Strong Data Consistency

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Course Topics

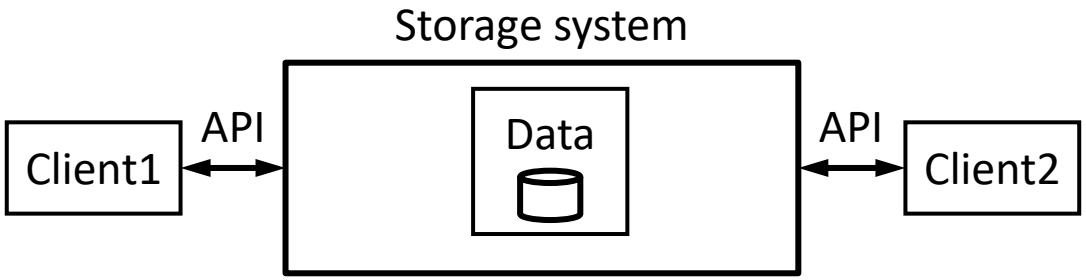
- Introduction
- Programming distributed systems
- Highly consistent, replicated systems
  - Strong data consistency
  - Replication
  - Consensus
  - Case study 1: Consensus in Raft
  - Case study 2: Coordination with ZooKeeper
- Highly available and scalable systems
- Transactional systems
- Byzantine systems

Overview

- Storage system and data consistency
- Linearizability
- Implementing linearizability
- Sequential consistency

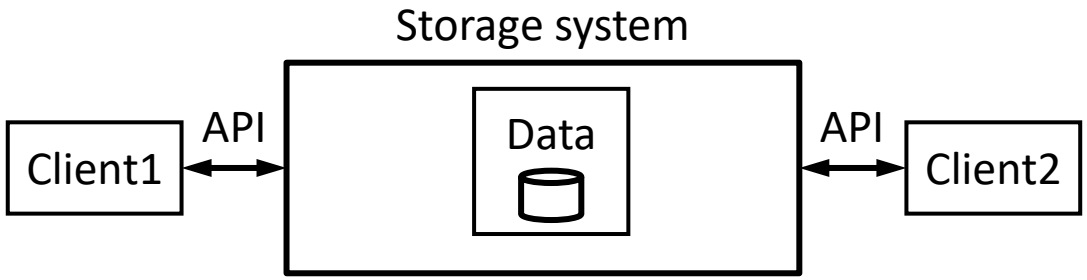
Storage system

- Assume that networked clients access a storage system
- Storage system API provides read/write operations
  - Block store: read/write fixed-size blocks
  - File system: read/write byte range within variable-sized files
  - Key-value store: get/put key-value pairs
  - Databases: read/update rows of tables



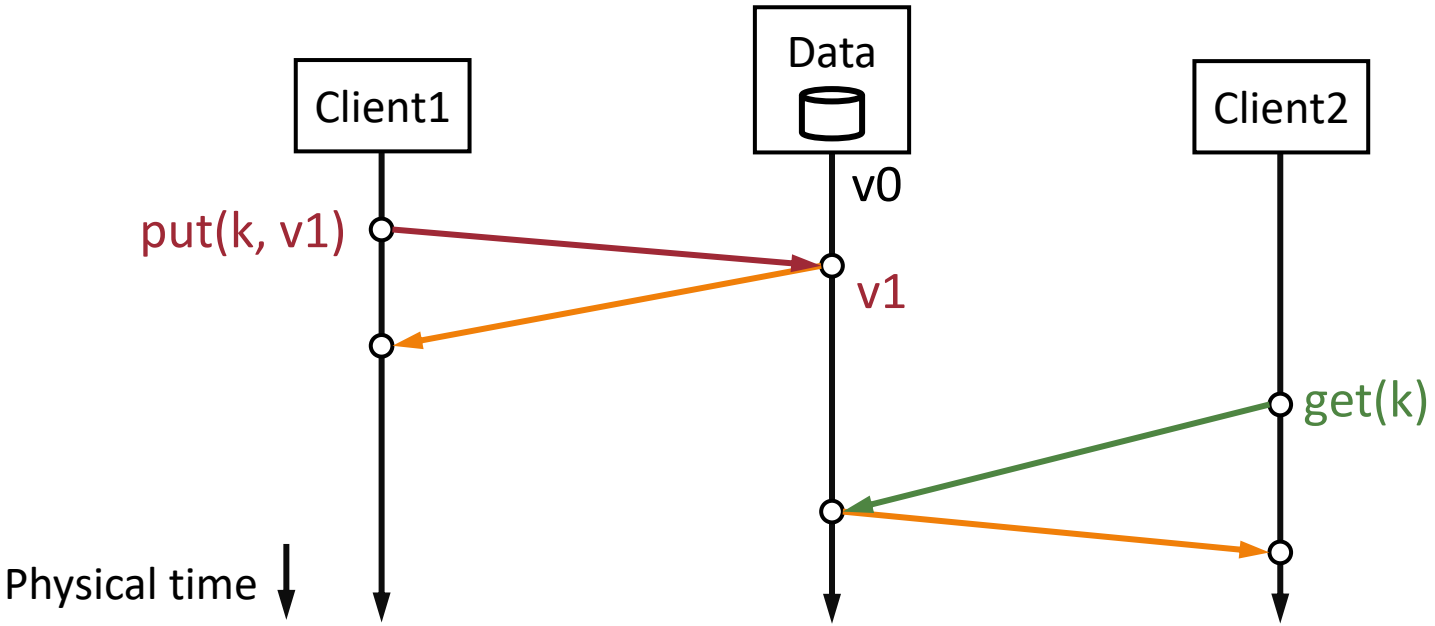
Clients of storage system

- Clients can be either end-users or other services
  - Users directly accessing cloud storage
  - Video delivery service storing data at a separate storage service
- Sometimes clients may be co-located with the storage system on the same physical servers
  - May allow optimizations, e.g., data placement



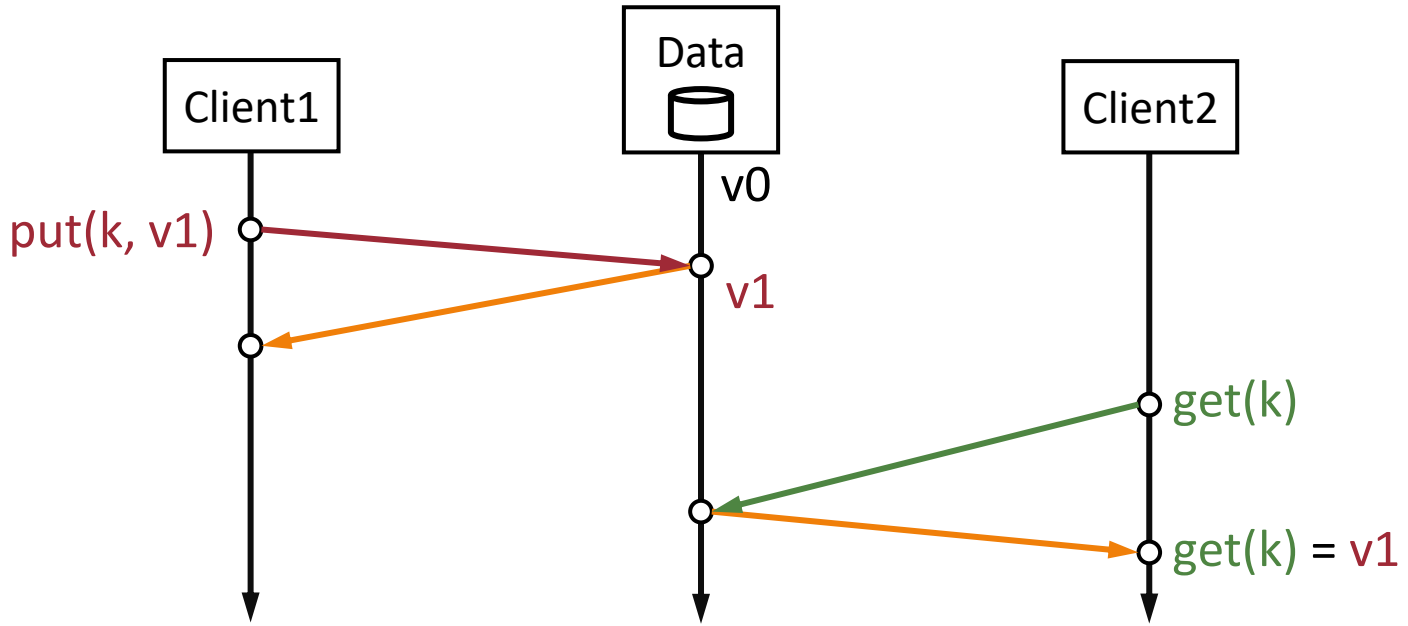
Expected behavior of storage system

- Say Client1 issues put(k, v1) and receives ack
- Then, Client 2 issues get(k)
  - What can the client expect get(k) to return?



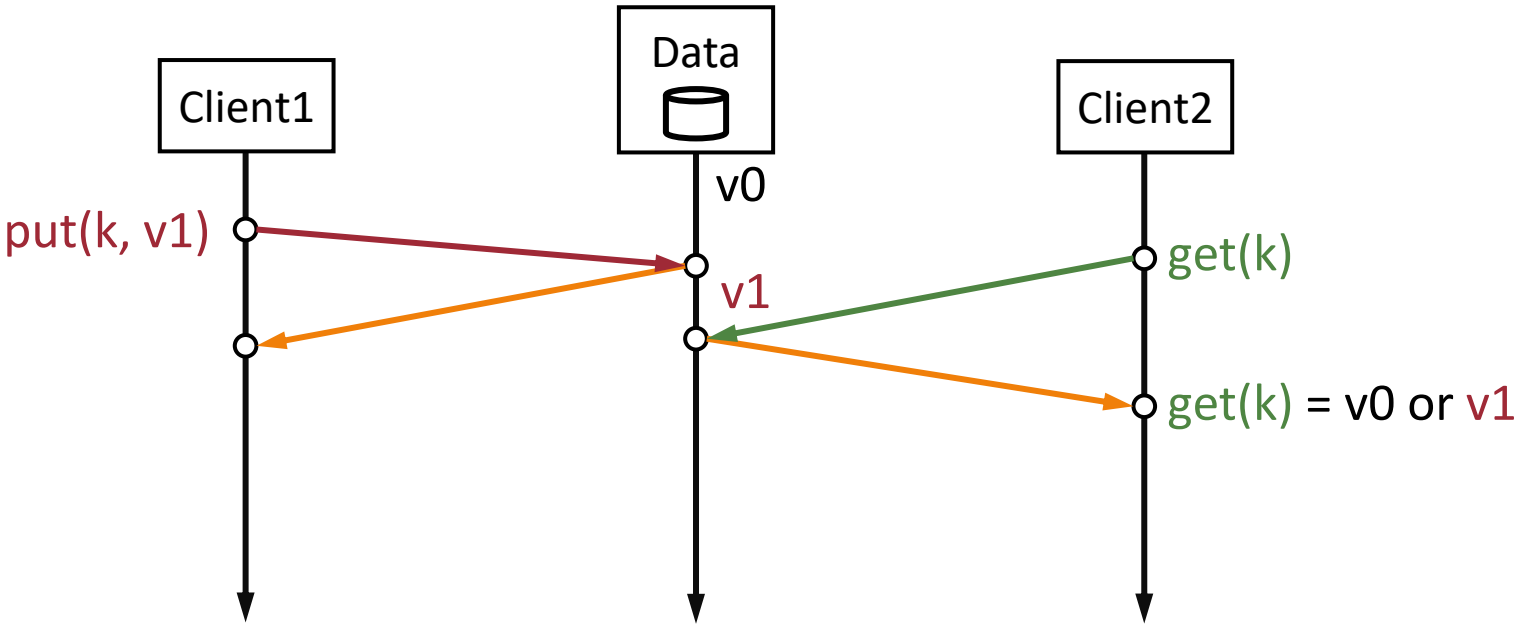
Ideal behavior of storage system

- Clients should read value written by most recent write
  - Regular, single-threaded programs expect this behavior when reading and writing from memory
- get(k) should return v1 (e.g., not v0)



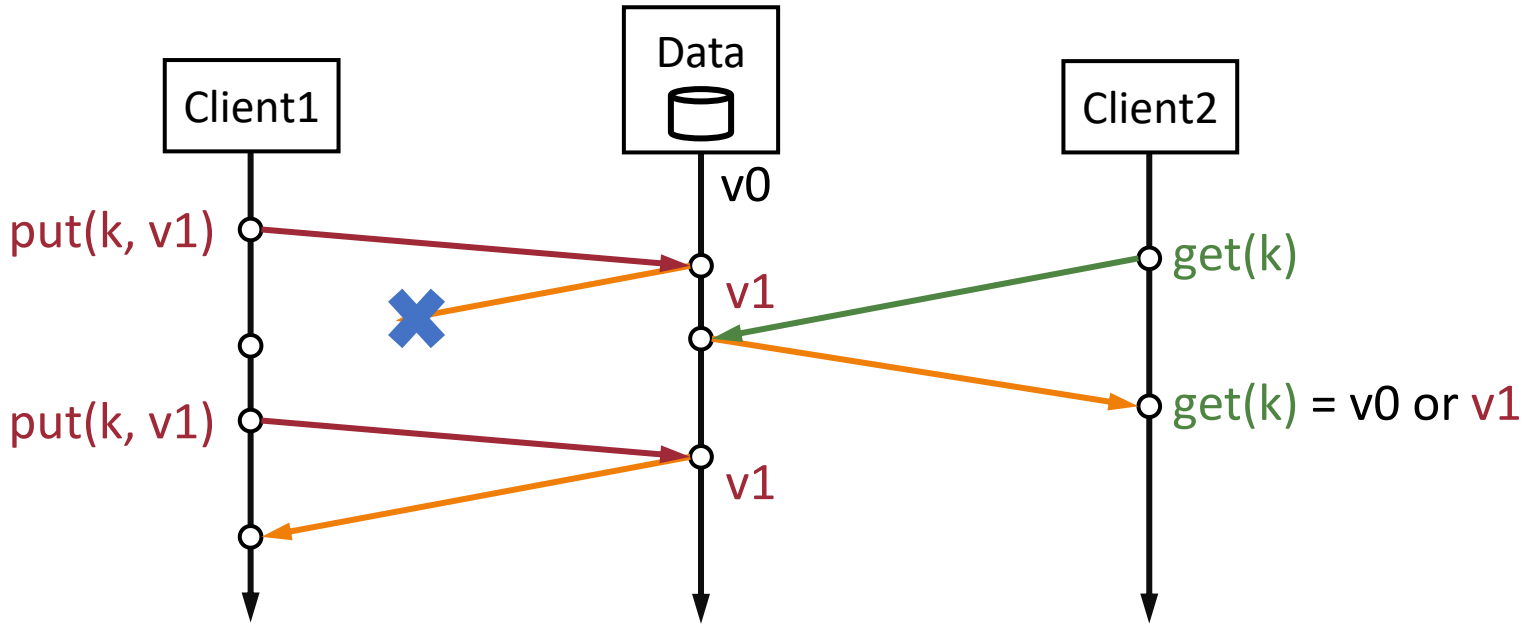
What should be expected behavior?

- Concurrent get()/put() operations (overlap in real time)
  - Should get() return v0 or v1?



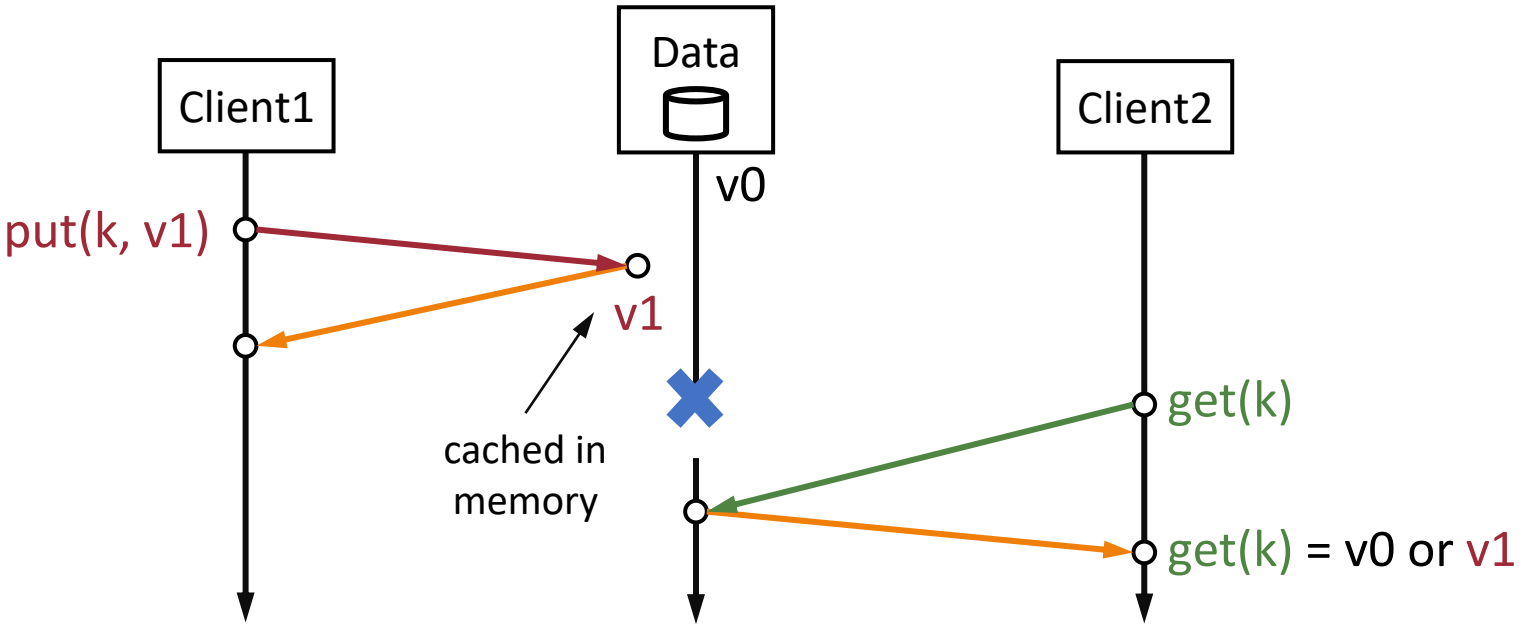
What should be expected behavior?

- Best-effort links lose, duplicate or reorder messages
  - Should get() return v0 or v1?



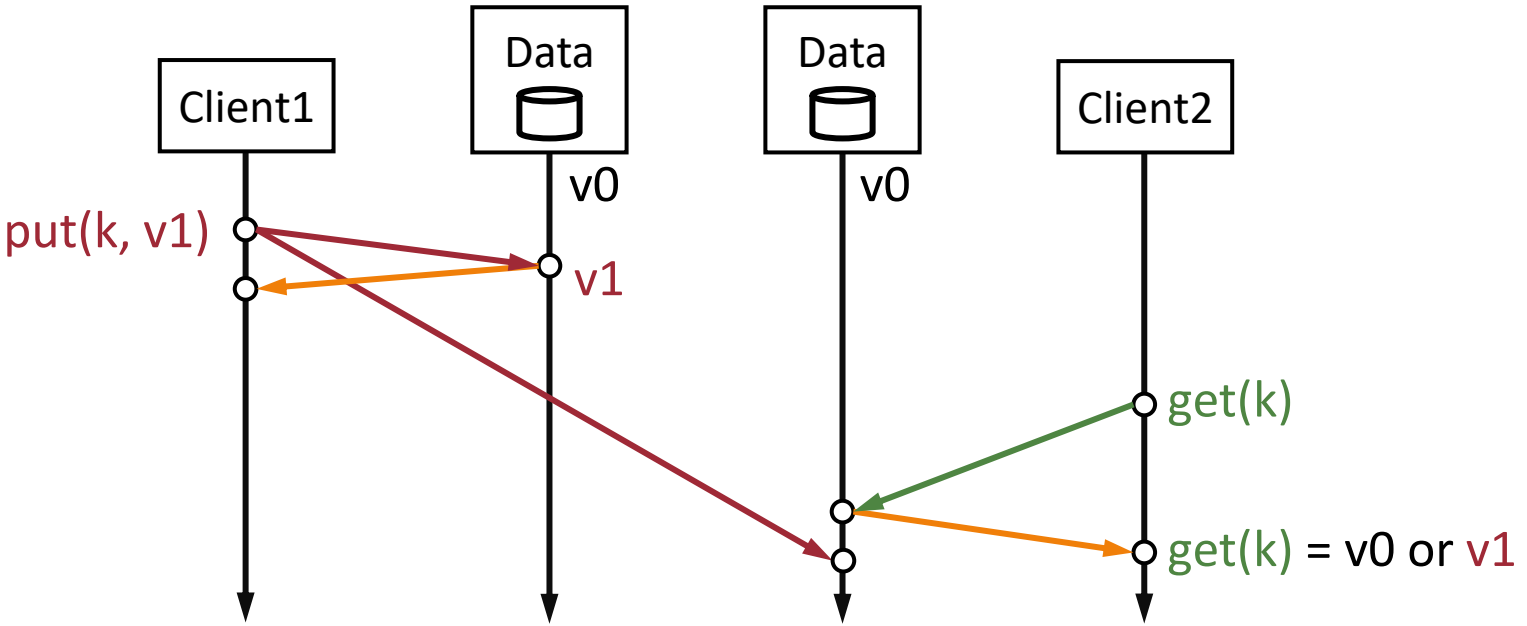
What should be expected behavior?

- Nodes crash and lose data on recovery
  - v1 is cached in memory, node crashes before it is stored on disk
  - Should get() return v0 or v1?



What should be expected behavior?

- Data is replicated at multiple locations
  - Should get() return v0 or v1?



Data consistency model

- A data consistency model is a specification (i.e., guarantee) that a storage system provides about expected behavior when clients access data
  - When clients issue get()/put(), what values can get() read?

Why data consistency model?

- Consider this simple coordination problem:

```
Coordinator:

put(config, “new config”)
put(config_done, TRUE)
```

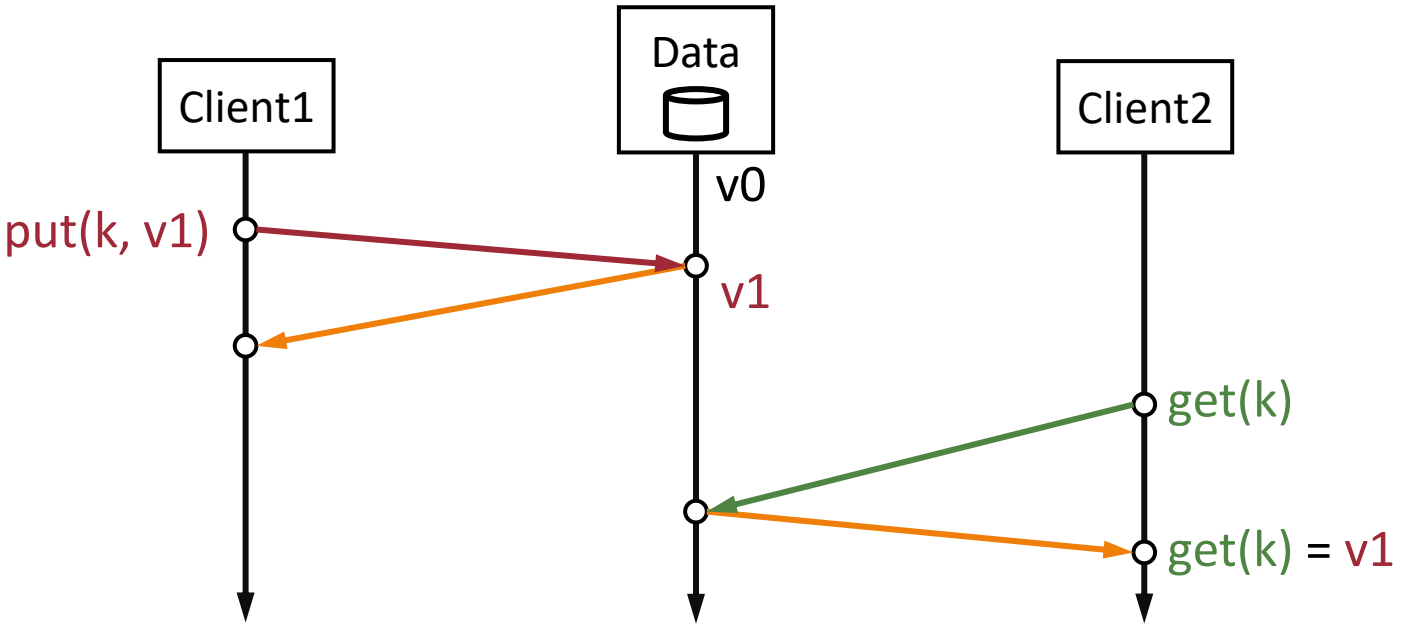
```
Clients:

while get(config_done) != TRUE:
    wait
get(config) // is it “new config”?
```

- For applications: what is correct behavior w/o any guarantees from storage system?
- For storage system: how to implement without a model?
  - Implementation involves complex interplay between concurrency, network model, node failure model, and replication…

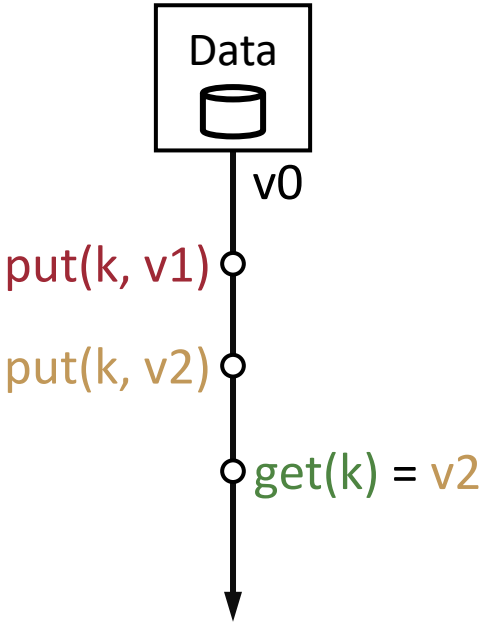
Understanding data consistency

- Recall, get() expects value written by most recent put()
  - get(k) should return v1 (e.g., not v0)
- But what does most recent mean?



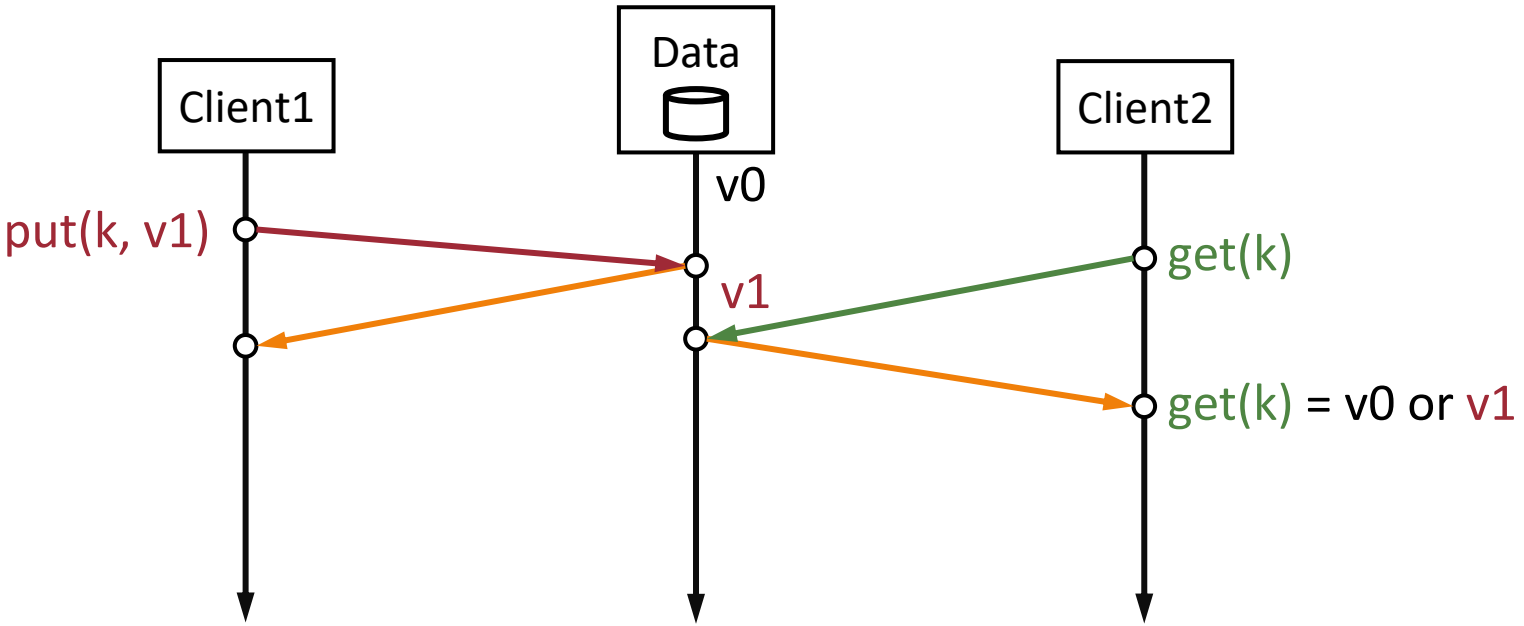
Understanding data consistency

- But what does most recent mean?
  - Intuitively, get()/put() operations can be totally ordered
  - On a single node, with a global clock, they are executed serially



Concurrent operations

- But what does most recent mean?
  - Intuitively, get()/put() operations can be totally ordered
- But get()/put() operations can be issued concurrently
  - Need to reason about concurrency to formalize consistency



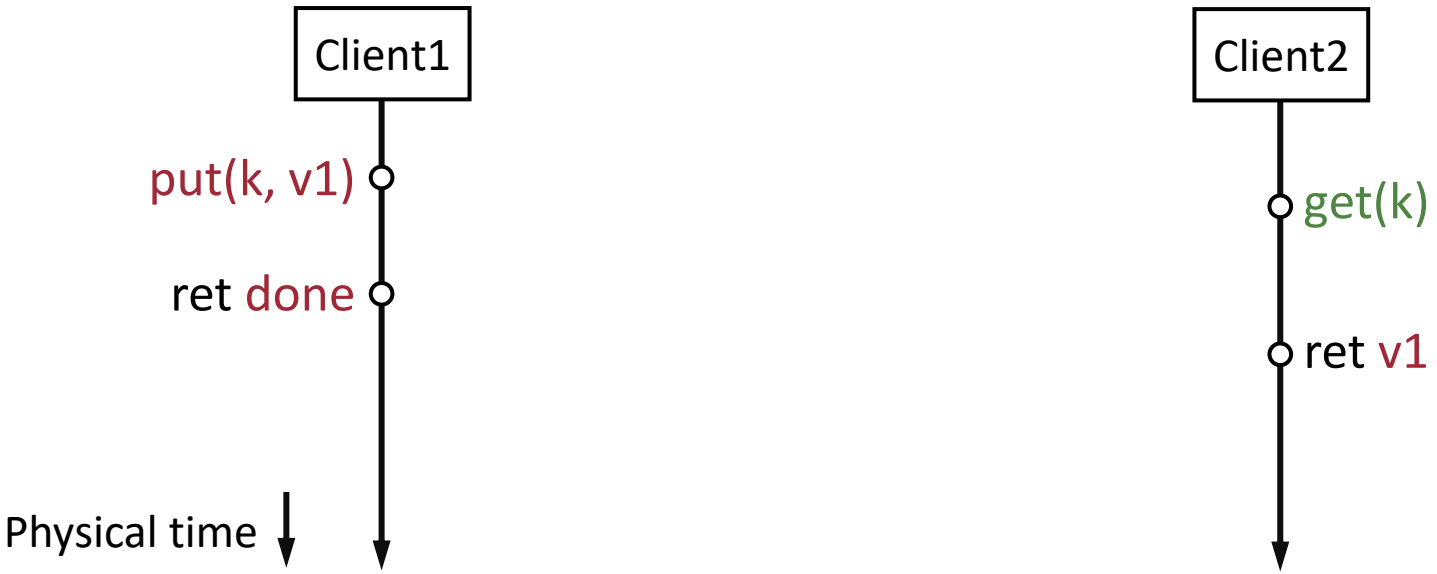
Linearizability

What is Linearizability?

- Linearizability is a data consistency model that closely matches programmer's expectations of storage behavior
  - Sometimes it is called "strong consistency" (loaded term)
- Definition of linearizability
  - Takes concurrent operations into account
  - Independent of network, node and timing model, replication
- To understand linearizability, we need to think in terms of concurrent operations

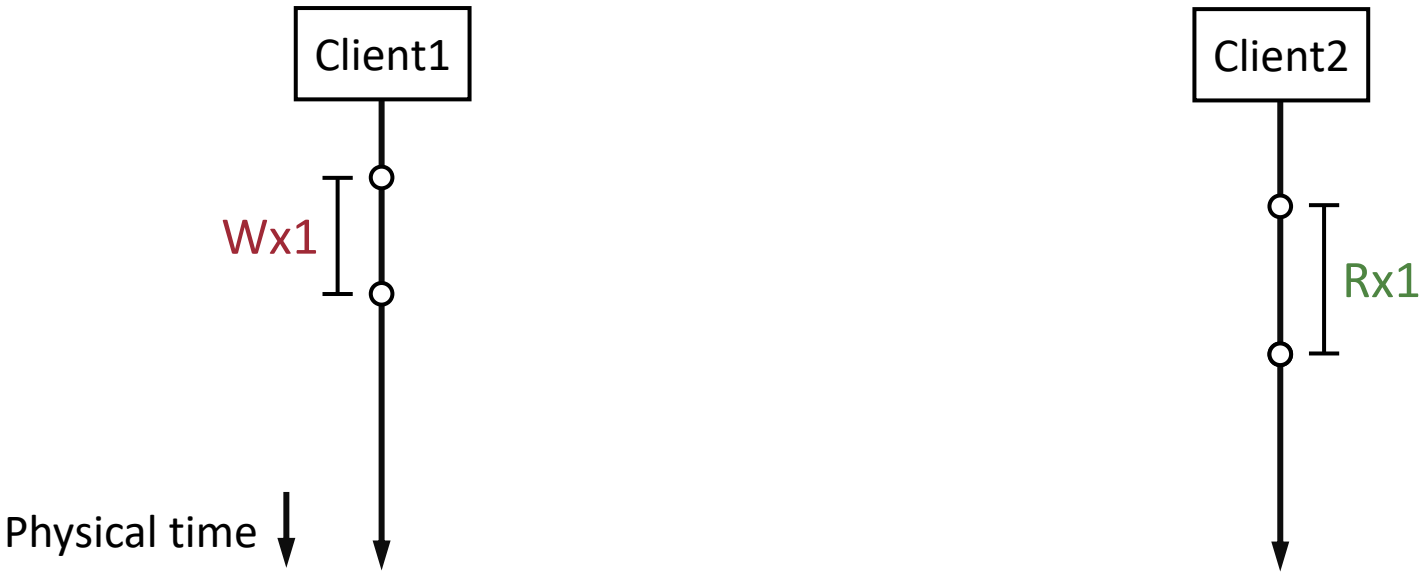
History of operations

- A history is a trace of possibly concurrent operations
- Think of each operation as an RPC with:
  - Invocation (with arguments), and
  - Response (with result values)
- Each operation accesses one data item



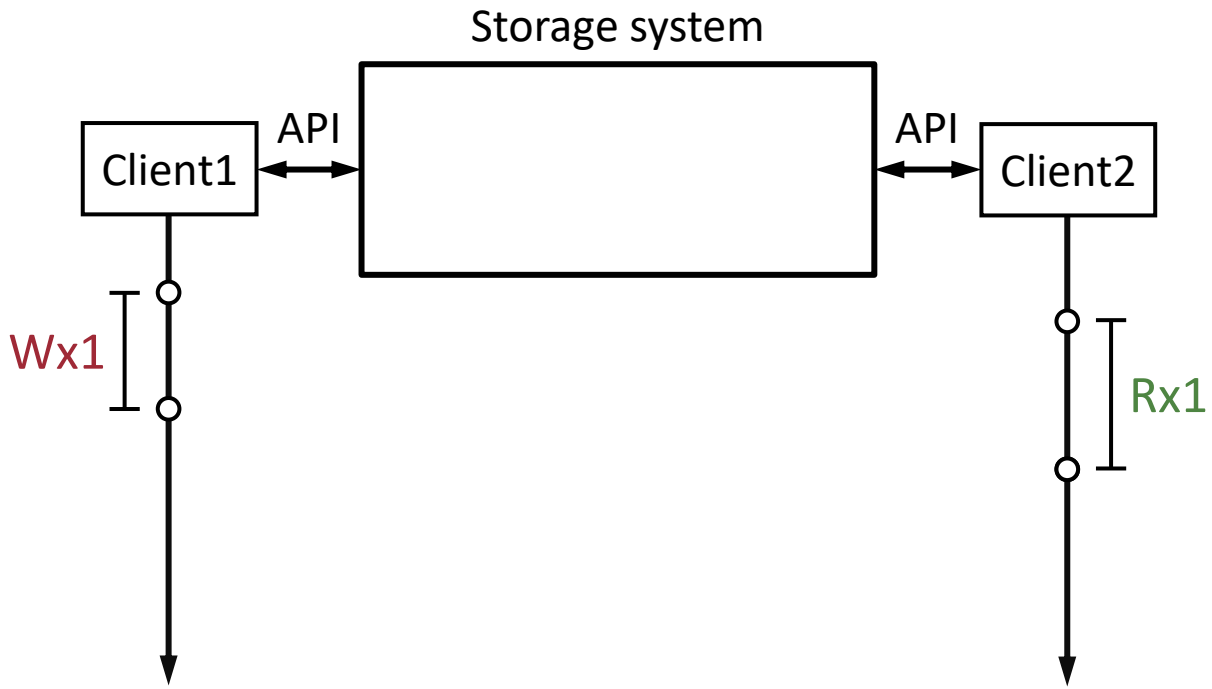
History of operations

- A history is a timeline of operations
  - Each operation has a duration in physical time
- Terminology
  - Wx1: write value 1 to record x, or put(x, 1)
  - Rx1: read record x returned 1, or get(x) = 1



History of operations

- History is shown from the perspective of clients
- We use it to reason about correctness of storage system
  - Note, we do not show storage system (think of it as a black box)



Linearizability definition

- A history is linearizable if every operation in the history takes effect (appears to execute) at some point of time (instantaneously) between its invocation and response
- Put another way:
  - You can find a point in time for each operation (called its linearization point) between its invocation and response, and
  - The result of every operation is the same as serial execution of the operations at their linearization points

Understanding linearizability

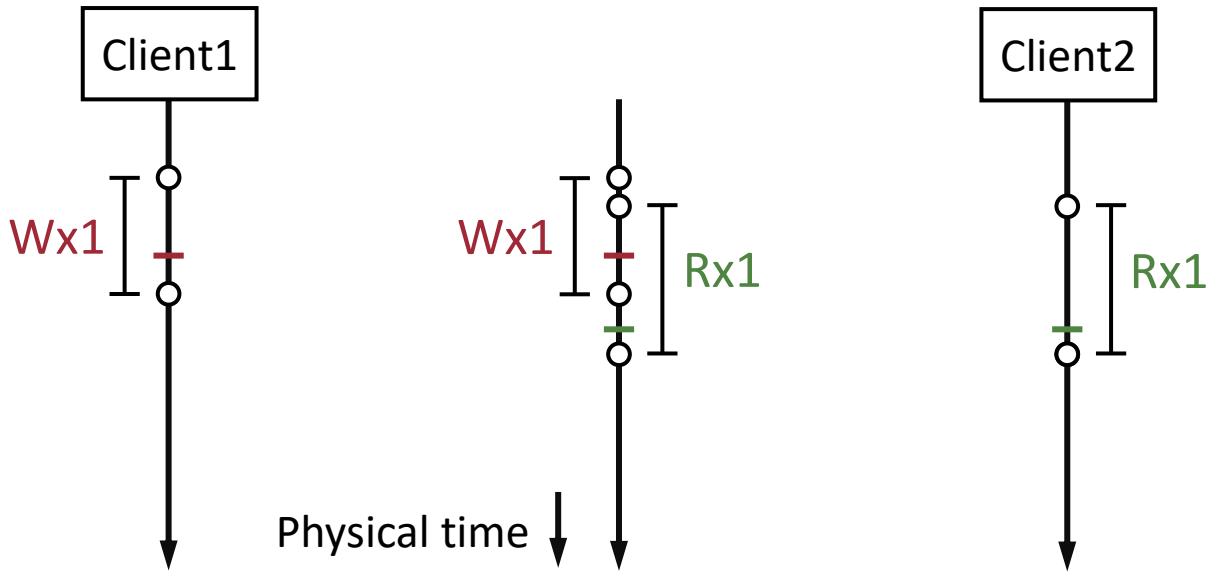
Linearizability imposes two conditions:

1. Operations appear to execute in a total order
2. Total order maintains real-time order between operations
   - If Operation A completes before Operation B begins in real-time, then A must be ordered before B
   - If neither A nor B completes before the other begins, then there is no real-time order, but there must be some total order

- What do the two conditions mean:
  1. Clients see same order of writes
  2. Clients read latest data
     - After a write completes, a later read (in real-time order) returns the value of the write (or later write)
     - Once a read returns a value, all later reads return that value (or the value of a later write)

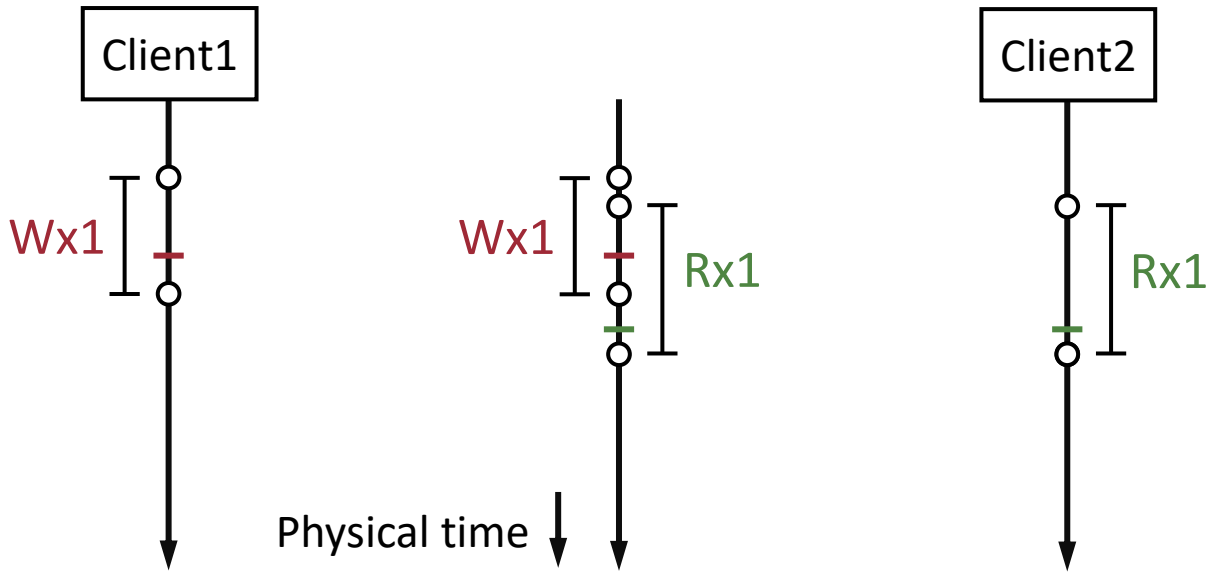
Why call it linearizable?

- The linearization points turn concurrent operations into a sequence of serial or linear operations on a timeline



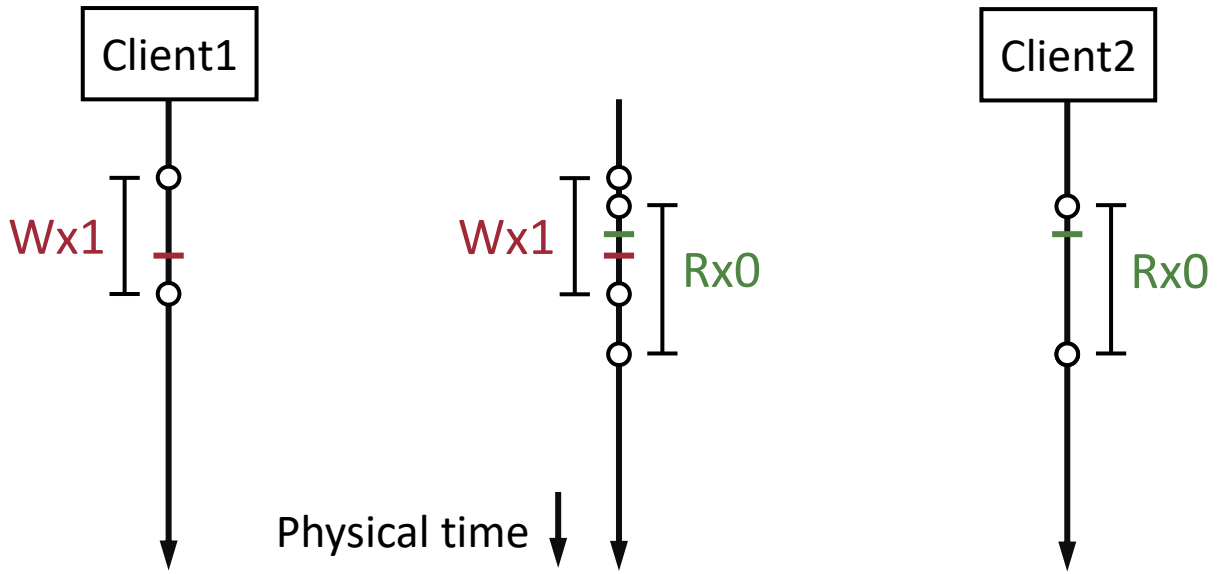
Concurrent operations

- In this example, get() reads the value written by put()



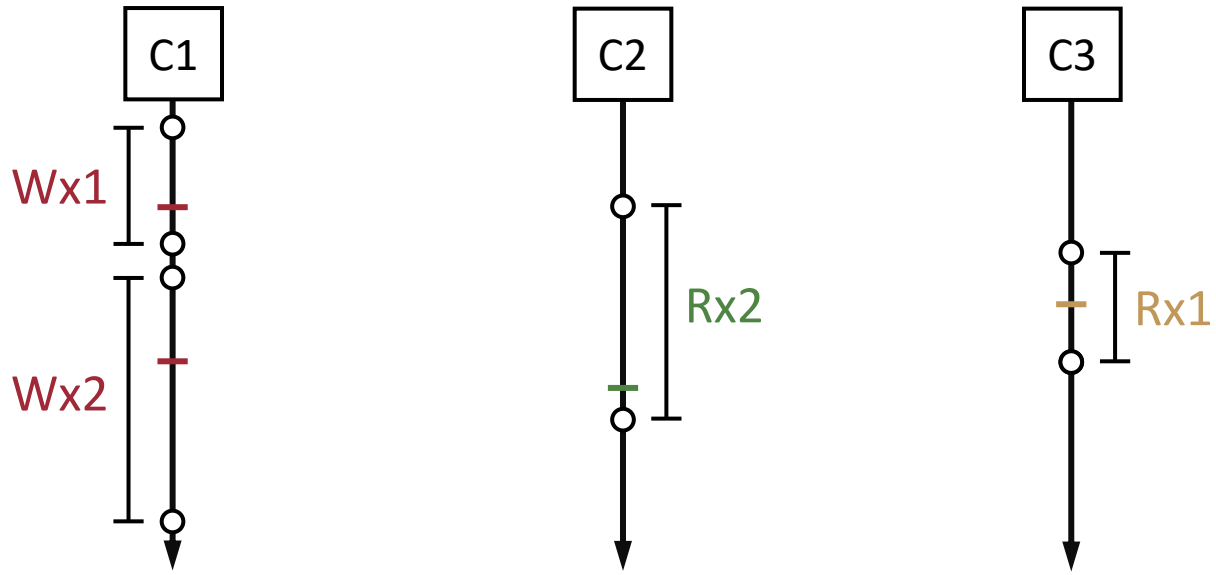
Concurrent operations

- But since get() and put() are concurrent, linearizability also allows get() to return value written before put()
  - Linearizability allows different results for concurrent operations
  - We can't tell in advance which result will be returned!



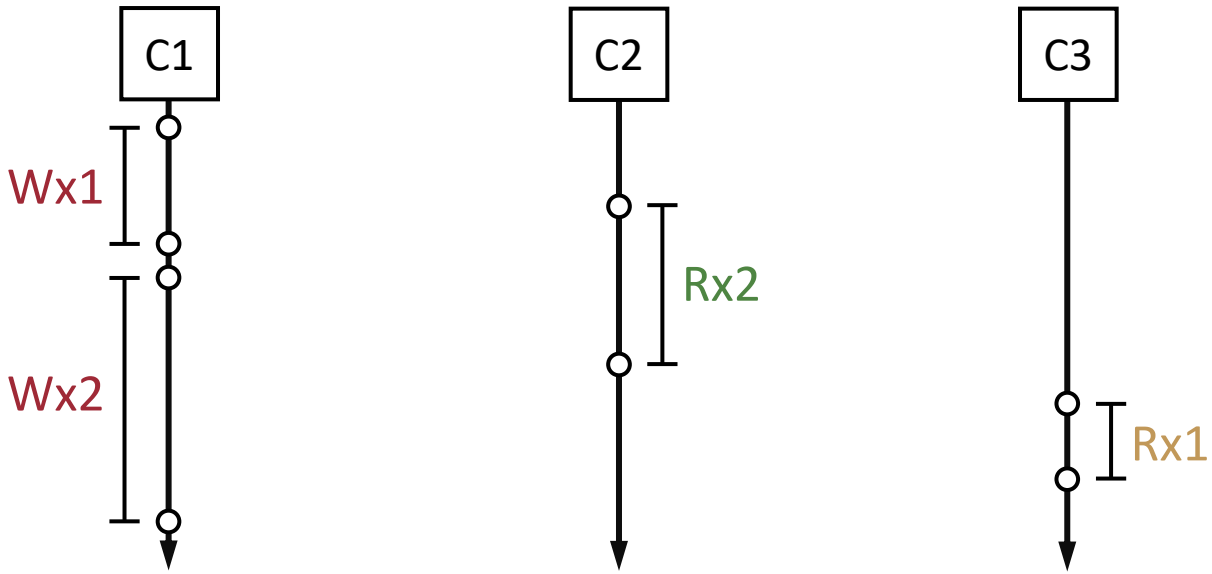
1: Is this history linearizable?

- Try assigning linearization points for each operation
- The order "Wx1 Rx1 Wx2 Rx2" satisfy linearizability



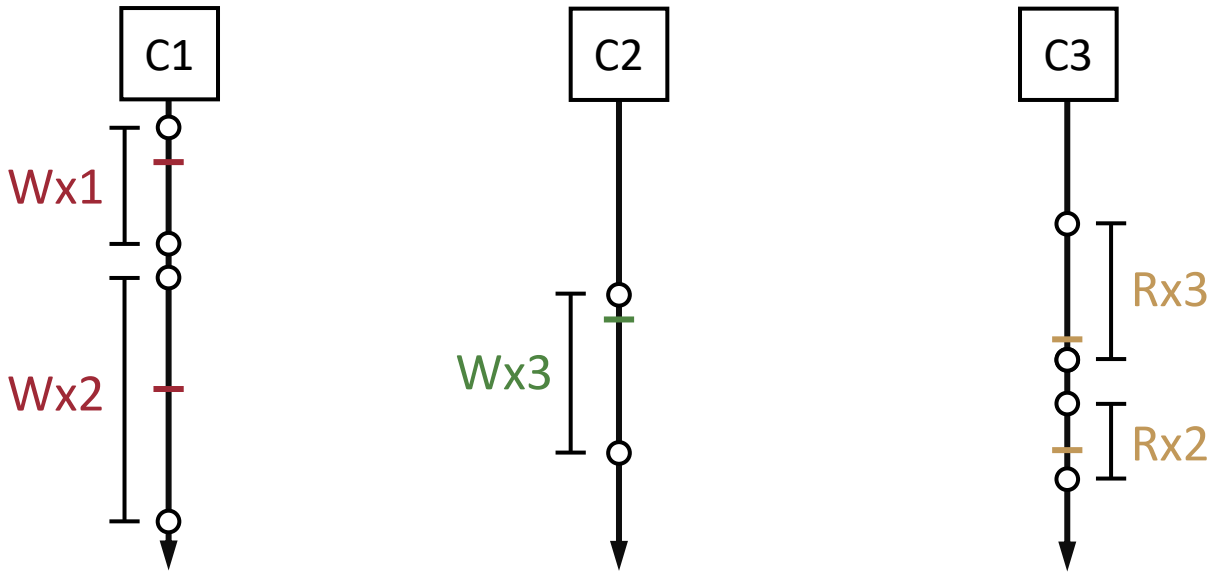
2: Is this history linearizable?

- Order must be "Wx1 Wx2 Rx2 Rx1"
  - Wx1 before Wx2 due to C1 timeline
  - Wx2 before Rx2 due to value returned
  - Rx2 before Rx1 due to real time
- But "Wx2 Rx1" not possible by linearizability
  - Even though Wx2 and Rx1 are concurrent!



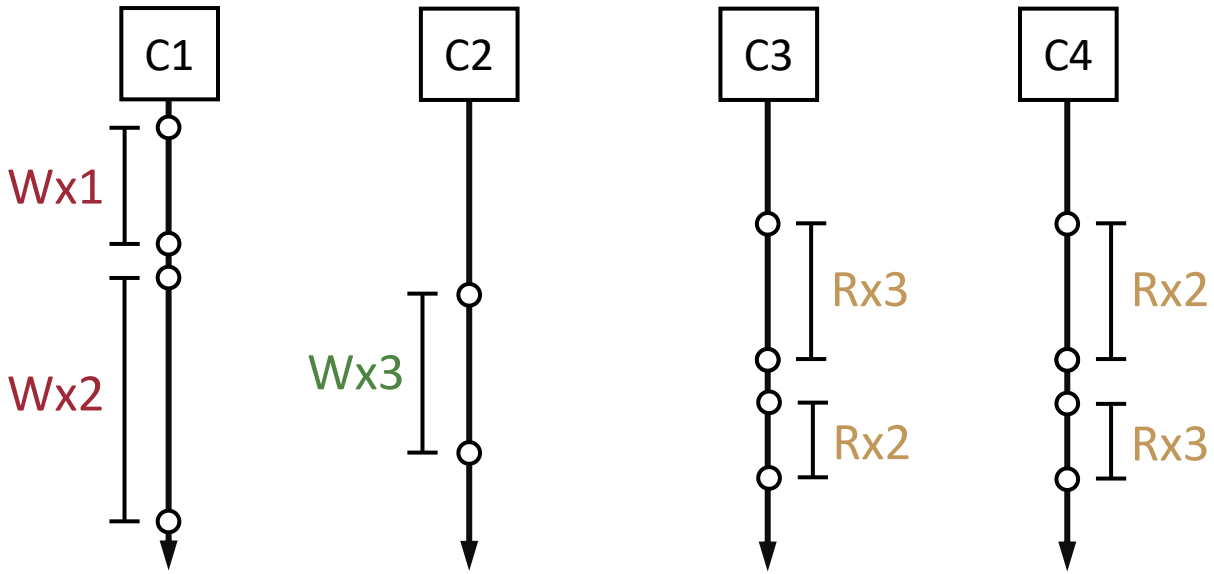
3: Is this history linearizable?

- This history seems non-linearizable since Rx3 would appear to force C3's second get() to also read 3
- Order "Wx1 Wx3 Rx3 Wx2 Rx2" satisfies linearizability
  - Wx3 and Wx2 are concurrent, either order of writes is okay



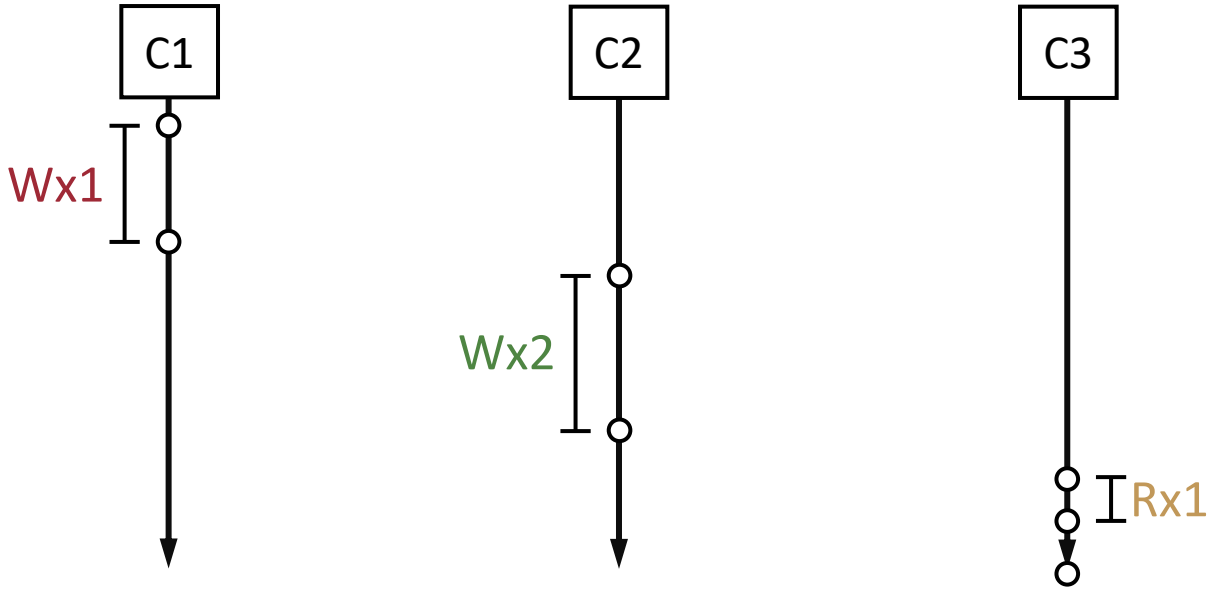
4: Is this history linearizable?

- Wx3 and Wx2 are concurrent, either order is okay
  - C3 needs the order "Wx3 Rx3 Wx2 Rx2" due to value returned,
    C4 needs the order "Wx2 Rx2 Wx3 Rx3" due to value returned
- Not linearizable
  - All clients must see same order of writes,
    potentially an issue for caching and replication



5: Is this history linearizable?

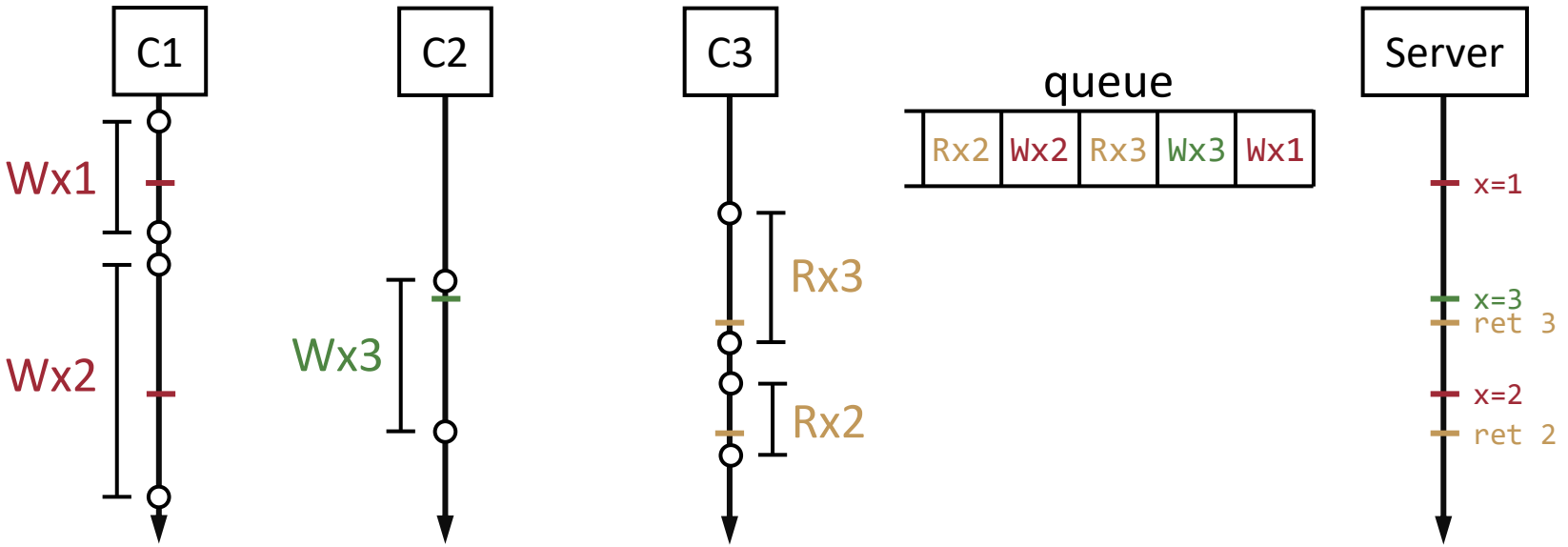
- Order must be "Wx1 Wx2 Rx1"
  - Wx1 before Wx2 due to real time,
    Wx2 before Rx1 due to real time
- But "Wx2 Rx1" not possible by linearizability
  - Clients read latest (not stale) data,
    potentially an issue for caching and replication



Implementing Linearizability

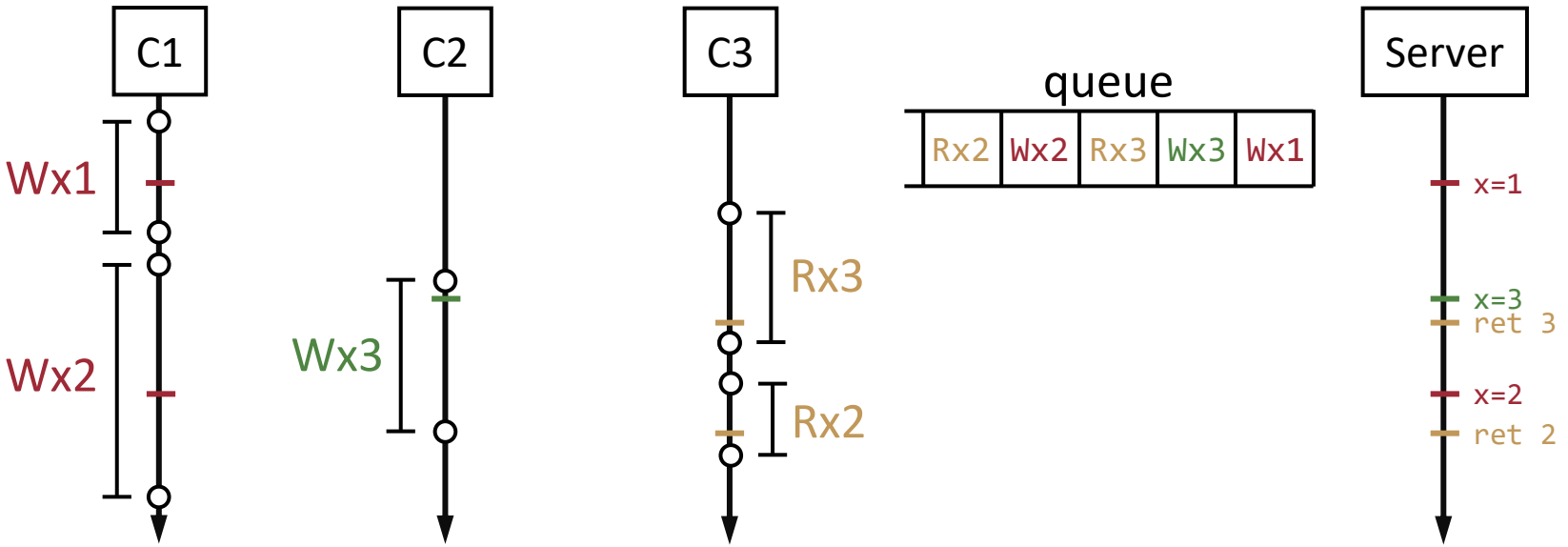
Basic implementation

- Let's assume we have a storage server that
  - Queues arriving requests
  - Queues concurrent requests in arbitrary order, e.g., by arrival time
  - Executes each request serially and returns results
- Does the server guarantee linearizability?



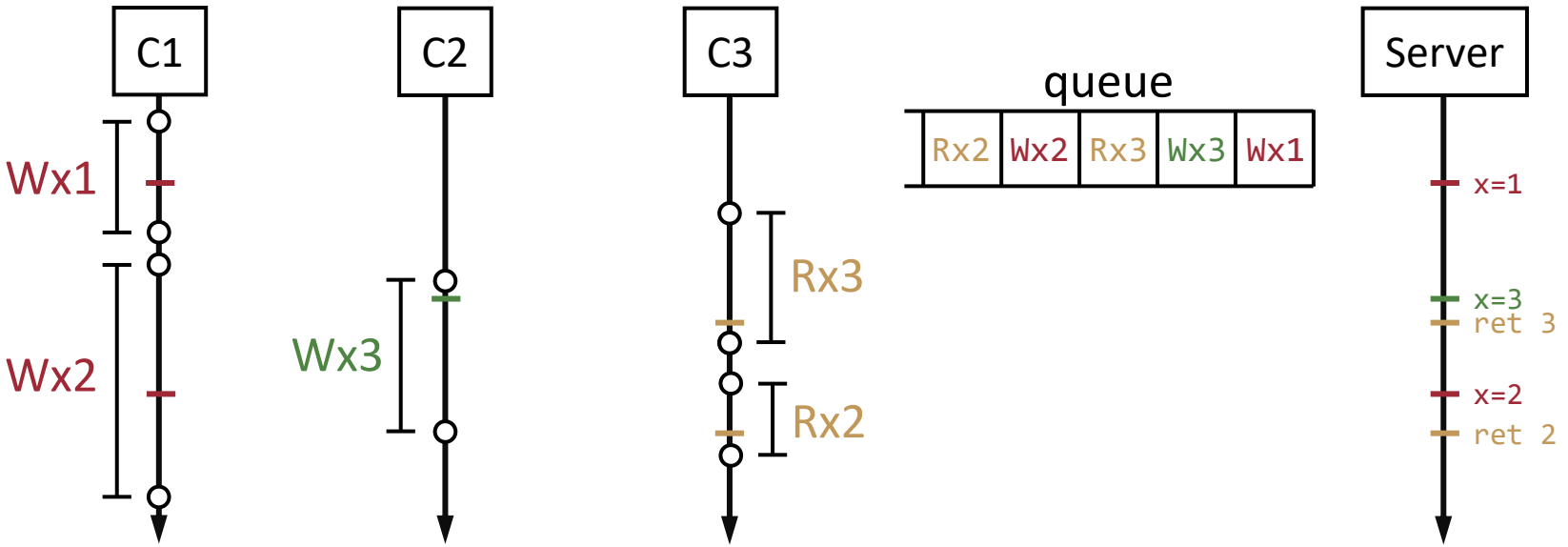
Exactly-once semantics

- Need to ensure exactly-once semantics for linearizability
- How to ensure exactly-once semantics?
  - Perform duplication detection
  - Store state updates durably and atomically
  - Optionally, use a fault-tolerant service



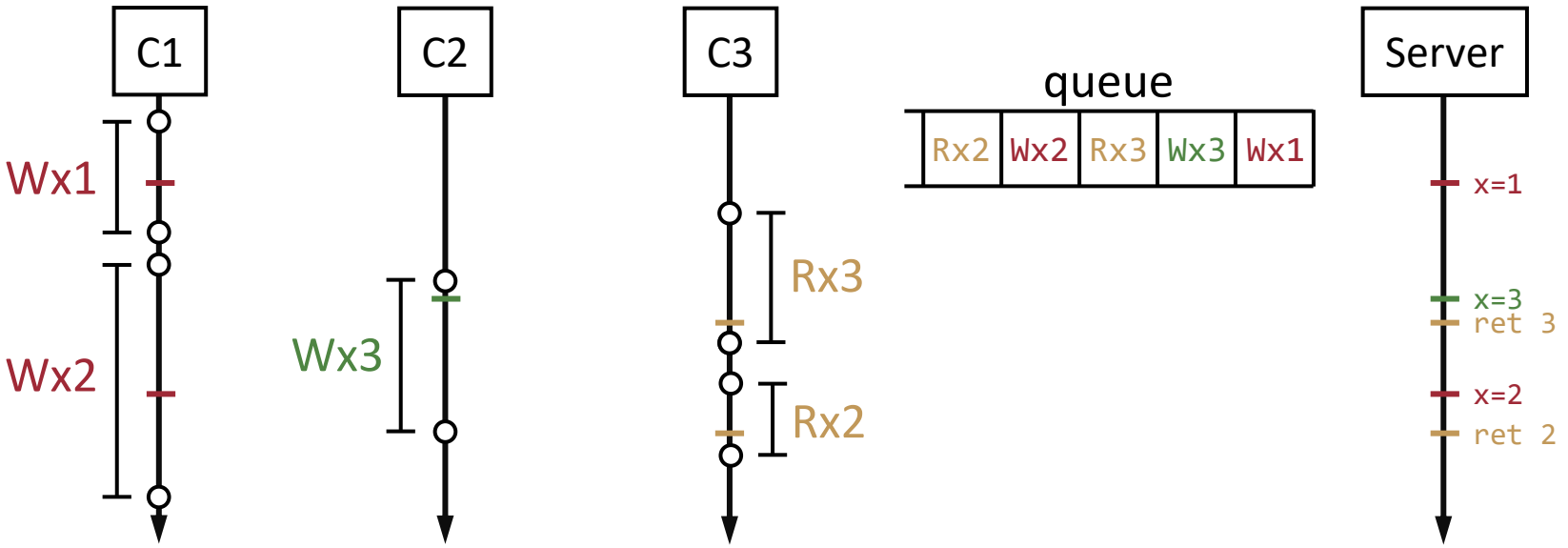
Duplication detection

- Server needs to deduplicate requests
  - What problem can occur otherwise?



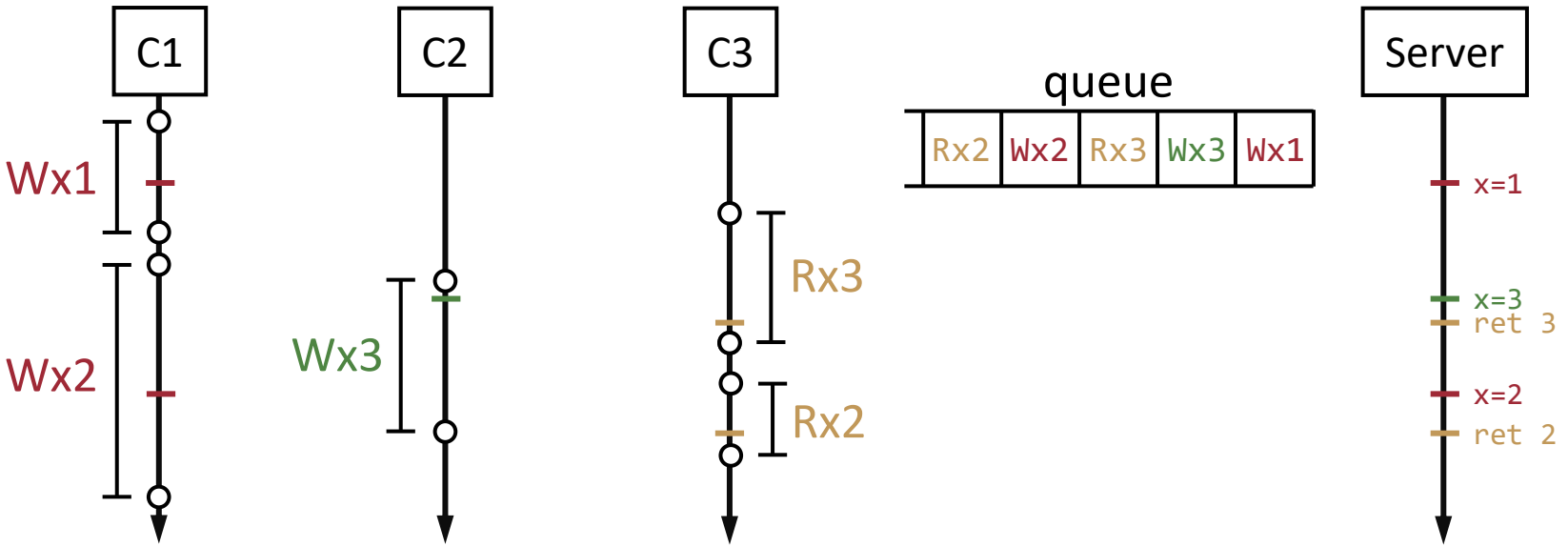
Durable and atomic state updates

- All updated state must be stored durably on disk
  - What problem can occur otherwise after a server failure?
- All updated state must be stored atomically
  - What problem an occur otherwise after a server failure?



Fault-tolerant service

- A single server can crash
  - While server recovers, no service, i.e., no availability
- Let's store copies of data (replicas) on multiple servers
  - Then we can provide availability even when some servers fail
  - Such a service is called a fault-tolerant service



Implementing exactly-once

- Need to ensure exactly-once semantics for linearizability
- How to ensure exactly-once semantics?
  - Perform duplication detection
  - Store state updates durably and atomically
  - Optionally, use a fault-tolerant service
- We have discussed duplication detection
- We will look at the design of Raft, a fault-tolerant replicated service that ensures linearizability
- Later in the course, we will look at how to store state updates durably and atomically

Benefits of linearizability

- Provides strong real-time data consistency guarantees
- For application programmers, the model is intuitive
  - Same as single machine processing one request at a time
    - All clients see data changes in same order
    - Reads see latest or fresh data
  - Hides complexity inherent in distributed systems
    - Independent of network, node and timing model, replication
- Model is general, can be applied to more than read/write
  - Delete, append, increment, CAS for locking, etc.

Issues with linearizability

- Low performance since it serializes operations
- Limits availability under network partitions
- Say a set of geographically distributed web servers cache data from a backend database server
  - Each data item may have copies (replicas) at the web servers
  - Ensuring that a response returns the latest copy requires expensive synchronization between all the caches and the database
  - Instead, a web server could directly return its cached item
  - This may occasionally return stale data, but it is faster, and it allows availability even when the database is unavailable or highly loaded
- Takeaway: need weaker consistency models for higher performance and availability

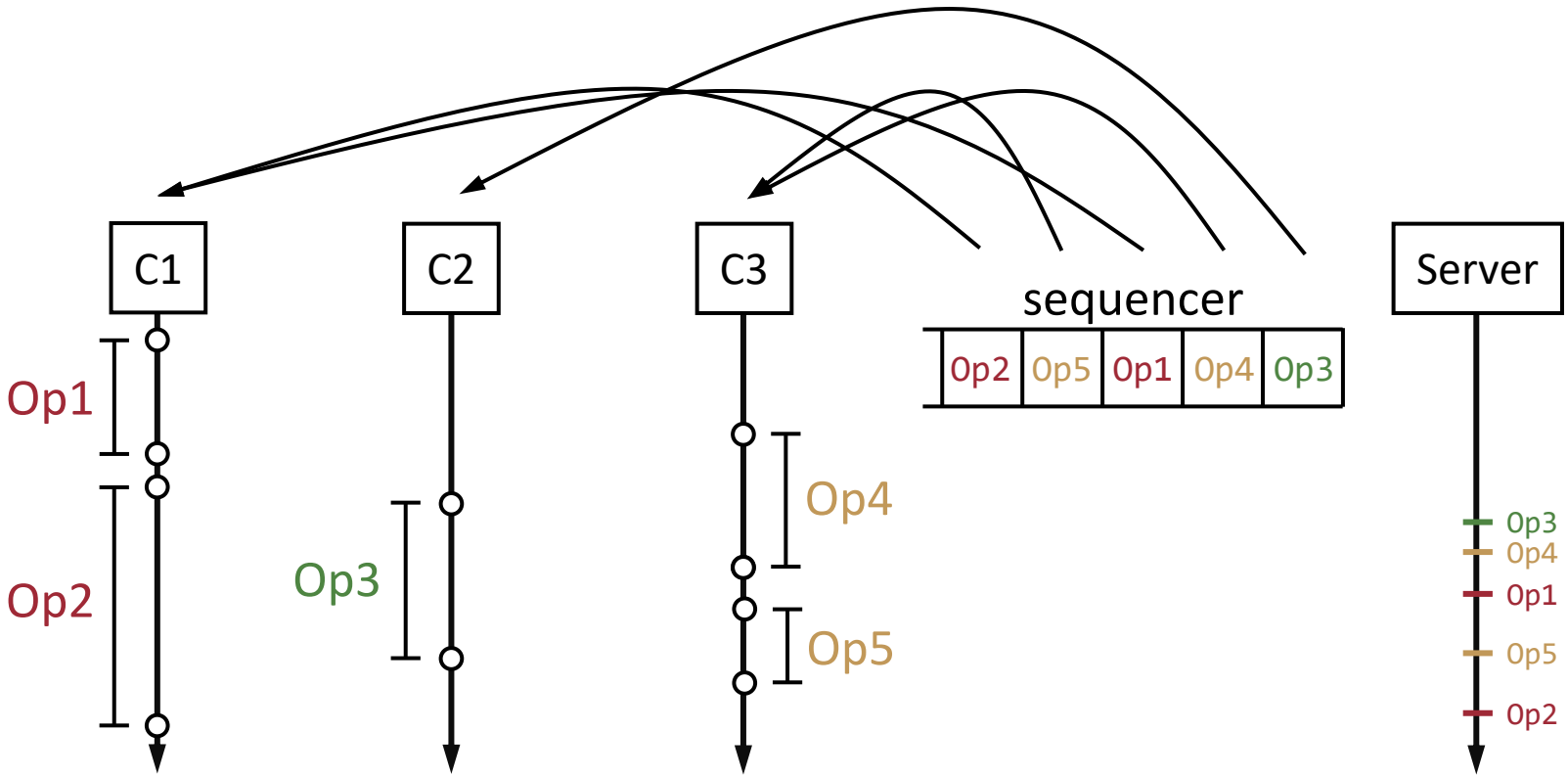
Sequential Consistency

Sequential consistency

- Sequential consistency weakens linearizability by not providing any real-time guarantees
- Sequential consistency: all processes execute operations in some total order, while preserving real-time ordering
  - Operations appear to occur instantaneously, consistent with program order, at some point in between invocation & response
- Provides better performance than linearizability because operations across processes can be reordered (provided there is some total order)

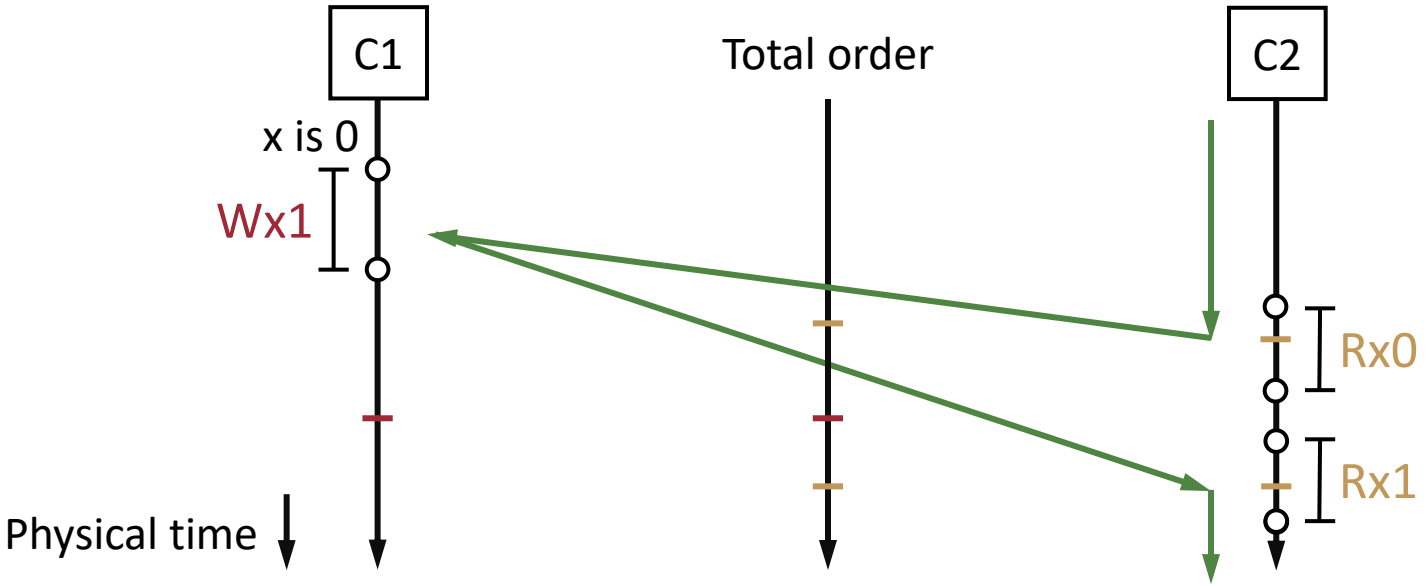
Implementing sequential consistency

- A sequencer repeatedly
  - Chooses an operation from some client (in program order)
  - Runs the operation serially



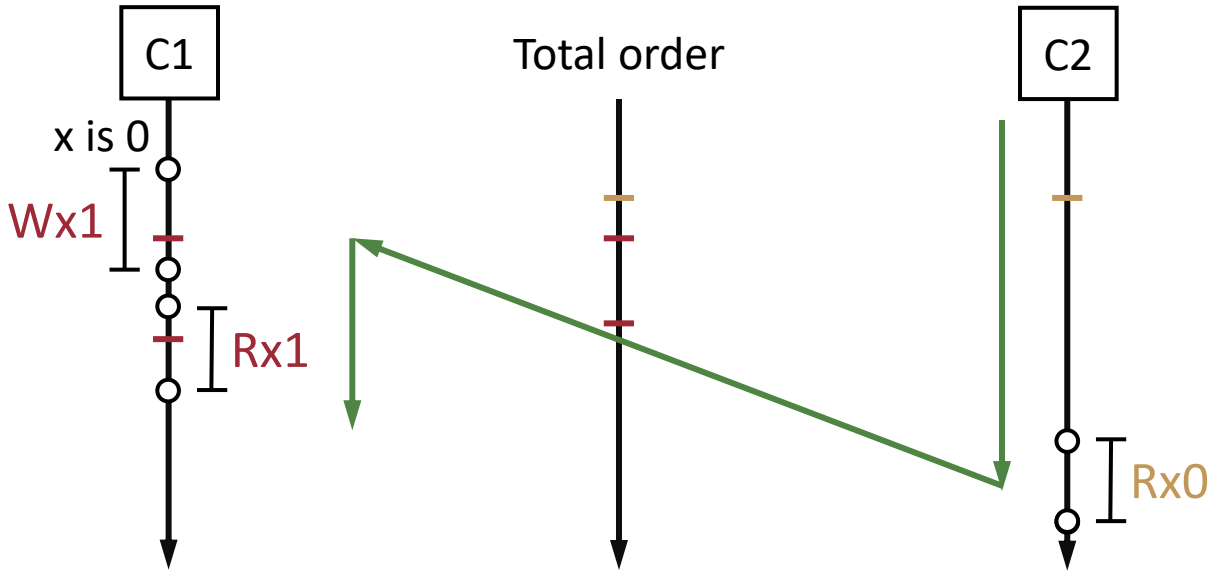
Sequential consistency - Example 1

- Sequentially consistent
- Writes may appear to be delayed



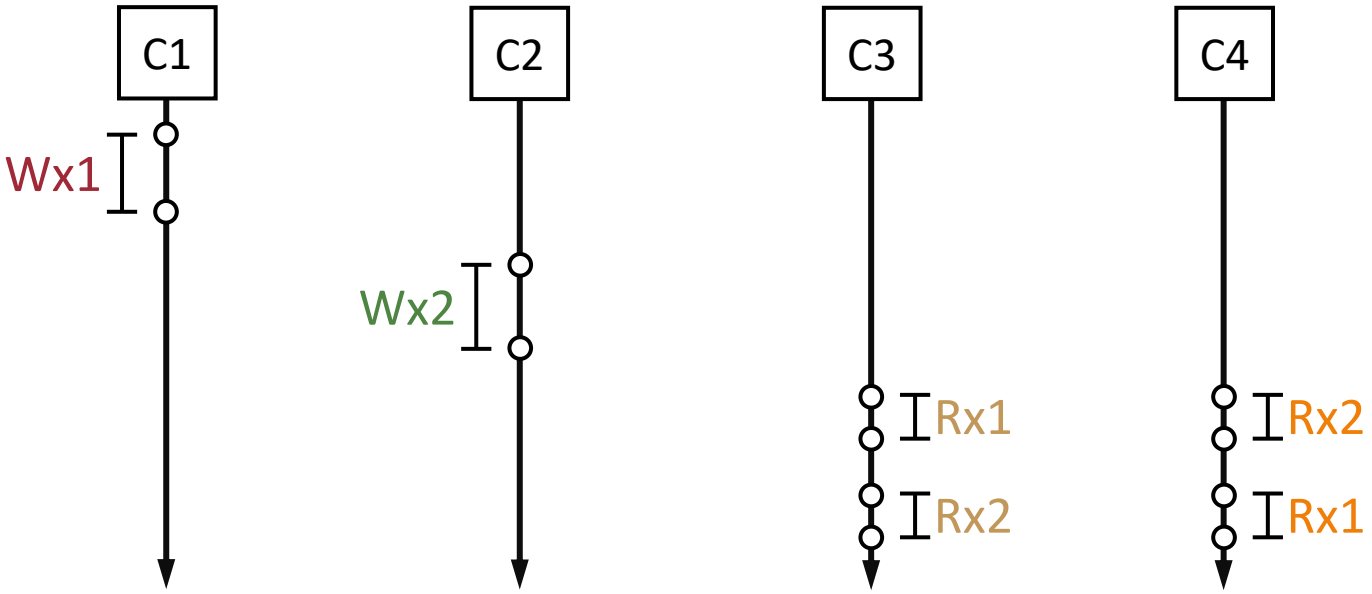
Sequential consistency - Example 2

- Sequentially consistent
- Reads may return stale data



Sequential consistency - Example 3

- Not sequentially consistent
- There is no possible total ordering of operations



Understanding sequential consistency

- There is a total ordering of operations, but
  - A write may be ordered much after its response (delayed write)
  - A read may return arbitrarily stale data (stale read)
- However, sequential consistency is still a strong model
  - Still ensures total order of operations
  - Once A observes data from B, A cannot observe B's prior state
- We will look at the design of Zookeeper, a highly-available coordination service that ensures sequential consistency
  - Improves performance, availability compared to linearizability, particularly for read-heavy workloads

Coordination problem redux

- Consider this simple coordination problem:

```
Coordinator:

put(config, “new config”)
put(config_done, TRUE)
```

```
Clients:

while get(config_done) != TRUE:
    wait
get(config) // is it “new config”?
```

- Is `get(config)` guaranteed to return “new config” with:
  - Linearizability?
  - Serializability?

Conclusions

- Linearizability is a strong real-time consistency model
  - Provides an intuitive programming model, but
  - Limits performance and availability
- Sequential consistency provides better performance and availability, but has weaker consistency
  - Writes may appear delayed, reads may read stale data
  - But still usable for many applications
- Later, we will discuss weaker consistency models that trade consistency for even better performance and availability
  - Causal consistency, eventual consistency, …