Google File System

Andrew DeOrio

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- GFS design
- Read
- Write
  - Consistency
- Append
- Fault tolerance
- Summary

Review: Distributed systems

- Distributed system: Multiple computers cooperating on a task

- MapReduce: Distributed system for compute
  - Run a program that would be too slow on one computer
  - Last time
- Google File System (GFS): Distributed system for storage
  - Store more data than fits on one computer
  - Today

Review: Distributed system implementation

- How are distributed systems implemented?
- Networking for communication
  - Next time
- Threads and processes for parallelization
  - Coming soon

Review: Datacenter

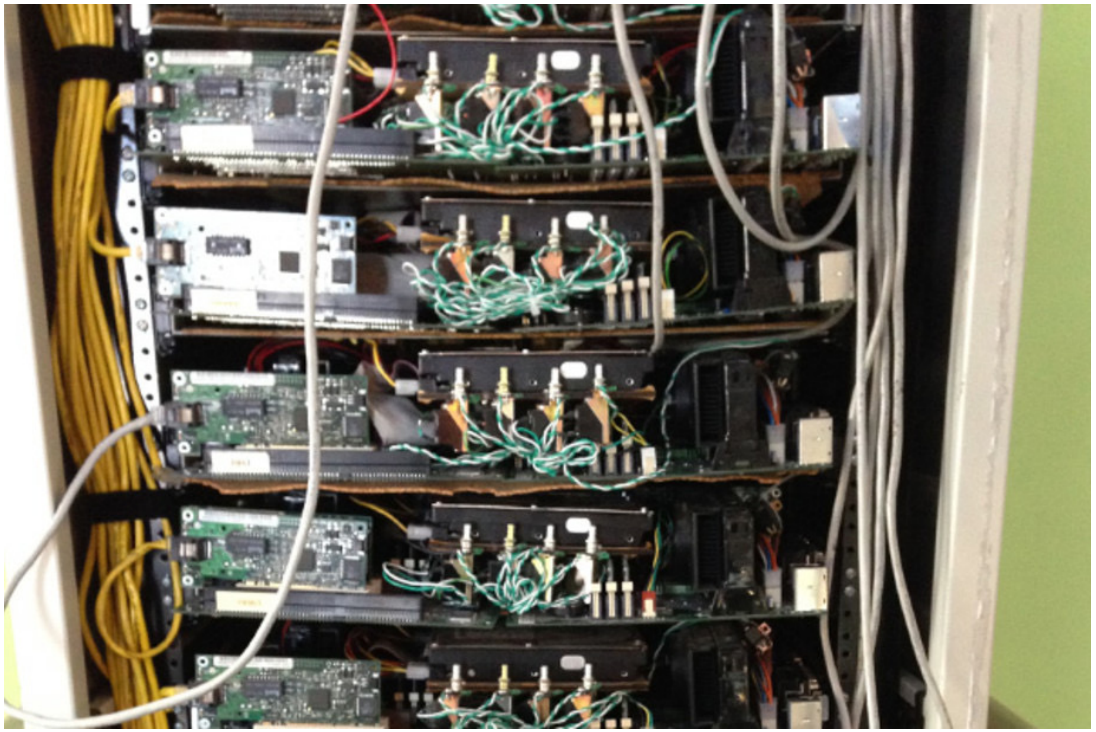
- Where are distributed systems implemented?
- Datacenter: Special building full of computers

- Very loud fan noise
- Hot in some spots and really cold in others
- Unique smell

- Lots of computers means lots of failures

Review: Datacenter computer failures

- Old strategy: buy a few expensive, reliable servers
- New strategy: buy lots of cheap, unreliable servers
  - Use software to make them look reliable
- Modern distributed systems need to solve this problem



Big storage examples

- When do we need storage that's too big for one computer?

- Store server logs from thousands of servers over many days
- Store a copy of the entire web
  - Needed to build a search engine inverted index ("look up table")
  - Several PB of HTML files
- Archive social media data
  - For millions ~ billions of users

Review: Why learn about distributed systems?

- Distributed systems are the solution for dealing with the scale of the web

- GFS is a **distributed system**
  - Many computers cooperating to store a huge amount of data

- Goal: See what the challenges are and what solutions look like

Not GFS: Traditional file systems

- From the user's perspective
  - Directories organized in a tree
  - Files are contiguous

```
$ tree p3-insta485-clientside
p3-insta485-clientside
├── bin
│   ├── insta485db
│   ├── insta485run
│   ├── insta485test
│   └── insta485test-html
├── insta485
│   ├── config.py
│   ├── __init__.py
│   ├── model.py
...
```

```
$ cat insta485/config.py
"""
Insta485 development
configuration.

Andrew DeOrio
<awdeorio@umich.edu>
"""

import os

...
```

Traditional file systems

- From the user's perspective
  - Directories organized in a tree
  - Files are contiguous
- From the disk's perspective
  - Files and directories are organized as sequences of non-contiguous blocks
- Disk AKA hard disk drive AKA HDD
  - Physical storage

Image credit: https://commons.wikimedia.org/w/index.php?curid=27940250
Image credit: https://commons.wikimedia.org/w/index.php?curid=30202520

Traditional file systems

- Disk is organized as an array of blocks
  - Analogous to memory organized as an array of bytes
  - Actual file content
- On-disk *inode* (index node) data structure describes each file or directory
  - Pointers to data blocks
  - Timestamps, e.g., last modified
  - *etc.*

Inodes and data blocks

- A file or directory's data blocks will probably not be contiguous on disk.
- Inode stores pointers to the data blocks.

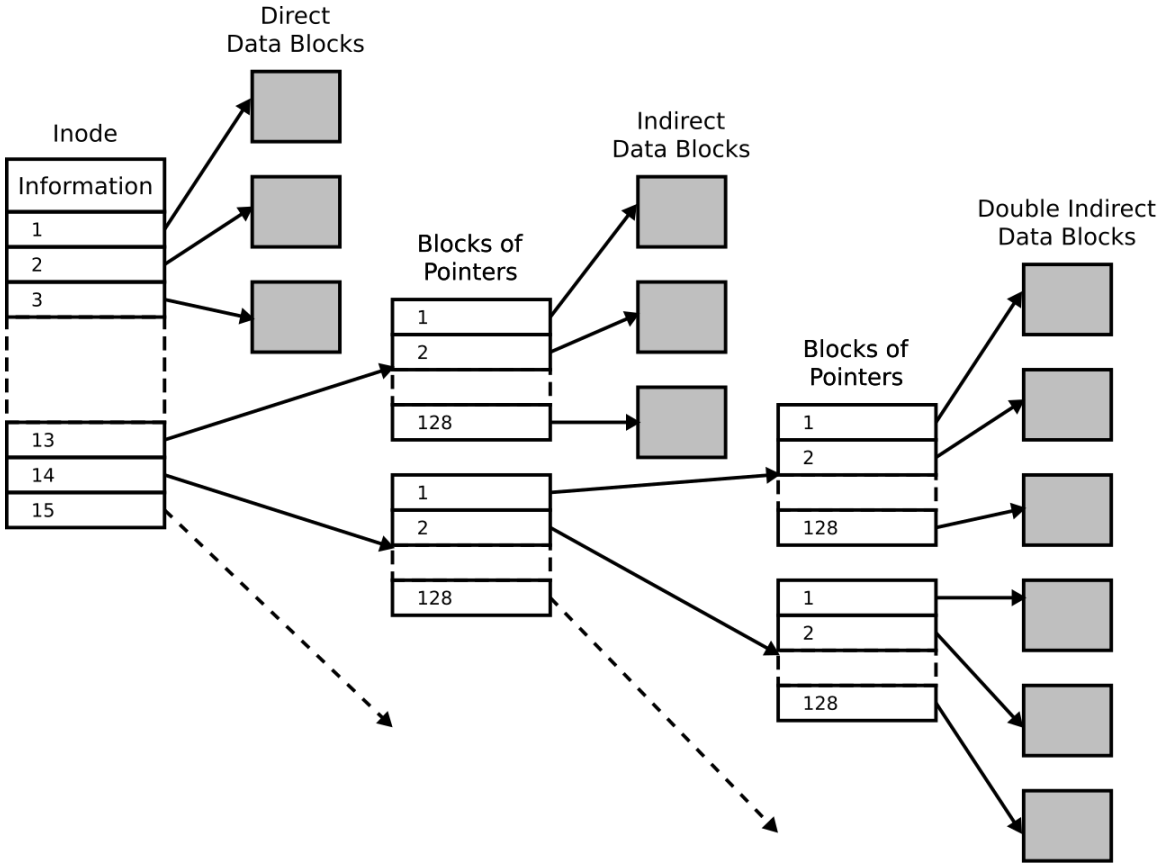


Image source: https://en.wikipedia.org/wiki/Inode_pointer_structure

Problem with traditional file systems

- A traditional file system organizes the data on **one** hard disk drive
- ~10 TB on one drive
- Size of the web is ~10 PB
  - ~1,000 drives
  - Before any backups
- The web is too big to fit on one disk or even one computer with several disks

Image credit: https://commons.wikimedia.org/w/index.php?curid=27940250
Image credit: https://commons.wikimedia.org/w/index.php?curid=30202520

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- **GFS design**
- Read
- Write
  - Consistency
- Append
- Fault tolerance
- Summary

Google File System (GFS)

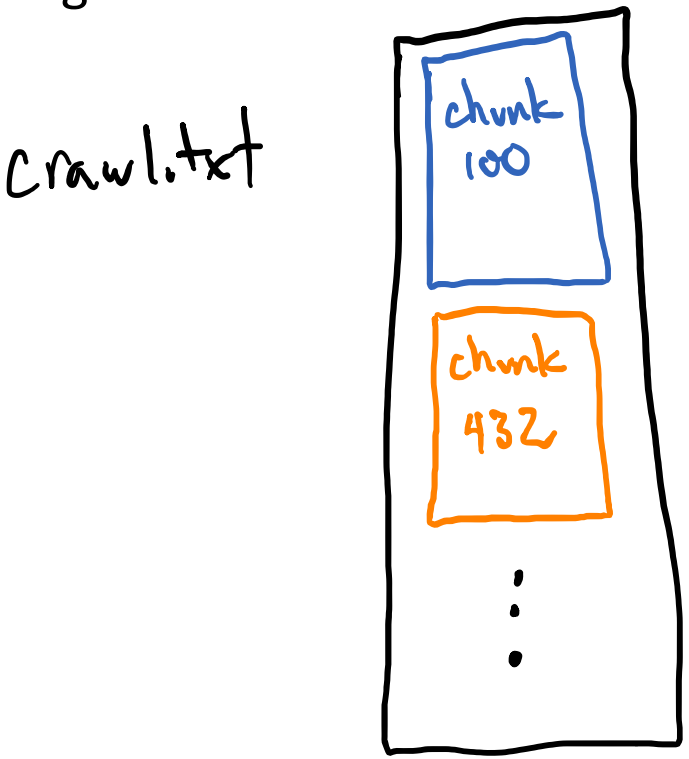
- Google File System: 2003

- Problem:
  - Store enormous amount of data
  - Don't lose data when a server fails

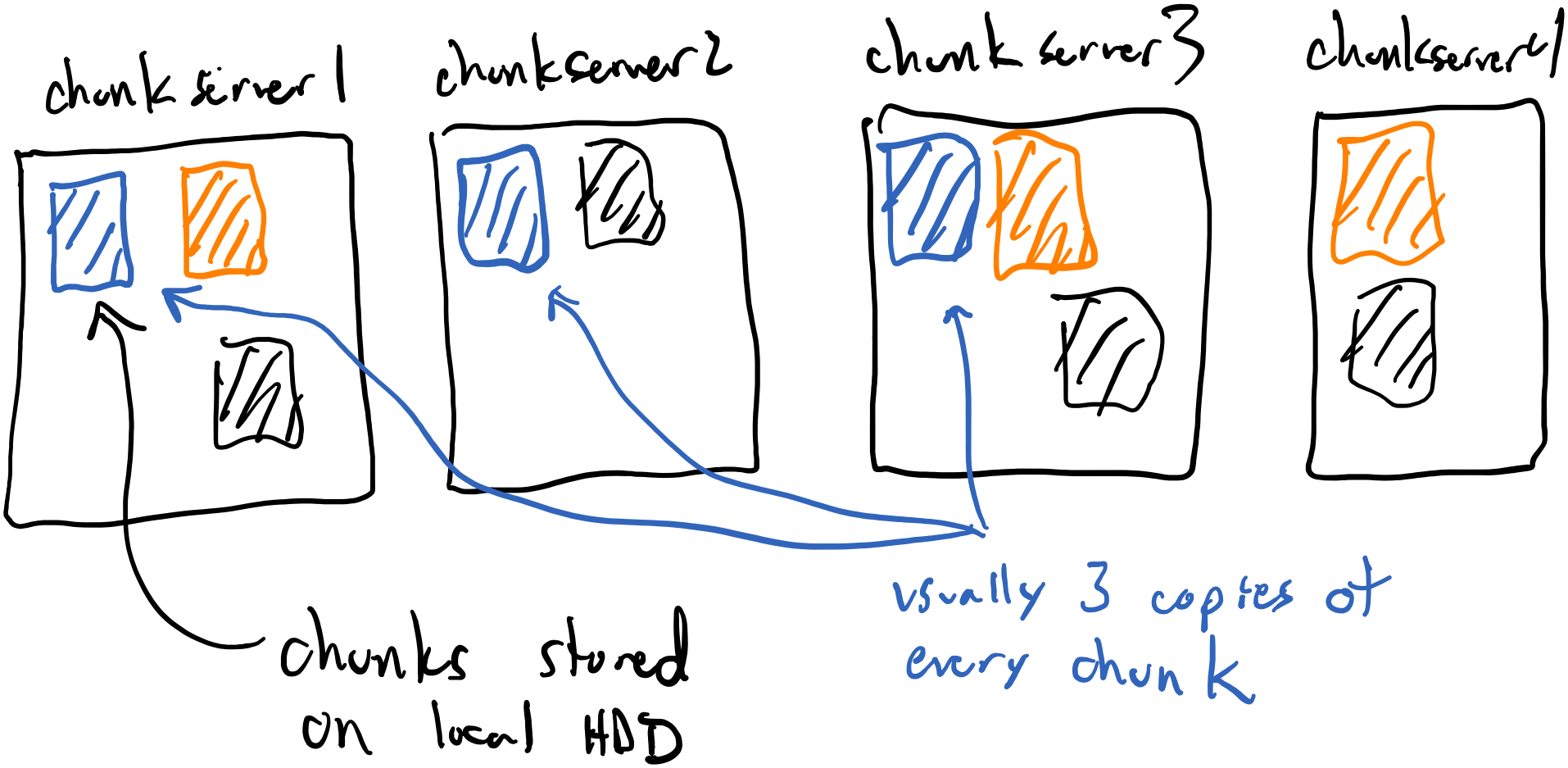
- Solution:
  - Store multiple copies of everything
  - Keep track of where those copies are

GFS design solution

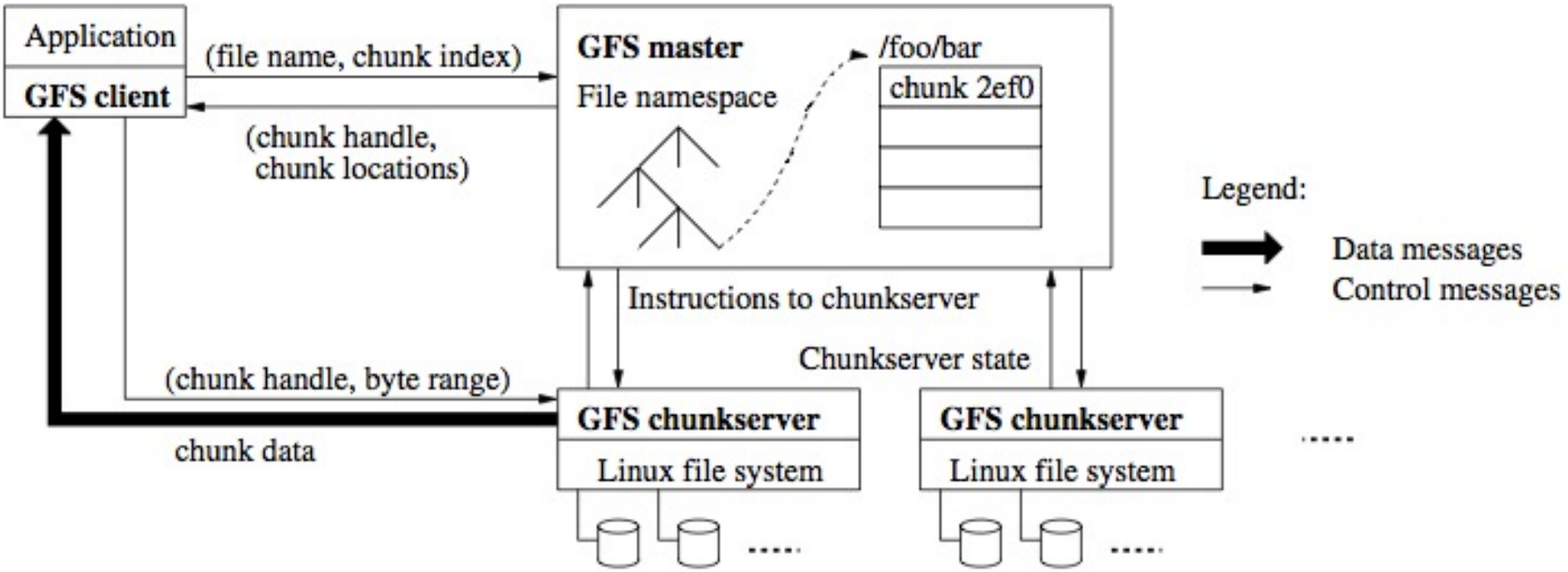
- Files stored as series of ***chunks***
- Reliability through replication
  - Each chunk replicated across 3+ ***chunkservers***
- Single main server coordinates access, metadata



Chunkservers



GFS design details



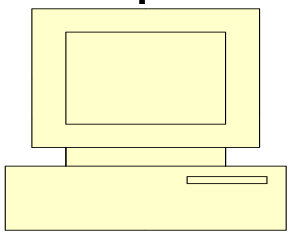
Agenda

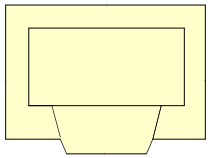
- Review: Distributed systems introduction
  - Google File System motivation
- GFS design
- **Read**
- Write
  - Consistency
- Append
- Fault tolerance
- Summary

GFS file read

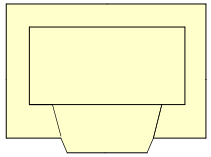
main

"crawl.txt"

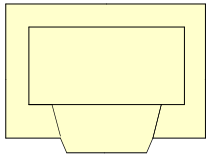




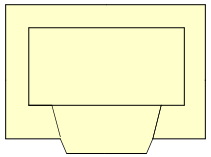
cserver 0



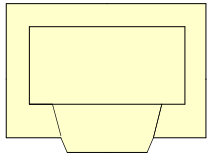
cserver 1



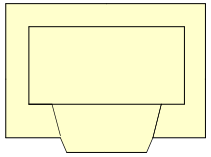
cserver 2



cserver 3



cserver 4

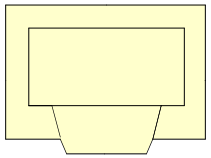


cserver 5

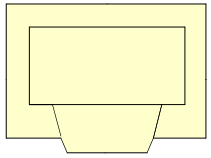
1. Client asks main for filename info

GFS file read

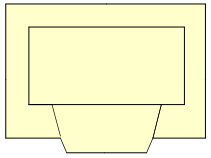
main



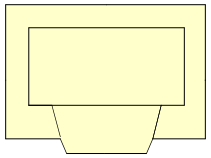
cserver 0



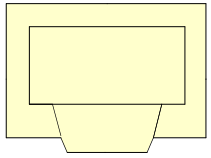
cserver 1



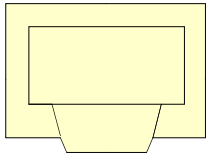
cserver 2



cserver 3

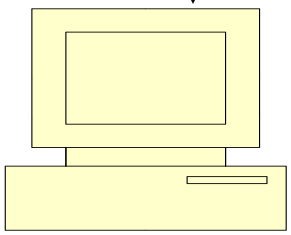


cserver 4



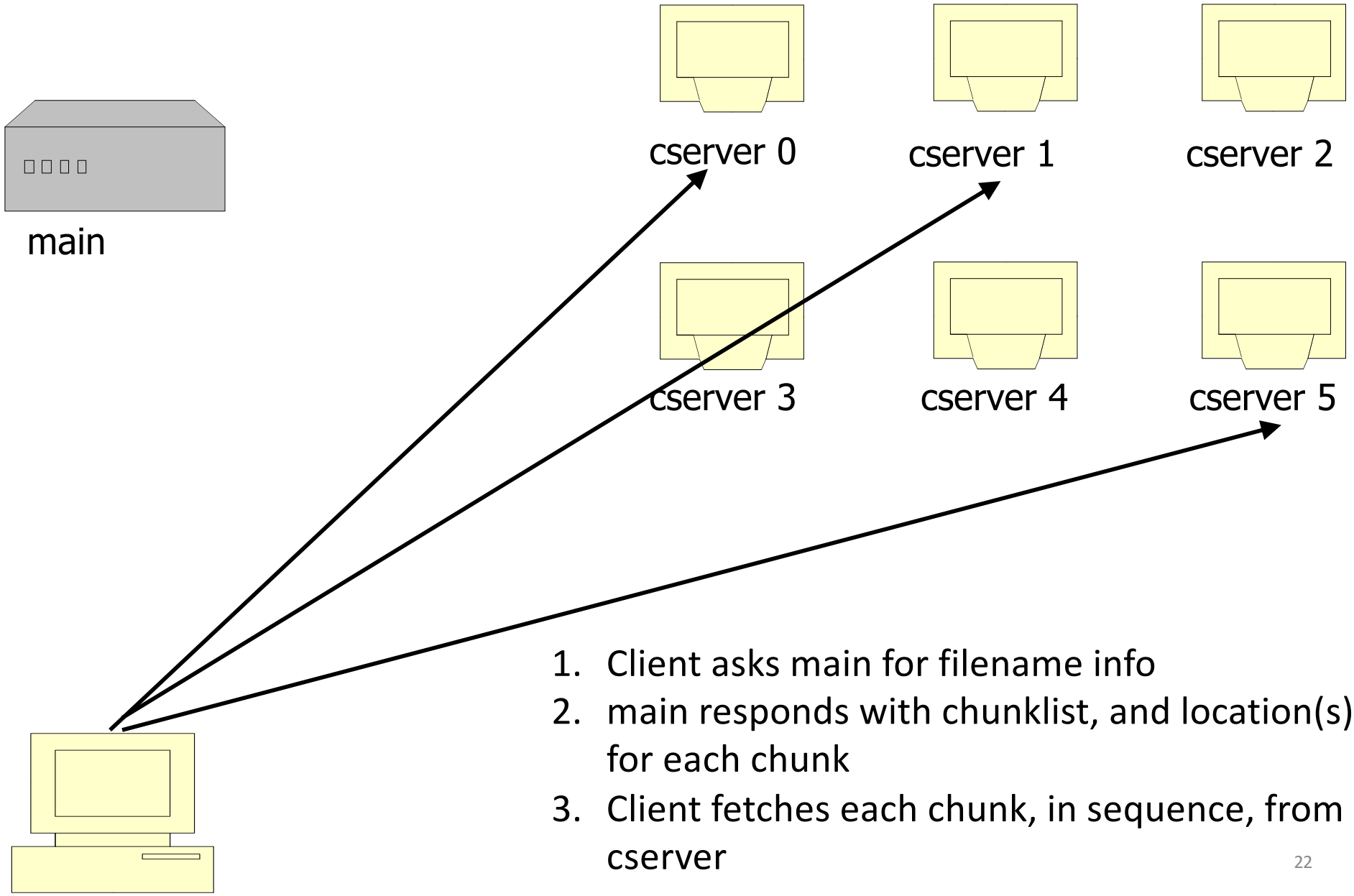
cserver 5

(chunk-33 / cserver 1, 4)
(chunk-95 / cserver 0, 2)
(chunk-65 / cserver 1, 4, 5)



1. Client asks main for filename info
2. main responds with chunklist, and location(s) for each chunk

GFS file read



1. Client asks main for filename info
2. main responds with chunklist, and location(s) for each chunk
3. Client fetches each chunk, in sequence, from a cserver

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- GFS design
- Read
- **Write**
  - **Consistency**
- Append
- Fault tolerance
- Summary

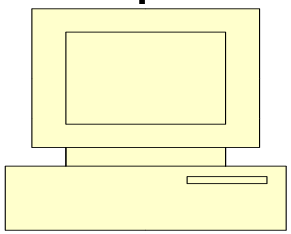
GFS file write

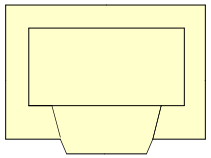
- A write would work just like a read if
  - Clients take turns; and
  - There are no errors
- Serial, successful writes
- Spoiler alert: these assumptions break down

One write

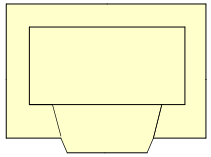
main

"crawl.txt"

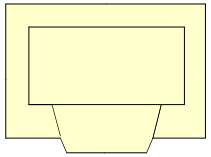




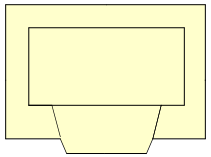
cserver 0



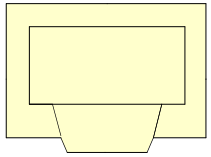
cserver 1



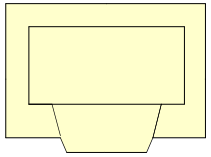
cserver 2



cserver 3



cserver 4

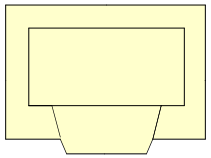


cserver 5

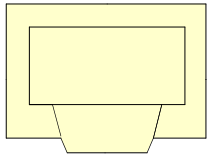
1. Client asks main for filename info

One write

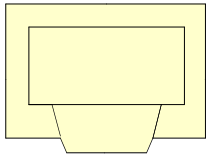
main



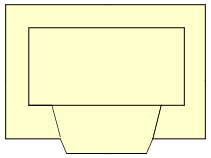
cserver 0



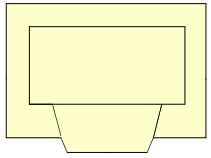
cserver 1



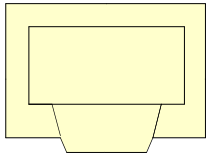
cserver 2



cserver 3

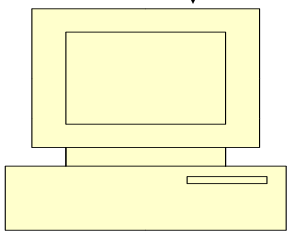


cserver 4



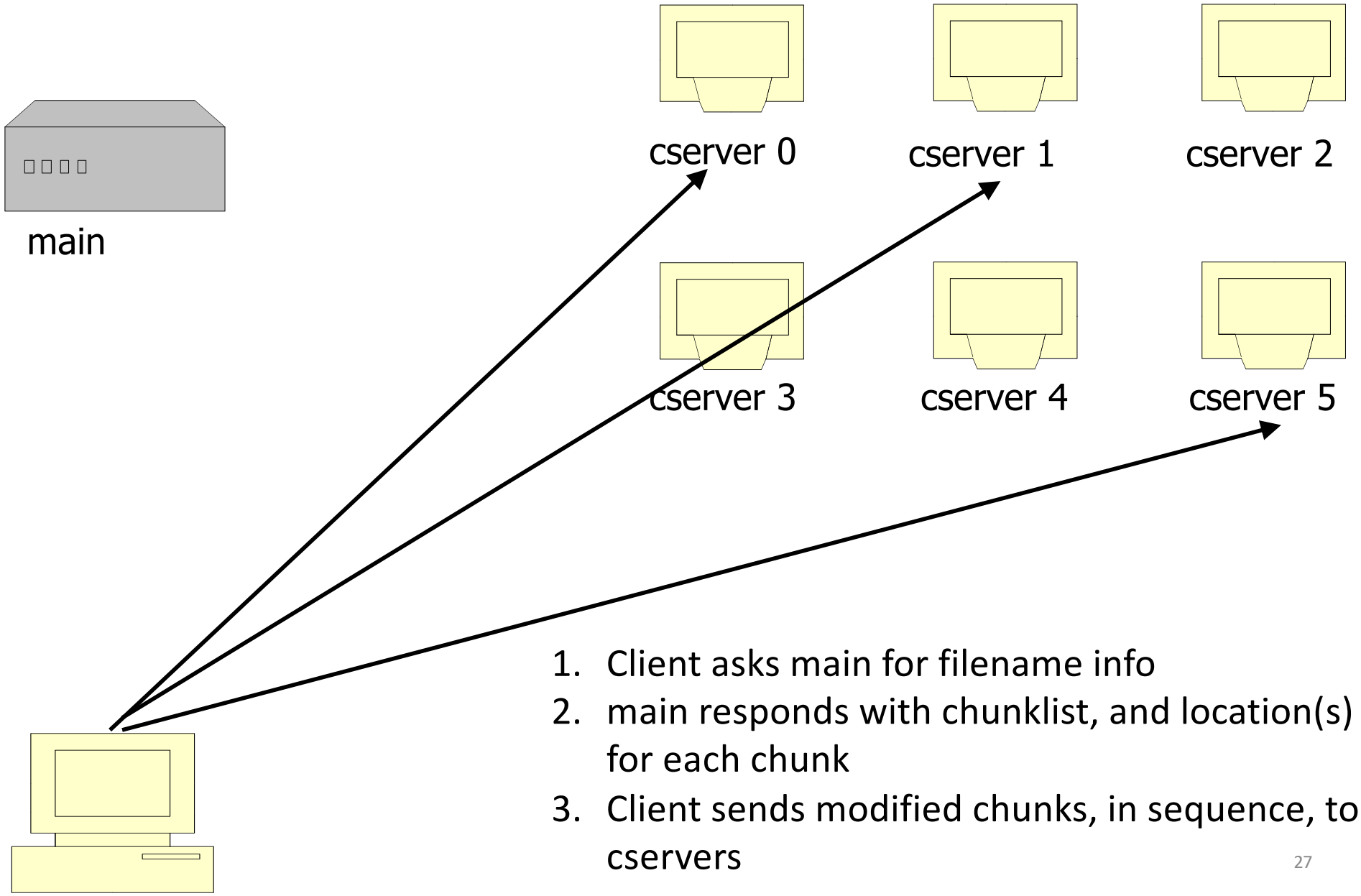
cserver 5

(chunk-33 / cserver 1, 4)
(chunk-95 / cserver 0, 2)
(chunk-65 / cserver 1, 4, 5)



1. Client asks main for filename info
2. main responds with chunklist, and location(s) for each chunk

One write

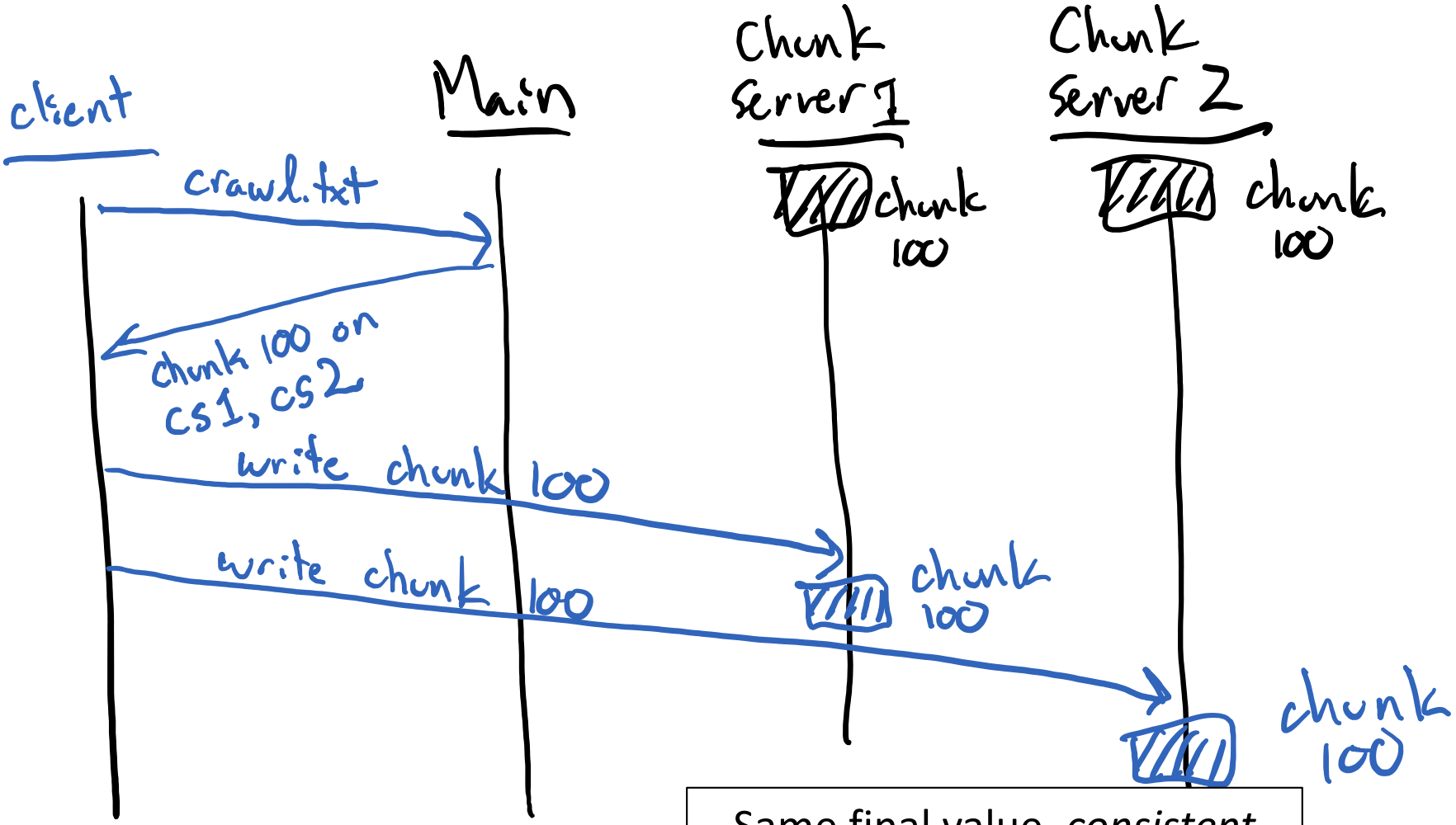


1. Client asks main for filename info
2. main responds with chunklist, and location(s) for each chunk
3. Client sends modified chunks, in sequence, to cservers

One write

- One write
- All chunkservers get the same value for the same chunk

One write

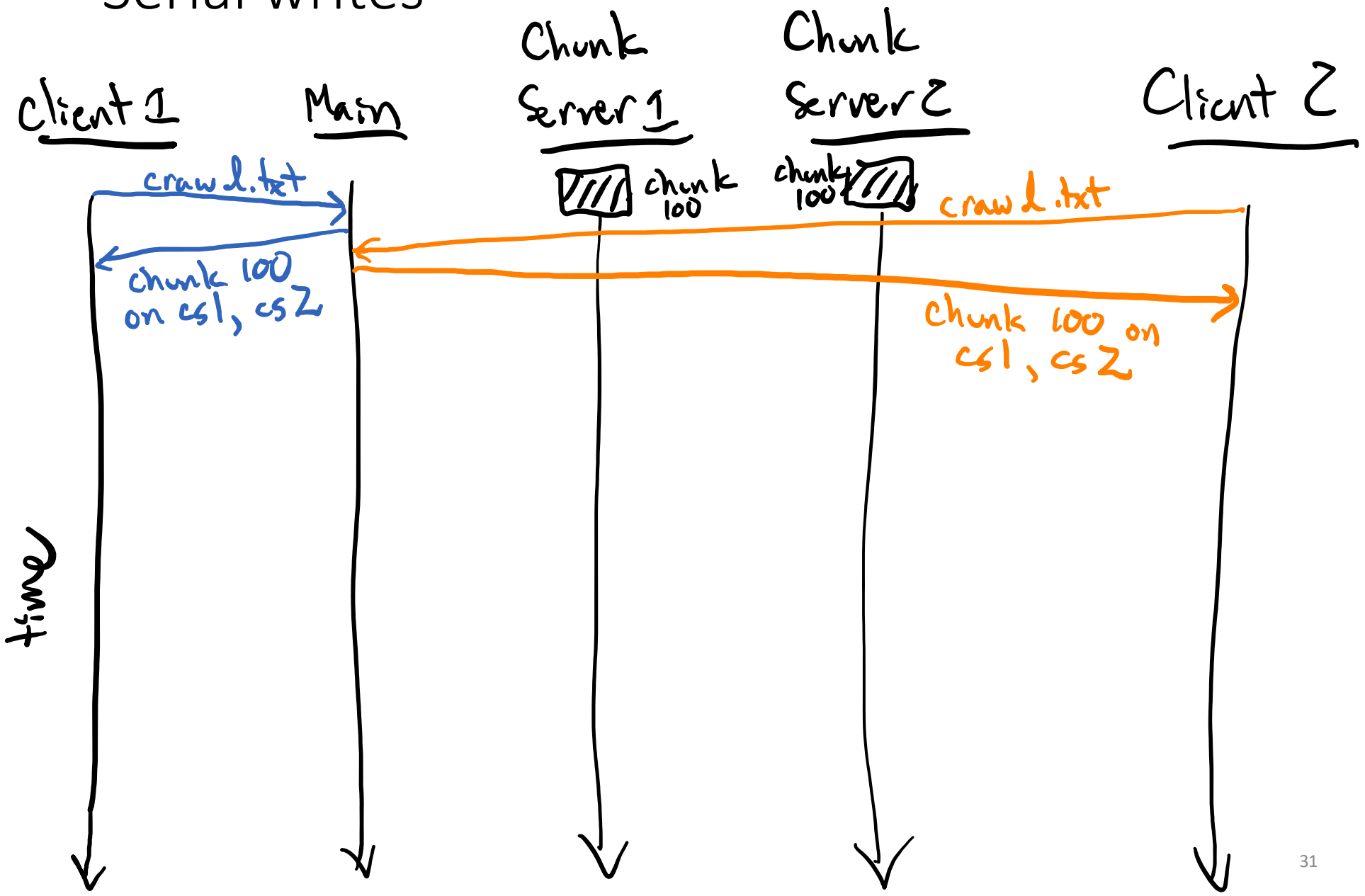


Same final value, *consistent*

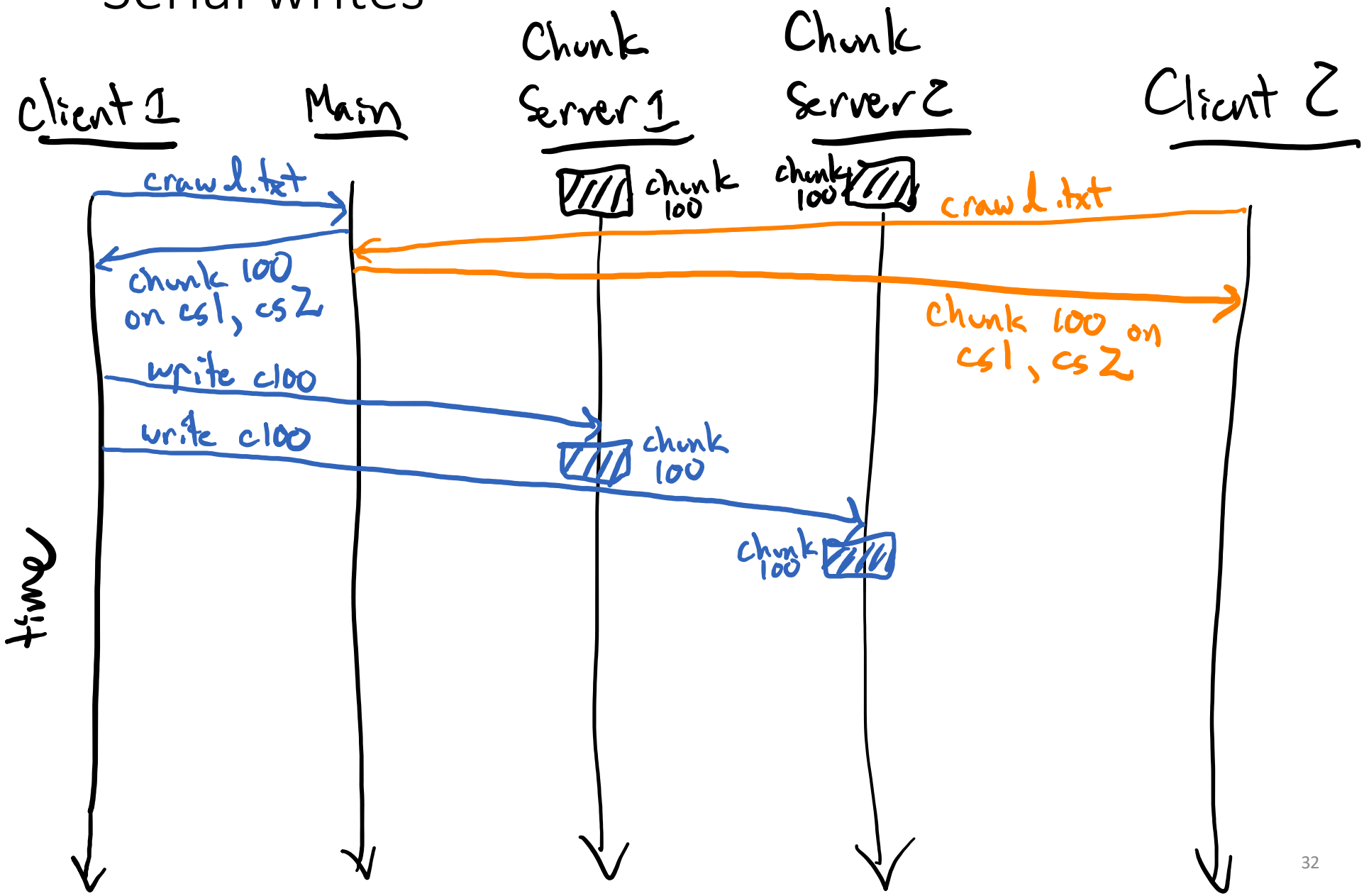
Serial writes

- Two serial writes
- First write
  - All chunkservers get the same value for the same chunk
- Second write
  - All chunkservers get the same new value for the same chunk

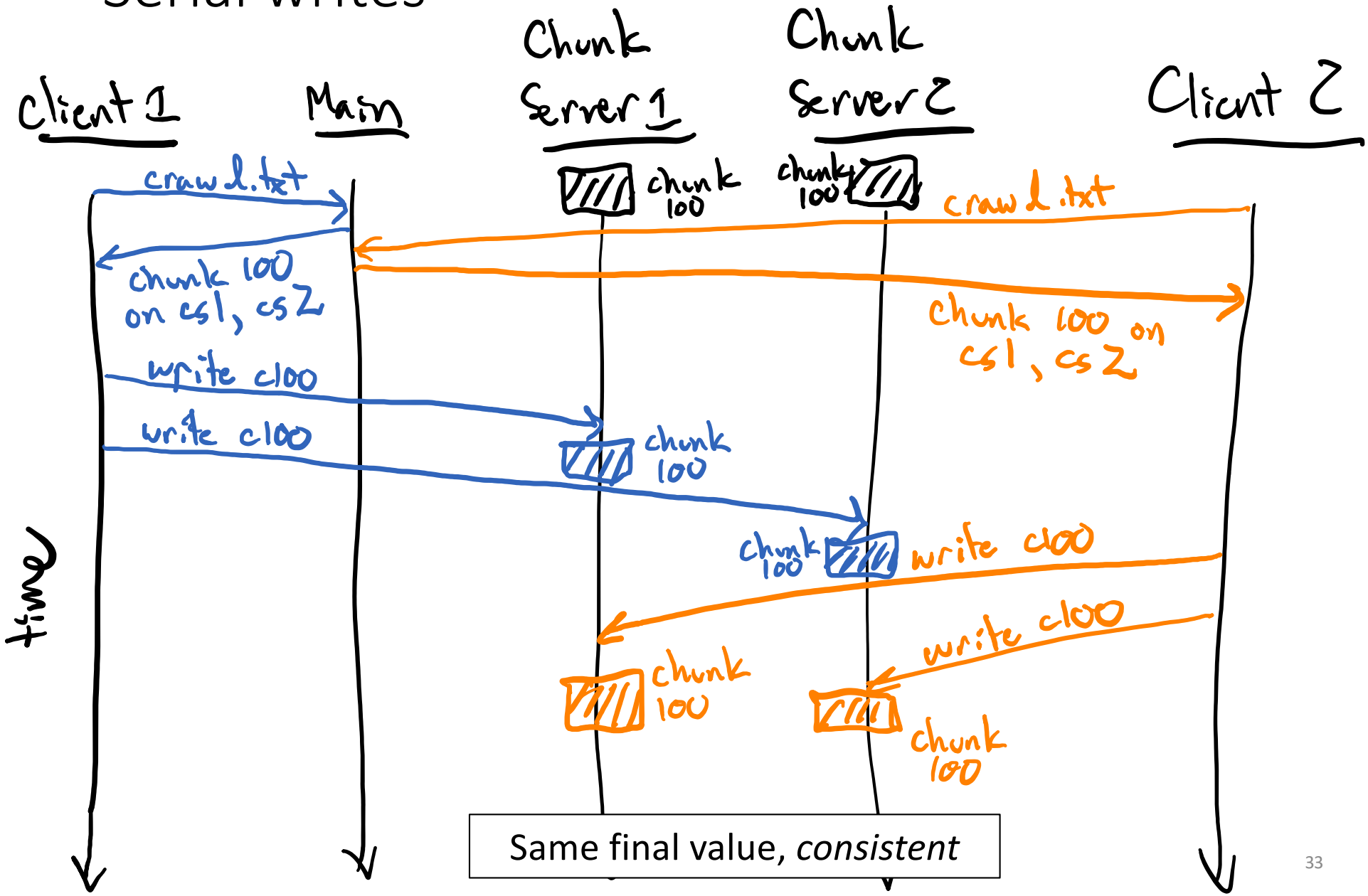
Serial writes



Serial writes



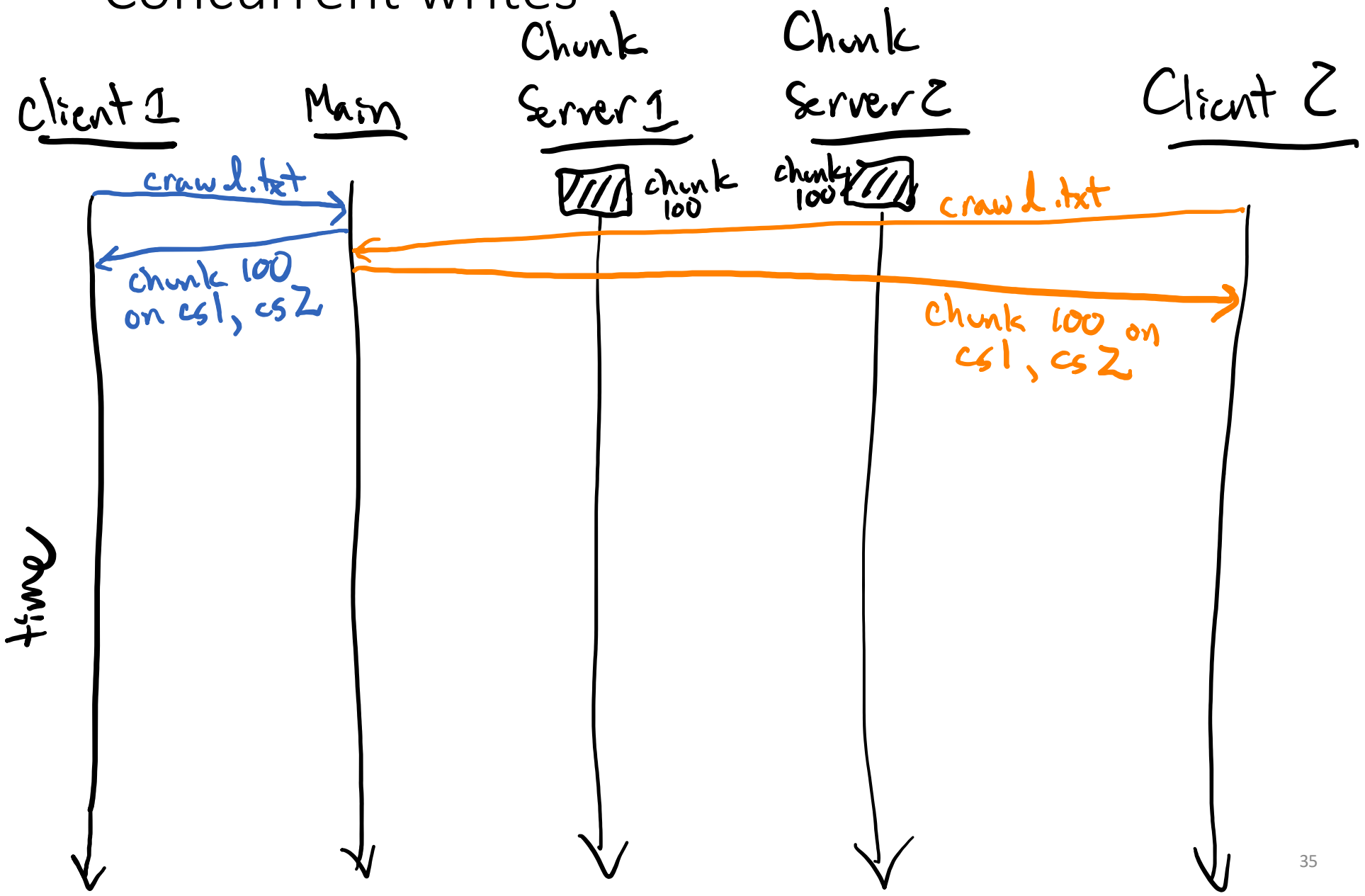
Serial writes



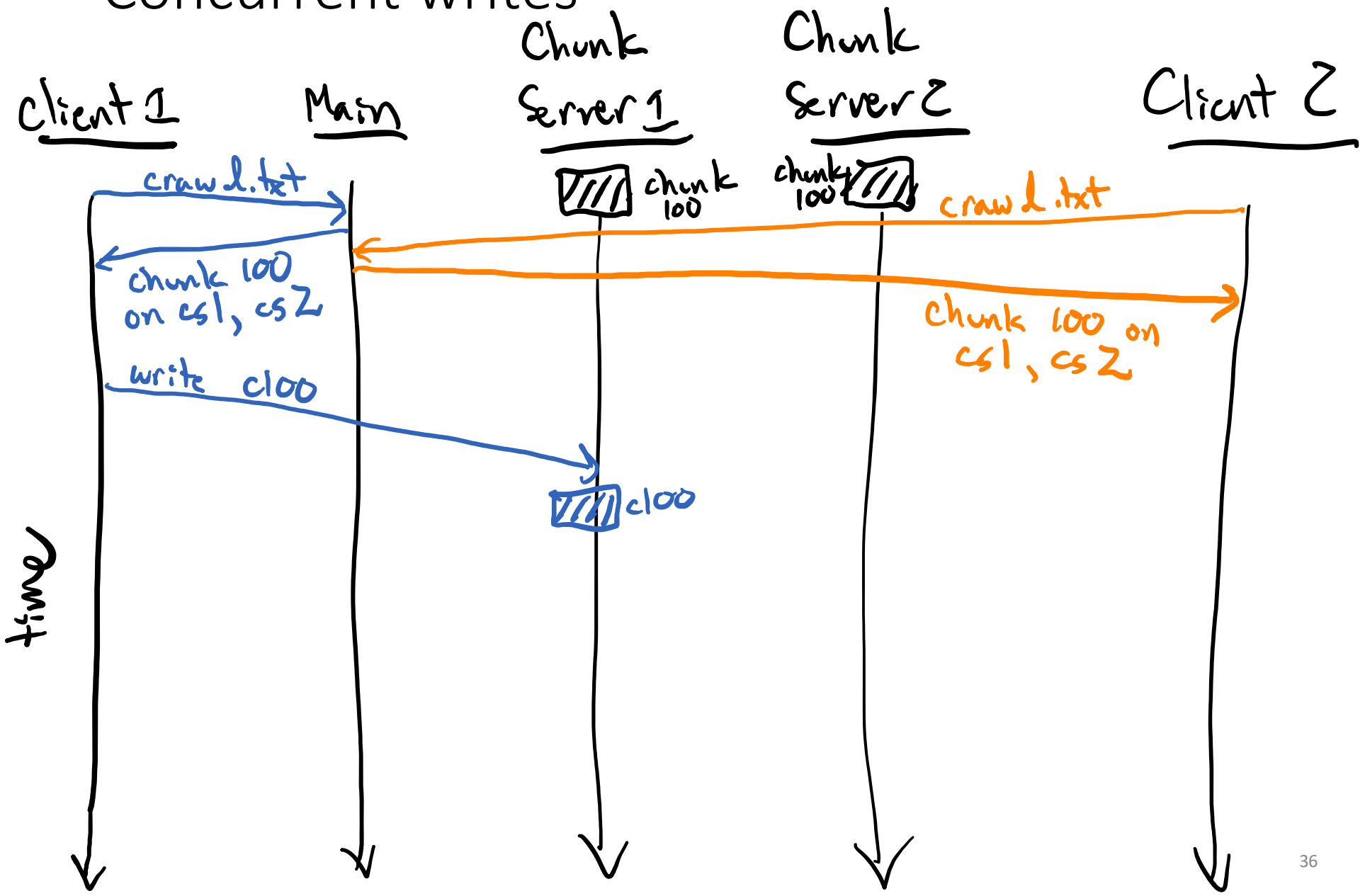
Concurrent writes

- Two writes from two clients happen at (almost) the same time
- Problem: two different values on two different chunkservers for the same chunk
  - *Inconsistent*

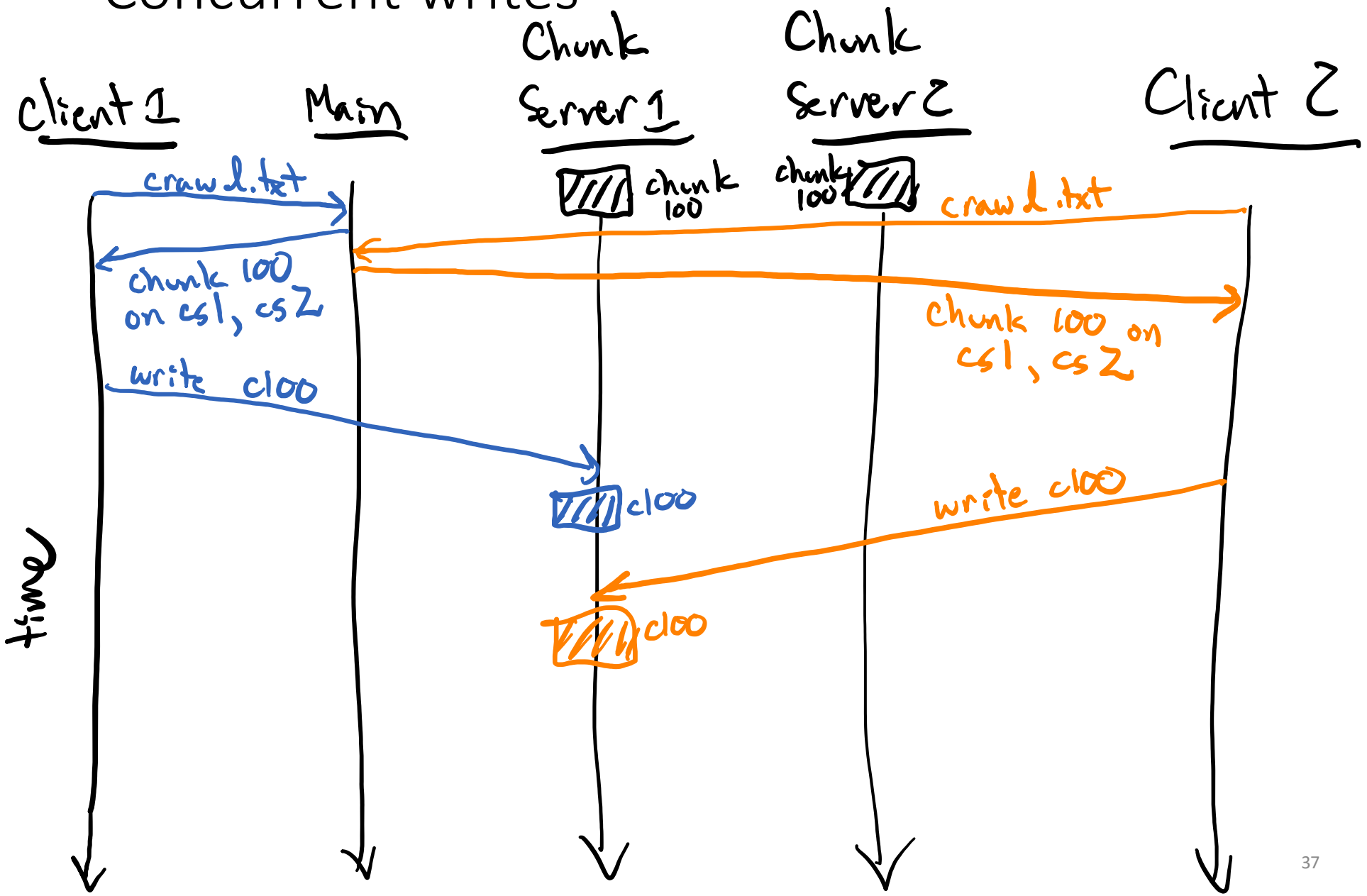
Concurrent writes



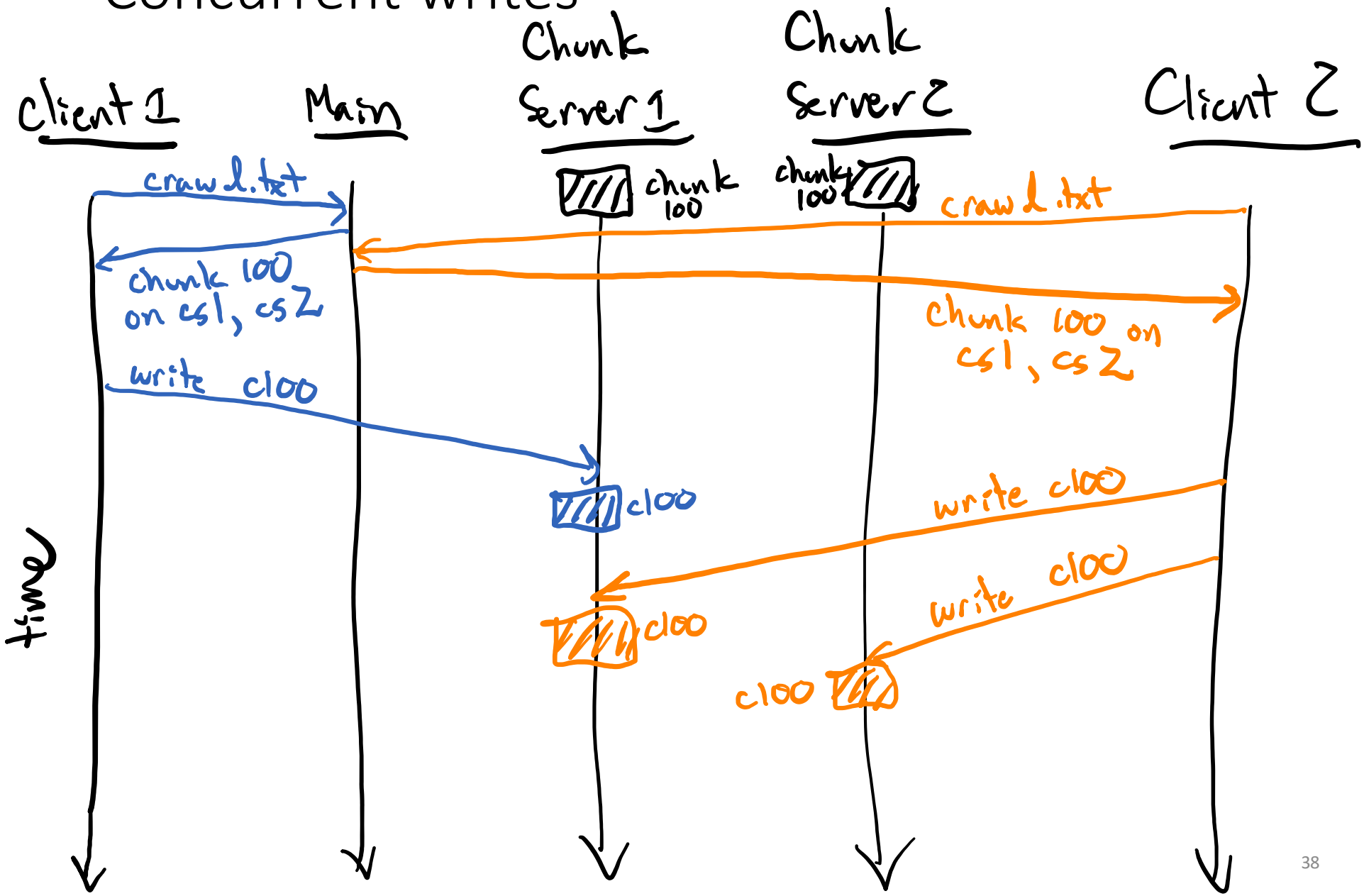
Concurrent writes



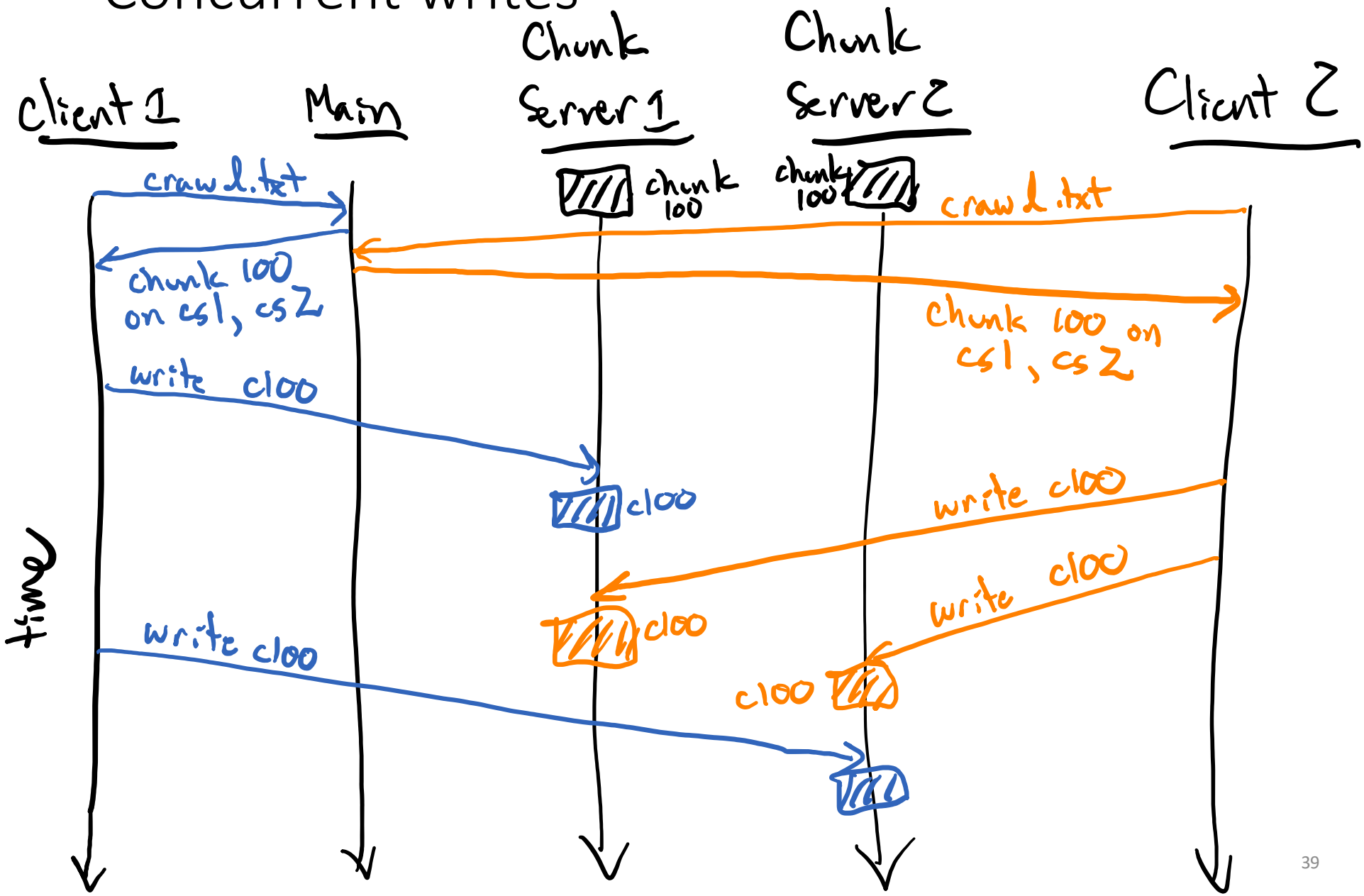
Concurrent writes



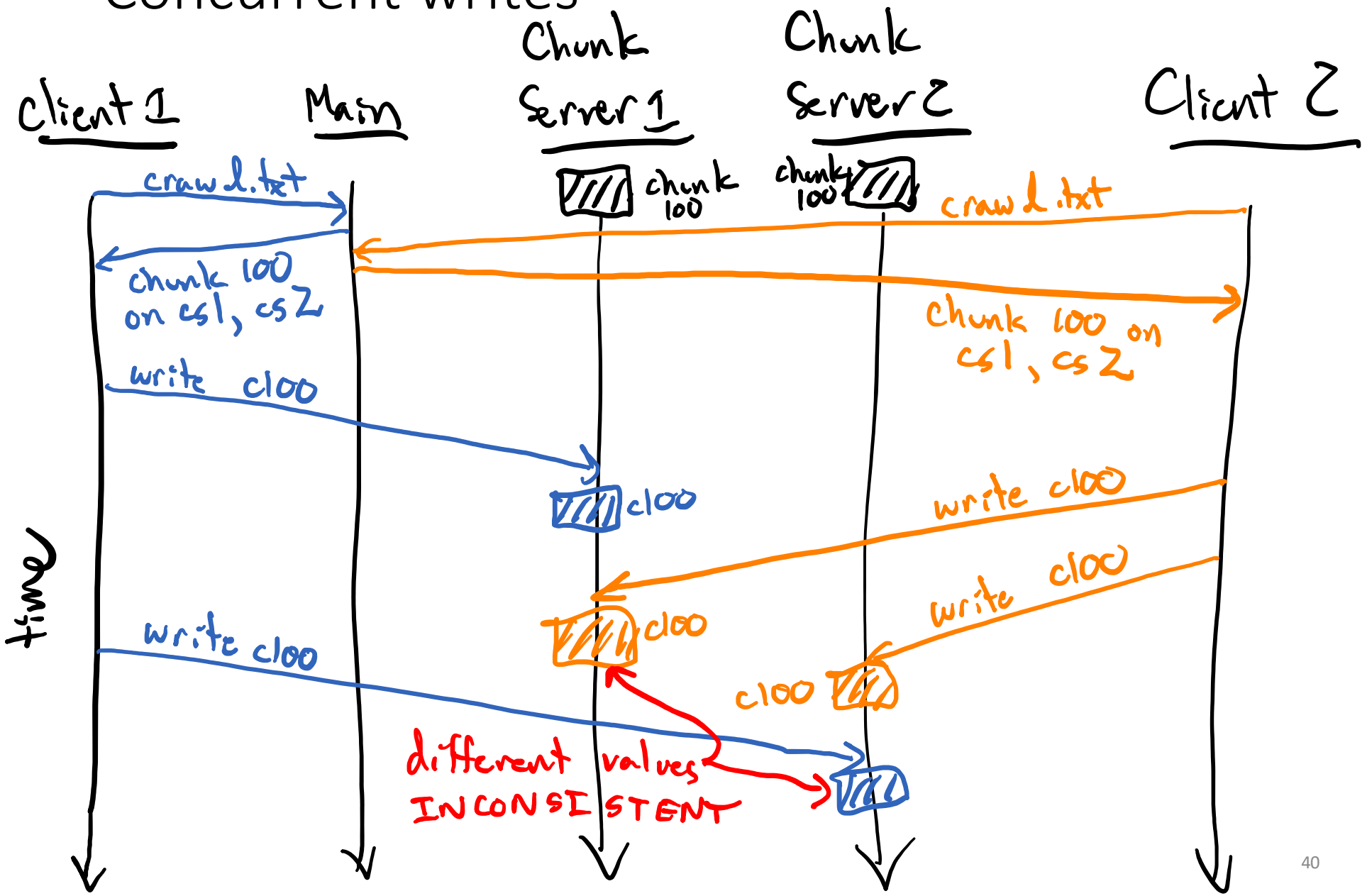
Concurrent writes



Concurrent writes



Concurrent writes



Consistency

- *Consistent*: Every copy has the same value

- To guarantee consistency, writes need to be coordinated

GFS write forwarding

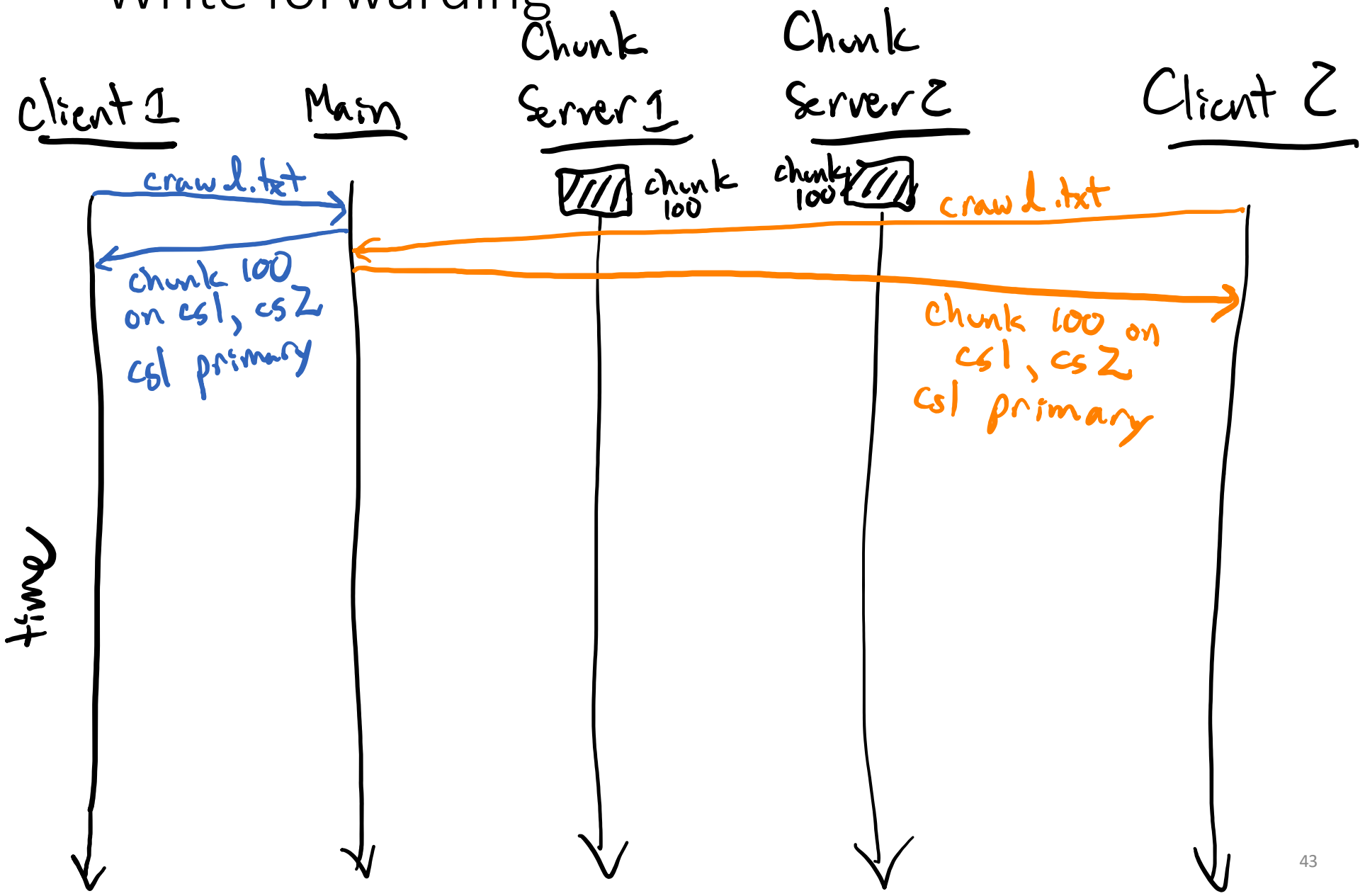
- *Consistent* in GFS: Every chunkserver has the same value for the same chunk

- To guarantee consistency, writes need to be coordinated
- GFS calls this *write forwarding*

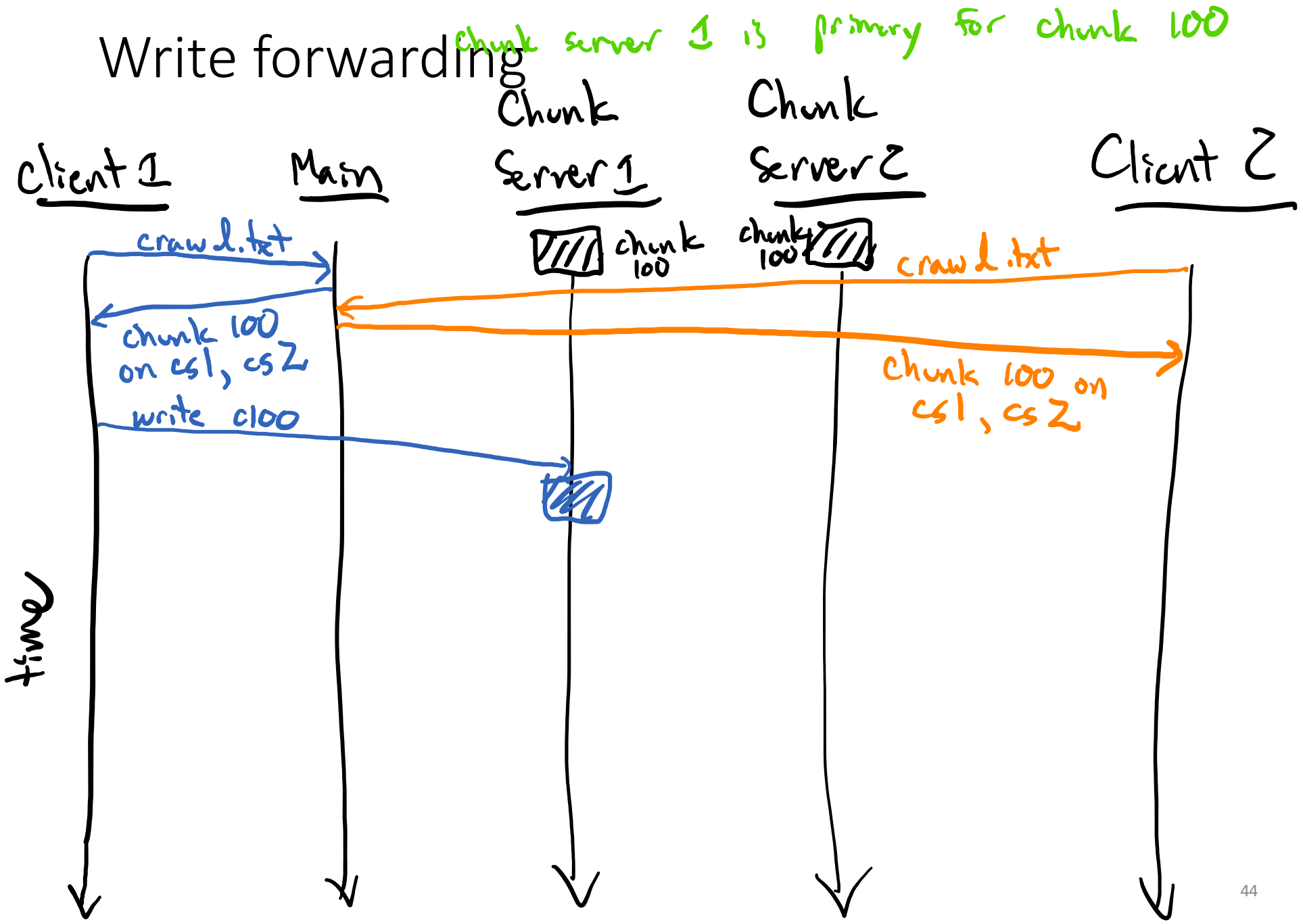
- One chunkserver is designated as *primary* for each chunk id
- Primary forwards writes to replica chunkservers*

* Simplification, see paper for full details.

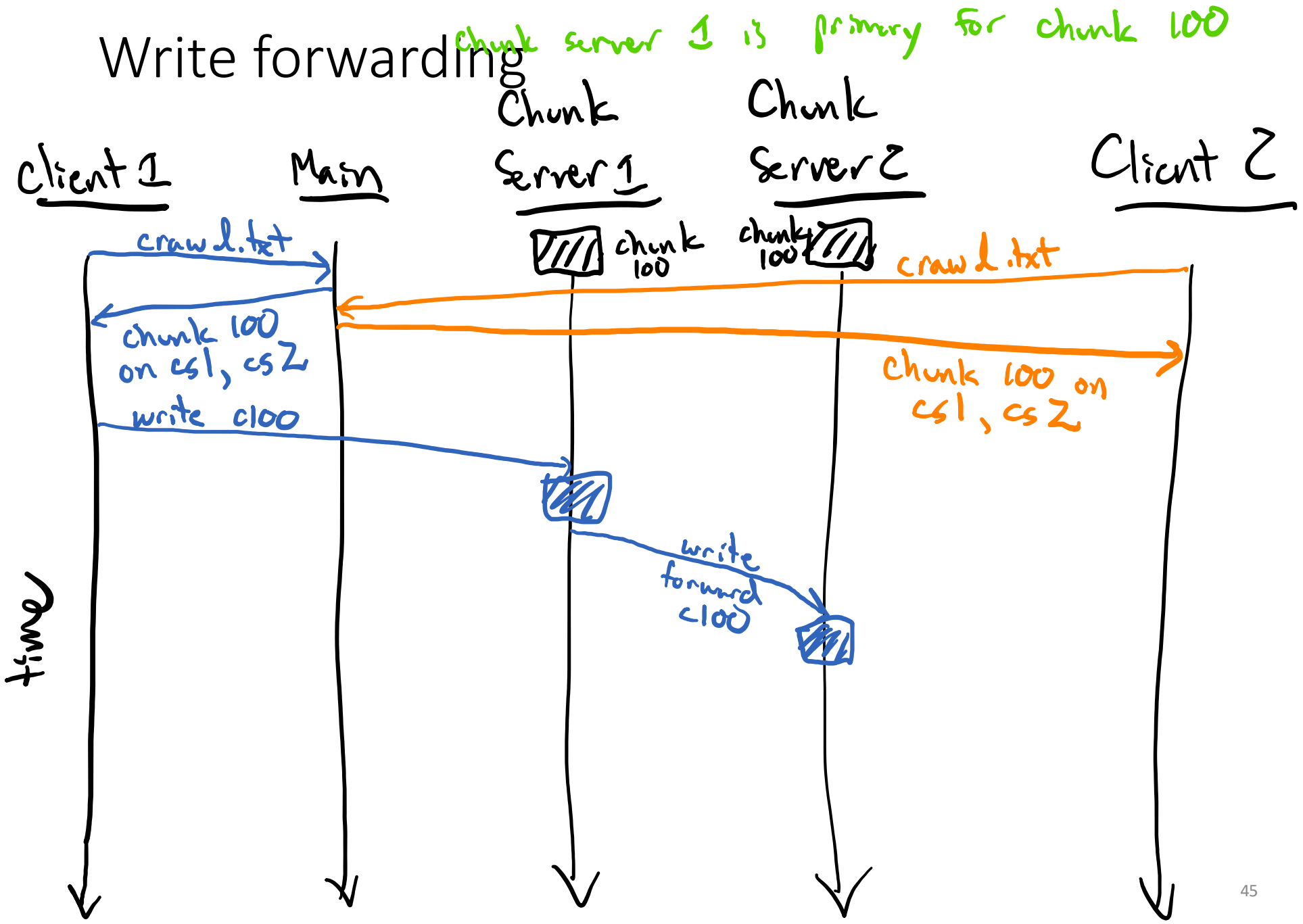
Write forwarding



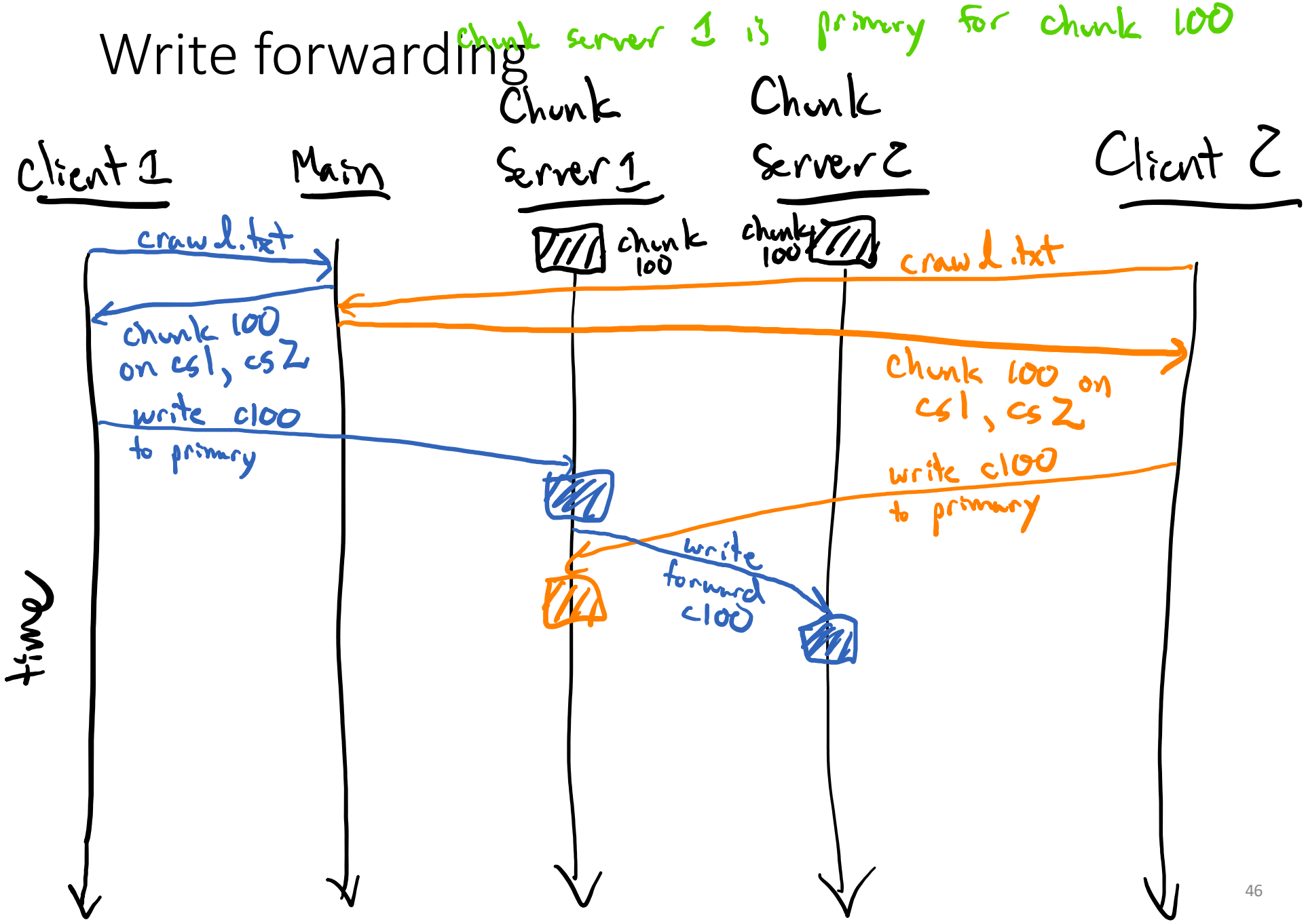
Write forwarding



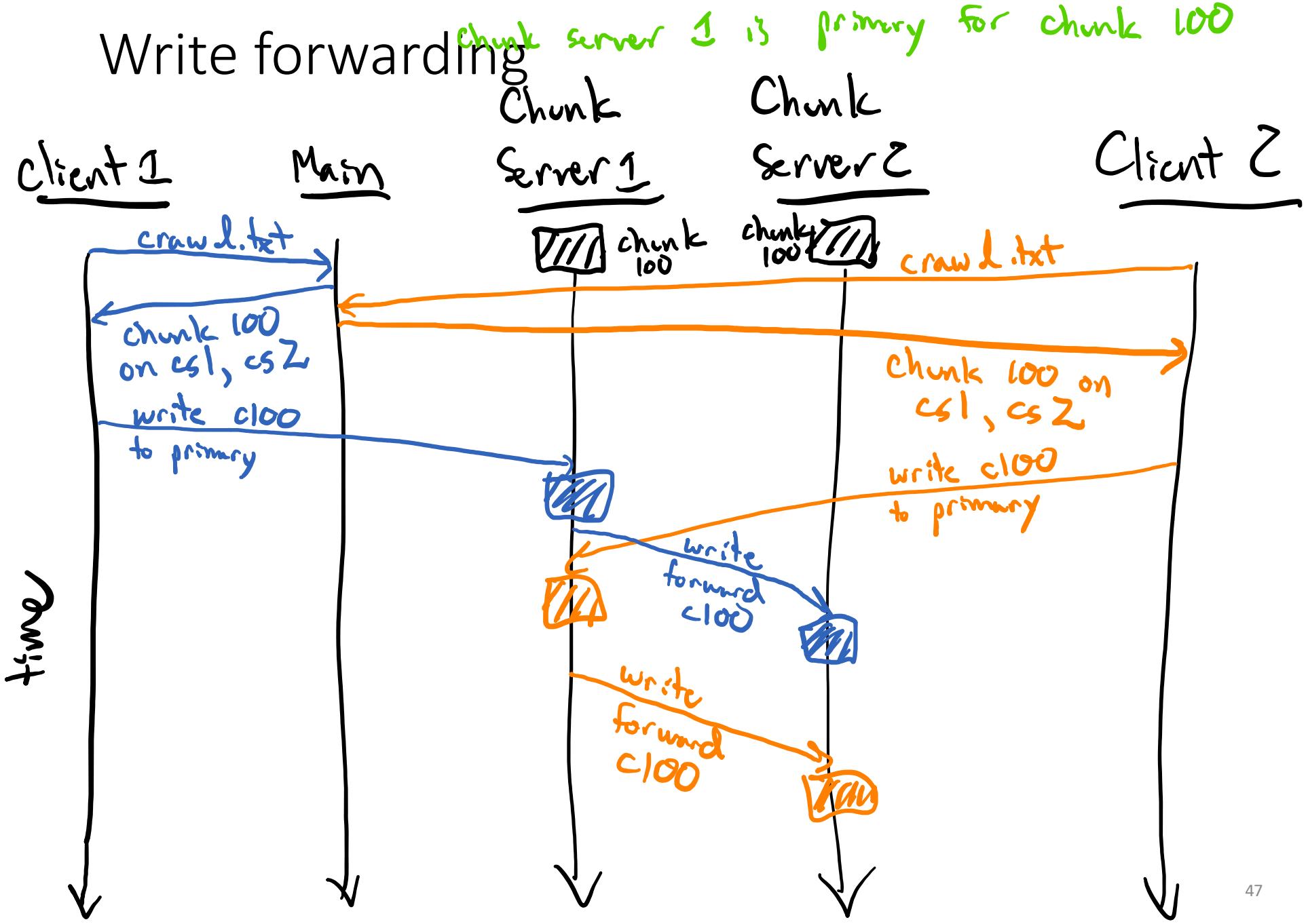
Write forwarding



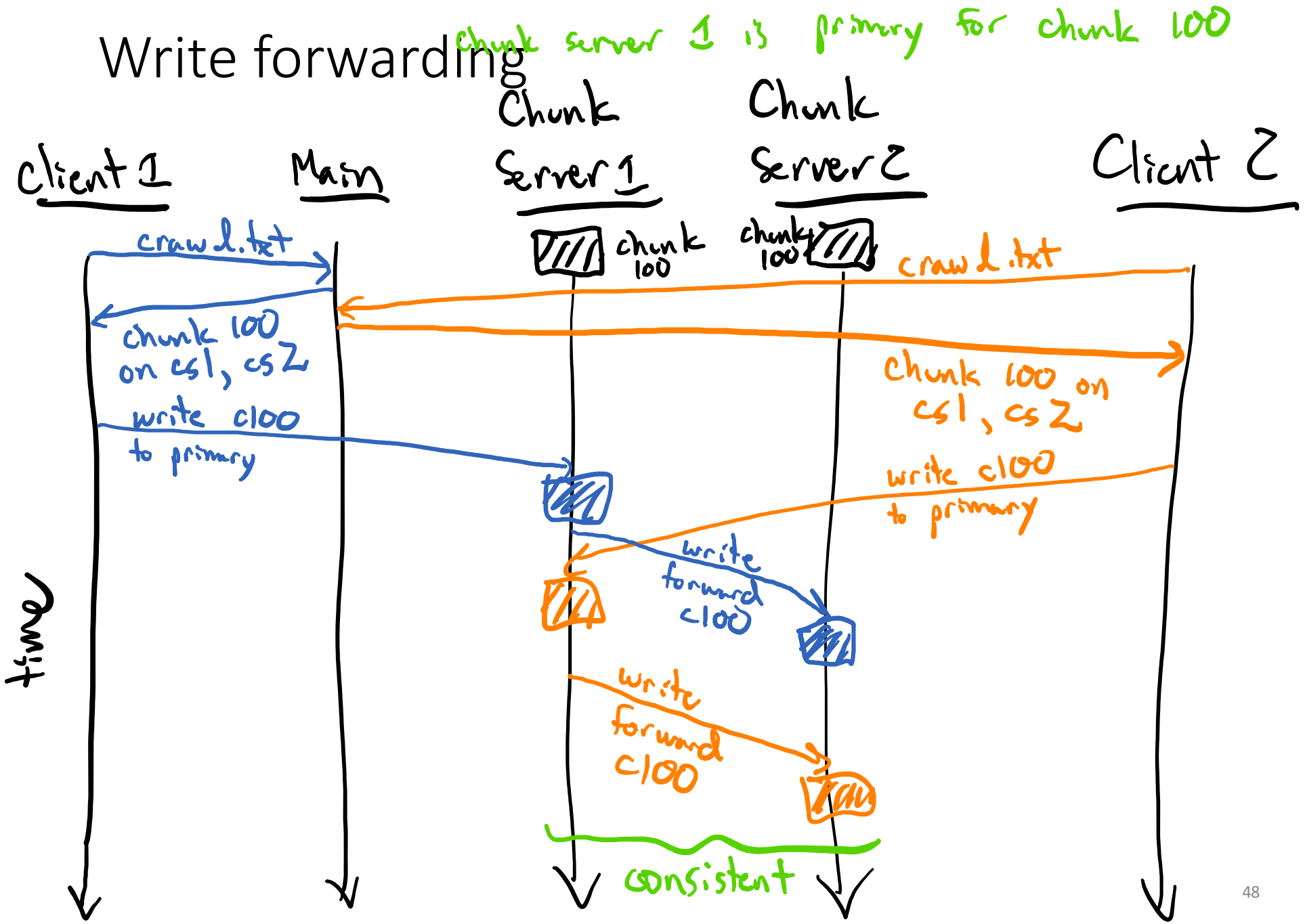
Write forwarding



Write forwarding



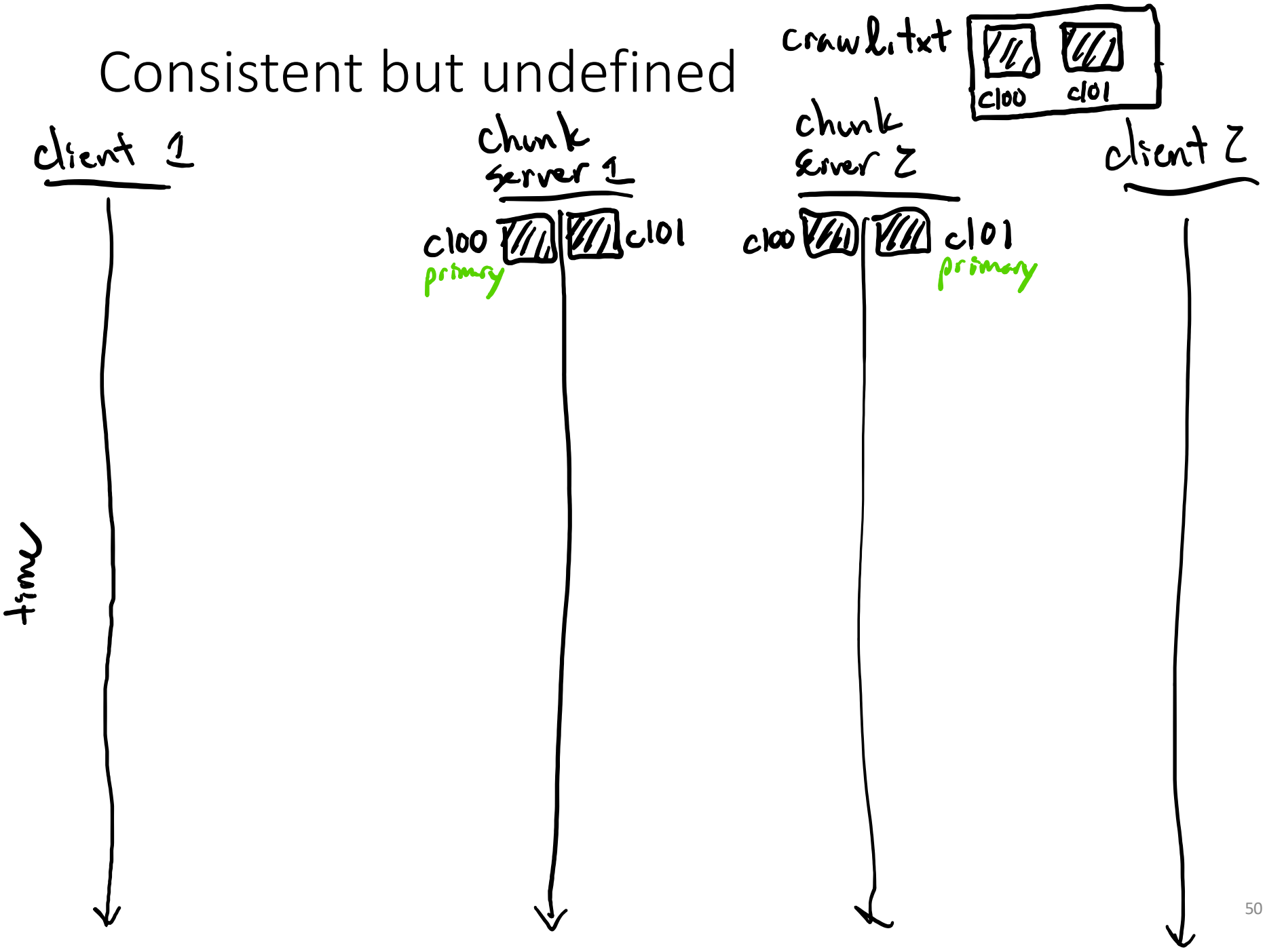
Write forwarding



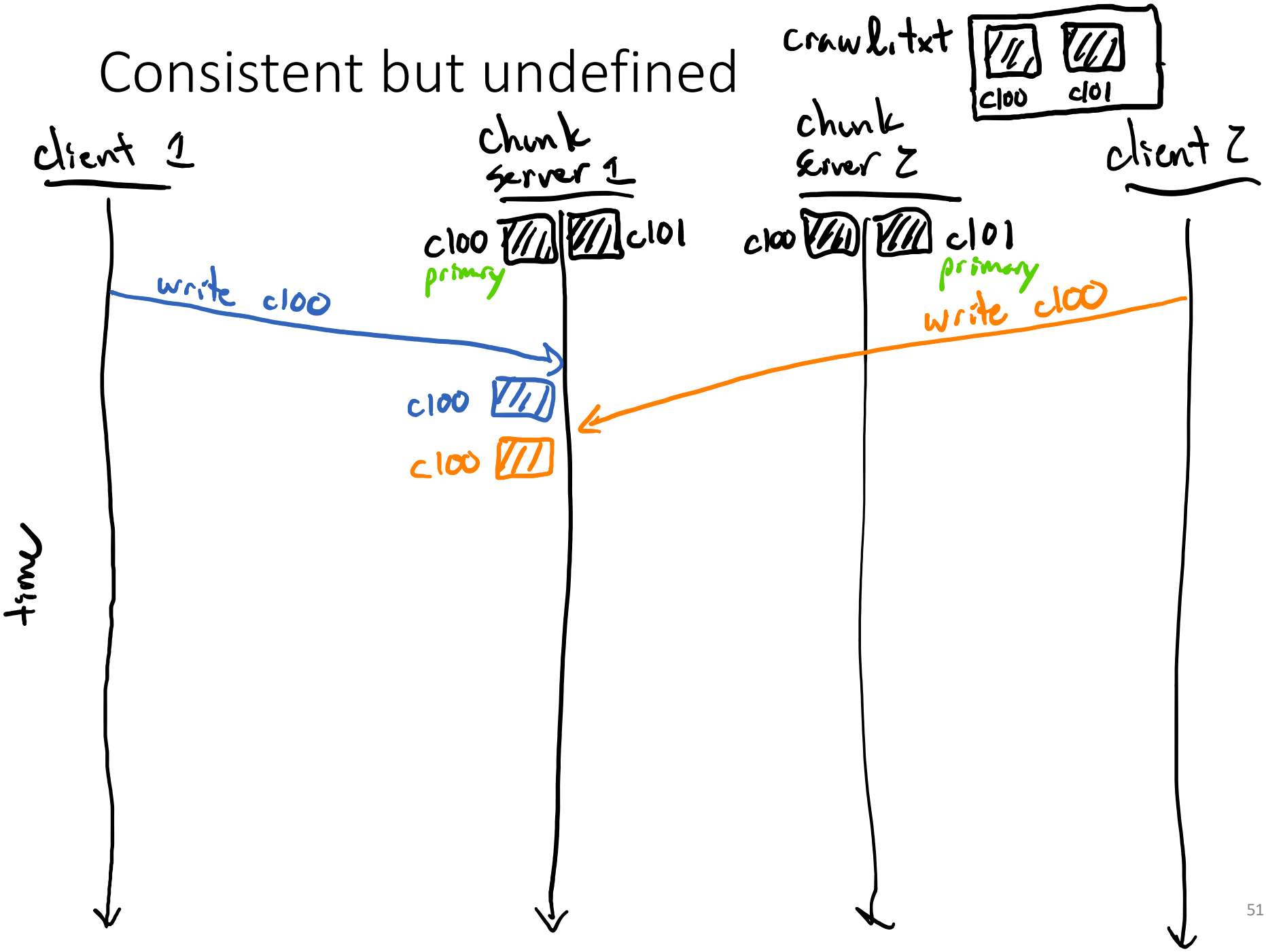
Properties: consistent, defined

- *Consistent*: Every chunkserver has the same value for the same chunk
- *Defined*: Changes to different chunks of the same file from different writers did not get mixed together
- How do undefined values happen?
- Simultaneous writes to different chunks in the same region of one file

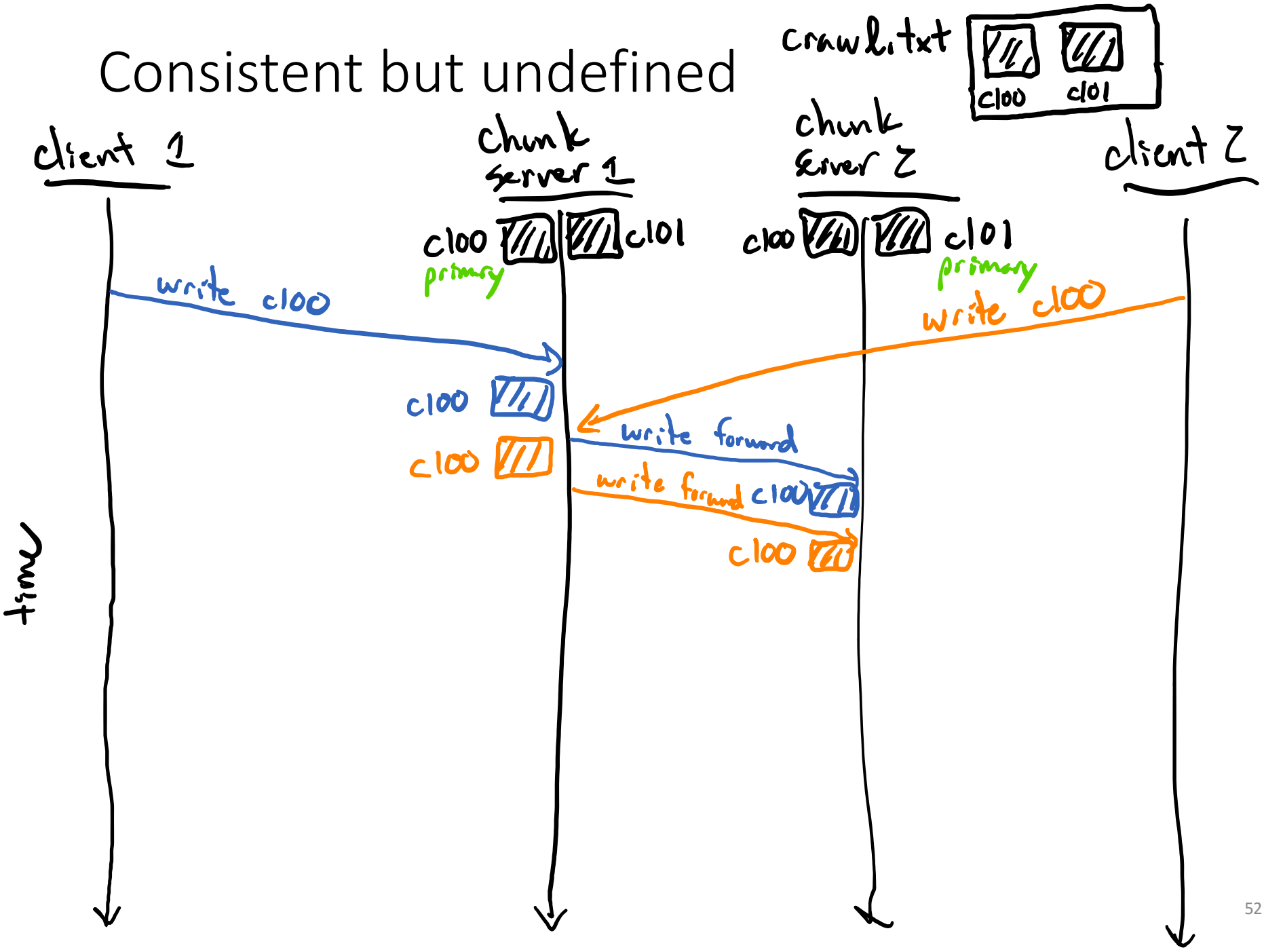
Consistent but undefined



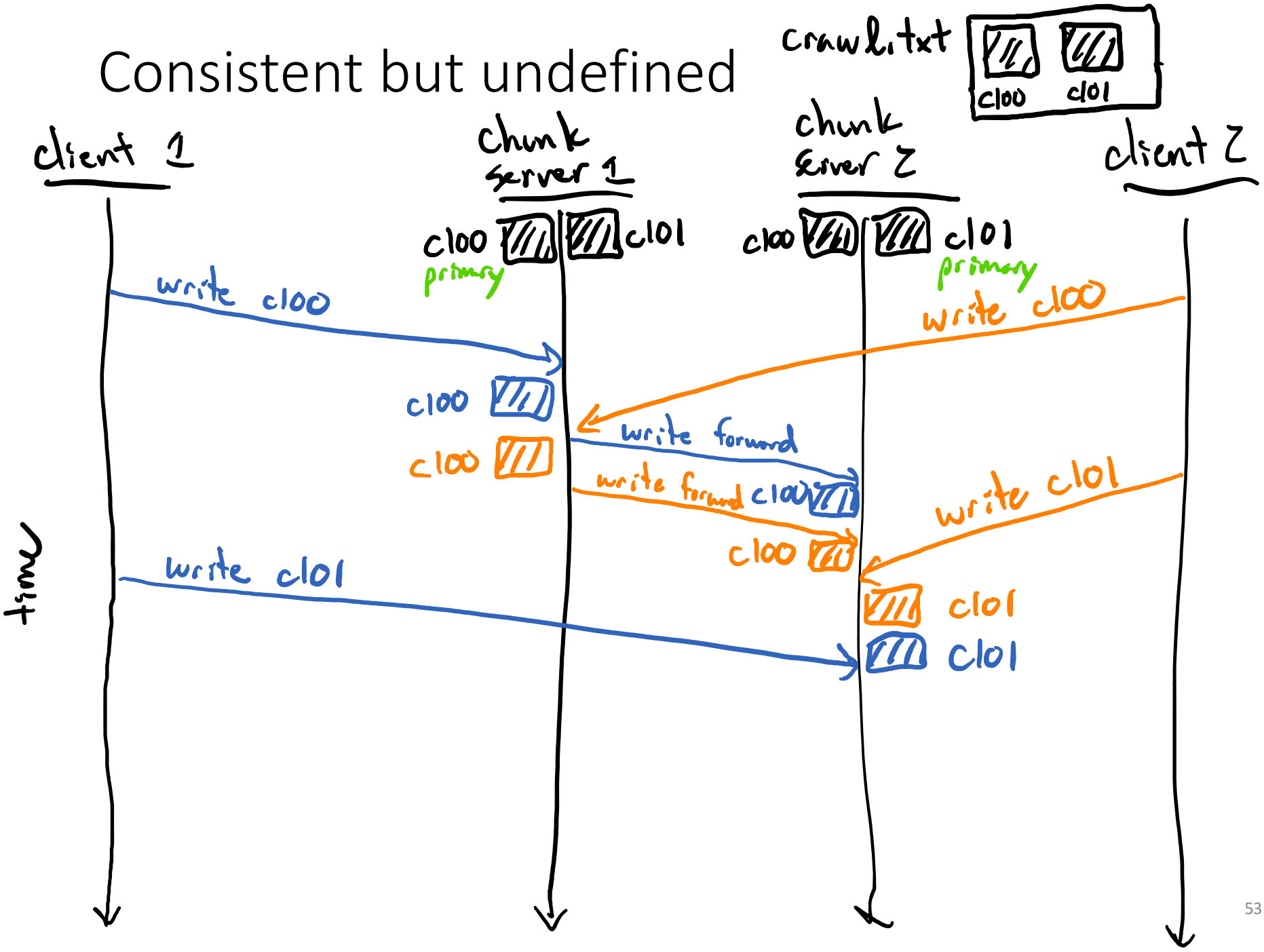
Consistent but undefined



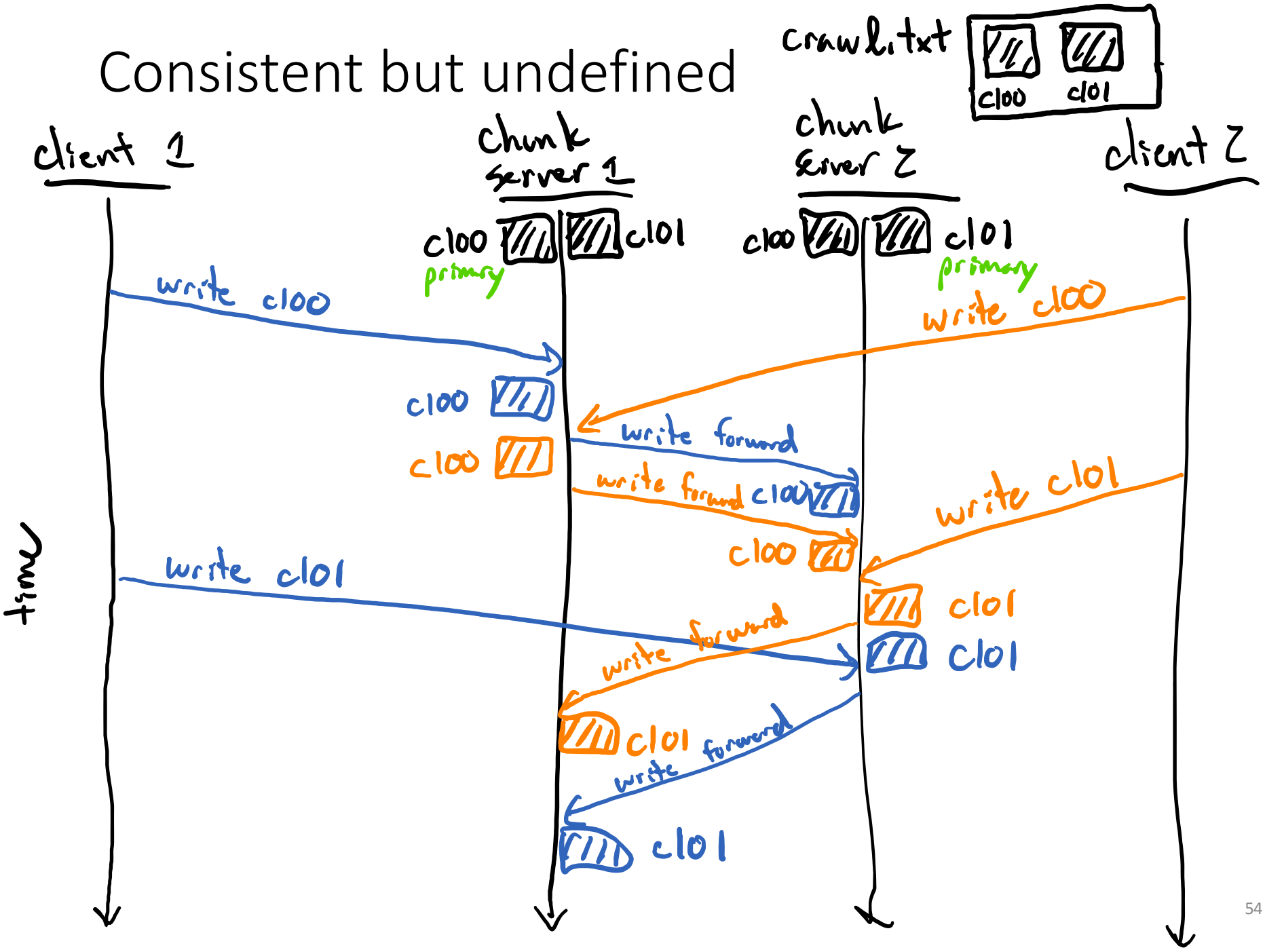
Consistent but undefined



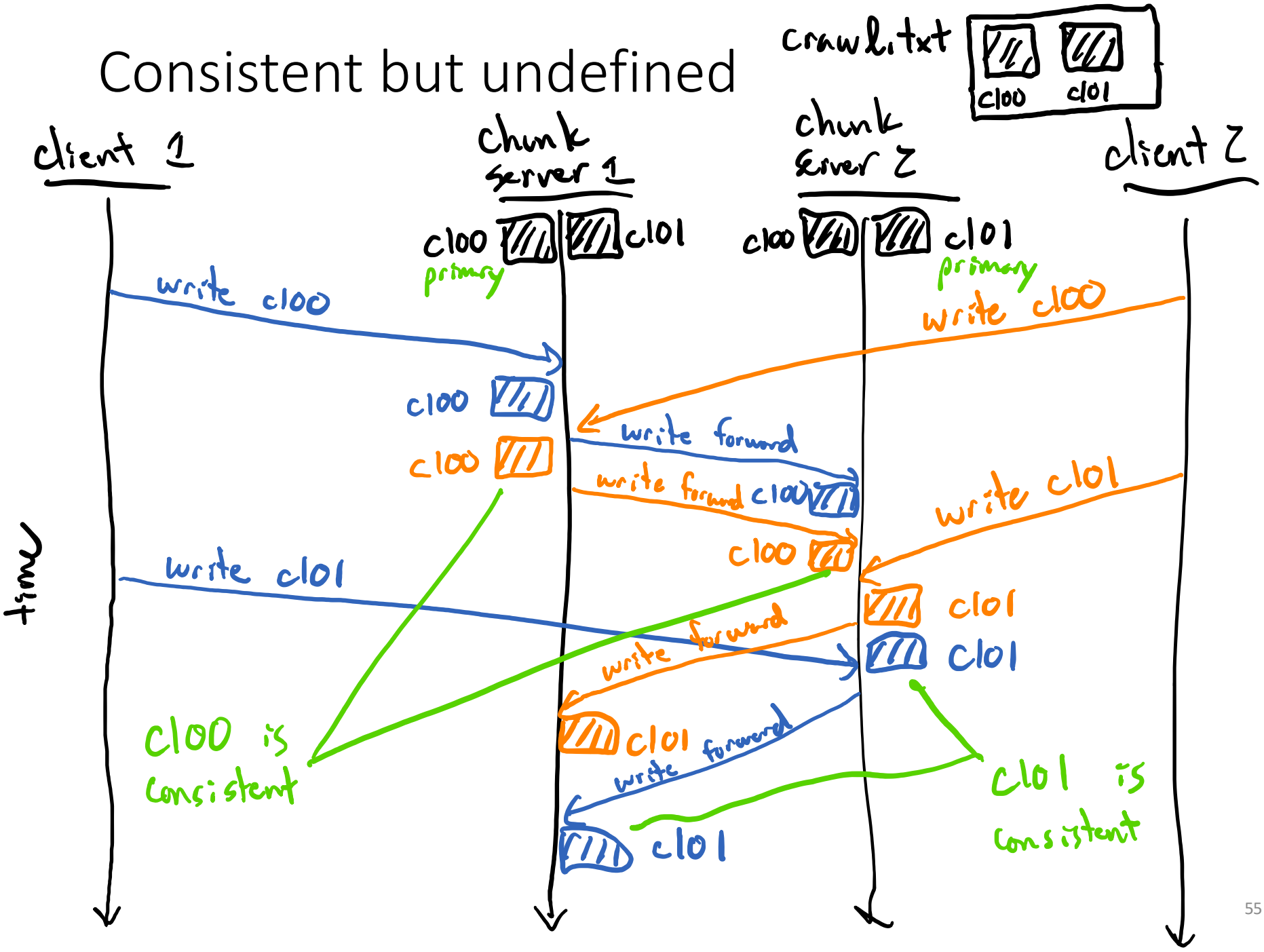
Consistent but undefined



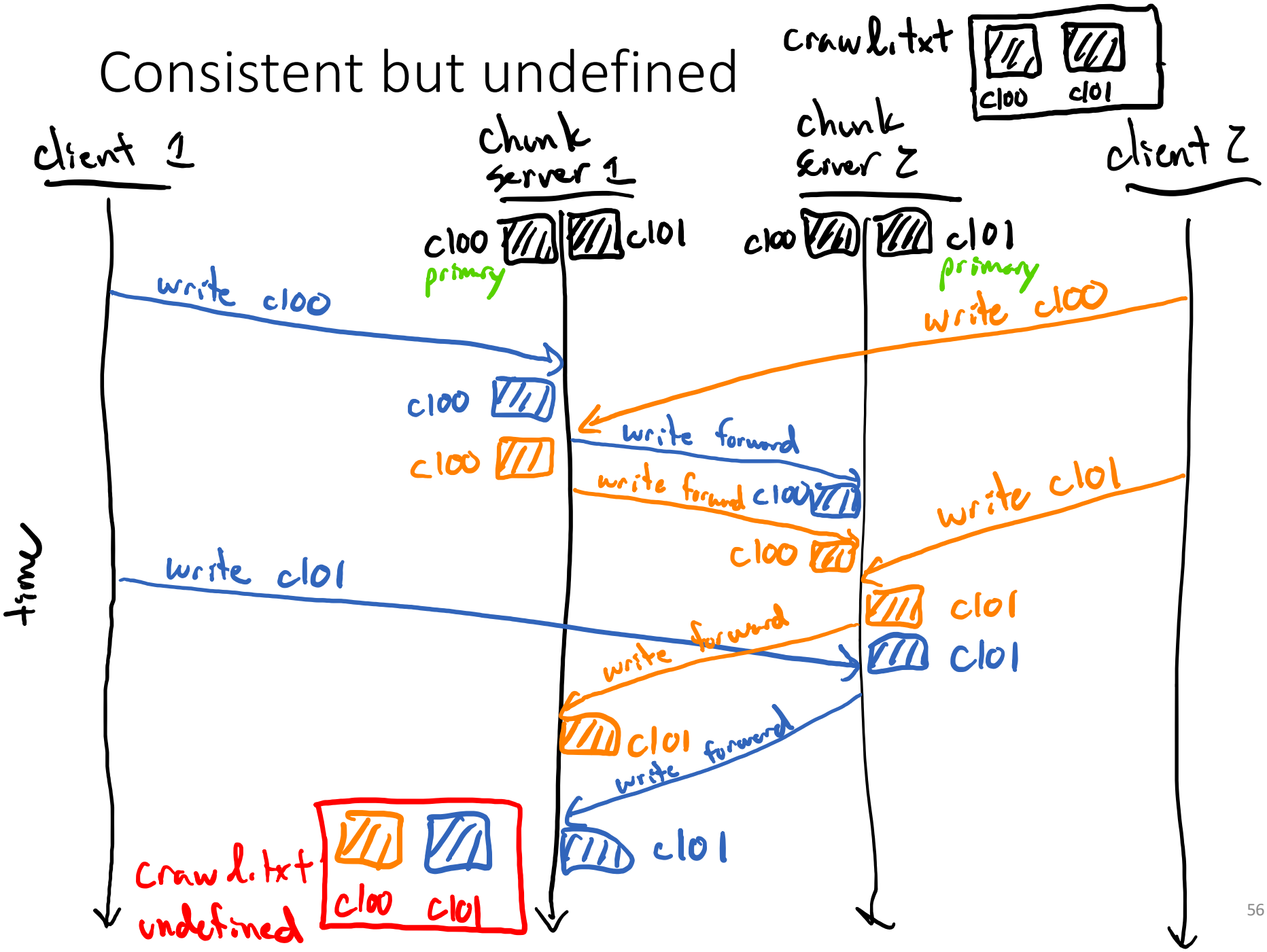
Consistent but undefined



Consistent but undefined



Consistent but undefined



Sequential consistency

- How do we fix consistent but undefined values?

- Option 1: Main server coordinates writes
  - Easy to implement
  - Slow
- Option 2: Distributed write coordination
  - Really, *really* hard to implement
  - Kinda slow

- *Sequential consistency:* Concurrent writes get the same answer as sequential writes, every time

Relaxed consistency

- How do we fix consistent but undefined values?
- Another solution: Don't bother

- *Relaxed consistency:* Sacrifice some correctness for performance

Relaxed consistency and GFS

- GFS made a deliberate design decision **not** to provide *sequential consistency*
- GFS provides *relaxed consistency*

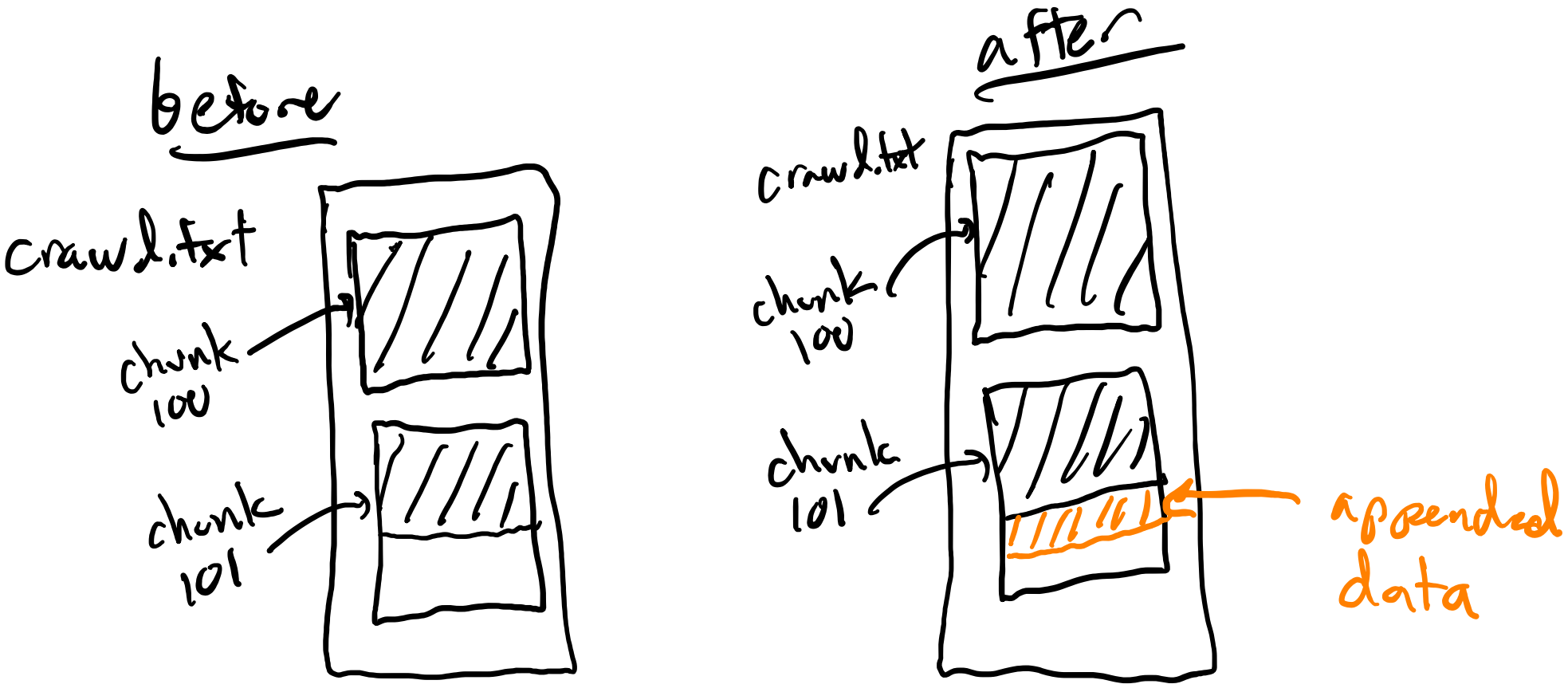
- What should the programmer using GFS do?
- Option 1: Application detects and corrects undefined file regions
- Option 2: GFS Atomic Record Append

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- GFS design
- Read
- Write
  - Consistency
- **Append**
- Fault tolerance
- Summary

Record append

- Append: add to the end of the file



Atomic operation

- An *atomic* operation appears to the rest of the system as if it happened instantaneously
- Avoid consistency problems

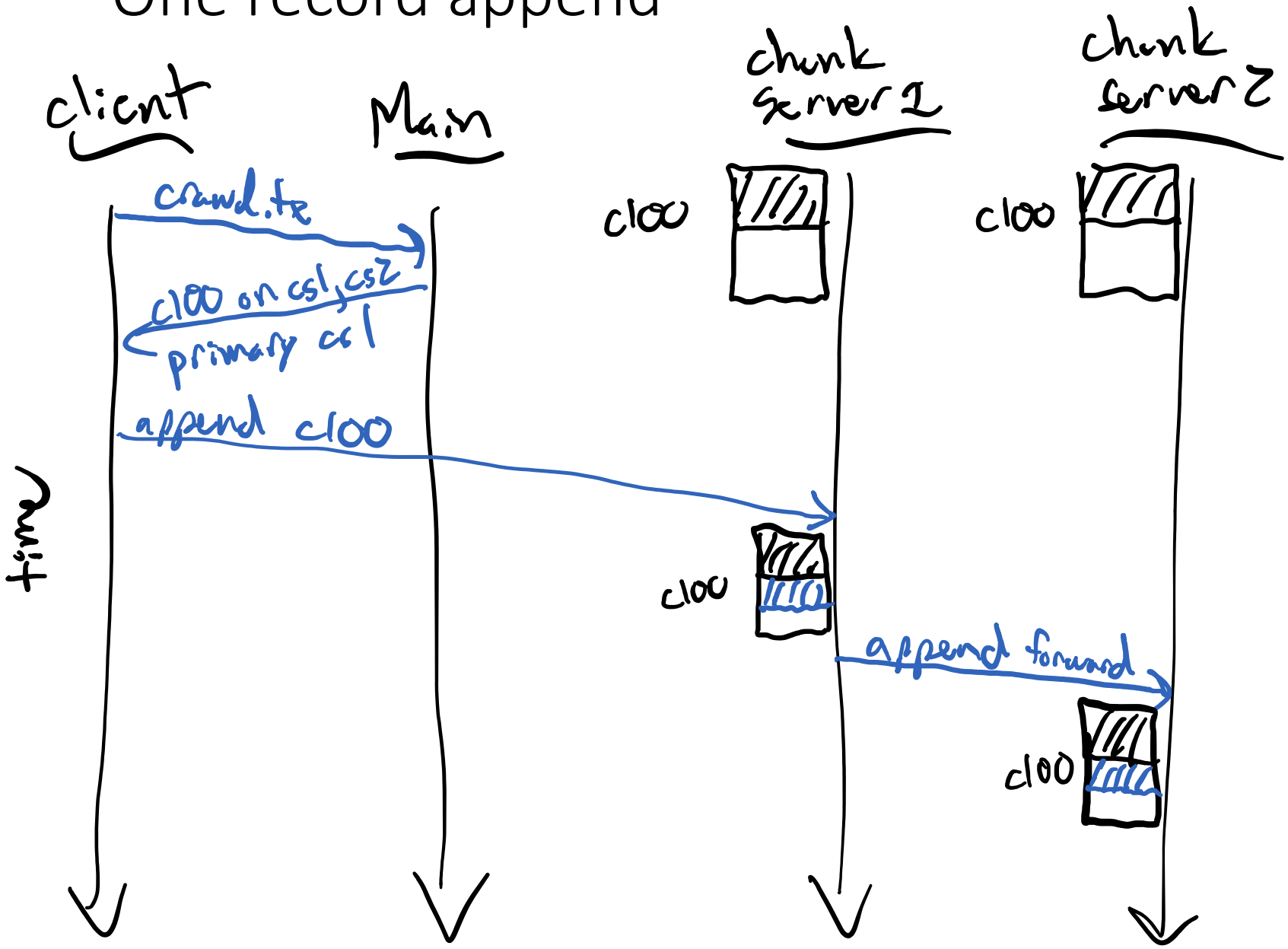
GFS atomic record append

- Insight: Record append only modifies one chunk
- Already solved the problem of consistency for one chunk
  - Write forwarding

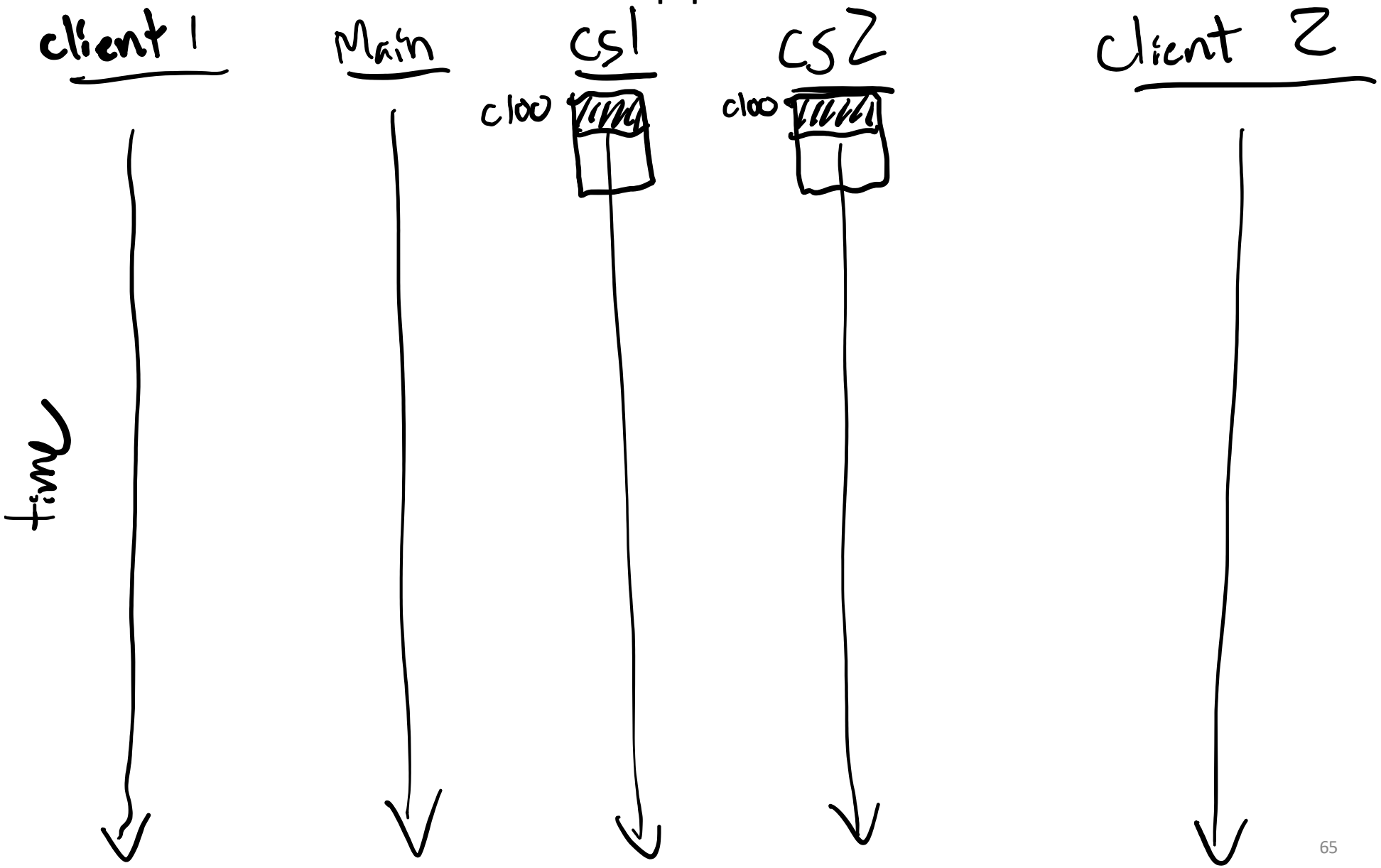
- Record append uses a similar strategy as write forwarding
- One chunkserver is designated as primary for each chunk id
- Primary forwards appends to replica chunkservers*

* Simplification, see paper for full details.

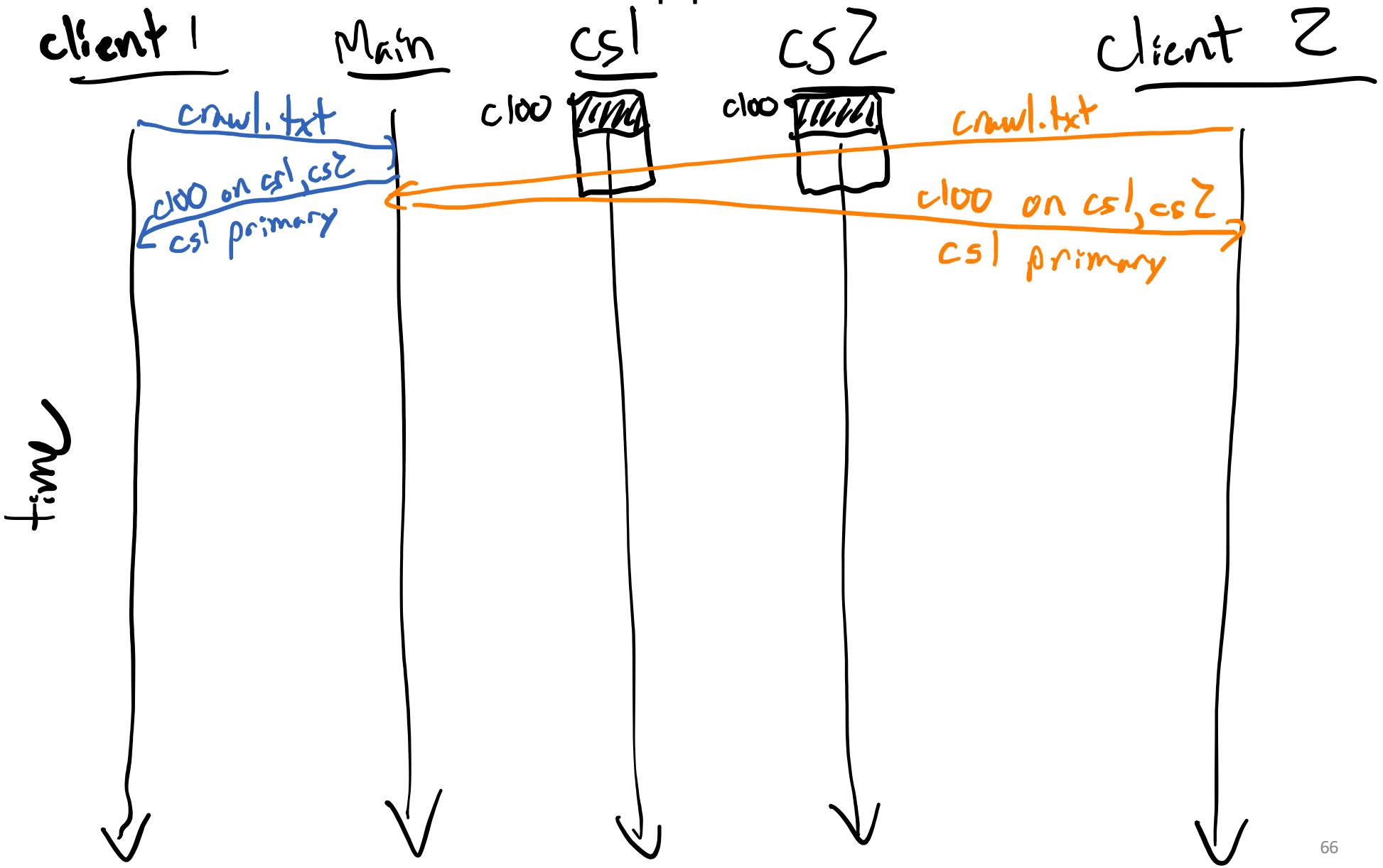
One record append



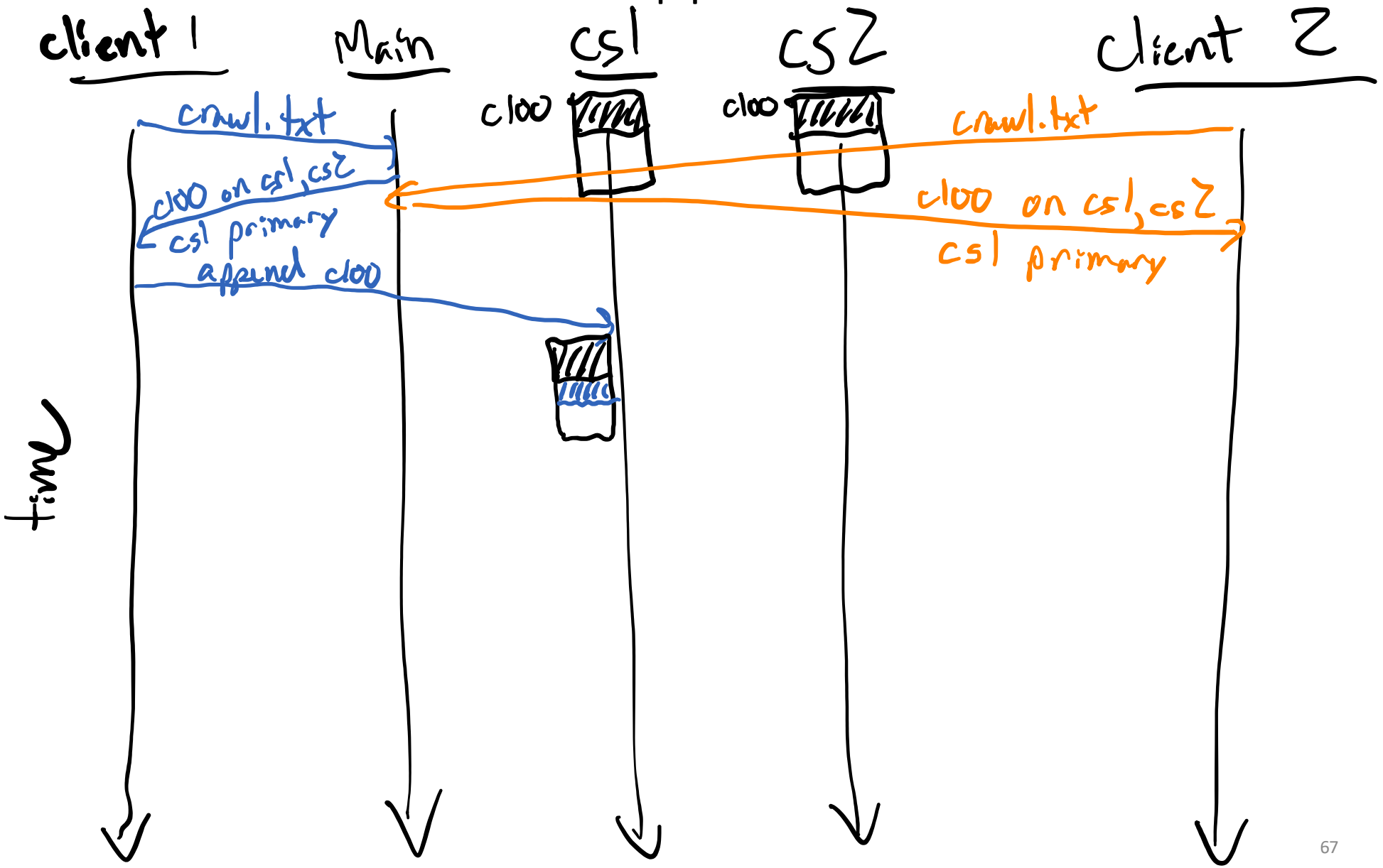
Concurrent record appends



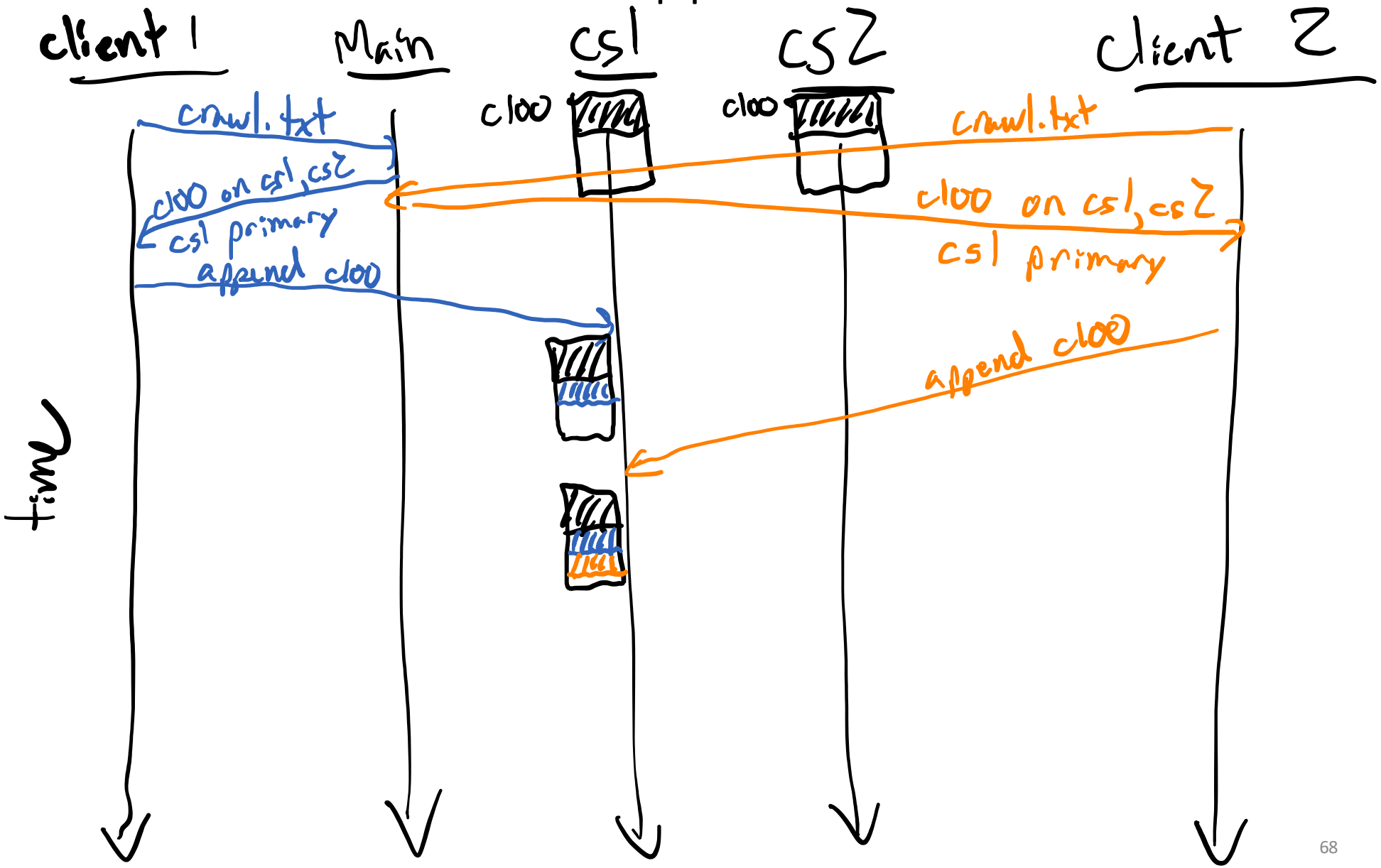
Concurrent record appends



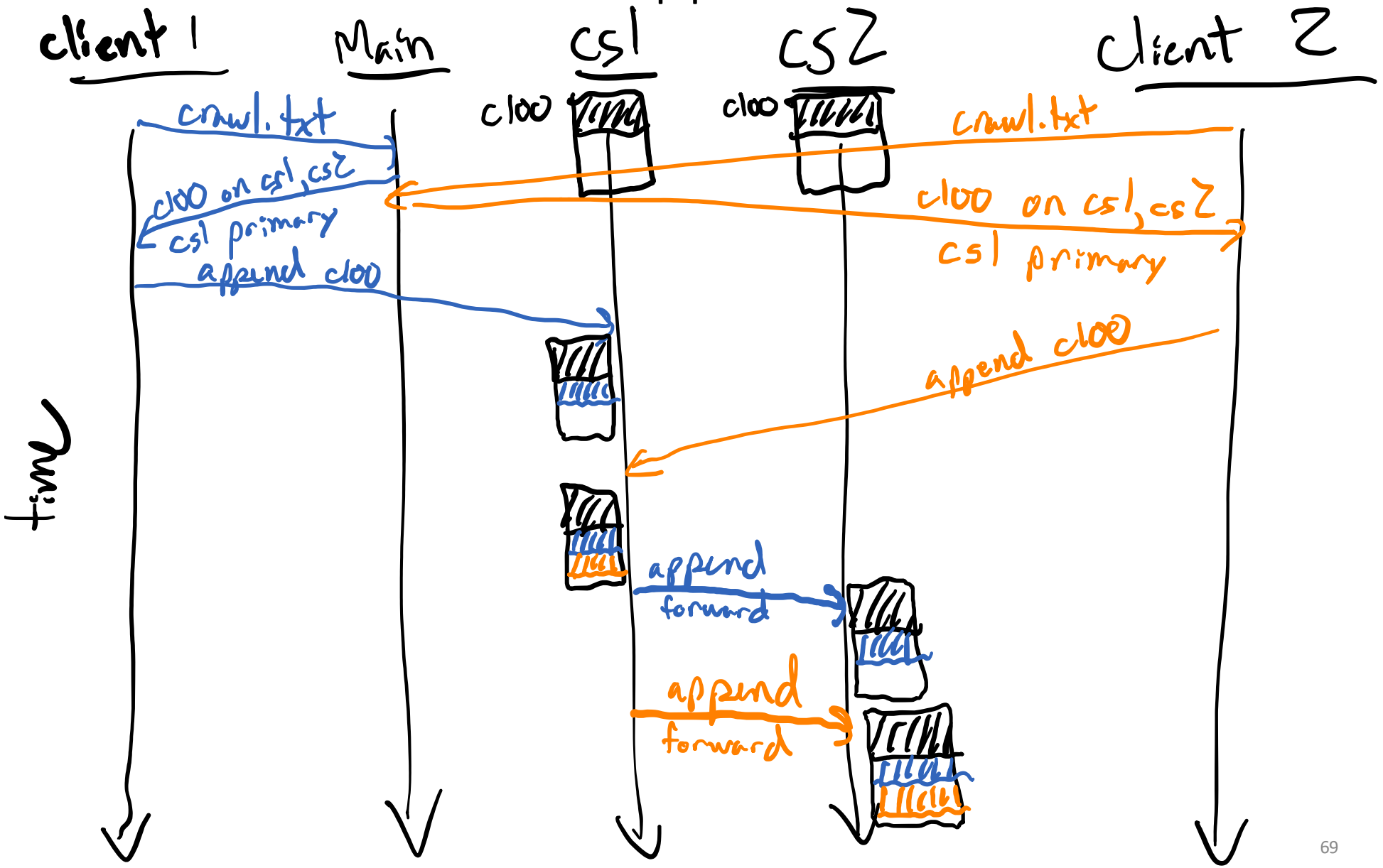
Concurrent record appends



Concurrent record appends



Concurrent record appends



GFS record append subtleties

- What if the new data is bigger than one chunk?
  - Not allowed

- What if the new data would cross a chunk boundary?
  - Pad the last chunk to "fill it up"
  - Reply to the client to try again

Optimize for append

- GFS is optimized for fast, atomic append
  - Unique to GFS

- Google applications that use GFS optimized to use append
- Example: Web crawling
  - Download a page and append it to crawl.txt

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- GFS design
- Read
- Write
  - Consistency
- Append
- **Fault tolerance**
- Summary

Fault tolerance

- Failed chunkserver
- Failed main server

Chunkserver fault tolerance

- Chunkservers report to main server every few seconds
  - *Heartbeat* message
- If the main loses the heartbeat, marks the server as down
- Main asks chunkservers to reduplicate data
- Typically 3 copies of every chunk

Main failure

- Main maintains critical data structures
  - filename -> chunkid map
  - chunkid -> location map
- Main writes a log to disk when data structures change

- **Shadow main** consumes log and keeps copies of these data structures up-to-date
- If main goes down, shadow main read-only access until main restart

Reliability is cool

- "In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process."
- Reliability is cool!

Agenda

- Review: Distributed systems introduction
  - Google File System motivation
- Read
- Consistency
- Write
  - Consistency
- Append
- Fault tolerance
- **Summary**

GFS strengths

- Store lots of data
- Fault tolerant
  - Internal chunk copies -> "backups" baked into the design
- High *throughput*
  - Can read lots of data from many chunkservers at same time

GFS weaknesses

- Bad for small files
  - Chunks are ~64MB
- Main node single point of failure
- High *latency*
  - Talk to two servers to fetch any data, might need multiple chunks

Distributed main servers

- Why didn't GFS use distributed main servers in the first place?
- Distributed main servers requires a distributed set of machines electing a new main node when the old one fails, in the face of a failing or partitioned network
  - This problem of *consensus* is difficult

Throughput vs. latency

- Good for batch applications
  - Web crawling
  - Web indexing
  - High throughput is important
- Bad for real-time applications
  - Gmail
  - Google Docs
  - YouTube
  - Low latency is important
- Throughput vs. Latency tradeoff

Traditional vs. Google File System

Traditional file system

- Organize files across one hard drive in one computer

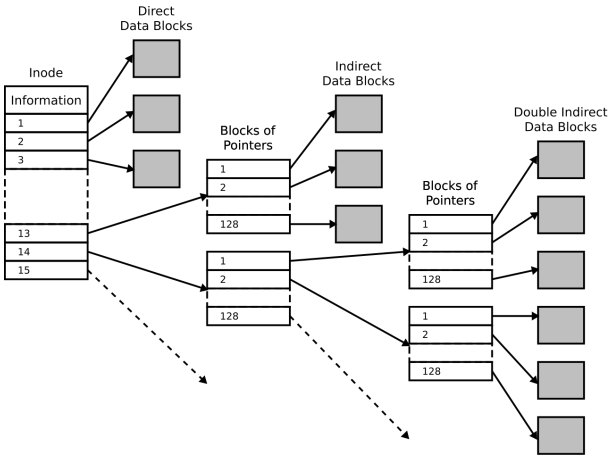
Google File System

- Organize files across many hard drives in many networked computers

Traditional vs. Google File System

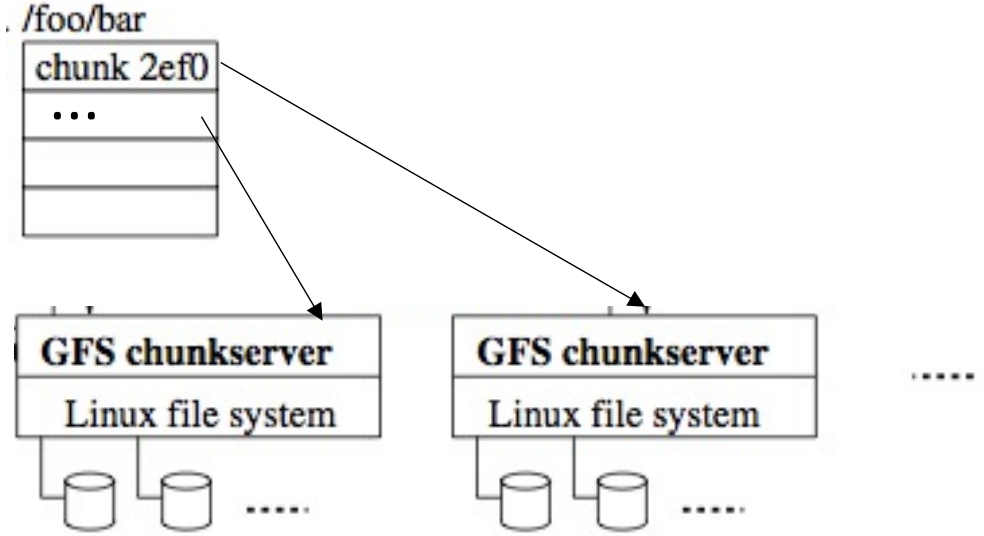
Traditional file system

- Divide file into ~kB blocks
- Place blocks on one HDD in one computer



Google File System

- Divide file into ~MB chunks
- Place chunks on different HDD in different computers



GFS summary

- Goal: store more data than fits on one computer

- Use lots of cheap, unreliable servers
- Use software to make them look reliable

- Optimized for atomic record append
- Optimized for throughput, not latency

Further reading

- The Google File System Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
  - http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
- Storage Architecture and Challenges
  - http://static.googleusercontent.com/media/research.google.com/en//university/relations/facultysummit2010/storage_architecture_and_challenges.pdf