Google File System

DS 5110: Big Data Systems (Spring 2023)

Lecture 3a

Yue Cheng

University of Virginia

Google file system (GFS)

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

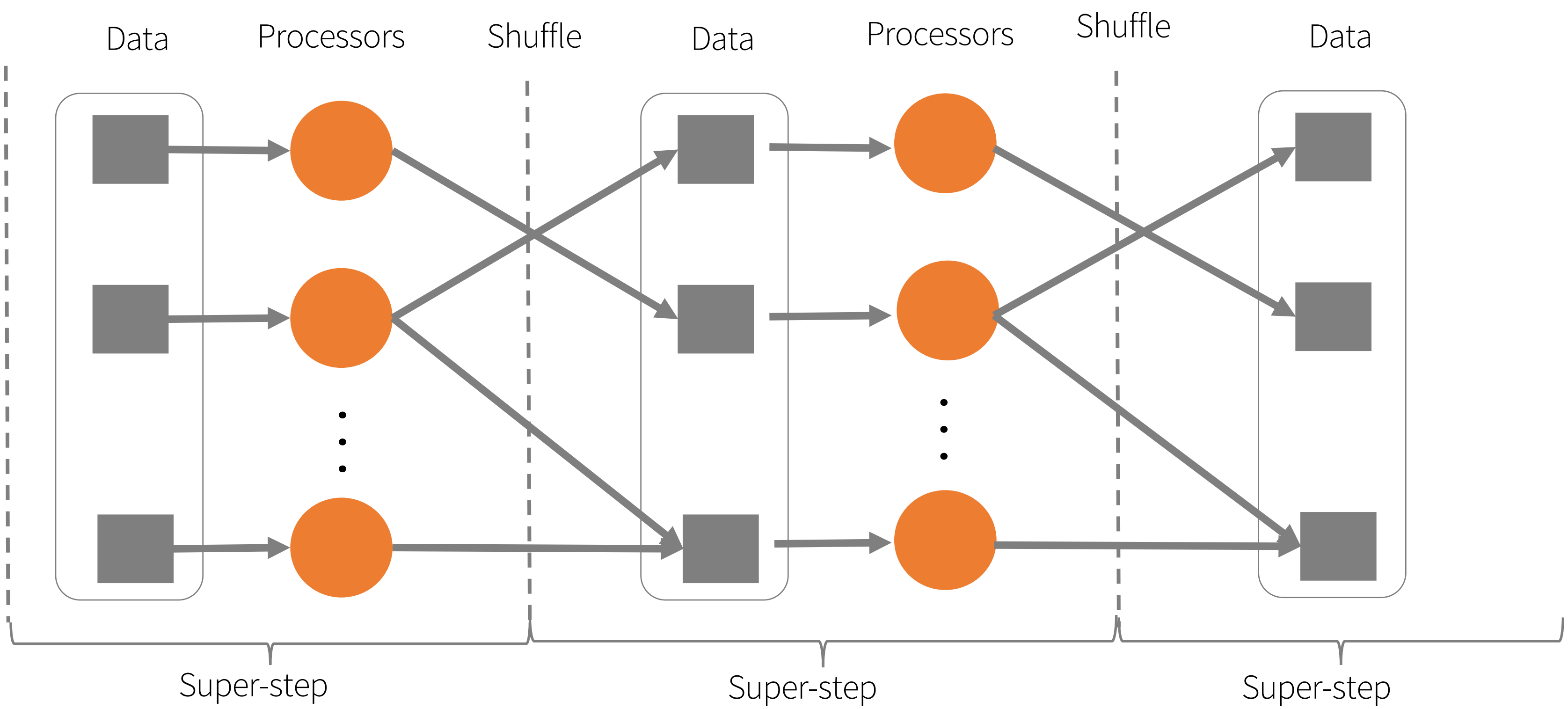
- Google published details in 2003

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

Workload-driven design

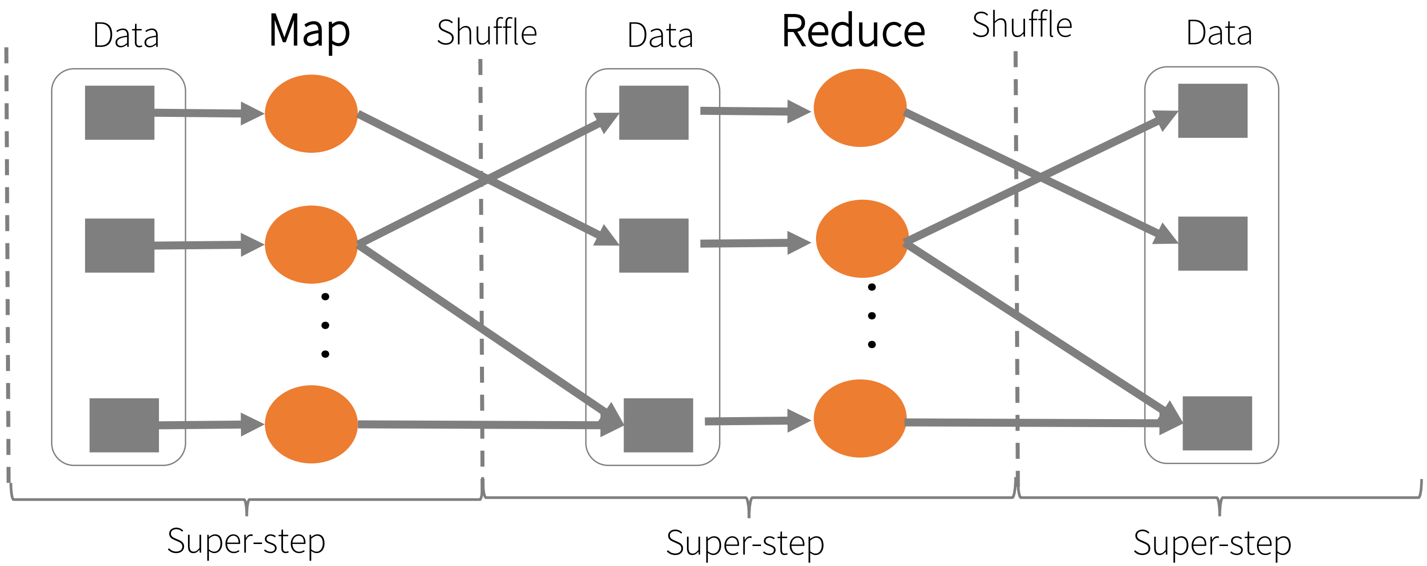
- MapReduce workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

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

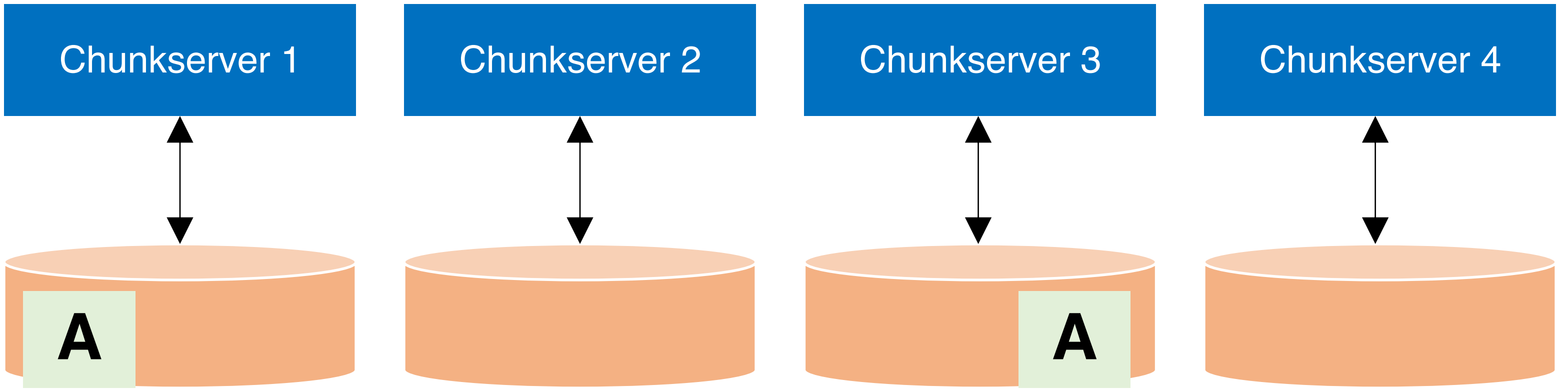
Workload-driven design

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

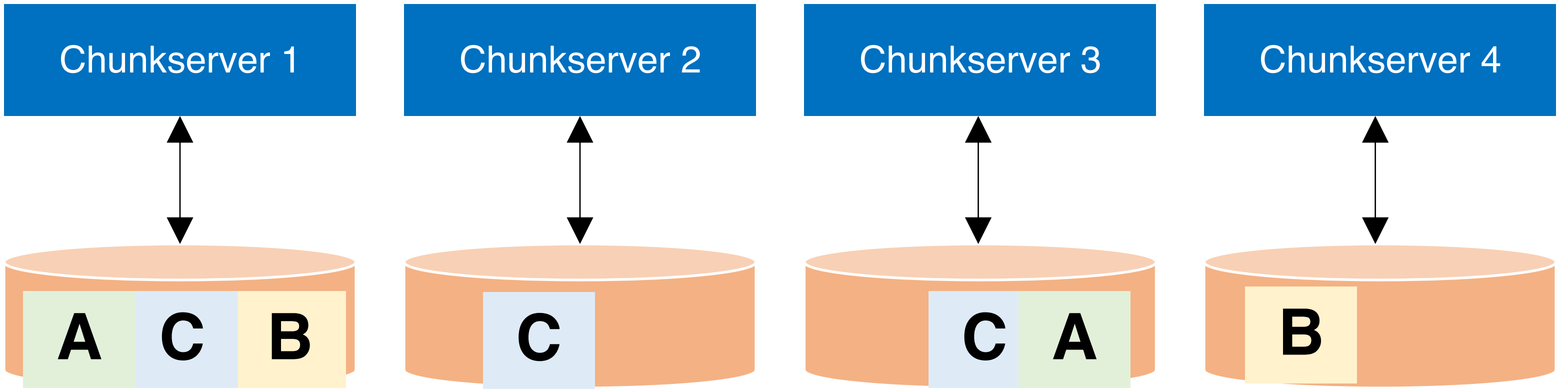
- Only supports **features required by applications**

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

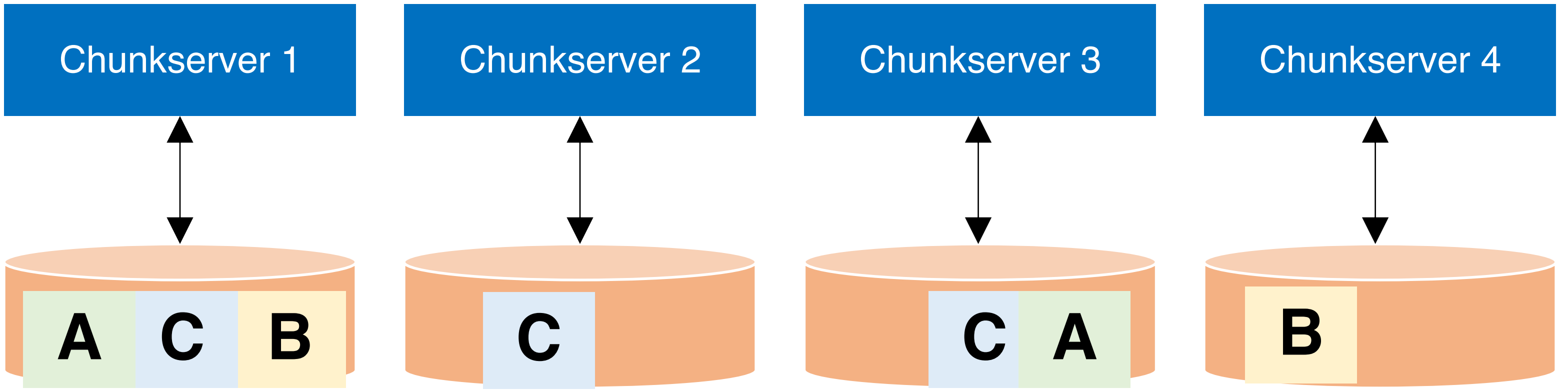
Replication



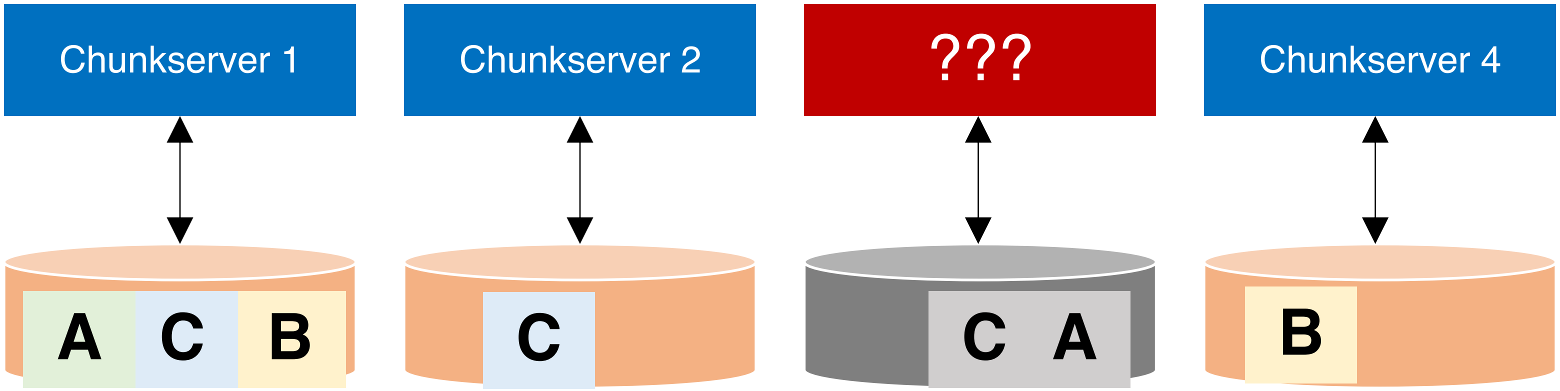
Replication



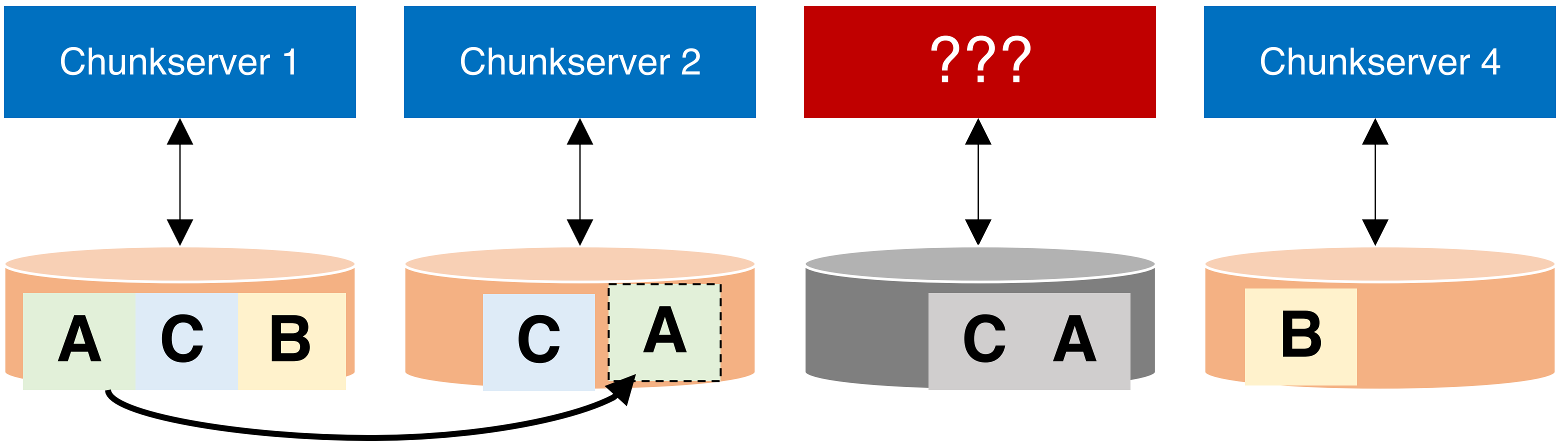
Resilience against failures



Resilience against failures

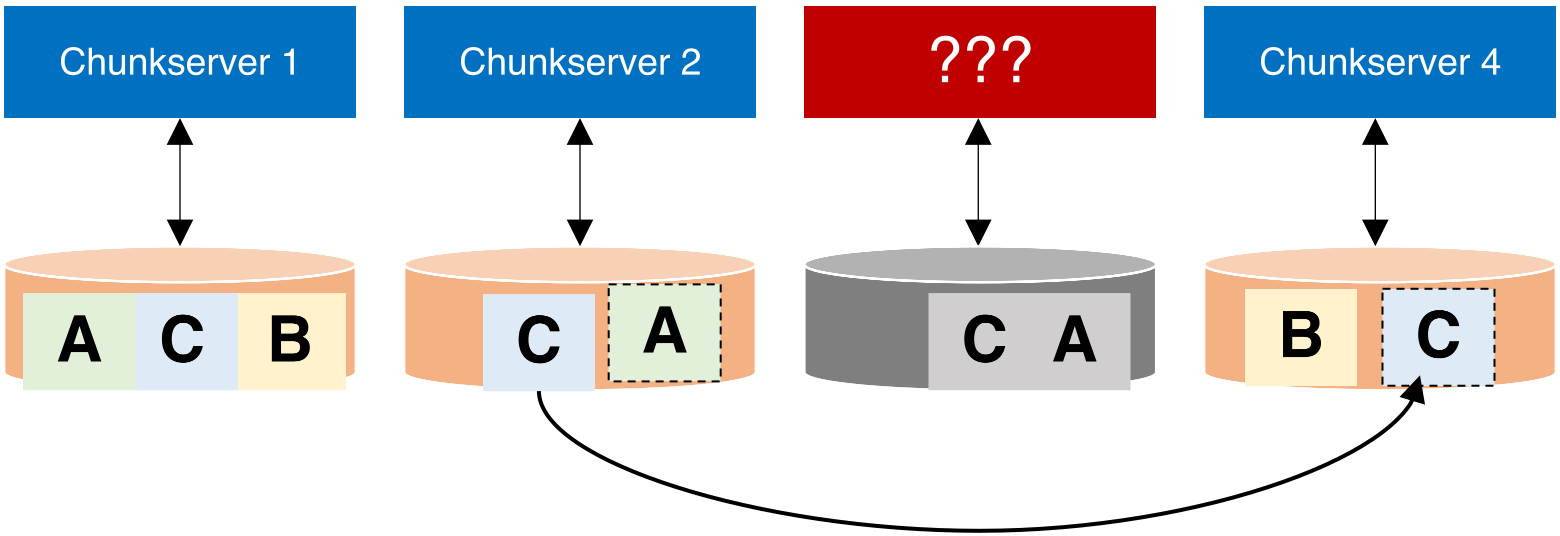


Data recovery



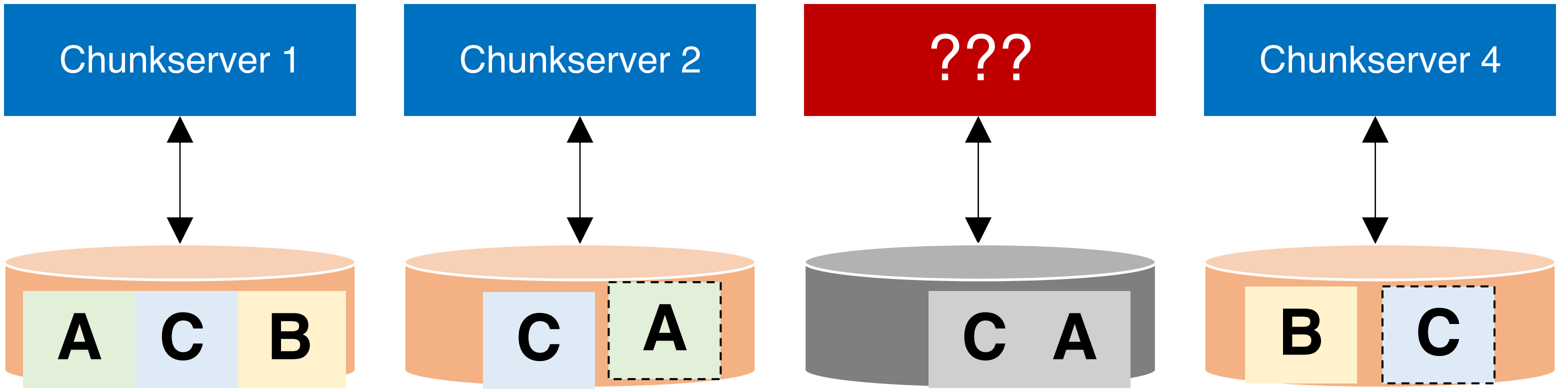
Replicating A to maintain a replication factor of 2

Data recovery



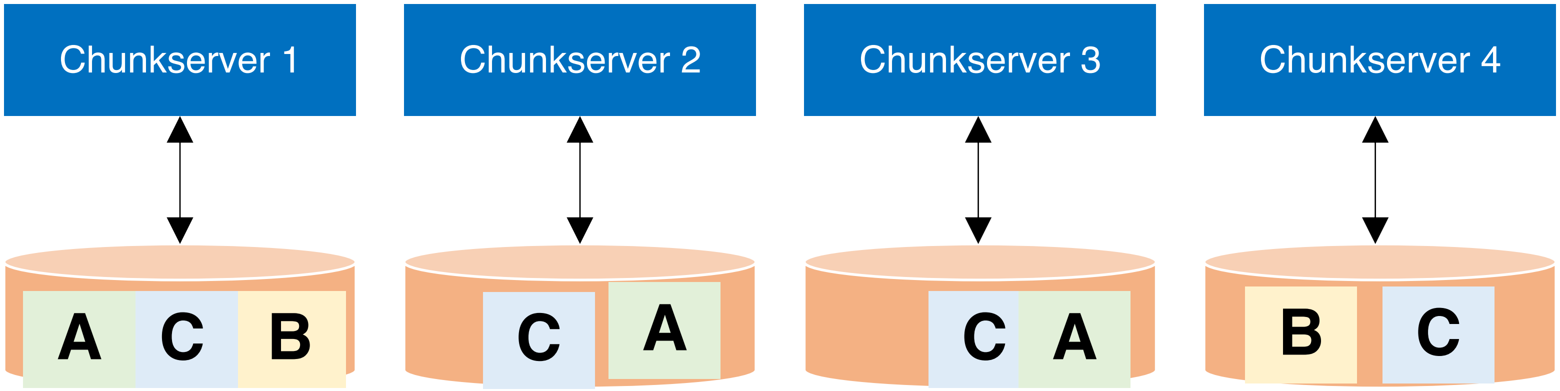
Replicating C to maintain a replication factor of 3

Data recovery



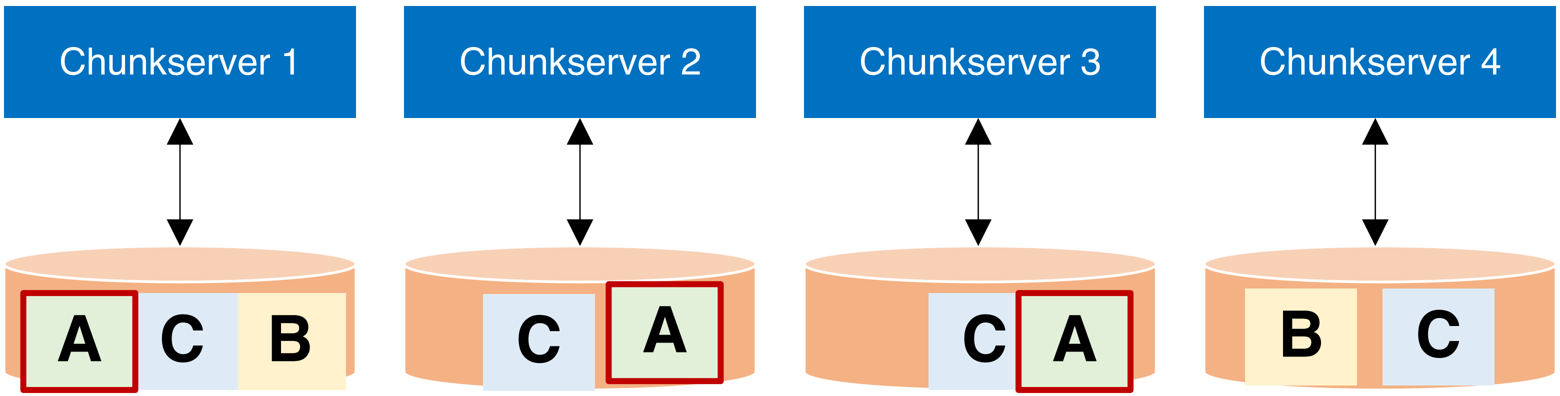
Machine may be dead forever, or it may come back

Data recovery

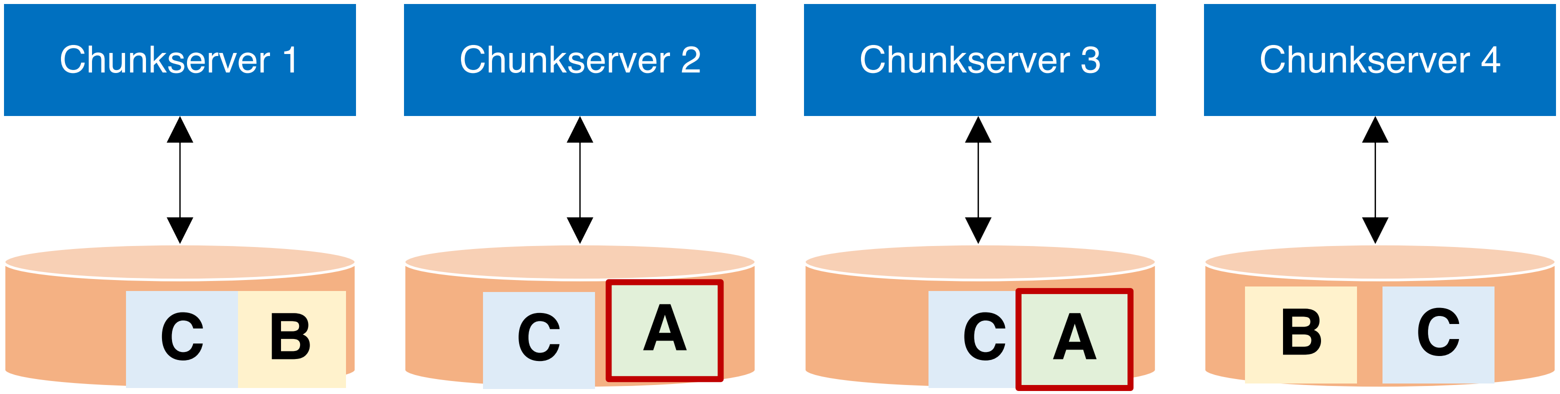


Machine may be dead forever, or it may come back

Data recovery



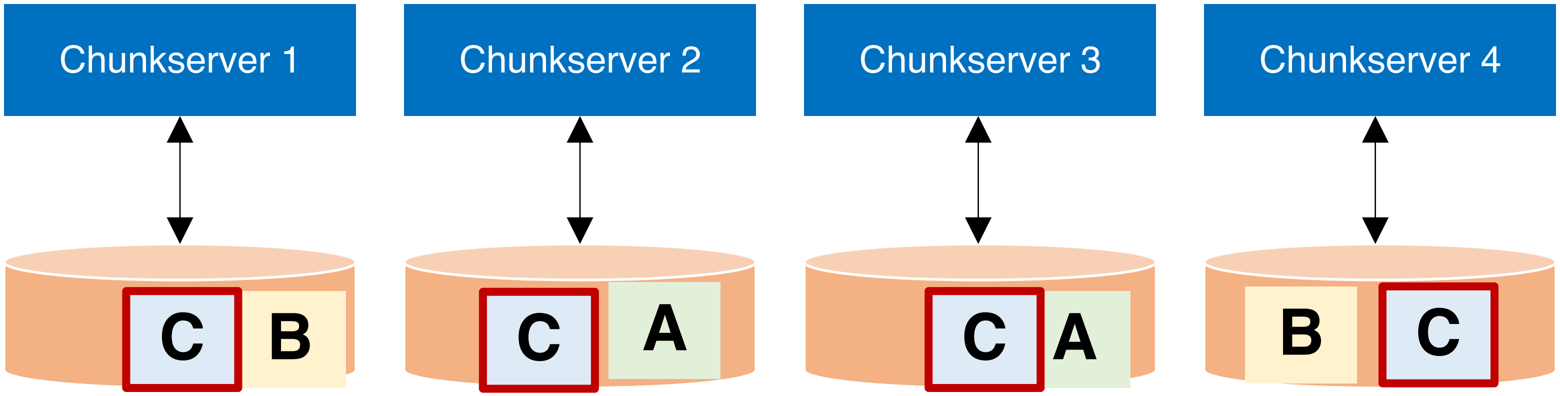
Data recovery



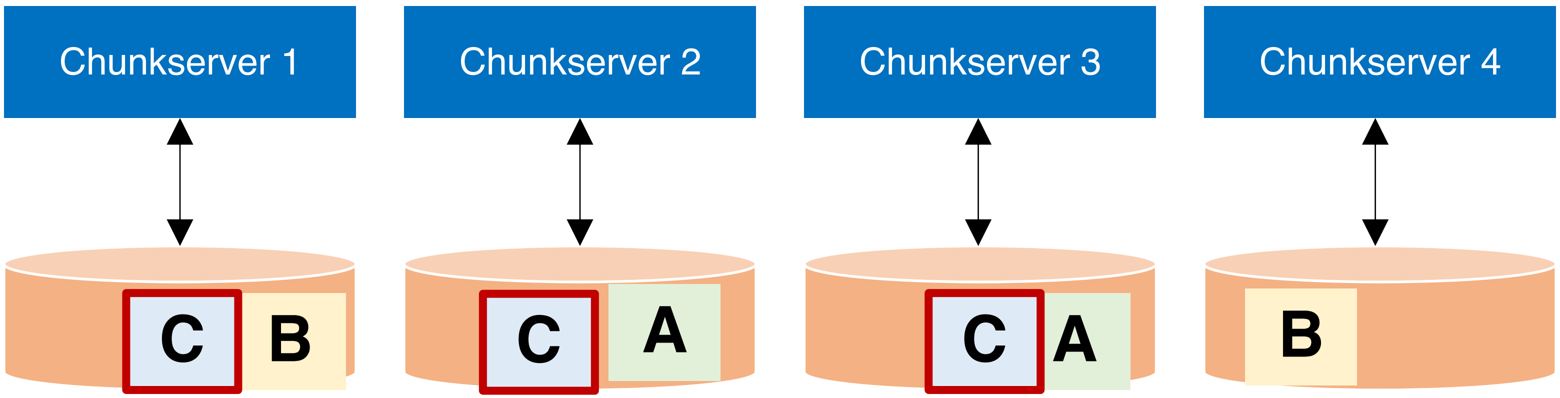
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data recovery



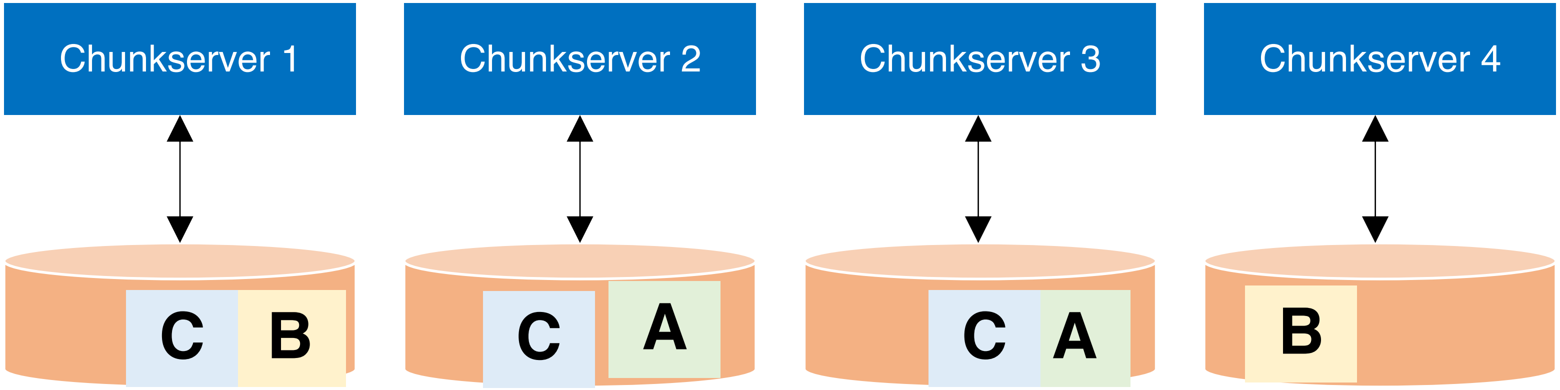
Data recovery



Data Rebalancing

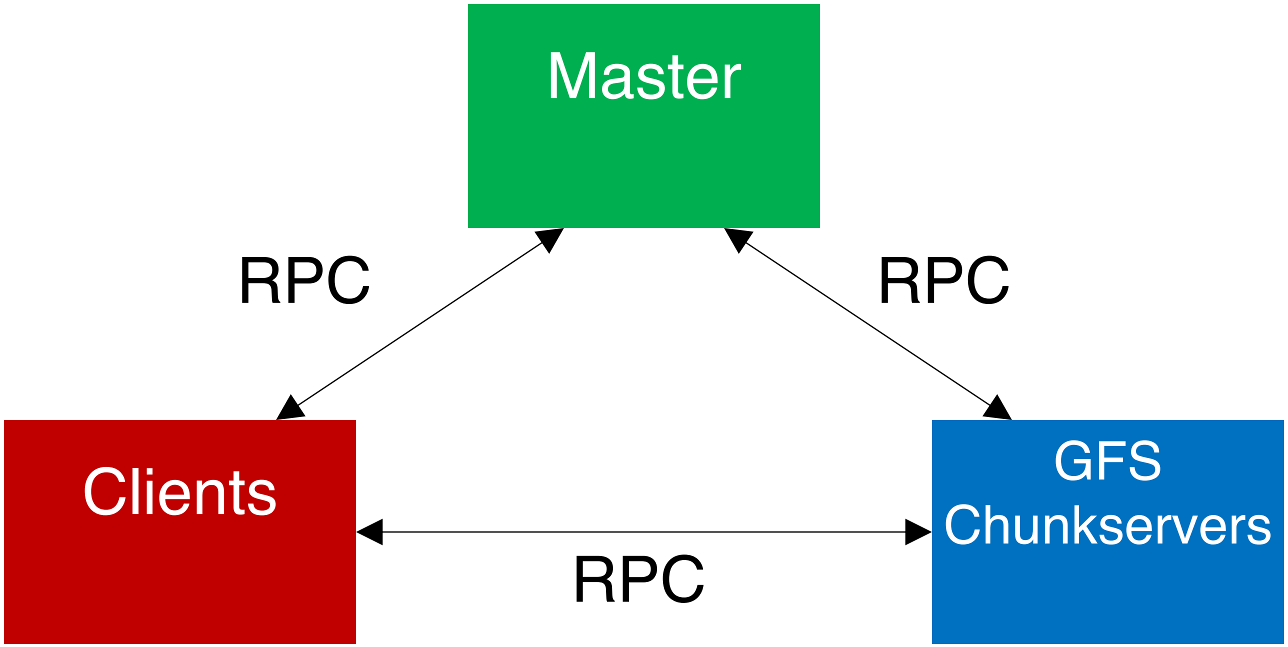
Deleting one C to maintain a replication factor of 3

Data recovery

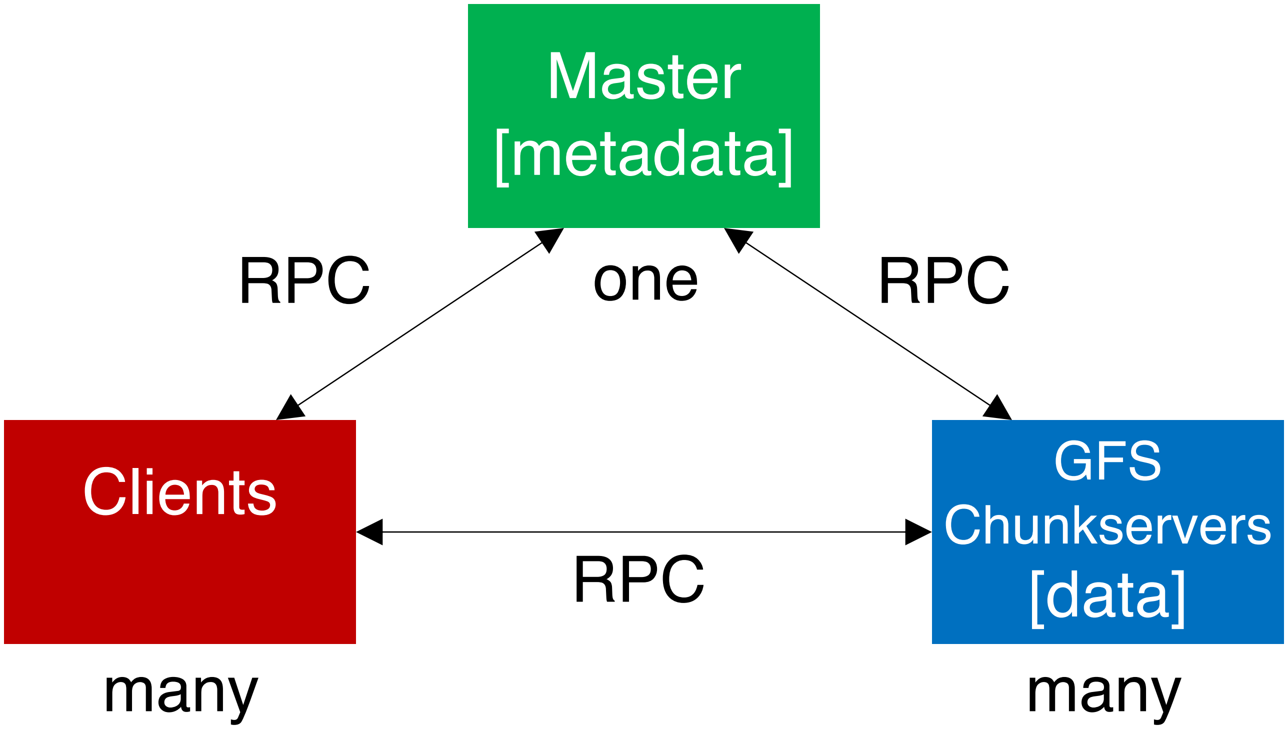


Question: how to maintain a global view of all data distributed across machines?

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

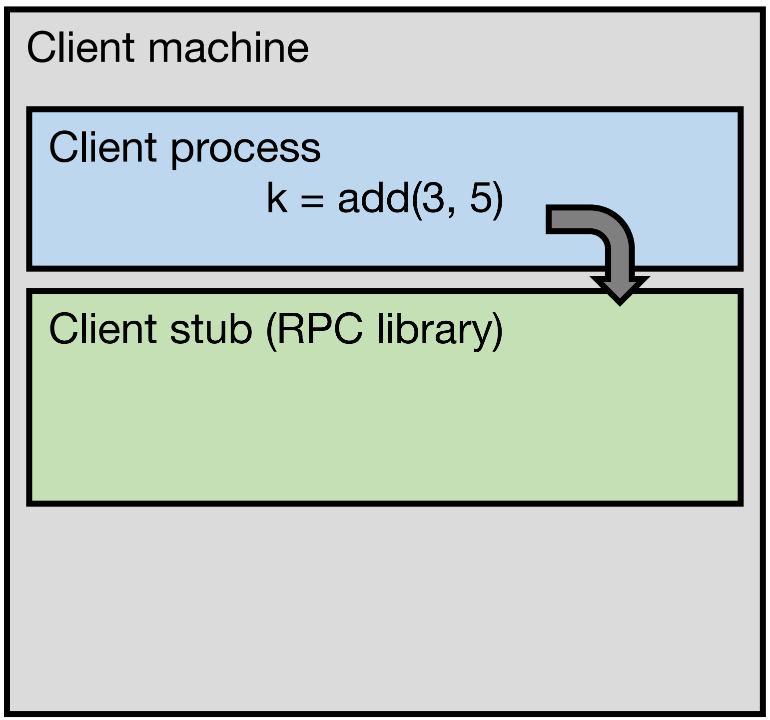
What’s the goal of RPC?

- 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

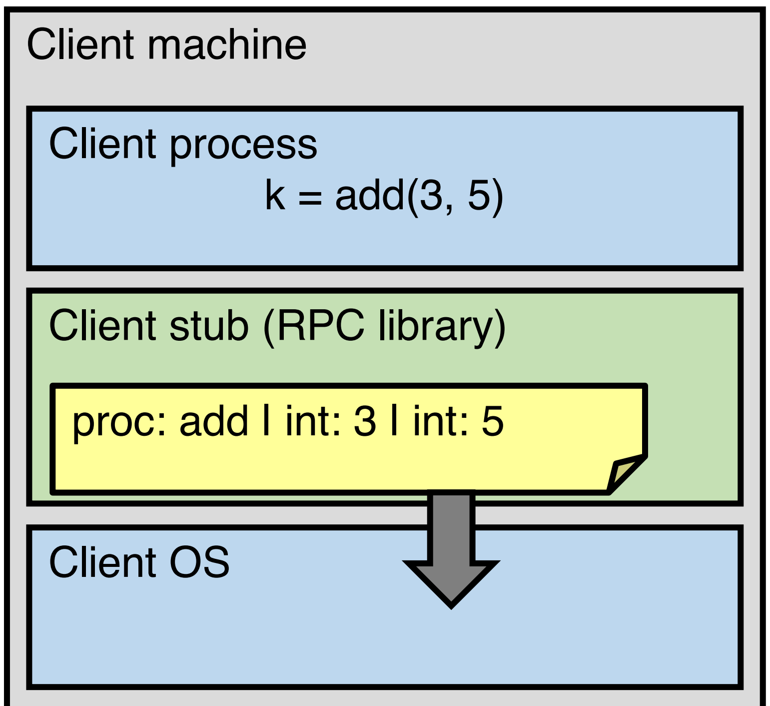
1. Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

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

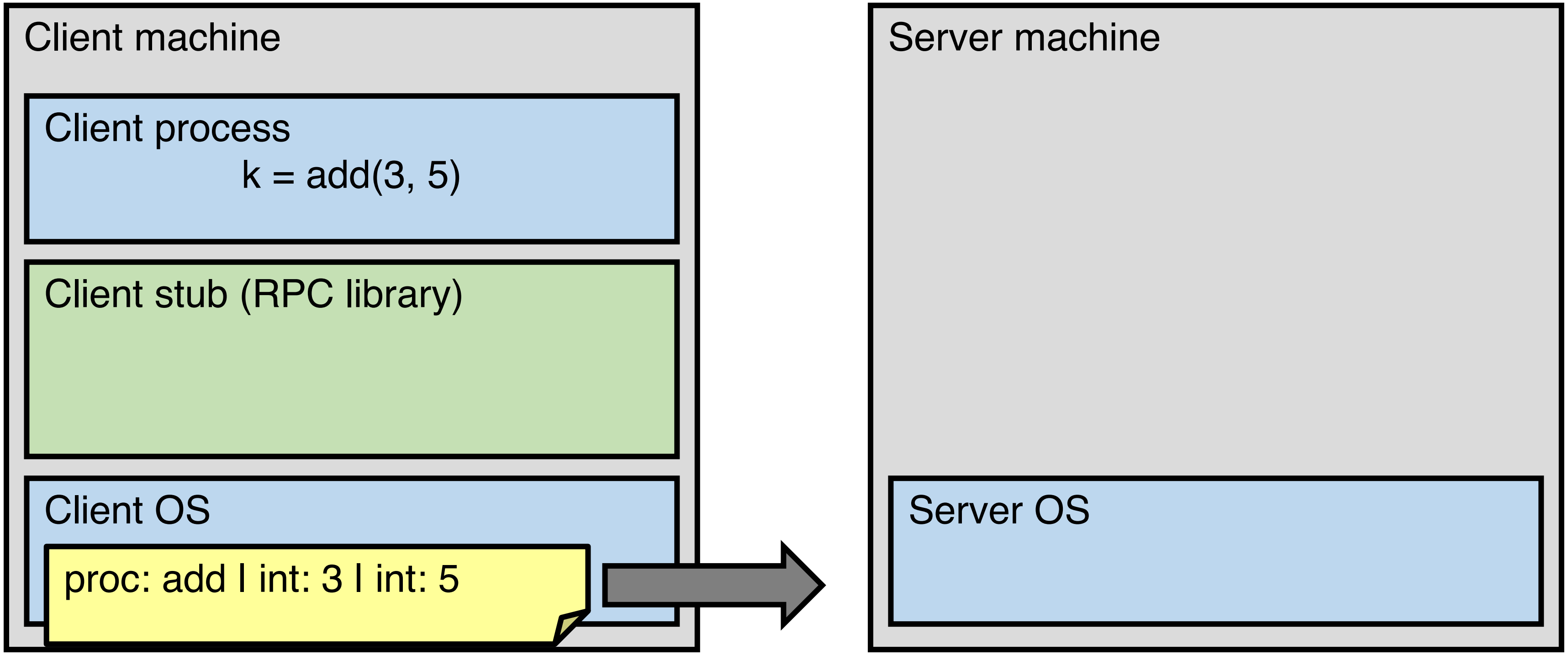
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

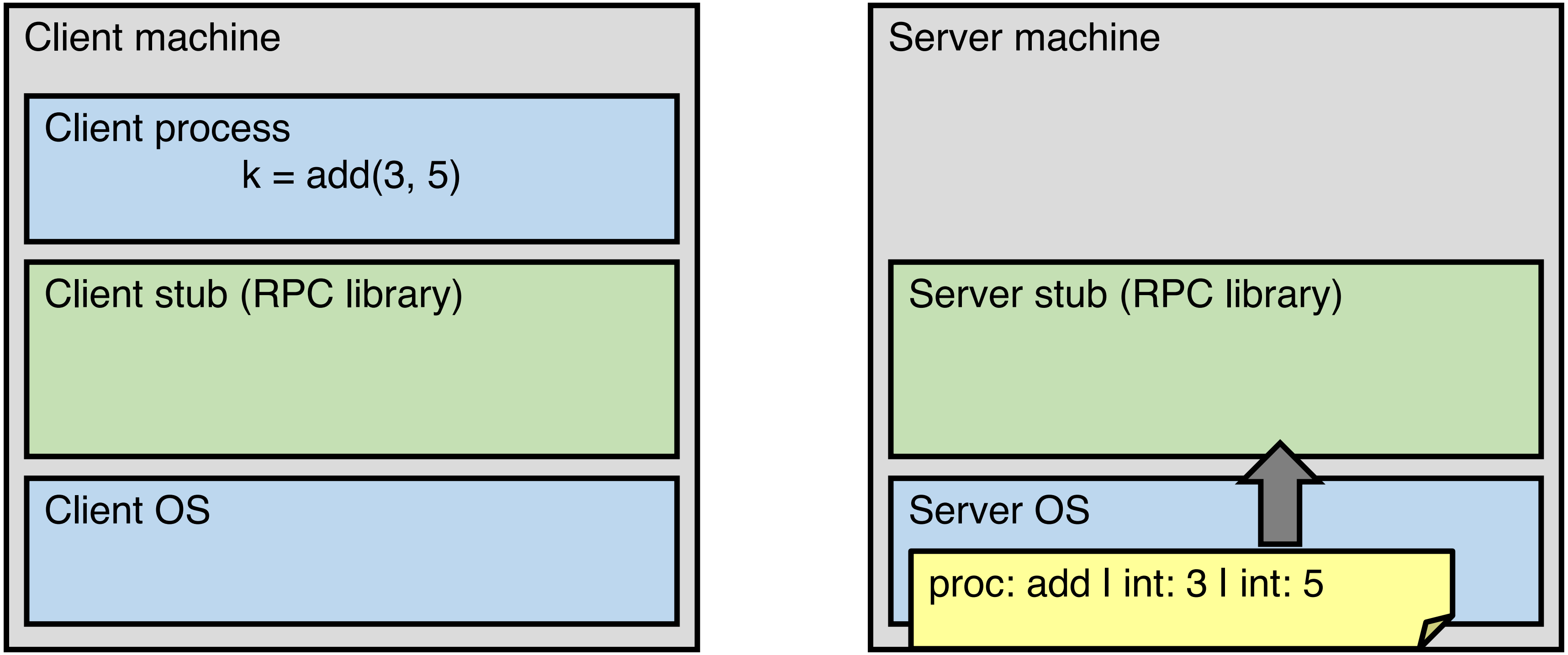
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

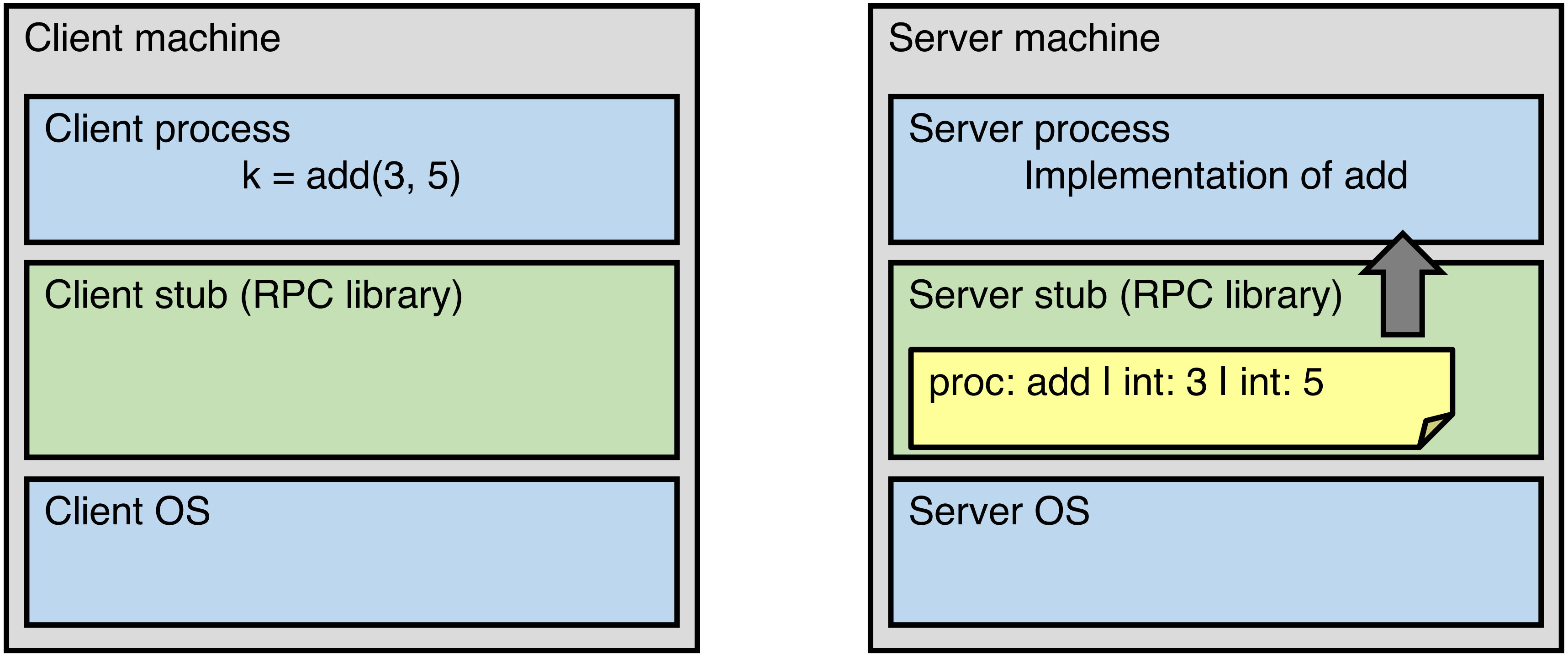
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

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

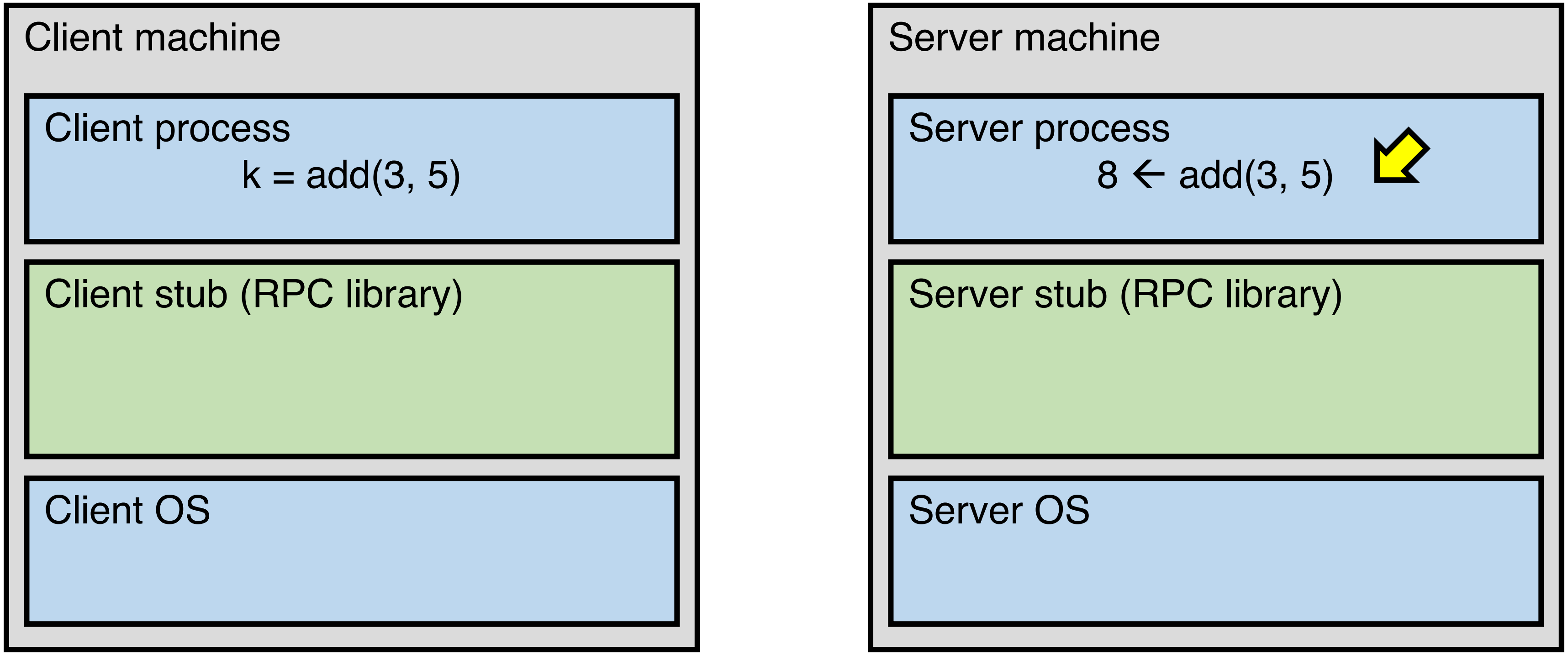
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

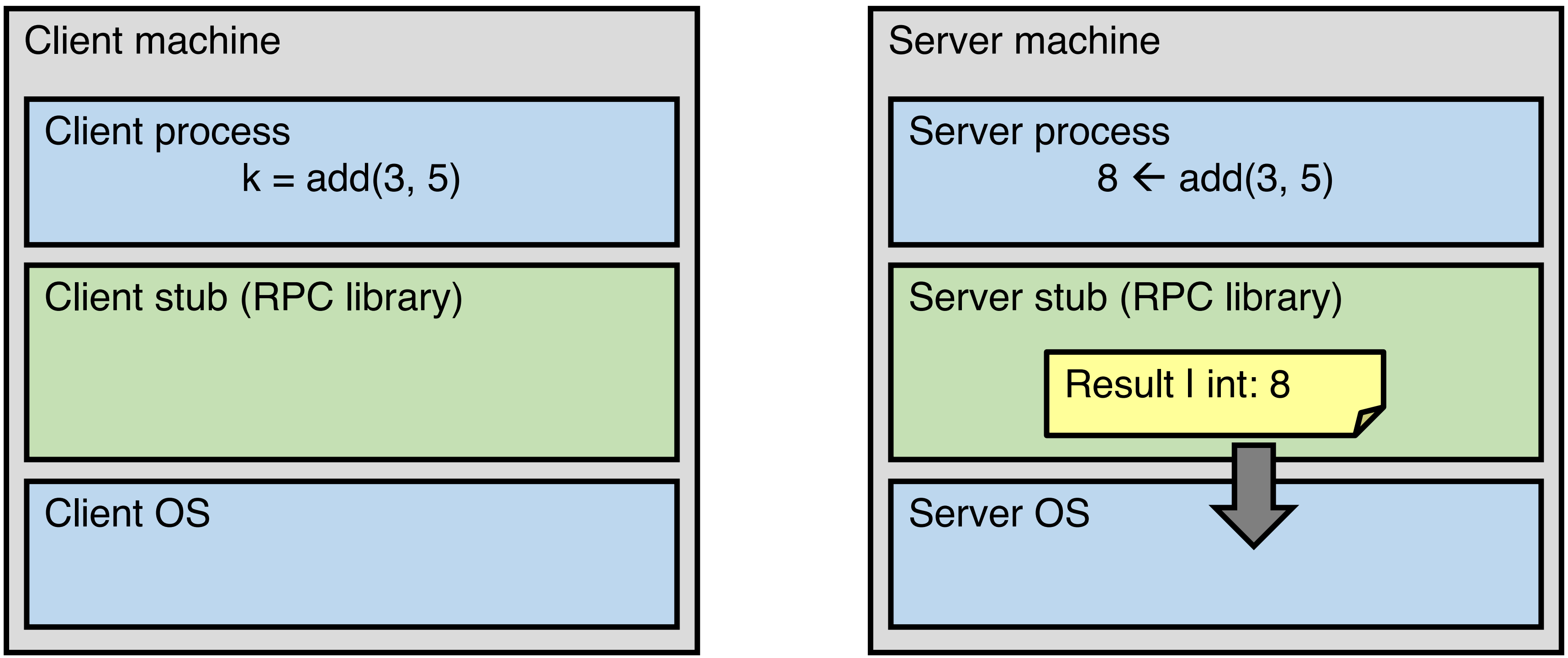
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

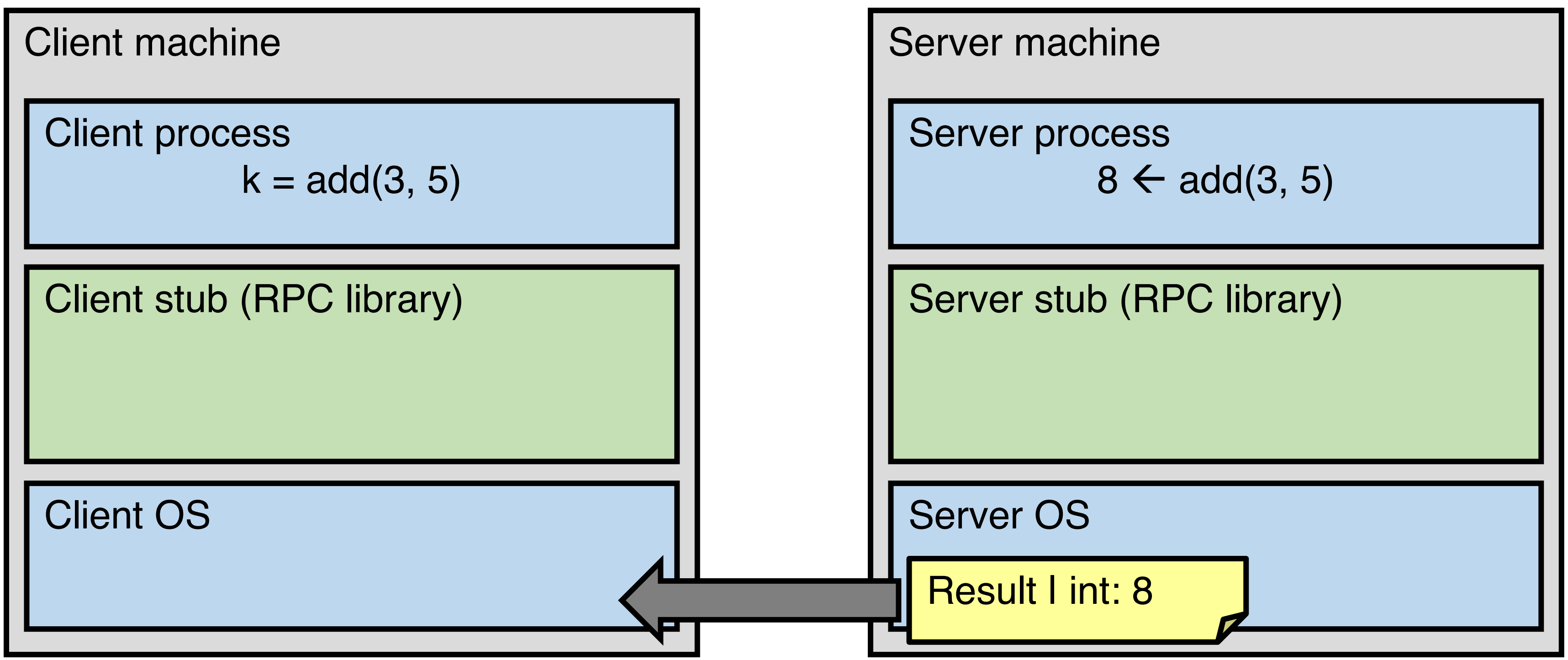
7. Server stub marshals the return value, sends message



A day in the life of an RPC

7. Server stub marshals the return value, sends message

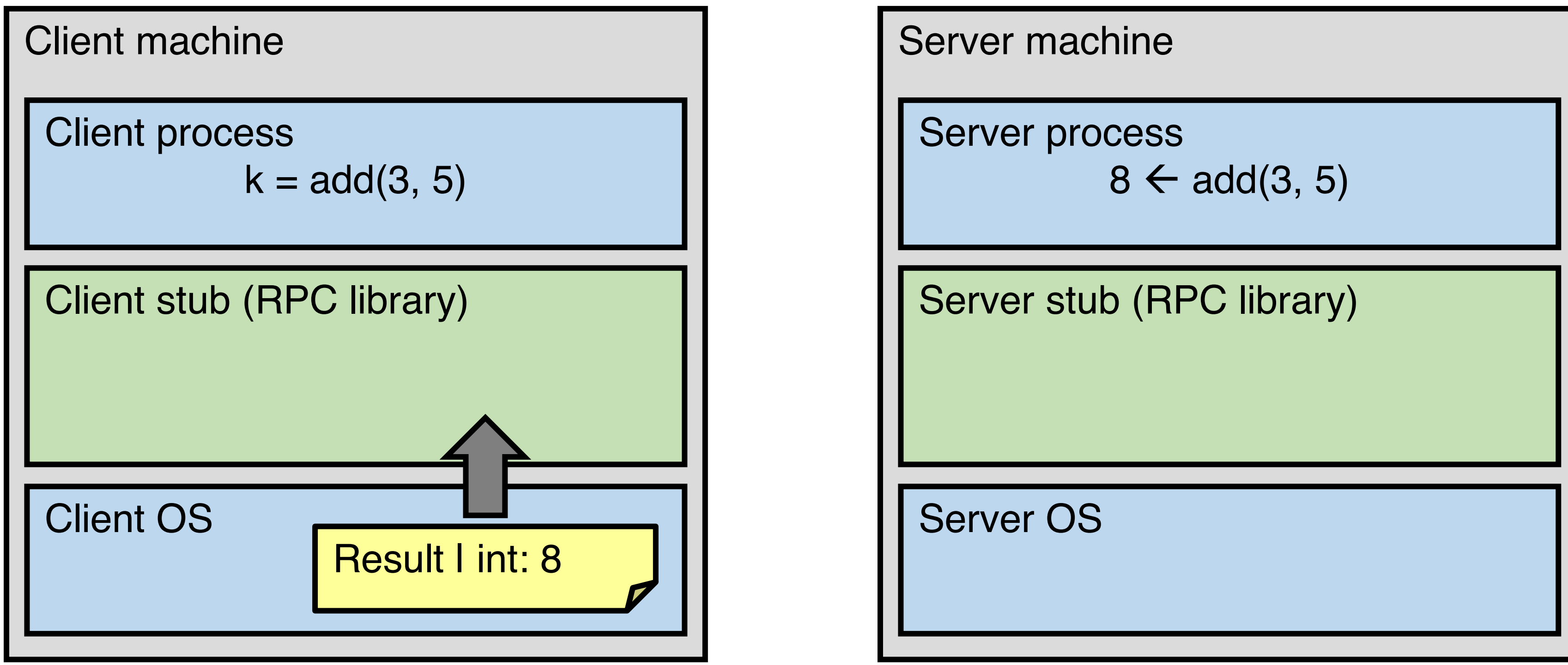
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

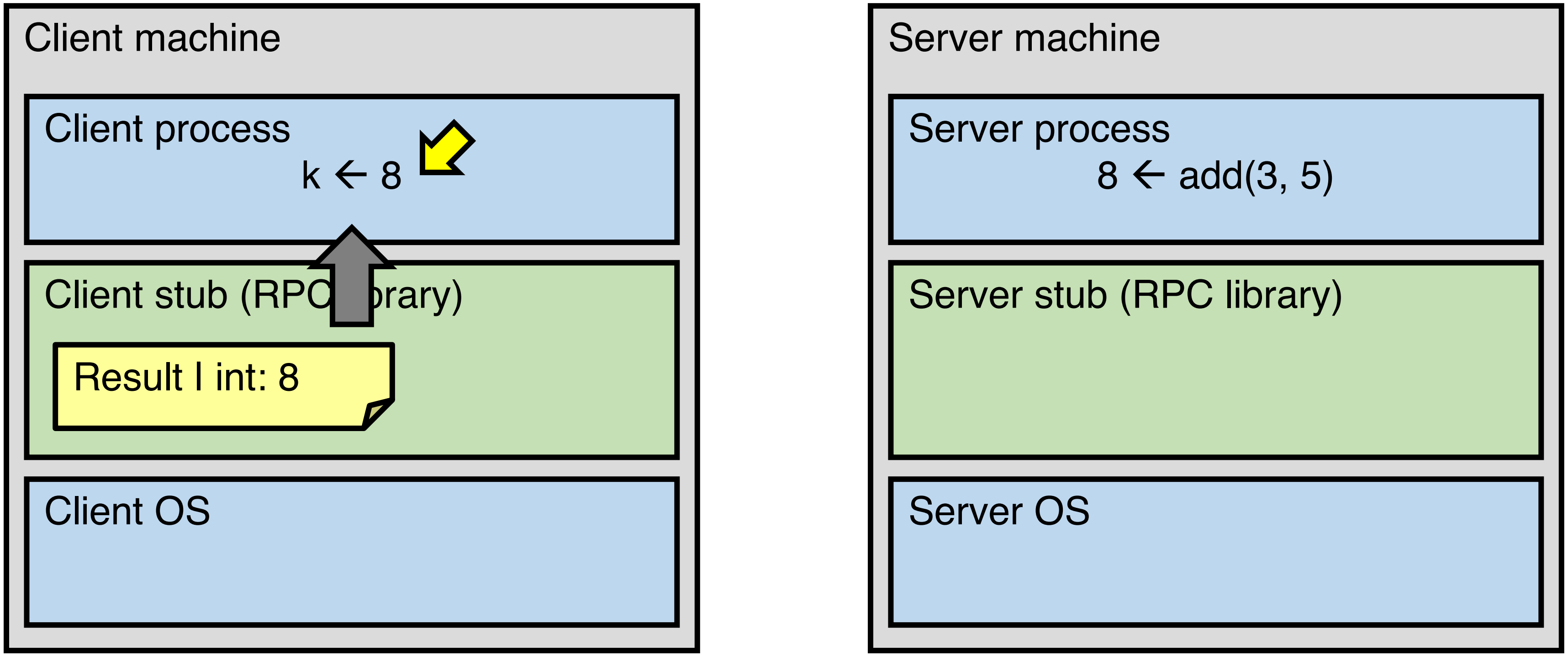
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

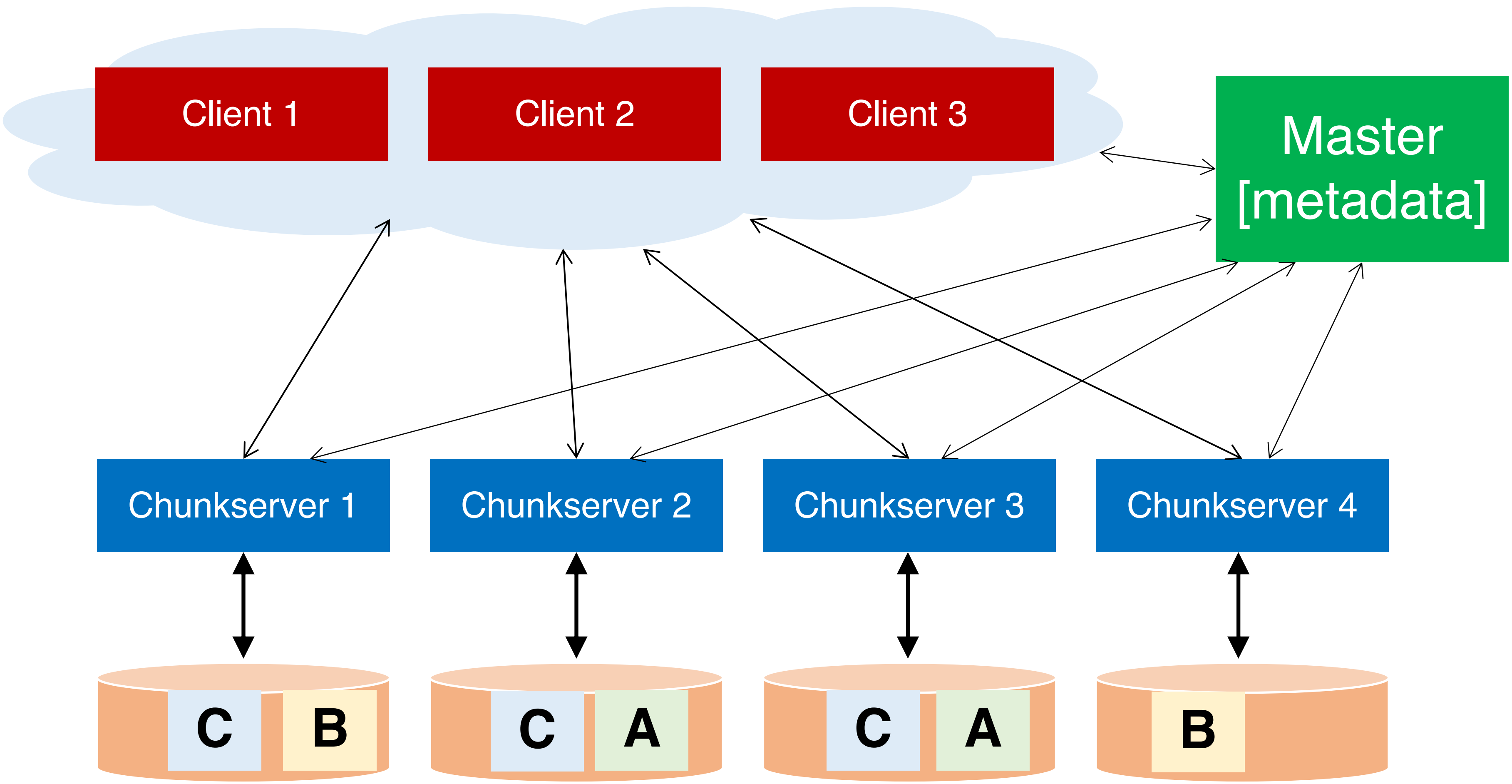
9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client



Back to GFS

GFS architecture: physical view



Data chunks

- Break large GFS files into coarse-grained data chunks (e.g., 64-128MB)
- GFS chunkservers store physical data chunks in local Linux file system
- Centralized master keeps track of mapping between logical and physical chunks

HDFS demo

Writing to a file

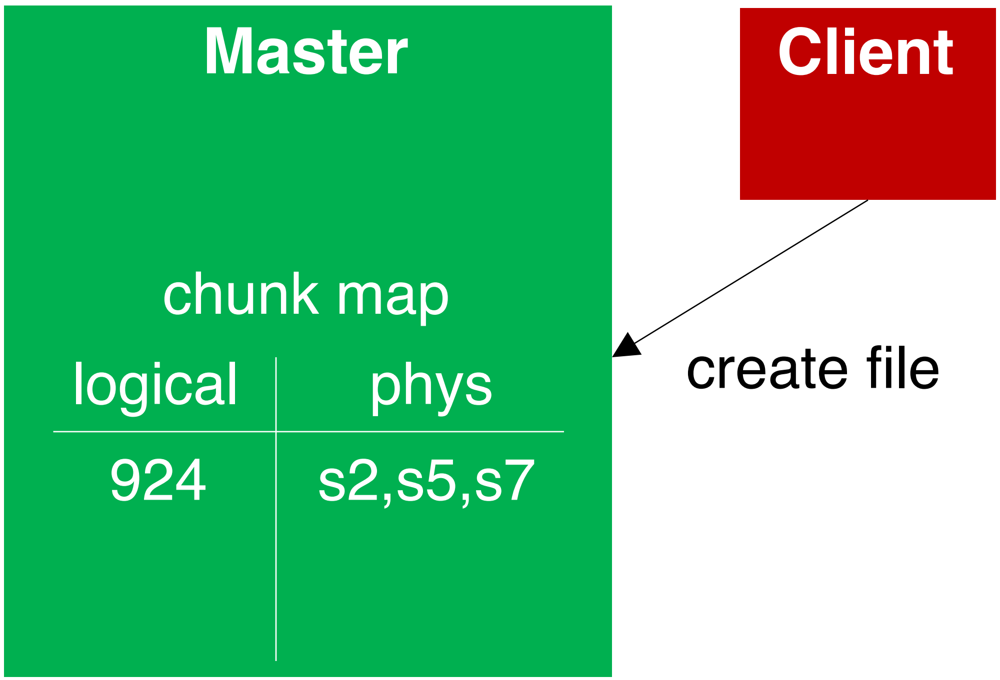
Chunk map: the metadata

Master

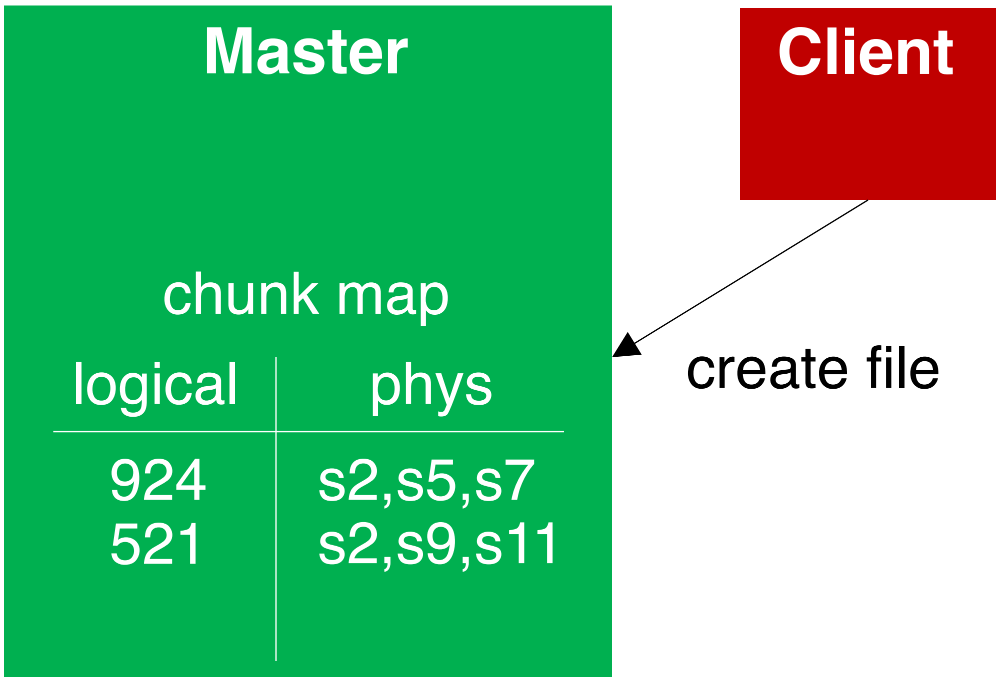
chunk map

logical	phys
924	s2,s5,s7

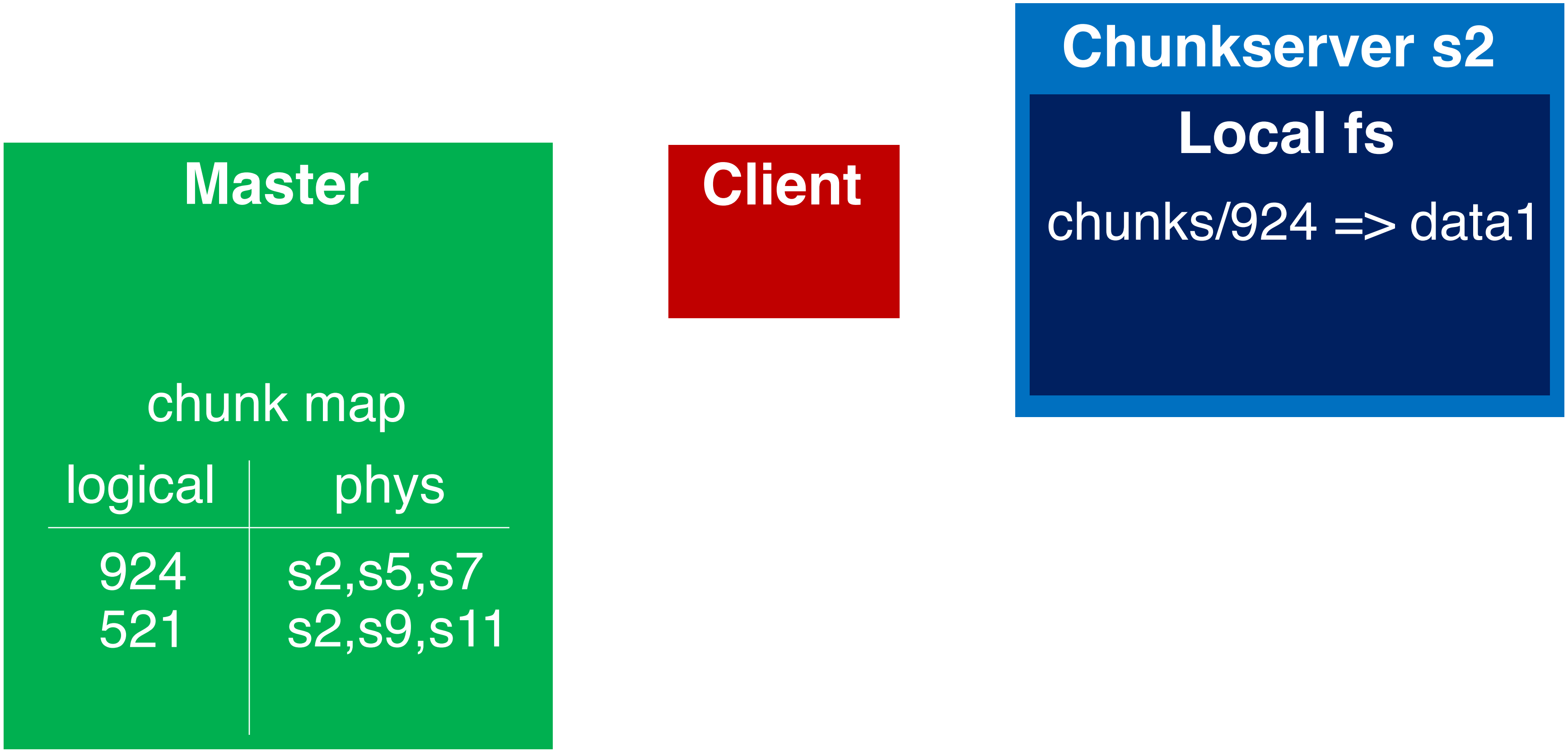
Client contacts the GFS master



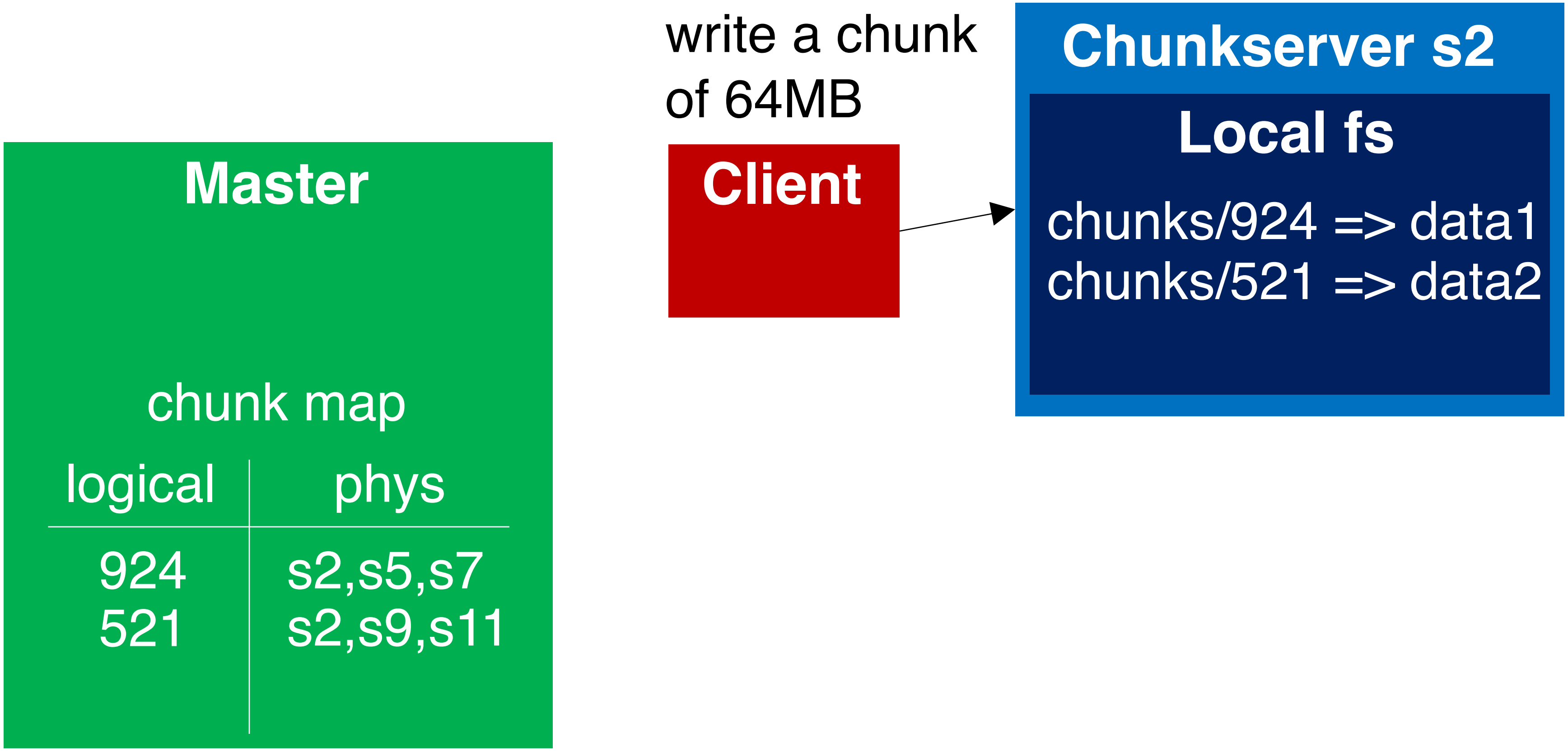
GFS master creates file metadata



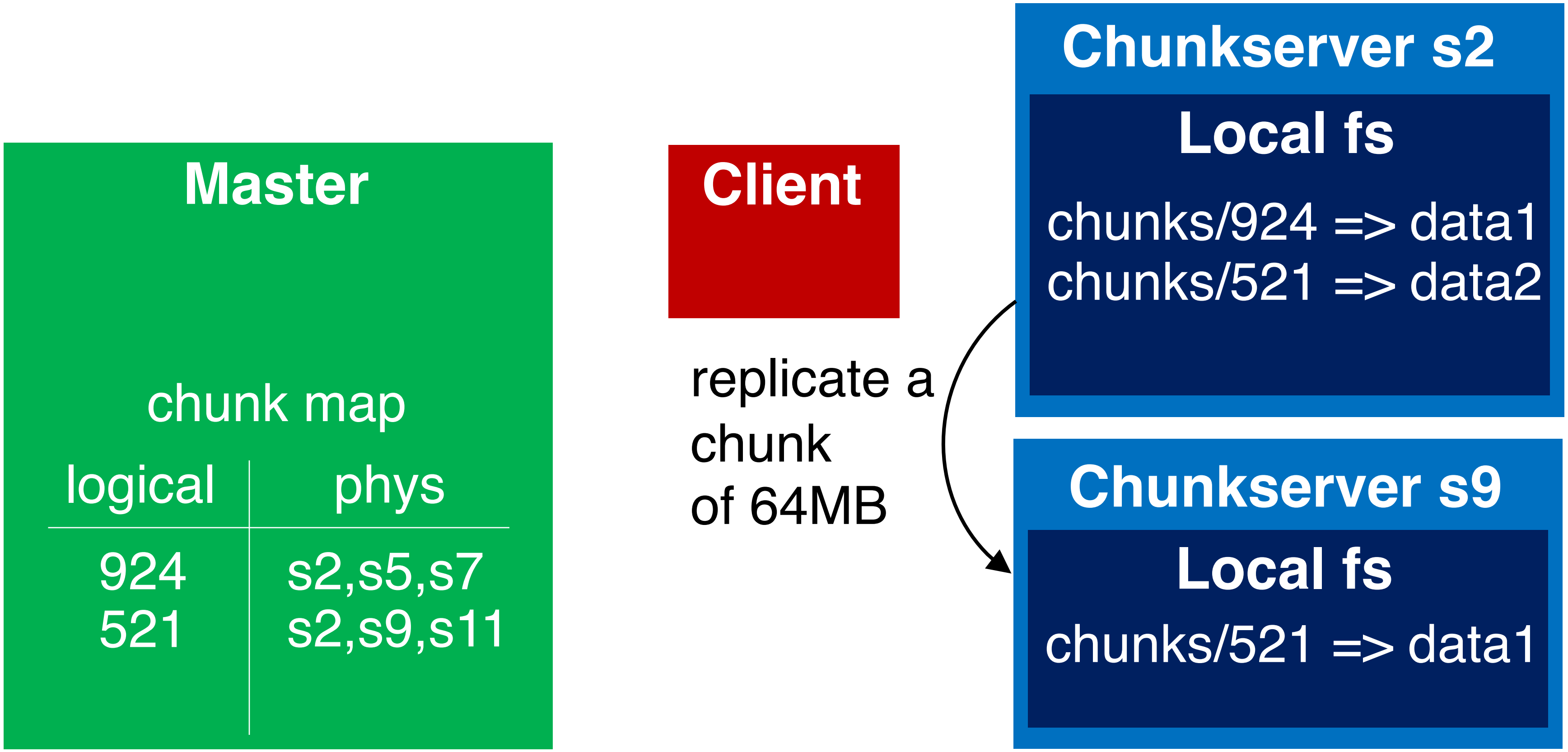
Client writes replicas to chunkservers



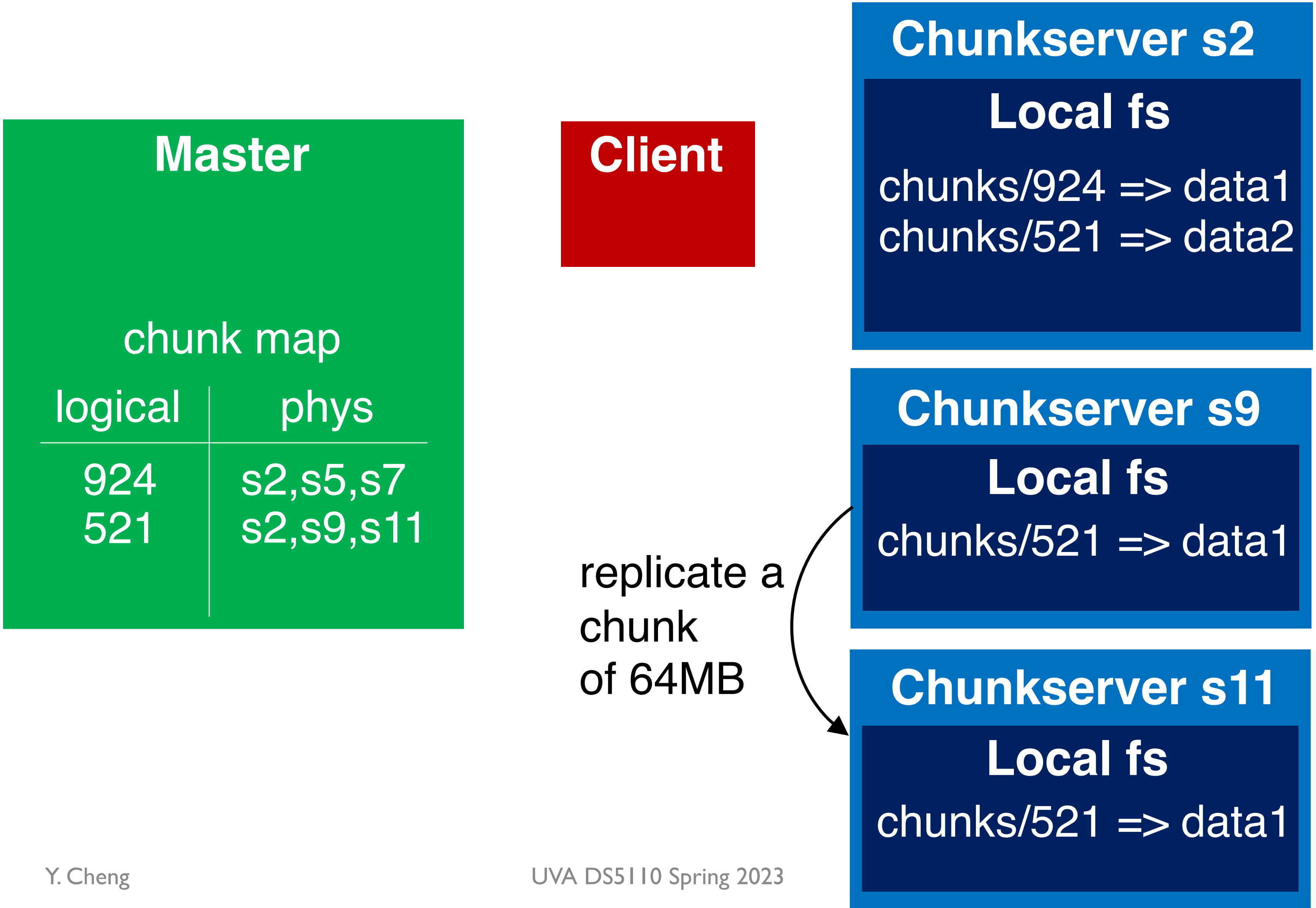
Client writes replica to s2



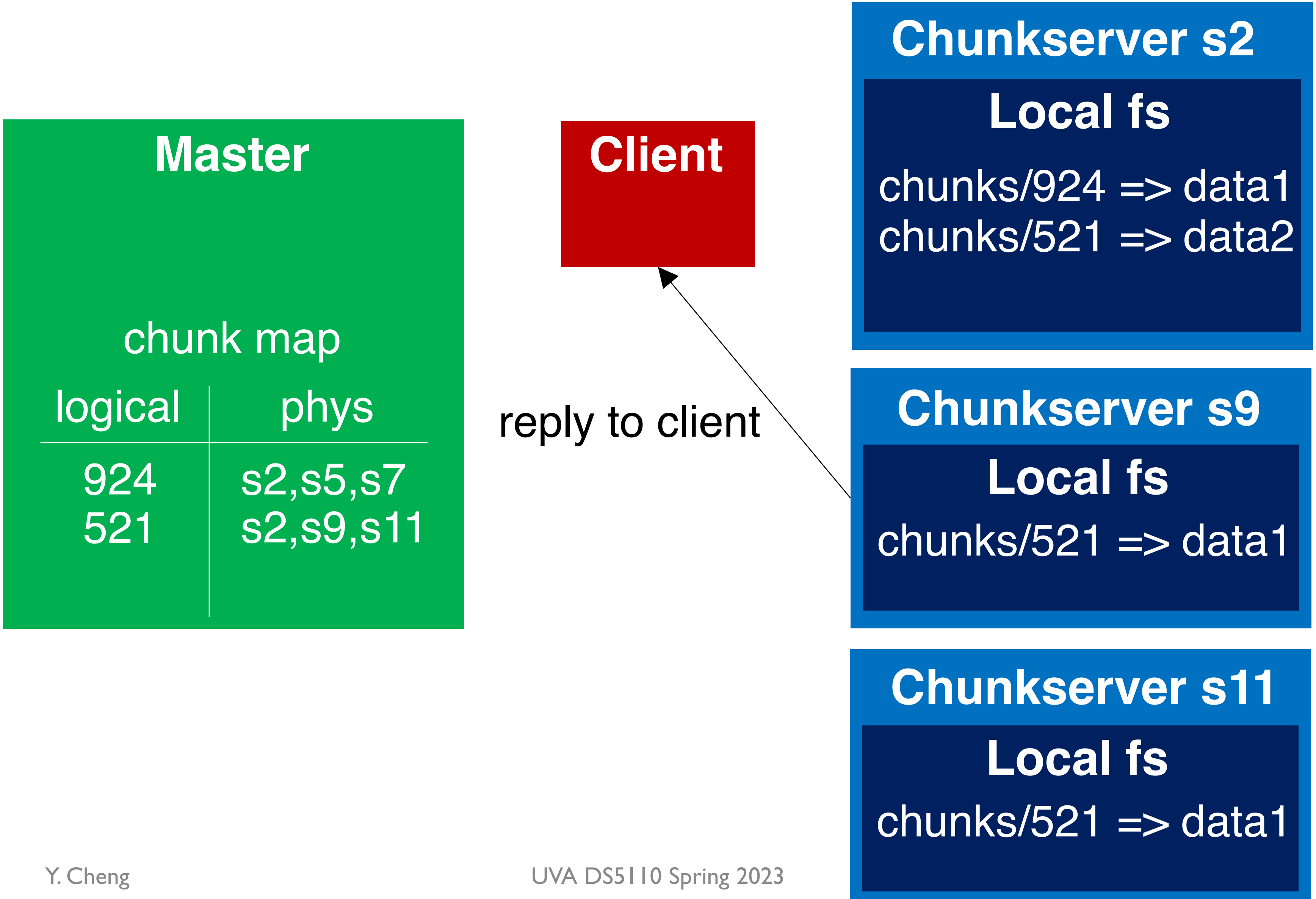
S2 streams replica to s9



S9 streams replica to s11



Primary replica s9 acks back



Reading a file

Chunk map: the metadata

Master

chunk map

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

Chunkservers {s2,s5,s7} hold a data chunk

Master

chunk map

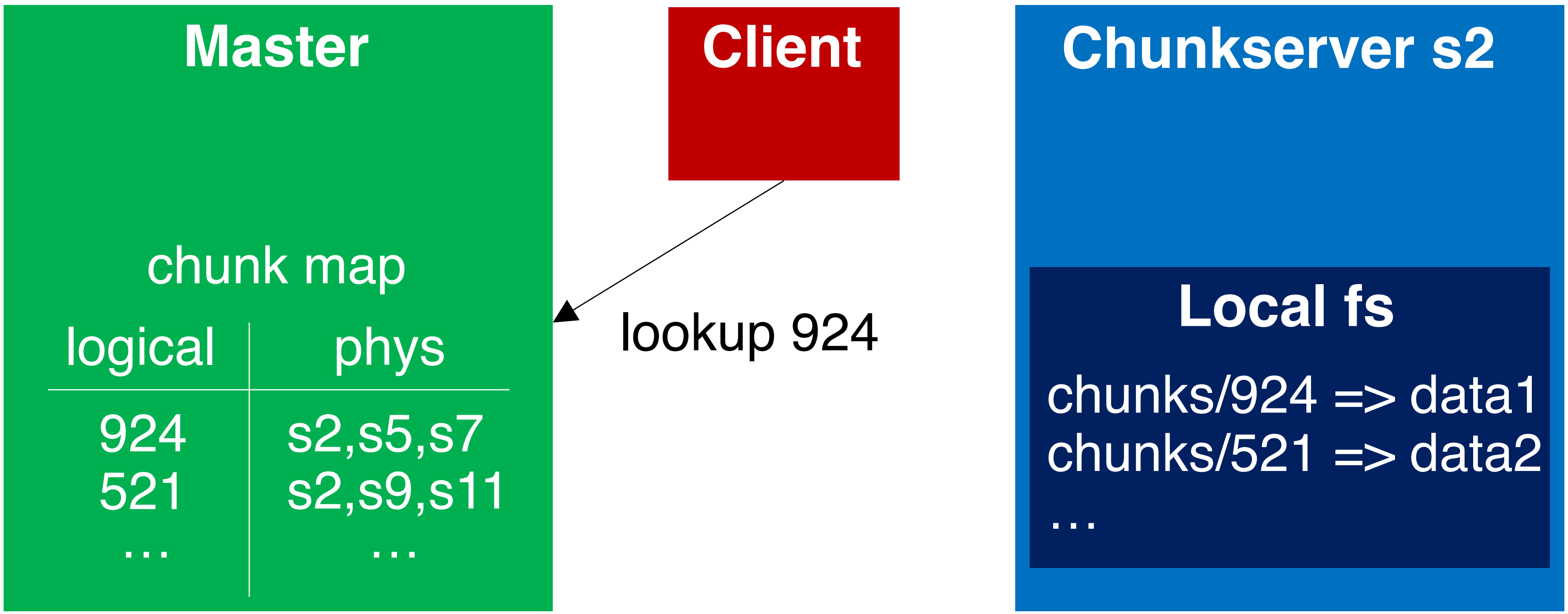
logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

Chunkserver s2

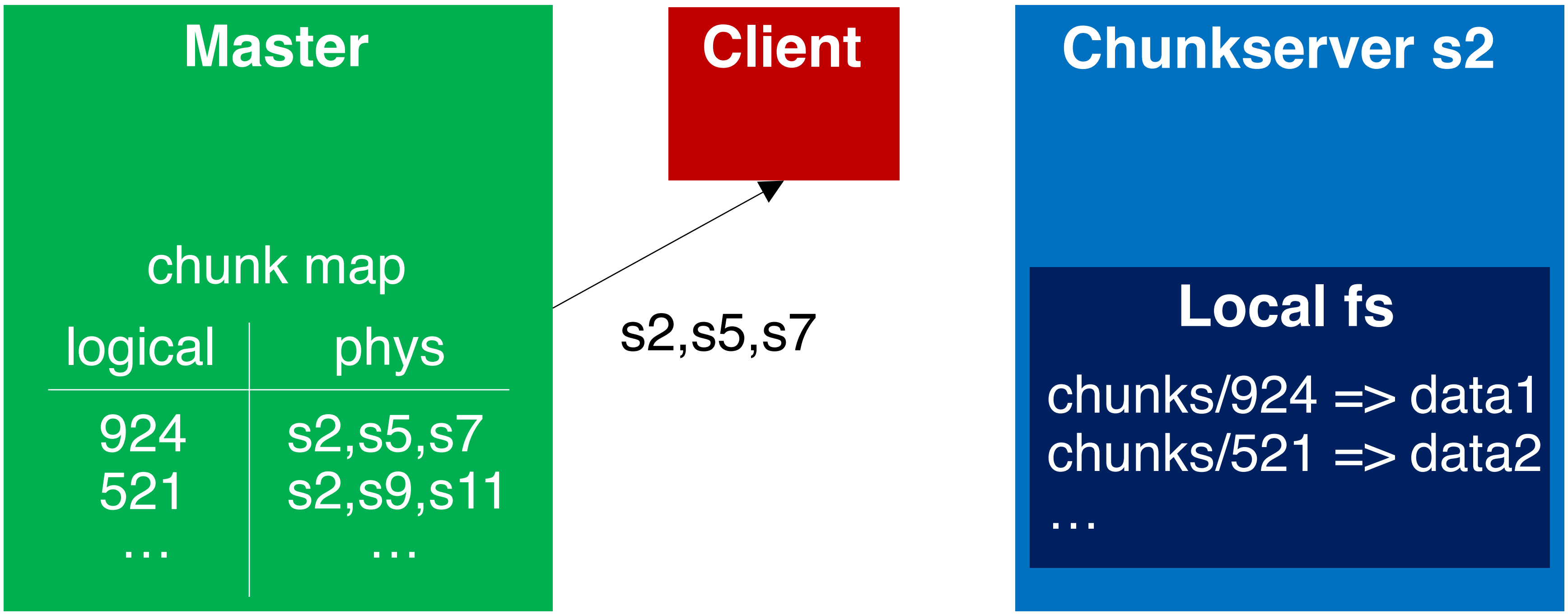
Local fs

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

Client asks for the location

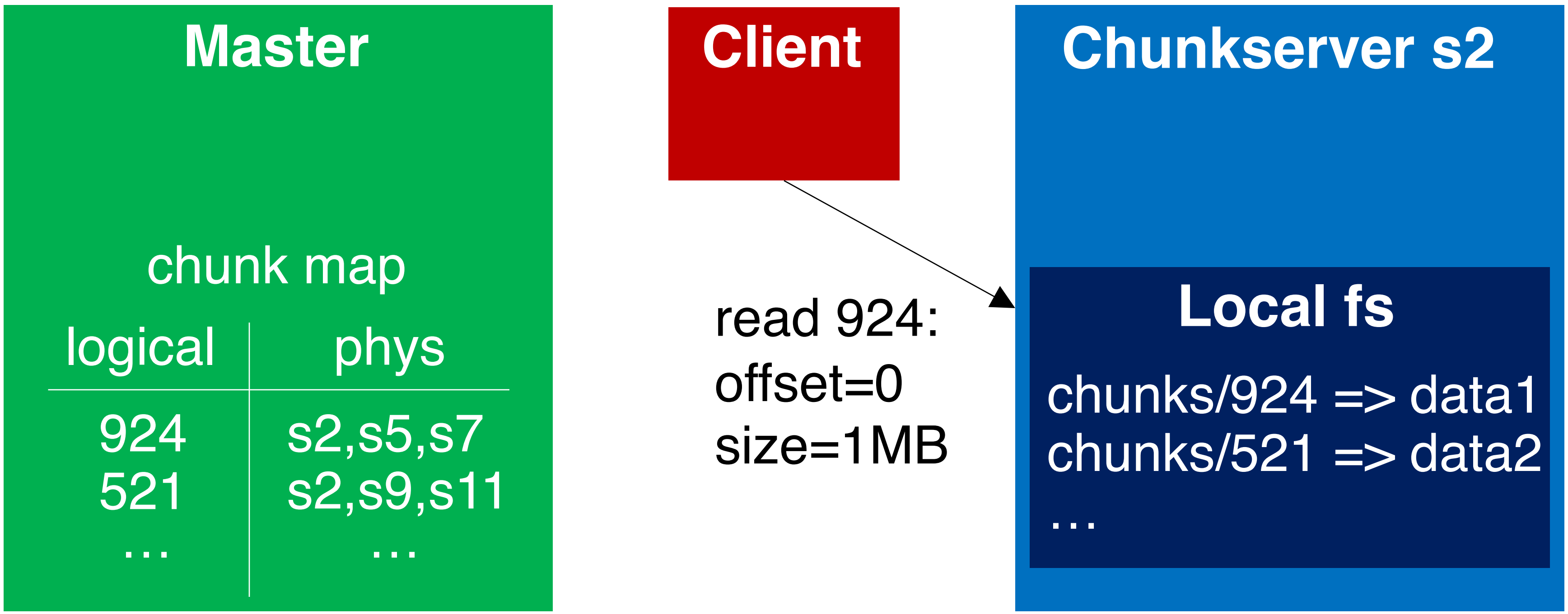


Client asks for the location

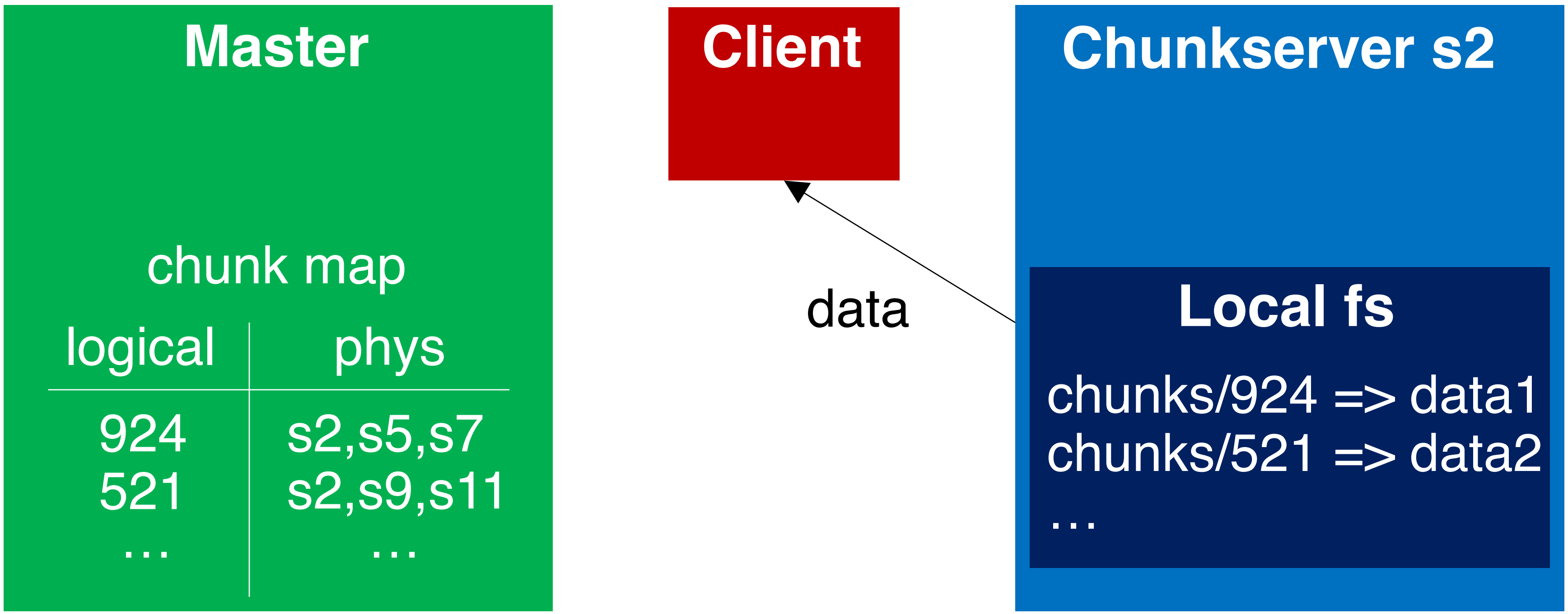


Client reads a chunk

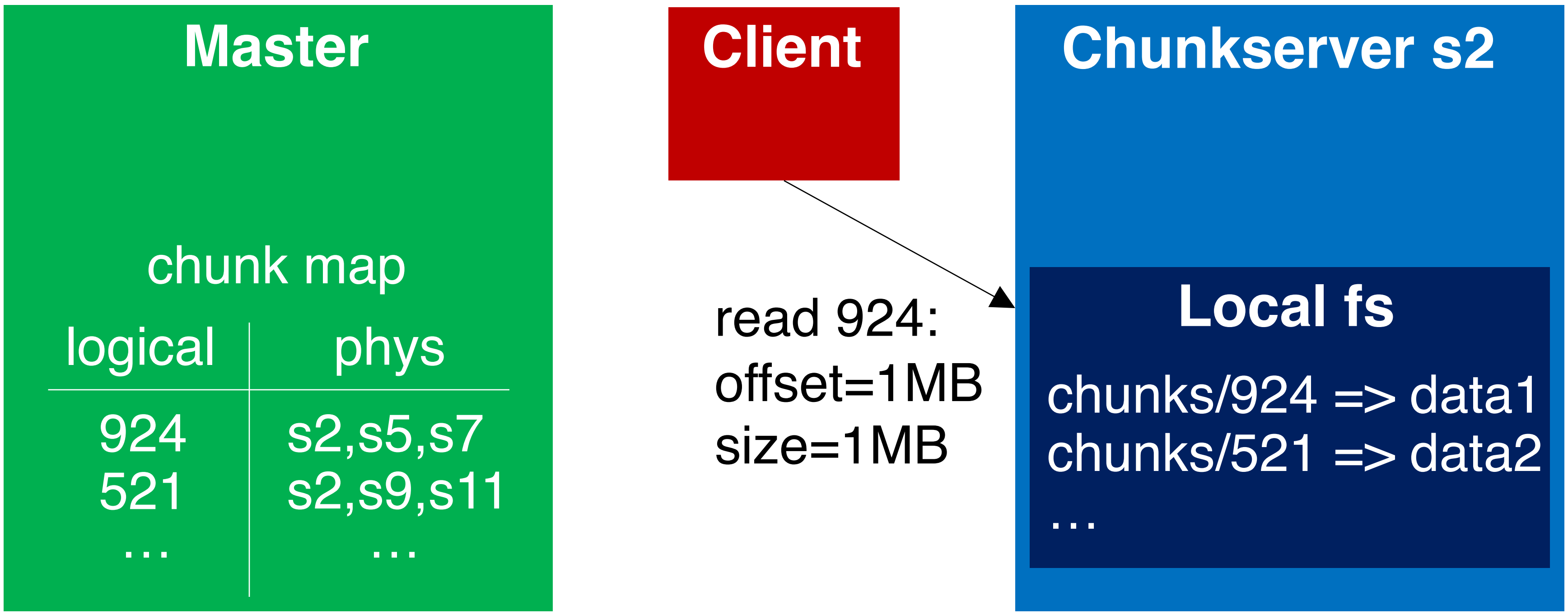
Client reads a chunk



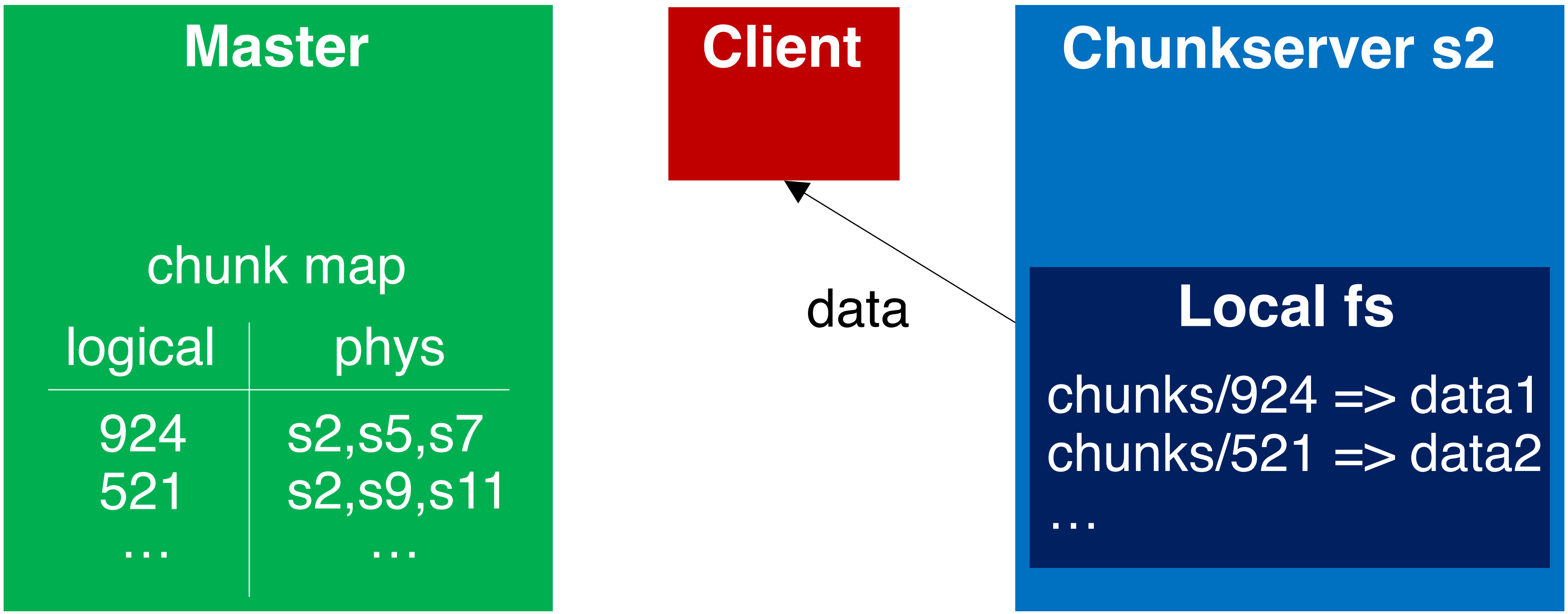
Client reads a chunk



Client reads a chunk



Client reads a chunk



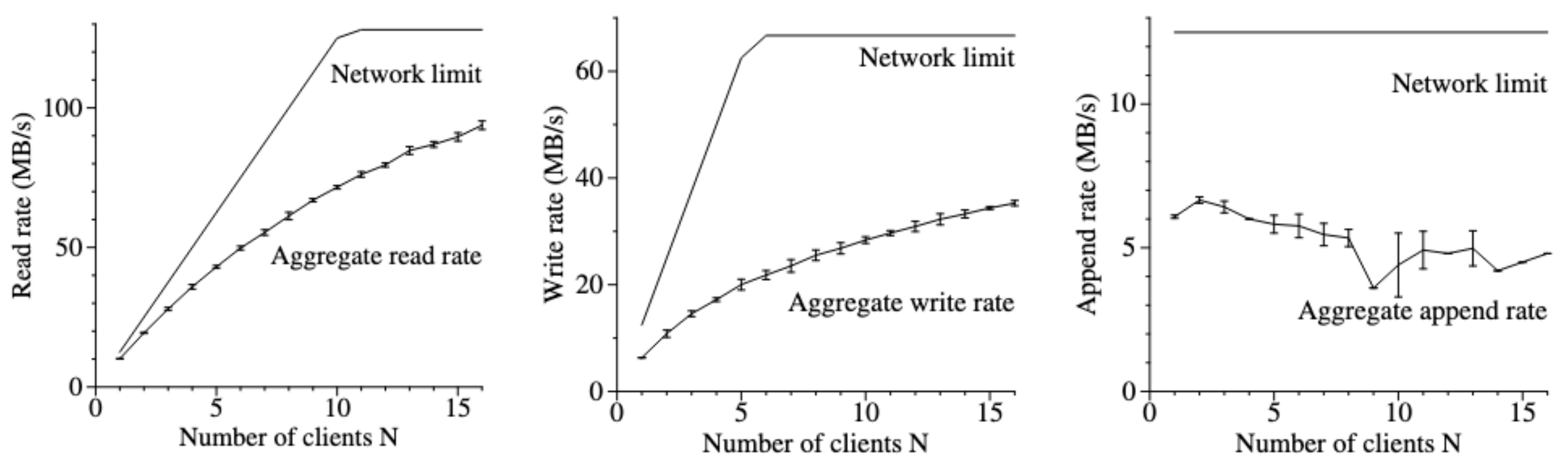
File namespace

path names mapped to logical names

Discussion

GFS evaluation

List your takeaways from “Figure 3: Aggregate Throughputs”



(a) Reads (b) Writes (c) Record appends

GFS scale

The evaluation in Table 2 shows clusters with up to 180 TB of data. What part of the design/configuration would need to change if we instead had **180 PB of data**?

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters