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.

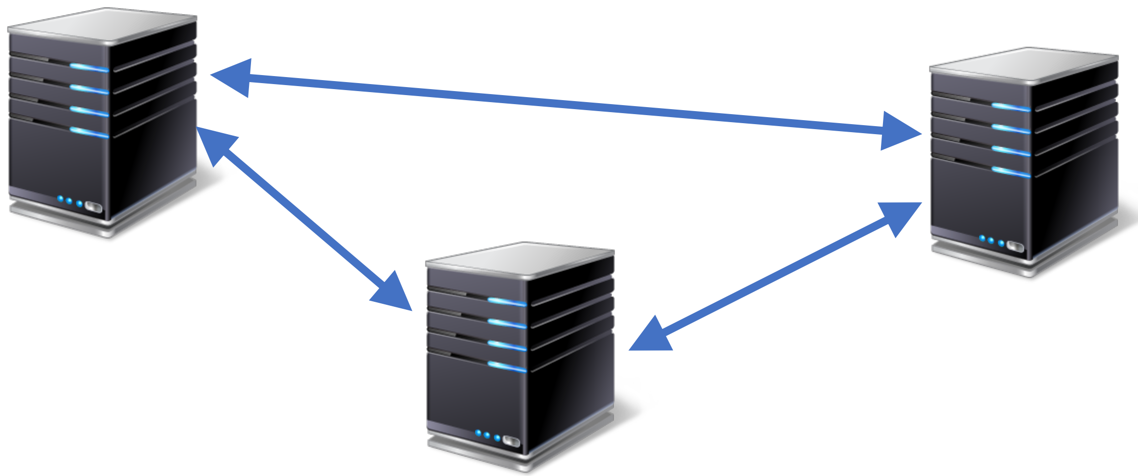
Announcement

- 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

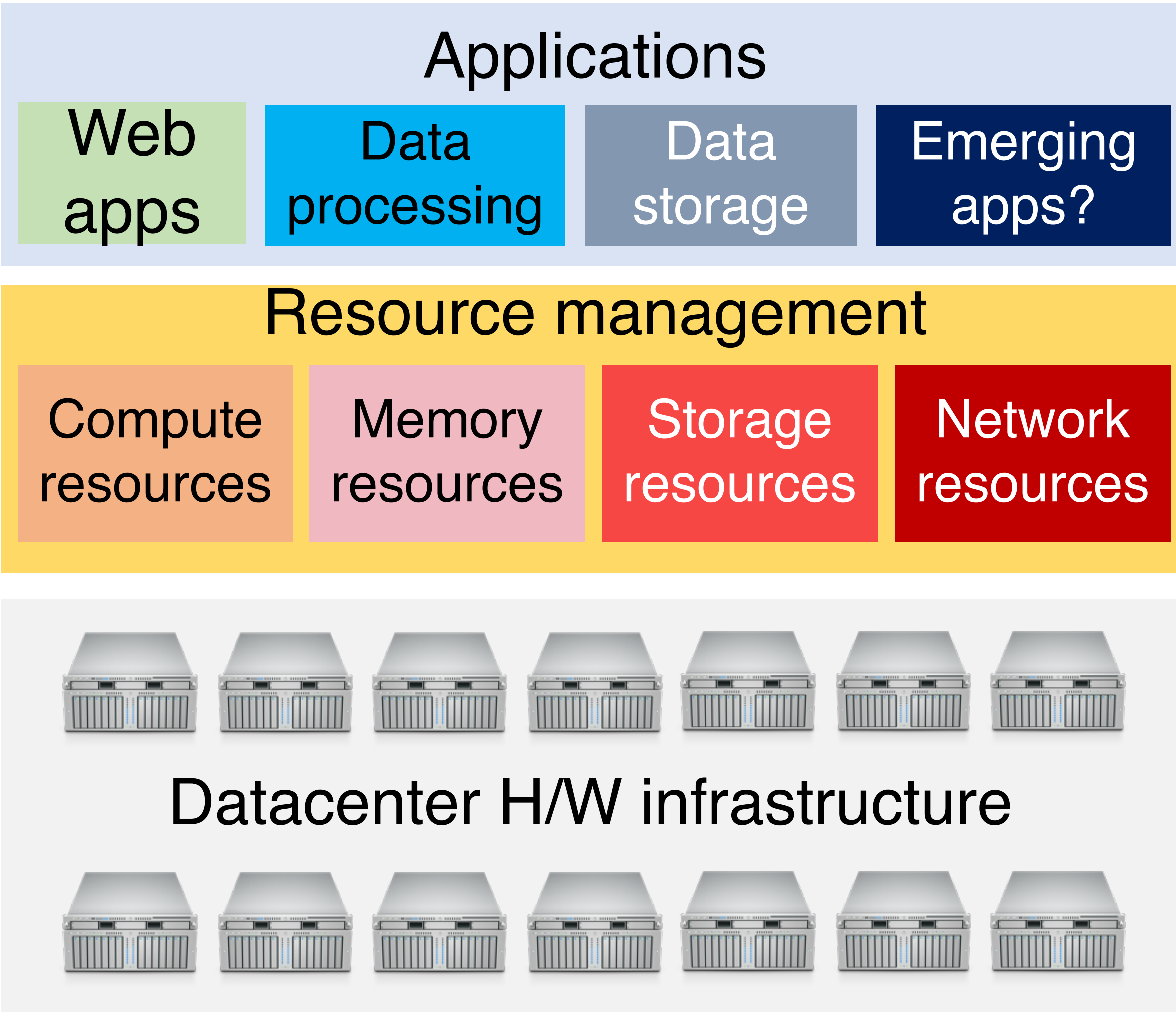
2. Google File System

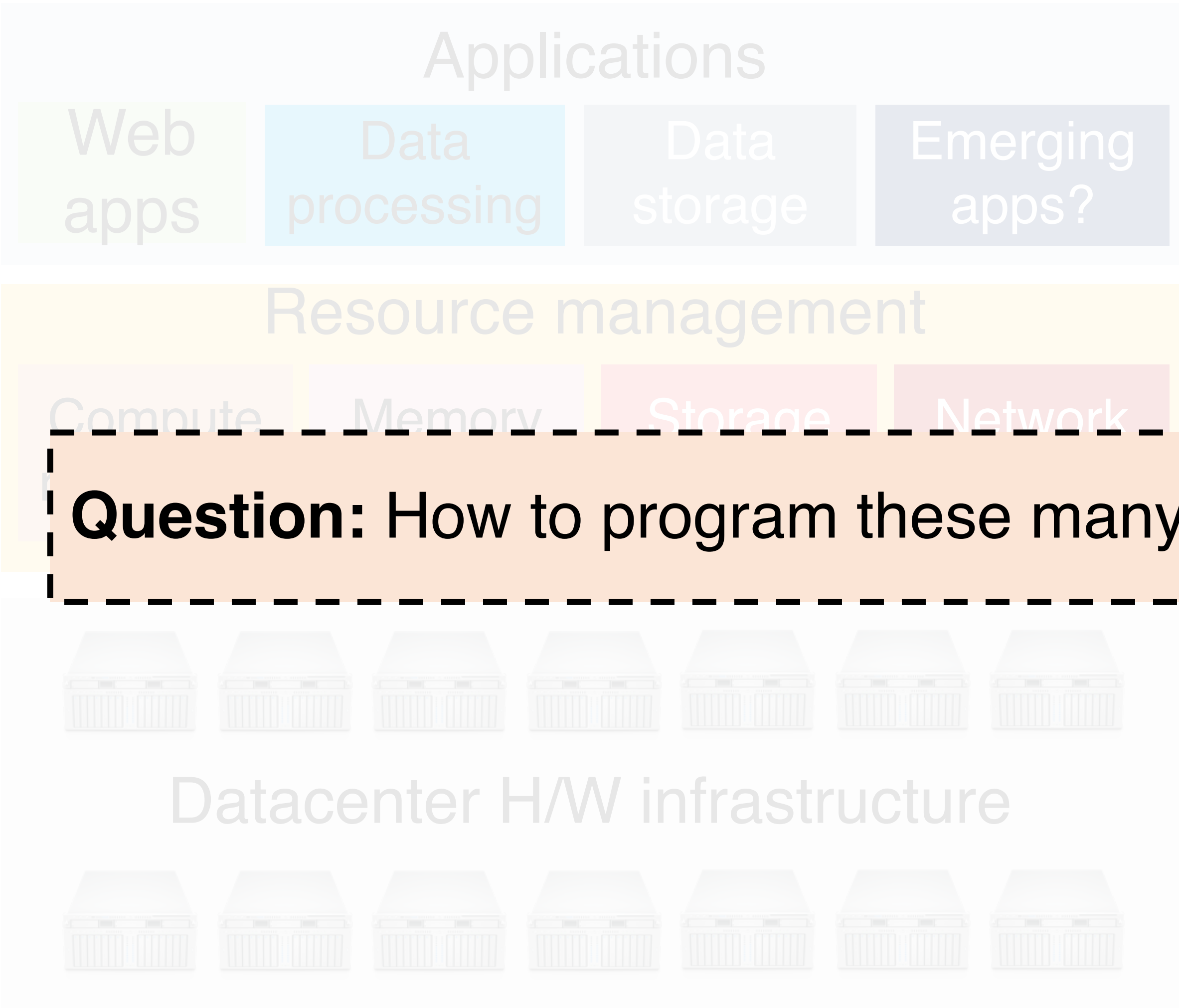
Today's outline

How can large computing jobs be parallelized?

1. MapReduce

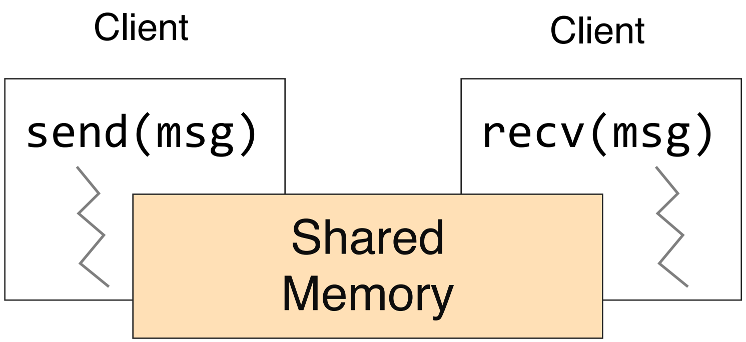
2. Google File System


Applications
Web apps
Data processing
Data storage
Emerging apps?
Resource management
Compute resources
Memory resources
Storage resources
Network resources
Datacenter H/W infrastructure



Question: How to program these many computers?

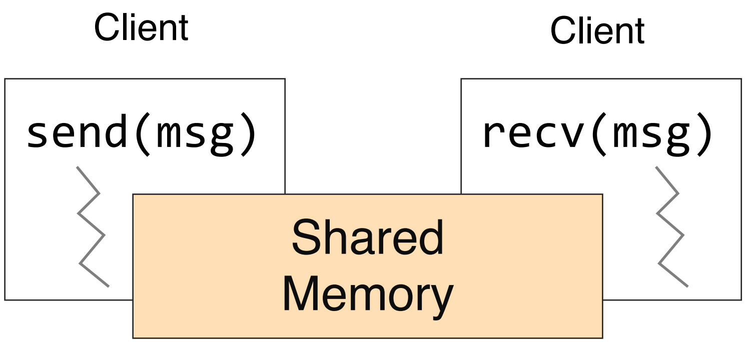
Review: Shared memory



- Shared memory: multiple processes to share data via memory

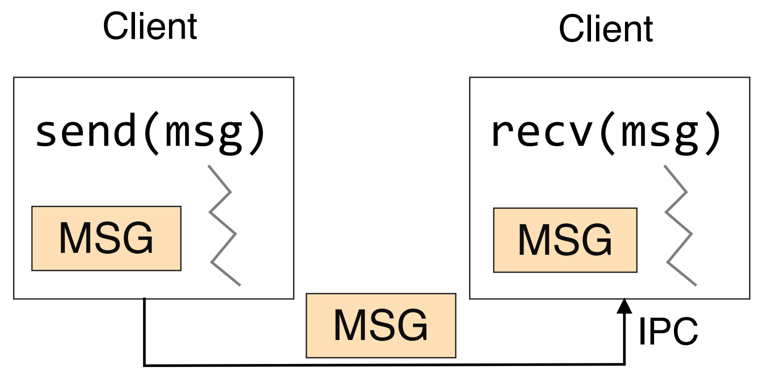
- Applications must locate and and map shared memory regions to exchange data

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**

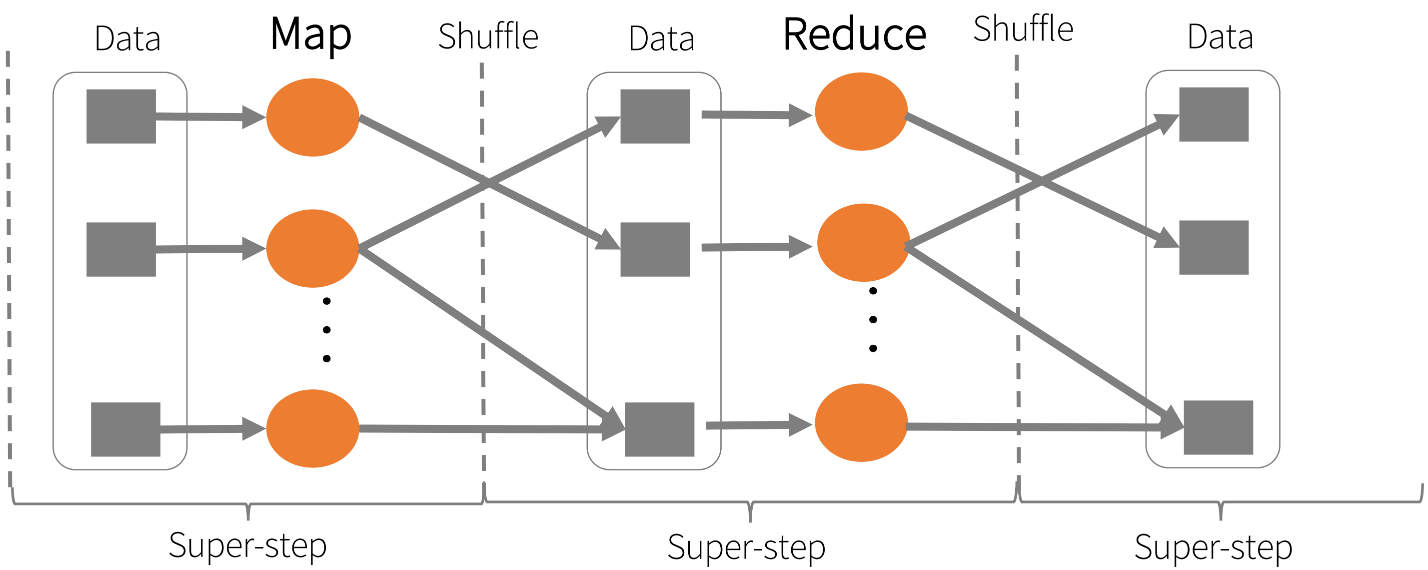
Context (Google circa 2000)

- 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

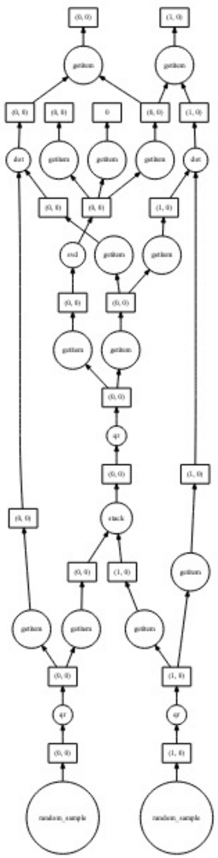
Context (Google circa 2000)

- 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

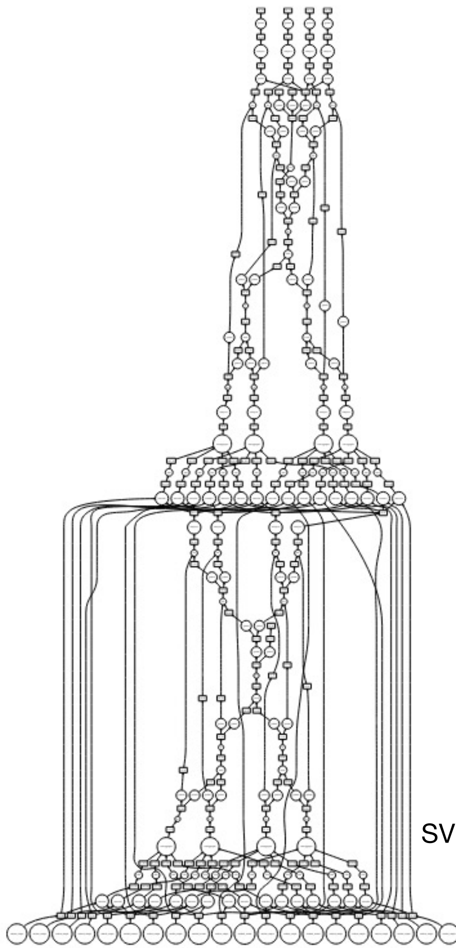
What MapReduce is good at?



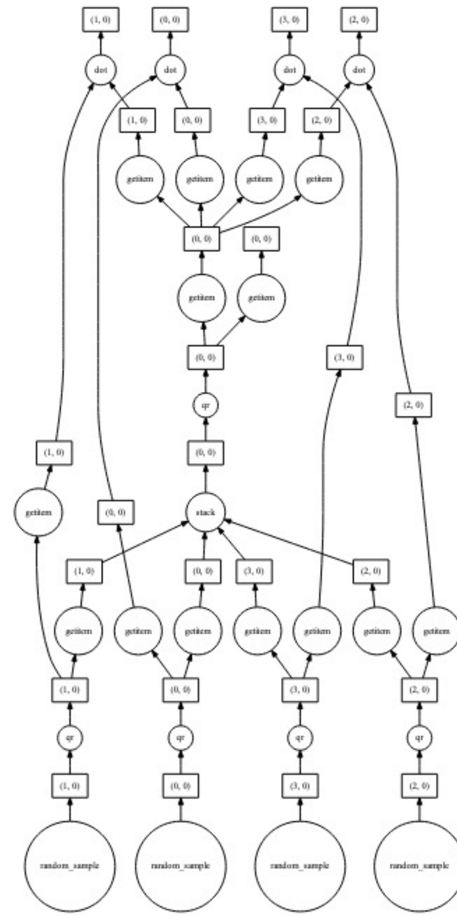
What MapReduce is not good at?



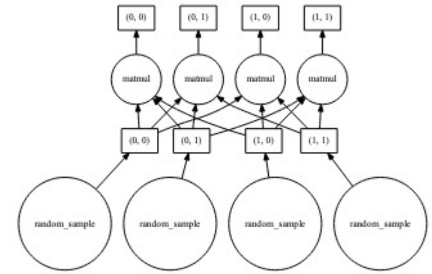
SVD1



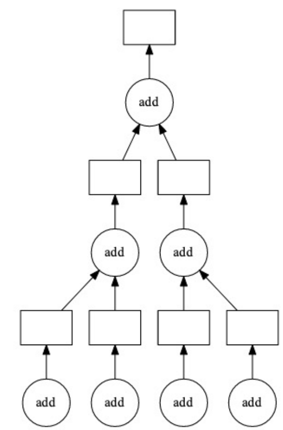
SVD2



TSQR



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

2. Then merge intermediate output

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`)

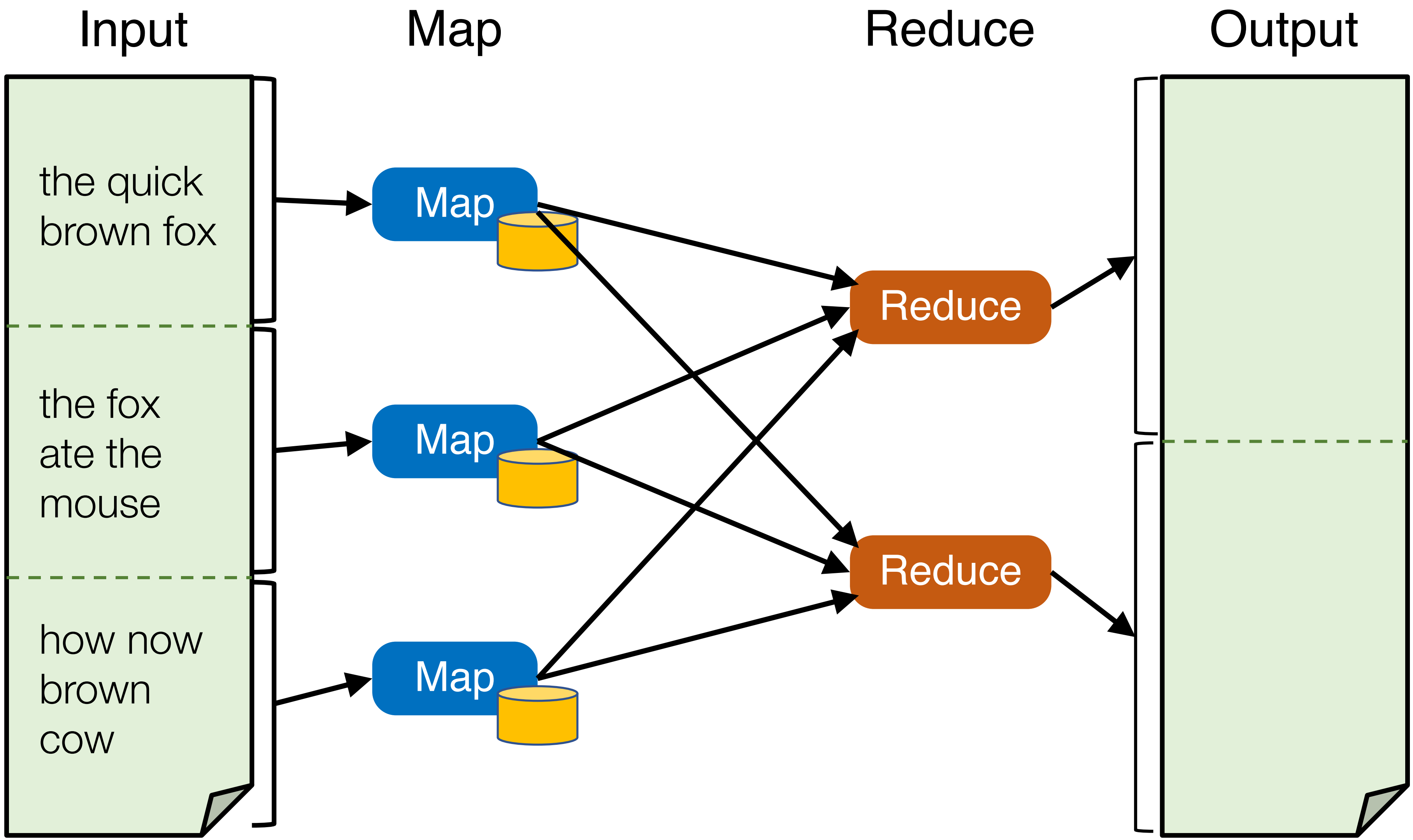
- `reduce(k2, list(v2))` → `list(k3, v3)`
  - Apply aggregation (reduce) function to values
  - Output results

MapReduce: Word Count

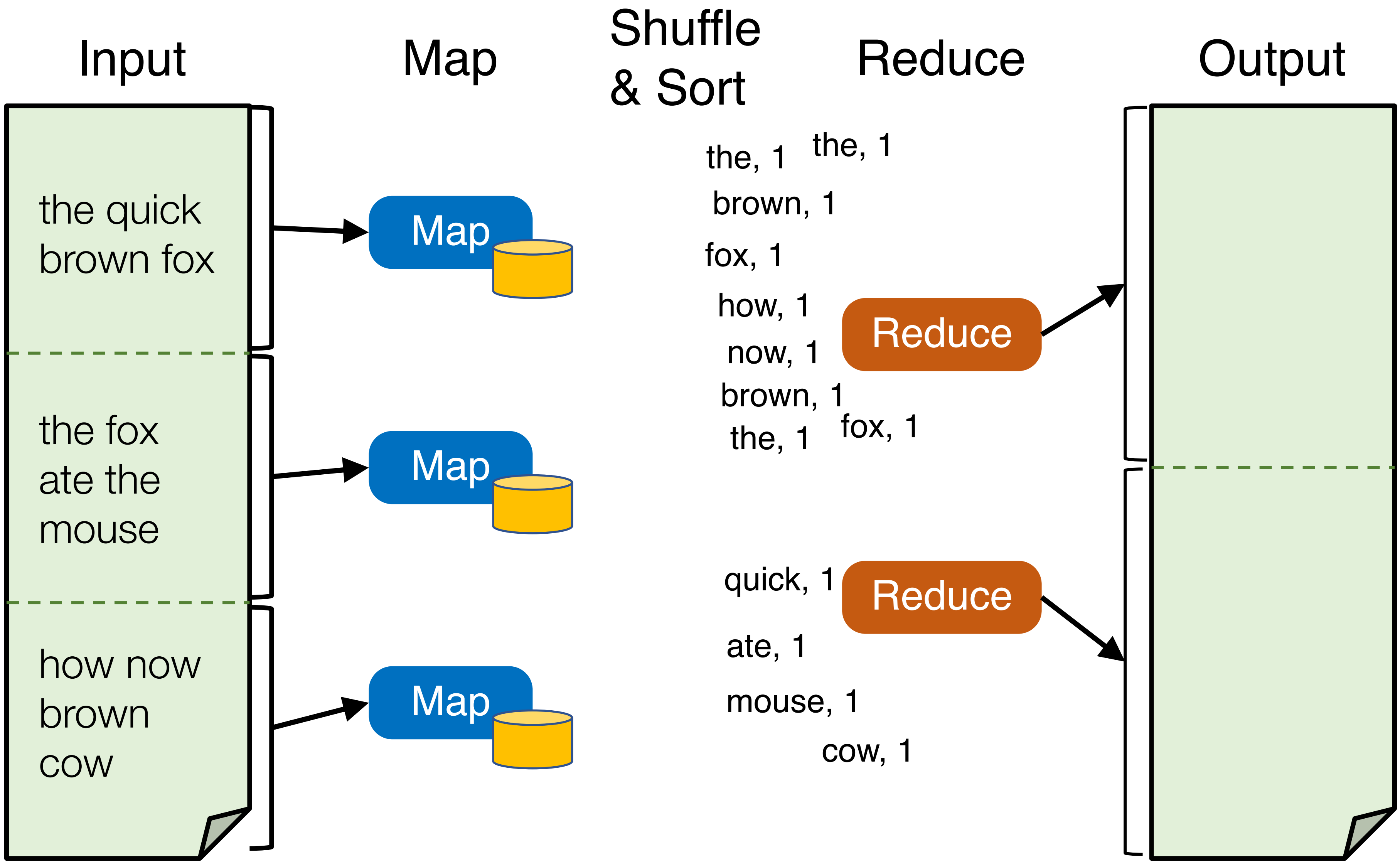
```
map(key, value):
    for each word w in value:
        EmitIntermediate(w, "1");

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

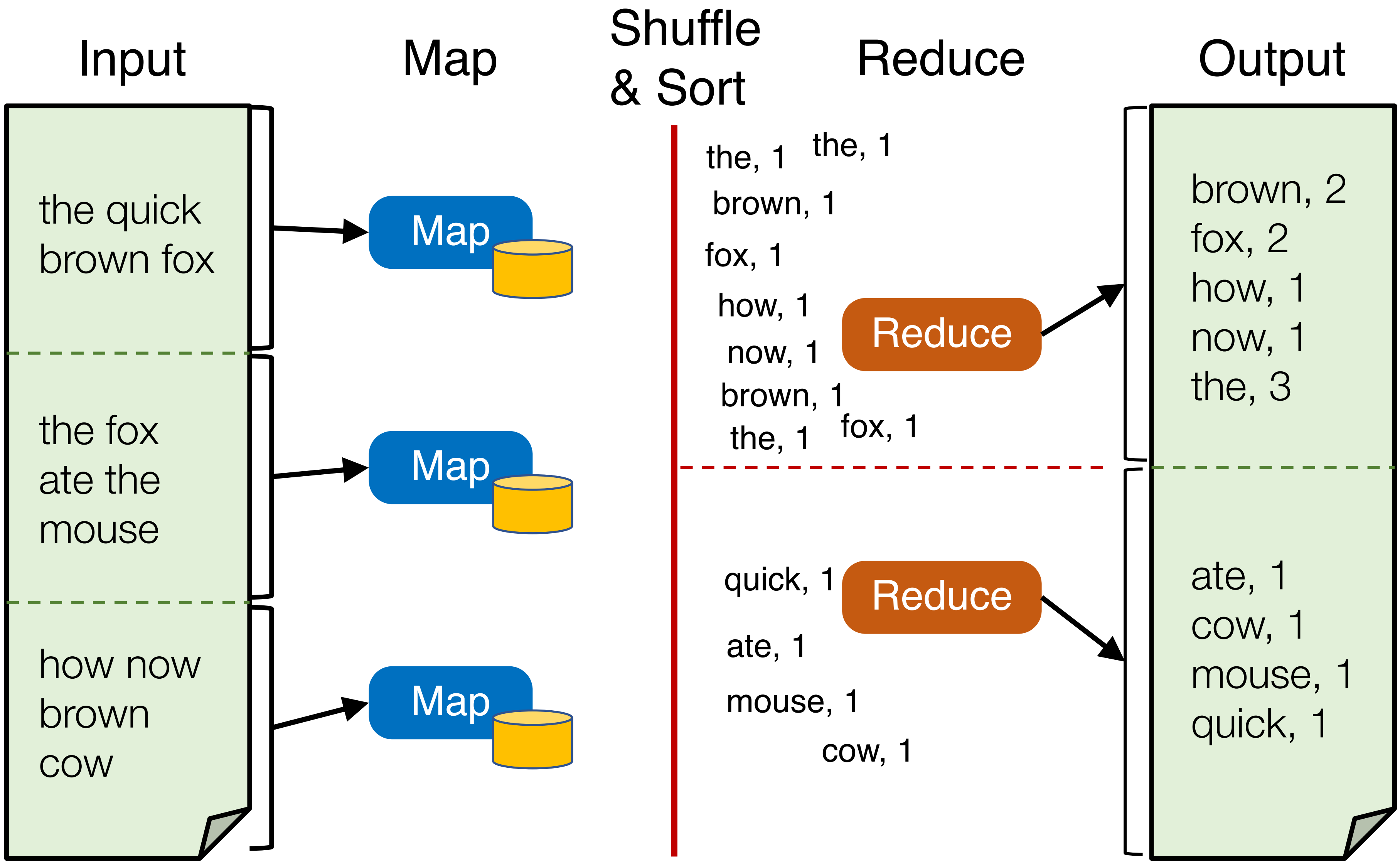
Word Count execution



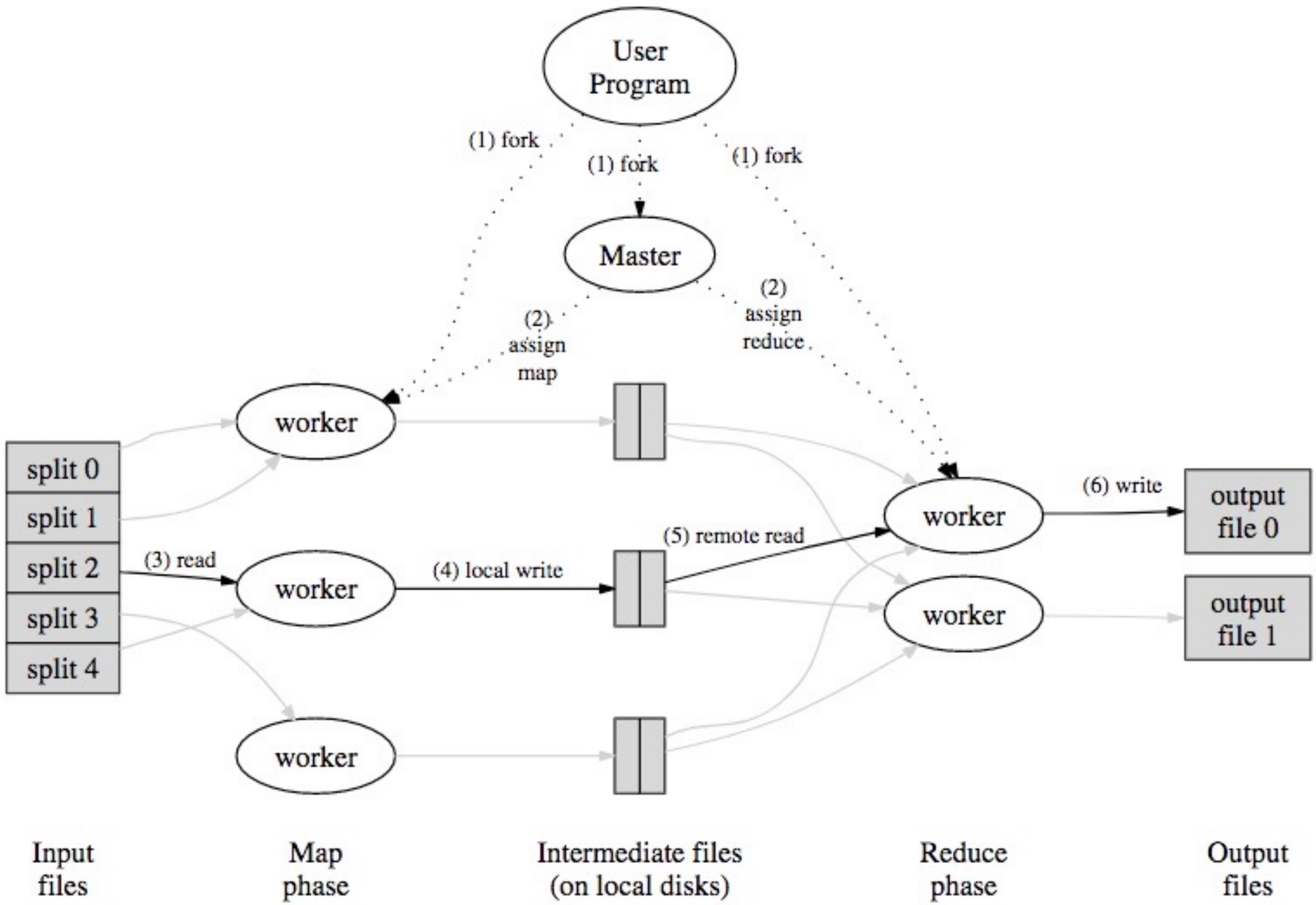
Word Count execution



Word Count execution

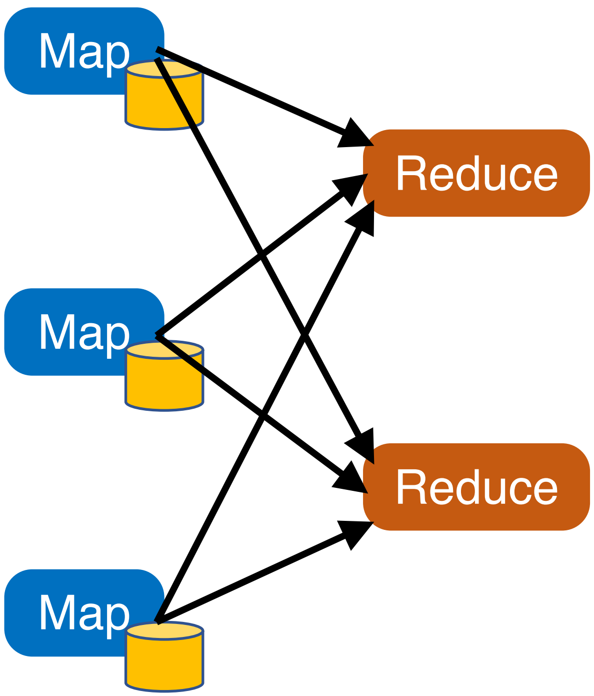


MapReduce data flows



MapReduce processes

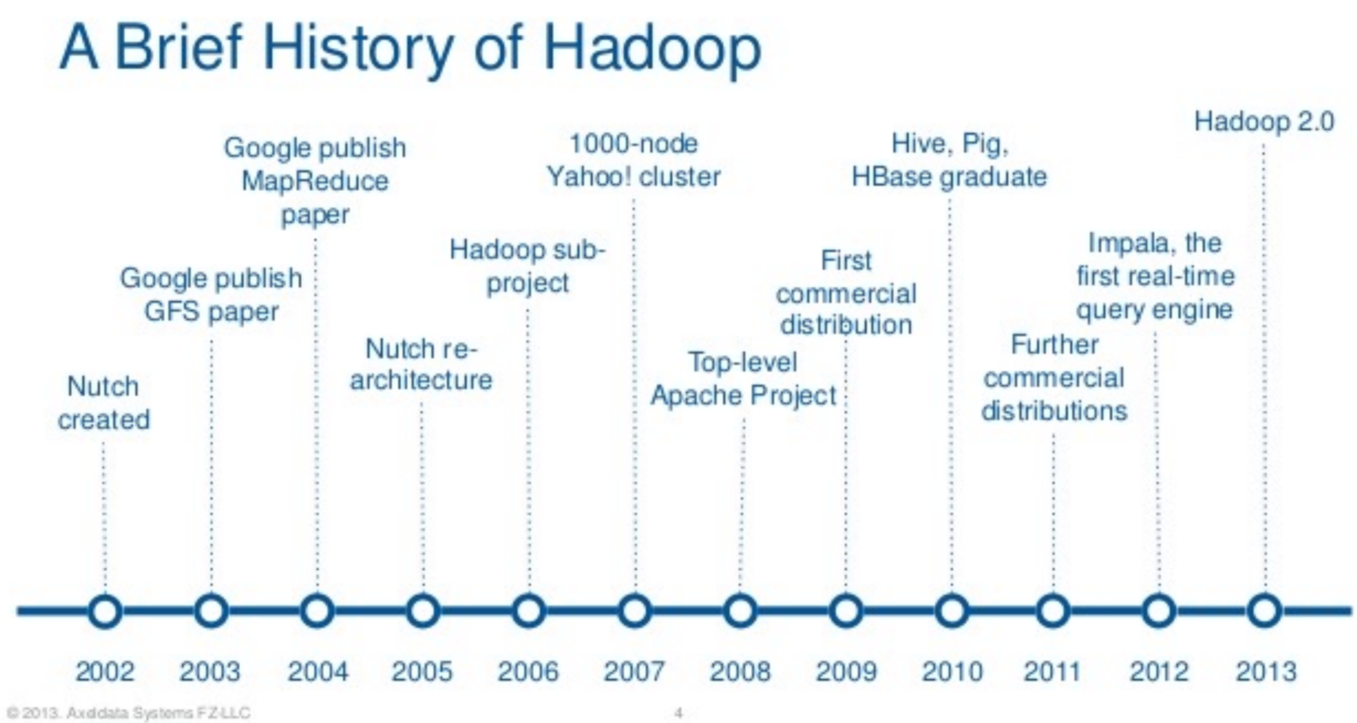
Map | Shuffle & Sort | Reduce



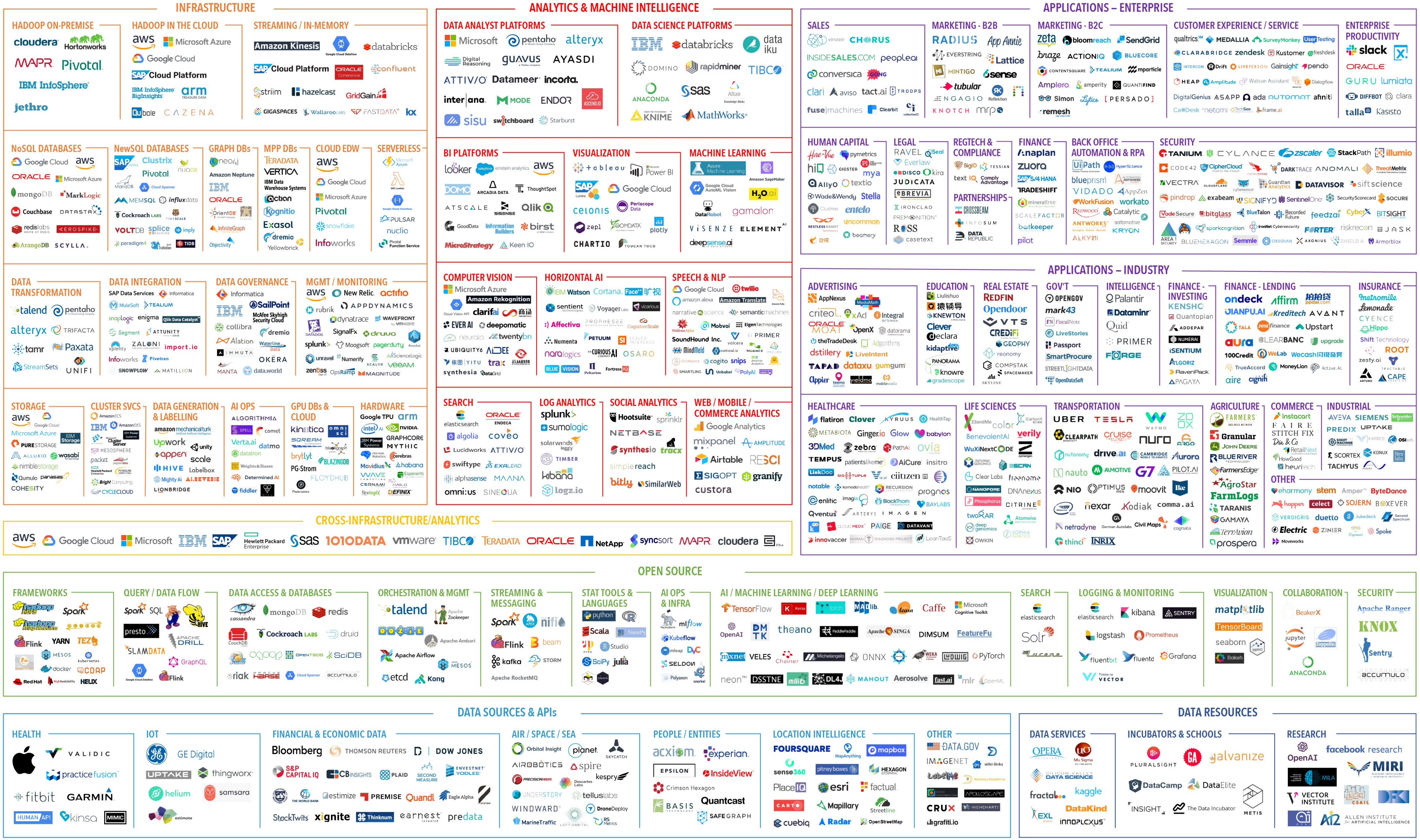
- Map workers write intermediate output to local disk, separated by partitioning. Once completed, tell master node
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - "All-to-all" shuffle b/w mappers and reducers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



DATA & AI LANDSCAPE 2019

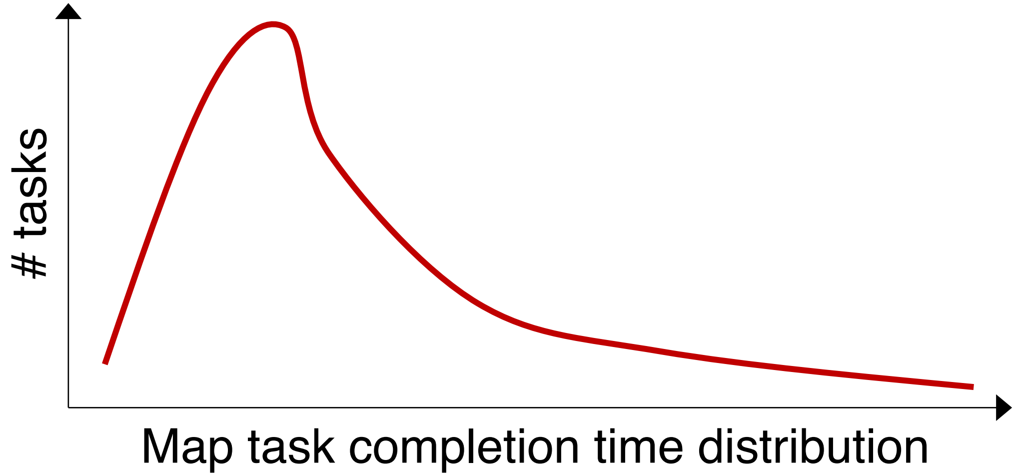


July 16, 2019 - FINAL 2019 VERSION

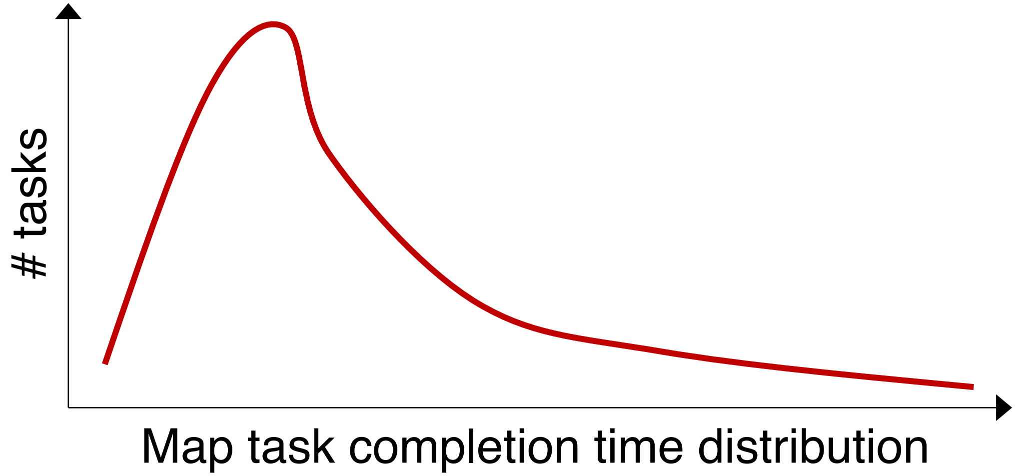
© Matt Turck (@mattturck), Lisa Xu (@lisaxu92), & FirstMark (@firstmarkcap) mattturck.com/data2019

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

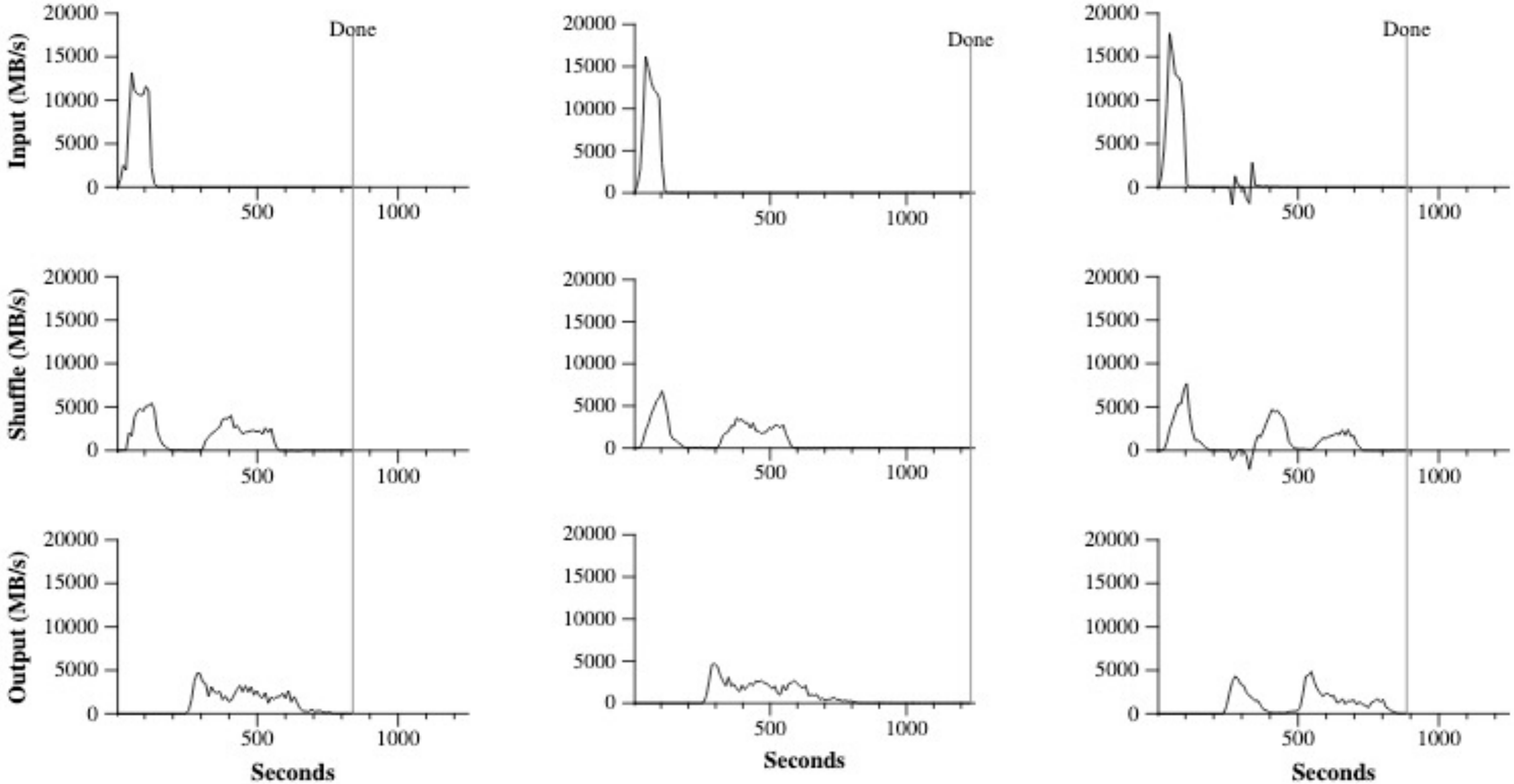
- How does MapReduce jobs get good load balance across worker machines?

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



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

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

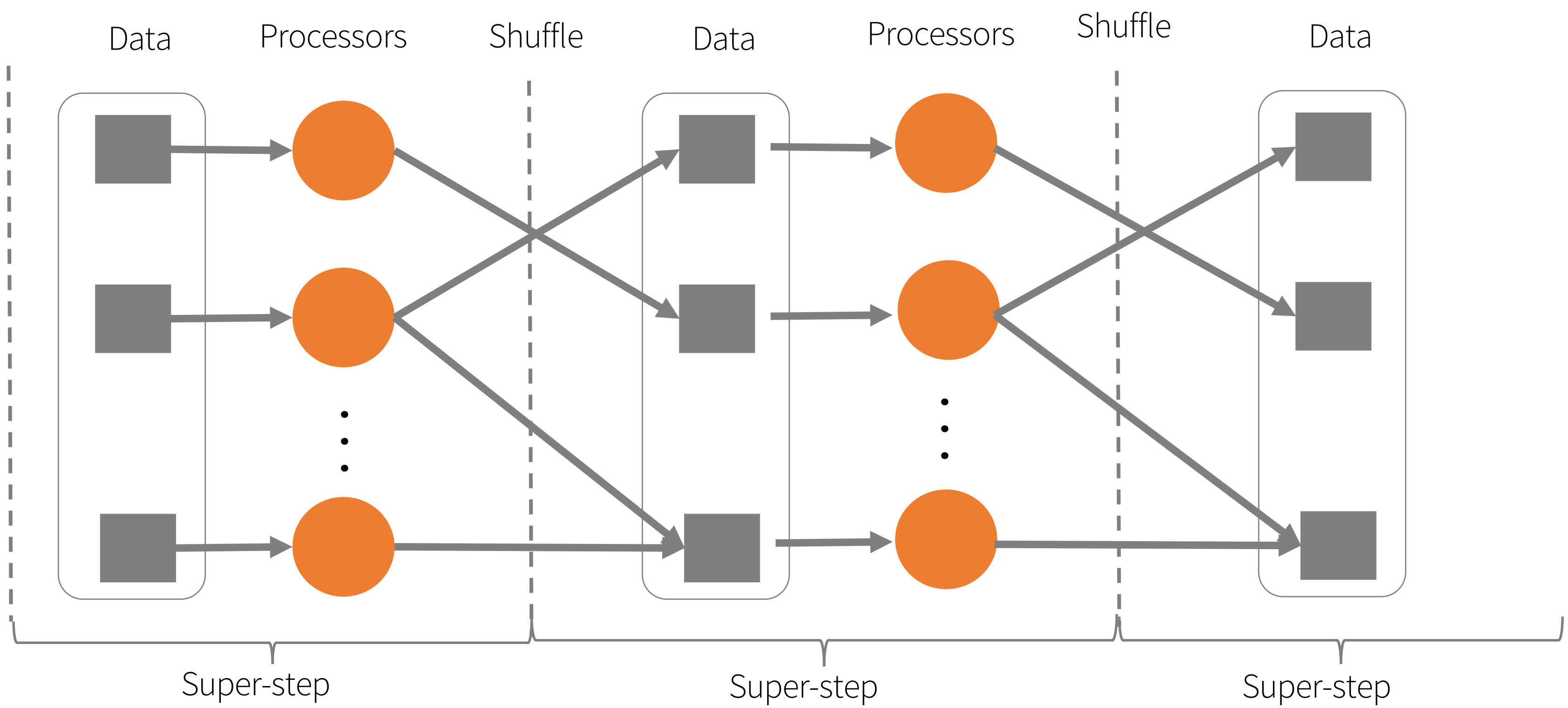
- Google published details in 2003

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

Workload-driven design

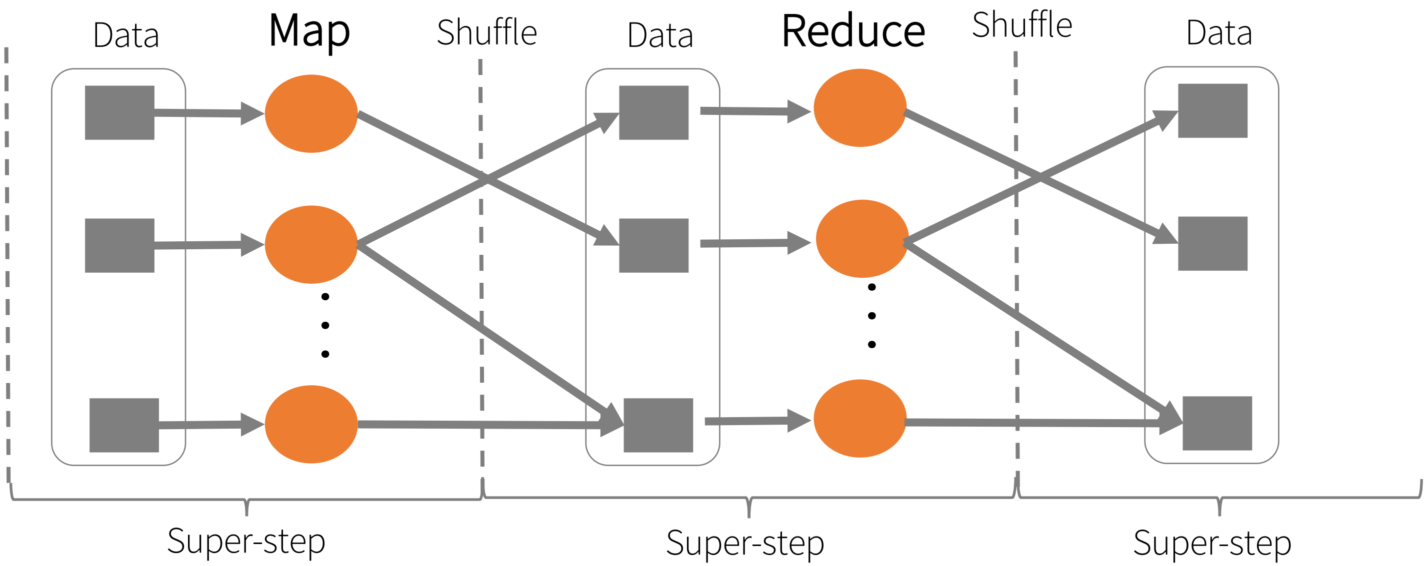
- MapReduce workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

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



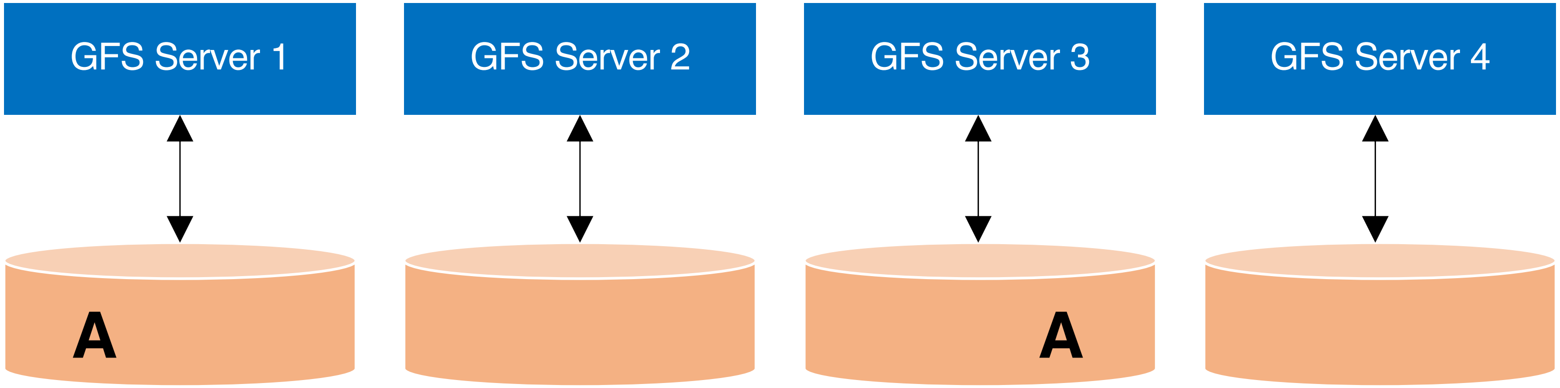
- Read entire dataset, do computation over it
  - Batch processing

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

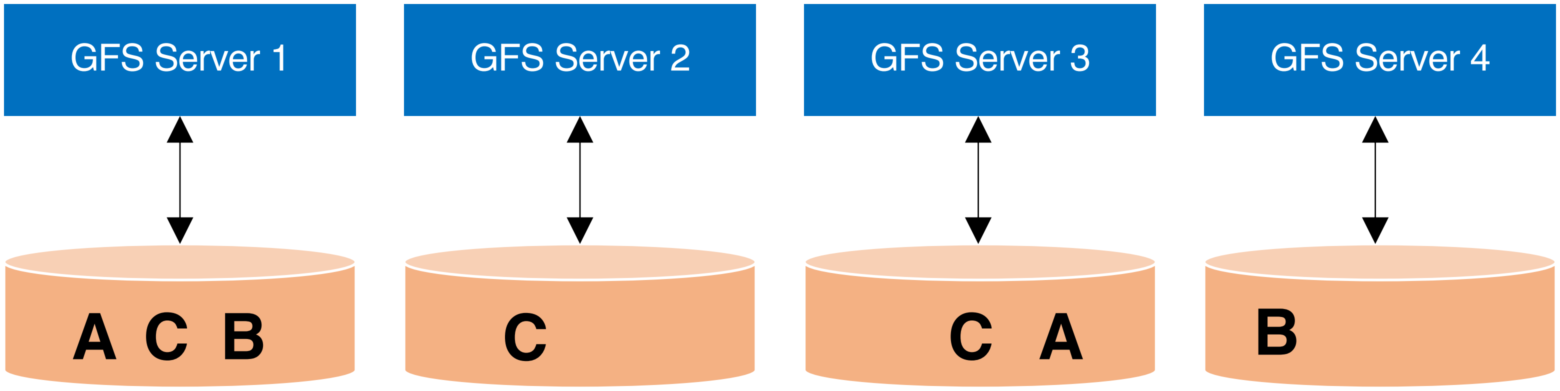
Workload-driven design

- 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

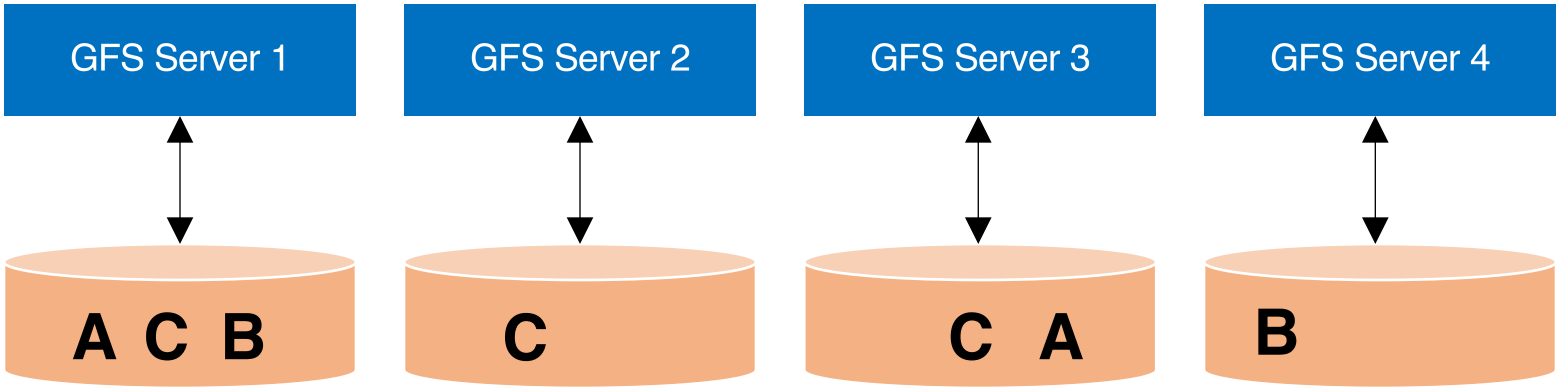
Replication



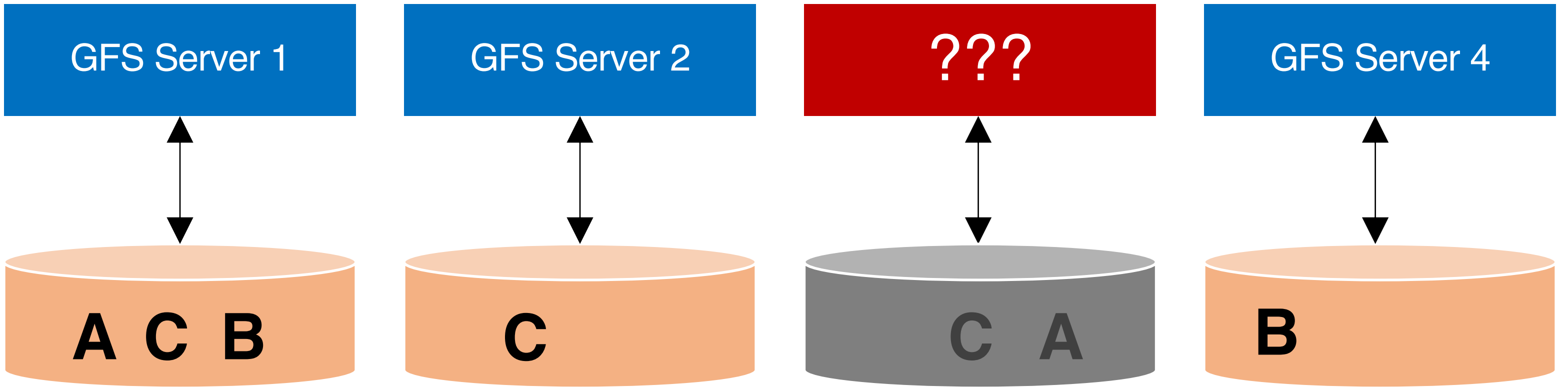
Replication



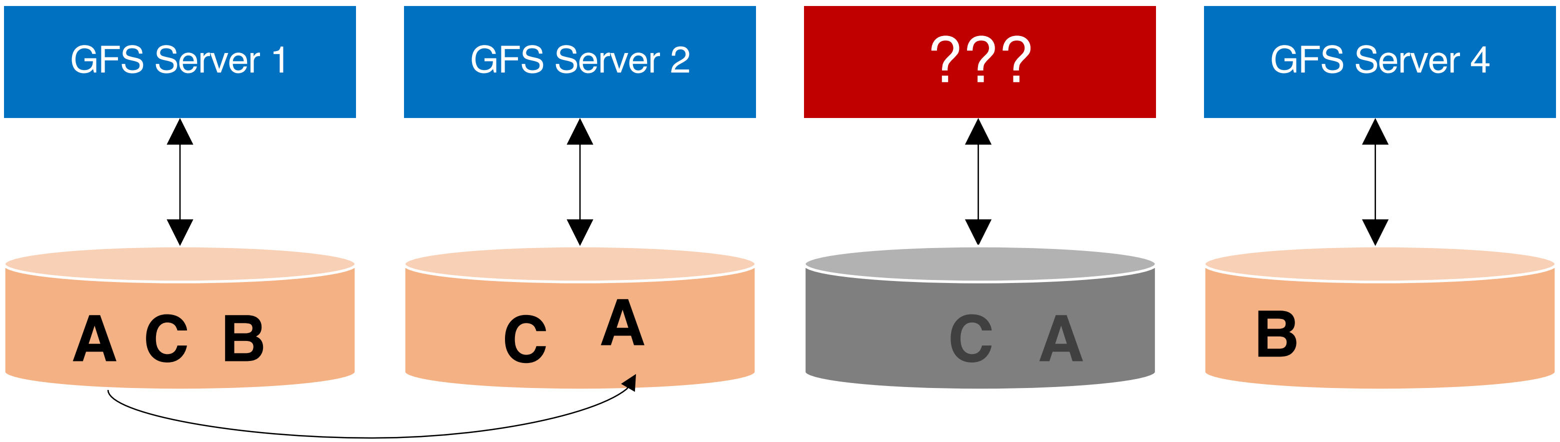
Resilience against failures



Resilience against failures

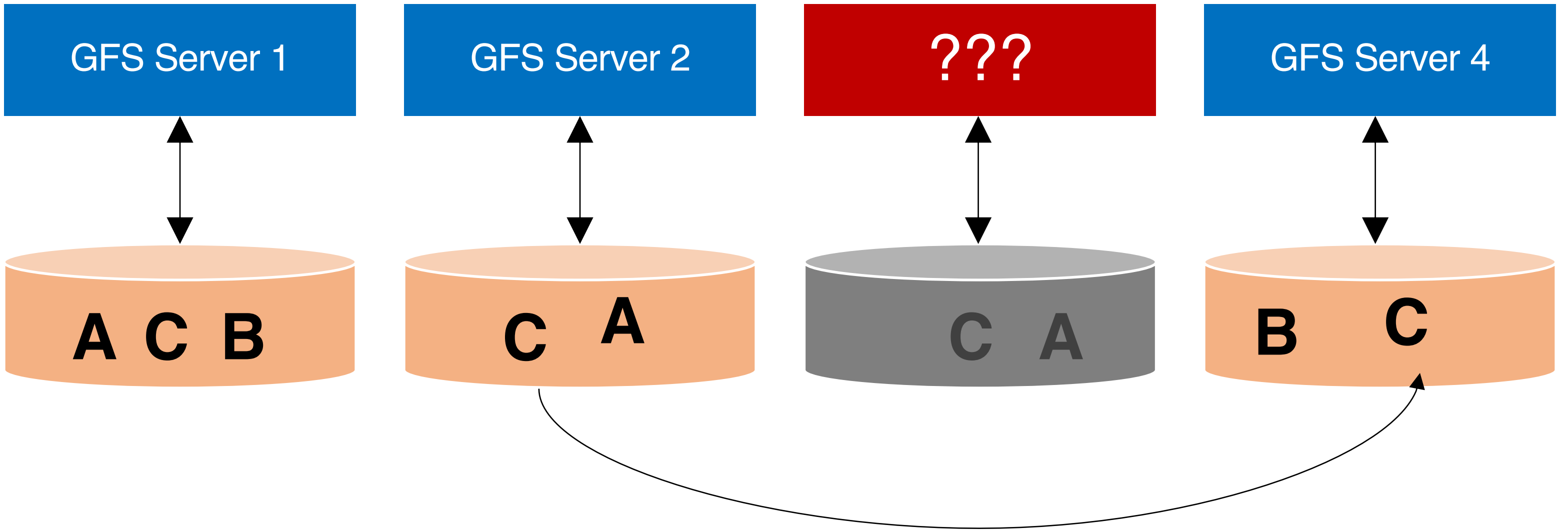


Data recovery



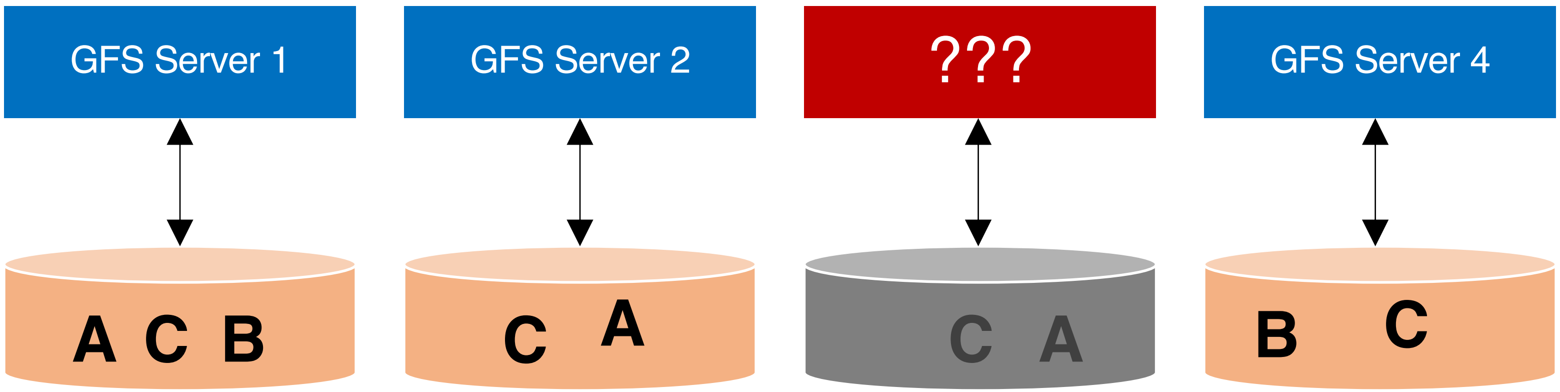
Replicating A to maintain a replication factor of 2

Data recovery



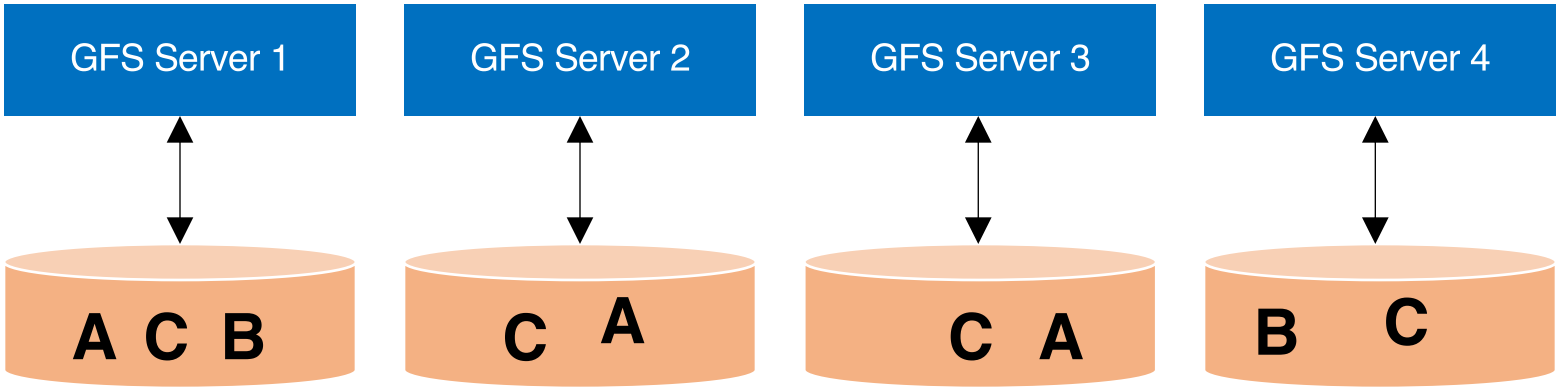
Replicating C to maintain a replication factor of 3

Data recovery



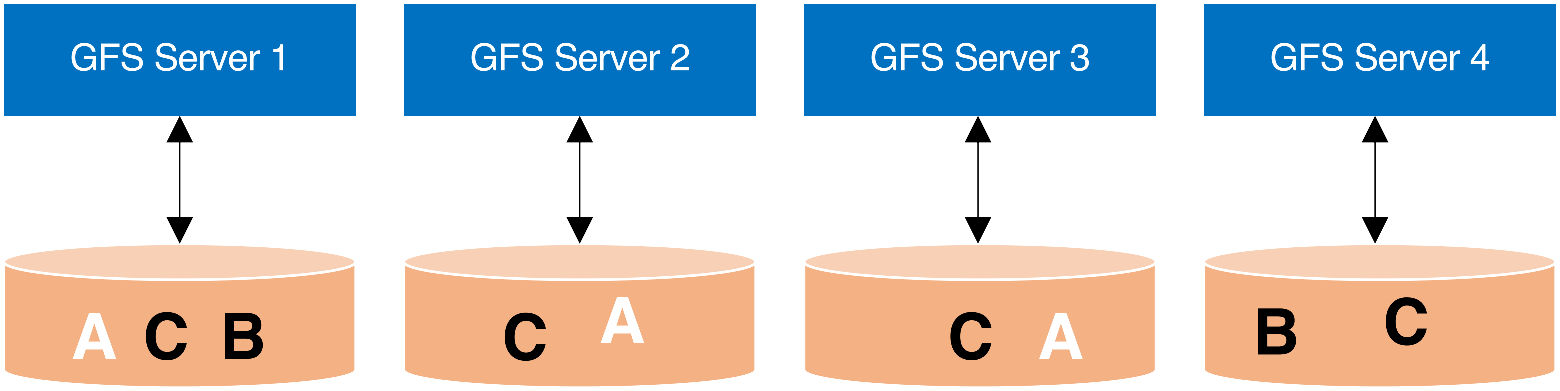
Machine may be dead forever, or it may come back

Data recovery

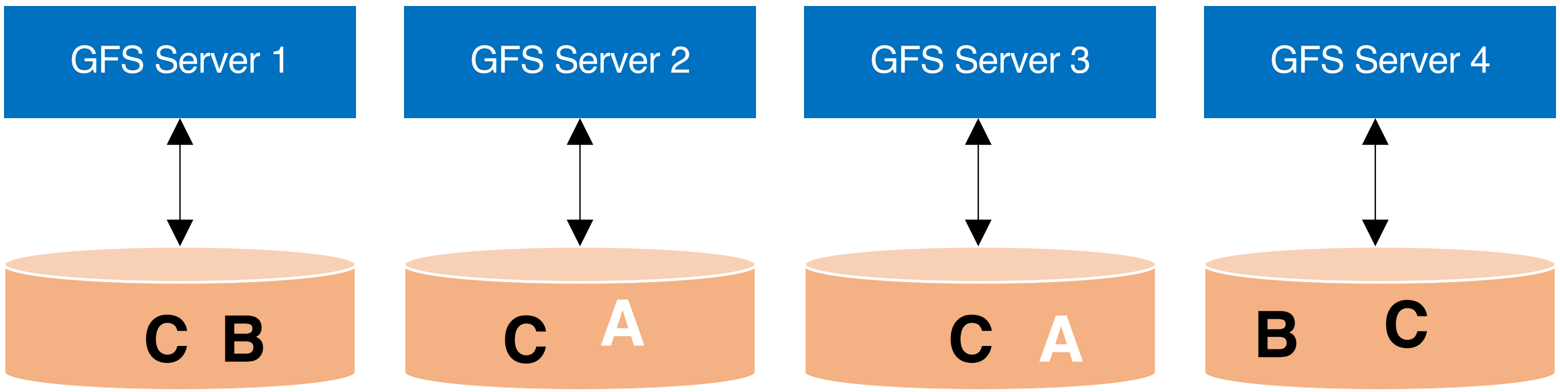


Machine may be dead forever, or it may come back

Data recovery



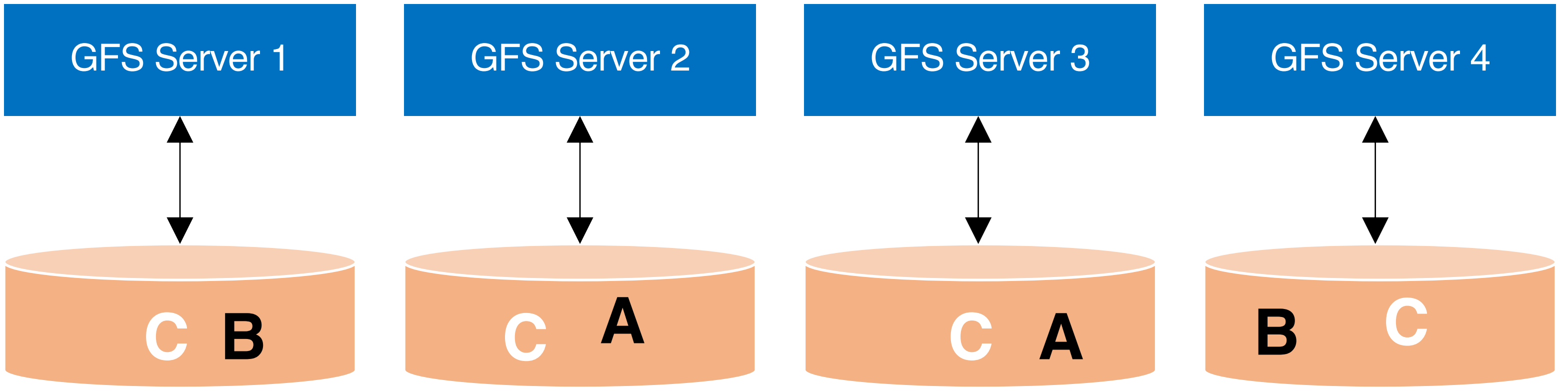
Data recovery



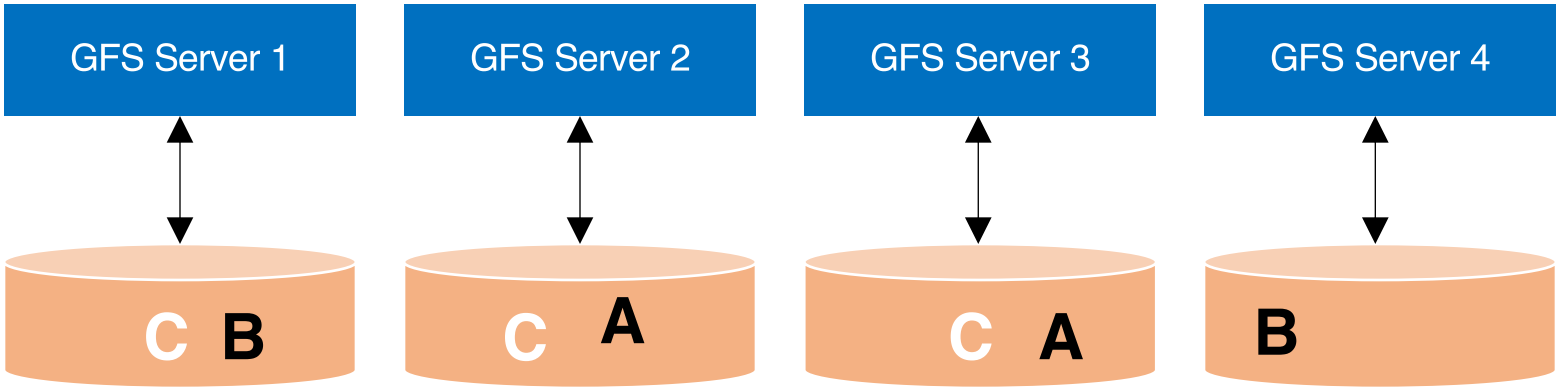
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data recovery



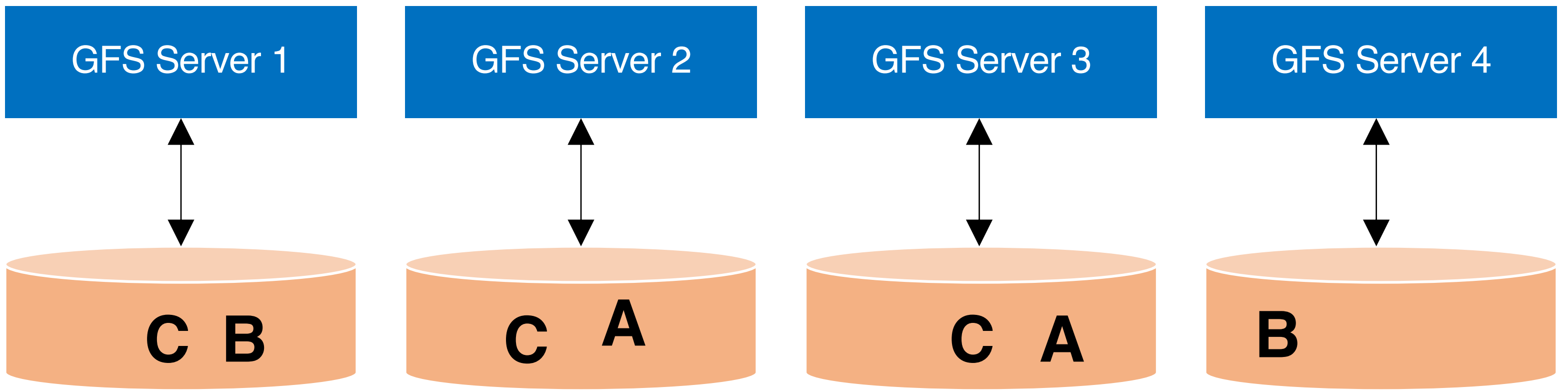
Data recovery



Data Rebalancing

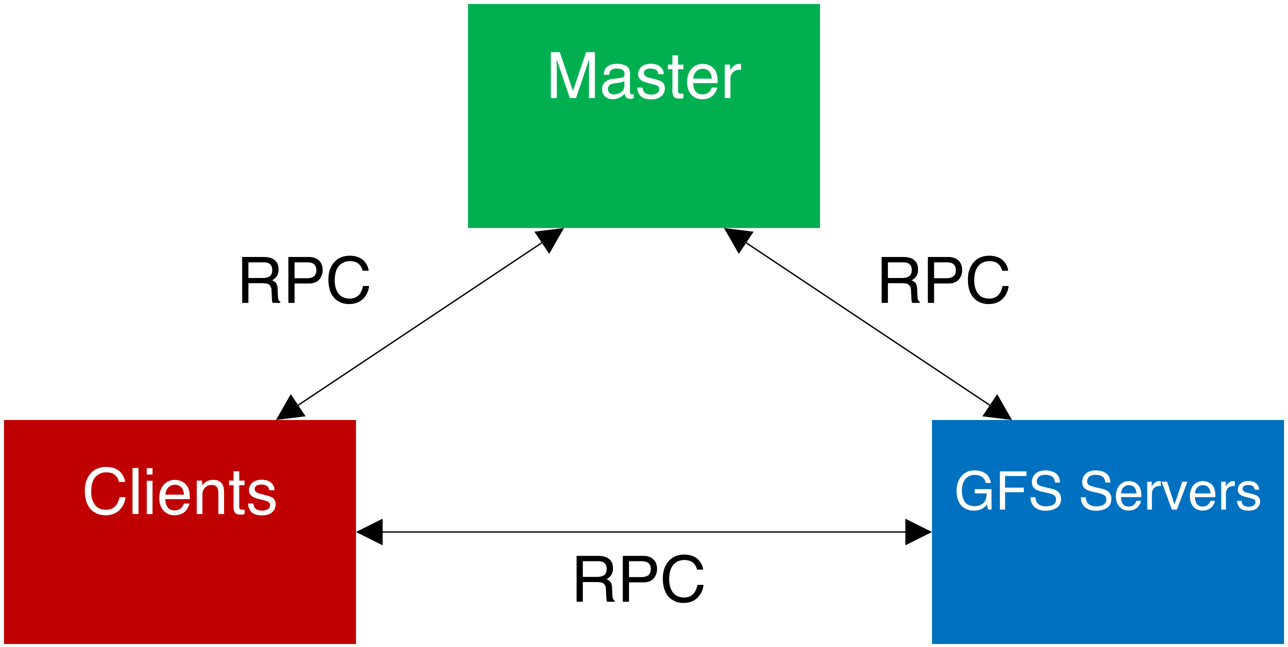
Deleting one C to maintain a replication factor of 3

Data recovery

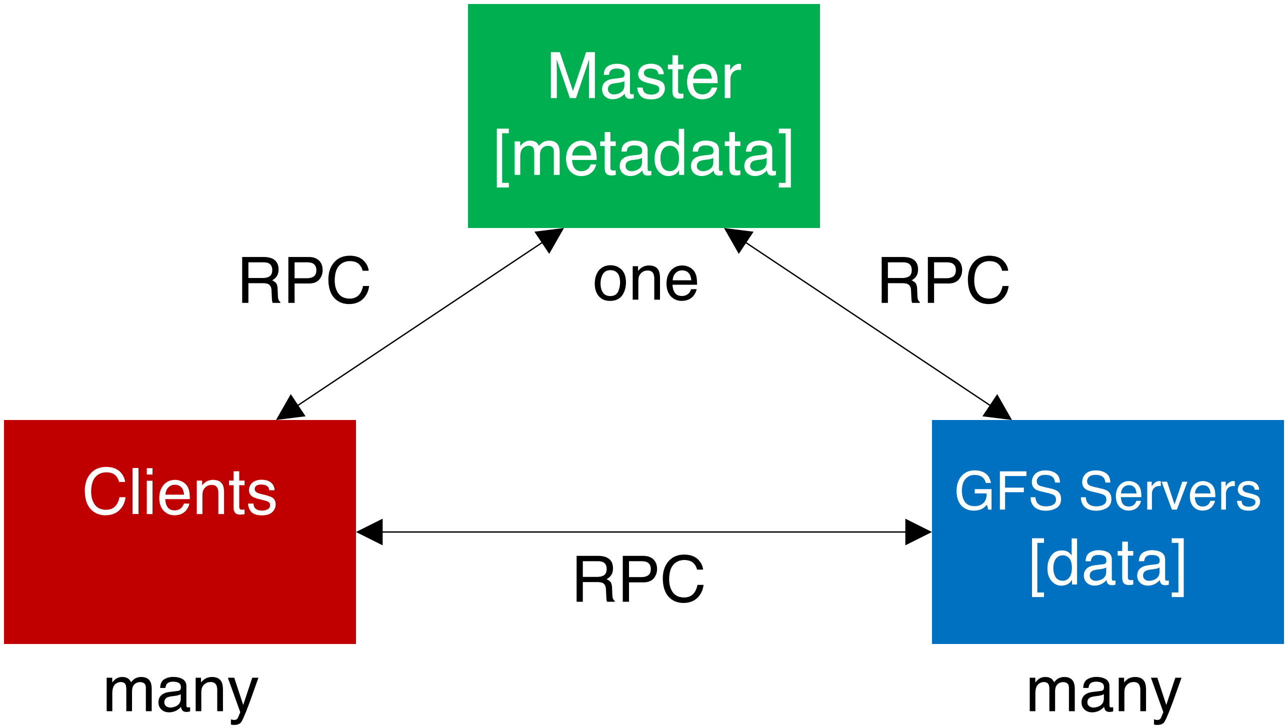


Question: how to maintain a global view of all data distributed across machines?

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

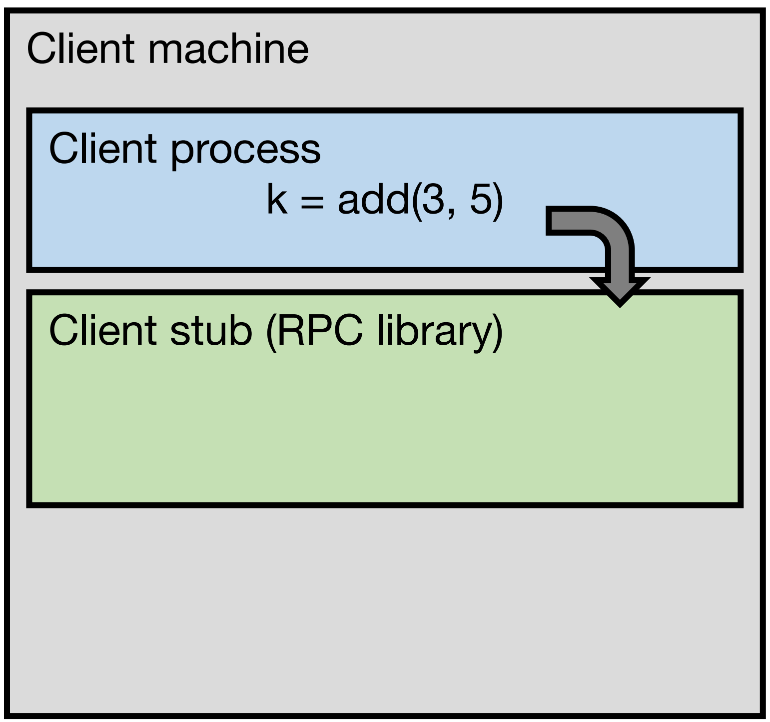
What's the goal of RPC?

- 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

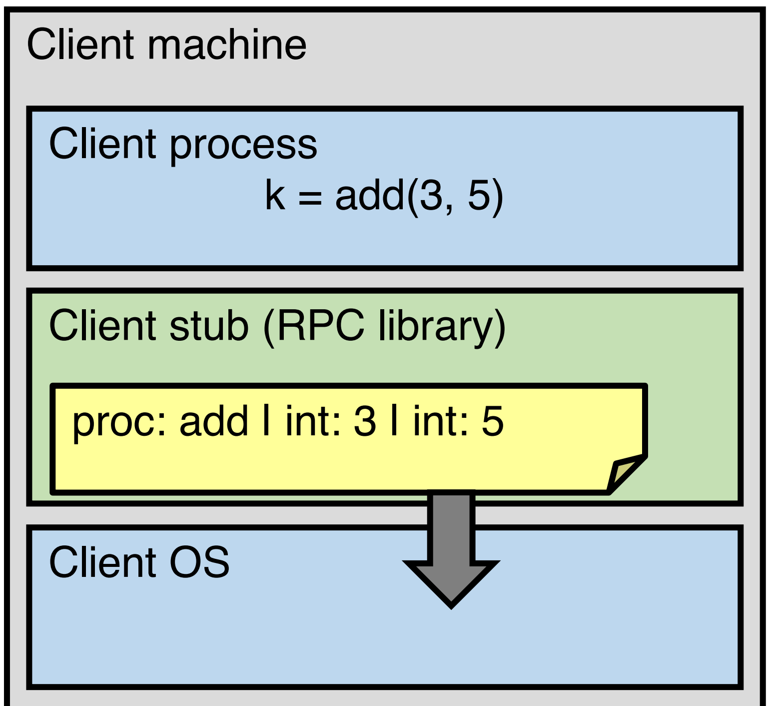
1. Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

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

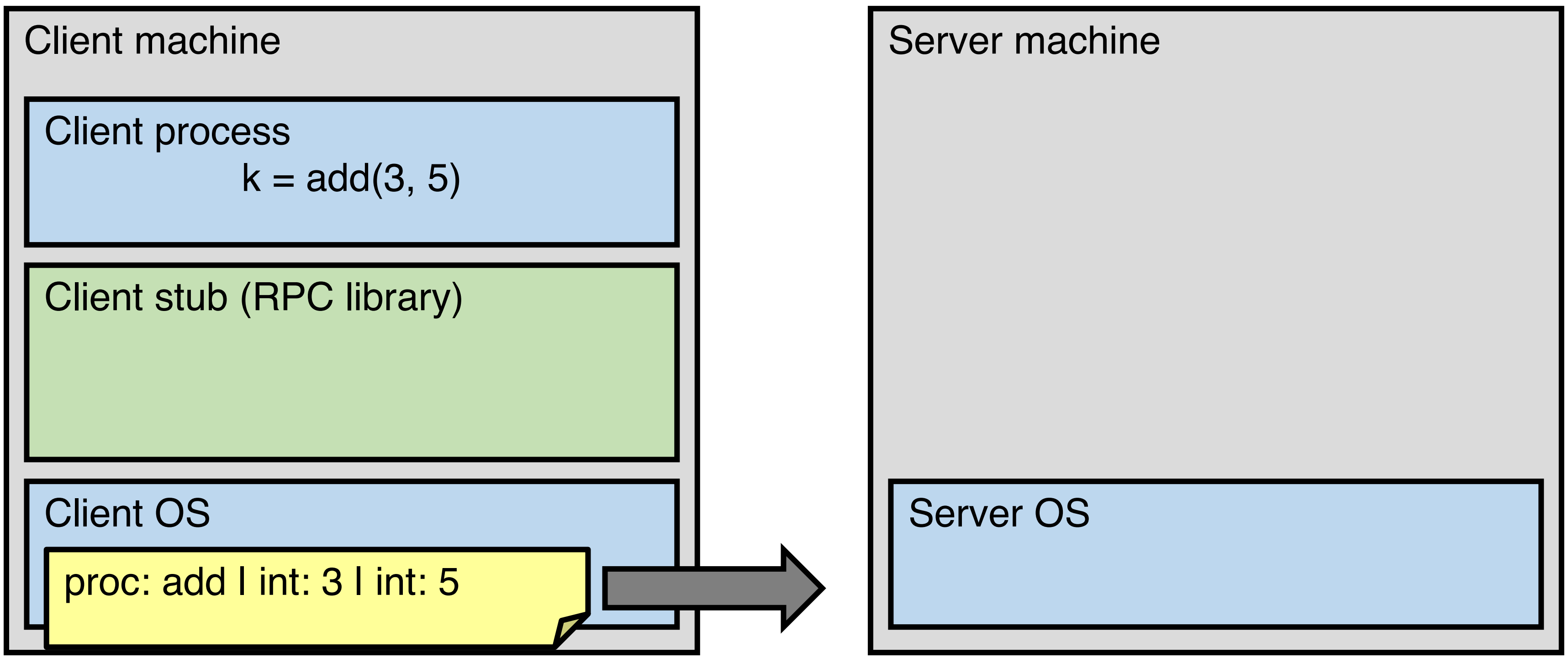
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

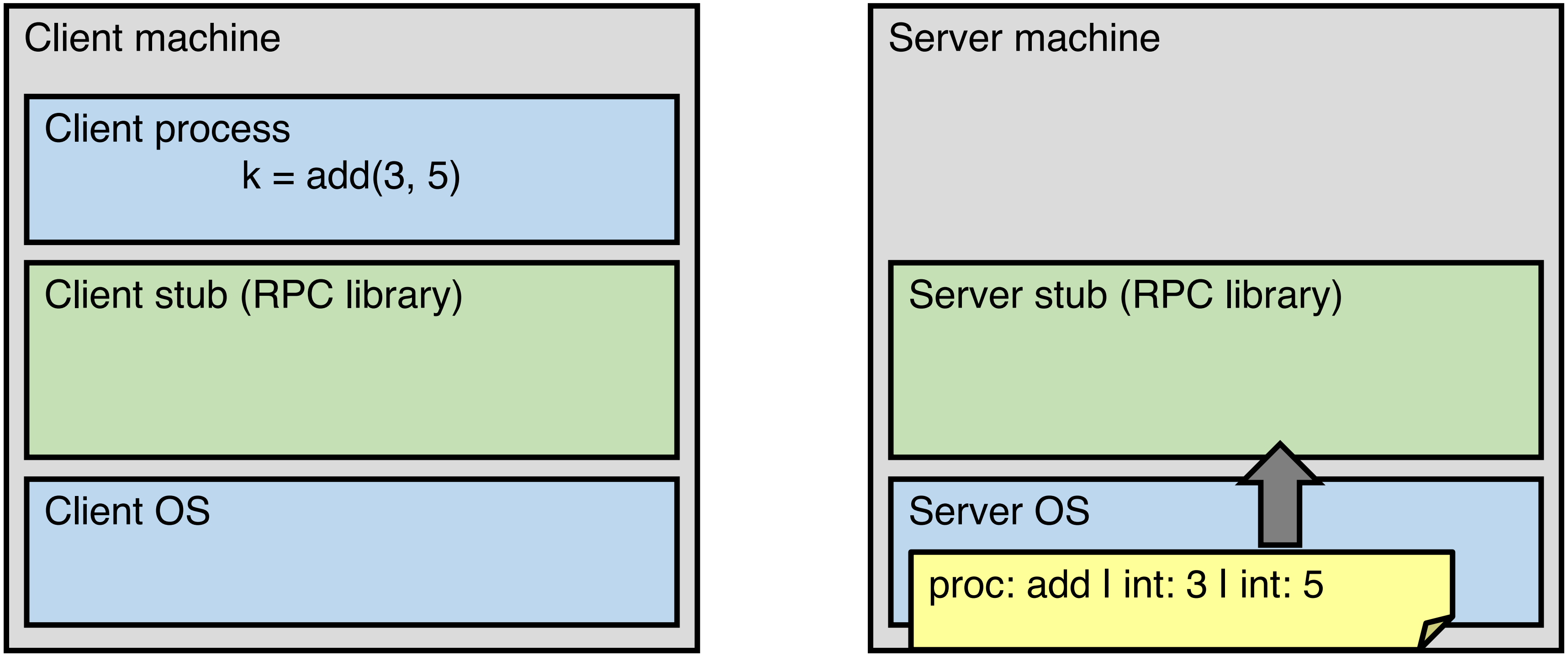
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

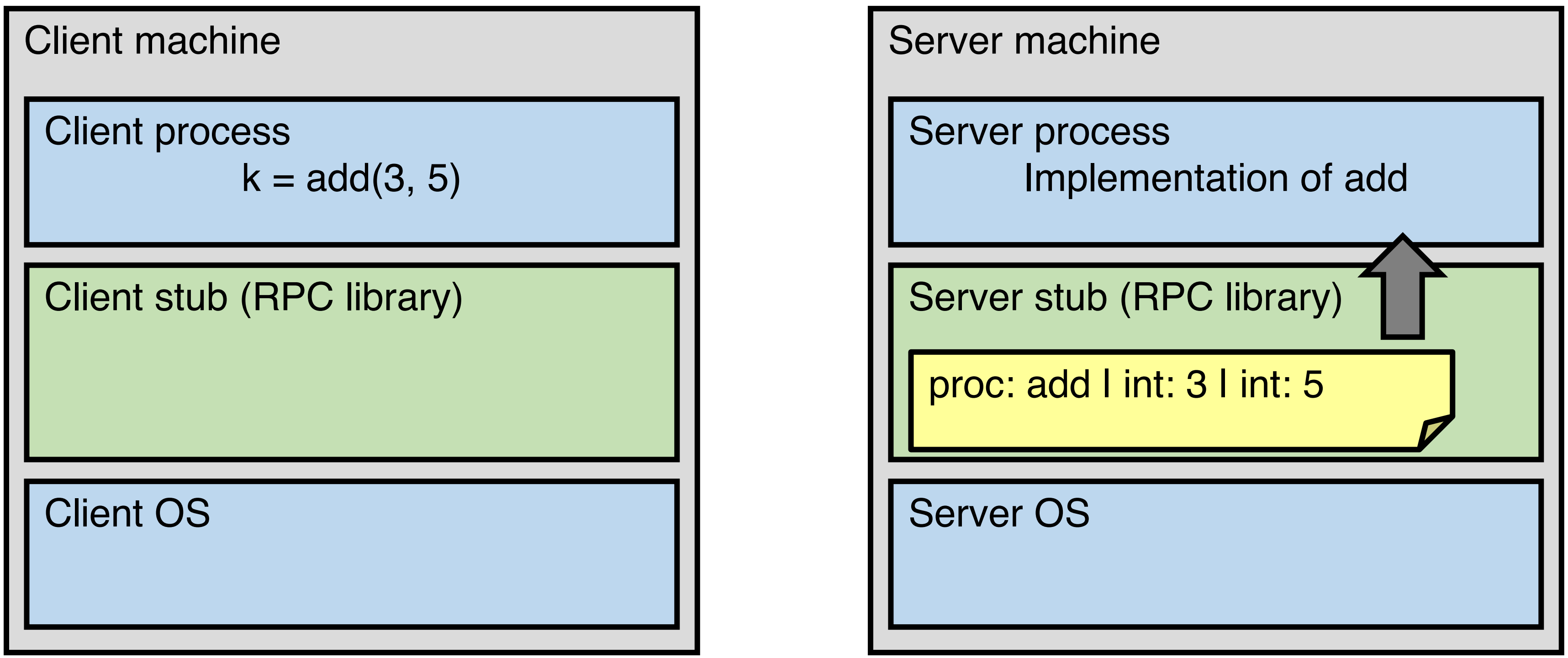
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

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

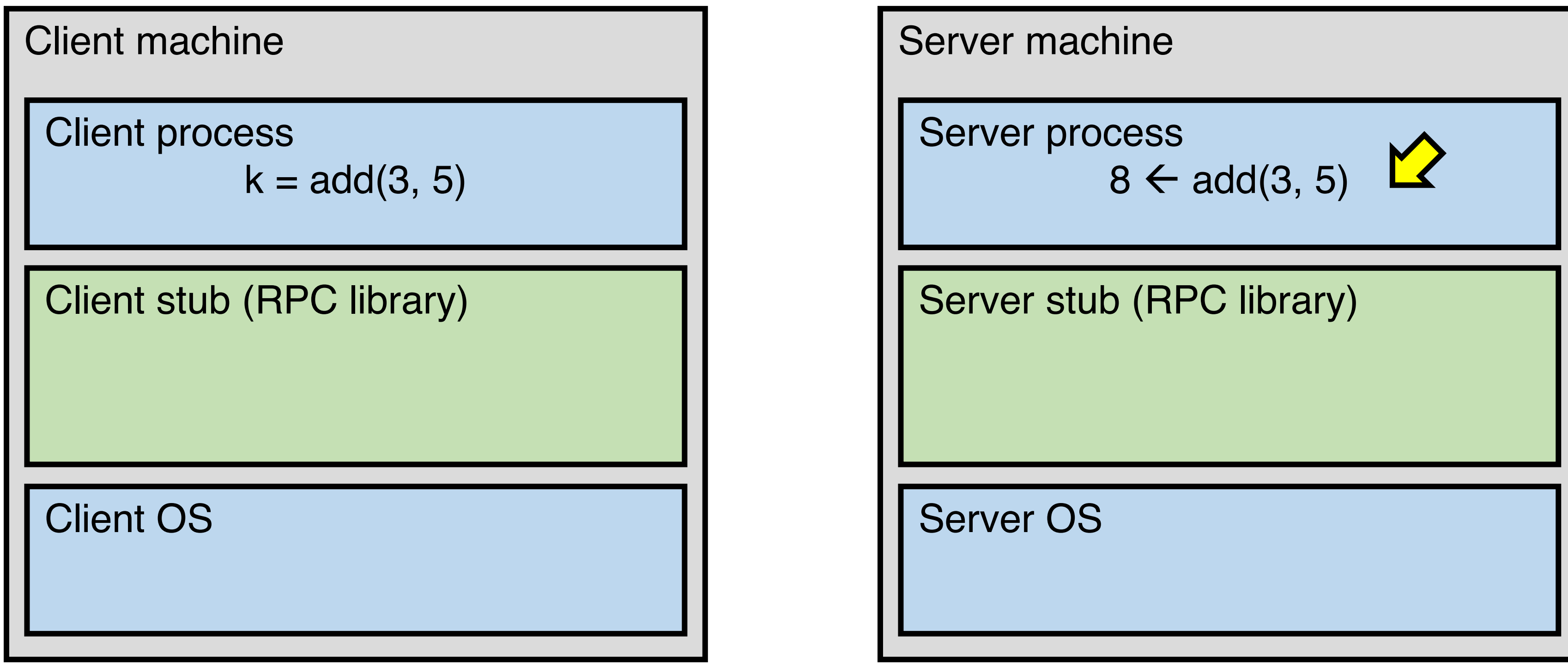
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

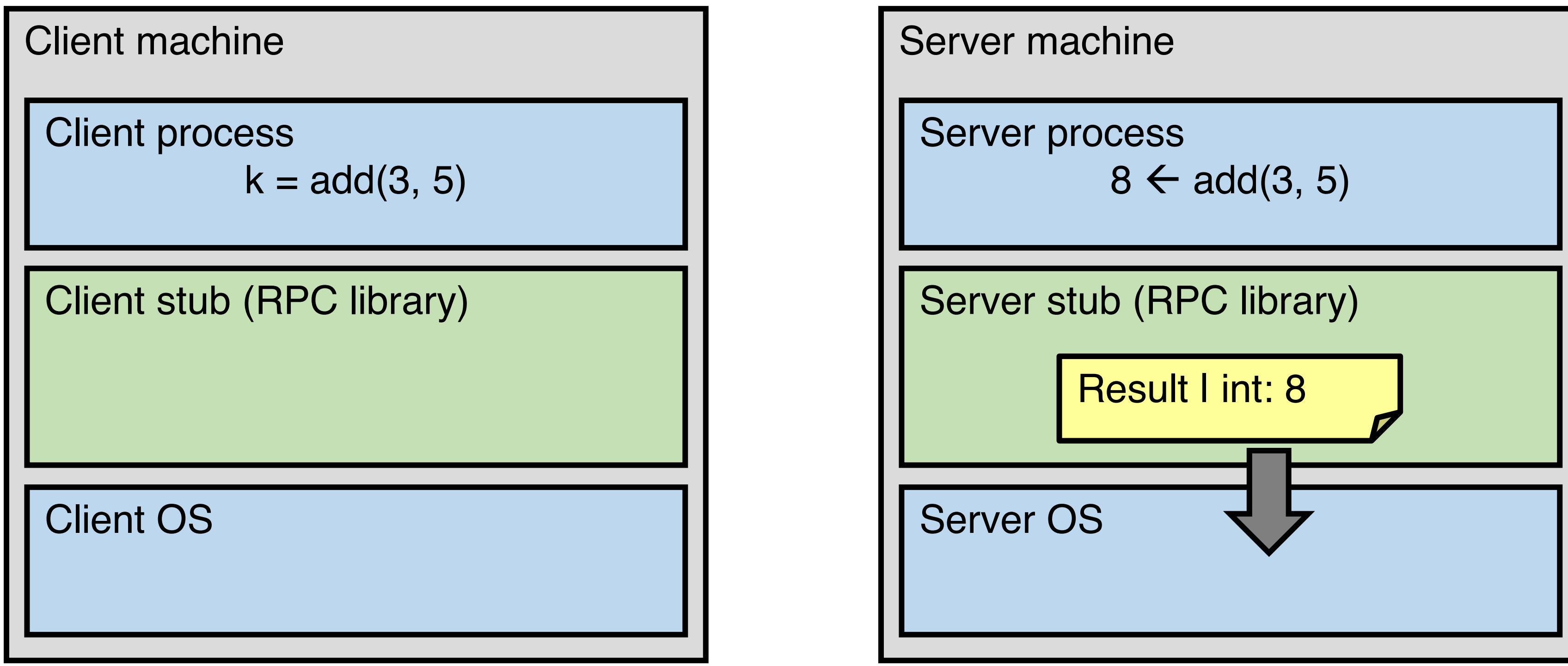
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

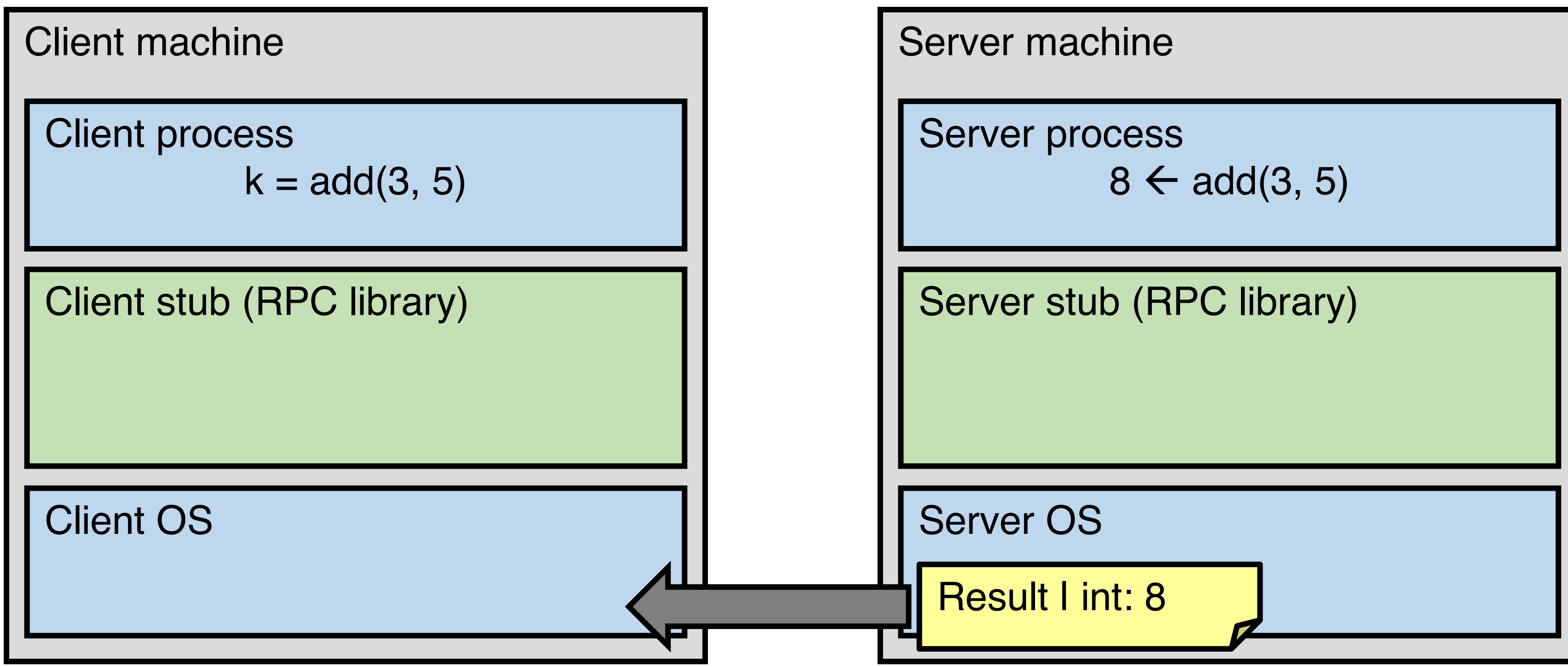
7. Server stub marshals the return value, sends message



A day in the life of an RPC

7. Server stub marshals the return value, sends message

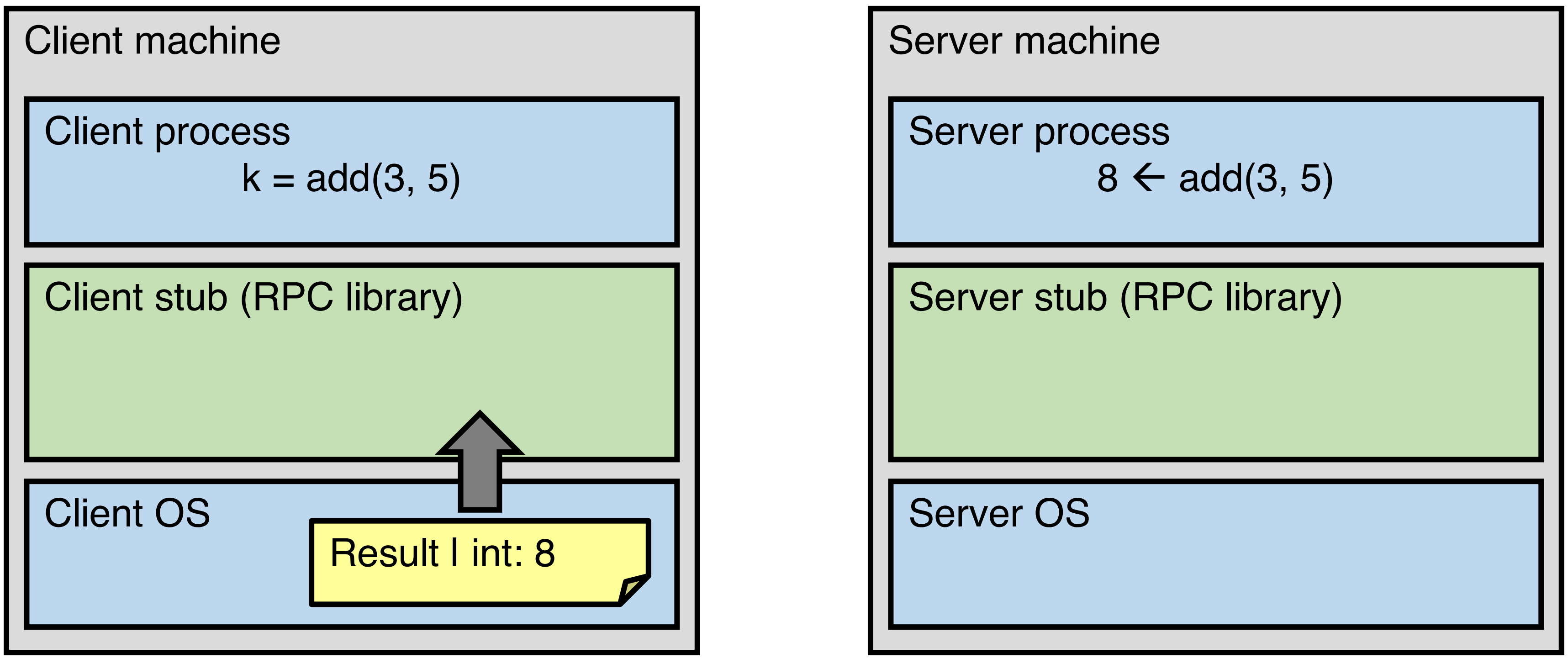
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

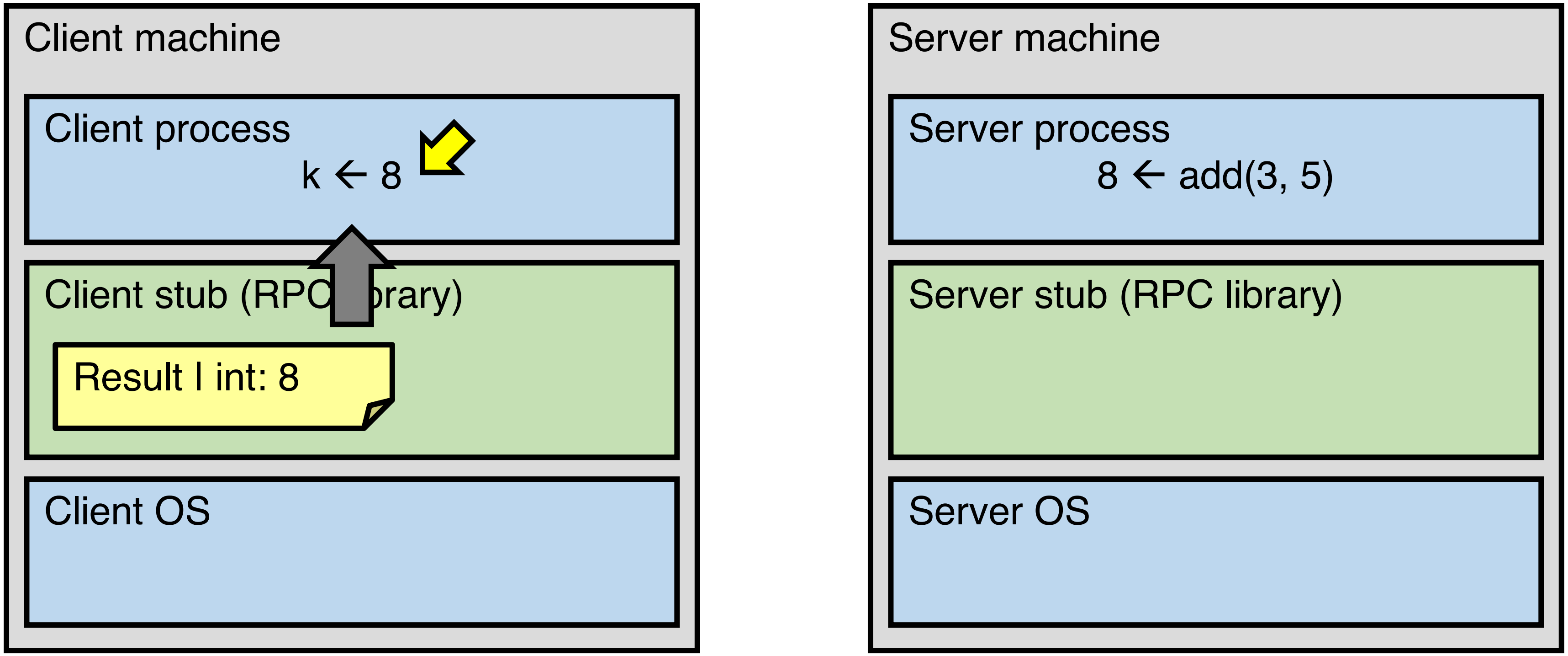
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

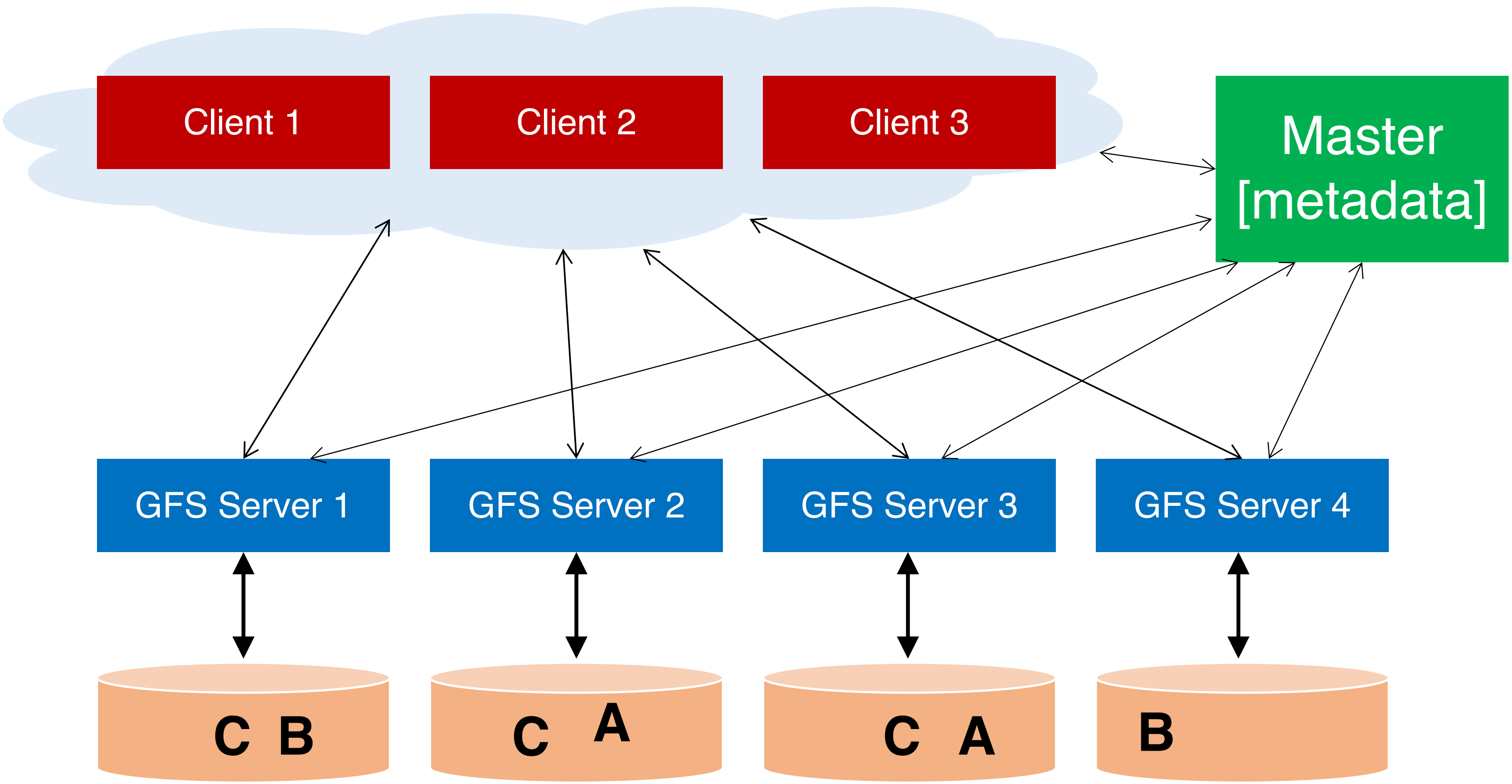
9. Client OS receives the reply and passes up to stub

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

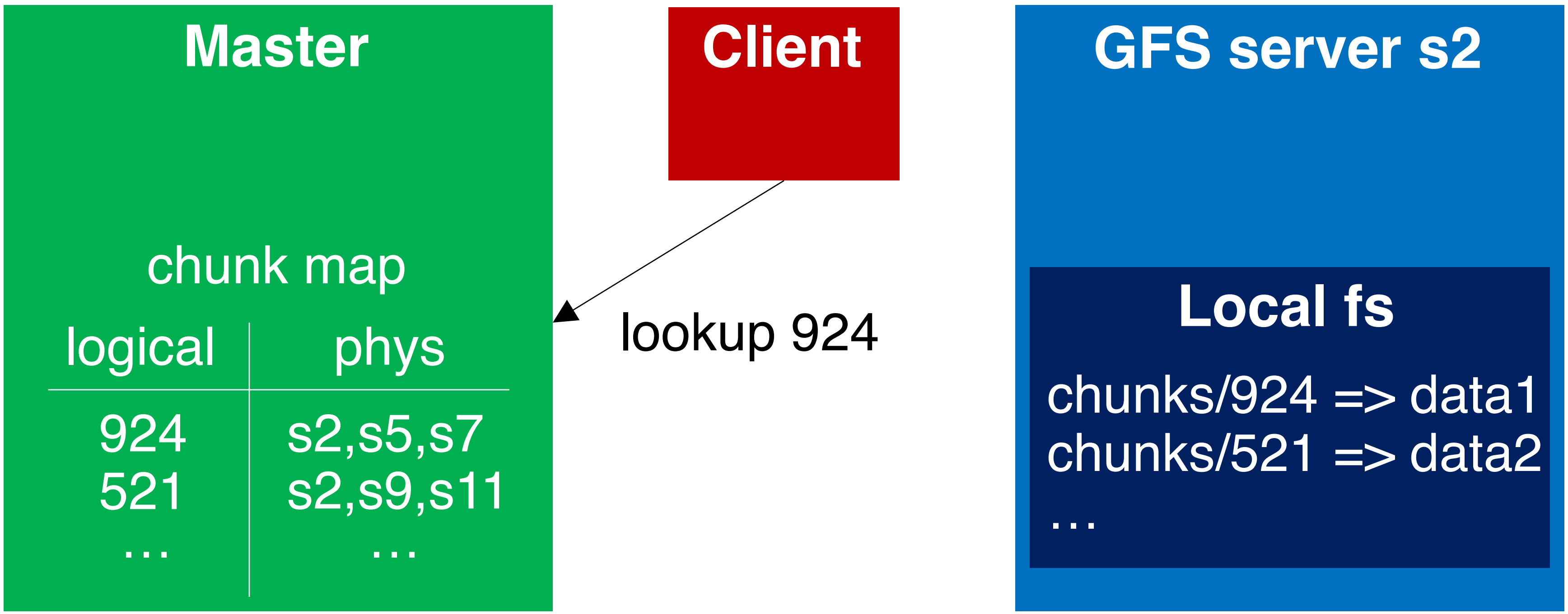
logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS server s2

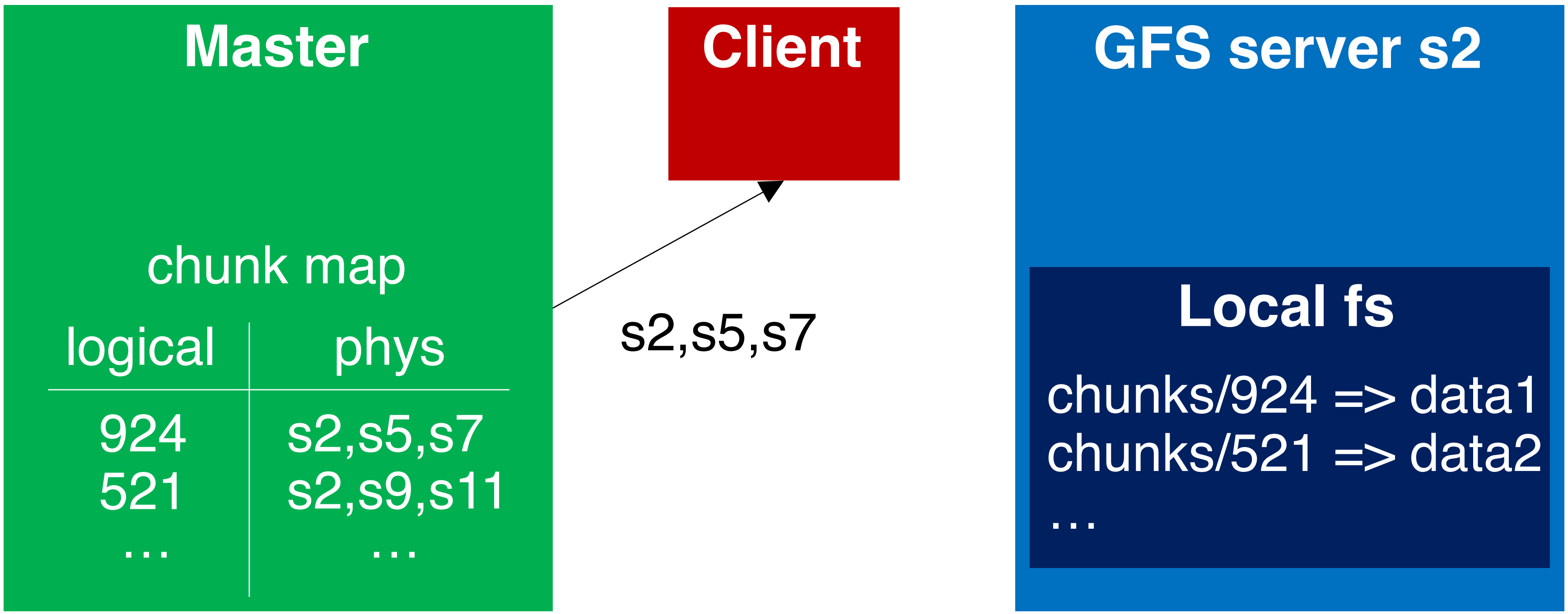
Local fs

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

Client reads a chunk

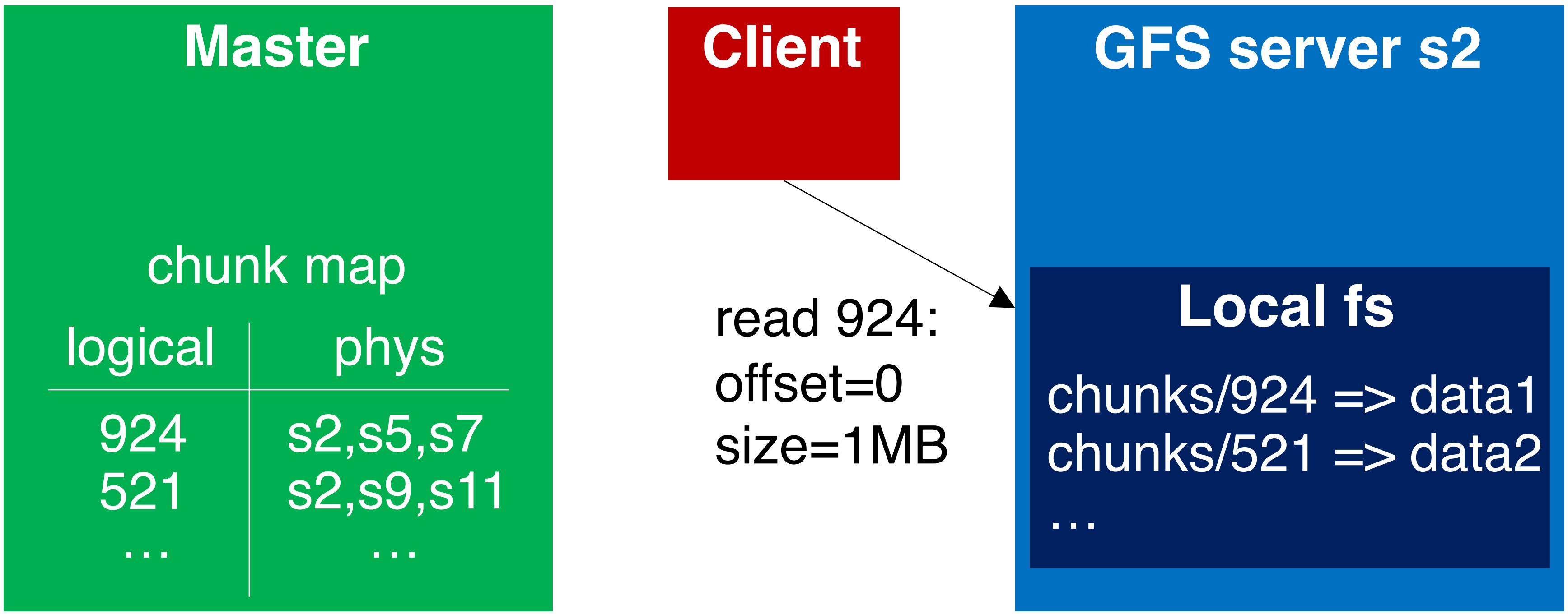


Client reads a chunk

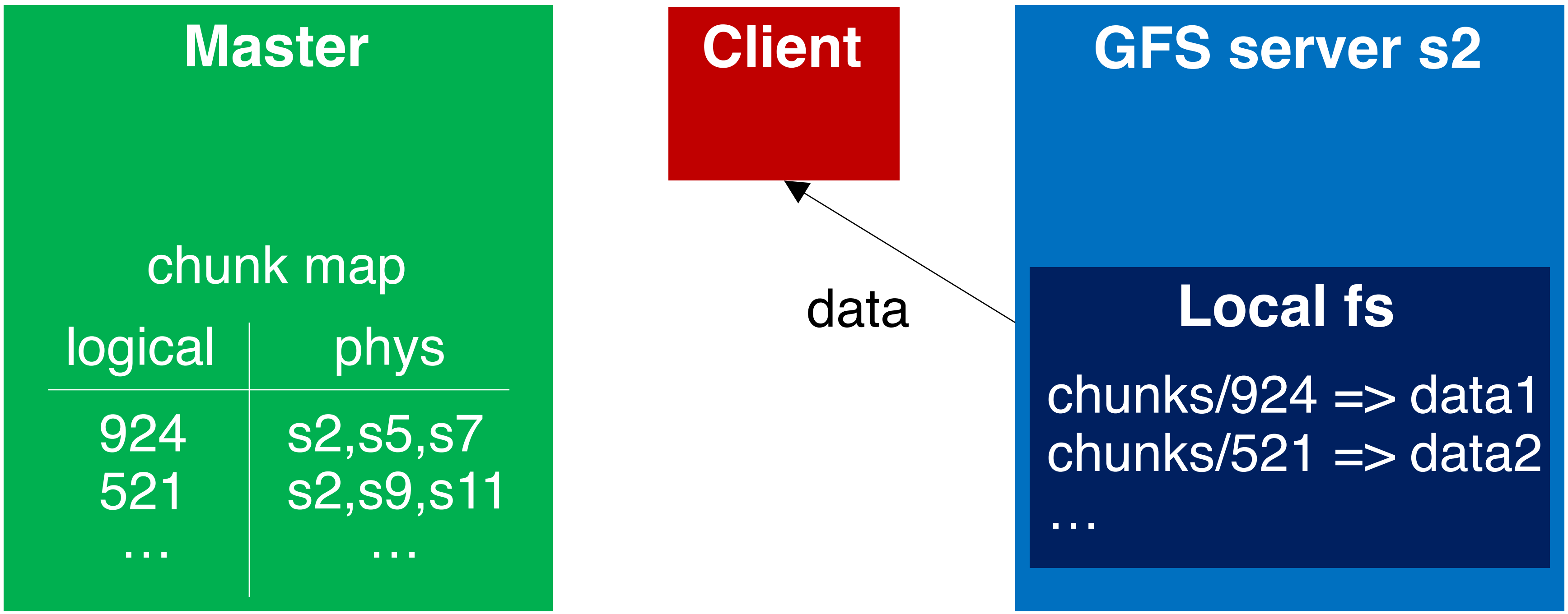


Client reads a chunk

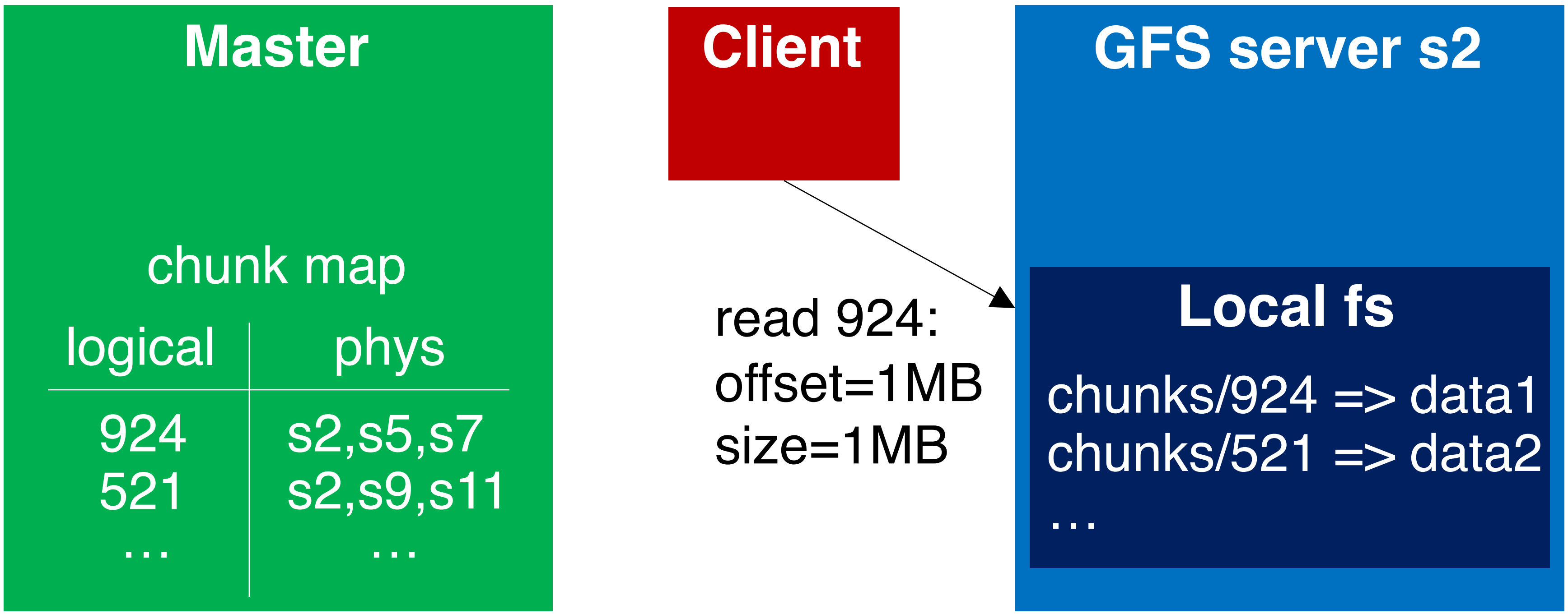
Client reads a chunk



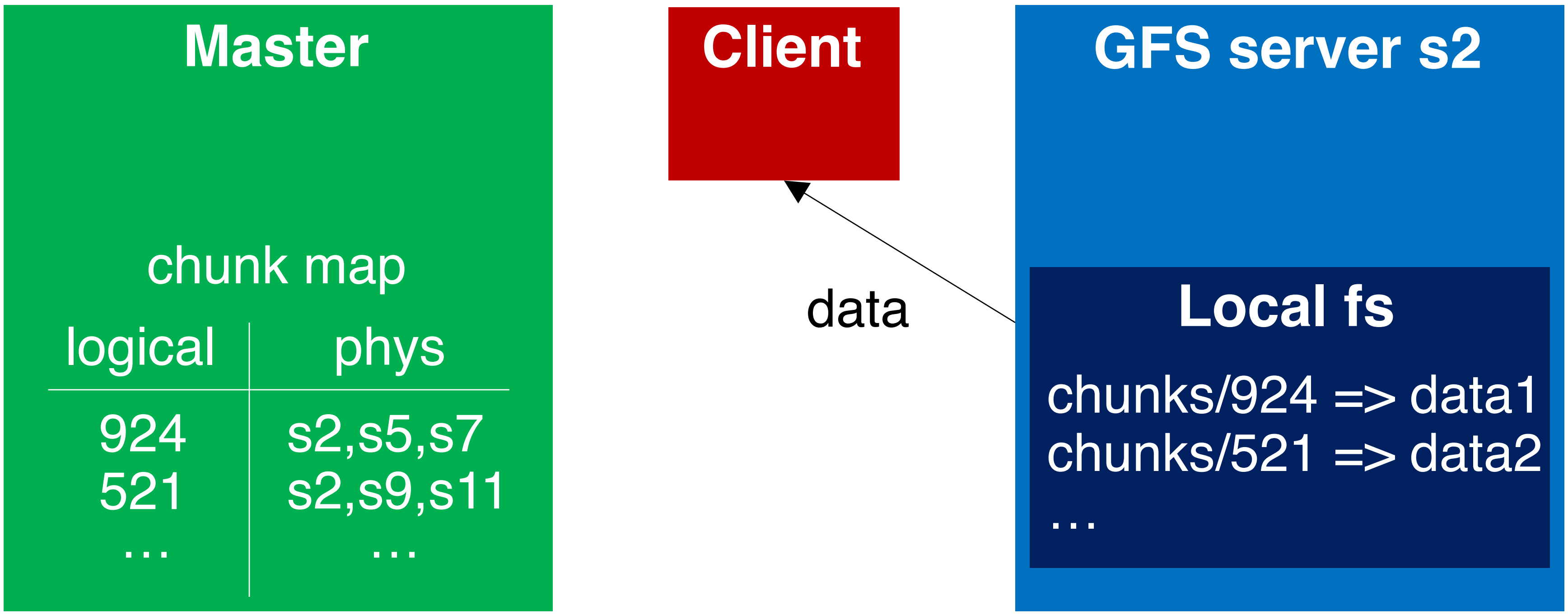
Client reads a chunk



Client reads a chunk



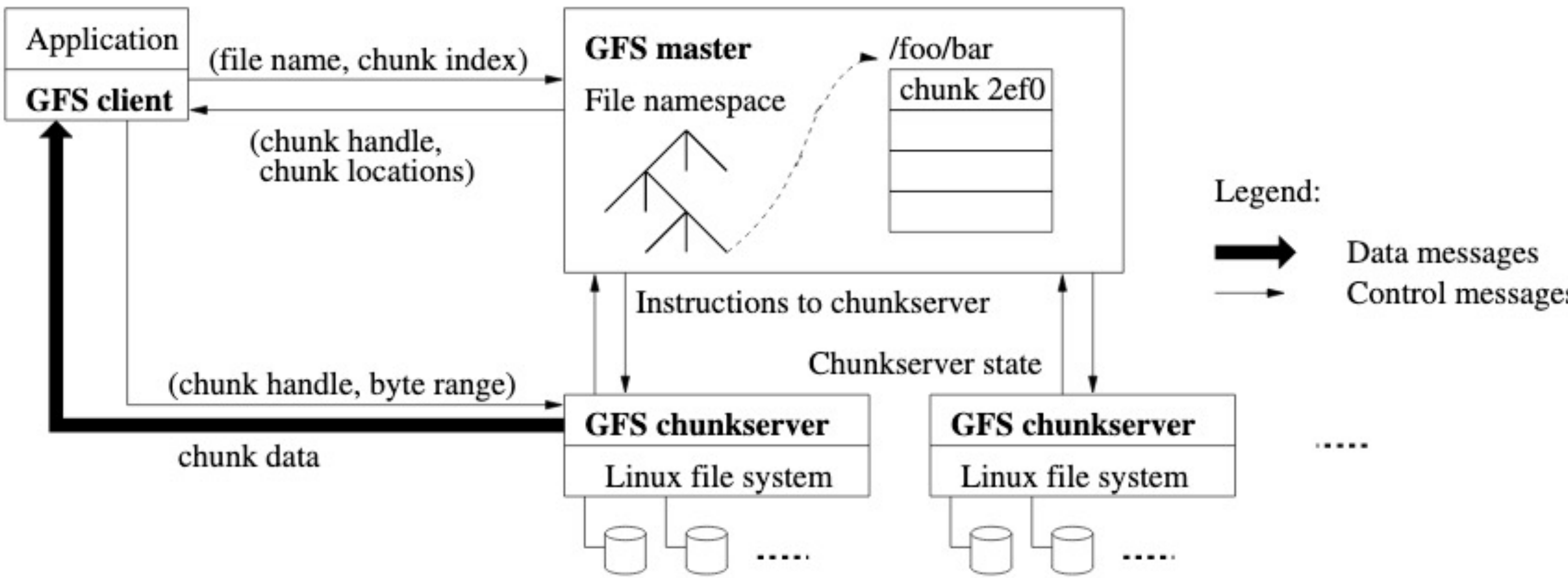
Client reads a chunk



File namespace

path names mapped to logical names

GFS architecture (original paper)



MapReduce+GFS: Put everything together