Slides: Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

ADVANCED TOPICS: GOOGLE FILE SYSTEM (GFS)

Questions answered in this lecture:

What are the **requirements** for GFS?

What techniques does GFS use to **scale**?

What is the role of the **master** vs. **chunkservers** in GFS?

What happens if the master or a chunkserver **crashes**?

How are replicas kept **consistent**?

GFS MOTIVATION

Measure then build

Google workload characteristics

- huge files (GBs); usually read in their entirety
- almost all writes are appends
- concurrent appends common
- high throughput is valuable
- low latency is not

Computing environment:

- 1000s of machines
- Machines sometimes fail (both permanently and temporarily)

Why not use NFS?

1. **Scalability : Must store > 100s of Terabytes of file data**

NFS only exports a local FS on one machine to other clients

GFS solution: store data on many server machines

2. **Failures: Must handle temporary and permanent failures**

NFS only recovers from temporary failure
- not permanent disk/server failure
- recovery means making reboot invisible
- technique: retry (stateless and idempotent protocol helps)

GFS solution: replication and failover (like RAID)

NEW FILE SYSTEM: GFS

Google published details in 2003

- Has evolved since then…

Open source implementation: Hadoop Distributed FS (HDFS)

Opportunity for Co-Design

Do not need general-purpose file system

- Does not need to be backwards-compatible with existing applications
- Does not need to adhere to POSIX specification

Opportunity to build FS and application together

- Make sure applications can deal with FS quirks

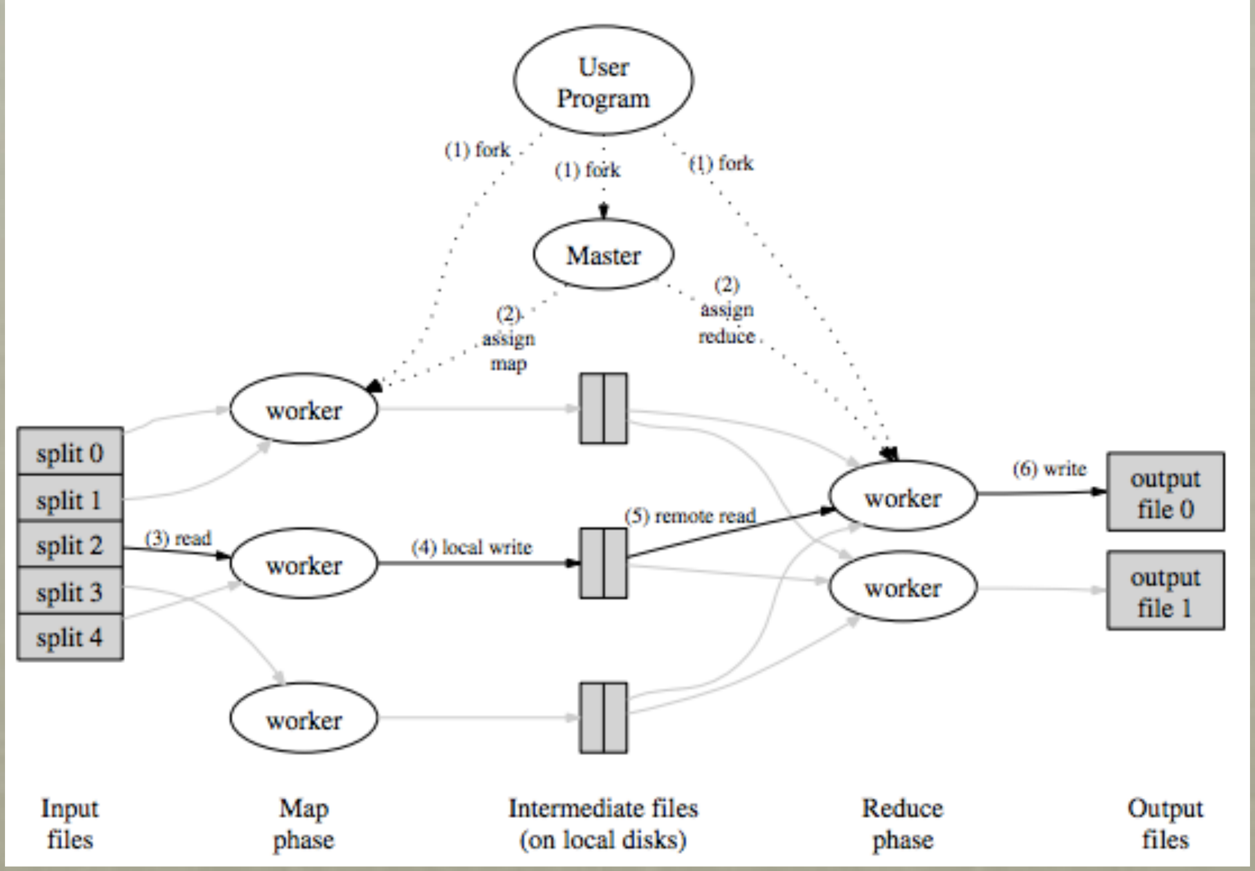
Avoid difficult FS features:

- Read directory (make new directory structure)
- Links
- Reading from an open, deleted file

What Workloads?

MapReduce (previous lecture)

- **read entire input file (in chunks)**
- compute over data
- **append** to separate output files



Producer/consumer

- many producers **append** work to shared file concurrently
- one consumer reads and does work and **appends** to output file

How to handle appends that are **not** idempotent?

- Require applications to handle duplicate records in data
- Add unique identifiers to records

GFS Overview

~~Motivation~~

Architecture

Master metadata

Chunkserver data

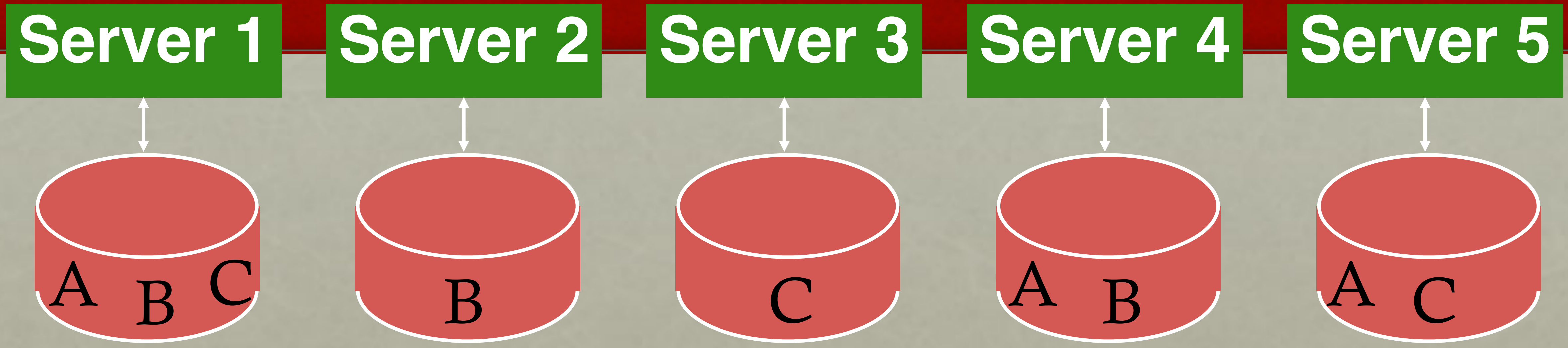
MACHINES FAIL

Fact: Machines storing data may fail

Implications for GFS

- Must **replicate** data (similar to RAID)
- Must **recover** (respond to machines stopping at starting)

1) REPLICATION



Each server holds “chunks” of data

Less structured than RAID (no static computation to determine locations)

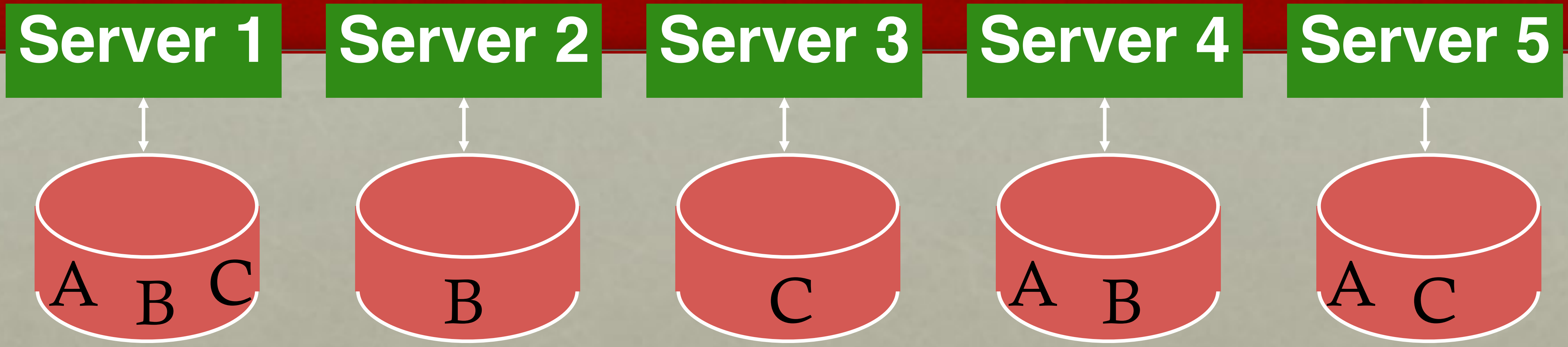
- machines come and go

- capacity may vary

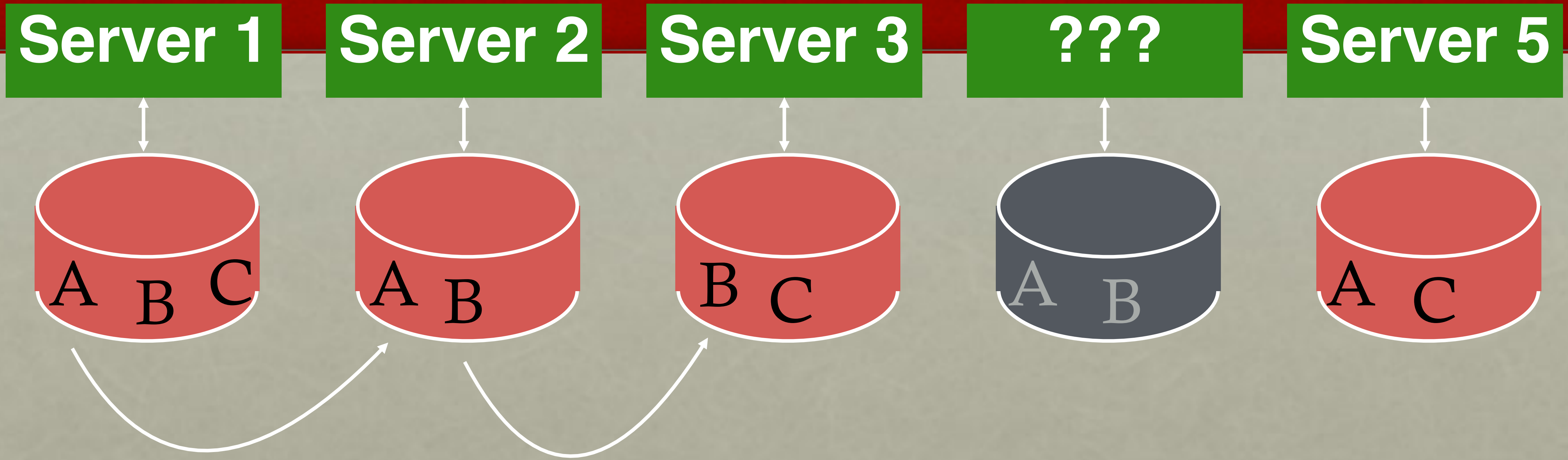
- different data may have different replication levels (e.g., 3 vs 5 copies)

Problem: How to map logical to physical locations?

2) Recovery



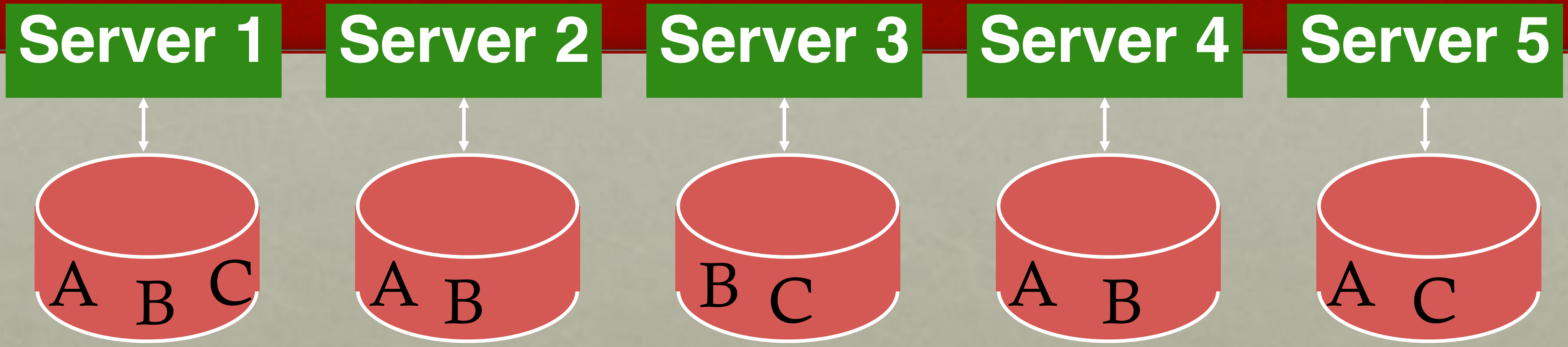
2) Recovery



Machine may come back, or may be dead forever

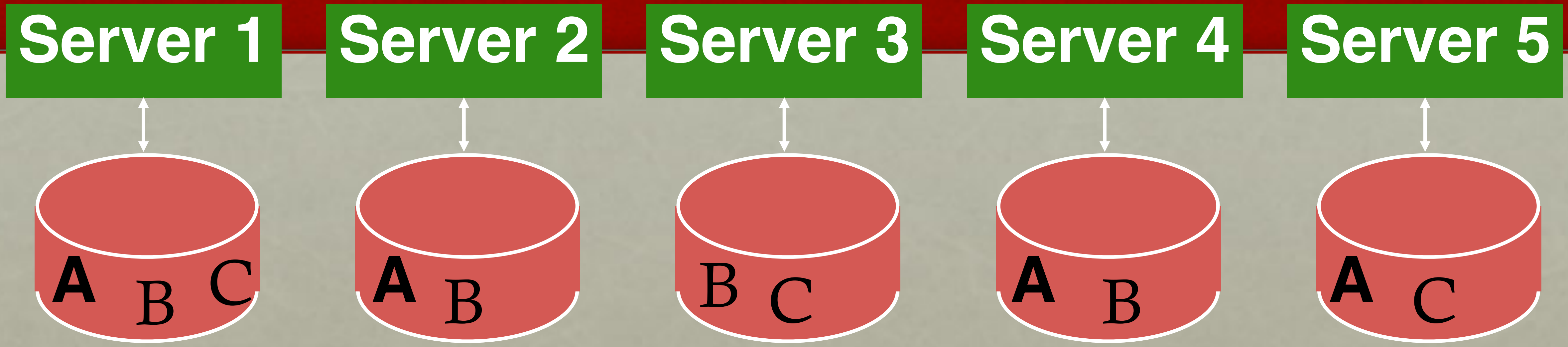
Must identify and replicate lost data on other servers

2) RECOVERY



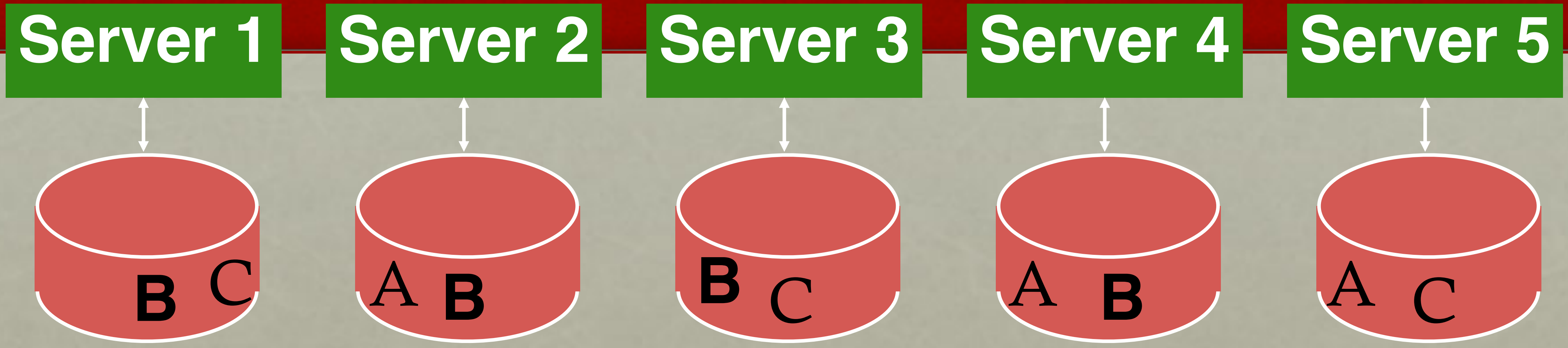
Machine may come back; disk space wasted with extra replicas

2) RECOVERY



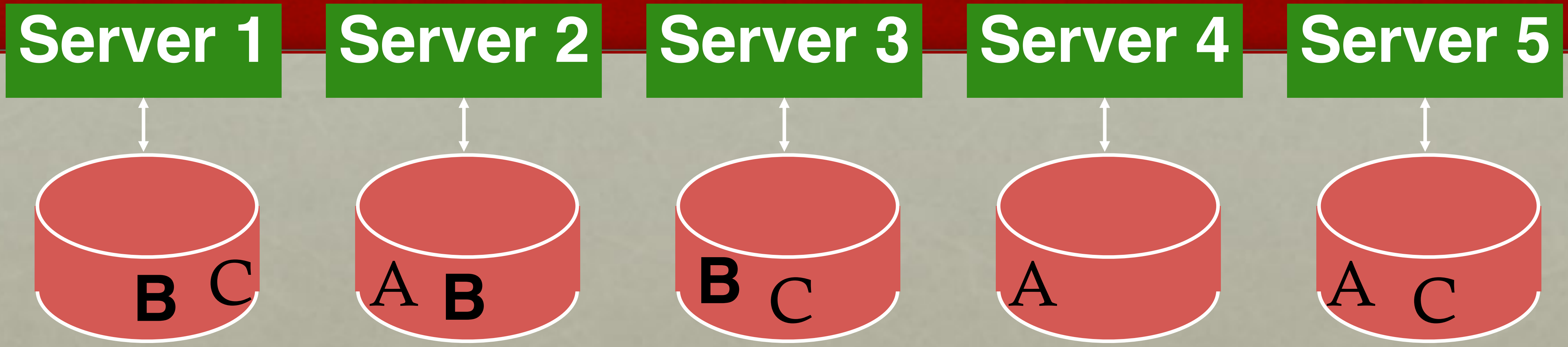
Identify number of replicas and choose to remove extras

2) RECOVERY



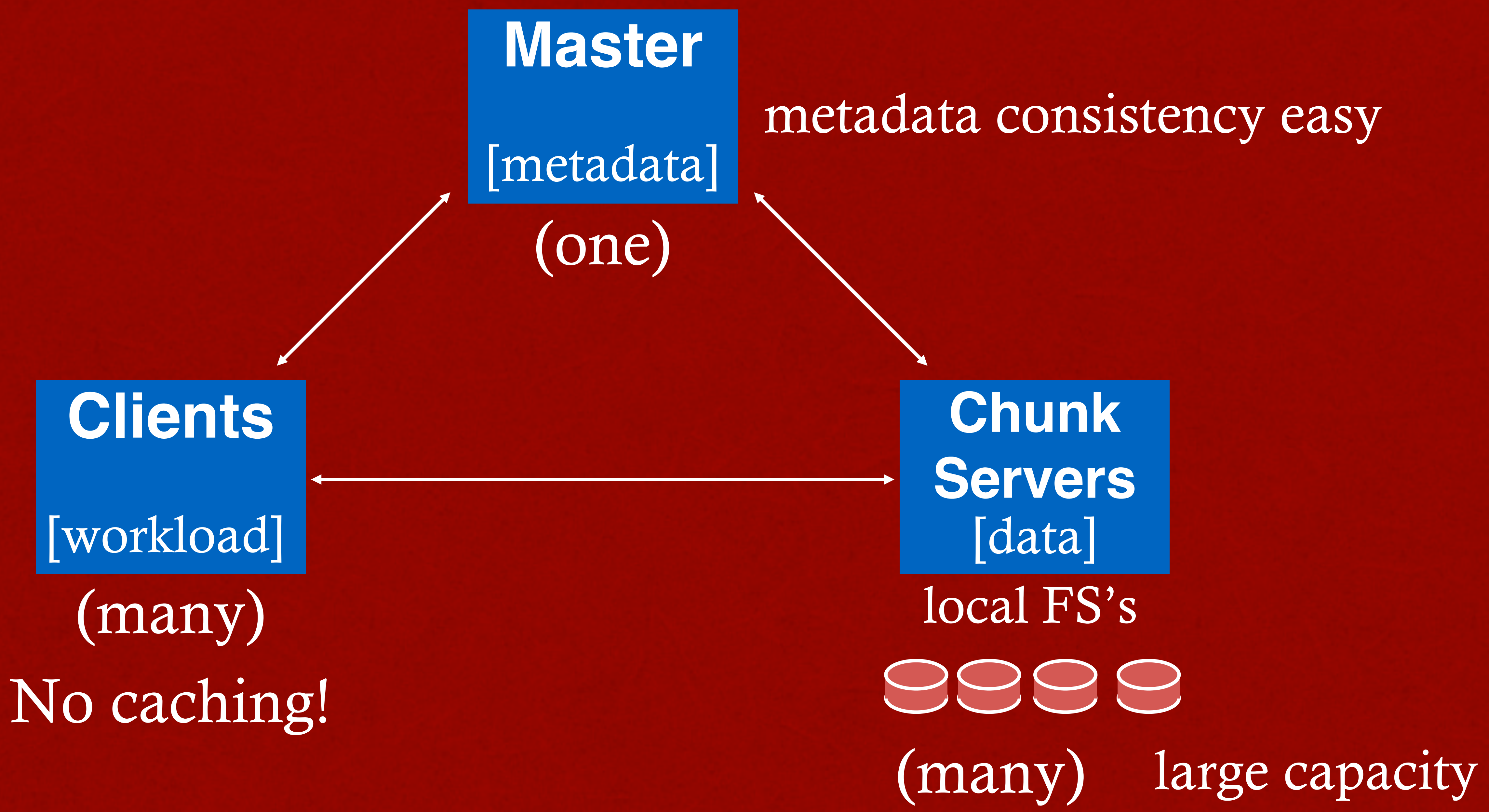
Identify number of replicas and choose to remove extras

OBSERVATION



Finding copies of data + maintaining replicas is difficult without **global view** of data

GFS Architecture



Master

[metadata]

(one)

metadata consistency easy

Clients

[workload]

(many)

No caching!

Chunk Servers

[data]

local FS's

(many)

large capacity

What is a chunk?

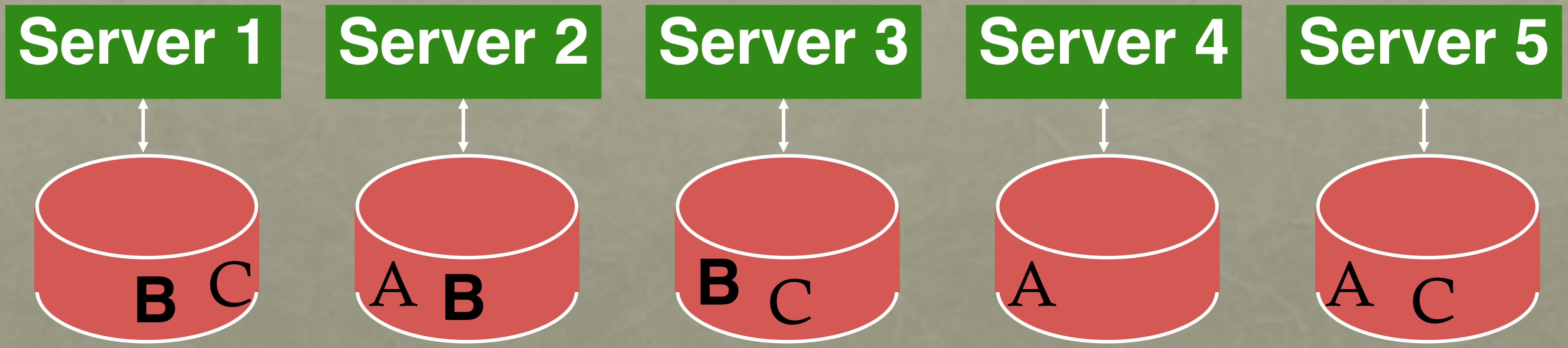
Break GFS files into large chunks (e.g., 64MB);
unit of replication; chunks not split across chunkservers

Why this size?

- Match chunk size to input size for each mapper in MapReduce

Chunk servers store physical chunks in Linux files

Master maps logical chunk to physical chunk locations



GFS OVERVIEW

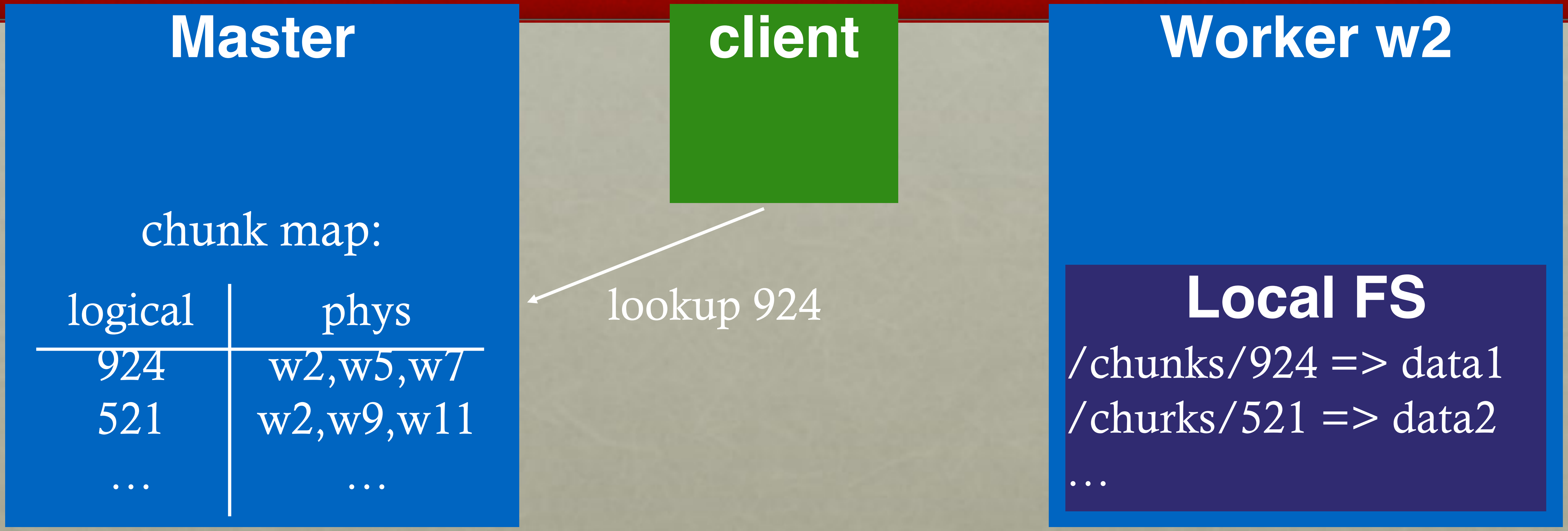
~~Motivation~~

~~Architecture~~

Master metadata

Chunkserver data

MASTER METADATA



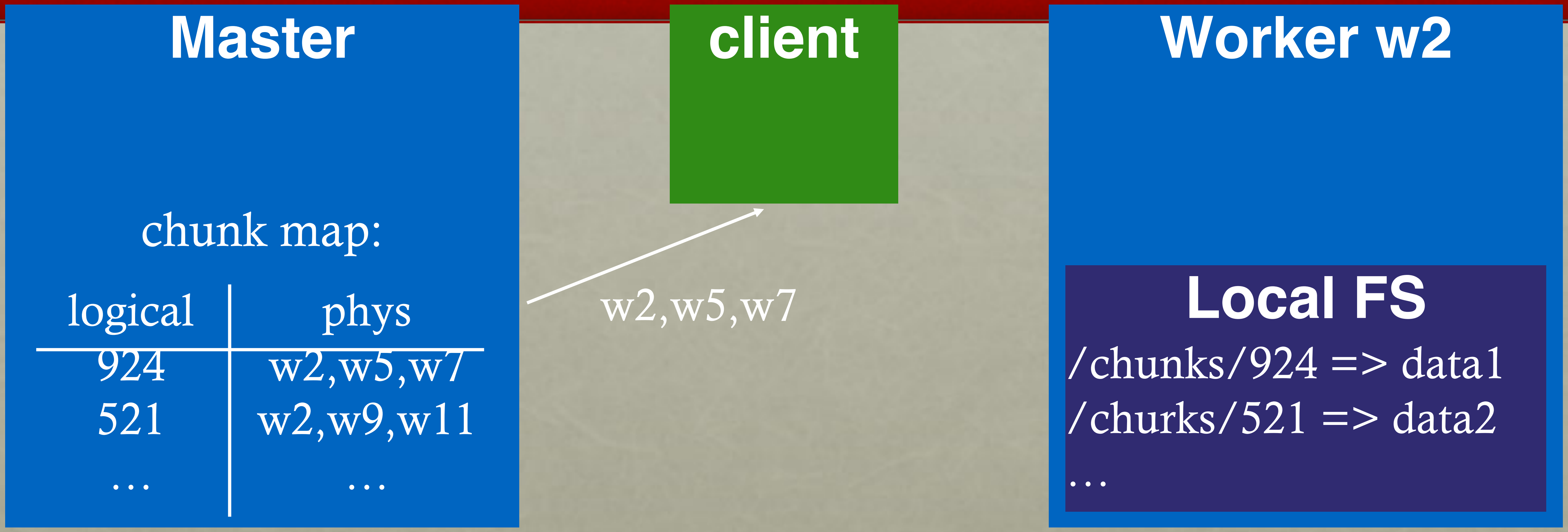
chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

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

Client wants to read a chunk (identified with unique id num)
How does it find where that chunk lives?

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

w2,w5,w7

client

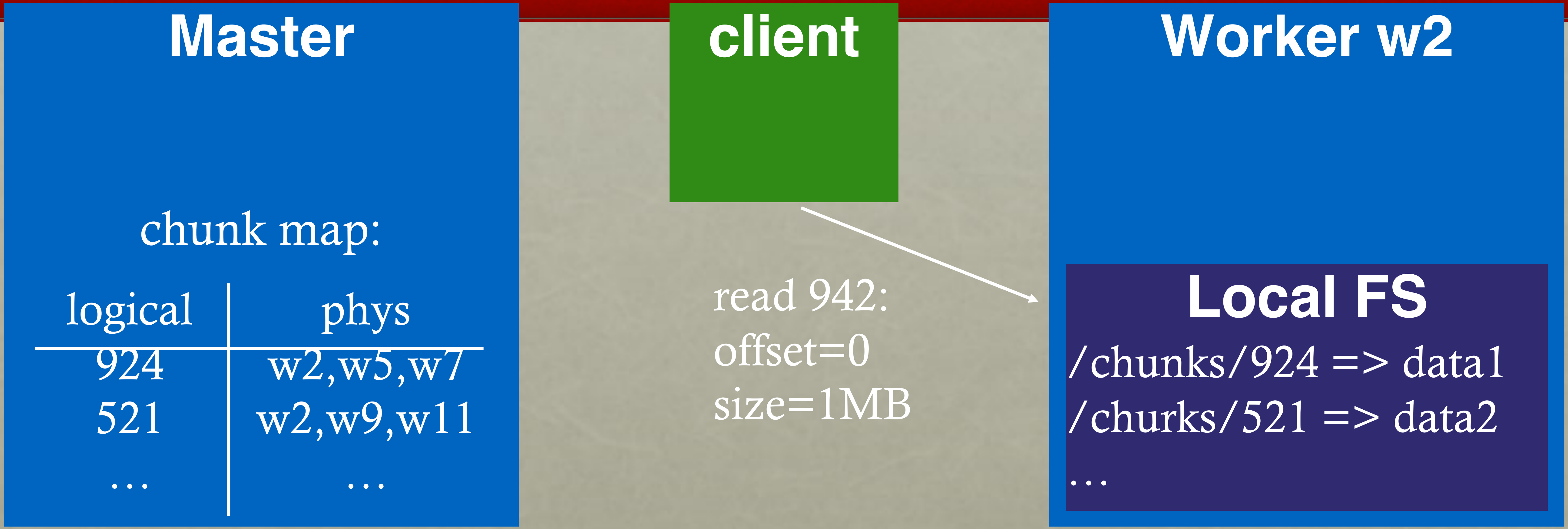
Worker w2

Local FS

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

Client can read from any of the listed replicas

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

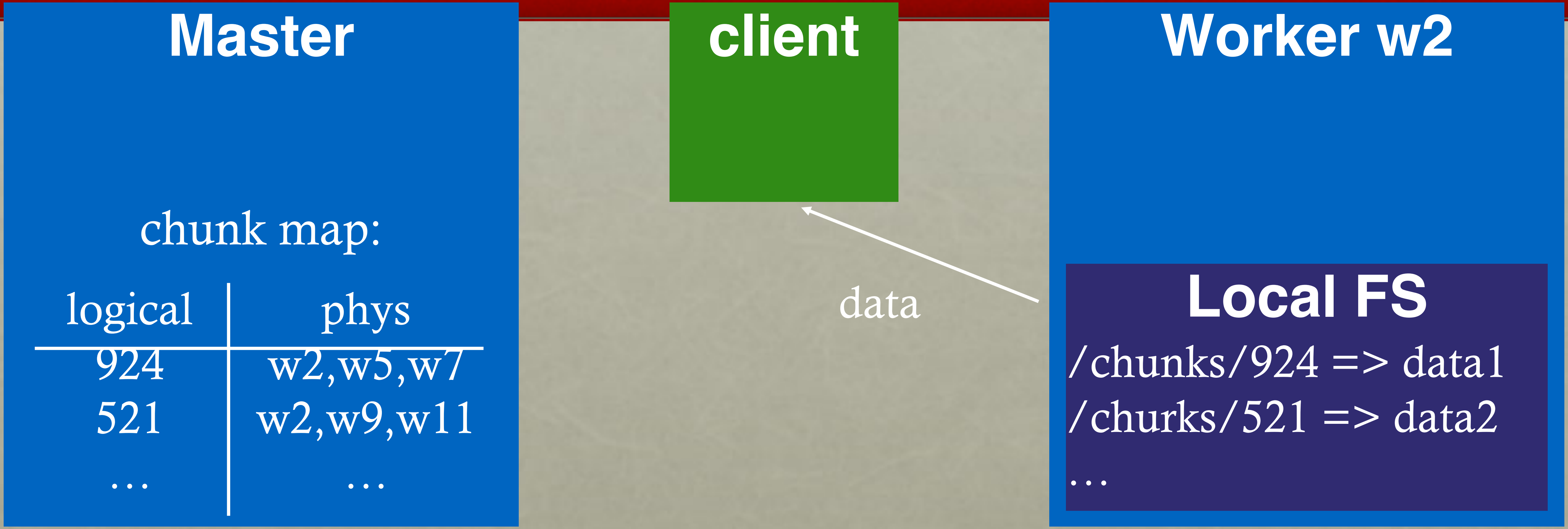
read 942:
offset=0
size=1MB

Worker w2

Local FS

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

Client Reads a Chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

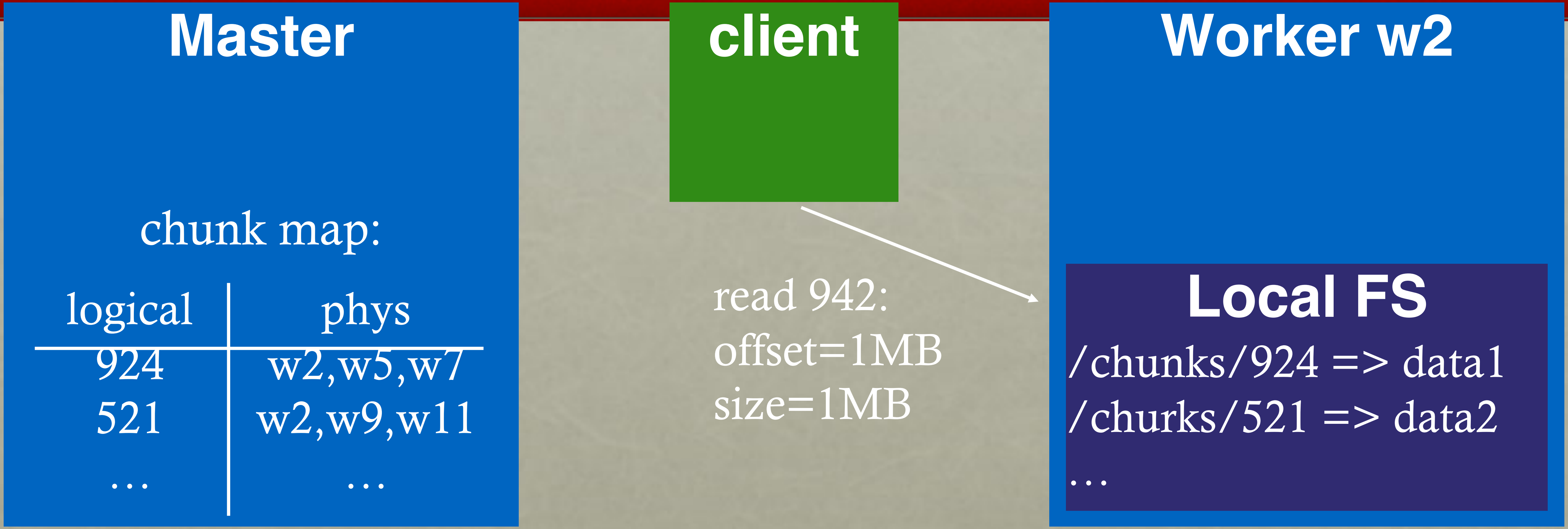
client

Worker w2

Local FS

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

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

read 942:
offset=1MB
size=1MB

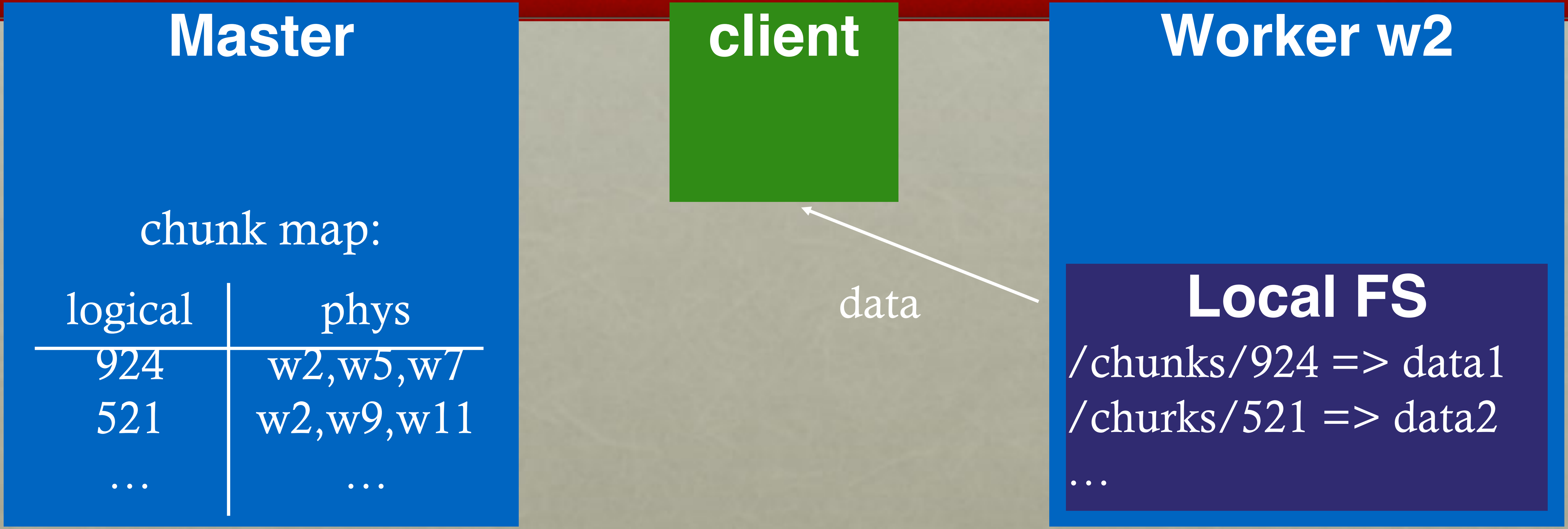
Worker w2

Local FS

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

Client tracks current offset of read within each 64MB chunk

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

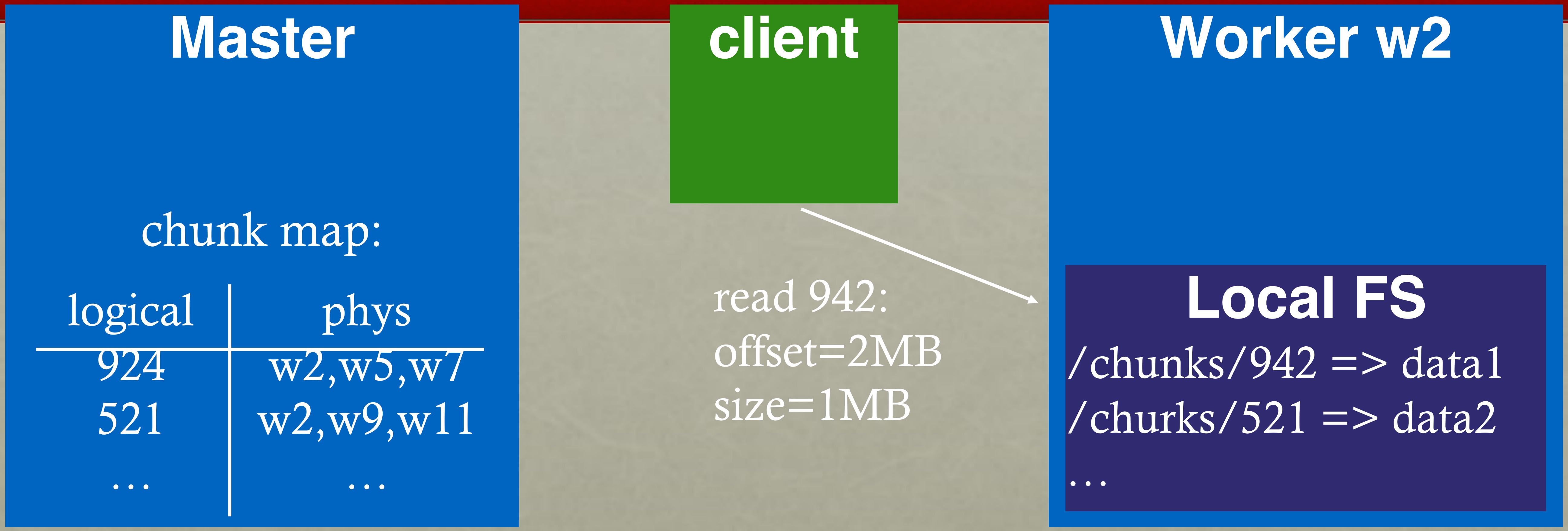
data

Worker w2

Local FS

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

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

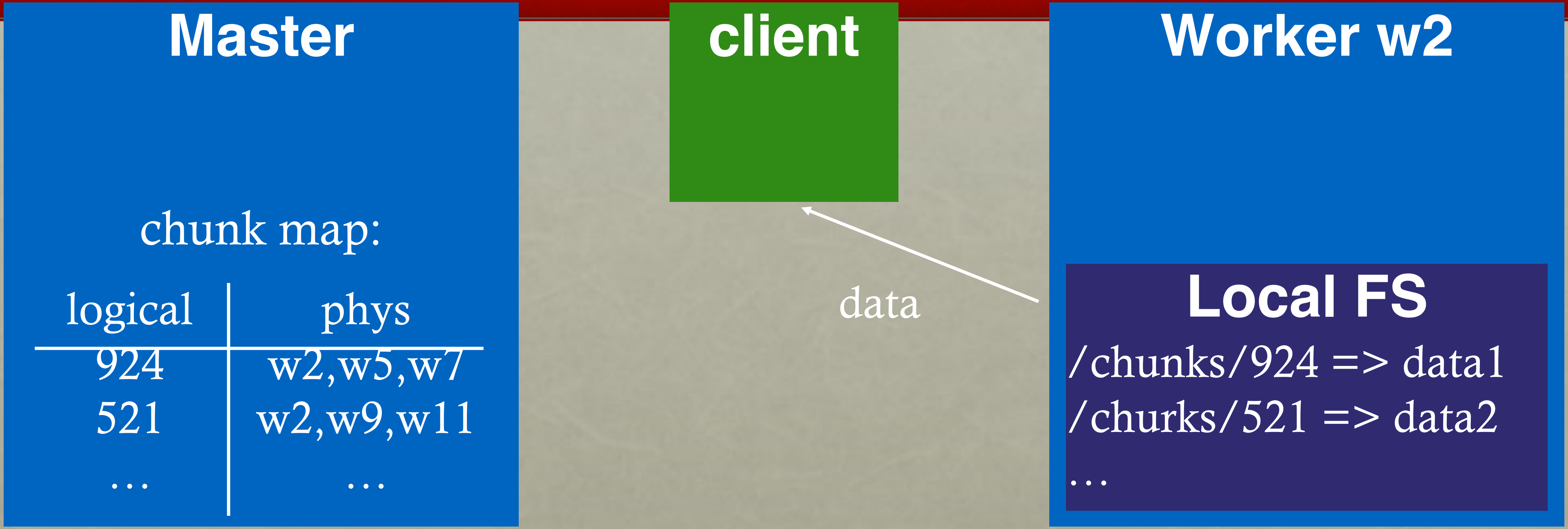
read 942:
offset=2MB
size=1MB

Worker w2

Local FS

/chunks/942 => data1
/churks/521 => data2
…

Client reads a chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

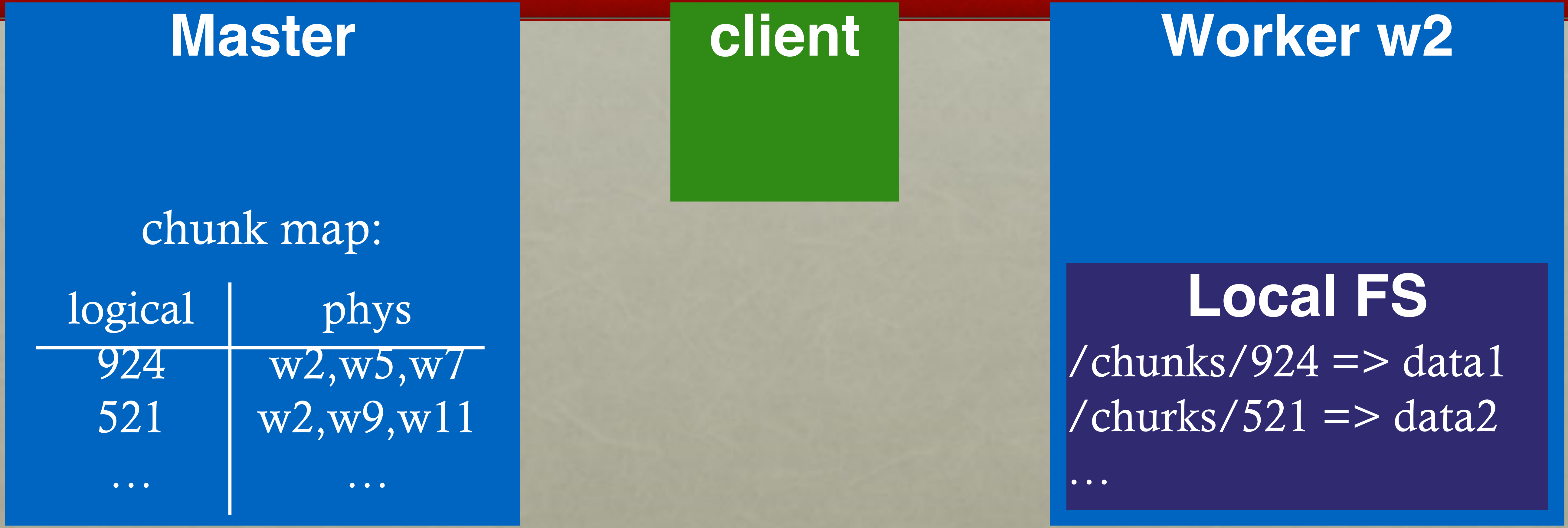
data

Worker w2

Local FS

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

Client Reads a Chunk



Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

Worker w2

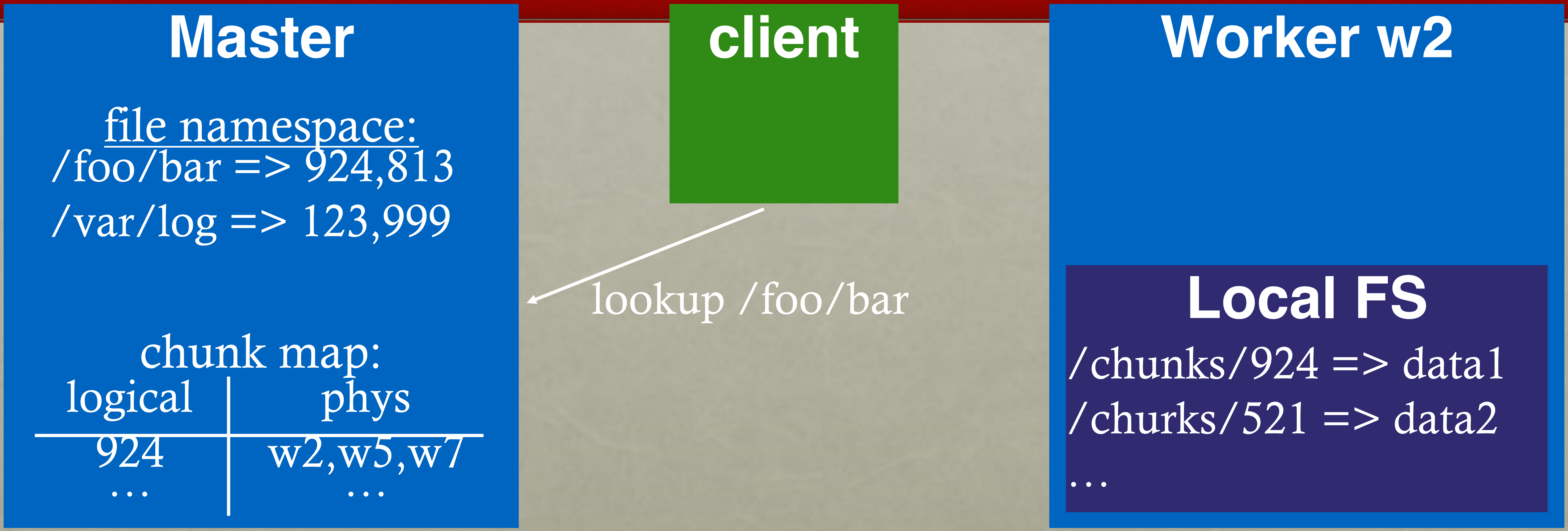
Local FS

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

Master is **not bottleneck** because not involved in most reads
1 master can handle many clients…

How does client know what chunk id num to read?

File Namespace



Master

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

chunk map:

logical	phys
924	w2,w5,w7
…	…

client

lookup /foo/bar

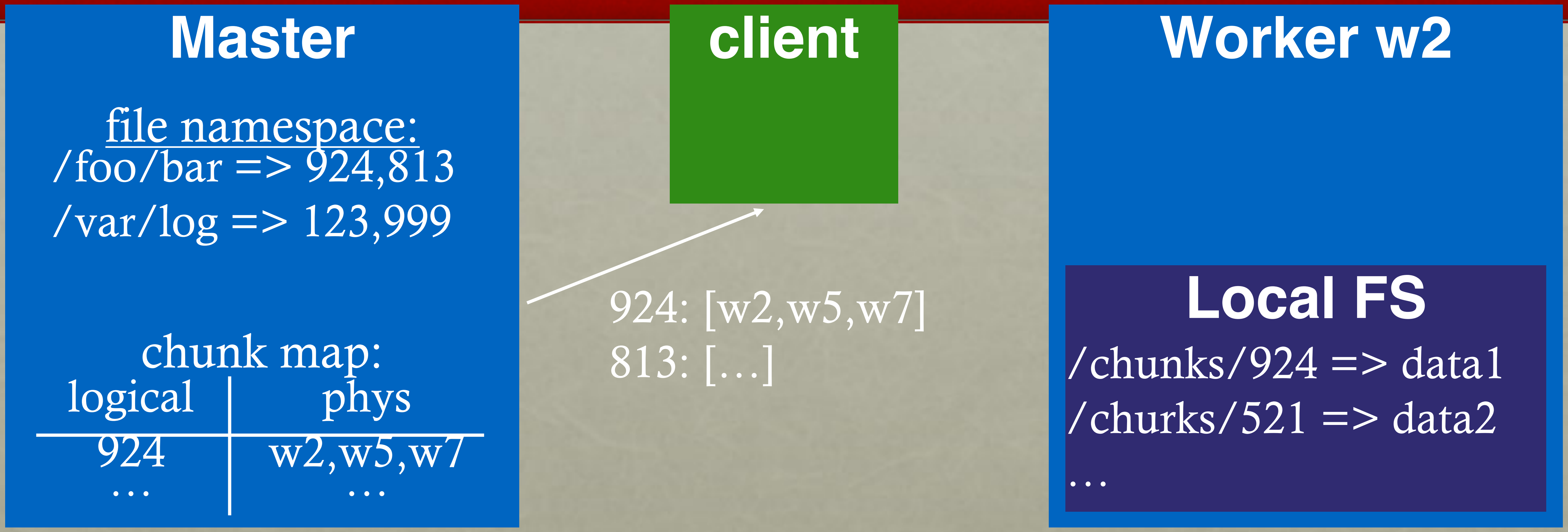
Worker w2

Local FS
/chunks/924 => data1
/churks/521 => data2
…

Master maps **path name** to **logical chunk list**
(expect many chunks per file)

1. Client sends path name to master
2. Master sends chunk locations to client
3. Client reads/writes to workers directly

File Namespace



Master

file namespace:

/foo/bar => 924,813

/var/log => 123,999

chunk map:

logical	phys
924	w2,w5,w7
…	…

client

924: [w2,w5,w7]

813: […]

Worker w2

Local FS

/chunks/924 => data1

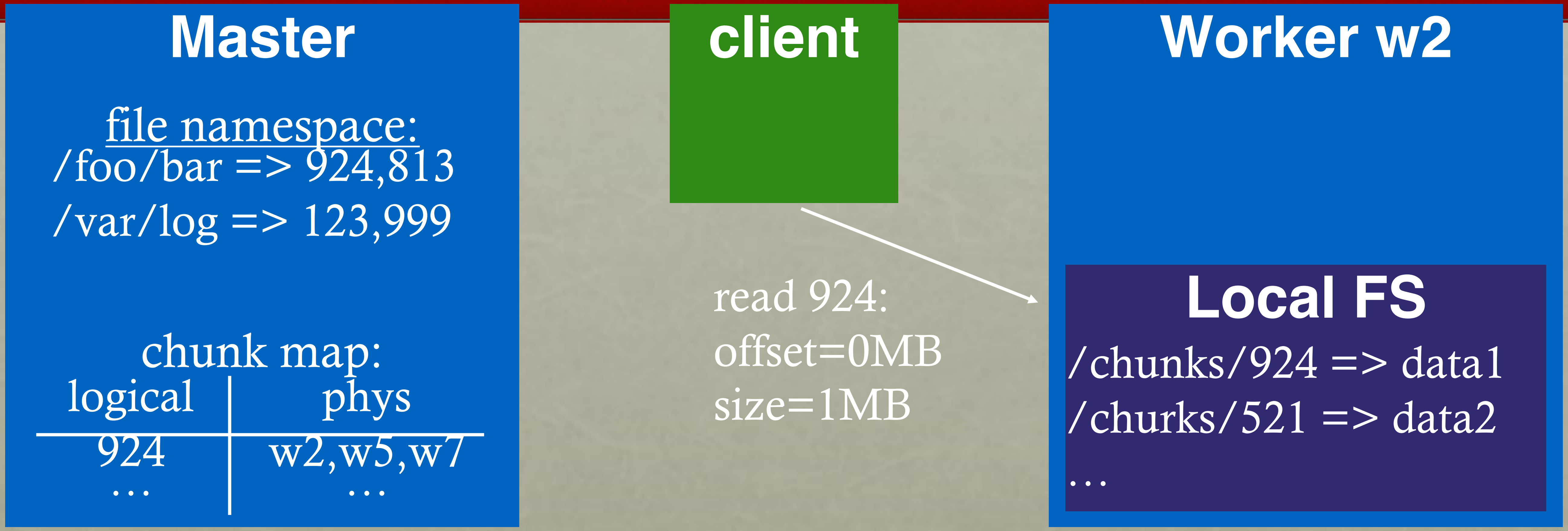
/churks/521 => data2

…

Master maps **path name** to **logical chunk list**

1. Client sends path name to master
2. Master sends chunk locations to client
3. Client reads/writes to workers directly

File Namespace



Master

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

chunk map:

logical	phys
924	w2,w5,w7
…	…

client

read 924:
offset=0MB
size=1MB

Worker w2

Local FS

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

How to pick chunk size?

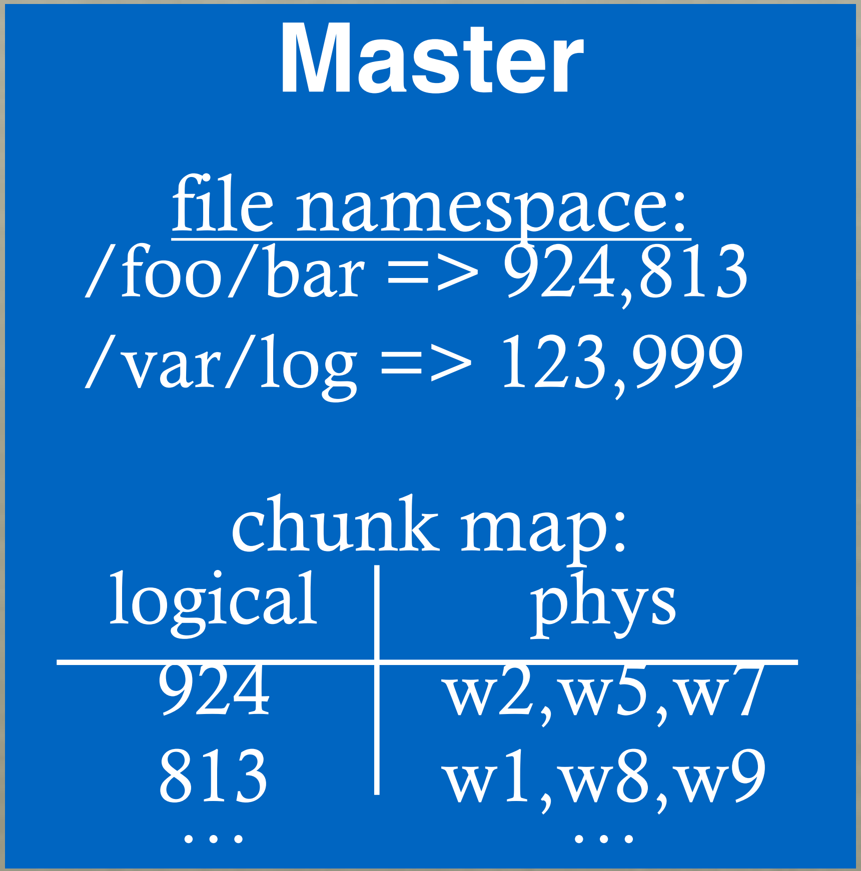
GFS uses large chunks, e.g., 64MB
(coordinate with MapReduce)

How does chunk size affect size of master data structs?

What if Chunk Size Doubles?

Any disadvantages to making chunks huge?

Cannot parallelize I/O as much



Master

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

chunk map:

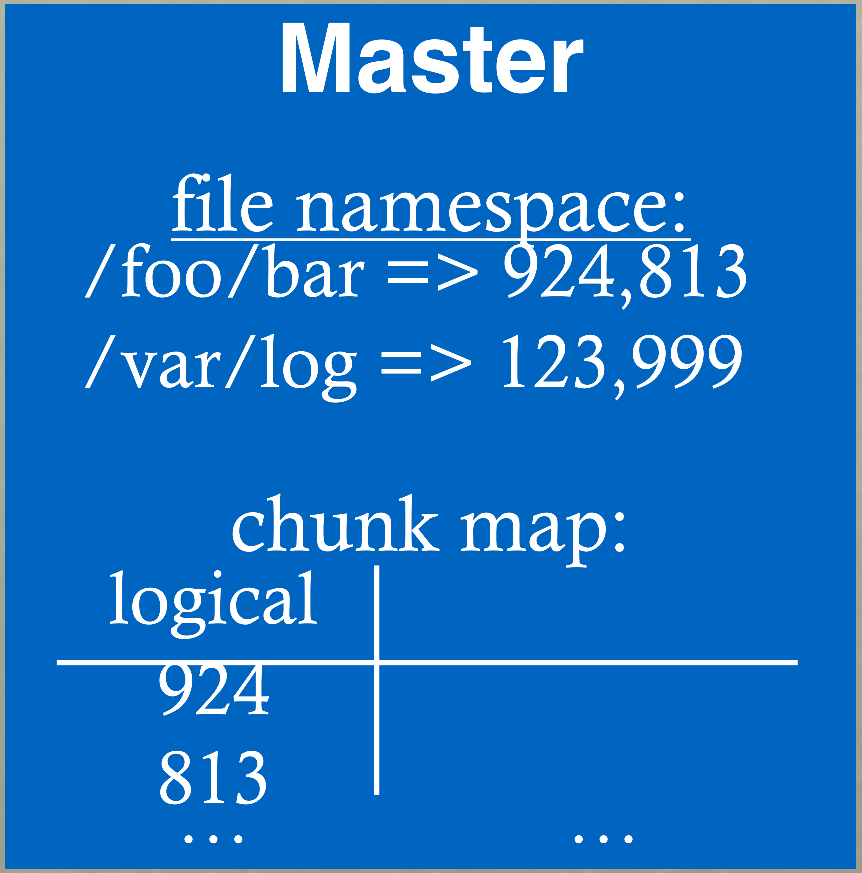
logical	phys
924	w2,w5,w7
813	w1,w8,w9
…	…

lists half as long

half as many entries

MASTER:
CRASHES + CONSISTENCY

Advantage to minimizing master data structures:



Master

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

chunk map:

logical	
924	
813	
…	…

File **namespace** and **chunk map** fit 100% in RAM

- Advantage?
  - Fast (Allows master to keep up with 1000's of workers)
- Disadvantage?
  - Limits size of namespace to what fits in RAM
  - What if master crashes?

How to handle master crashing

Two data structures to worry about

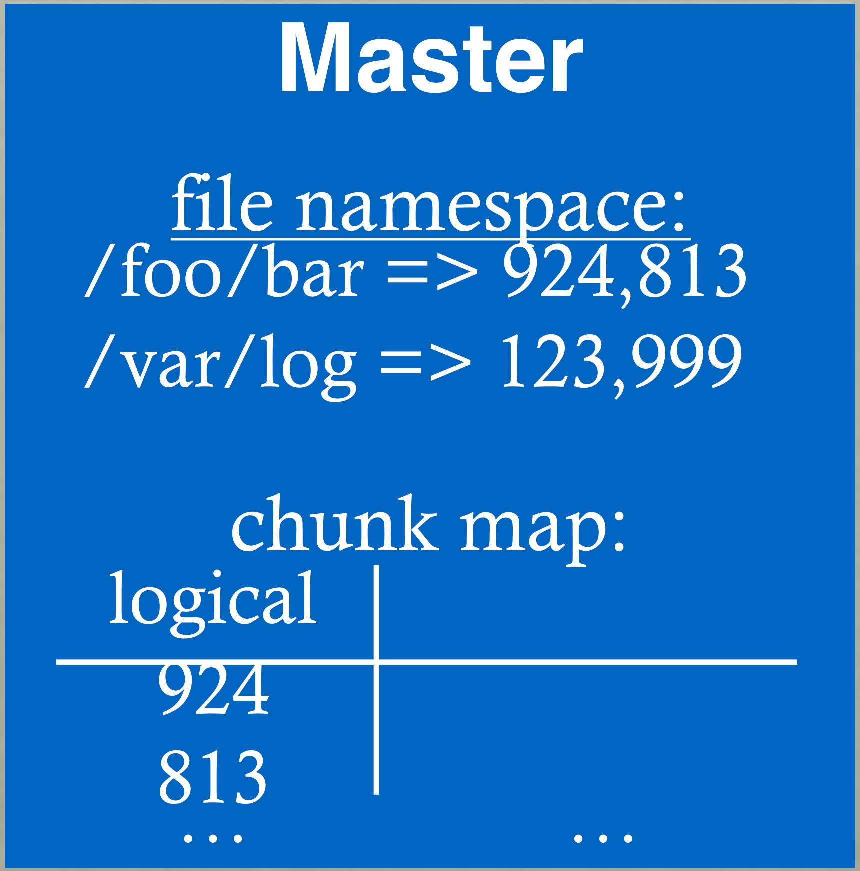
How to make **namespace** persistent?
Write updates to namespace to multiple logs

Where should these logs be located?

- Local disk (disk is never read except for crash)
- Disks on backup masters
- Shadow read-only masters (may lag state, temporary access)

Result: High availability when master crashes!

What about **chunk map**?



Master

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

chunk map:

logical	
924	
813	
…	…

Chunk Map Consistency

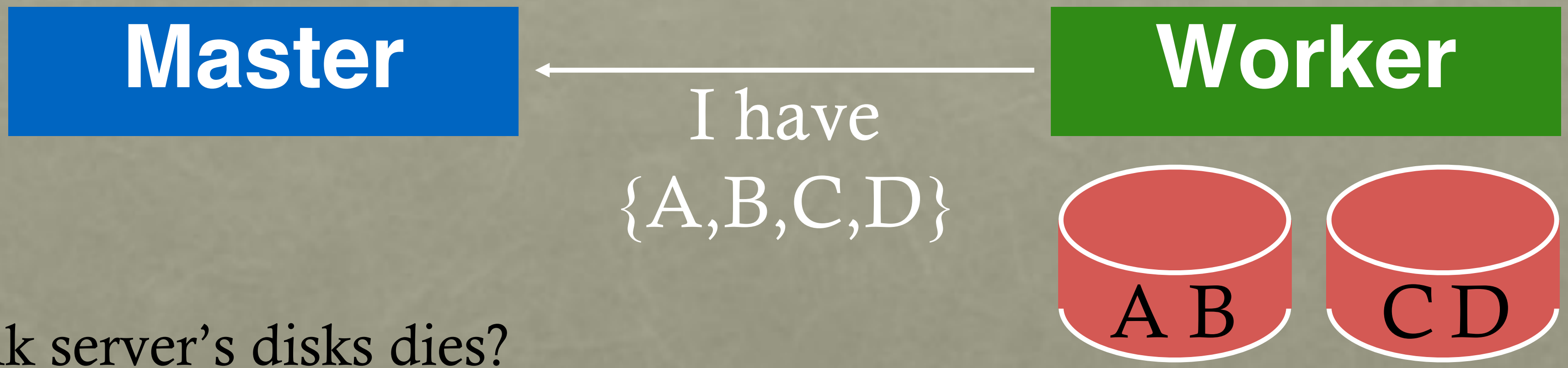
Don’t persist chunk map on master

Approach:

- After crash (and periodically for cleanup), master asks each chunkserver which chunks it has

What if chunk server dies too?

Doesn’t matter, that worker can’t serve chunks anyway



What if one of chunk server’s disks dies?

GFS Overview

~~Motivation~~

~~Architecture~~

~~Master metadata~~

Chunkserver data

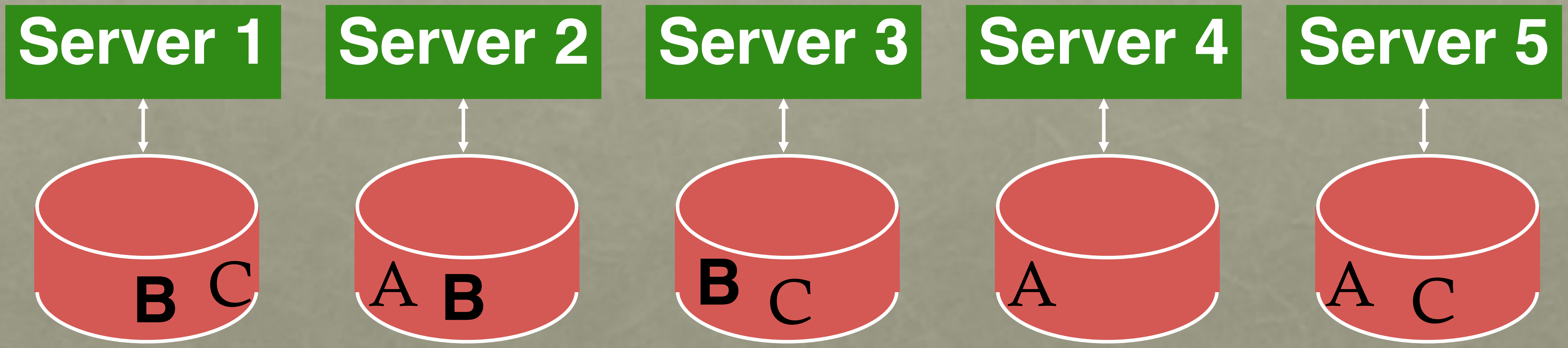
Chunkserver Consistency

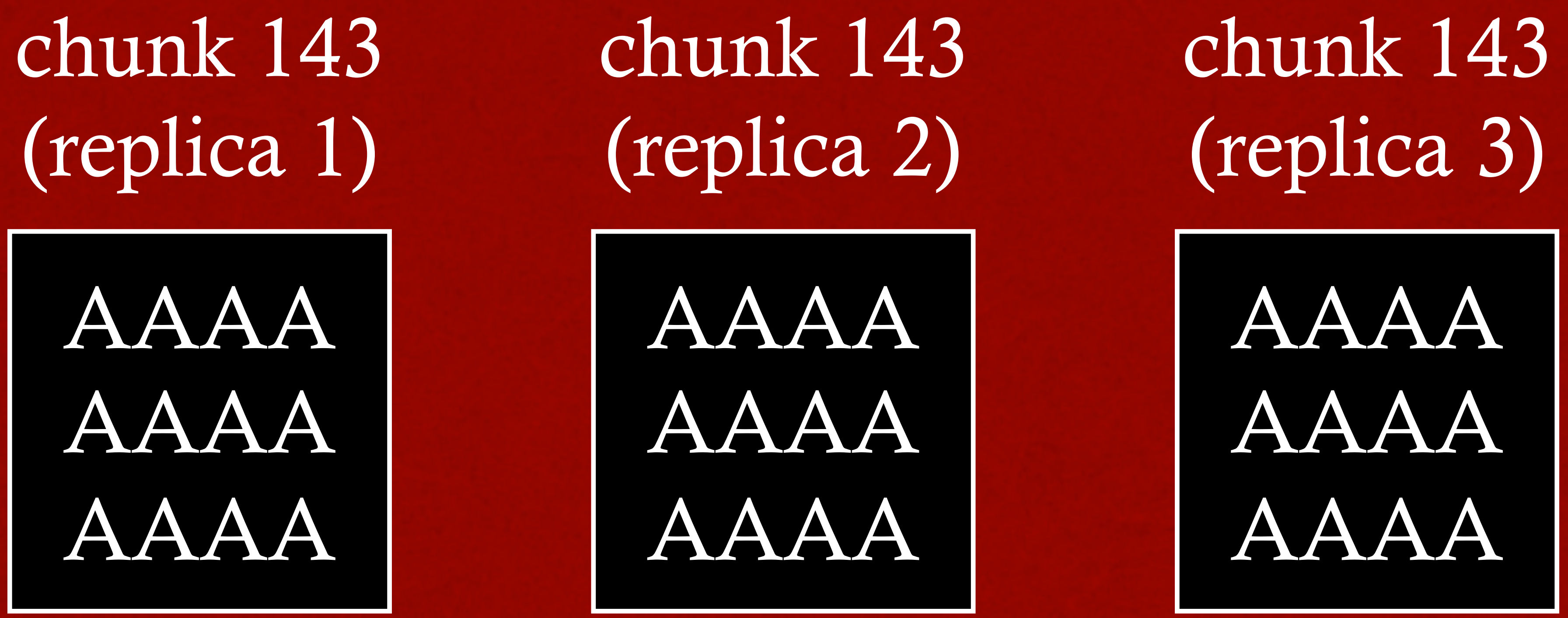
How does GFS ensure physical chunks on different chunkservers are consistent with one another?

Corruption: delete chunks that violate **checksum**

- Master eventually sees chunk has < desired replication

What about concurrent writes (or appends) from different clients? (e.g., multiple producers)





chunk 143
(replica 1)

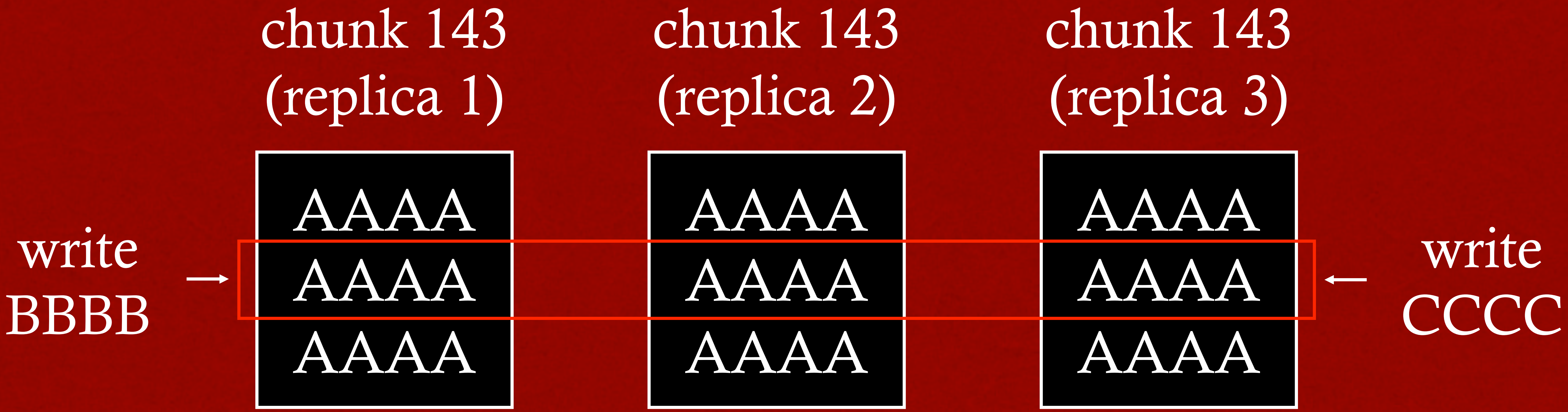
AAAA
AAAA
AAAA

chunk 143
(replica 2)

AAAA
AAAA
AAAA

chunk 143
(replica 3)

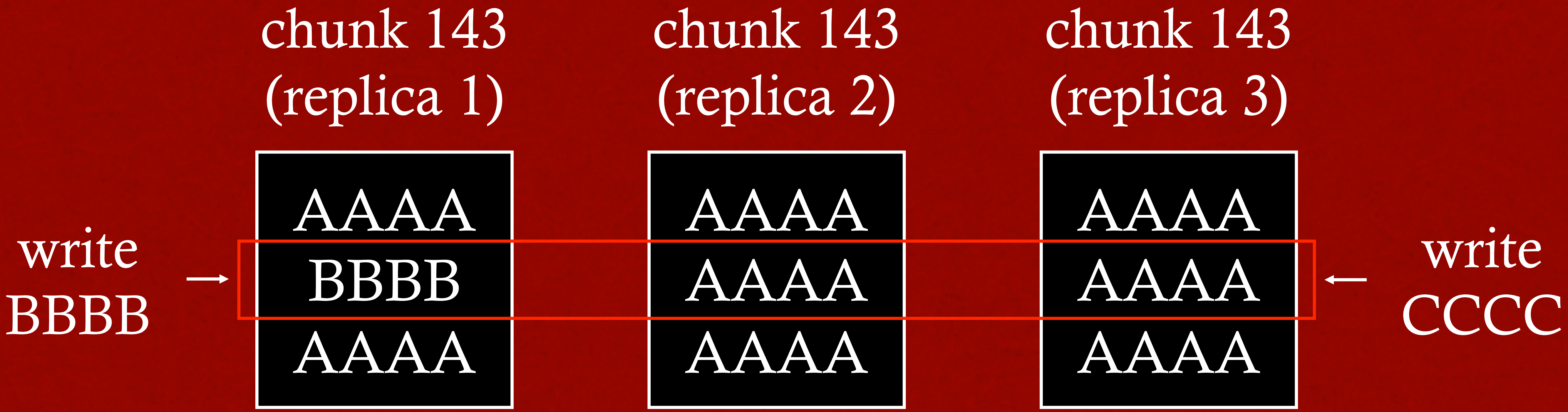
AAAA
AAAA
AAAA



chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA	AAAA	AAAA
AAAA	AAAA	AAAA
AAAA	AAAA	AAAA

write BBBB →

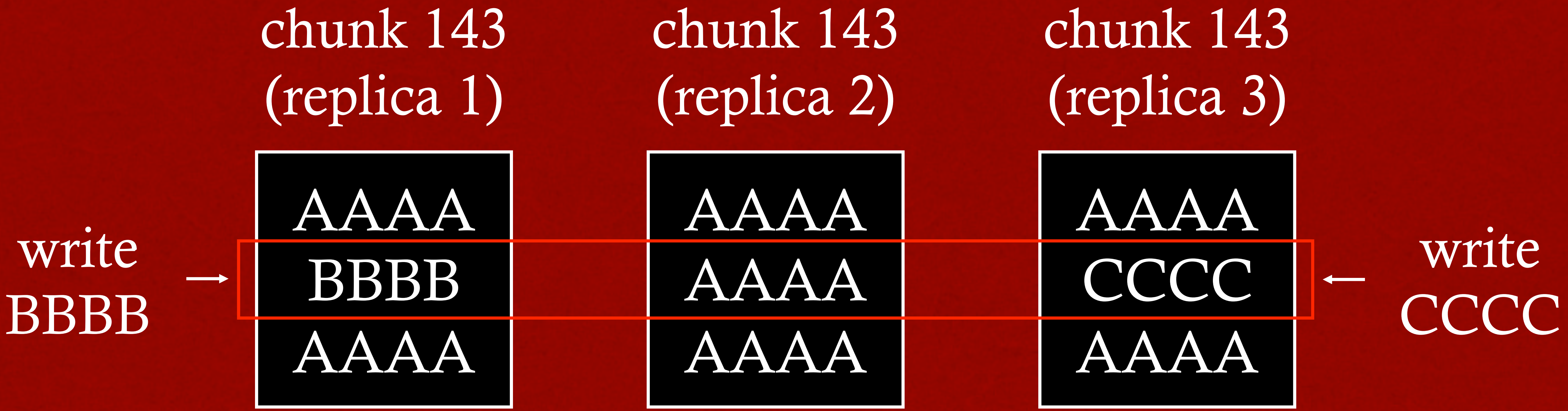
← write CCCC



chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA	AAAA	AAAA
BBBB	AAAA	AAAA
AAAA	AAAA	AAAA

write BBBB →

← write CCCC



chunk 143
(replica 1)

chunk 143
(replica 2)

chunk 143
(replica 3)

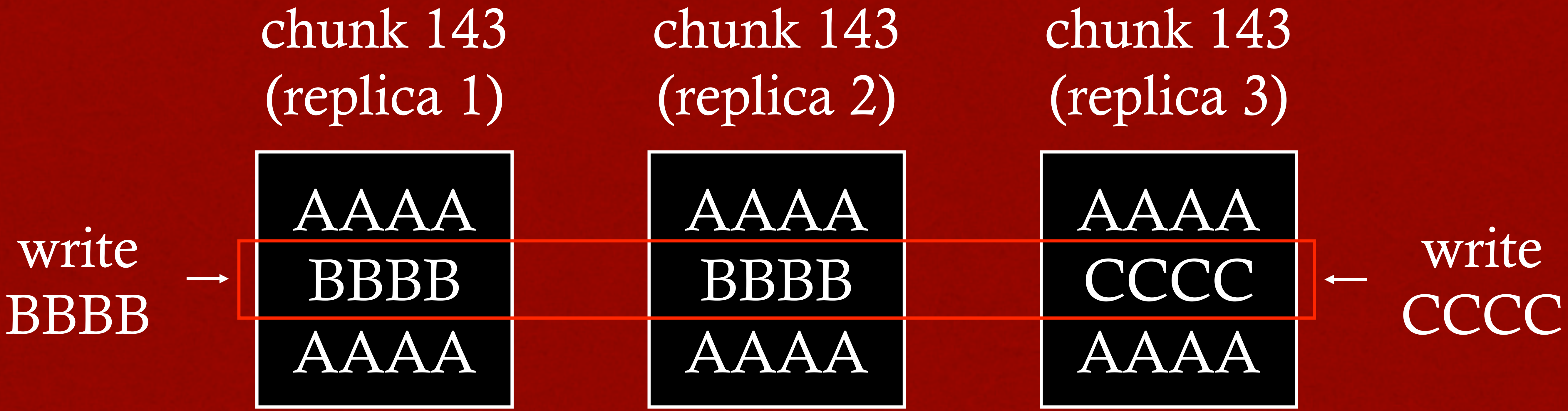
AAAA
BBBB
AAAA

AAAA
AAAA
AAAA

AAAA
CCCC
AAAA

write
BBBB

write
CCCC



chunk 143
(replica 1)

AAAA
BBBB
AAAA

chunk 143
(replica 2)

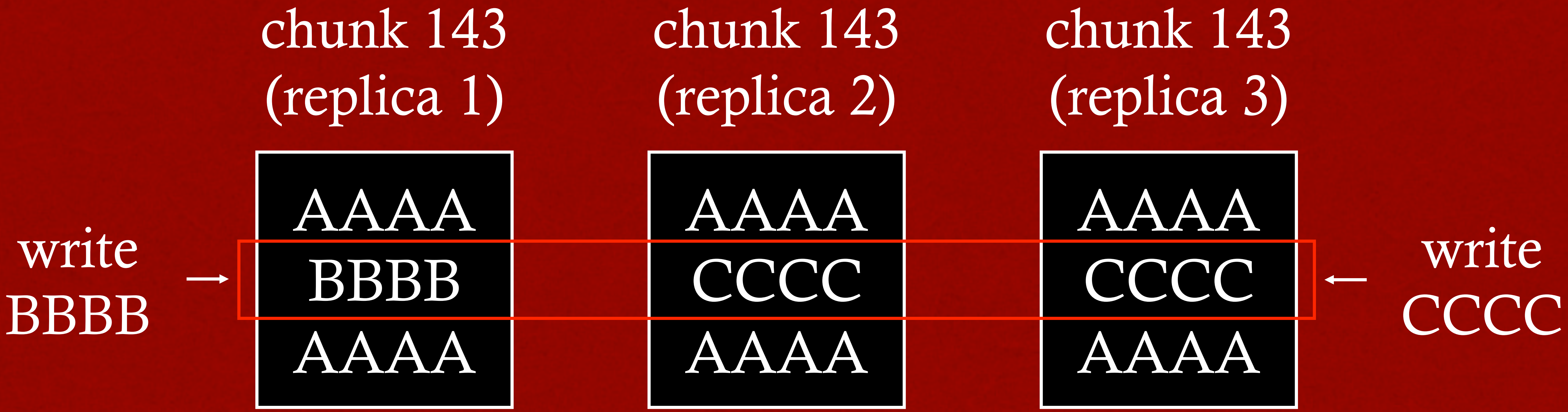
AAAA
BBBB
AAAA

chunk 143
(replica 3)

AAAA
CCCC
AAAA

write
BBBB

write
CCCC



chunk 143
(replica 1)

AAAA
BBBB
AAAA

chunk 143
(replica 2)

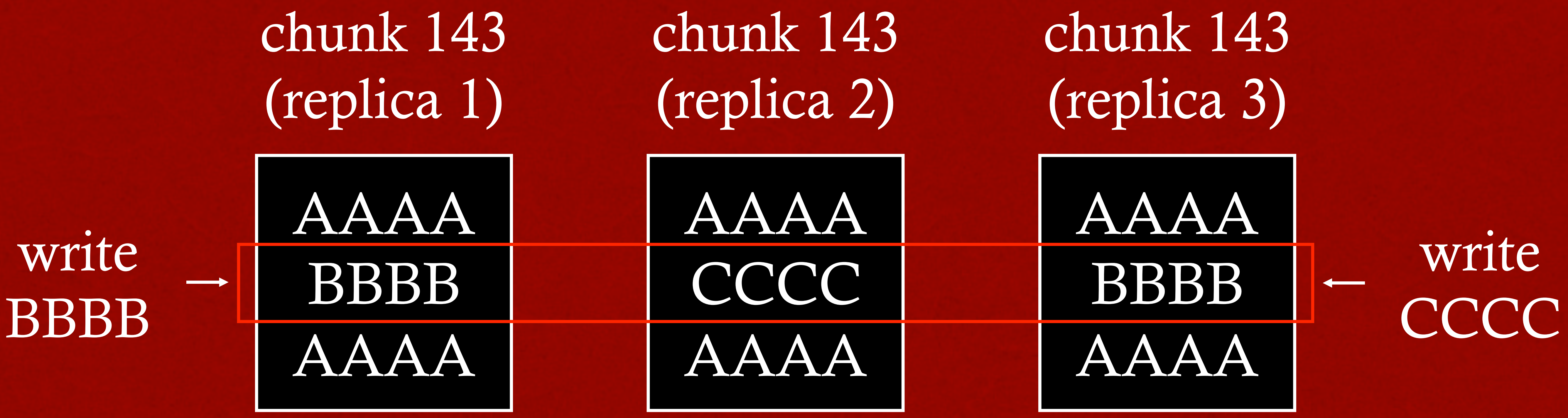
AAAA
CCCC
AAAA

chunk 143
(replica 3)

AAAA
CCCC
AAAA

write
BBBB

write
CCCC



chunk 143
(replica 1)

AAAA
BBBB
AAAA

chunk 143
(replica 2)

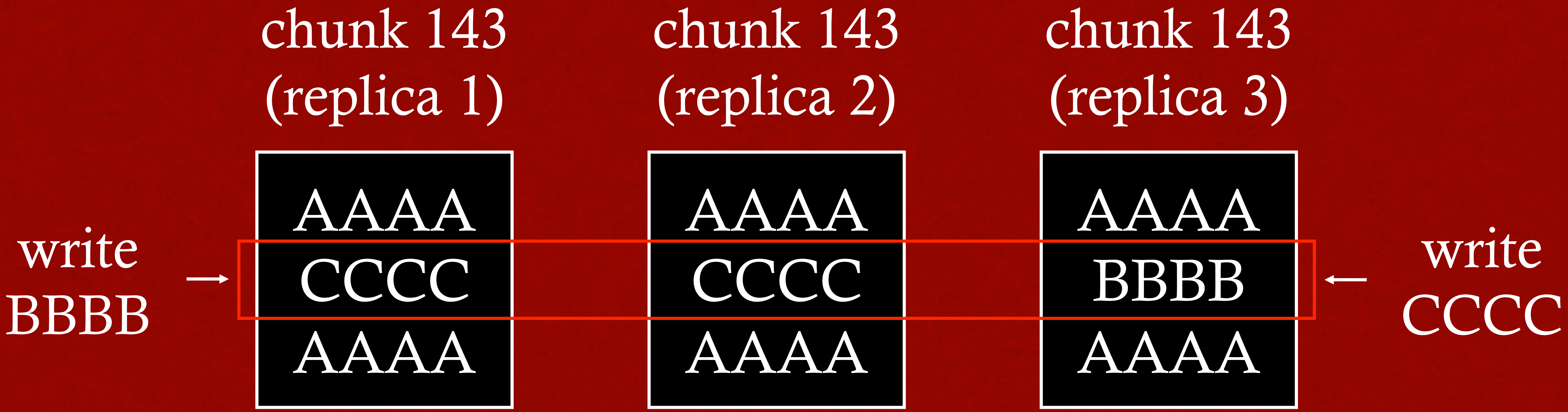
AAAA
CCCC
AAAA

chunk 143
(replica 3)

AAAA
BBBB
AAAA

write
BBBB →

← write
CCCC



chunk 143
(replica 1)

AAAA
CCCC
AAAA

chunk 143
(replica 2)

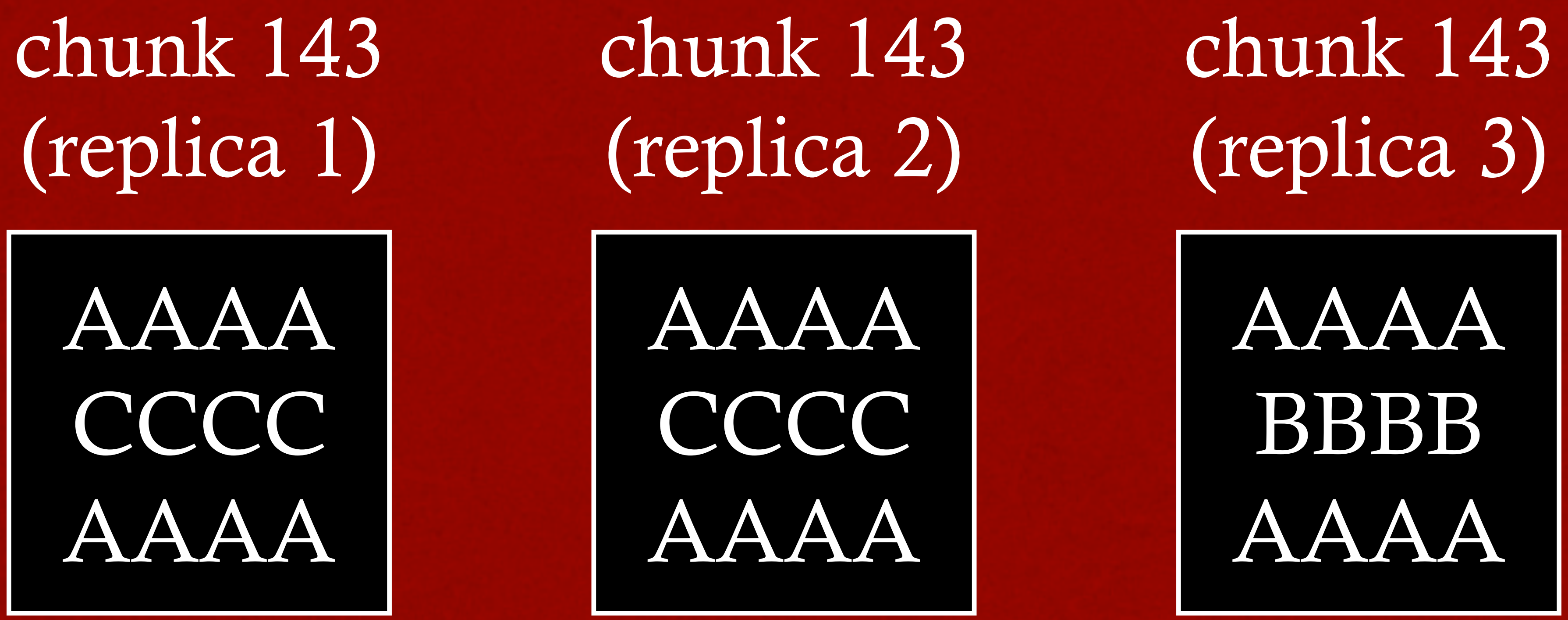
AAAA
CCCC
AAAA

chunk 143
(replica 3)

AAAA
BBBB
AAAA

write
BBBB

write
CCCC



chunk 143
(replica 1)

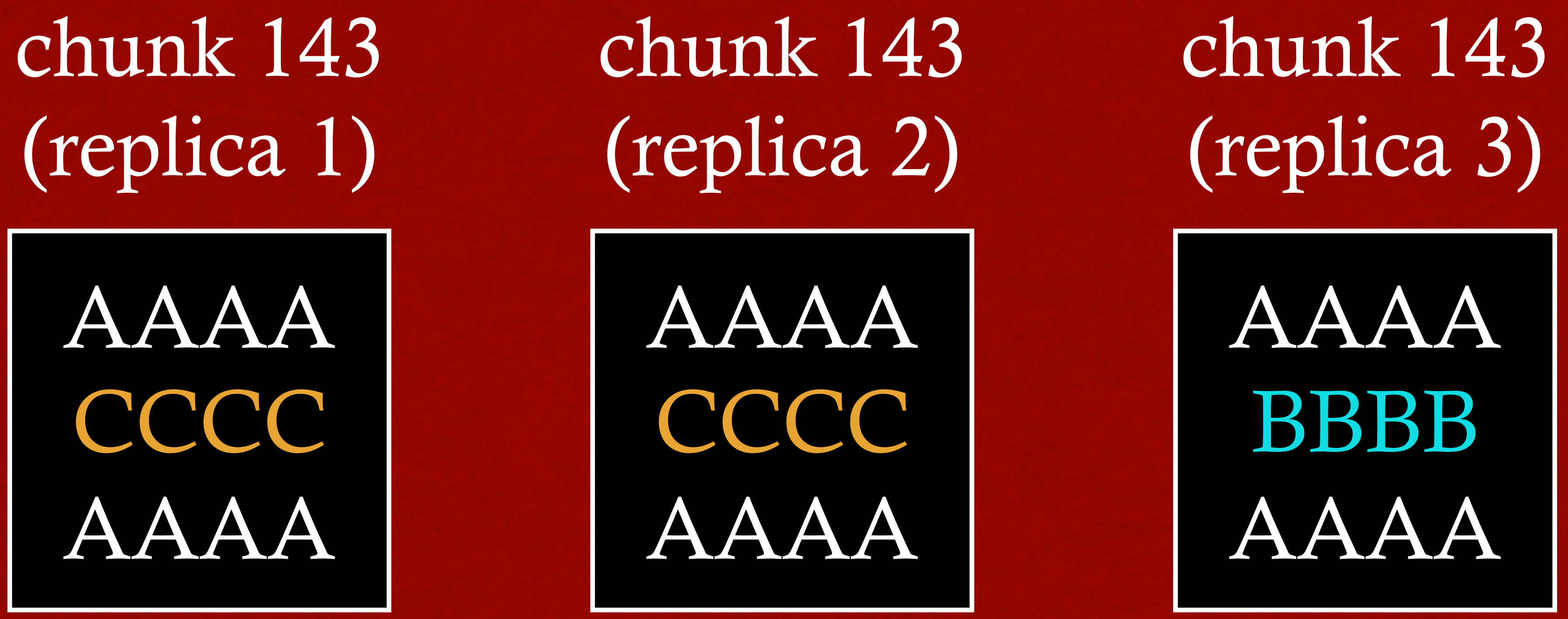
AAAA
CCCC
AAAA

chunk 143
(replica 2)

AAAA
CCCC
AAAA

chunk 143
(replica 3)

AAAA
BBBB
AAAA



chunk 143
(replica 1)

AAAA
CCCC
AAAA

chunk 143
(replica 2)

AAAA
CCCC
AAAA

chunk 143
(replica 3)

AAAA
BBBB
AAAA

Chunks disagree,
but all checksums are correct,
all writes suceeeded,
and no machines ever failed!!

Ideas?

Chunkserver Consistency

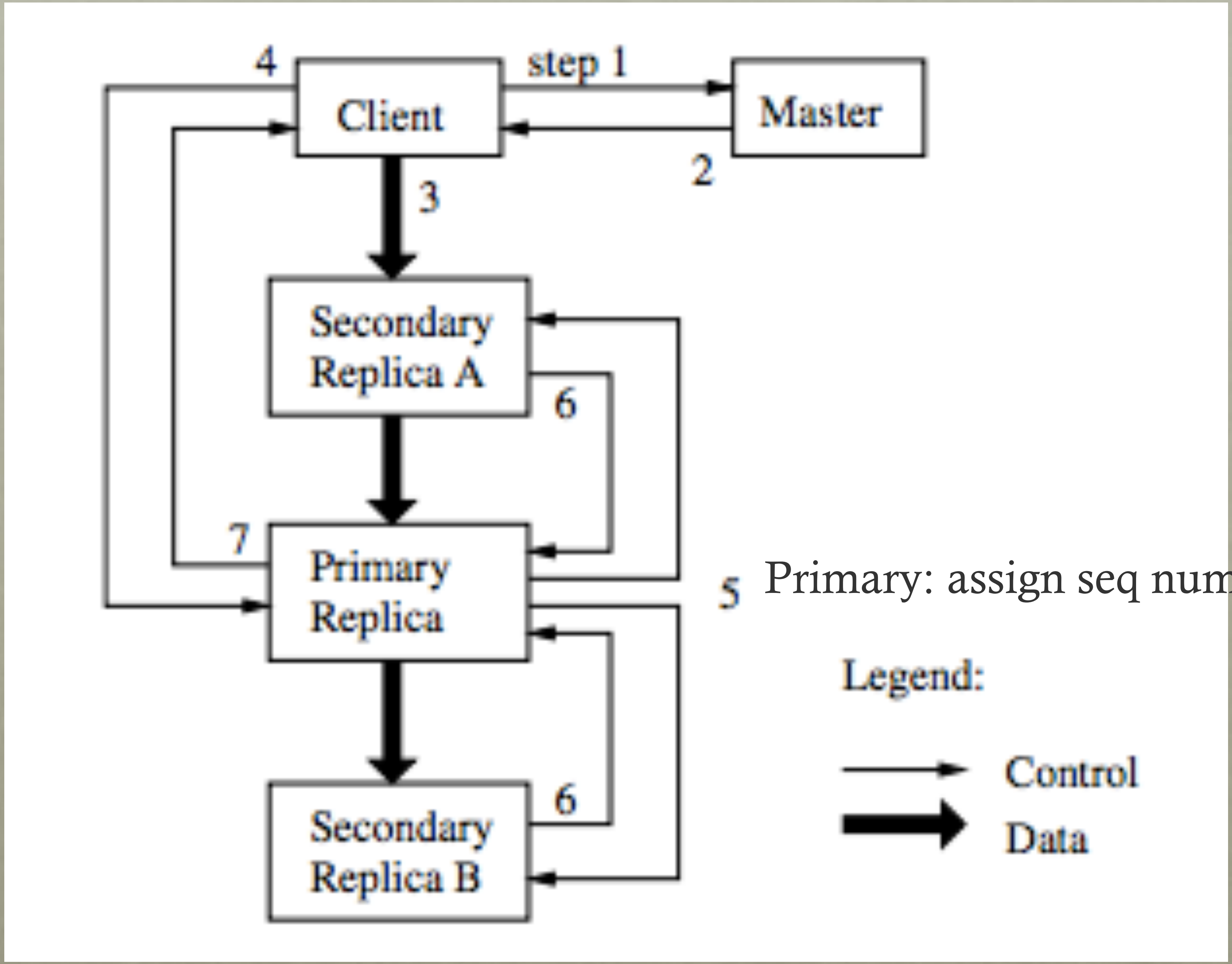
GFS must “serialize” writes across chunkservers

- Decide an order of writes and ensure order is followed by every chunkserver

How to decide on an order?

- don’t want to overload master
- let one replica be primary and decide order of writes from clients

STEPS OF GFS WRITE



Primary: assign seq num to each write

Performance Optimization: Data flows w/ most efficient network path
Correctness: Control flow ensures data committed in same order

PRIMARY REPLICA

Master chooses primary replica for each logical chunk

What if primary dies?

Give primary replica a **lease** that expires after 1 minute

If master wants to reassign primary, and it can't reach old primary, just wait 1 minute

GFS Summary

Fight failure with replication

Metadata consistency is hard, centralize to make it easier

Data consistency is easier, distribute it for scalability