Distributed Systems I: MapReduce, Google File System

CS 571: Operating Systems (Spring 2022)

Lecture 11

Yue Cheng

Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

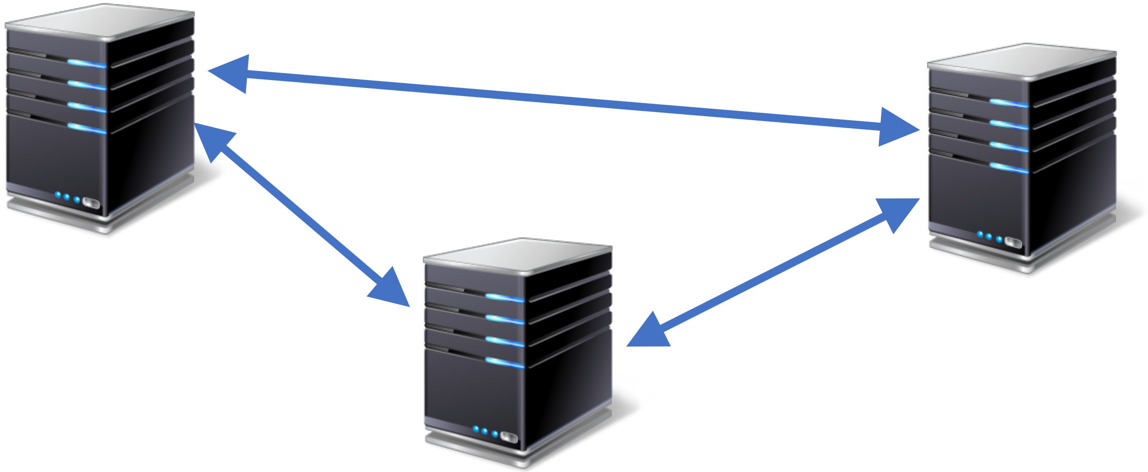
Announcement

- Grade of mini exam 2 released on BB

- Project presentation video due in two weeks
  - Make sure your video is ready by Monday, May 2

- If you prefer to do an online demo (Friday, May 6), let me know
  - We can only schedule 4-5 teams in the online session, so FCFS; rest of 9-10 teams will do the in-classroom demo on Wednesday

What is a distributed system?



- Multiple computers
- Connected by a network
- Doing something together

- A *distributed system* is many cooperating computers that appear to users as a single service

Today's outline

How can large computing jobs be parallelized?

1. MapReduce

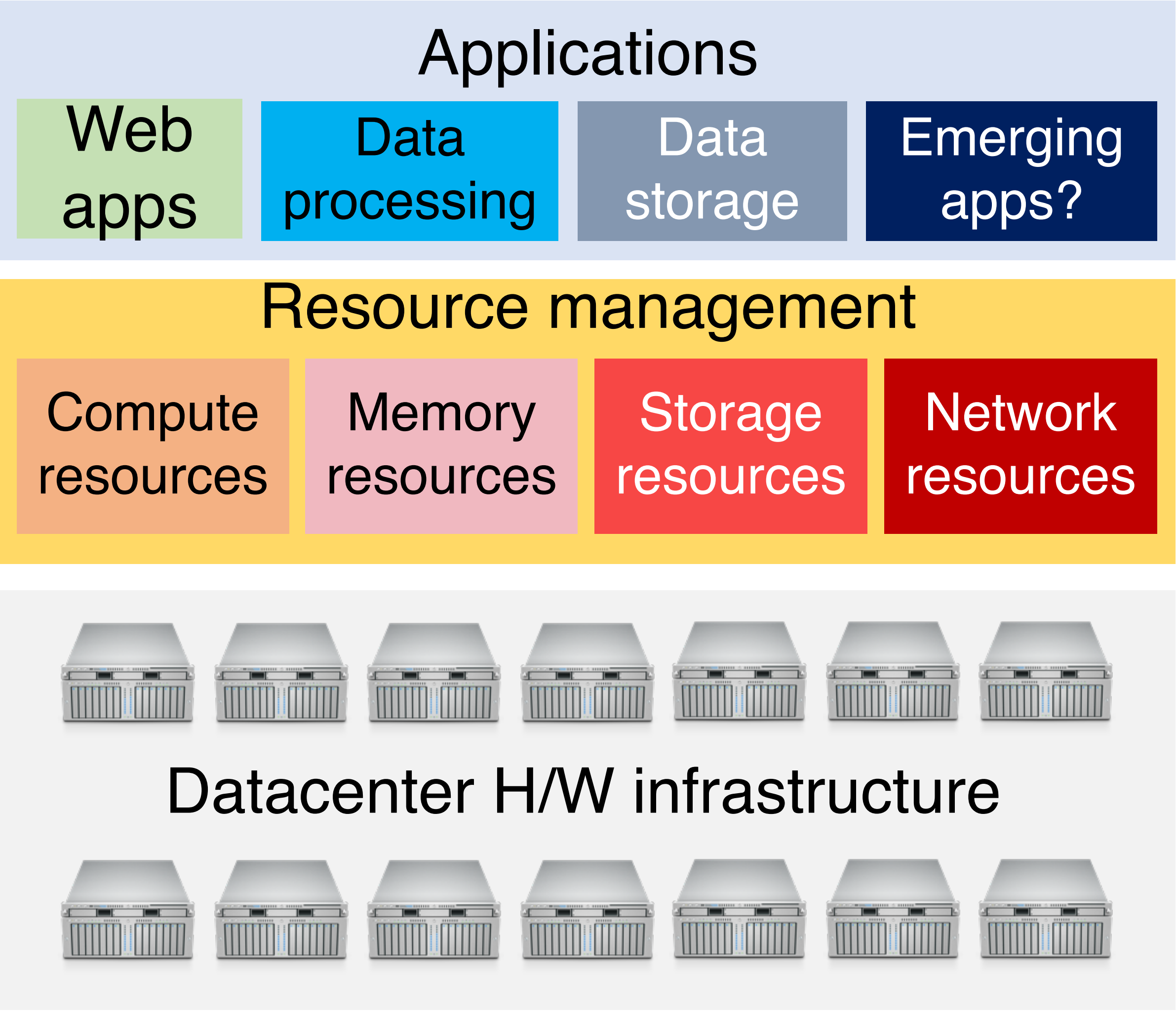
2. Google File System

Today's outline

How can large computing jobs be parallelized?

1. MapReduce

2. Google File System



Applications

Web apps

Data processing

Data storage

Emerging apps?

Resource management

Compute resources

Memory resources

Storage resources

Network resources

Datacenter H/W infrastructure

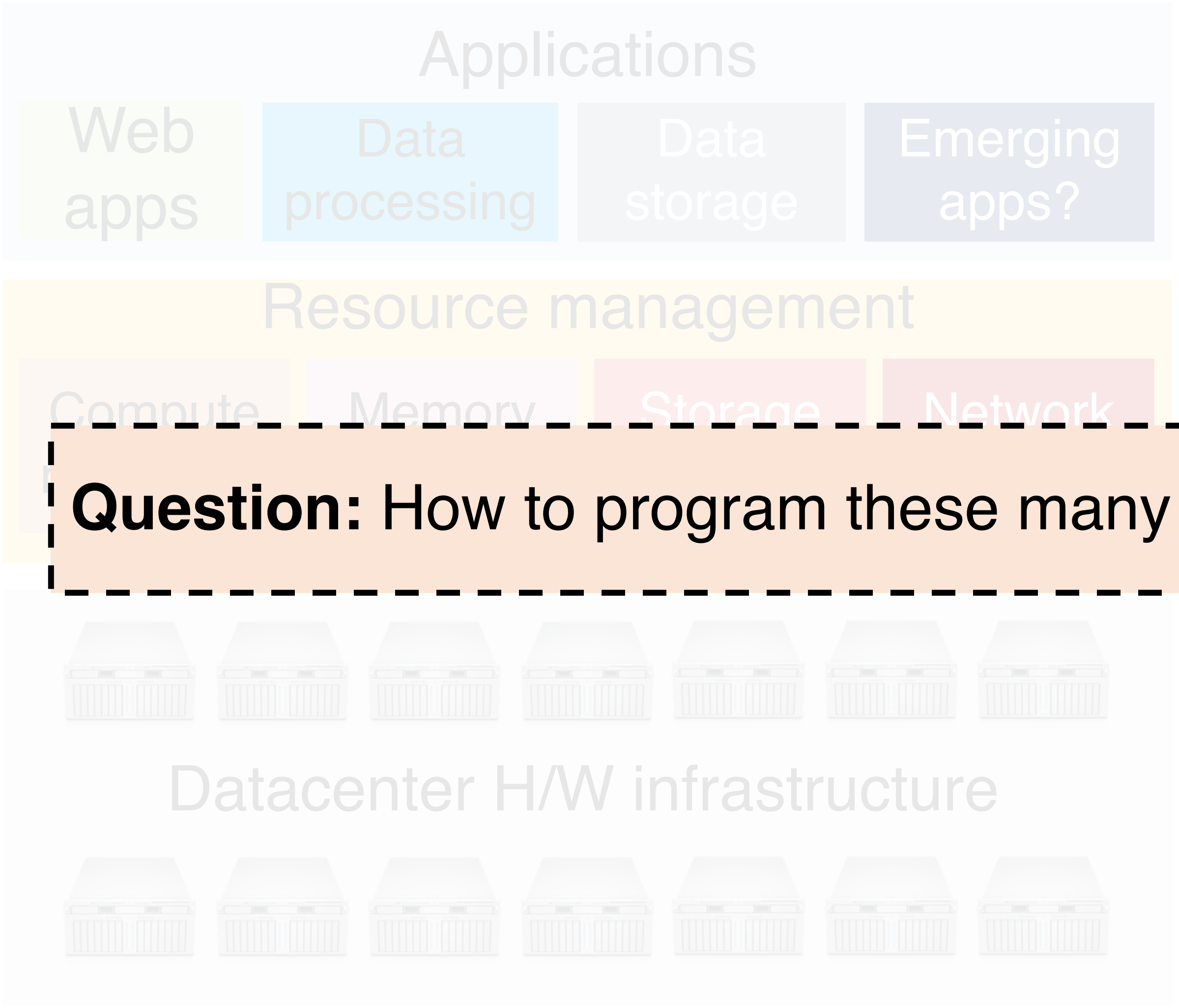
OS. Apps.

Cluster OS.

D/C OS.

OS { Sched. mem storage N/W.

H/W.



Applications

Web apps

Data processing

Data storage

Emerging apps?

Resource management

Compute

Memory

Storage

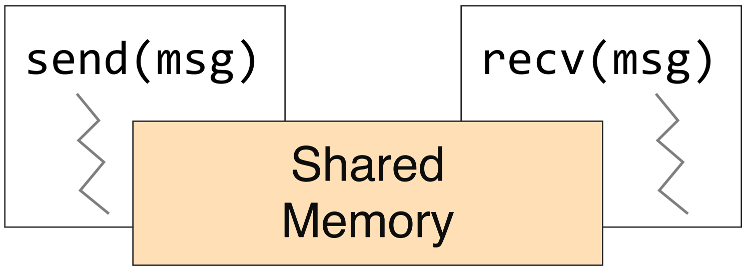
Network

Datacenter H/W infrastructure

Question: How to program these many computers?

Review: Shared memory

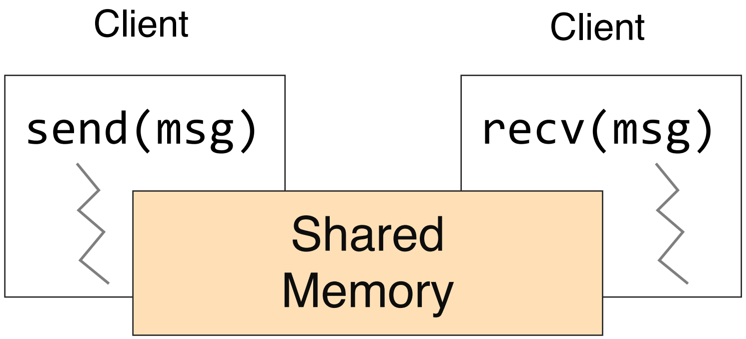
DSM



- Shared memory: multiple processes to share data via memory

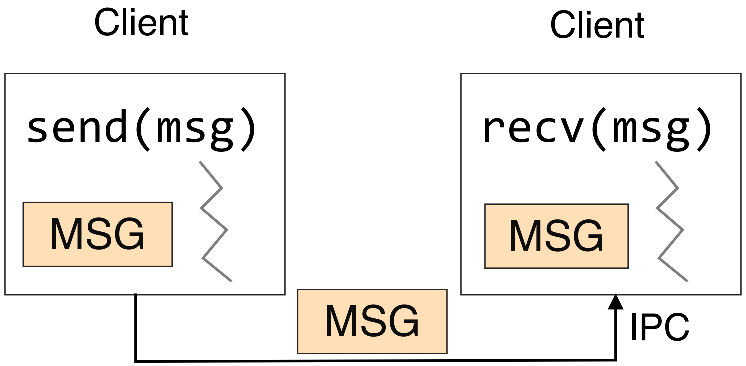
- Applications must locate and and map shared memory regions to exchange data

Review: Shared memory vs. Message passing



- Shared memory: multiple processes to share data via memory

- Applications must locate and and map shared memory regions to exchange data



- Message passing: exchange data explicitly via IPC

- Application developers define protocol and exchanging format, number of participants, and each exchange

Review:
Shared memory vs. Message passing

- Easy to program; just like a single multi-threaded machines
- Hard to write high perf. apps:
  - Cannot control which data is local or remote (remote mem. access much slower)
- Hard to mask failures

- Message passing: can write very high perf. apps
- Hard to write apps:
  - Need to manually decompose the app, and move data
- Need to manually handle failures

Shared memory: Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX (e.g., Linux) OSes

Shared memory: Pthread

```
void *myThreadFun(void *vargp) {
      sleep(1);
      printf(“Hello world!\n”);
      return NULL;
}

int main() {
      pthread_t thread_id_1, thread_id_2;
      pthread_create(&thread_id_1, NULL, myThreadFun, NULL);
      pthread_create(&thread_id_2, NULL, myThreadFun, NULL);
      pthread_join(thread_id_1, NULL);
      pthread_join(thread_id_2, NULL);
      exit(0);
}
```

Message passing: MPI

- MPI – Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing

- Portable: one standard, many implementations
  - Available on almost all parallel machines in C and Fortran
  - De facto standard for the HPC & parallel computing community

Message passing: MPI

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf(“Hello world from rank %d out of %d processors\n”,
            world_rank, workld_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

Message passing: MPI

```
mpirun -n 4 -f host_file ./mpi_hello_world
```

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf(“Hello world from rank %d out of %d processors\n”,
            world_rank, workld_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

MapReduce

The big picture (motivation)

- Datasets are too big to process using a single computer

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)

The big picture (motivation)

- Datasets are too big to process using a single computer

- Good parallel processing engines are rare (back then in the late 90s)

- Want a parallel processing framework that:
  - is **general** (works for many problems)
  - is **easy to use** (no locks, no need to explicitly handle communication, no race conditions)
  - can **automatically parallelize** tasks
  - can **automatically handle machine failures**

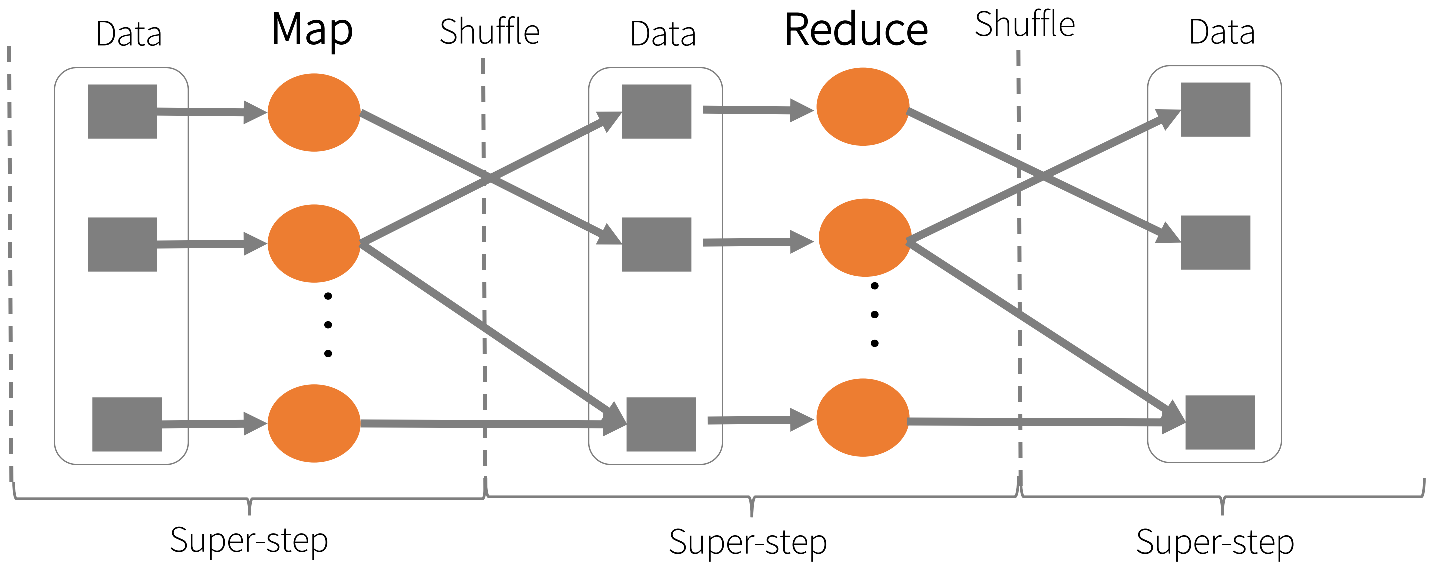
Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures

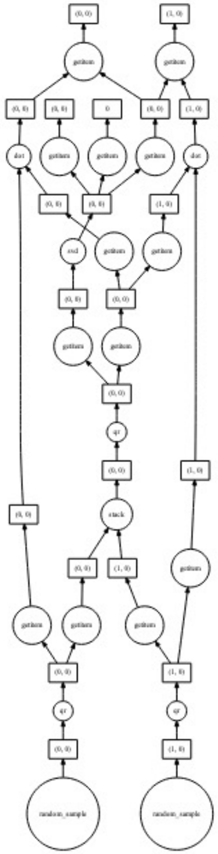
Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures
- **Key question:** how can every Google engineer be imbued with the ability to write parallel, scalable, distributed, fault-tolerant code?
- **Solution:** abstract out the redundant parts
- **Restriction:** relies on job semantics, so restricts which problems it works for

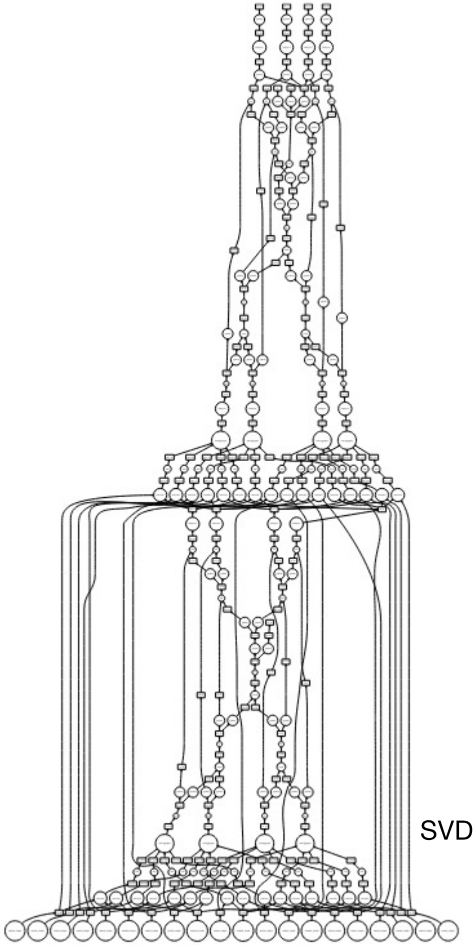
What MapReduce is good at?



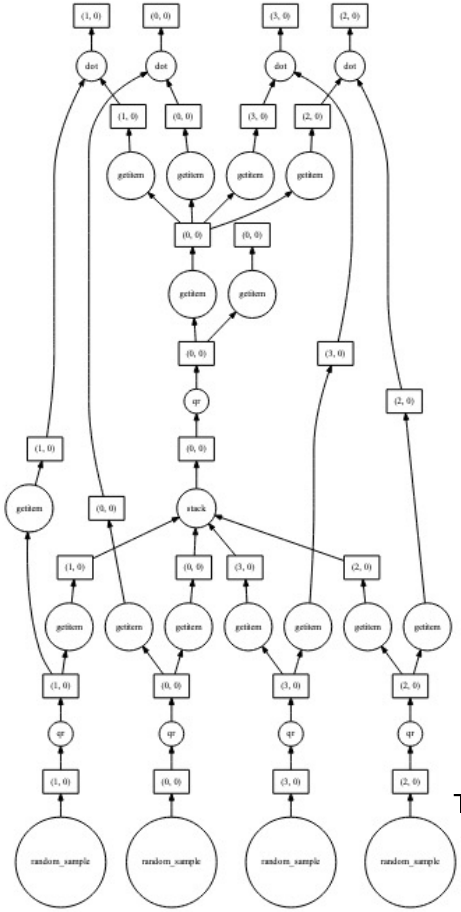
What MapReduce is not good at?



SVD1

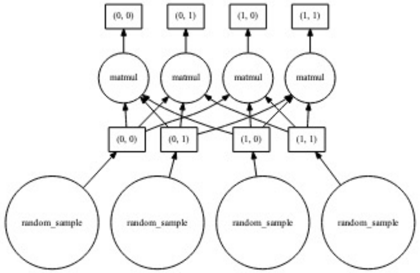


SVD2

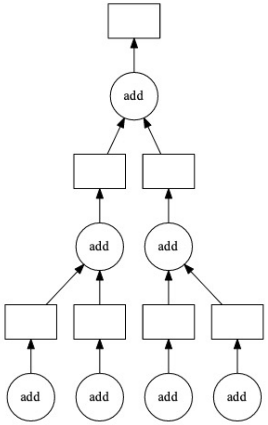


TSQR

Tall- skinny.



GEMM



Tree reduction

Application: Word Count

```
cat data.txt
    | tr –s ‘[[:punct:][:space:]]’ ‘\n’
    | sort | uniq -c
```

```
SELECT count(word), word FROM data
    GROUP BY word
```

Deal with multiple files?

1. Compute word counts from individual files

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

3. Compute word count on merged outputs

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

map

shuffle (sort).

2. Then merge intermediate output

reduce.

3. Compute word count on merged intermediates

MapReduce: Programming interface

- `map(k1, v1) → list(k2, v2)`
  - Apply function to (`k1, v1`) pair and produce set of intermediate pairs (`k2, v2`)

- `reduce(k2, list(v2)) → list(k3, v3)`
  - Apply aggregation (reduce) function to values
  - Output results

MapReduce: Word Count

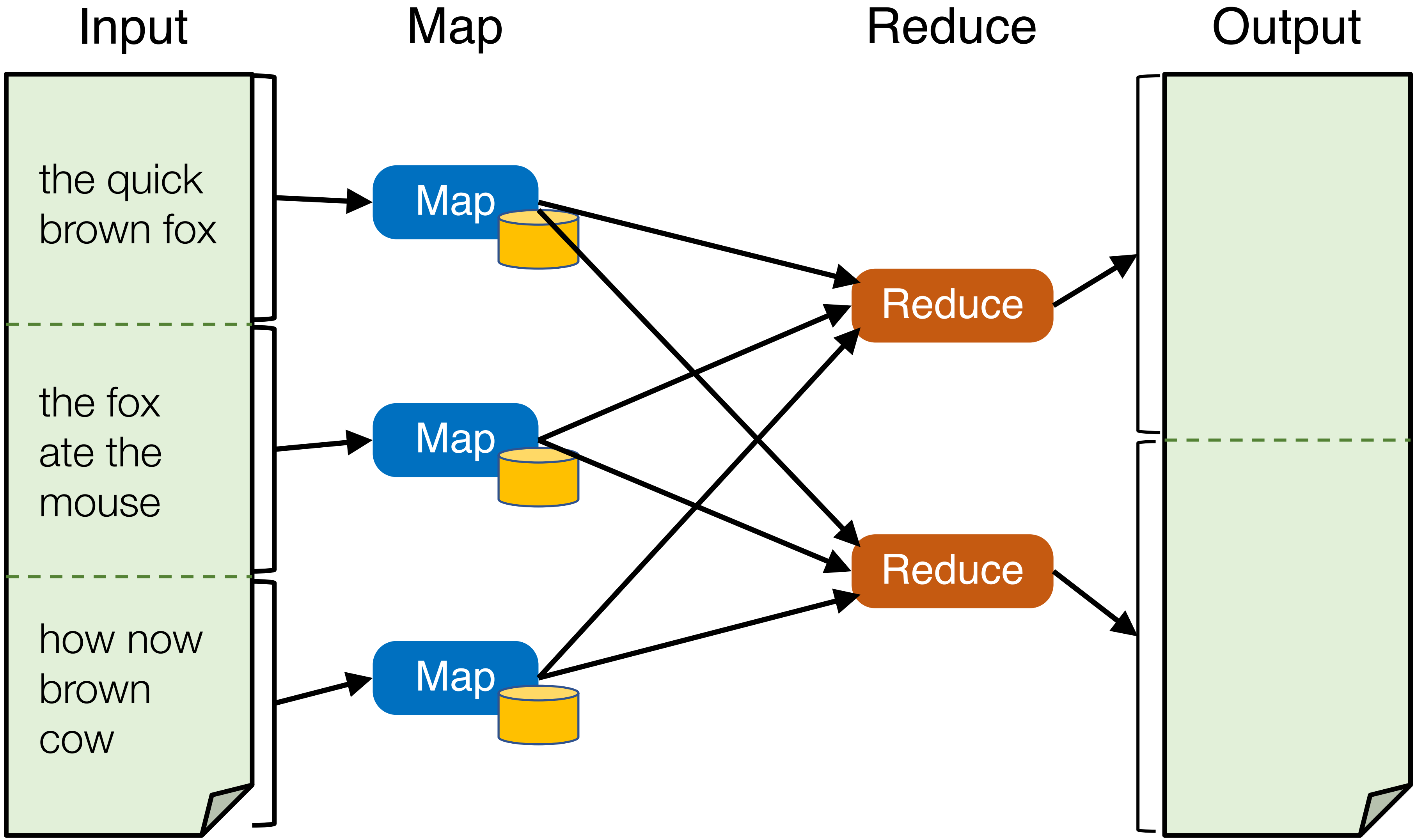
file name.

```
map(key, value):
    for each word w in value:
        EmitIntermediate(w, “1”);
```

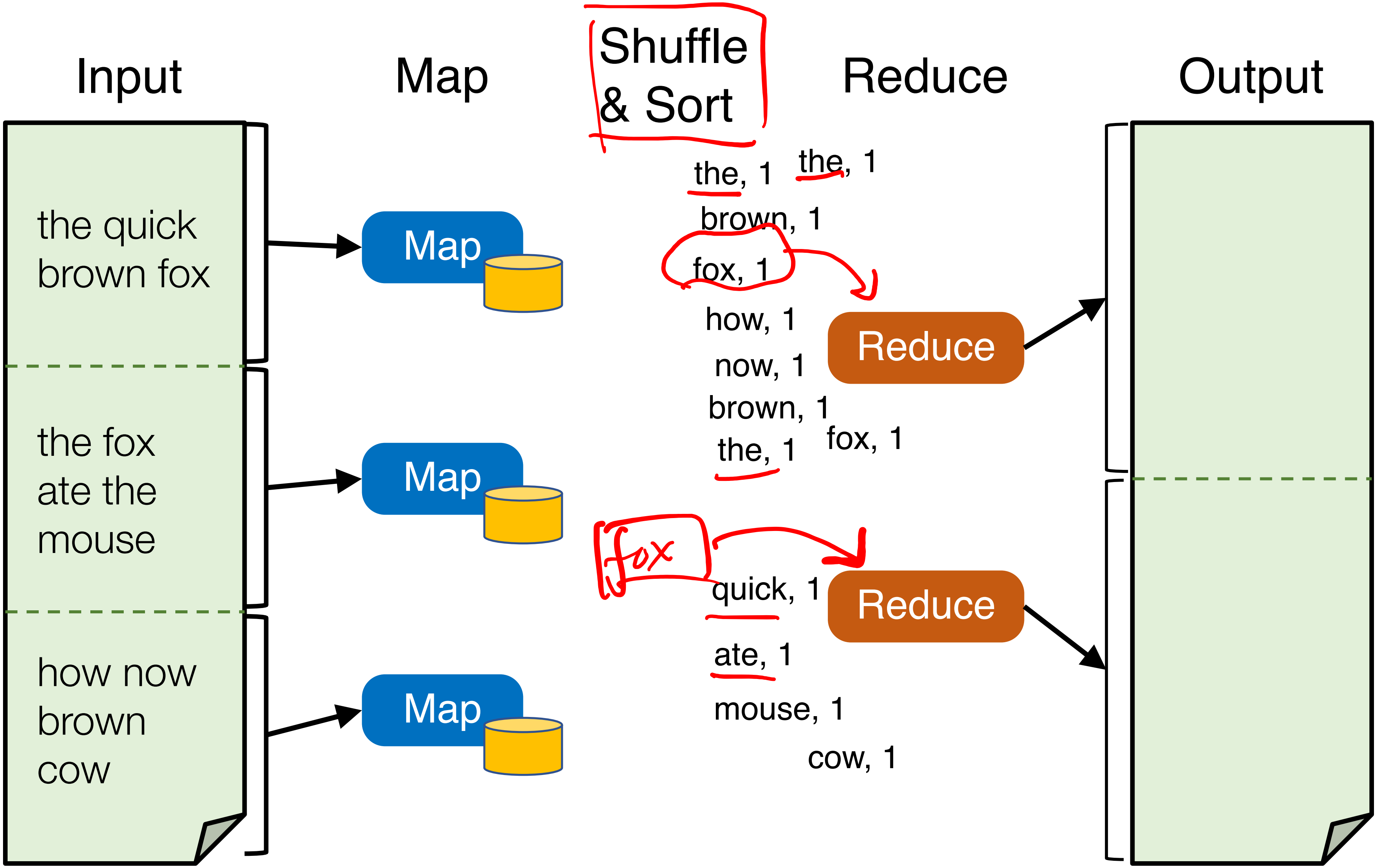
word. [1, 1, 1, 1 ..., 1]. key val.

```
reduce(key, values):
    int result = 0;
    for each v in values:
        results += ParseInt(v);
    Emit(AsString(result));
```

Word Count execution



Word Count execution



Input

Map

Shuffle & Sort

Reduce

Output

the quick brown fox

the fox ate the mouse

how now brown cow

Map

Map

Map

the, 1

the, 1

brown, 1

fox, 1

how, 1

now, 1

brown, 1

the, 1

fox, 1

Reduce

fox

quick, 1

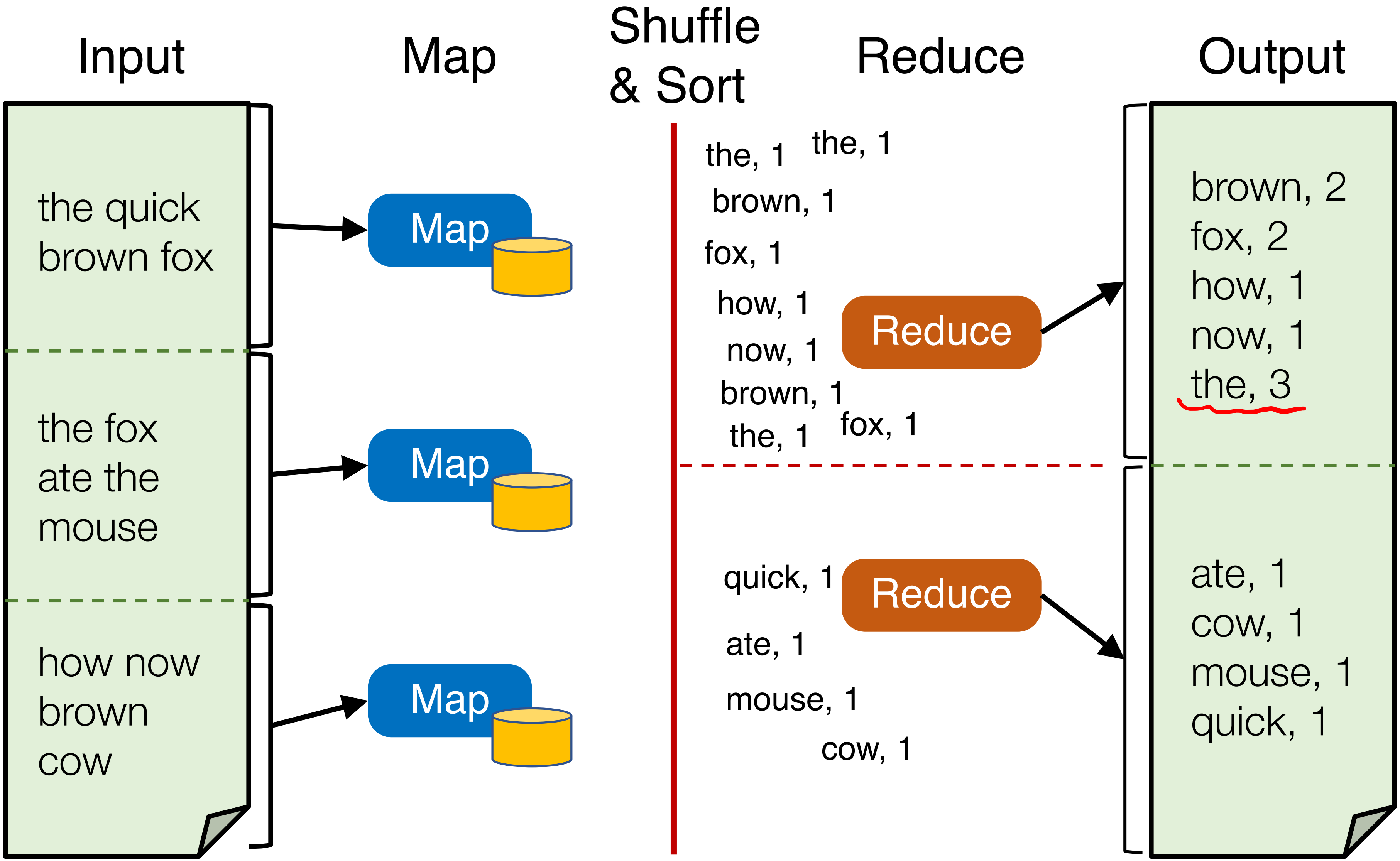
ate, 1

mouse, 1

cow, 1

Reduce

Word Count execution



Input

Map

Shuffle & Sort

Reduce

Output

the quick brown fox

the fox ate the mouse

how now brown cow

Map

Map

Map

the, 1
the, 1
brown, 1
fox, 1
how, 1
now, 1
brown, 1
the, 1
fox, 1

quick, 1
ate, 1
mouse, 1
cow, 1

Reduce

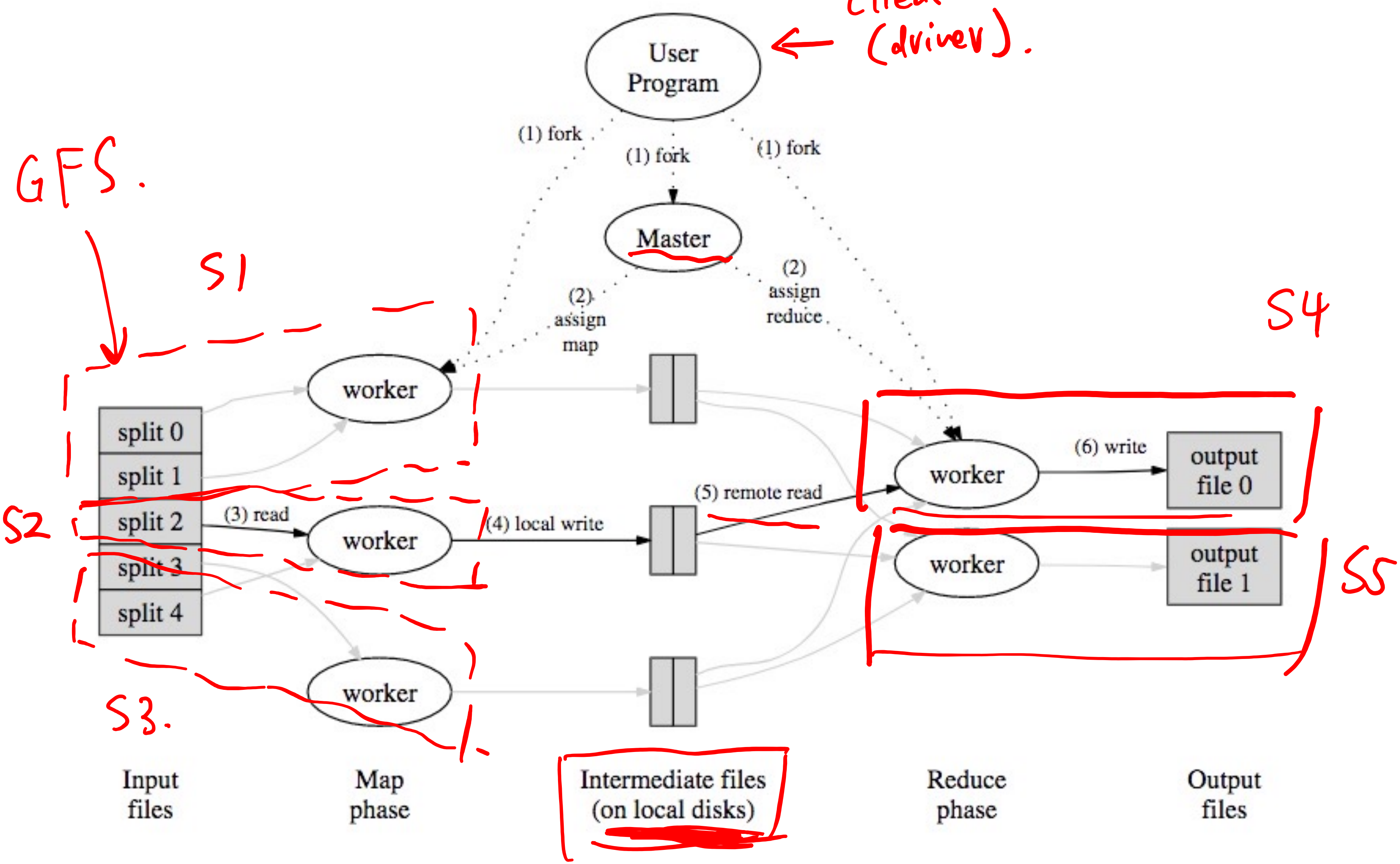
Reduce

brown, 2
fox, 2
how, 1
now, 1
the, 3

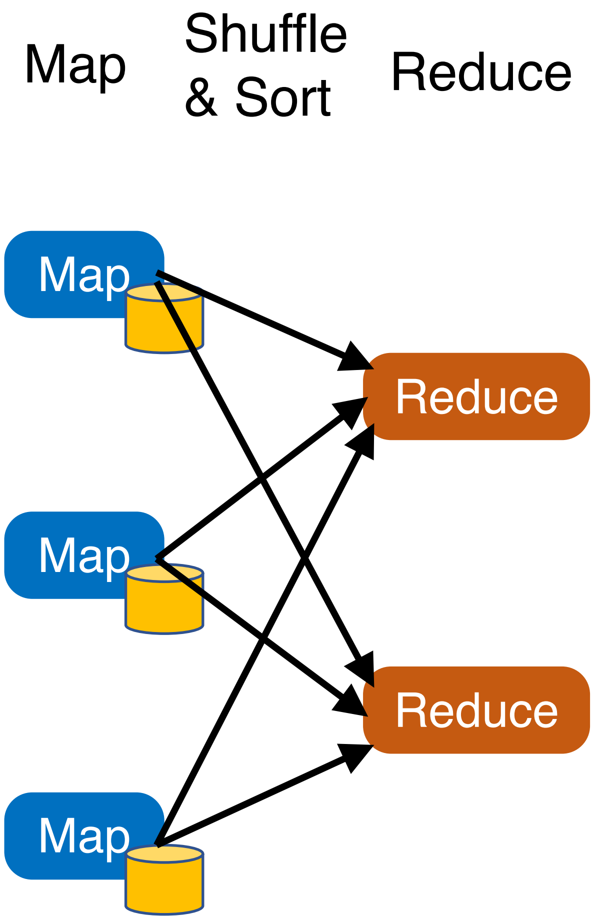
ate, 1
cow, 1
mouse, 1
quick, 1

MapReduce data flows

$O(M \times R)$



MapReduce processes



- Map workers write intermediate output to local disk, separated by partitioning. Once completed, tell master node
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
    - "All-to-all" shuffle b/w mappers and reducers
    - Written to disk ("materialized") b/w each state

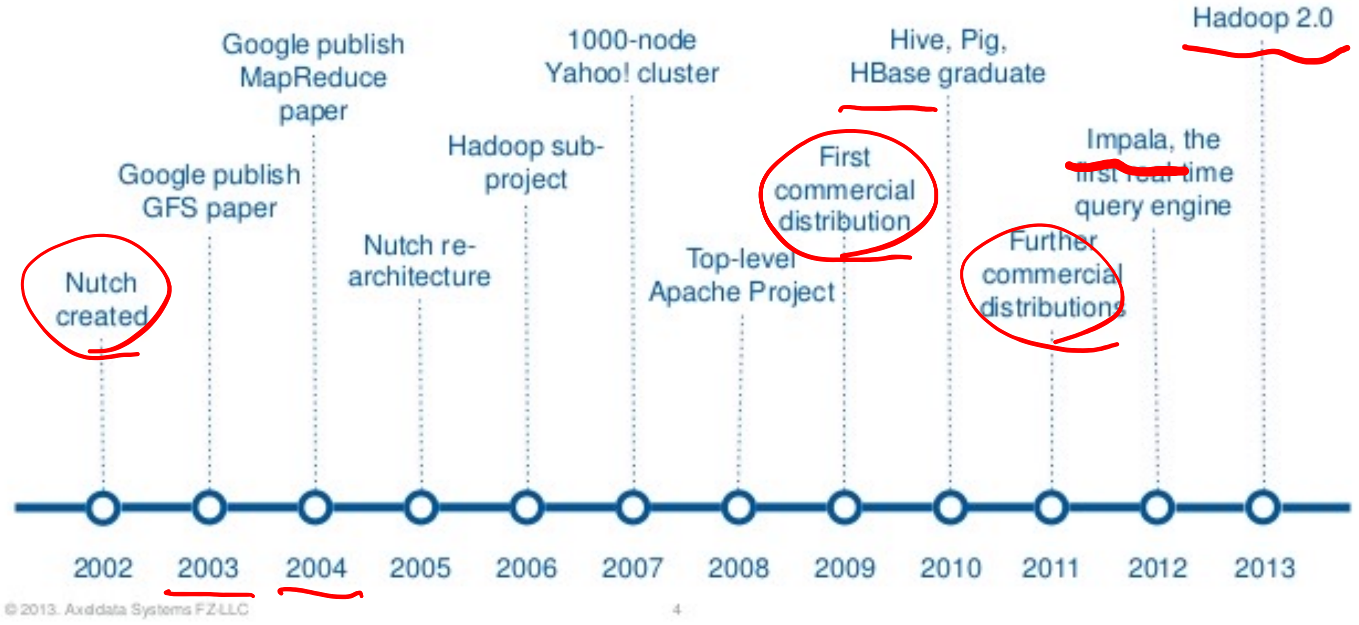
Apache Hadoop

- An open-source implementation of Google's MapReduce framework
  - Hadoop MapReduce atop Hadoop Distributed File System (HDFS)

A Brief History of Hadoop

Big Table

YARN.



2002: Nutch created

2003: Google publish GFS paper

2004: Google publish MapReduce paper

2005: Nutch re-architecture

2006: Hadoop sub-project

2007: 1000-node Yahoo! cluster

2008: Top-level Apache Project

2009: First commercial distribution

2010: Hive, Pig, HBase graduate

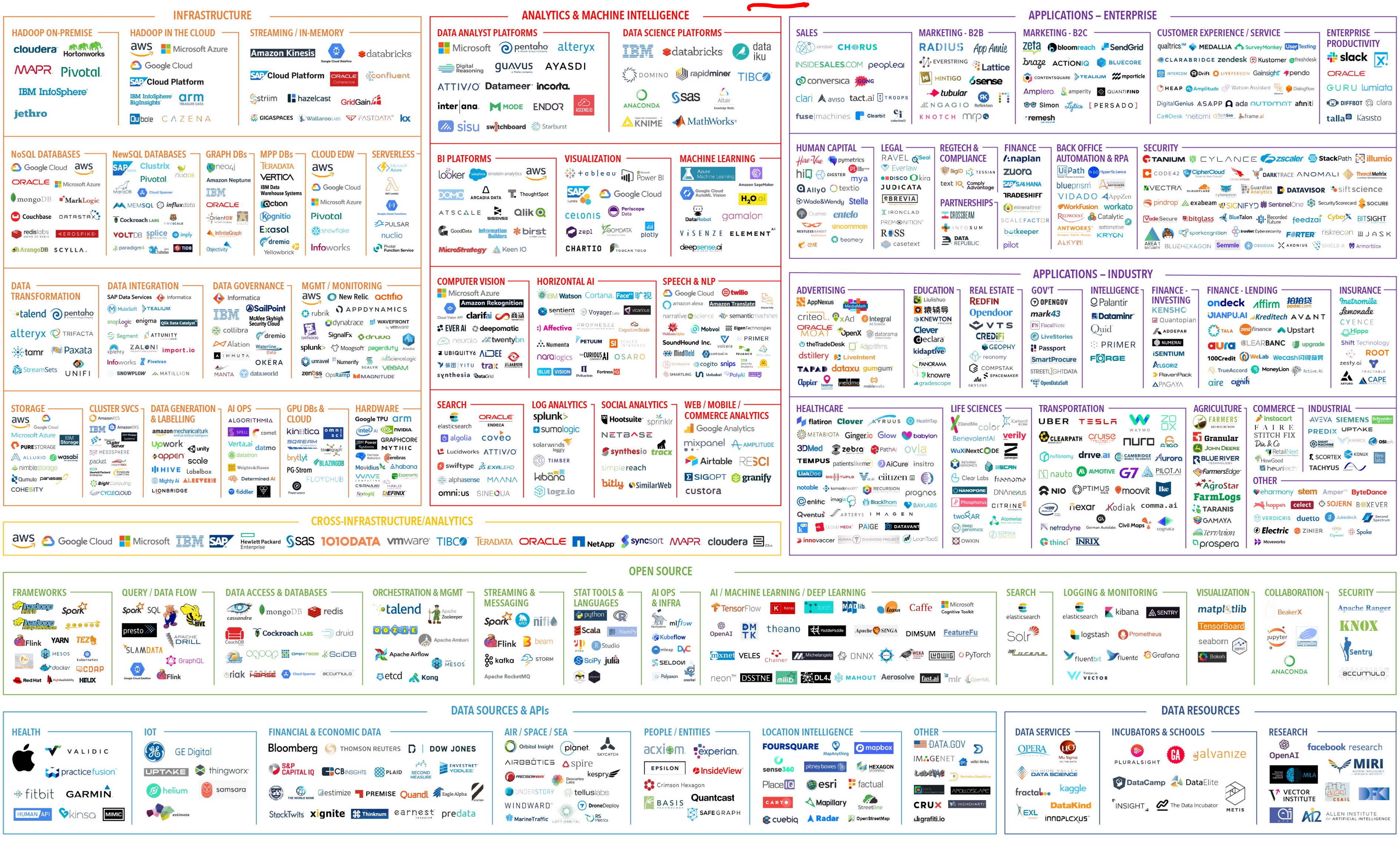
2011: Further commercial distributions

2012: Impala, the first real-time query engine

2013: Hadoop 2.0

© 2013. Axiddata Systems FZ-LLC 4

DATA & AI LANDSCAPE 2019



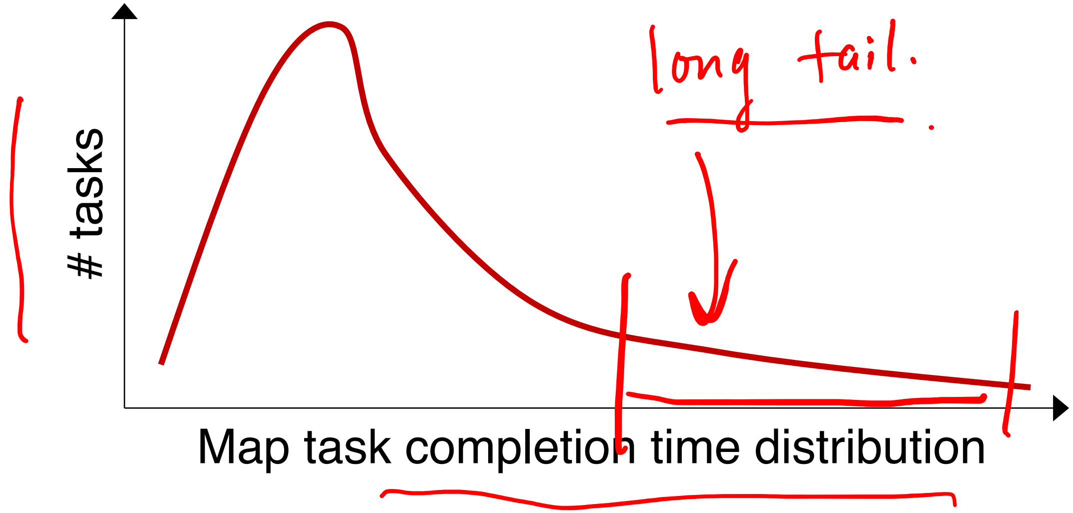
July 16, 2019 - FINAL 2019 VERSION

© Matt Turck (@mattturck), Lisa Xu (@lisaxu92), & FirstMark (@firstmarkcap)

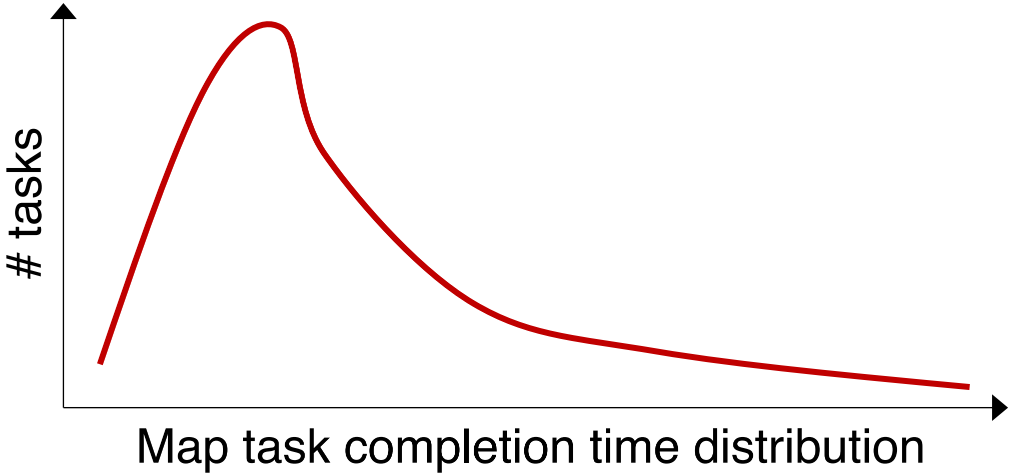
mattturck.com/data2019

FIRSTMARK
EARLY STAGE VENTURE CAPITAL

Stragglers



Stragglers



- Tail latency means some workers (always) finish late

- Q: How can MR work around this?
  - Hint: its approach to **fault-tolerance** provides the right tool

Resilience against stragglers

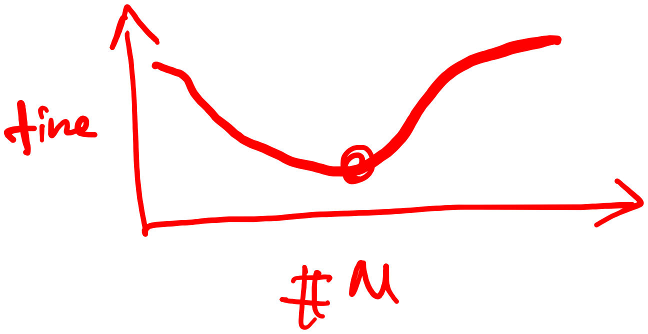
- If a task is going slowly (i.e., straggler):
  - Launch second copy of task on another node
  - Take the output of whichever finishes first

More design

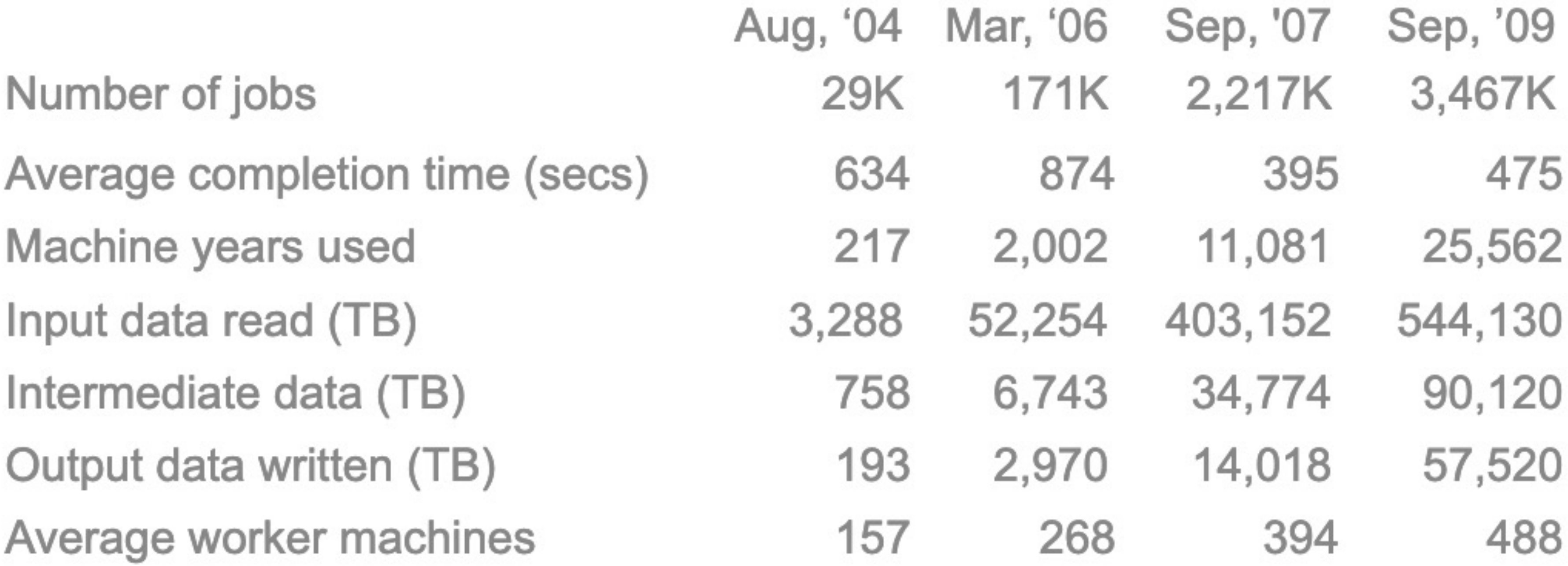
- Master failure

→ - Locality

→ - Task granularity



MapReduce usage statistics over time



	Aug, ‘04	Mar, ‘06	Sep, '07	Sep, ’09
Number of jobs	29K	171K	2,217K	3,467K
Average completion time (secs)	634	874	395	475
Machine years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average worker machines	157	268	394	488

* Jeff Dean, LADIS 2009

GFS usage at Google

- 200+ clusters
- Many clusters of 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems
- 40 GB/s read/write load
  - In the presence of frequent hardware failures

* Jeff Dean, LADIS 2009

MapReduce discussion

- What will likely serve as a performance bottleneck for Google’s MapReduce used back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

MapReduce discussion

- What will likely serve as a performance bottleneck for Google's MapReduce used back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

- How does MapReduce reduce the effect of slow network?

MapReduce discussion

- How does MapReduce jobs get good load balance across worker machines?

MapReduce discussion

- Consider the indexing pipeline where you start with HTML documents. You want to index the documents after removing the most commonly occurring words:
    1. Compute the most common words;
    2. Remove them and build the index

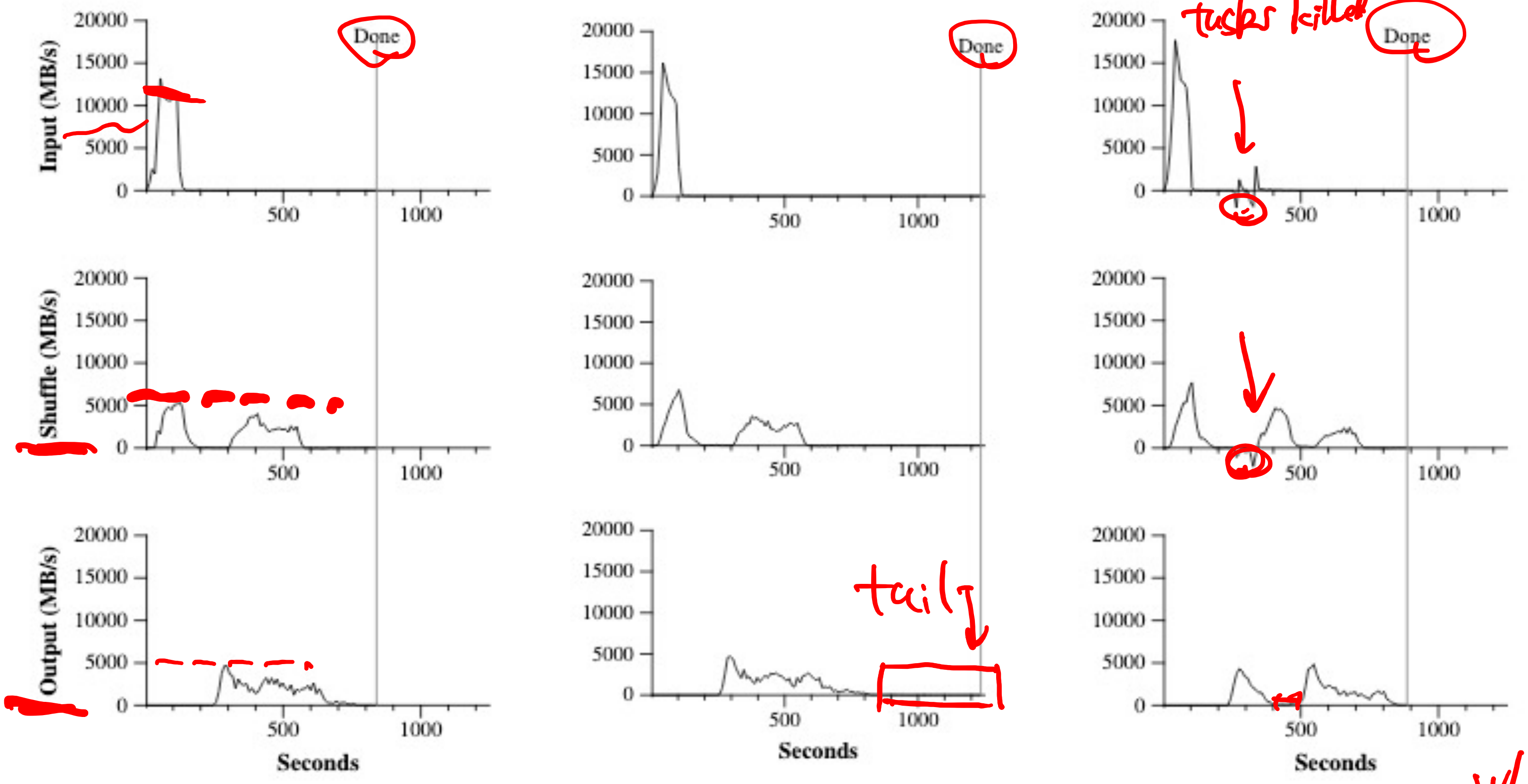
What are the main shortcomings of using MapReduce to support such pipeline-like applications?

MapReduce discussion

820 sec.

1200 sec.

880 sec.



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

w/ backup.

TeraSort.

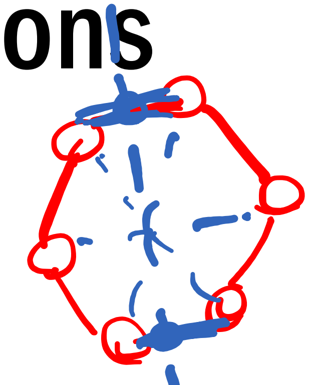
Today's outline

How can large computing jobs be parallelized?

1. MapReduce

2. Google File System

Review: MapReduce assumptions



- Commodity hardware
  - Economies of scale!
  - Commodity networking with less bisection bandwidth
  - Commodity storage (hard disks) is cheap

- Failures are common

- Replicated, distributed file system for data storage

Review: Fault tolerance

- If a task crashes:
  - Retry on another node
    - Why is this okay?
  - If the same task repeatedly fails, end the job

Review: Fault tolerance

- If a task crashes:
    - Retry on another node
        - Why is this okay?
    - If the same task repeatedly fails, end the job

- If a node crashes:
    - Relaunch its current tasks on another node
        - What about task inputs?

Google file system (GFS)

- Goal: a global (distributed) file system that stores data across many machines
  - Need to handle 100's TBs

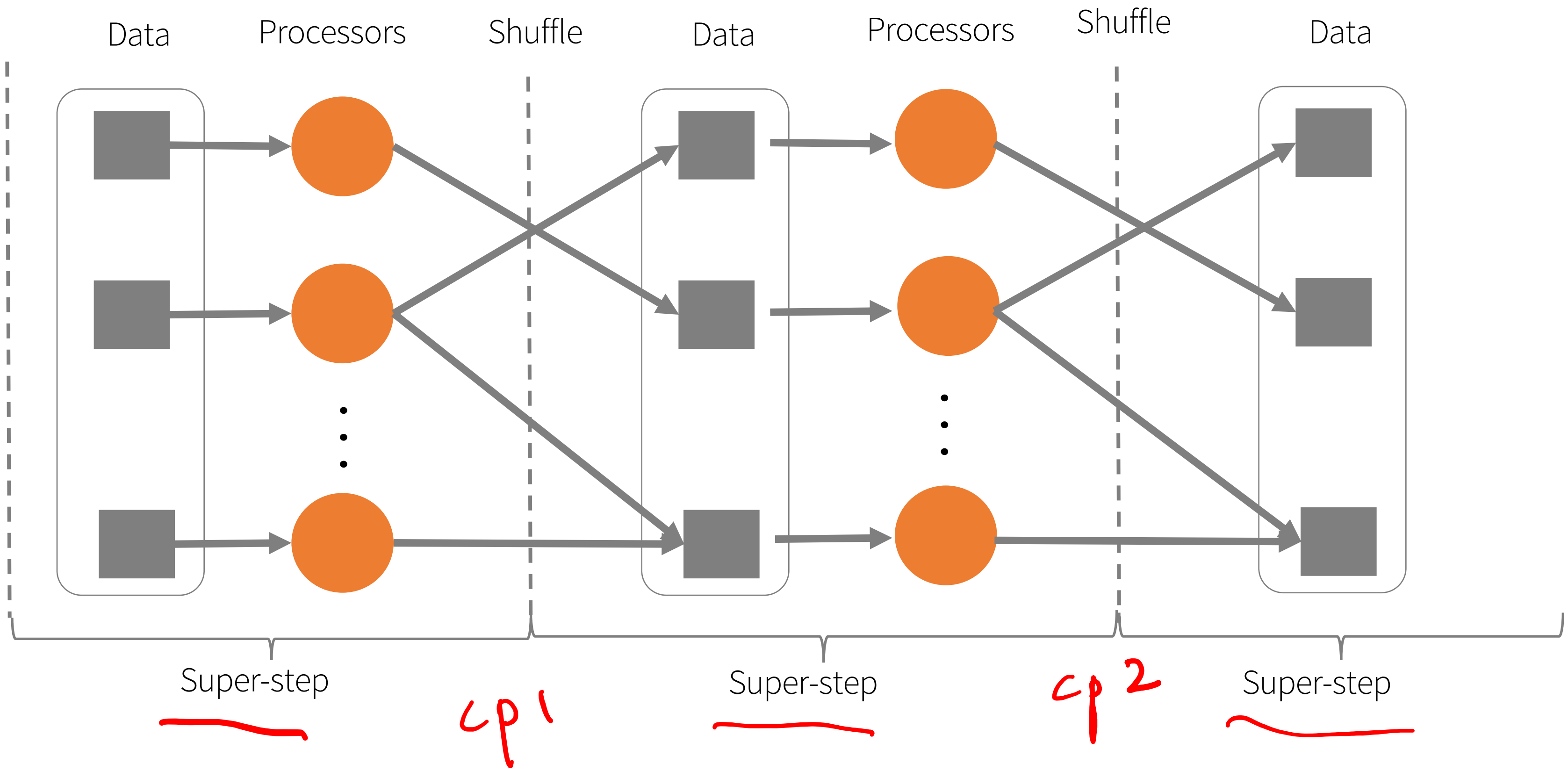
- Google published details in 2003

- Open source implementation:
  - Hadoop Distributed File System (HDFS)

Workload-driven design

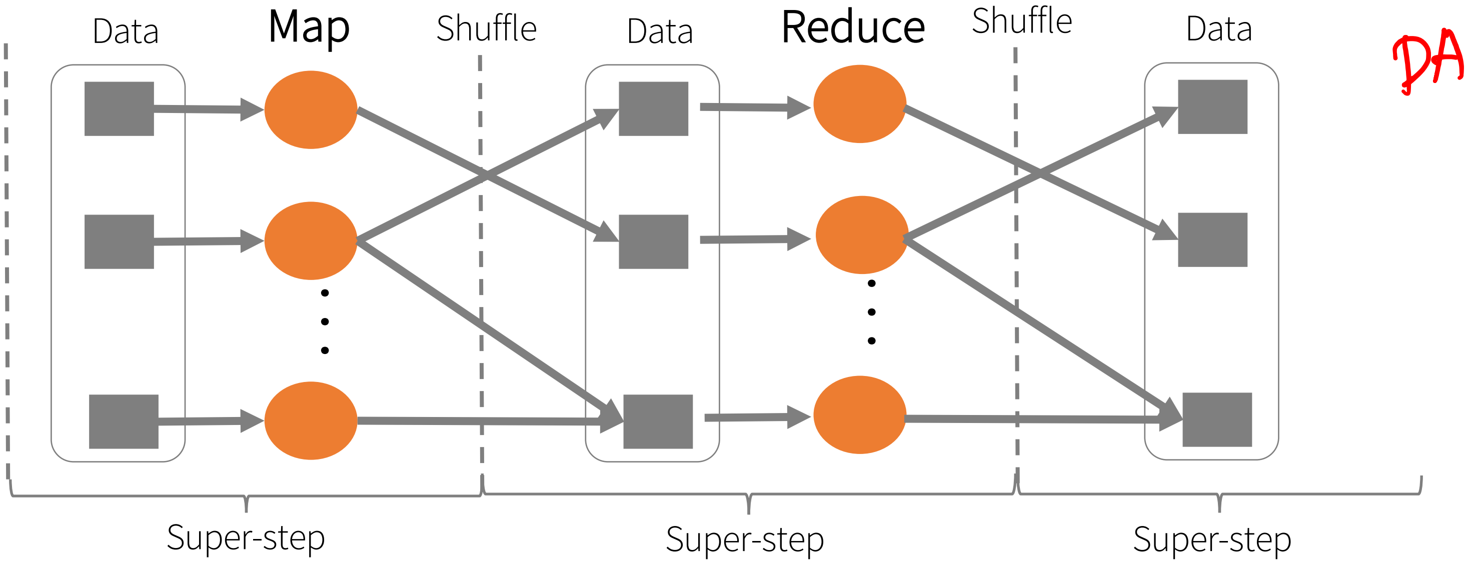
- MapReduce workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

Example workloads: Bulk Synchronous Processing (BSP)



*Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990

MapReduce as a BSP system



DAG.

- Read entire dataset, do computation over it
  - Batch processing

- Producer/consumer: many producers append work to file concurrently; one consumer reads and does work

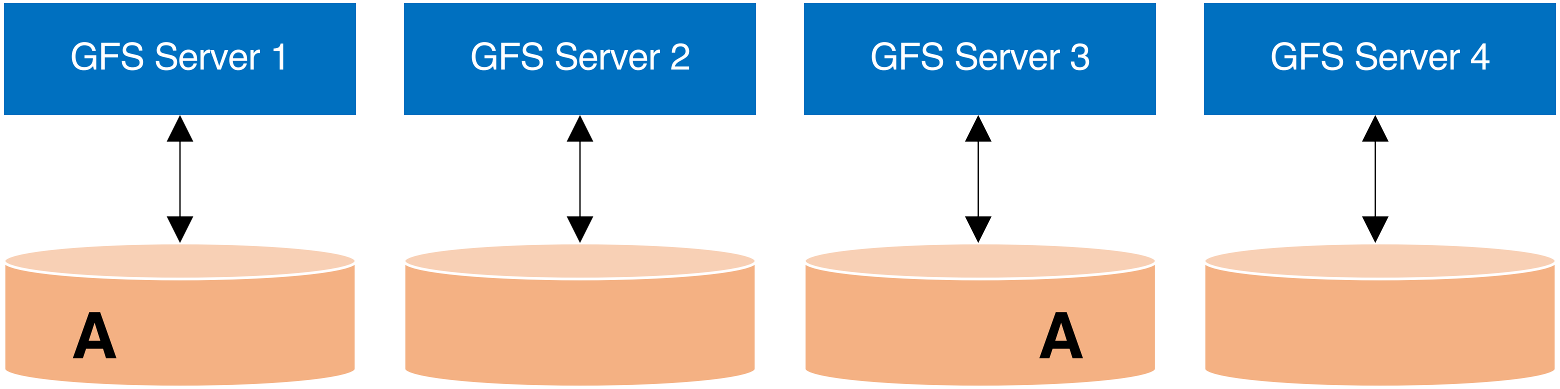
Workload-driven design

- Build a global (distributed) file system that incorporates all these application properties

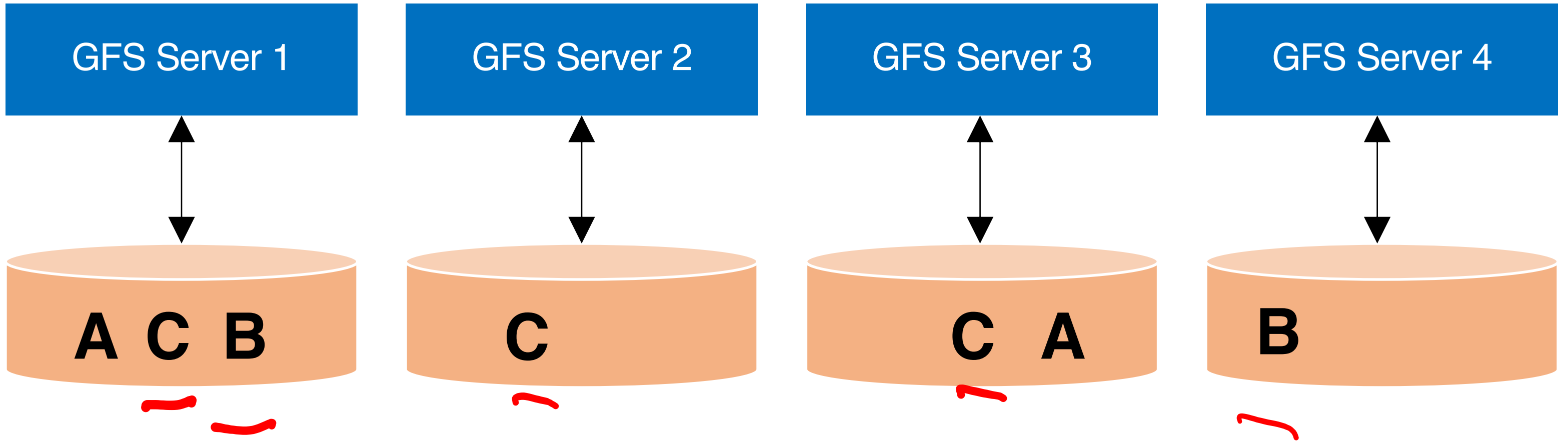
- Only supports **features required by applications**

- Avoid difficult local file system features, e.g.:
  - links

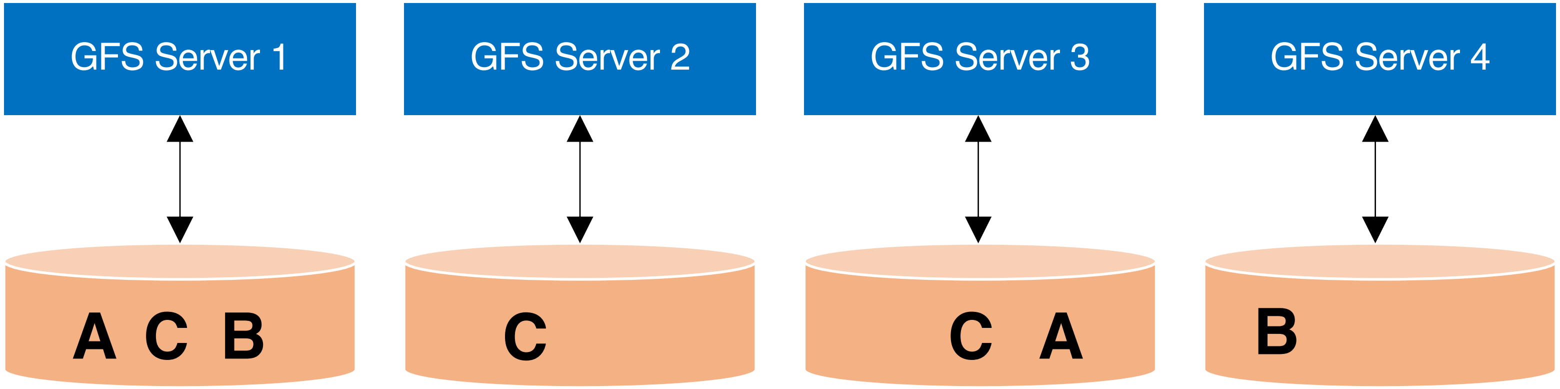
Replication



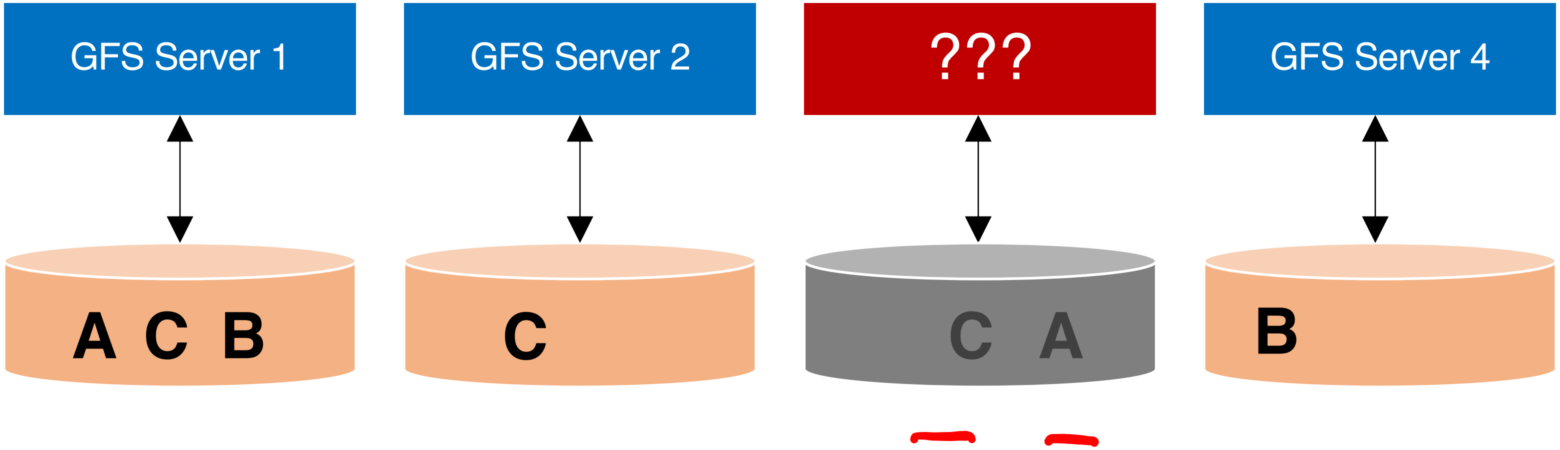
Replication



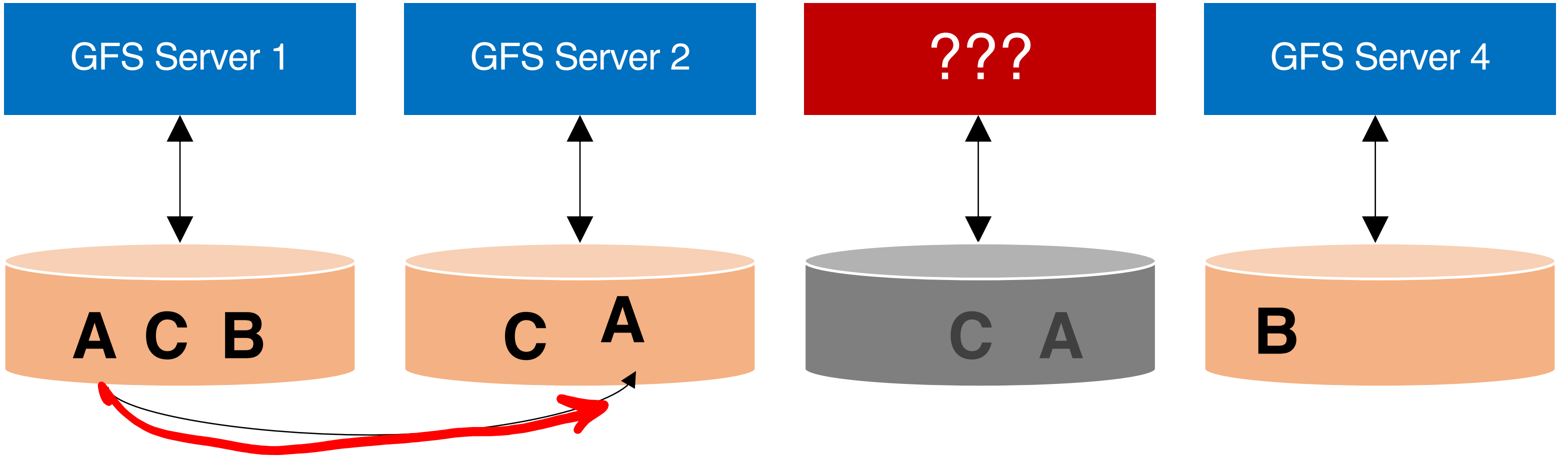
Resilience against failures



Resilience against failures

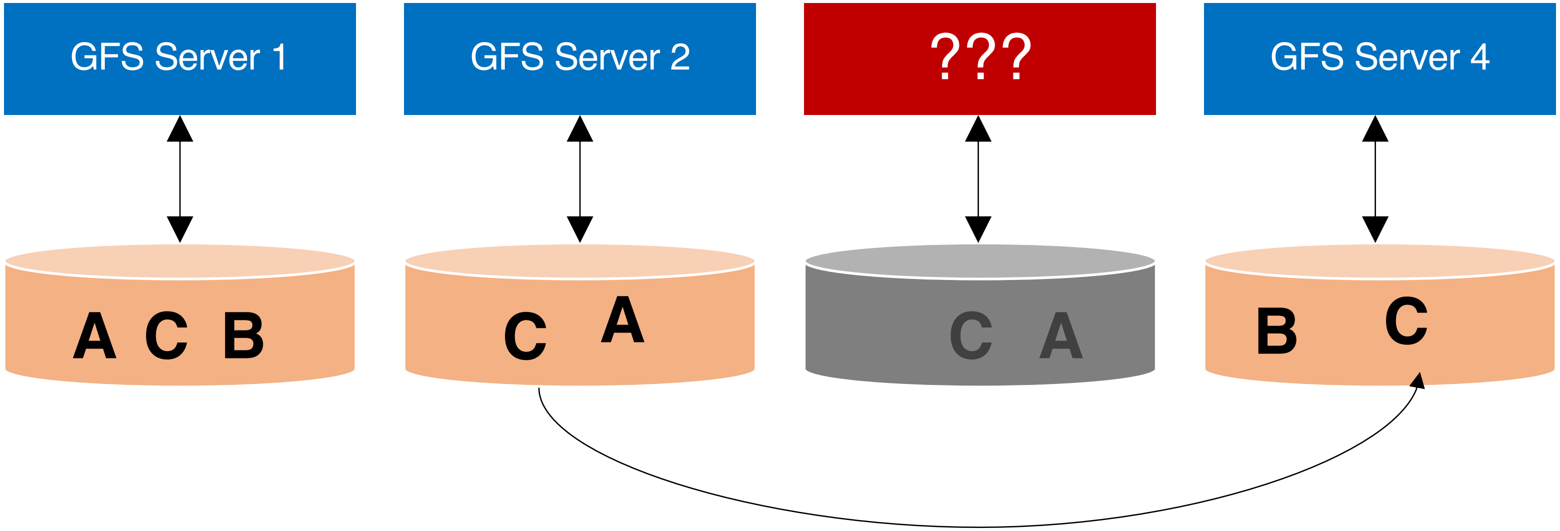


Data recovery



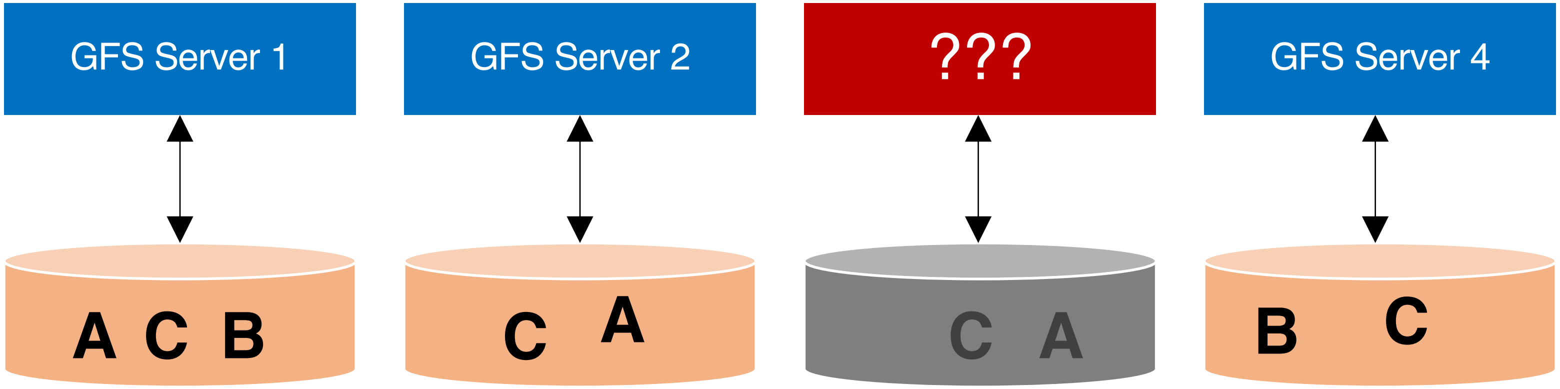
Replicating A to maintain a replication factor of 2

Data recovery



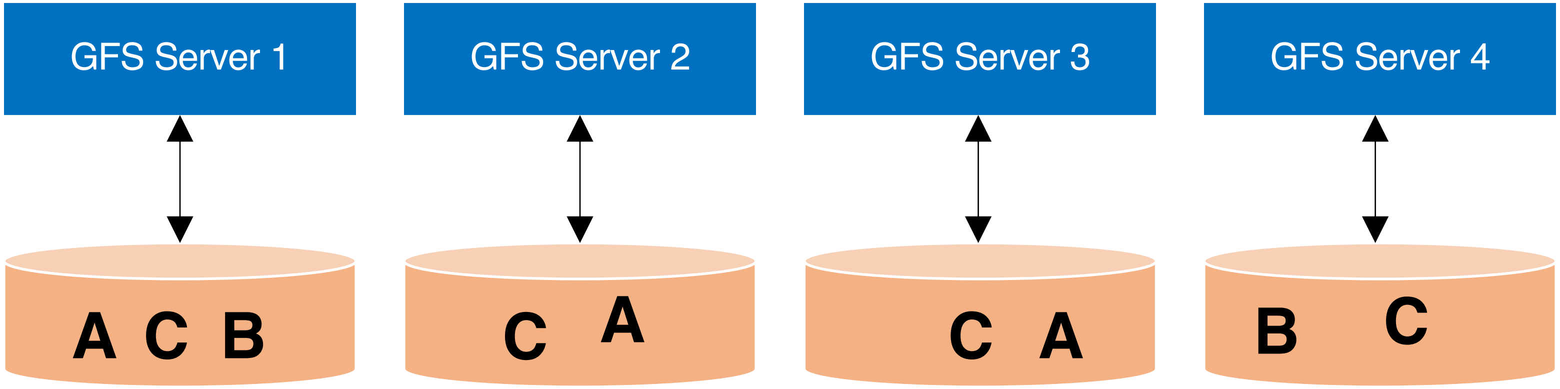
Replicating C to maintain a replication factor of 3

Data recovery



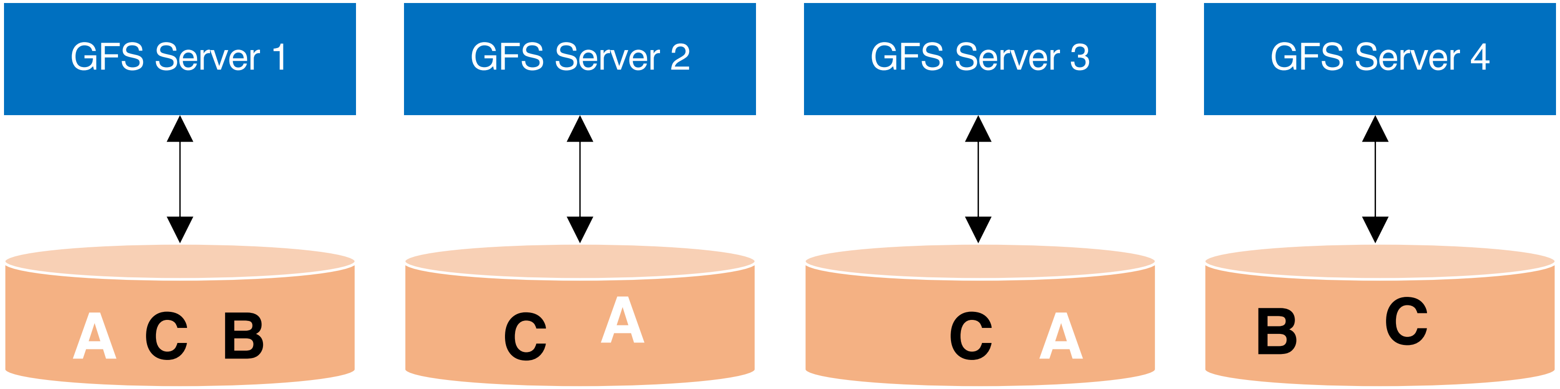
Machine may be dead forever, or it may come back

Data recovery

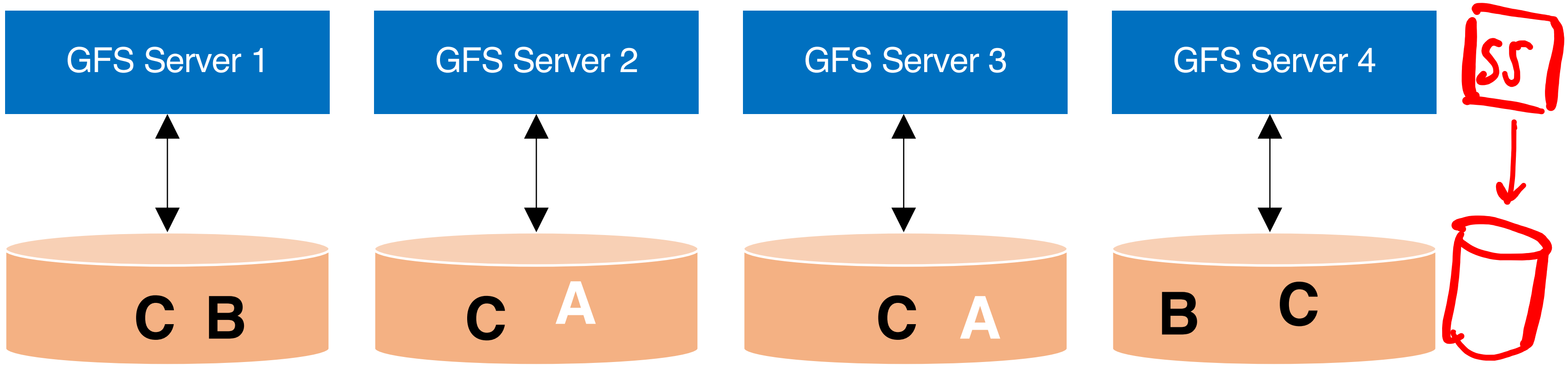


Machine may be dead forever, or it may come back

Data recovery



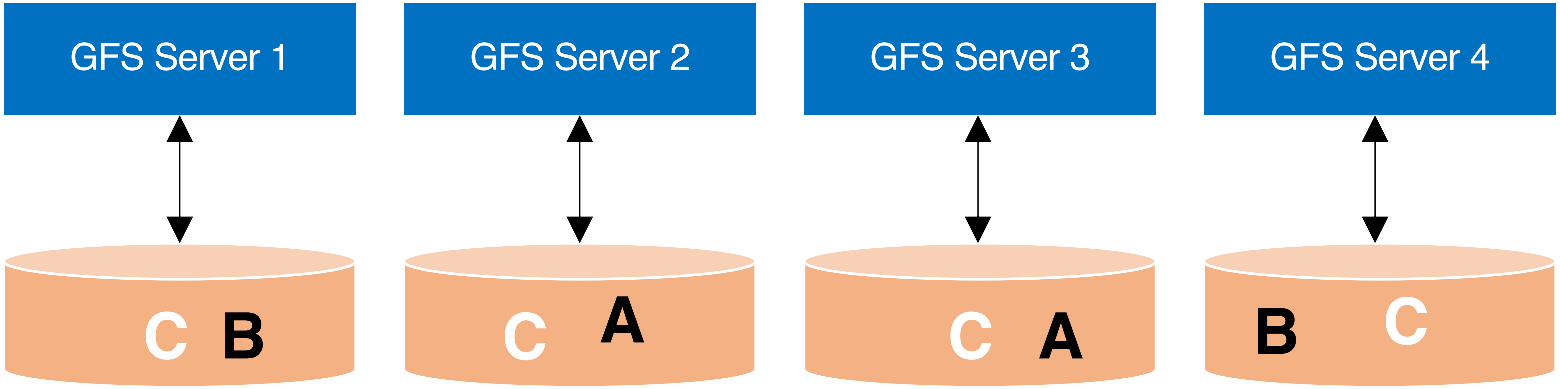
Data recovery



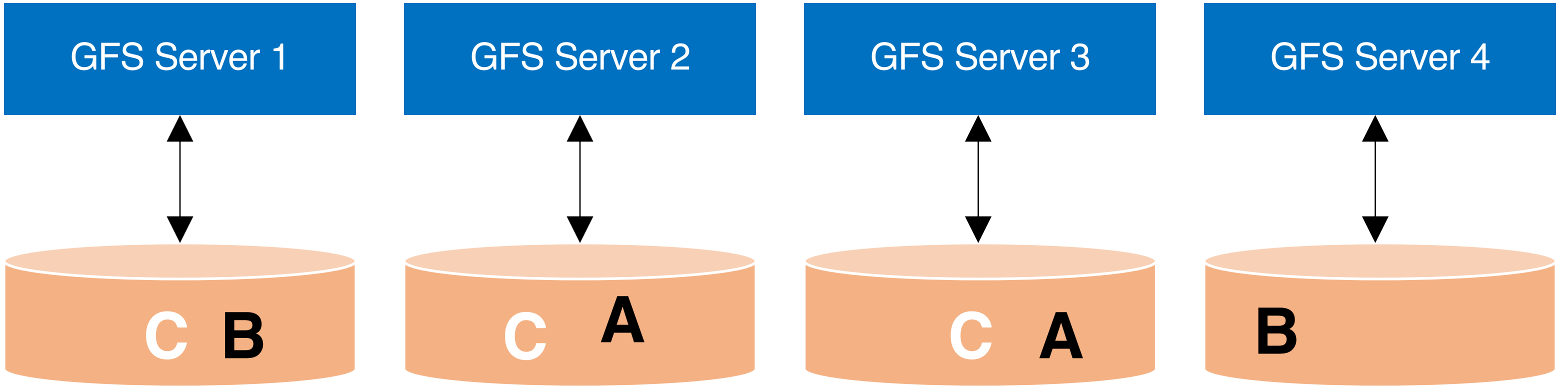
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data recovery



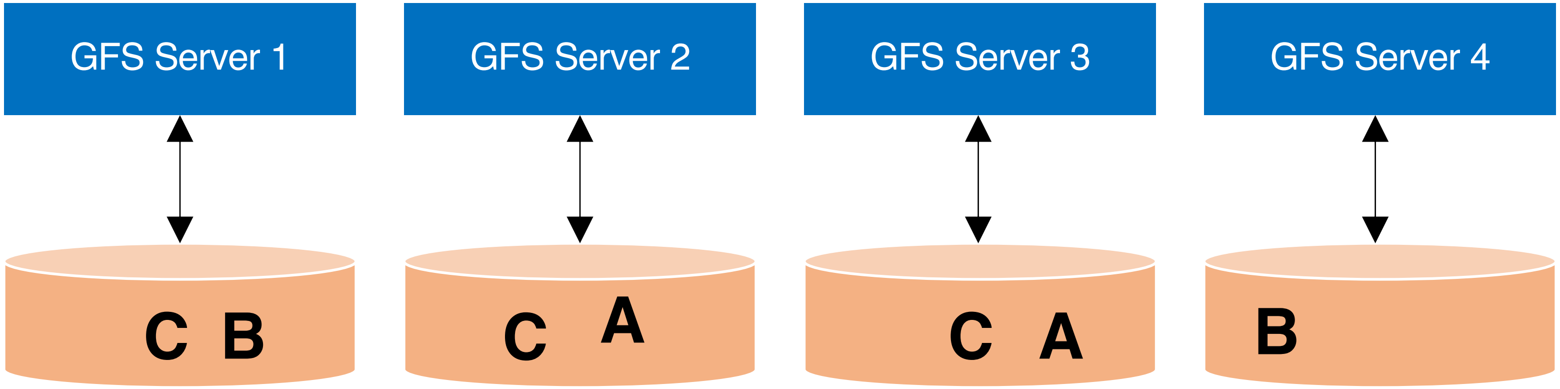
Data recovery



Data Rebalancing

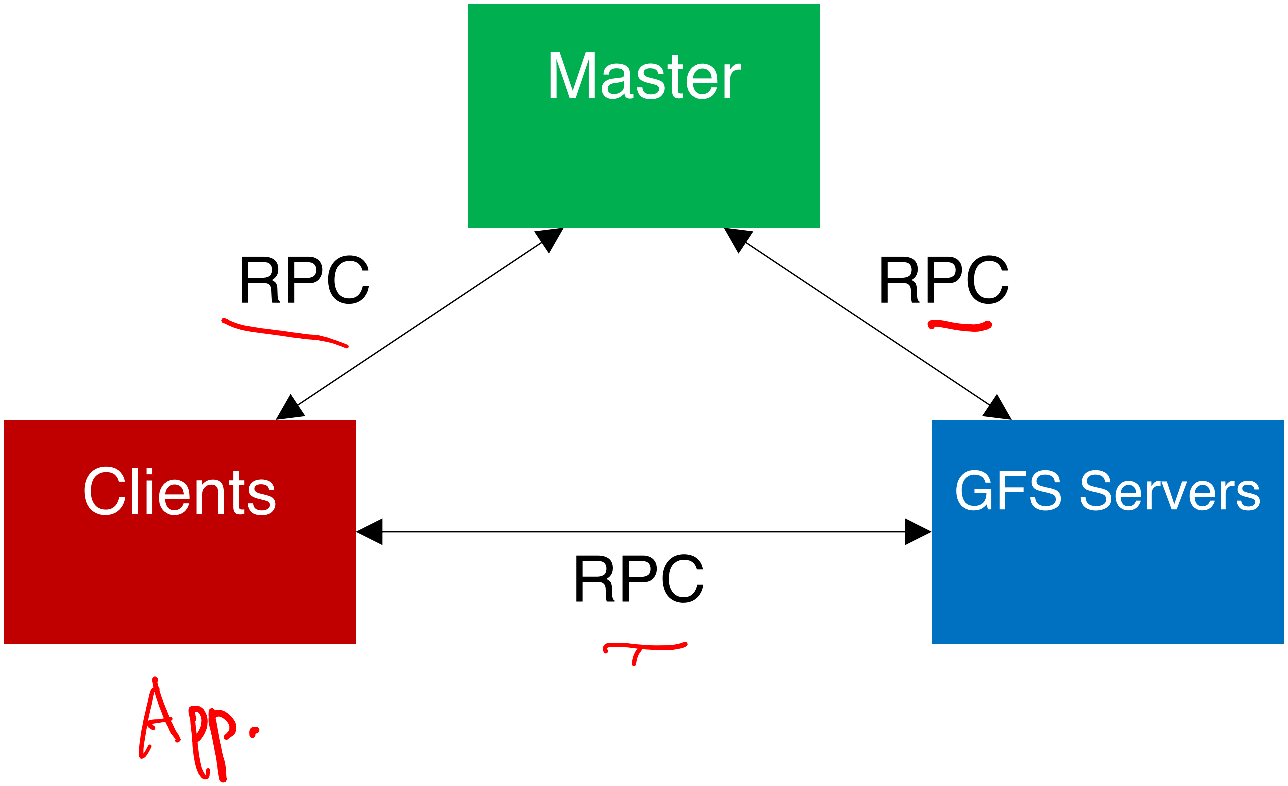
Deleting one C to maintain a replication factor of 3

Data recovery

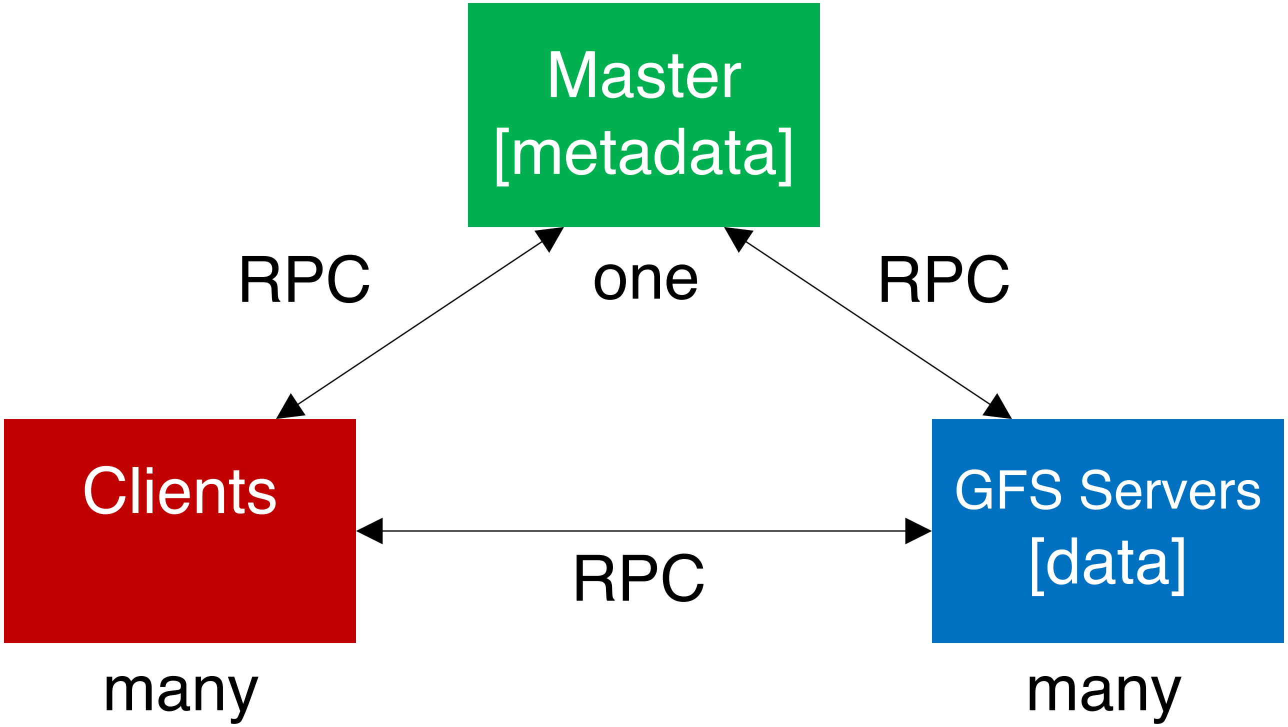


Question: how to maintain a global view of all data distributed across machines?

GFS architecture: logical view



GFS architecture: logical view



BTW, what is RPC?

- RPC = Remote procedure call

Motivation: Why RPC?

- The typical programmer is trained to write single-threaded code that runs in one place

- **Goal:** Easy-to-program network communication that makes client-server communication transparent
  - Retains the "feel" of writing centralized code
    - Programmer needn't think about the network
    - Avoid tedious socket programming

What’s the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a procedure call:
  - Caller pushes arguments onto stack,
    - jumps to address of callee function
  - Callee reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

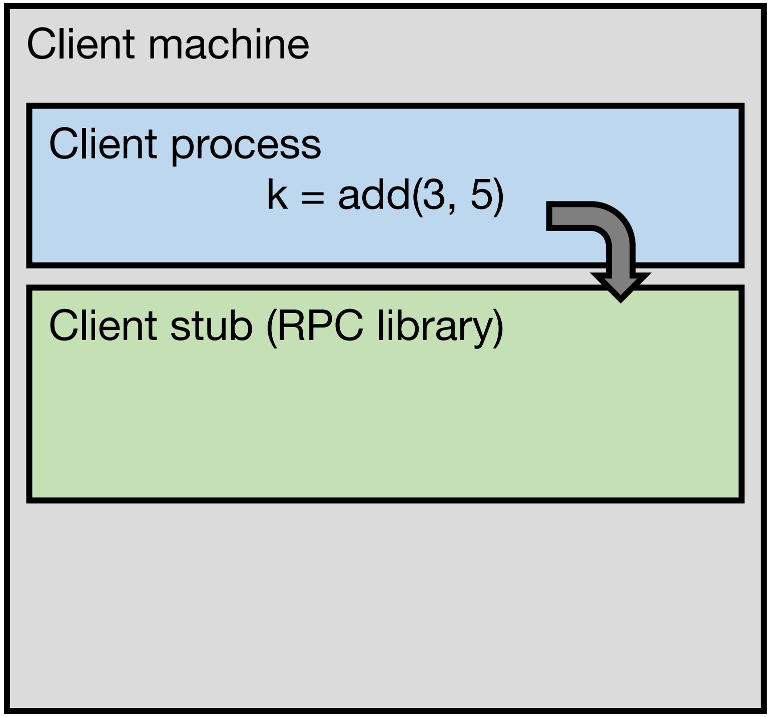
What's the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a procedure call:
  - Caller pushes arguments onto stack,
    - jumps to address of callee function
  - Callee reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

RPC's Goal: make communication appear like a local procedure call: transparency for procedure calls – way less painful than sockets…

A day in the life of an RPC

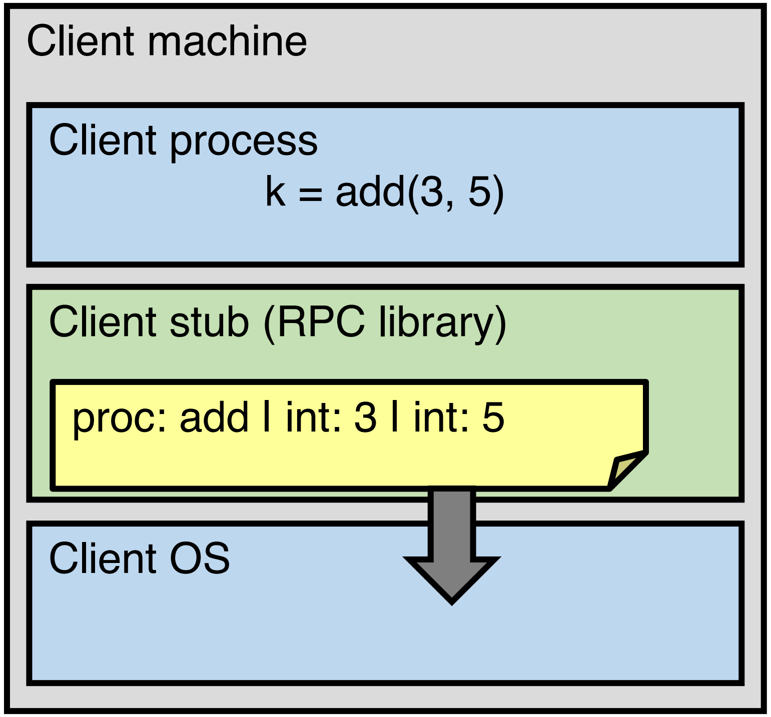
1. Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

1. Client calls stub function (pushes parameters onto stack)

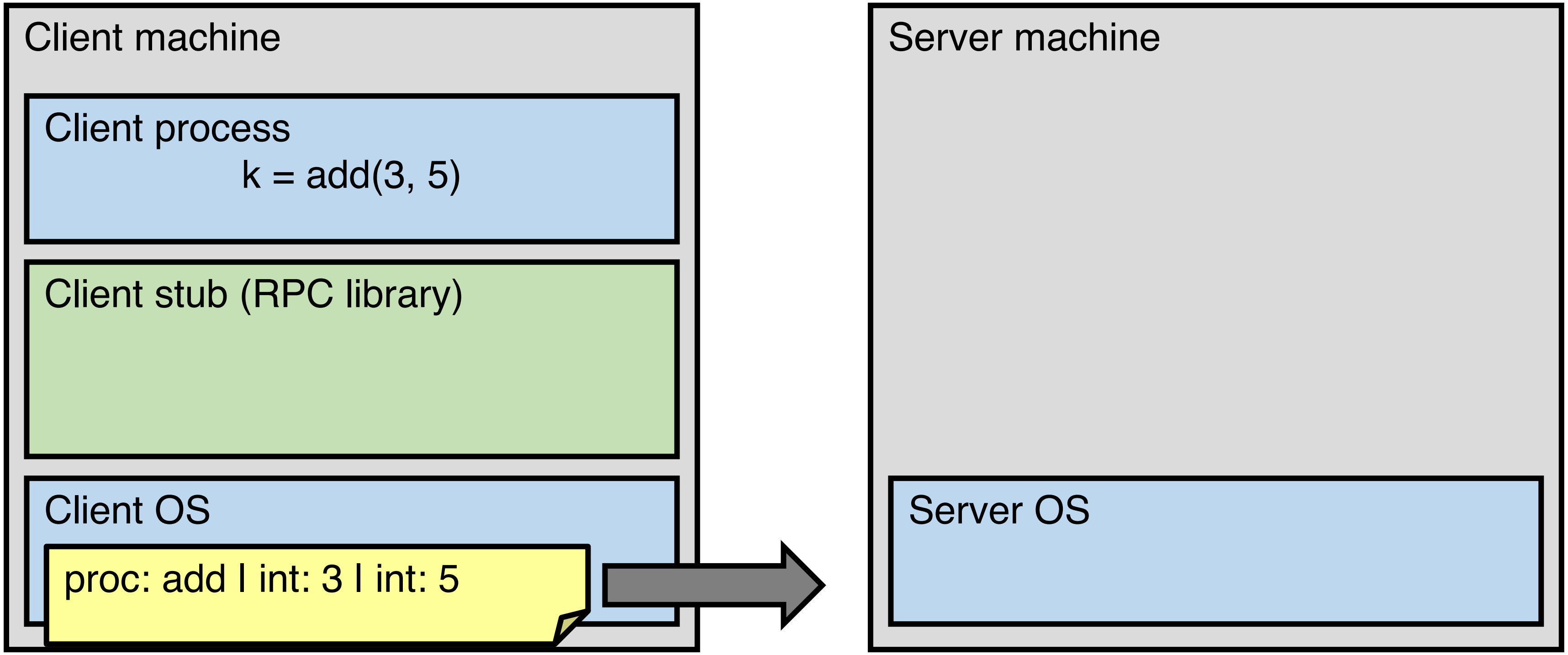
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

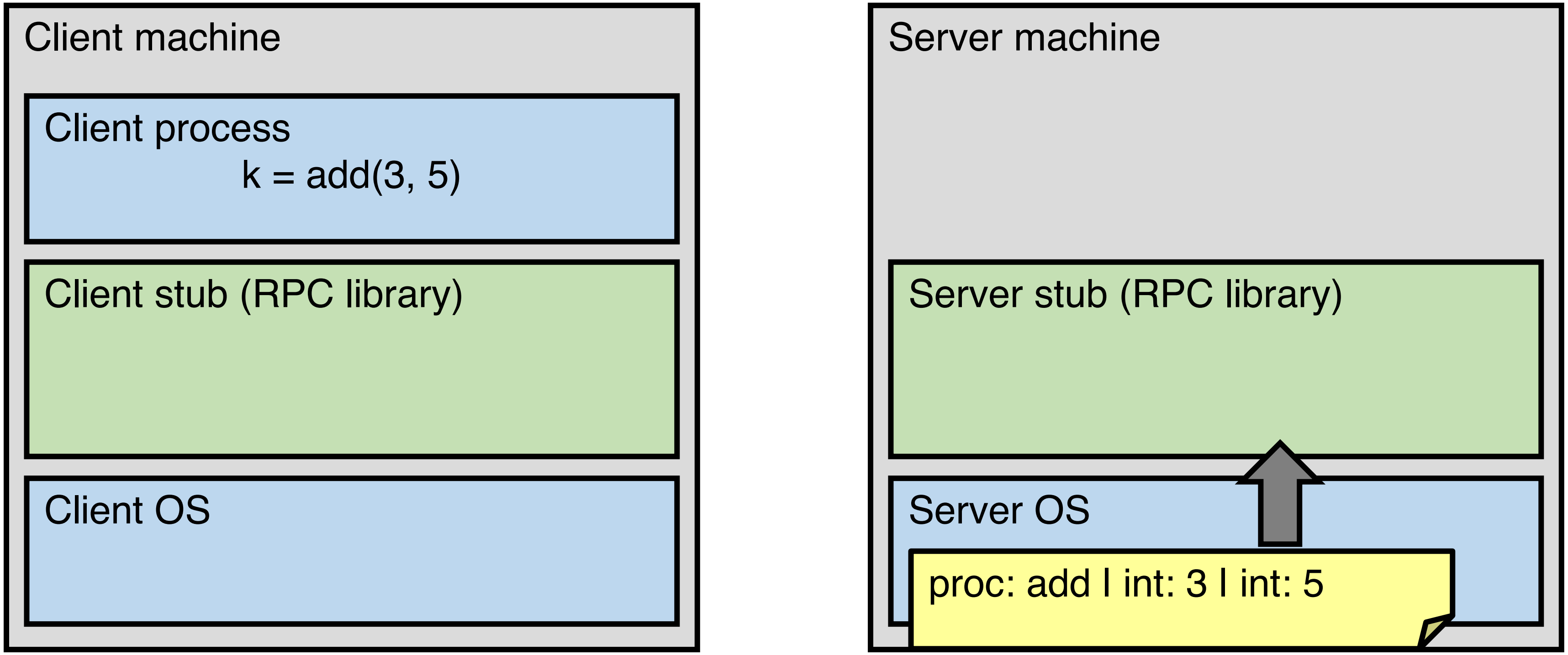
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

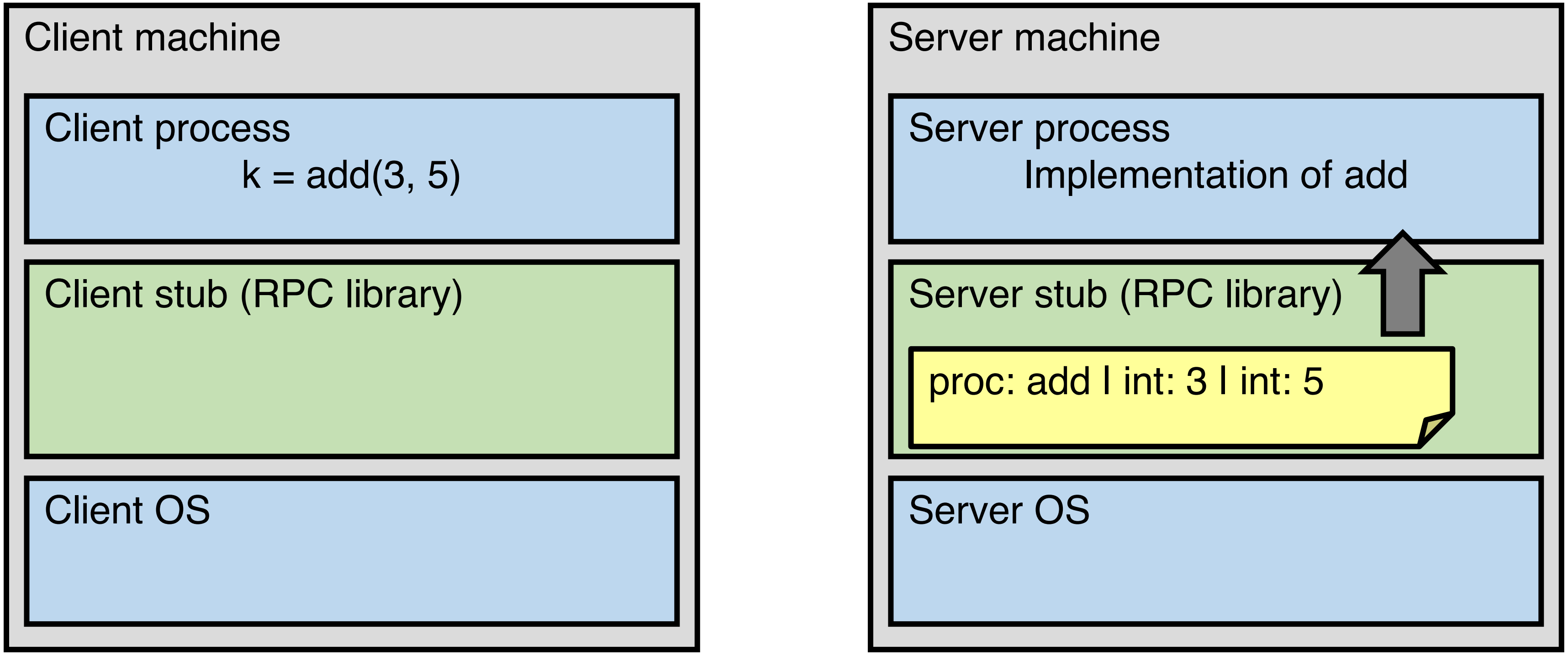
4. Server OS receives message, sends it up to stub


Client machine
Client process
k = add(3, 5)
Client stub (RPC library)
Client OS
Server machine
Server stub (RPC library)
Server OS
proc: add | int: 3 | int: 5

A day in the life of an RPC

4. Server OS receives message, sends it up to stub

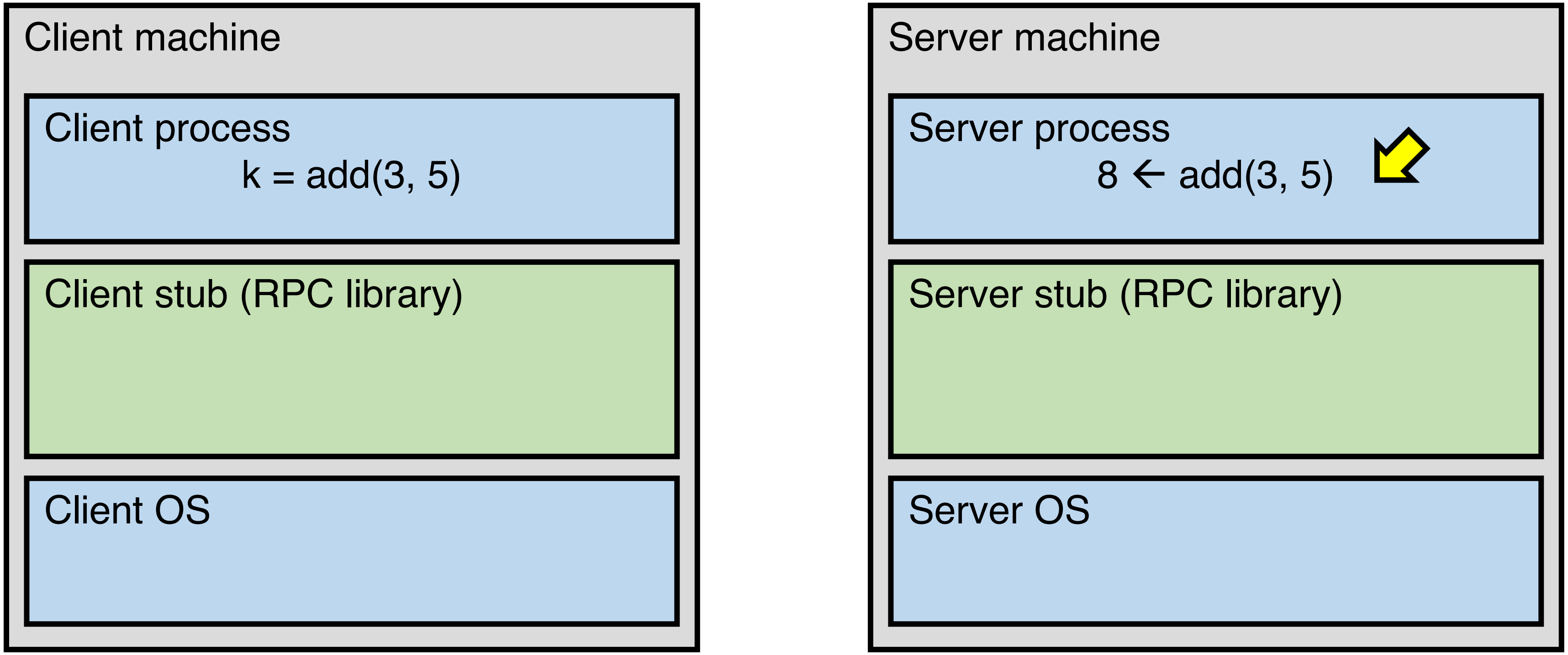
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

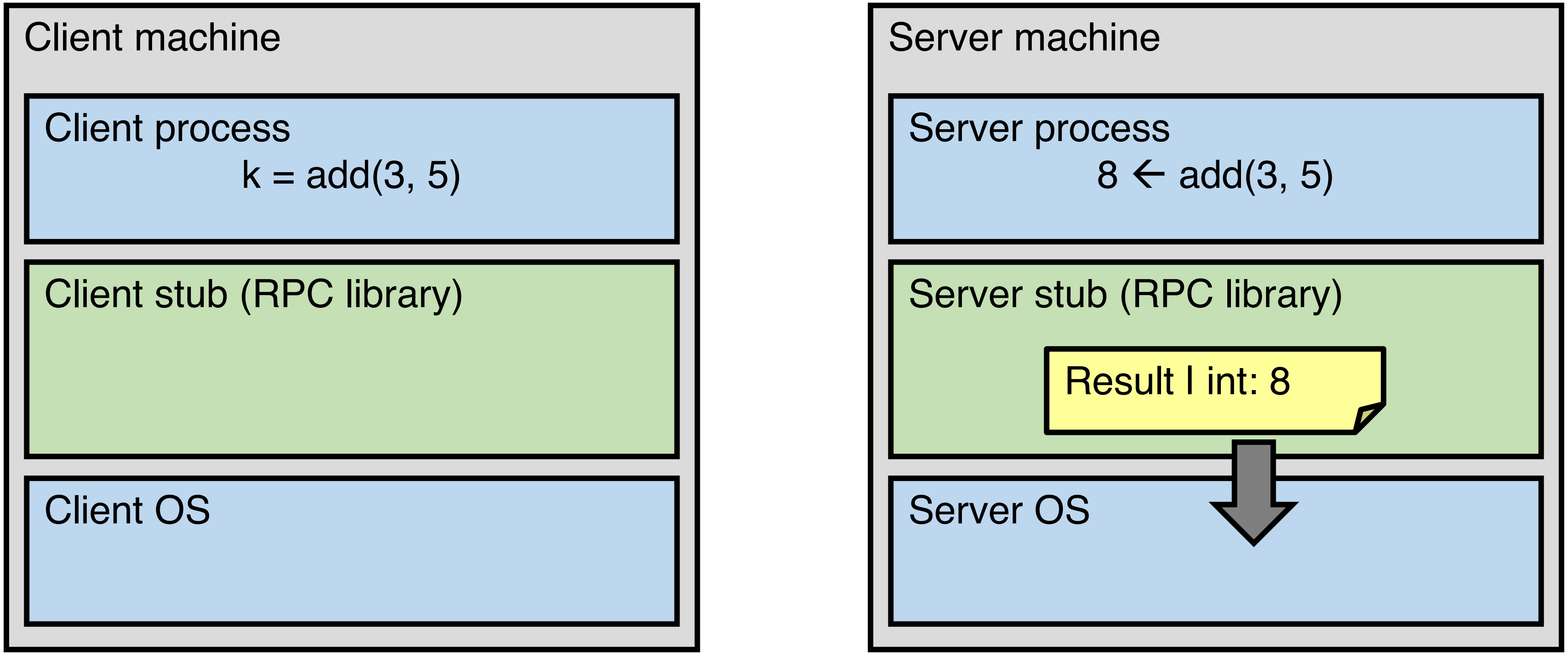
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

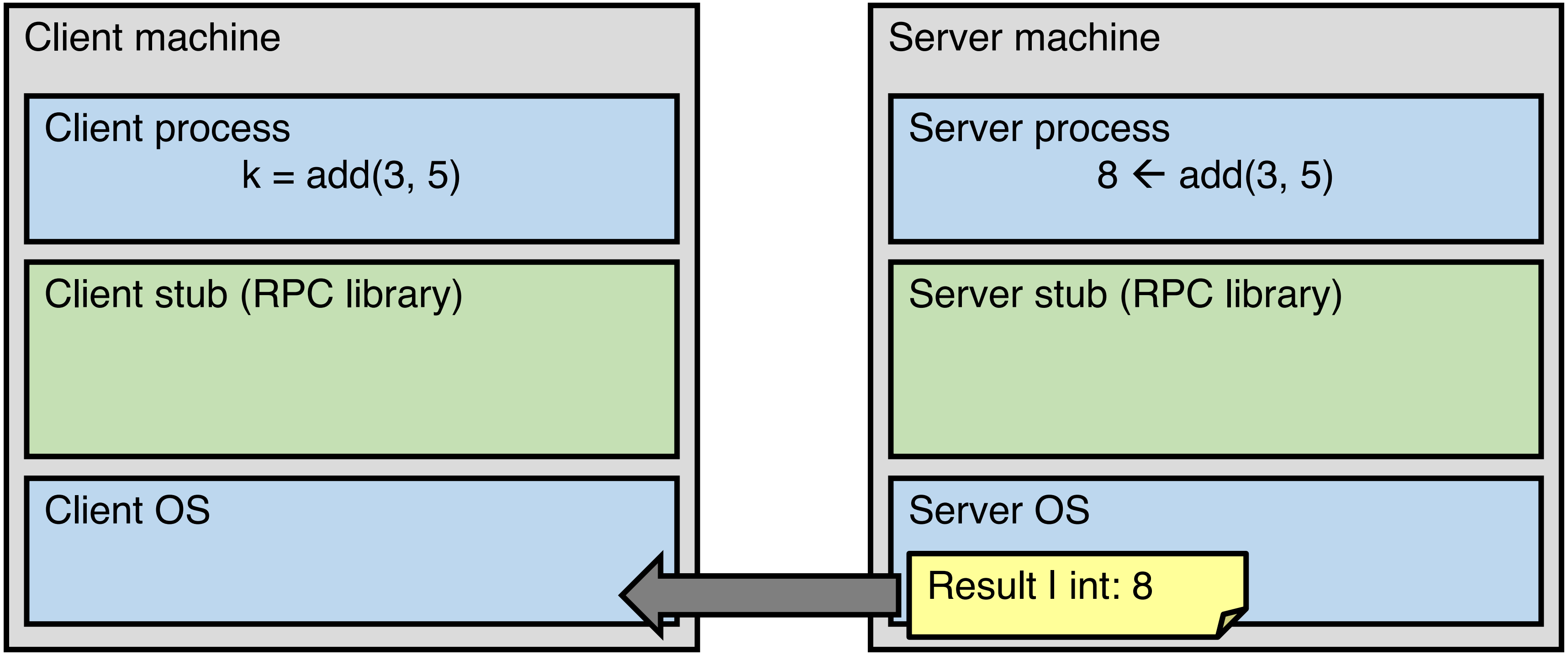
7. Server stub marshals the return value, sends message



A day in the life of an RPC

7. Server stub marshals the return value, sends message

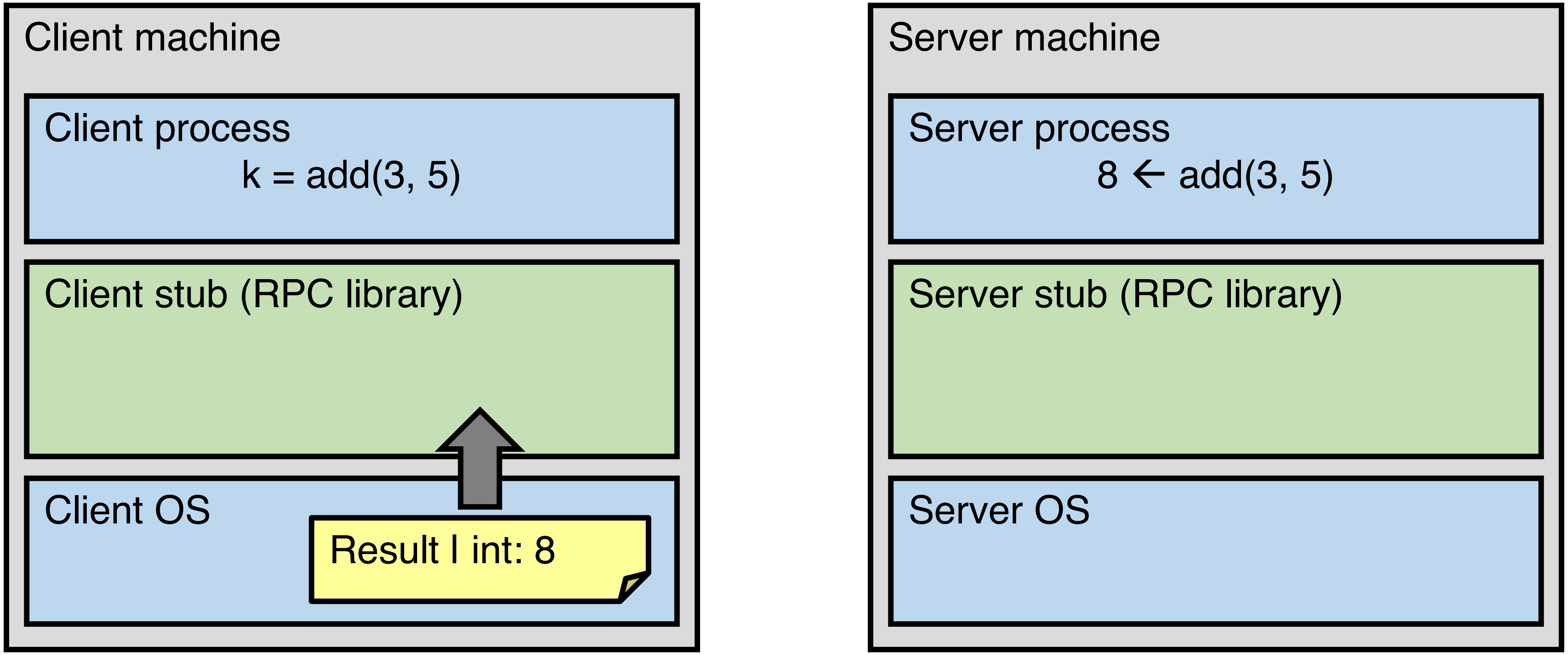
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

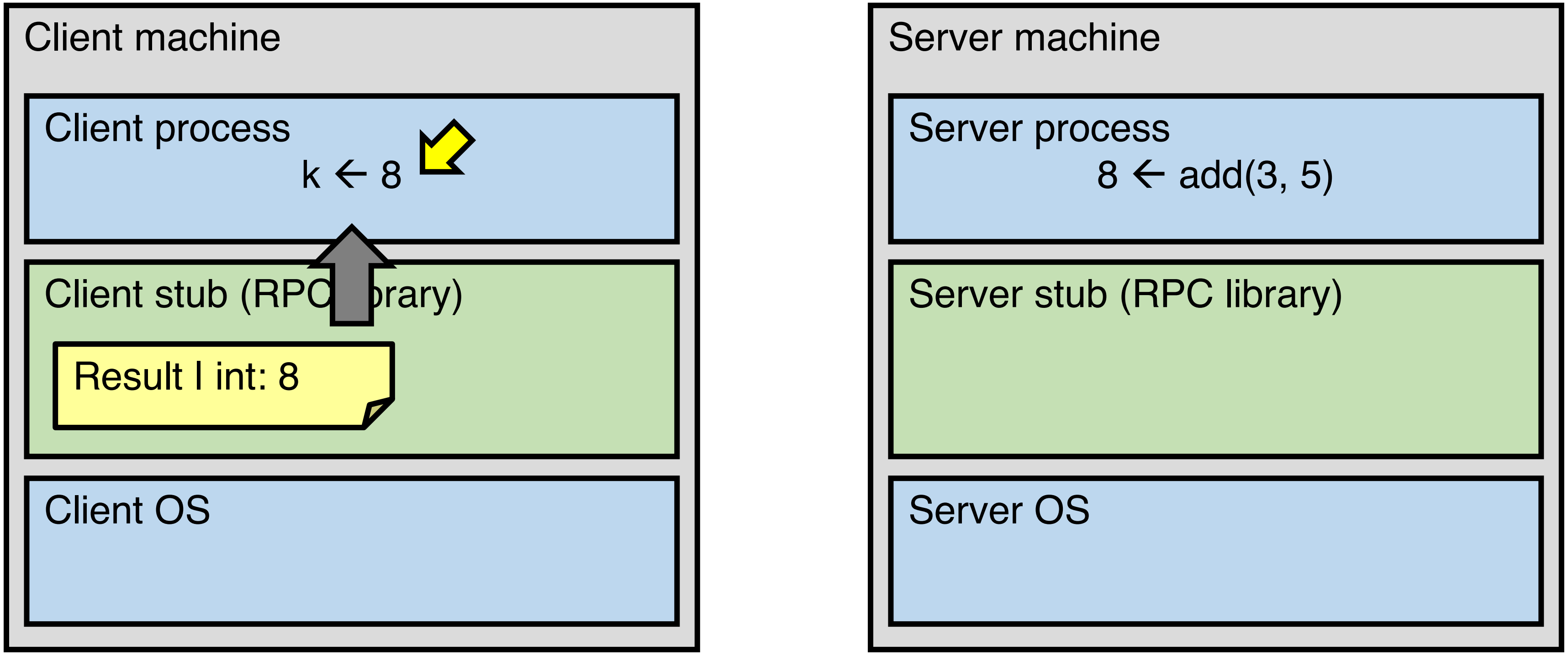
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

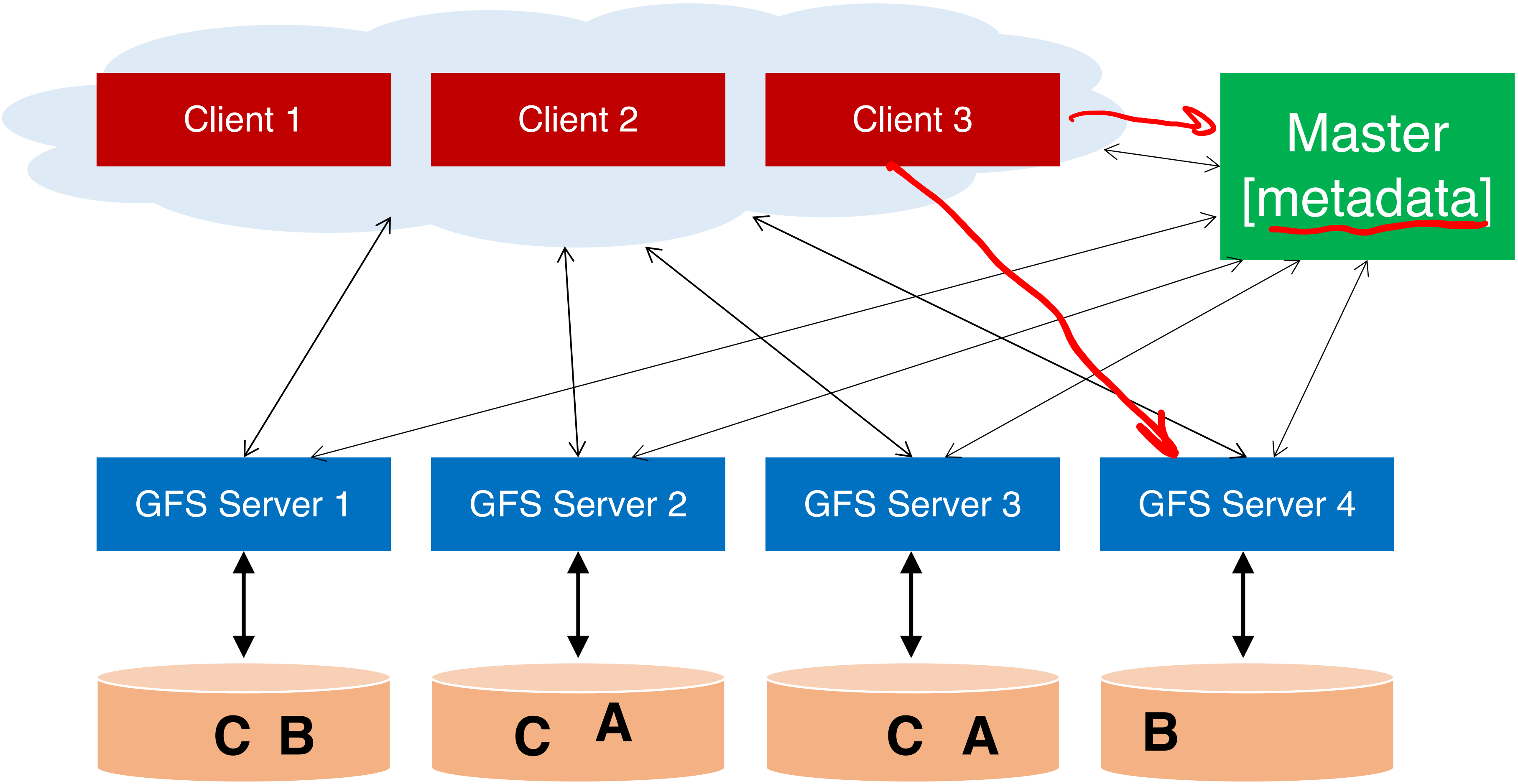
9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client



Then, get back to GFS

GFS architecture: physical view



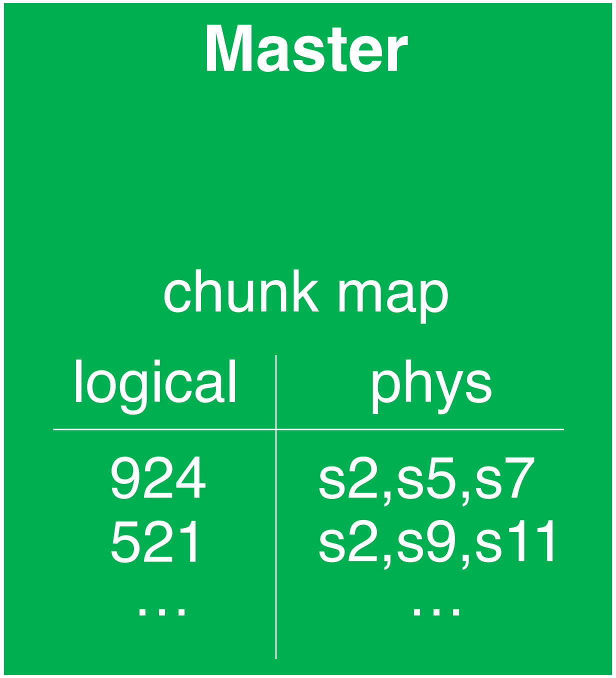
Data chunks

- Break large GFS files into coarse-grained data chunks (e.g., 64MB)

- GFS servers store physical data chunks in local Linux file system

- Centralized master keeps track of mapping between logical and physical chunks

Chunk map

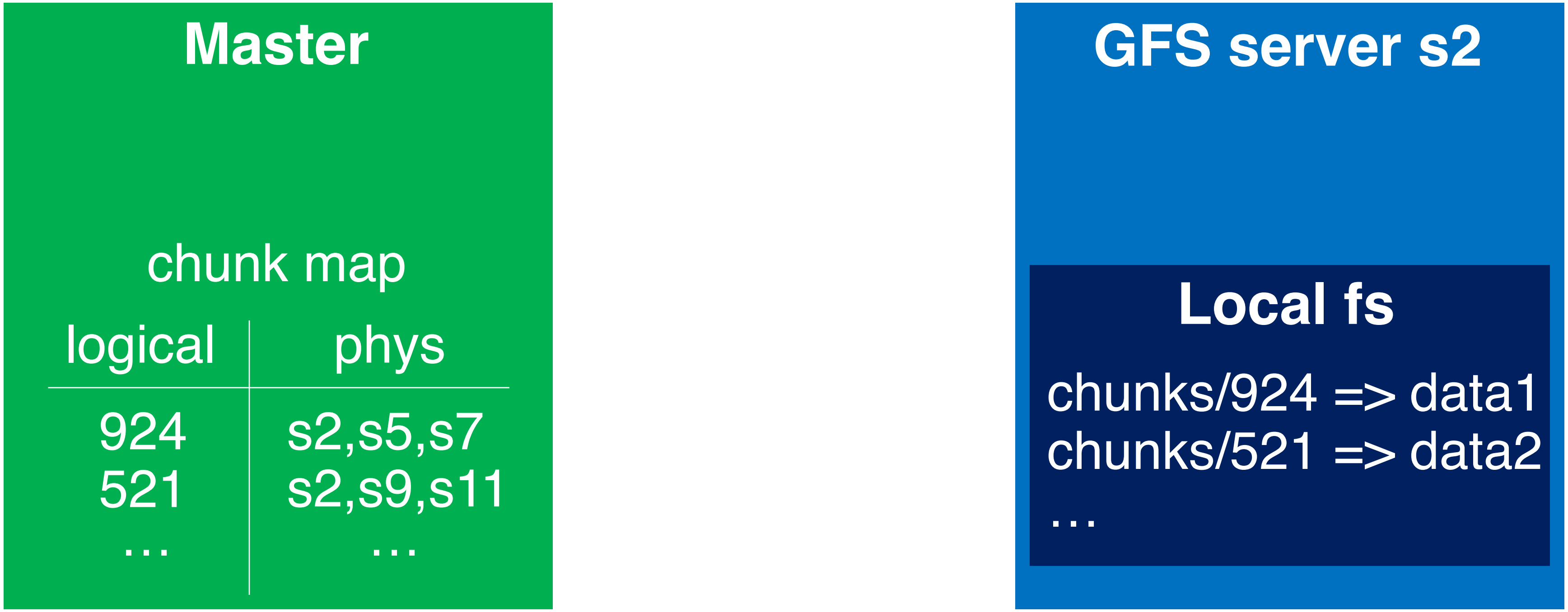


Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS server s2



Master

chunk map

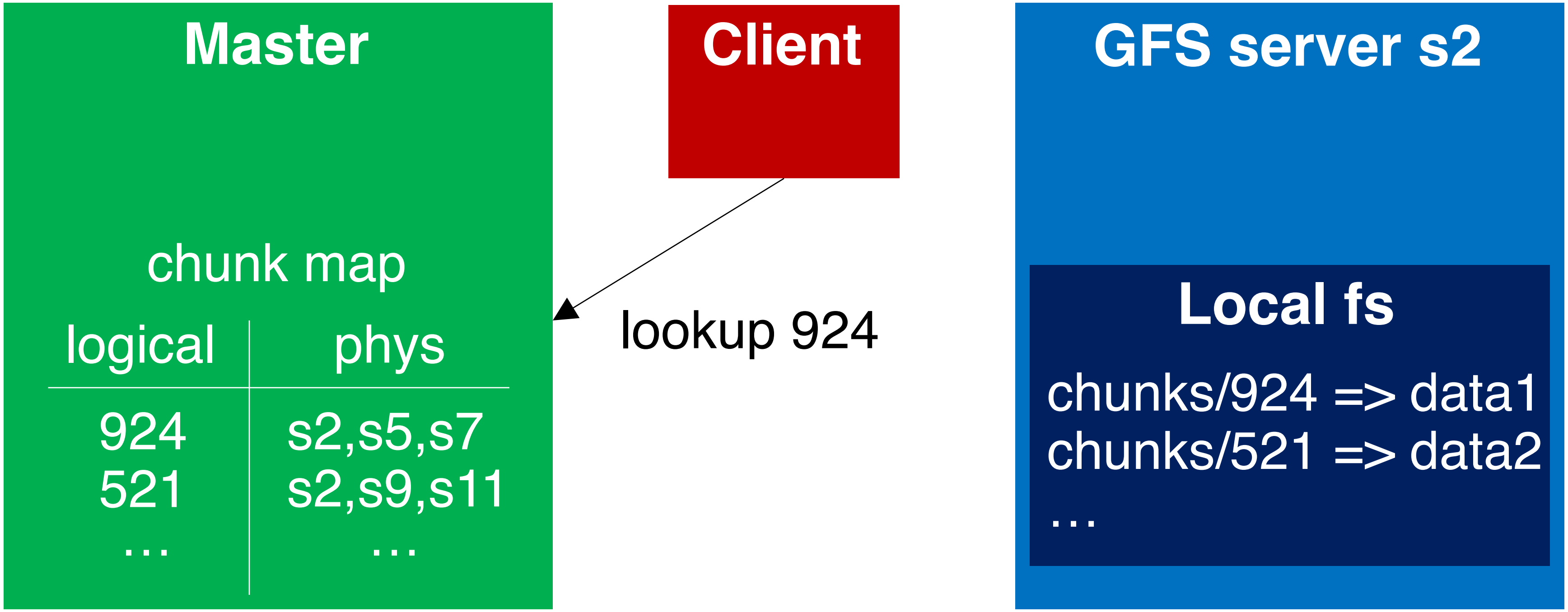
logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



Master

chunk map

logical	phys
924 521 …	s2,s5,s7 s2,s9,s11 …

Client

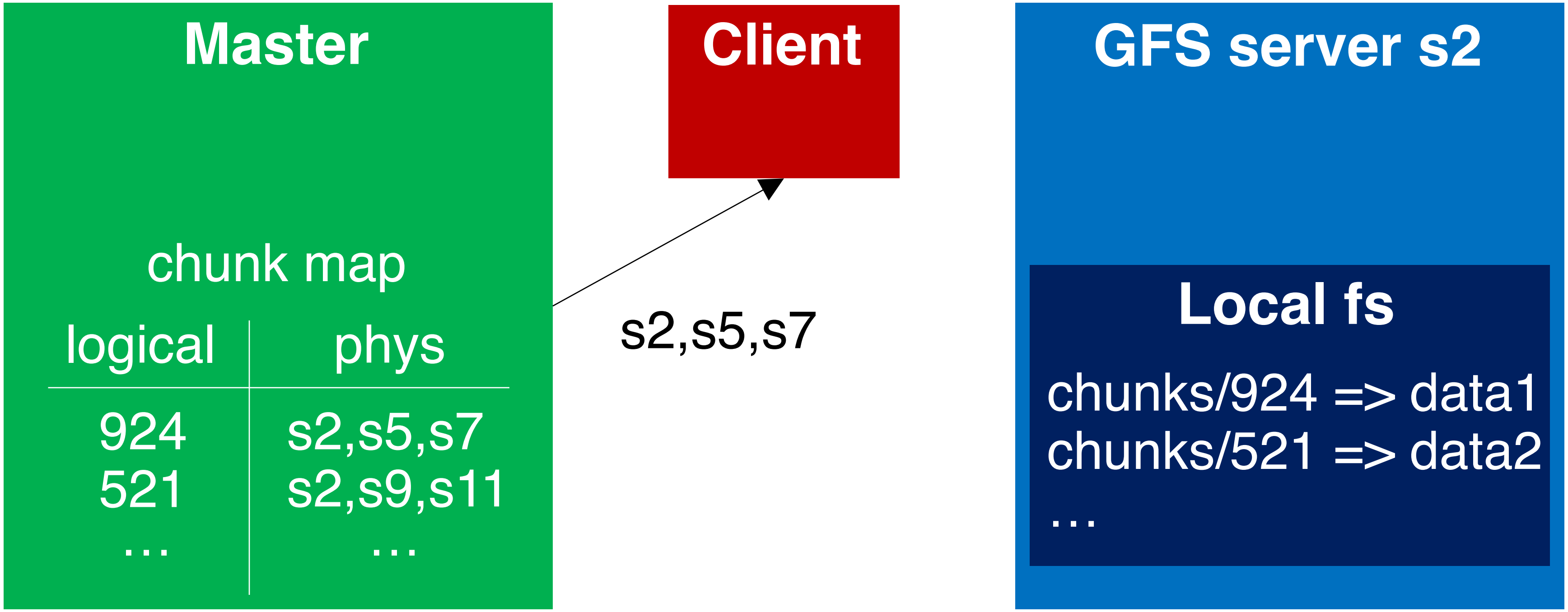
lookup 924

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

Client

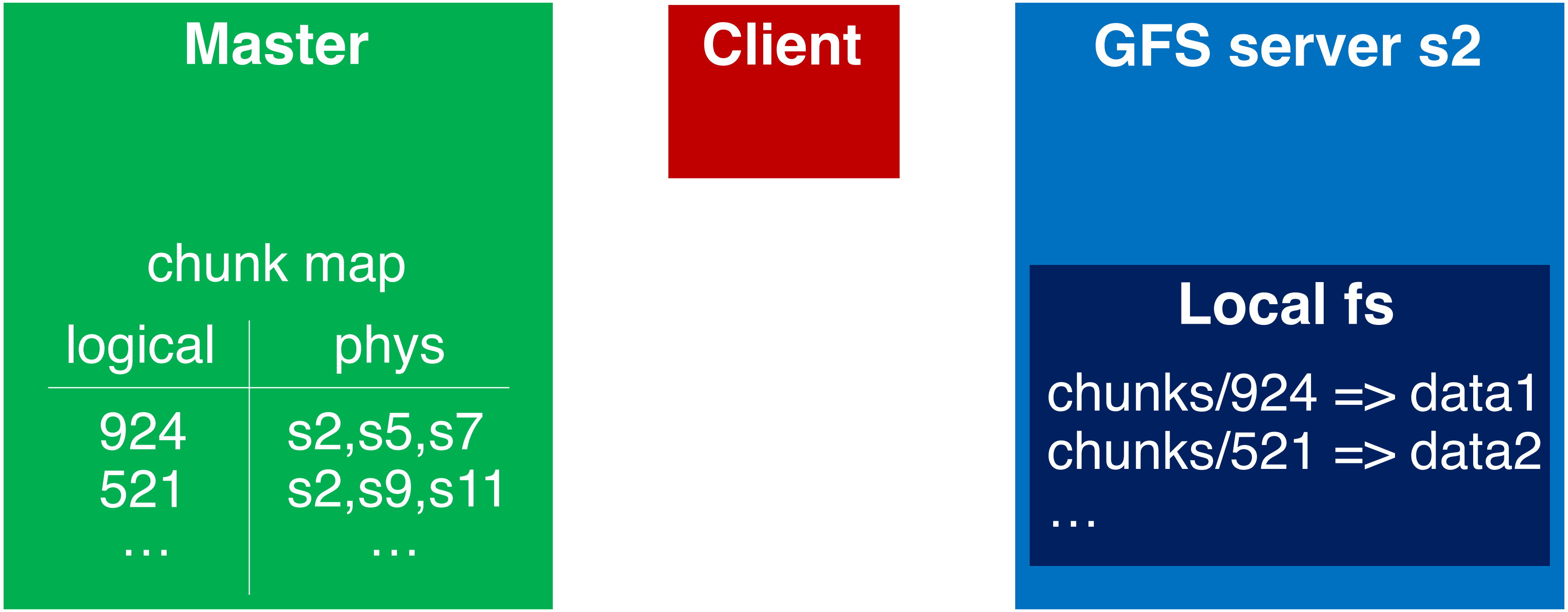
s2,s5,s7

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

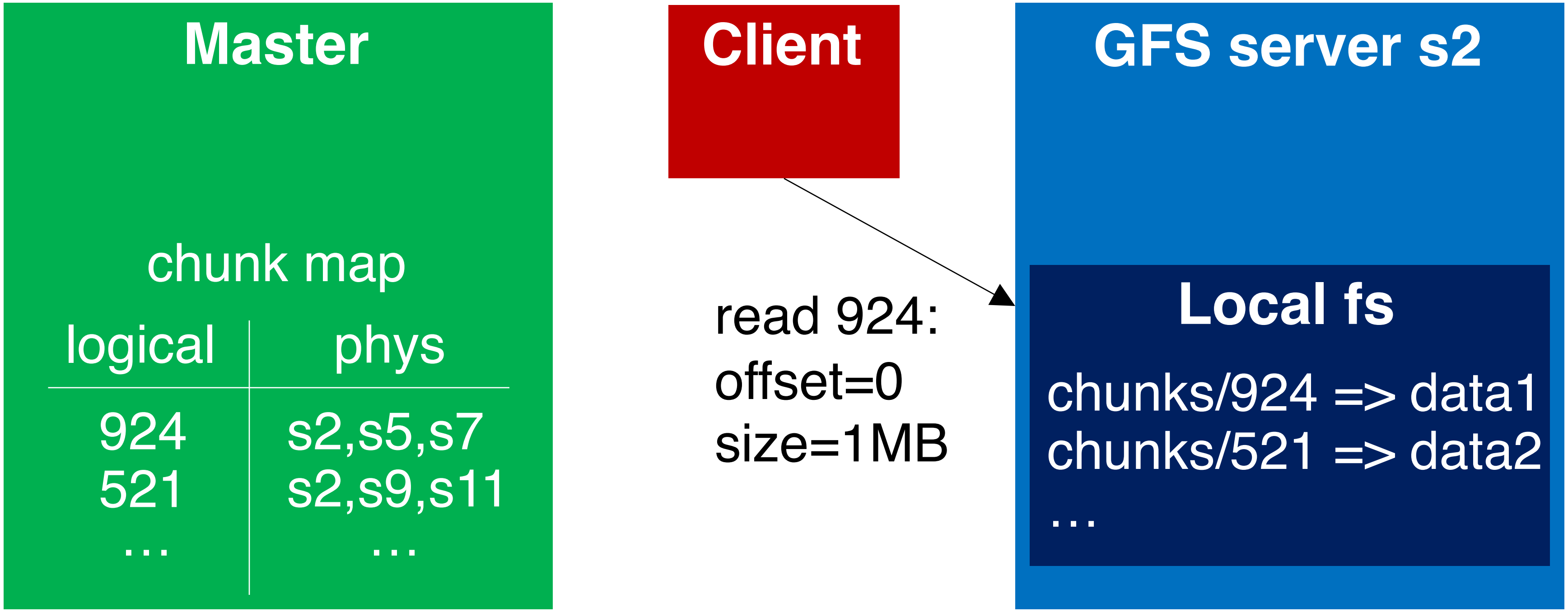
Client

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

Client

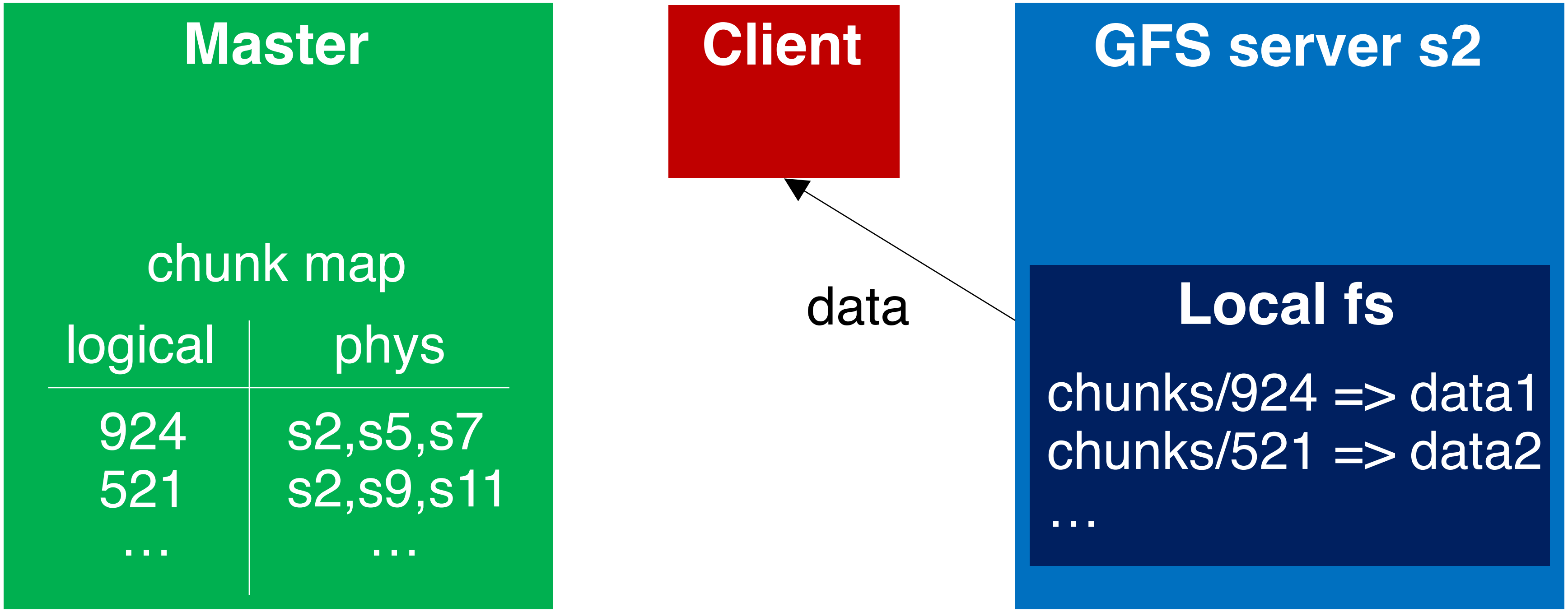
read 924:
offset=0
size=1MB

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

Client

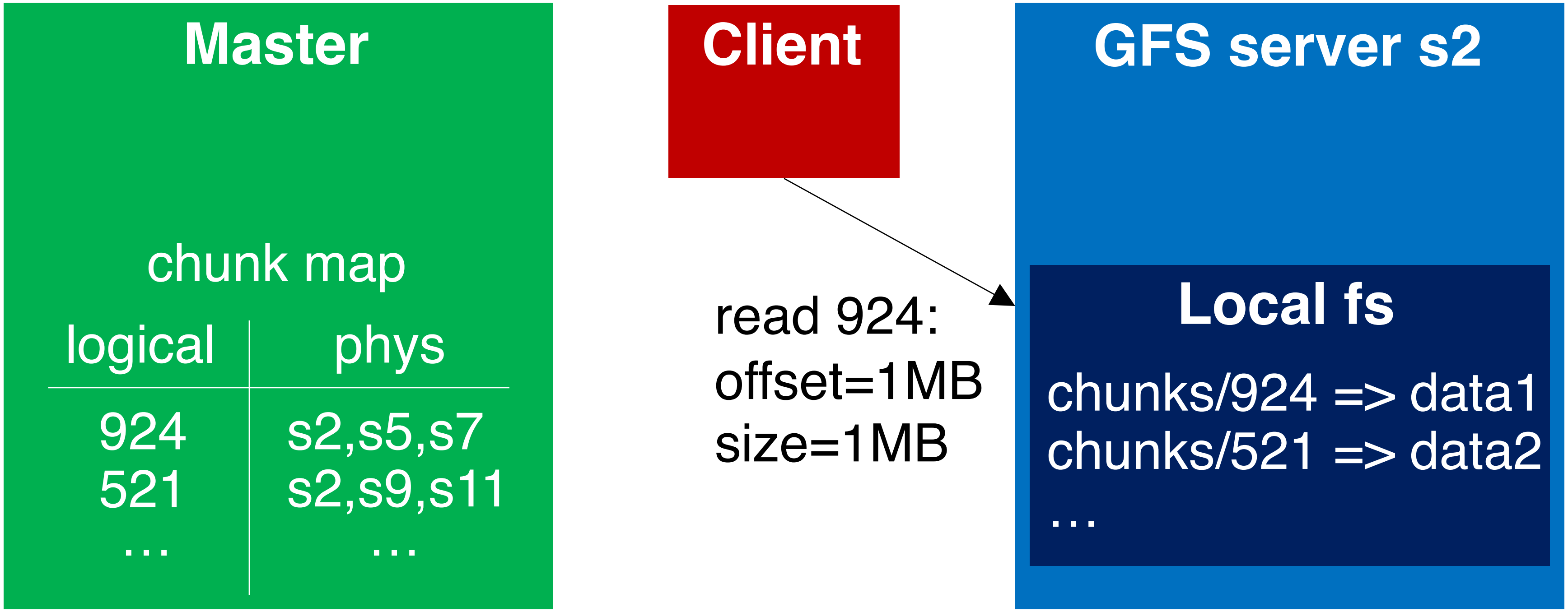
data

GFS server s2

Local fs

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk



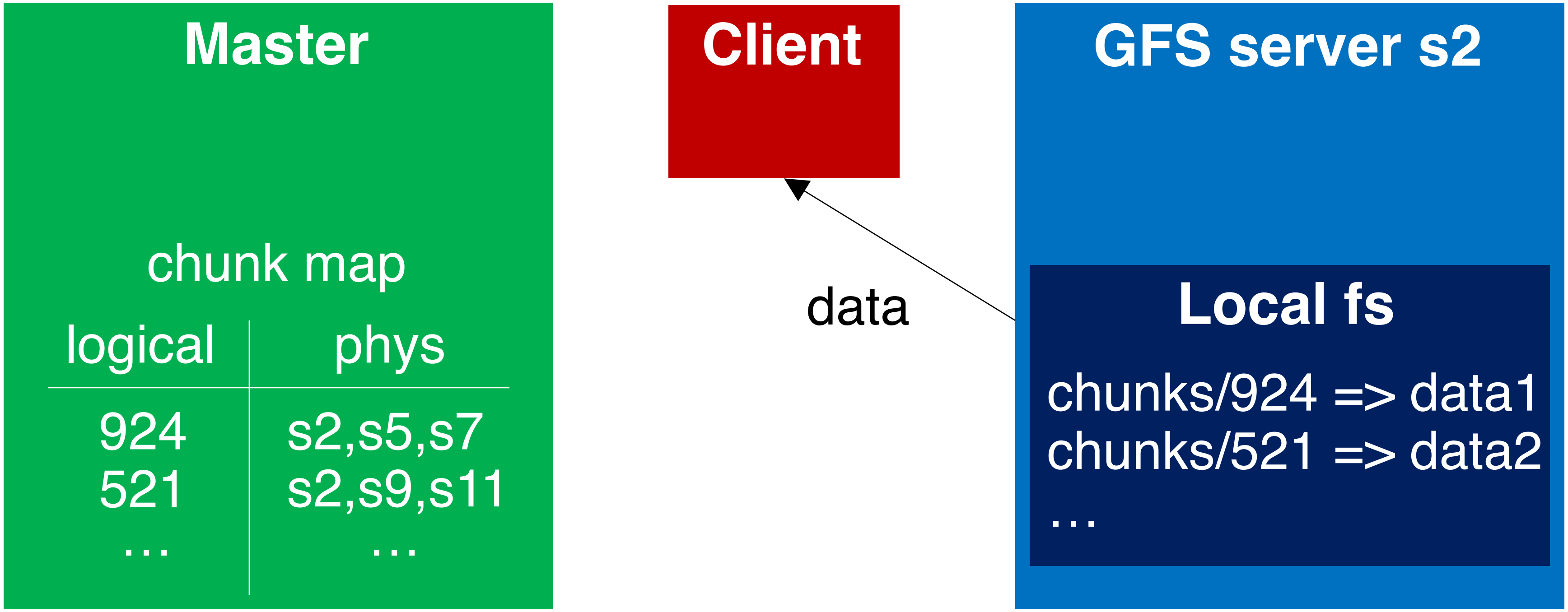
chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

read 924:
offset=1MB
size=1MB

chunks/924 => data1
chunks/521 => data2
…

Client reads a chunk

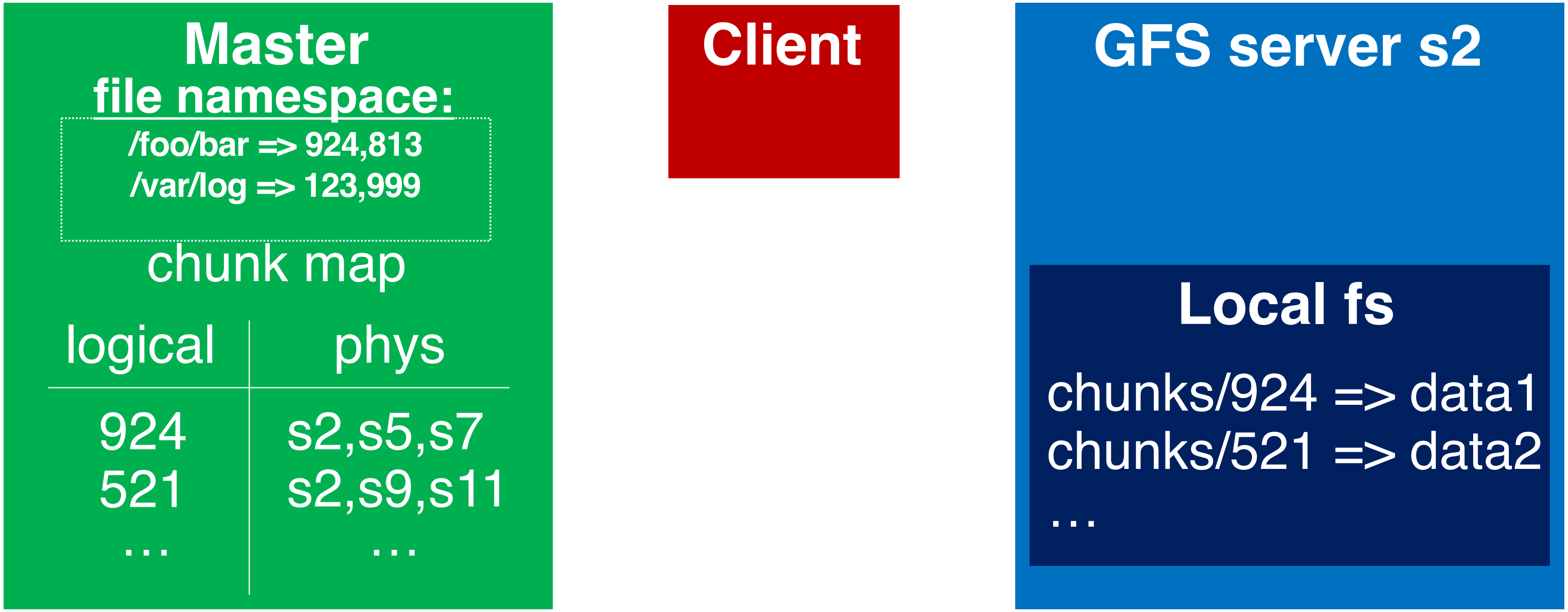


chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

chunks/924 => data1
chunks/521 => data2
…

File namespace



Master

file namespace:

/foo/bar => 924,813
/var/log => 123,999

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

Client

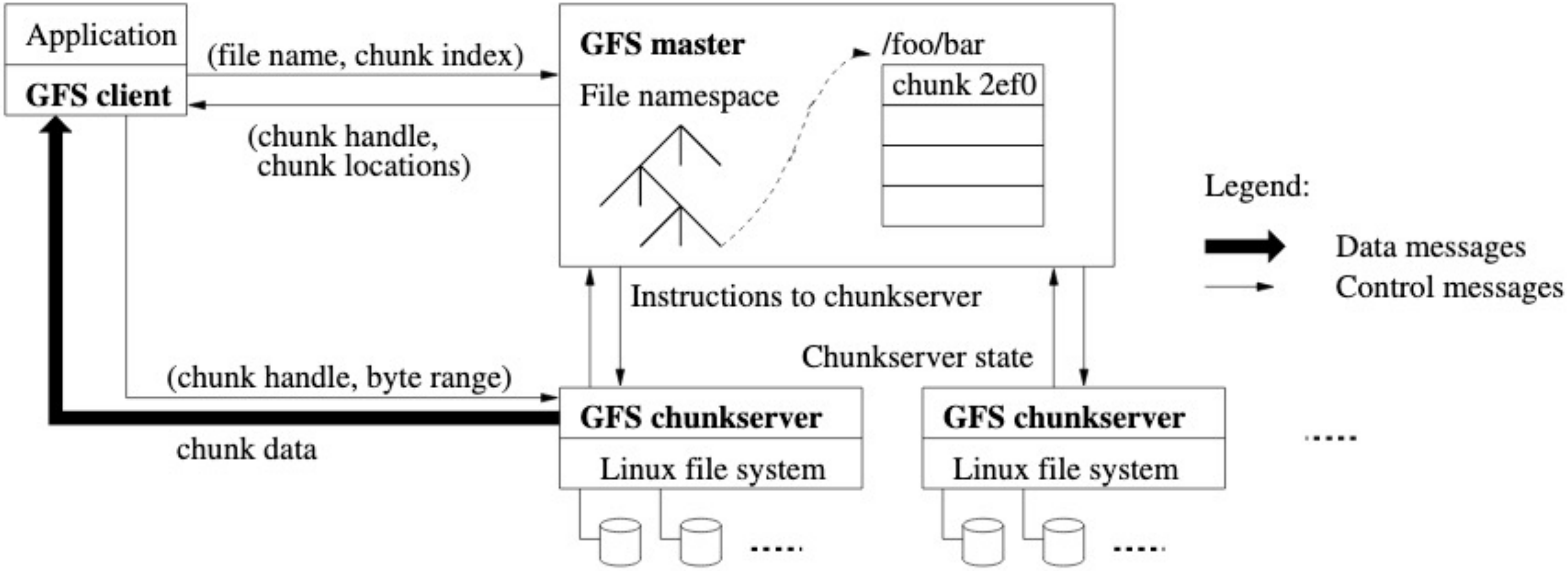
GFS server s2

Local fs

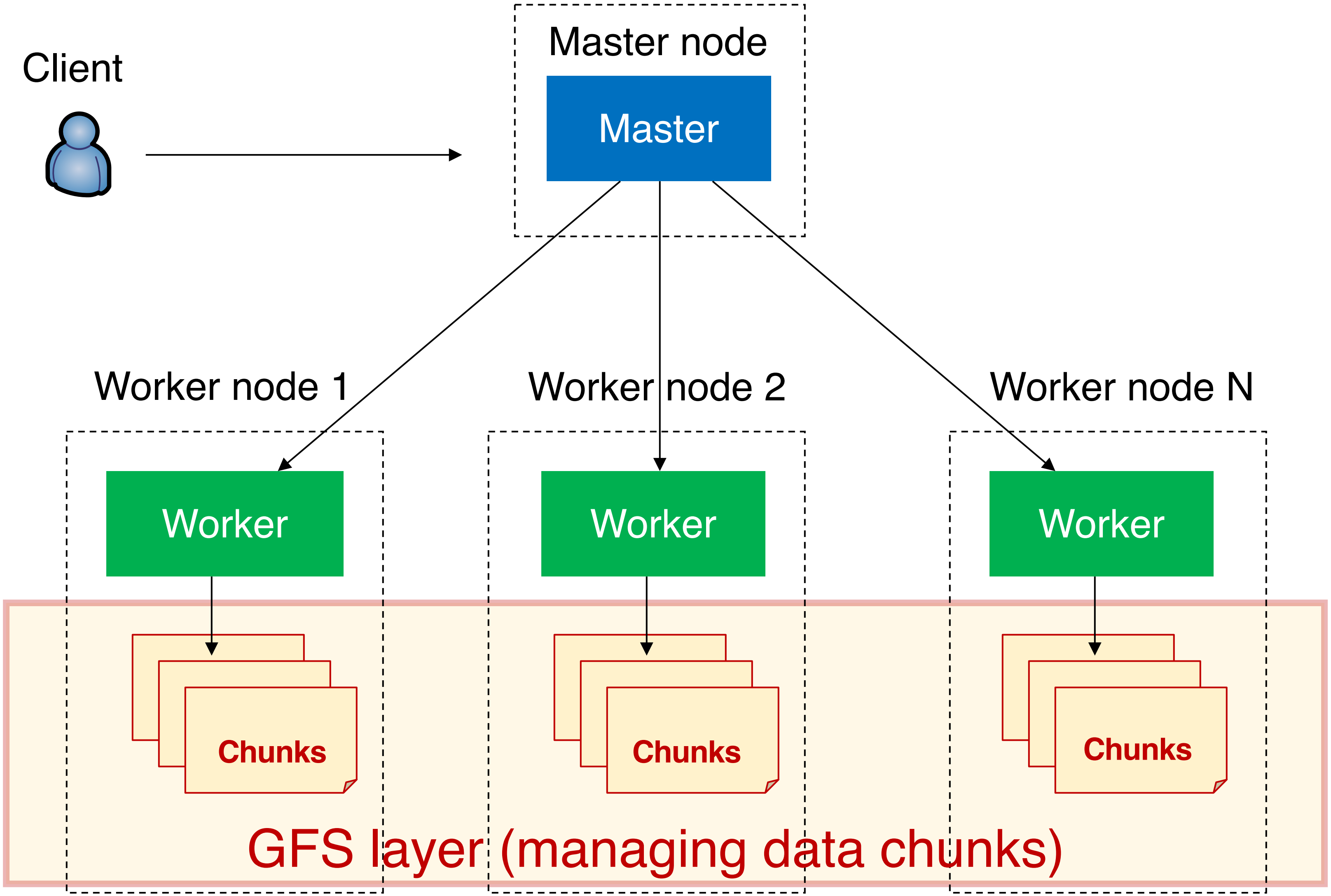
chunks/924 => data1
chunks/521 => data2
…

path names mapped to logical names

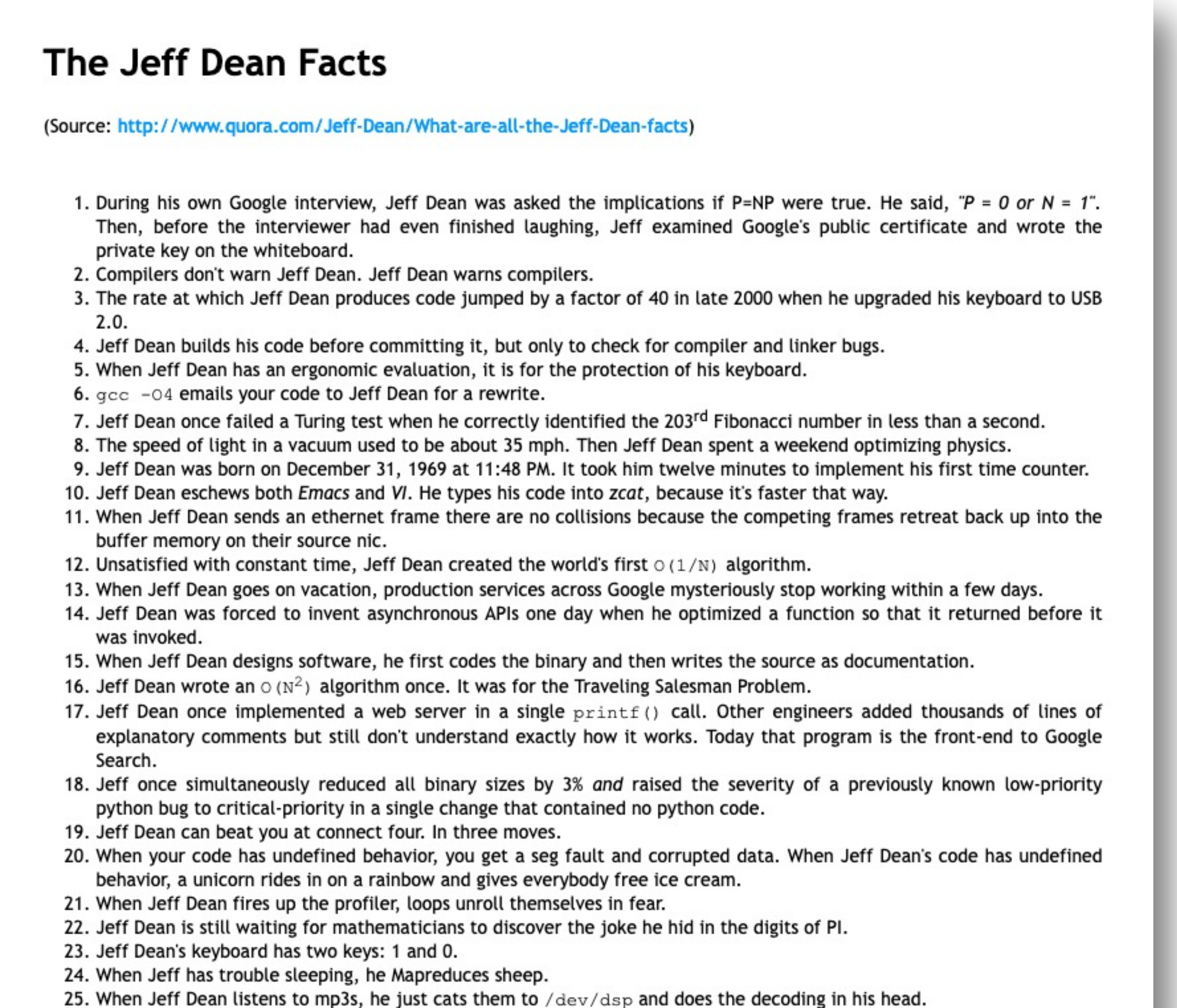
GFS architecture (original paper)



MapReduce+GFS: Put everything together



The Jeff Dean fact... 😄



The Jeff Dean Facts

(Source: http://www.quora.com/Jeff-Dean/What-are-all-the-Jeff-Dean-facts)

1. During his own Google interview, Jeff Dean was asked the implications if P=NP were true. He said, *"P = 0 or N = 1"*. Then, before the interviewer had even finished laughing, Jeff examined Google's public certificate and wrote the private key on the whiteboard.
2. Compilers don't warn Jeff Dean. Jeff Dean warns compilers.
3. The rate at which Jeff Dean produces code jumped by a factor of 40 in late 2000 when he upgraded his keyboard to USB 2.0.
4. Jeff Dean builds his code before committing it, but only to check for compiler and linker bugs.
5. When Jeff Dean has an ergonomic evaluation, it is for the protection of his keyboard.
6. `gcc -O4` emails your code to Jeff Dean for a rewrite.
7. Jeff Dean once failed a Turing test when he correctly identified the 203rd Fibonacci number in less than a second.
8. The speed of light in a vacuum used to be about 35 mph. Then Jeff Dean spent a weekend optimizing physics.
9. Jeff Dean was born on December 31, 1969 at 11:48 PM. It took him twelve minutes to implement his first time counter.
10. Jeff Dean eschews both *Emacs* and *VI*. He types his code into *zcat*, because it's faster that way.
11. When Jeff Dean sends an ethernet frame there are no collisions because the competing frames retreat back up into the buffer memory on their source nic.
12. Unsatisfied with constant time, Jeff Dean created the world's first `O(1/N)` algorithm.
13. When Jeff Dean goes on vacation, production services across Google mysteriously stop working within a few days.
14. Jeff Dean was forced to invent asynchronous APIs one day when he optimized a function so that it returned before it was invoked.
15. When Jeff Dean designs software, he first codes the binary and then writes the source as documentation.
16. Jeff Dean wrote an $O(N^2)$ algorithm once. It was for the Traveling Salesman Problem.
17. Jeff Dean once implemented a web server in a single `printf()` call. Other engineers added thousands of lines of explanatory comments but still don't understand exactly how it works. Today that program is the front-end to Google Search.
18. Jeff once simultaneously reduced all binary sizes by 3% *and* raised the severity of a previously known low-priority python bug to critical-priority in a single change that contained no python code.
19. Jeff Dean can beat you at connect four. In three moves.
20. When your code has undefined behavior, you get a seg fault and corrupted data. When Jeff Dean's code has undefined behavior, a unicorn rides in on a rainbow and gives everybody free ice cream.
21. When Jeff Dean fires up the profiler, loops unroll themselves in fear.
22. Jeff Dean is still waiting for mathematicians to discover the joke he hid in the digits of PI.
23. Jeff Dean's keyboard has two keys: 1 and 0.
24. When Jeff has trouble sleeping, he Mapreduces sheep.
25. When Jeff Dean listens to mp3s, he just cats them to `/dev/dsp` and does the decoding in his head.