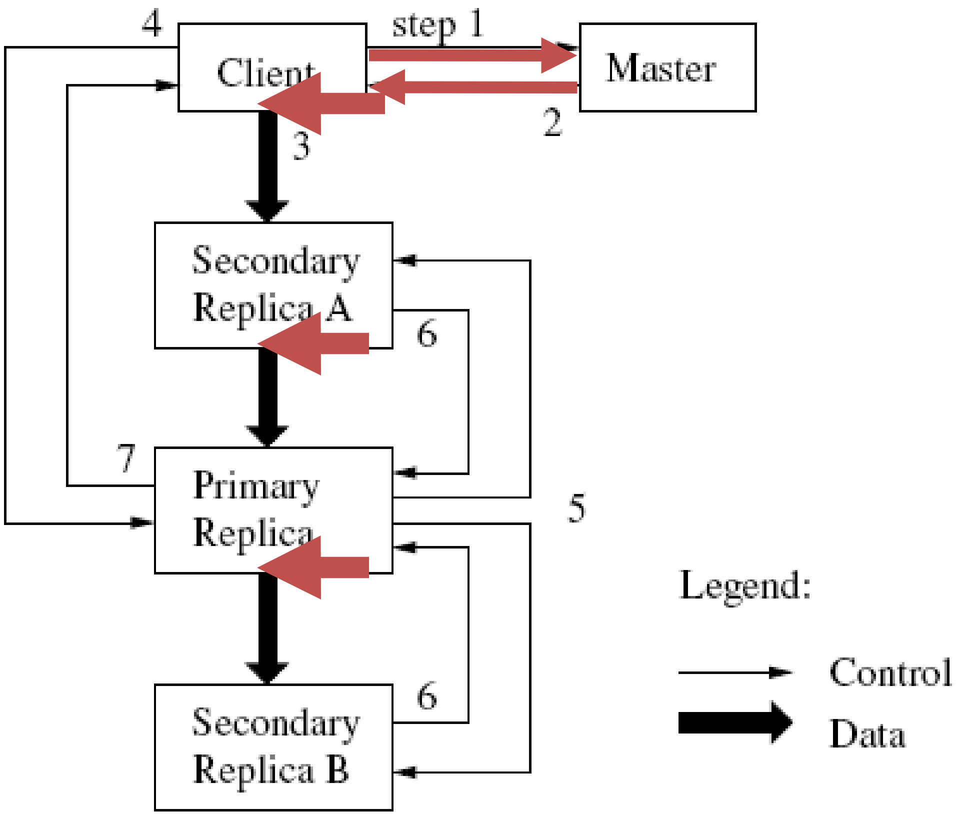
Example mutation

- **Secondary replicas**
  - Ack primary
  - Like “votes”



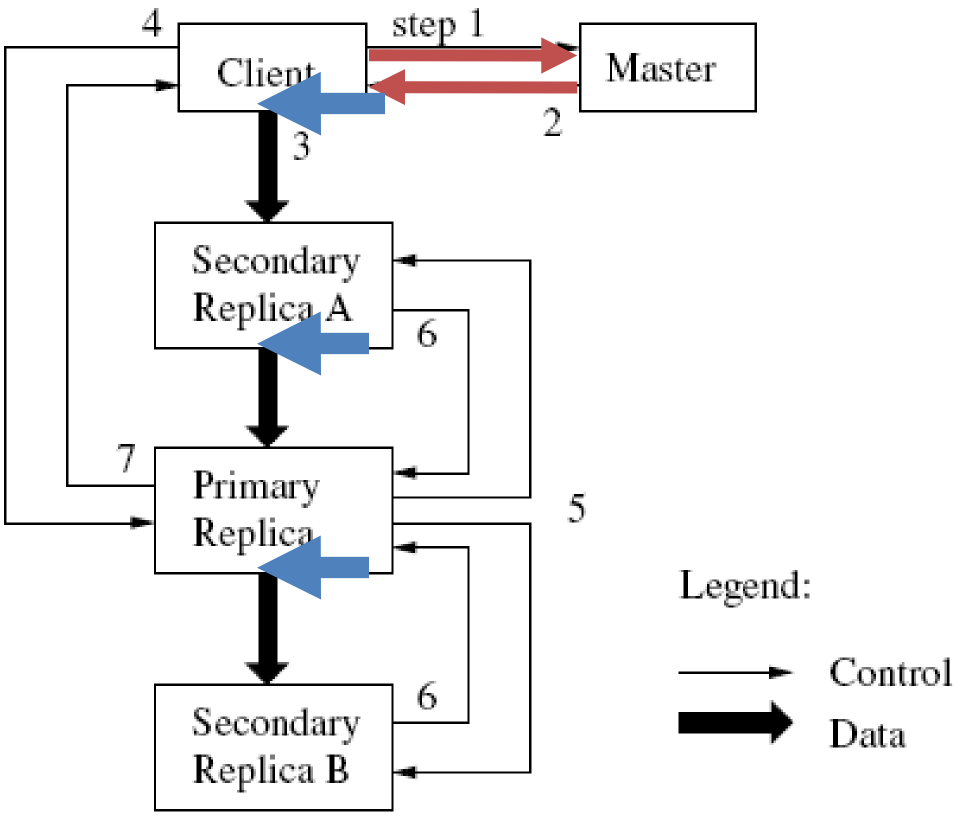
Example mutation

- **Primary replica**
  - Ack client
  - Like a commit



Example mutation

- **Errors?**
  - Require consensus
  - Just retry



Other approaches to storage

- **Distributed data structures**
  - Have seen some of this with the DNS tree
  - Will now look at hash tables (i.e., DHTs)
- **Distributed hash tables**
  - Provide the foundation for many key-value stores
  - Found in p2p systems, big cloud stores, etc.

Map-Reduce

- **Widely applicable, simple way to program**
- **Hides lots of messy details**
  - Automatic parallelization
  - Load balancing
  - Network/disk transfer optimization
  - Handling of machine failures
  - Robustness

Typical MapReduce problem

1. **Read a lot of data (TBs)**
2. **Map**
   - Extract something you care about from each record
1. **Shuffle and sort Map output**
2. **Reduce**
   - Aggregate, summarize, filter or transform sorted output
1. **Write out the results**

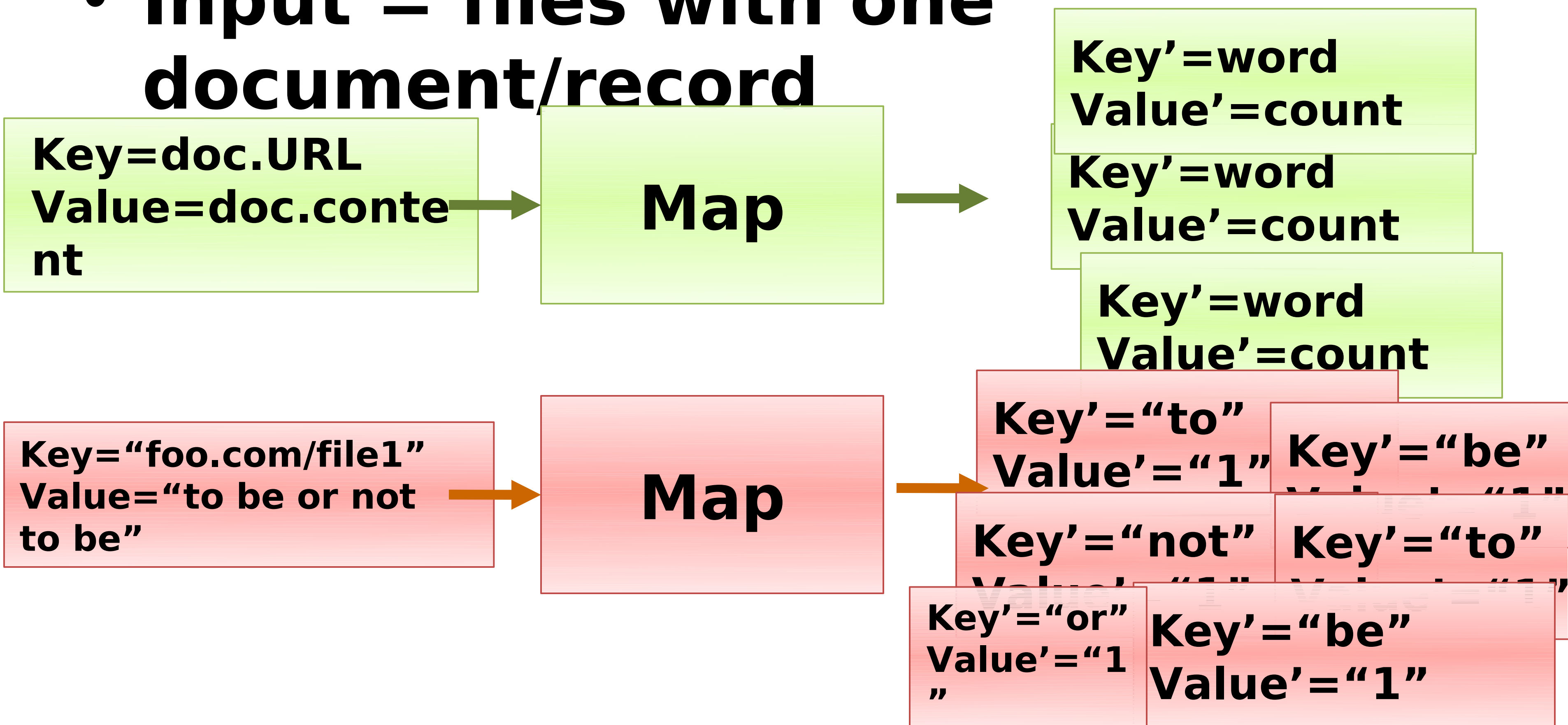
Outline remains the same, only change the map and reduce functions

More specifically

- **Programmer specifies two main methods**
  - `Map (k,v) → <k’, v’>*`
  - `Reduce (k’, <v’>*) → <k’, v’>*`
- **All v’ and k’ are reduced together, in order**
- **Usually also specify**
  - `Partition(k’, total partitions) → partition for k’`
  - Often a simple hash of the key

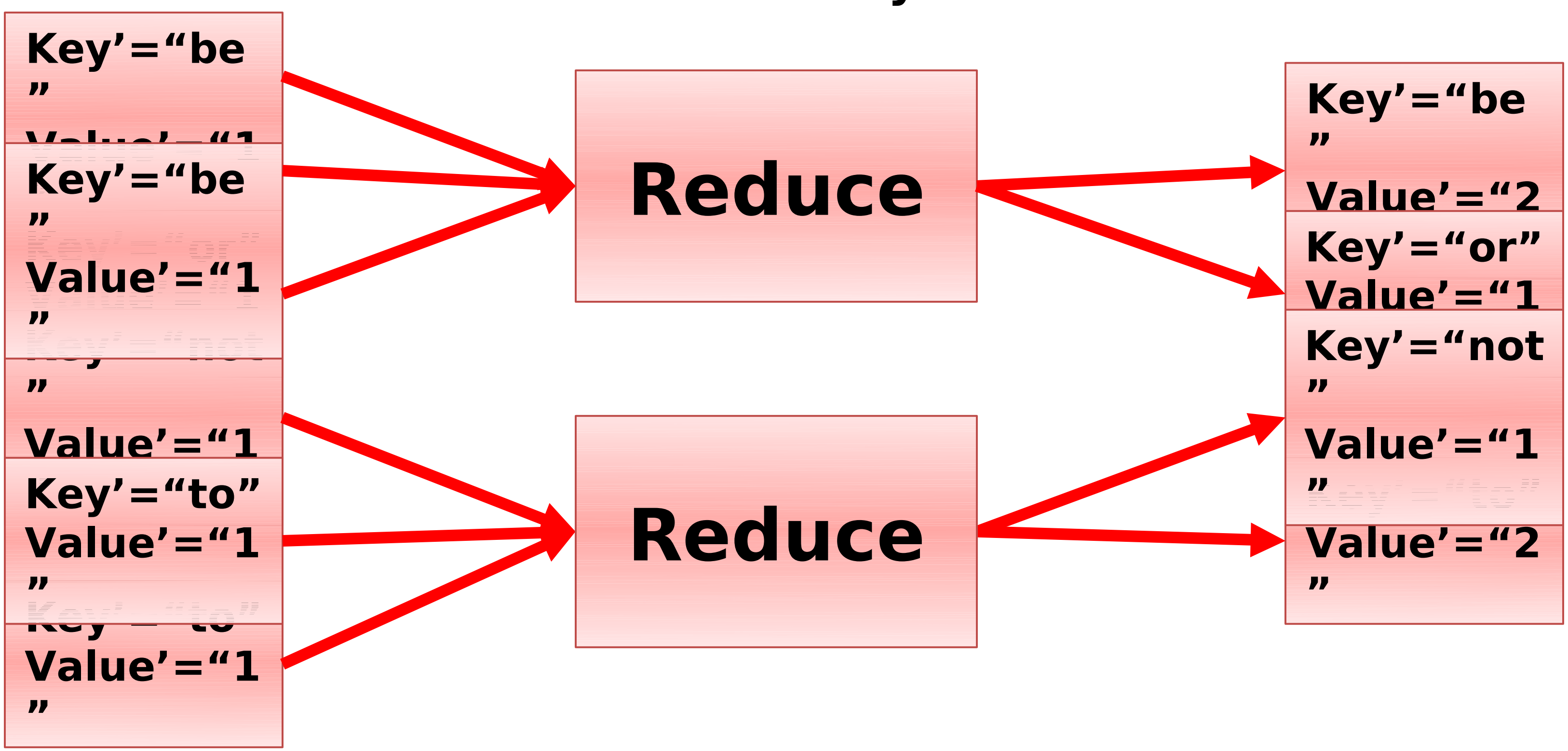
Example

- **Word frequencies in web pages**
- **Input = files with one document/record**



Example continued

- **MapReduce lib gathers all pairs with same key**
  - (shuffle and sort)
- **Reduce combines values for a key**



Example pseudo-code

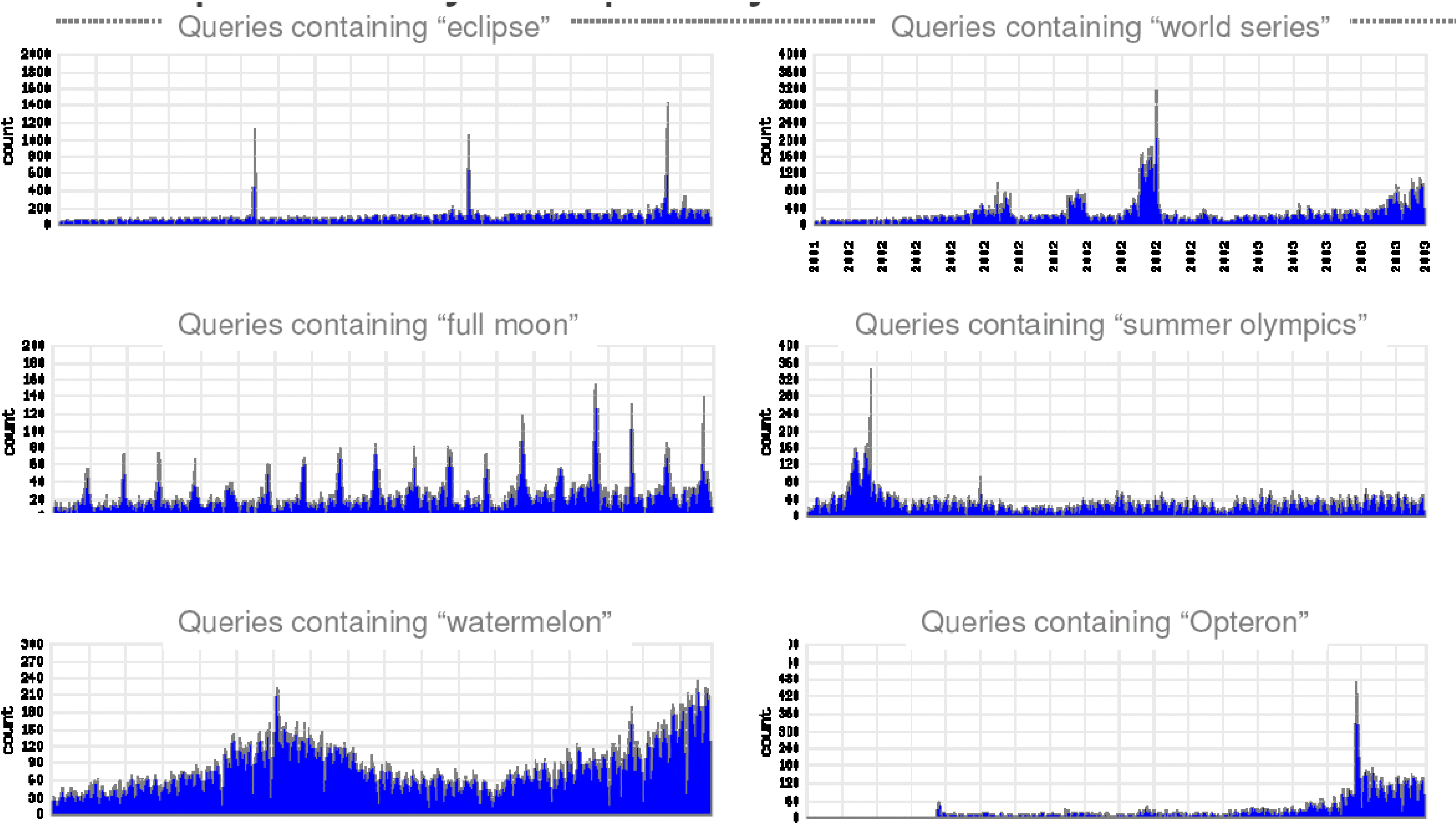
```
Map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_values:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
  // key: a word, same for input and output
  // intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Widely applicable at Google

- **Implemented as a C++ library**
  - Linked to user programs
  - Can read and write many data types

distributed grep	**web access log stats**
distributed sort	**web link-graph reversal**
term-vector per host	**inverted index construction**
document clustering	**statistical machine translation**
machine learning	

Example: query freq. over time



Example: language model stats

- **Used in machine learning translation**
  - Need to count # of times every 5-word sequence occurs
  - Keep all those where count >= 4
- **Easy with MapReduce:**
  - Map: extract 5-word sequences → count from document
  - Reduce: combine counts, write out count if large enough

Example: joining with other data

- **Generate per-doc summary**
  - Include per-host info
  - E.g., # of pages on host, important terms on host
- **Easy with MapReduce:**
  - Map
  - Extract hostname from URL
  - Lookup per-host info
  - Combine with per-doc data and emit
  - Reduce
  - Identity function (just emit key/value directly)

MapReduce architecture

- **How is this implemented?**
- **One master, many workers**
  - Input data split into M map tasks (64MB each)
  - Reduce phase partitioned into R reduce tasks
  - Tasks are assigned to workers dynamically
  - Often: M=200,000; R=4,000; workers=2,000

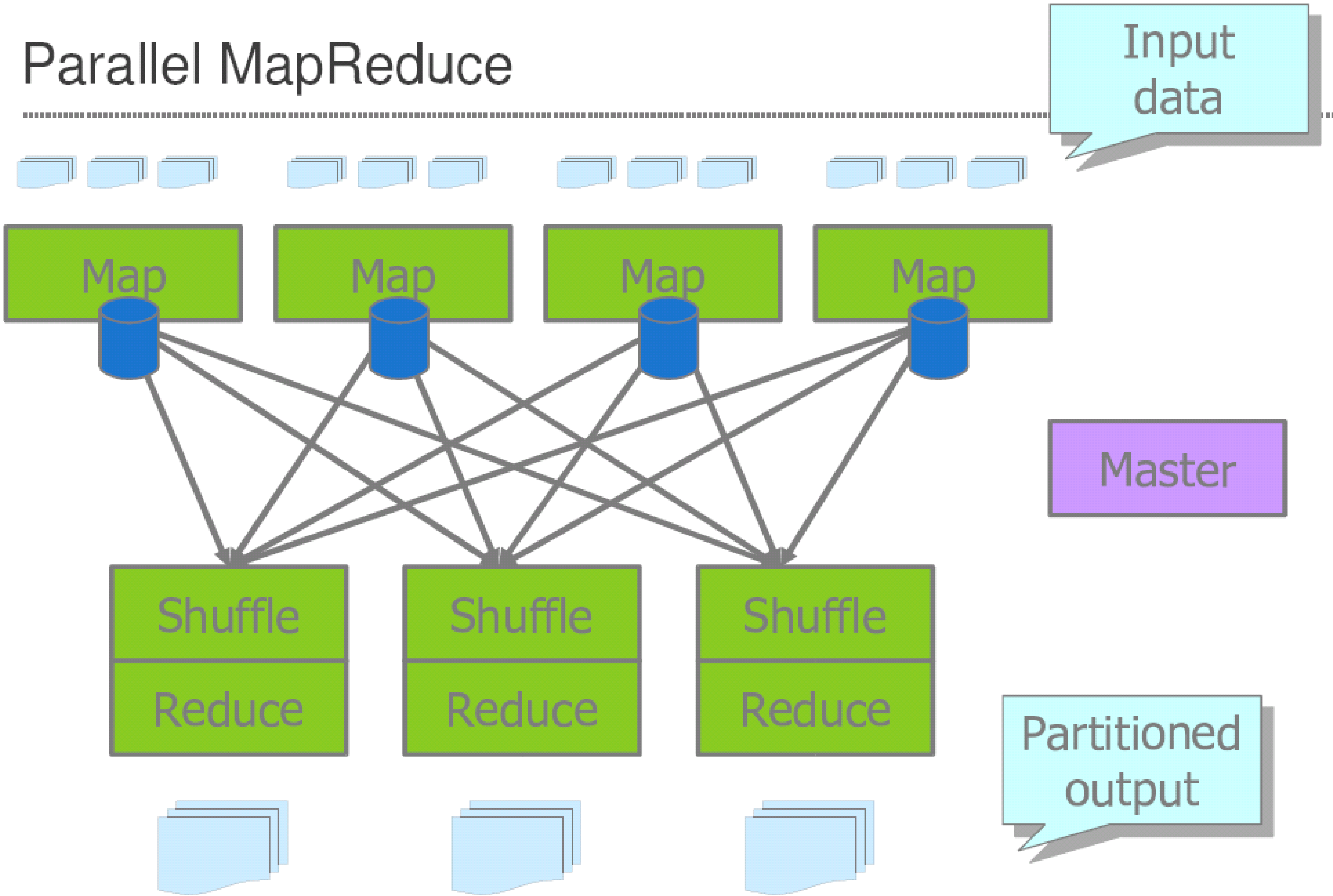
MapReduce architecture

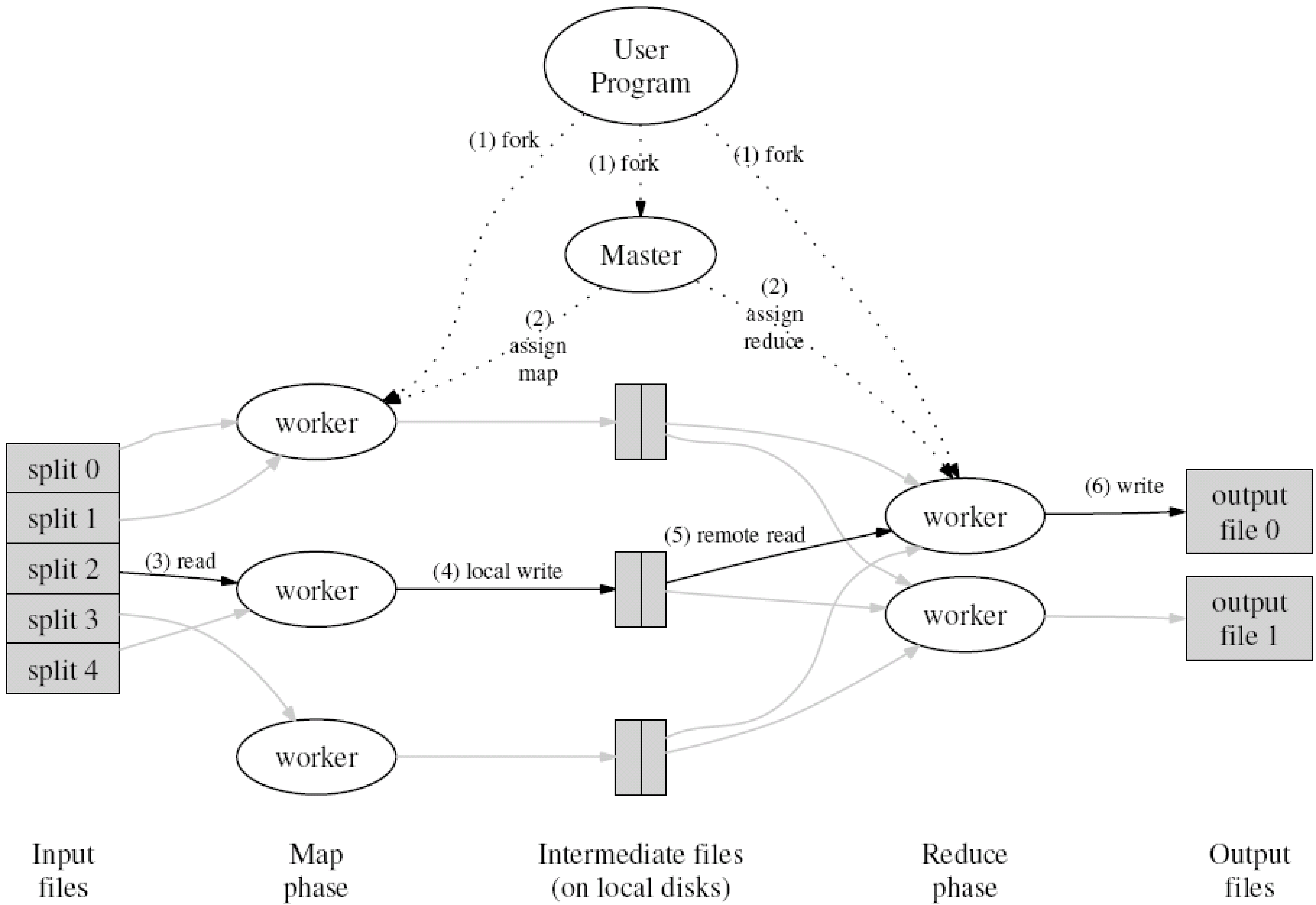
- **Why is a single coordinator (master) nice?**
  - Reduces complexity
  - Can monitor progress and status from one logical place
- **Why use multiple workers?**
  - Take advantage of parallelism
- **Useful approach**
  - Centralize coordination
  - De-centralize heavy lifting

MapReduce architecture

1. **Master assigns each map to a free worker**
    - Considers locality of data to worker
    - Worker reads task input (often from local disk)
    - Worker produces R local files with k/v pairs
1. **Master assigns each reduce task to a free worker**
    - Worker reads intermediate k/v pairs from map workers
    - Worker sorts & applies user’s Reduce op to get output

Parallel MapReduce




User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce fault tolerance

- **What is the downside of a centralized Master?**
  - Can become a single point of failure
- **Worry about it becoming a performance bottleneck?**
  - Not really
  - Master isn’t in the critical path for heavy lifting
  - Just there to make sure everything runs smoothly
- **How can we recover from a Master failure?**
  - Log state transformations to Google File System
  - New master uses log to recover and continue
  - Same idea as transactions covered in storage lectures

MapReduce fault tolerance

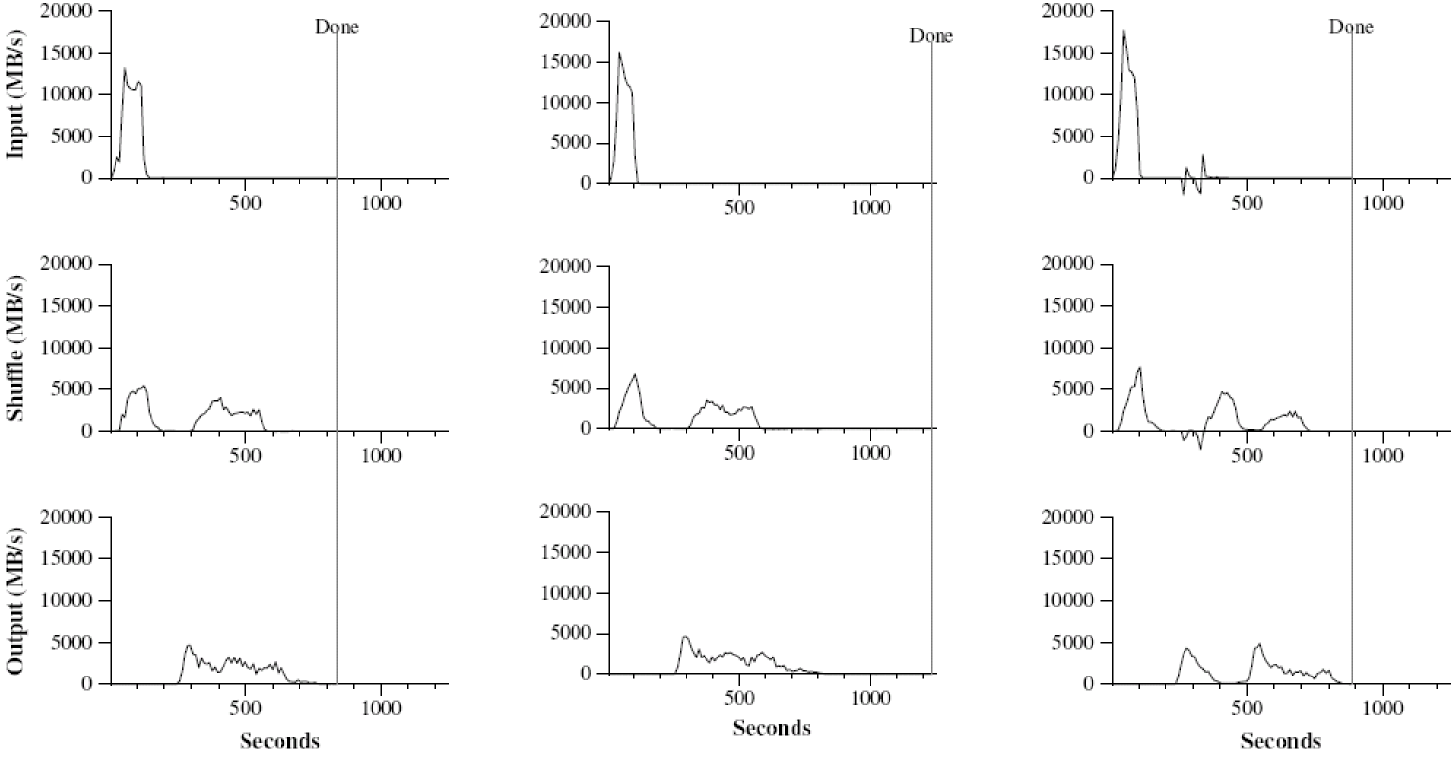
- **How likely is master to fail?**
  - Not likely
  - Individual machine can run for three years
  - P(node failure)
- **How likely is it that at least one worker will fail?**
  - Very likely
  - For N workers
  - 1 – P(no nodes fail)
  - = 1 – (P(worker1 doesn’t fail)*…*P(workerN doesn’t fail))
  - = 1 – ((1-P(worker1 failure))*… *(1-P(worker1 failure)))
  - = 1 – (1-P(node failure))^N

Failure exponentially more likely as N grows!!

MapReduce fault tolerance

- **Worker failures handled via re-execution**
- **On worker failure:**
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in-progress reduce tasks
  - Task completion committed through master

MapReduce performance



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Sort 10^10 100-byte records (~1TB) in ~10.5 minutes.
50 lines of C++ code running on 1800 machines.