Distributed Systems

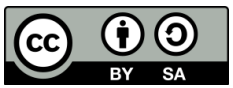
The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

University of Cambridge

Computer Science Tripos, Part IB



Lecture 7

Replica consistency

“Consistency”

A word that means many different things in different contexts!

“Consistency”

A word that means many different things in different contexts!

- **ACID**: a transaction transforms the database from one “consistent” state to another

“Consistency”

A word that means many different things in different contexts!

- **ACID**: a transaction transforms the database from one “consistent” state to another

  Here, “consistent” = satisfying application-specific invariants

  e.g. “every course with students enrolled must have at least one lecturer”

“Consistency”

A word that means many different things in different contexts!

- **ACID**: a transaction transforms the database from one “consistent” state to another

  Here, “consistent” = satisfying application-specific invariants

  e.g. “every course with students enrolled must have at least one lecturer”

- **Read-after-write consistency** (lecture 5)

“Consistency”

A word that means many different things in different contexts!

- **ACID**: a transaction transforms the database from one “consistent” state to another

  Here, “consistent” = satisfying application-specific invariants

  e.g. “every course with students enrolled must have at least one lecturer”

- **Read-after-write consistency** (lecture 5)
- **Replication**: replica should be “consistent” with other replicas

"Consistency"

A word that means many different things in different contexts!

- **ACID**: a transaction transforms the database from one "consistent" state to another

  Here, "consistent" = satisfying application-specific invariants

  e.g. "every course with students enrolled must have at least one lecturer"

- **Read-after-write consistency** (lecture 5)
- **Replication**: replica should be "consistent" with other replicas

  "consistent" = in the same state? (when exactly?)

  "consistent" = read operations return same result?

- **Consistency model**: many to choose from

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- A transaction either **commits** or **aborts**

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- A transaction either **commits** or **aborts**
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- A transaction either **commits** or **aborts**
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects
- ACID consistency (preserving invariants) relies on atomicity

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- A transaction either **commits** or **aborts**
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects
- ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

- Either all nodes must commit, or all must abort

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- A transaction either **commits** or **aborts**
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects
- ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

- Either all nodes must commit, or all must abort
- If any node crashes, all must abort

Ensuring this is the **atomic commitment** problem.
Looks a bit similar to consensus?

Atomic commit versus consensus

Consensus	**Atomic commit**
One or more nodes propose a value	Every node votes whether to commit or abort

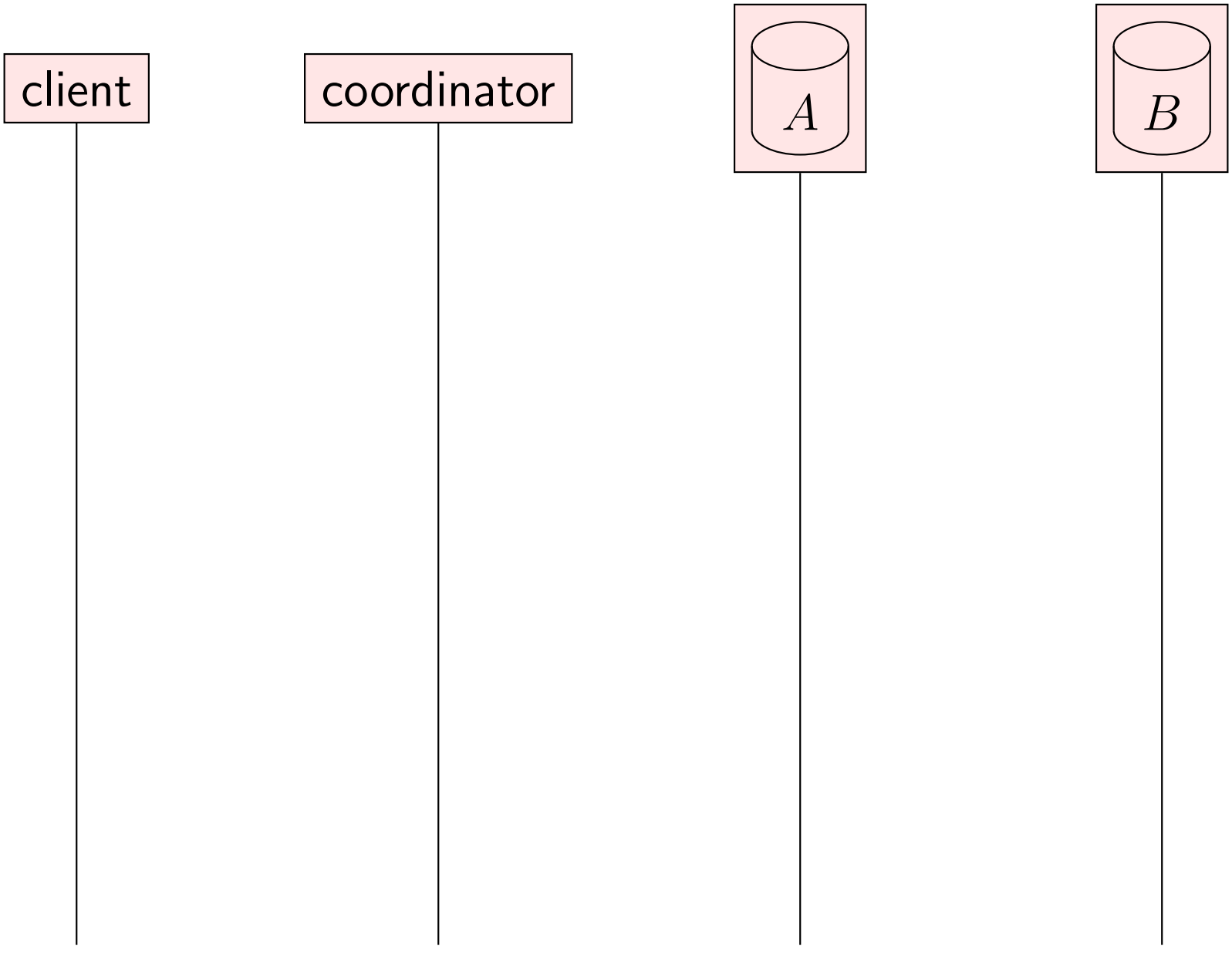
Atomic commit versus consensus

Consensus	**Atomic commit**
One or more nodes propose a value	Every node votes whether to commit or abort
Any one of the proposed values is decided	Must commit if all nodes vote to commit; must abort if ≥ 1 nodes vote to abort

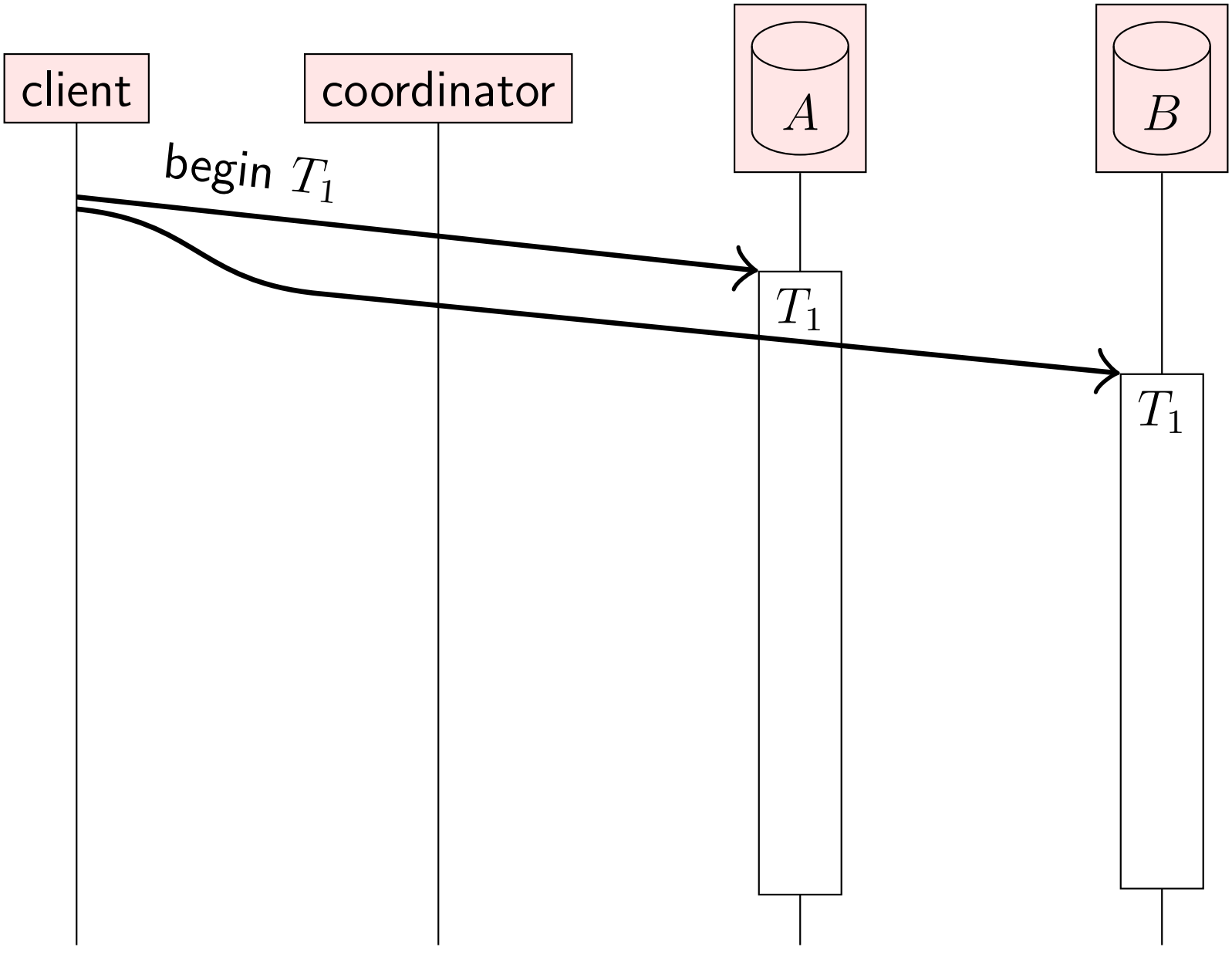
Atomic commit versus consensus

Consensus	**Atomic commit**
One or more nodes propose a value	Every node votes whether to commit or abort
Any one of the proposed values is decided	Must commit if all nodes vote to commit; must abort if ≥ 1 nodes vote to abort
Crashed nodes can be tolerated, as long as a quorum is working	Must abort if a participating node crashes

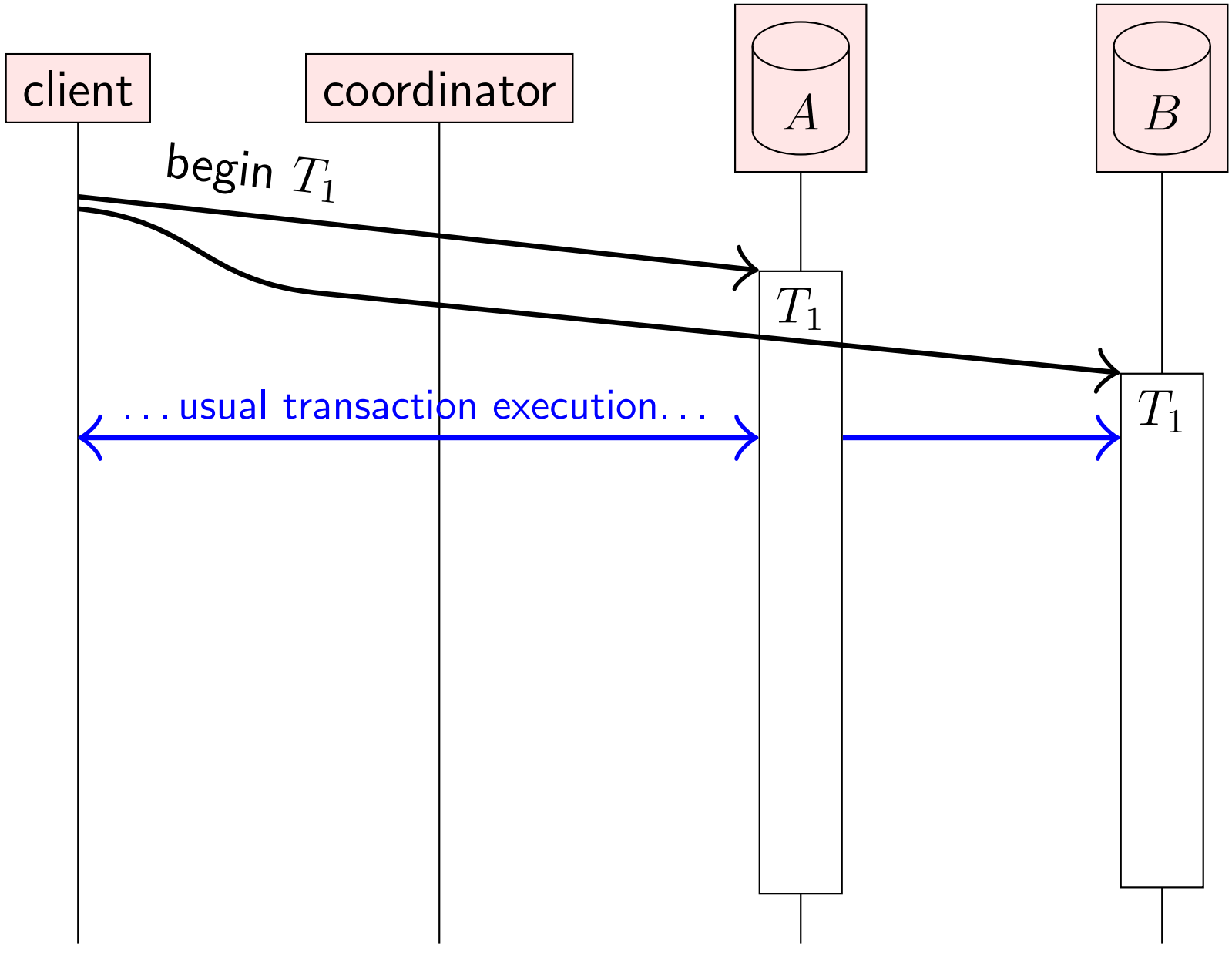
Two-phase commit (2PC)



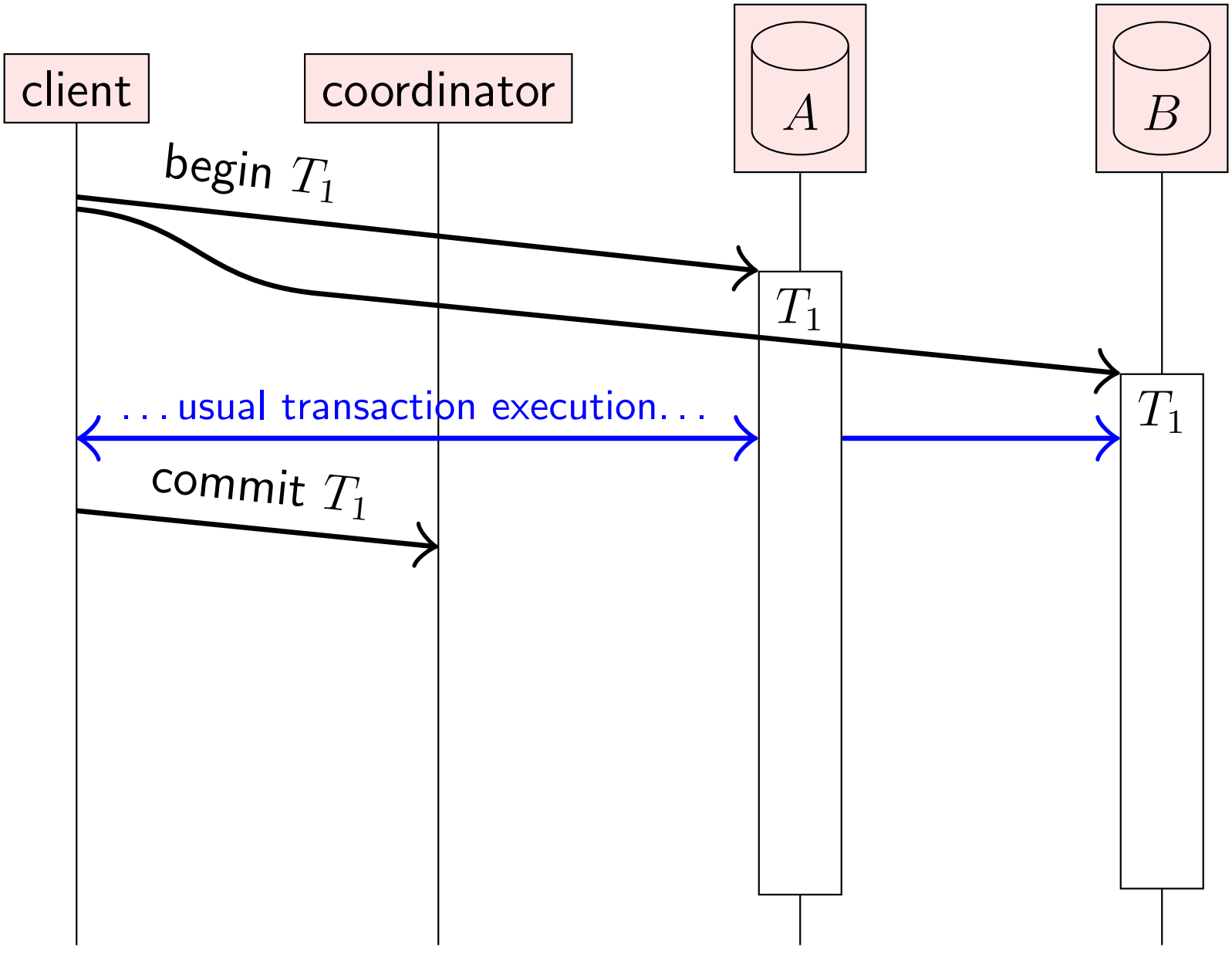
Two-phase commit (2PC)



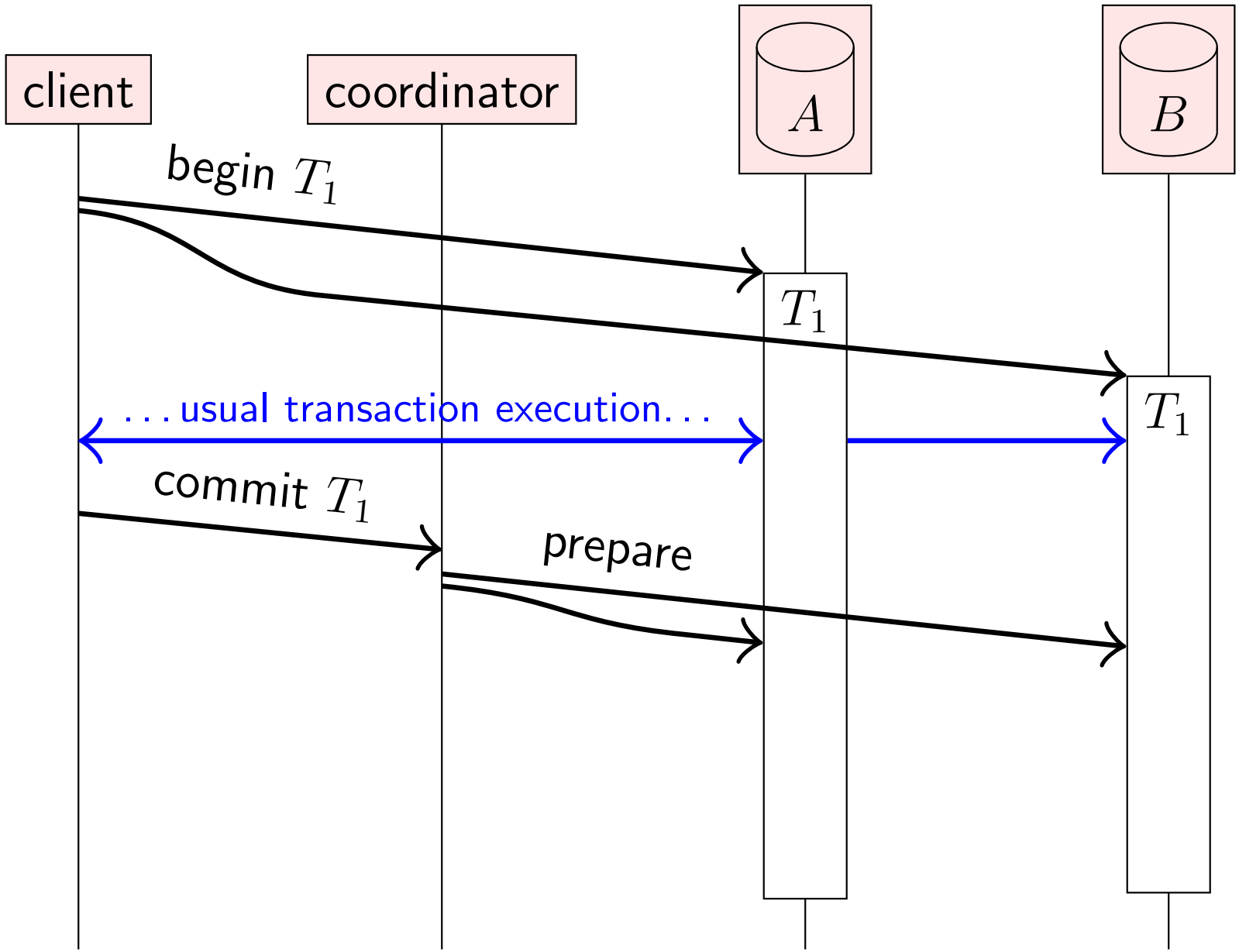
Two-phase commit (2PC)



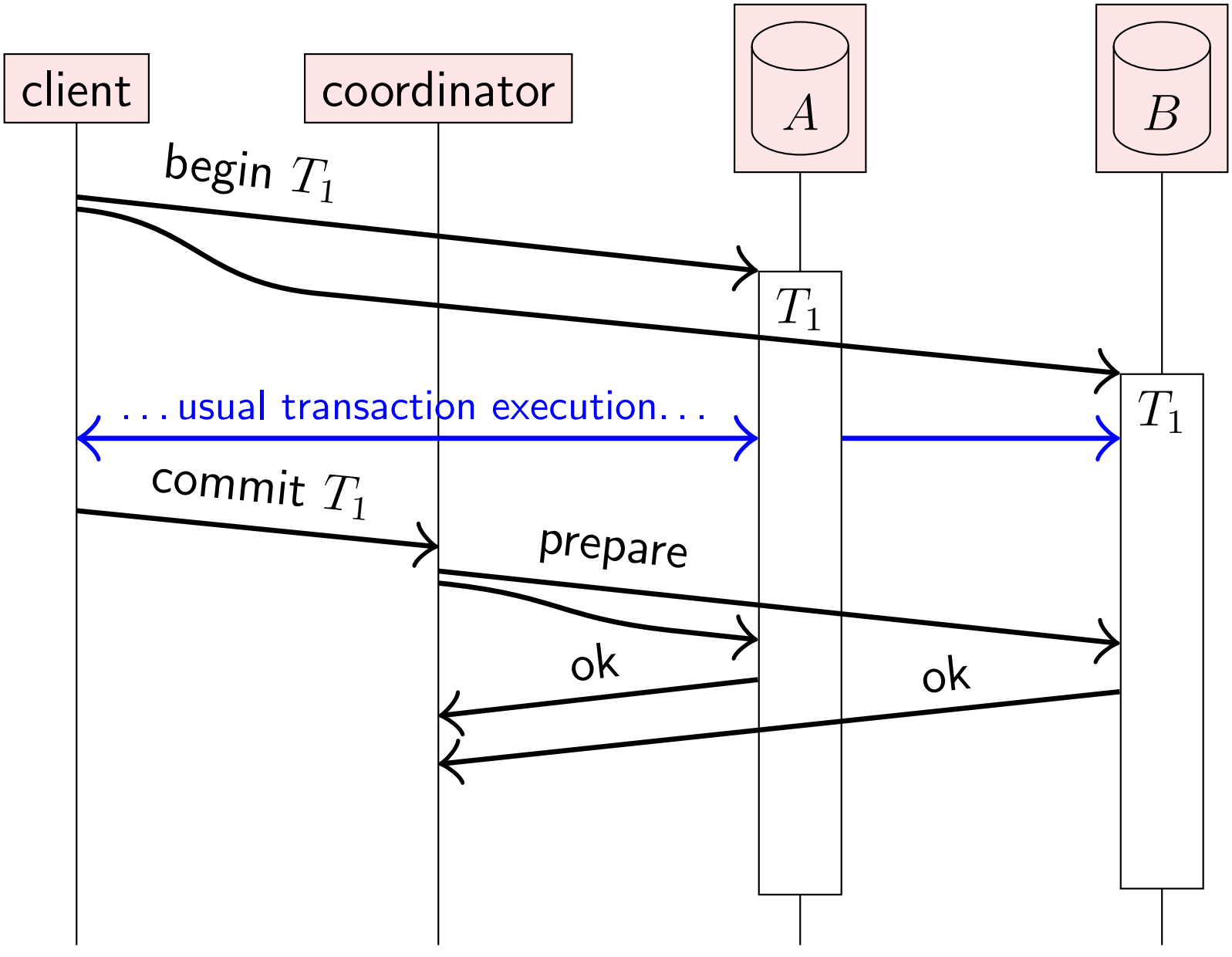
Two-phase commit (2PC)



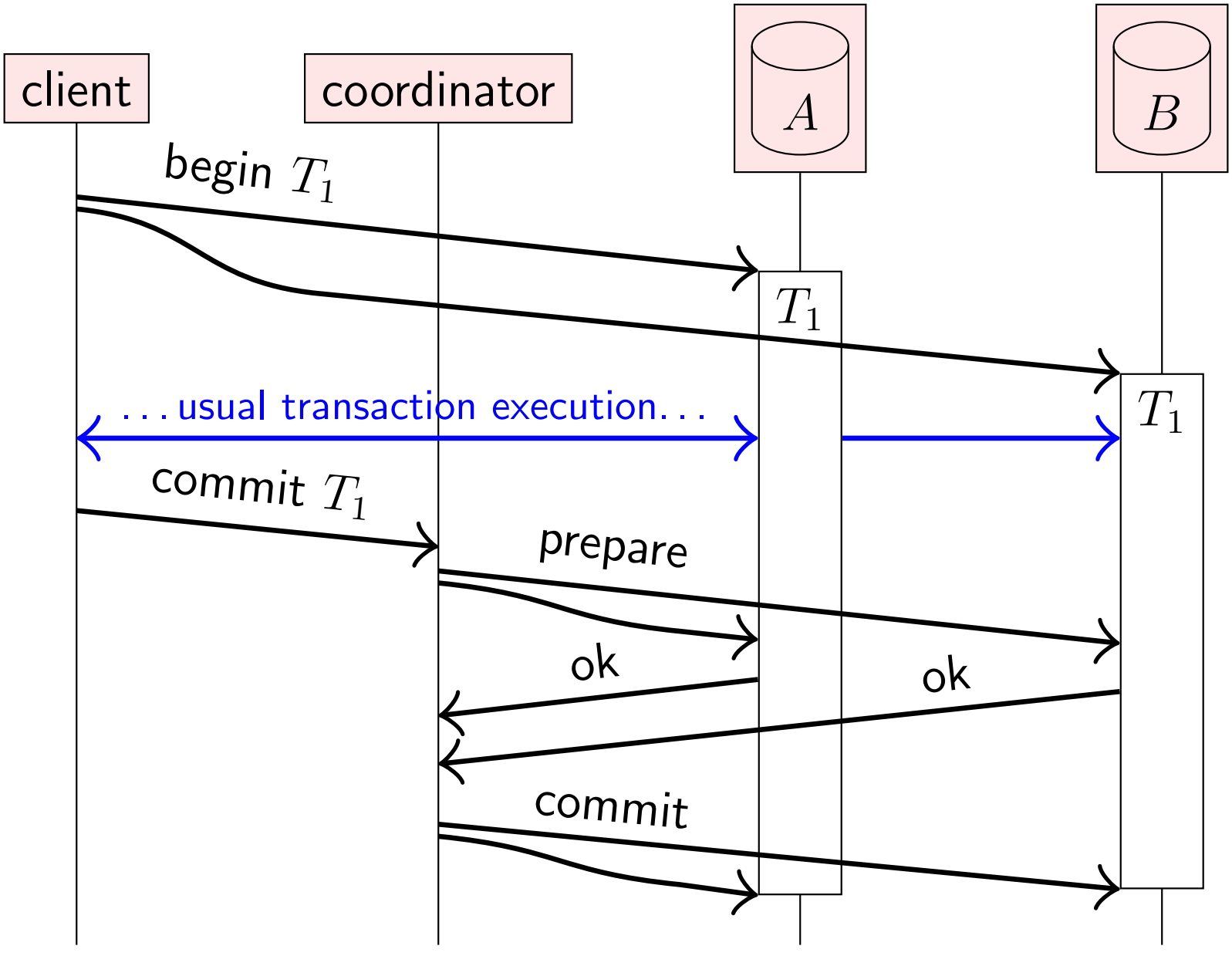
Two-phase commit (2PC)



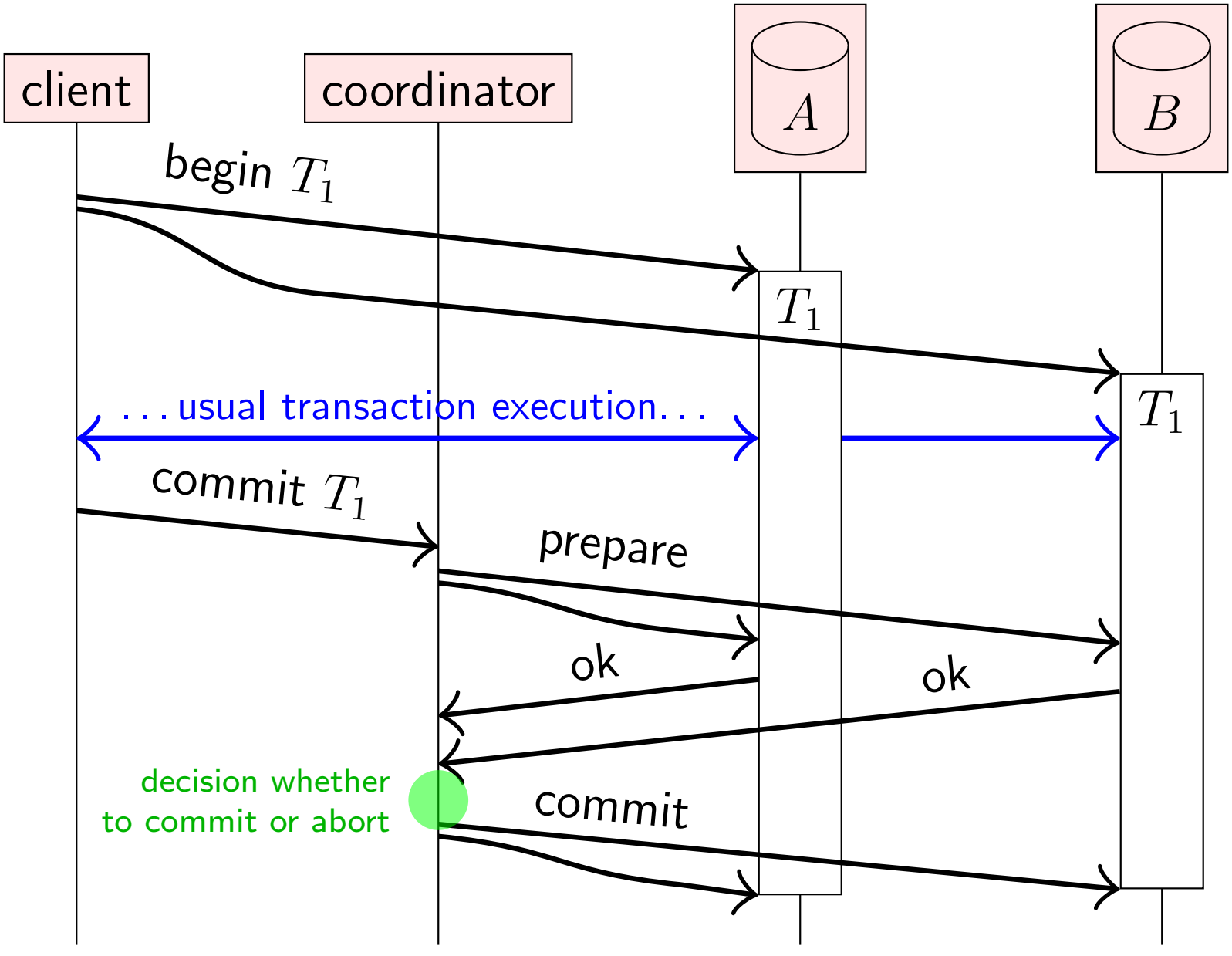
Two-phase commit (2PC)



Two-phase commit (2PC)



Two-phase commit (2PC)



The coordinator in two-phase commit

What if the coordinator crashes?

The coordinator in two-phase commit

What if the coordinator crashes?

- Coordinator writes its decision to disk
- When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)

The coordinator in two-phase commit

What if the coordinator crashes?

- Coordinator writes its decision to disk
- When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided

The coordinator in two-phase commit

What if the coordinator crashes?

- Coordinator writes its decision to disk
- When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- Replicas participating in transaction cannot commit or abort after responding "ok" to the *prepare* request (otherwise we risk violating atomicity)

The coordinator in two-phase commit

What if the coordinator crashes?

- Coordinator writes its decision to disk
- When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- Replicas participating in transaction cannot commit or abort after responding "ok" to the *prepare* request (otherwise we risk violating atomicity)
- Algorithm is blocked until coordinator recovers

Fault-tolerant two-phase commit (1/2)

on initialisation for transaction T **do**
    $commitVotes[T] := \{\};\ replicas[T] := \{\};\ decided[T] :=$ false
end on

on request to commit transaction T with participating nodes R **do**
    for each $r \in R$ **do** send $(\text{Prepare}, T, R)$ to r
end on

on receiving $(\text{Prepare}, T, R)$ at node $replicaId$ **do**
    $replicas[T] := R$
    $ok =$ "is transaction T able to commit on this replica?"
    total order broadcast $(\text{Vote}, T, replicaId, ok)$ to $replicas[T]$
end on

on a node suspects node $replicaId$ to have crashed **do**
    for each transaction T in which $replicaId$ participated **do**
        total order broadcast $(\text{Vote}, T, replicaId, \text{false})$ to $replicas[T]$
    end for
end on

Fault-tolerant two-phase commit (2/2)

on delivering $(\text{Vote}, T, replicaId, ok)$ by total order broadcast **do**
  if $replicaId \notin commitVotes[T] \ \wedge \ replicaId \in replicas[T] \wedge \neg decided[T]$ **then**
    if $ok = \text{true}$ **then**
      $commitVotes[T] := commitVotes[T] \cup \{replicaId\}$
      if $commitVotes[T] = replicas[T]$ **then**
        $decided[T] := \text{true}$
        commit transaction T at this node
      end if
    else
      $decided[T] := \text{true}$
      abort transaction T at this node
    end if
  end if
end on

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define “consistency” here?

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

- Informally: every operation takes effect **atomically** sometime after it started and before it finished

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

- Informally: every operation takes effect **atomically** sometime after it started and before it finished
- All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

- Informally: every operation takes effect **atomically** sometime after it started and before it finished
- All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- Consequence: every operation returns an "up-to-date" value, a.k.a. "strong consistency"

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

- Informally: every operation takes effect **atomically** sometime after it started and before it finished
- All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- Consequence: every operation returns an "up-to-date" value, a.k.a. "strong consistency"
- Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

Linearizability

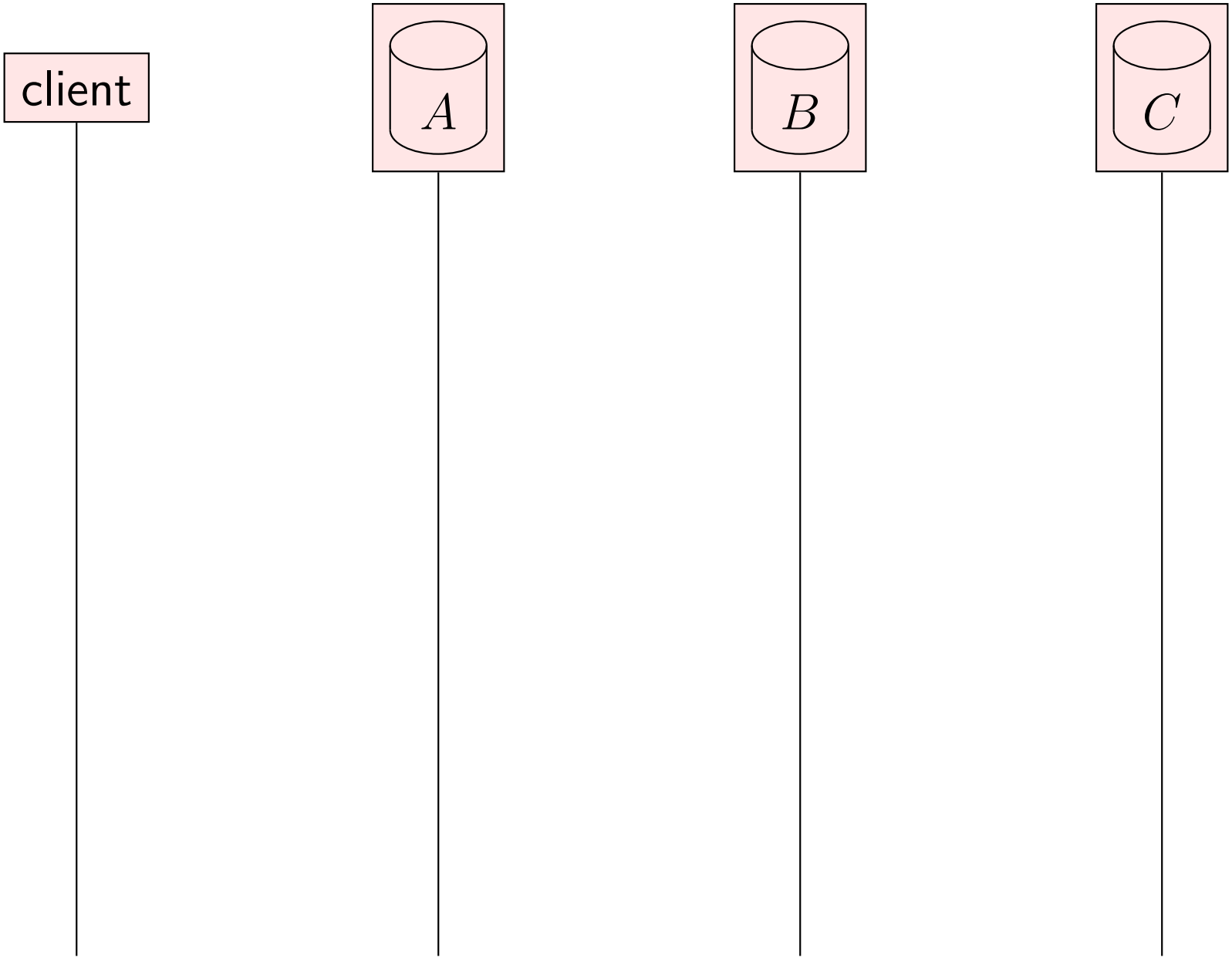
Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

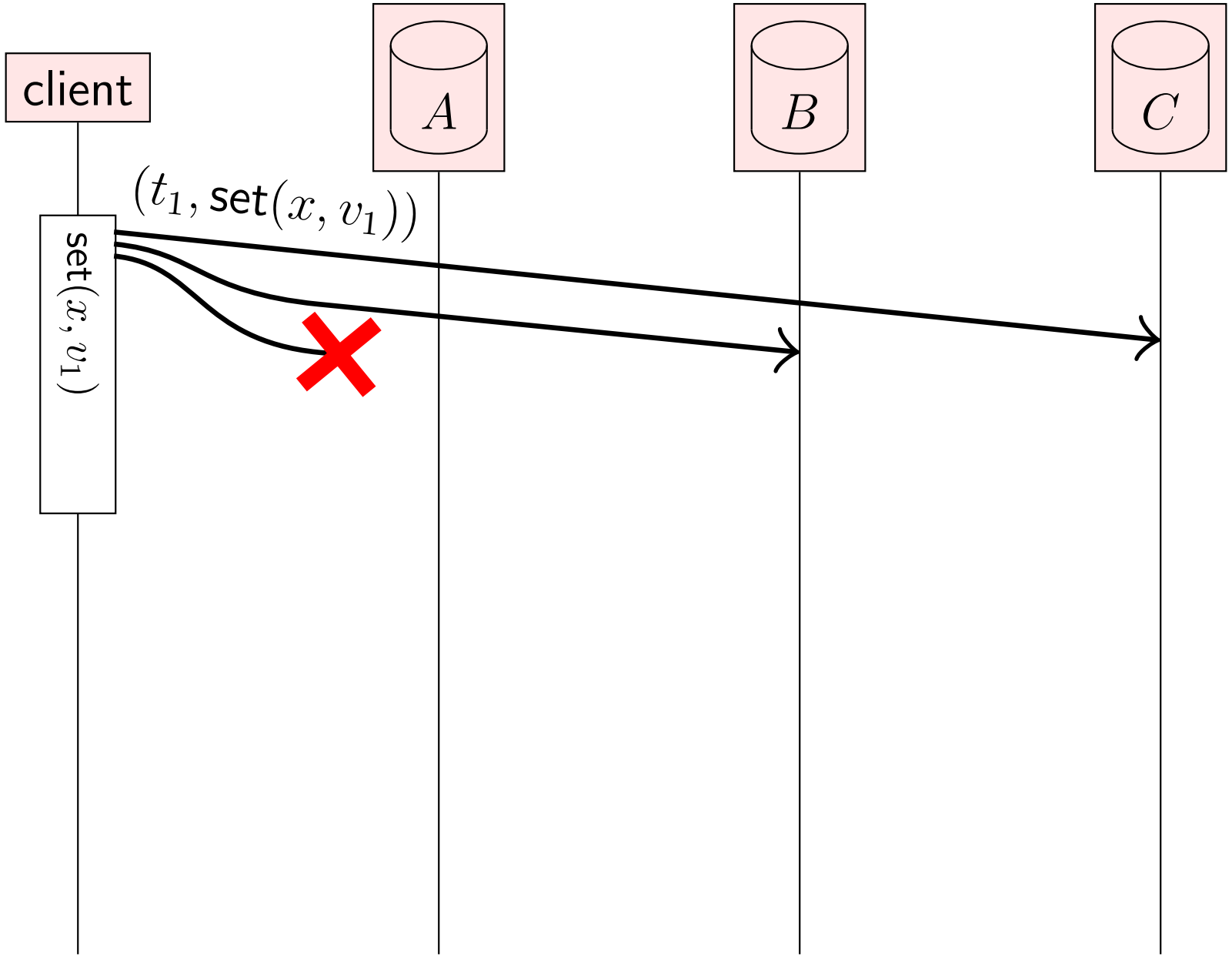
- Informally: every operation takes effect **atomically** sometime after it started and before it finished
- All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- Consequence: every operation returns an "up-to-date" value, a.k.a. "strong consistency"
- Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

Note: linearizability $\neq$ serializability!

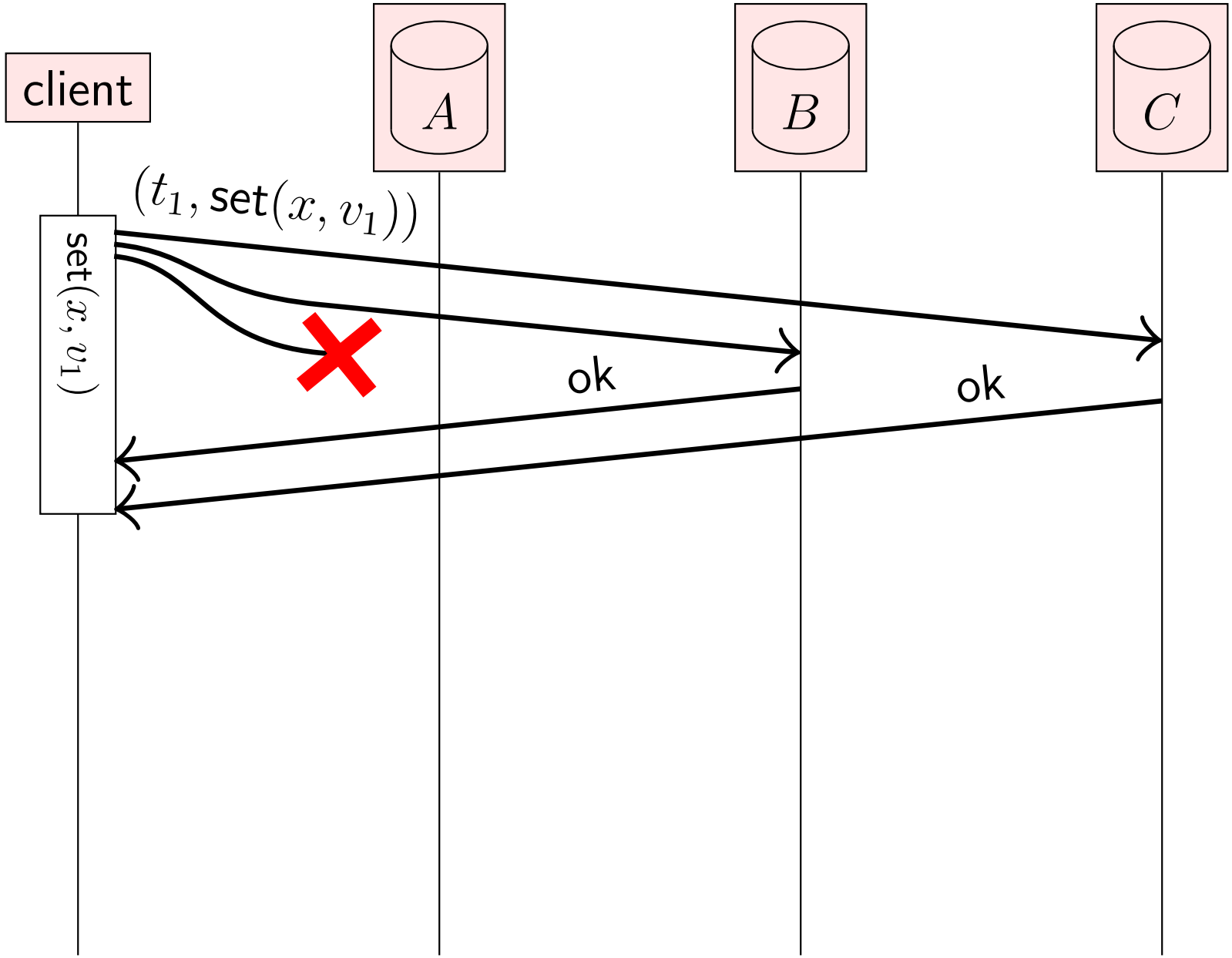
Read-after-write consistency revisited



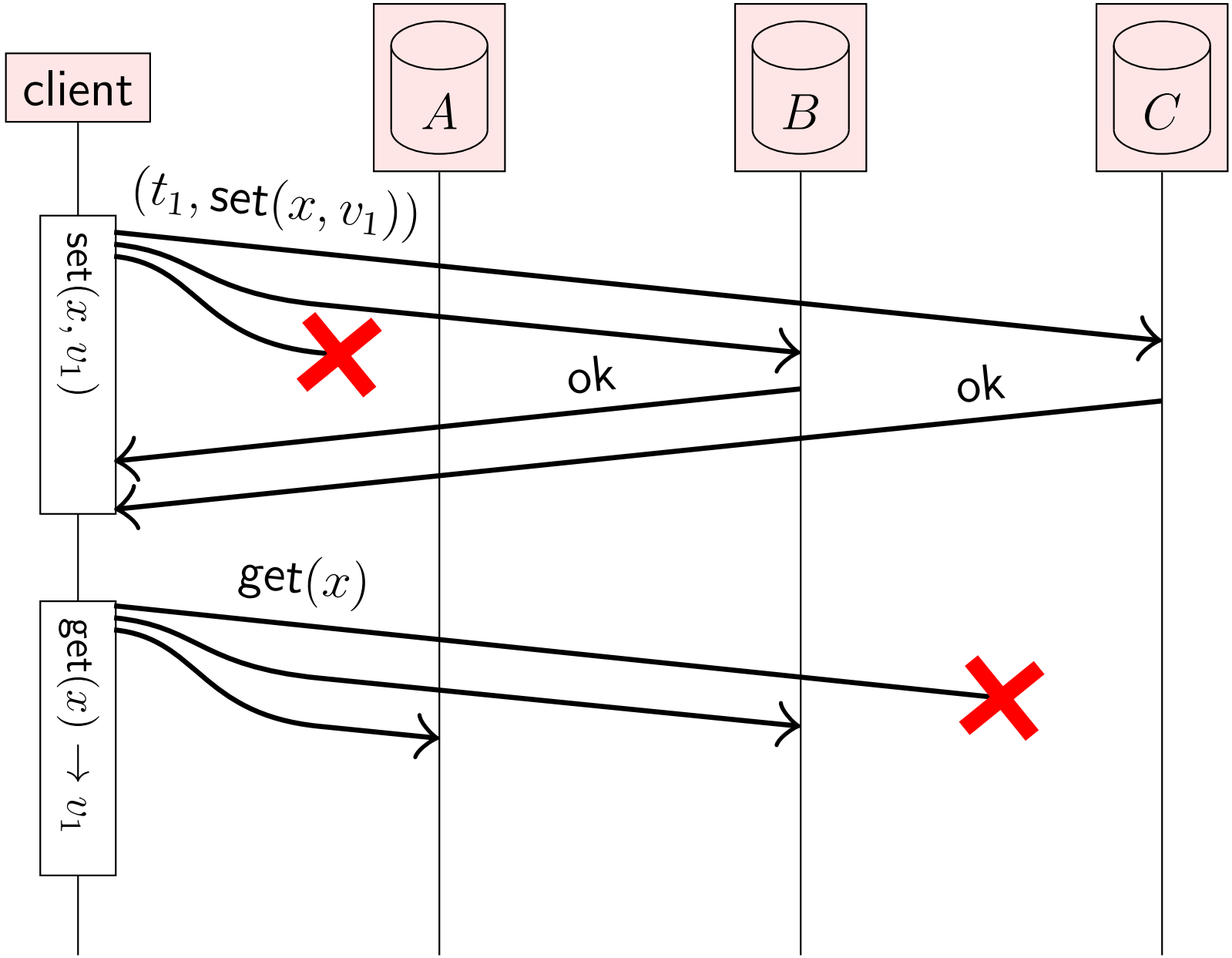
Read-after-write consistency revisited



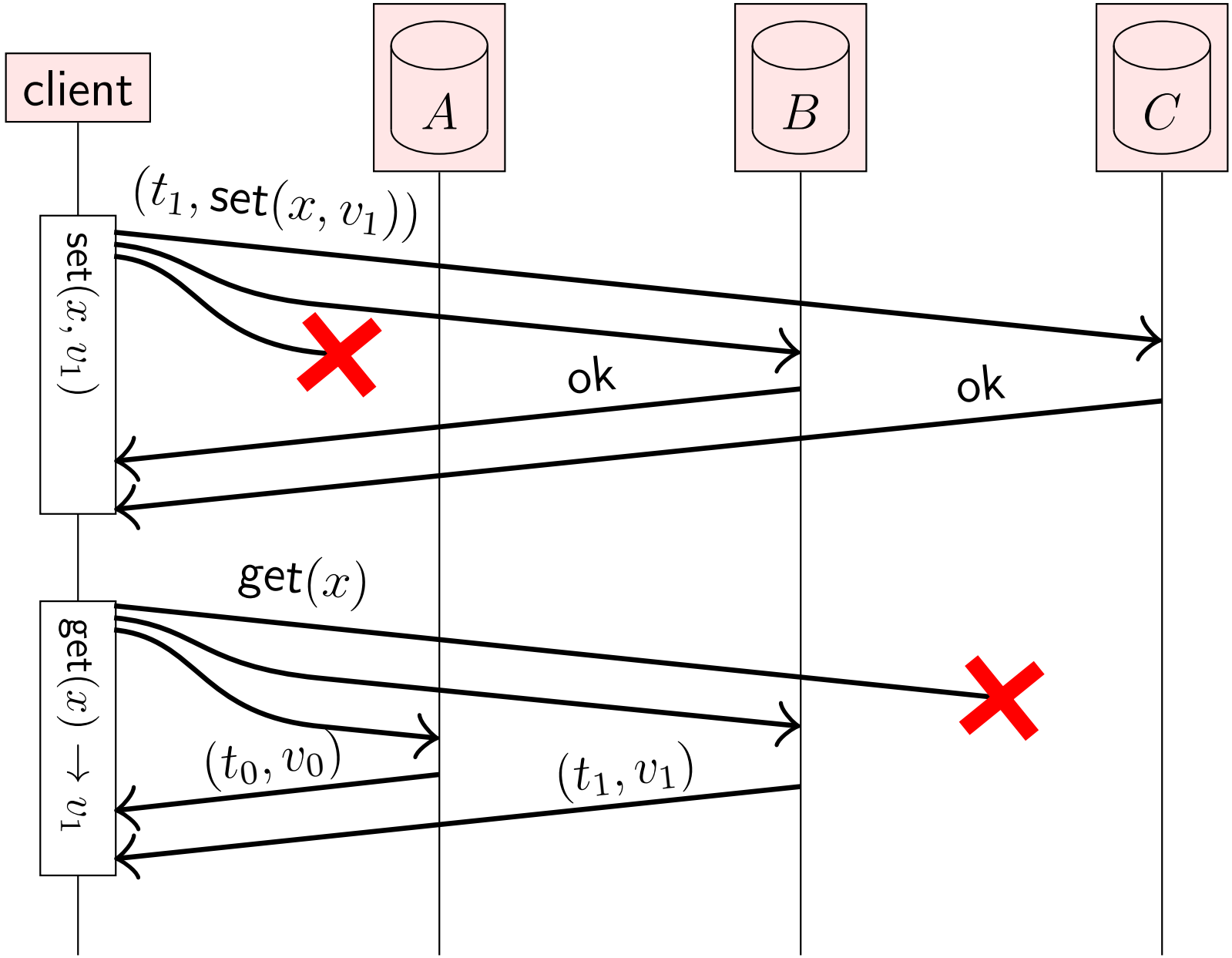
Read-after-write consistency revisited



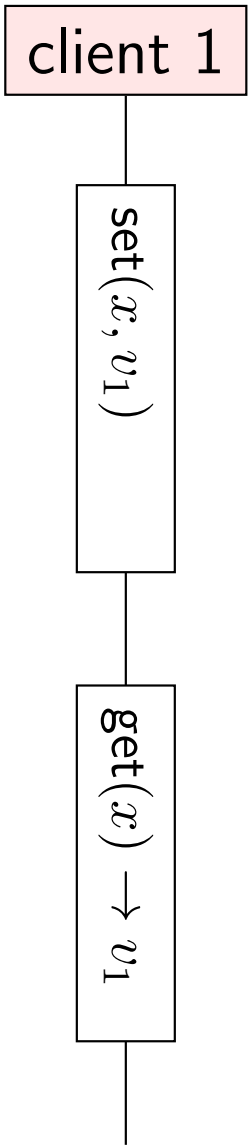
Read-after-write consistency revisited



Read-after-write consistency revisited

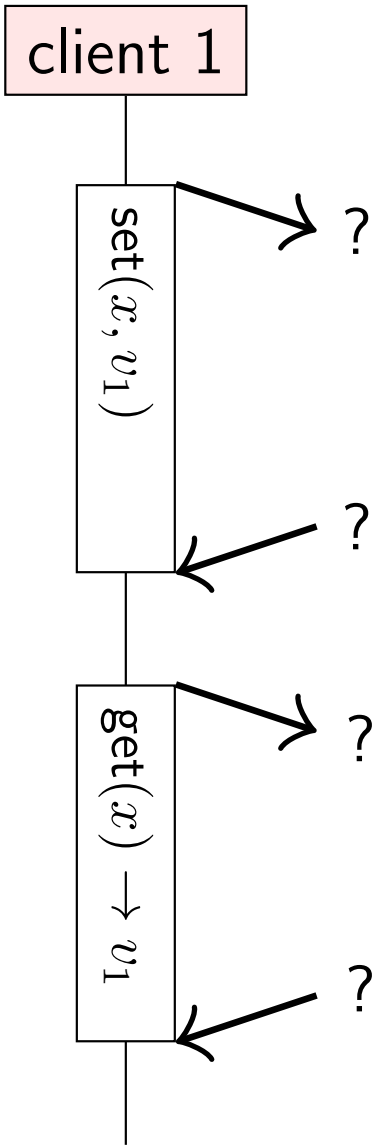


From the client's point of view



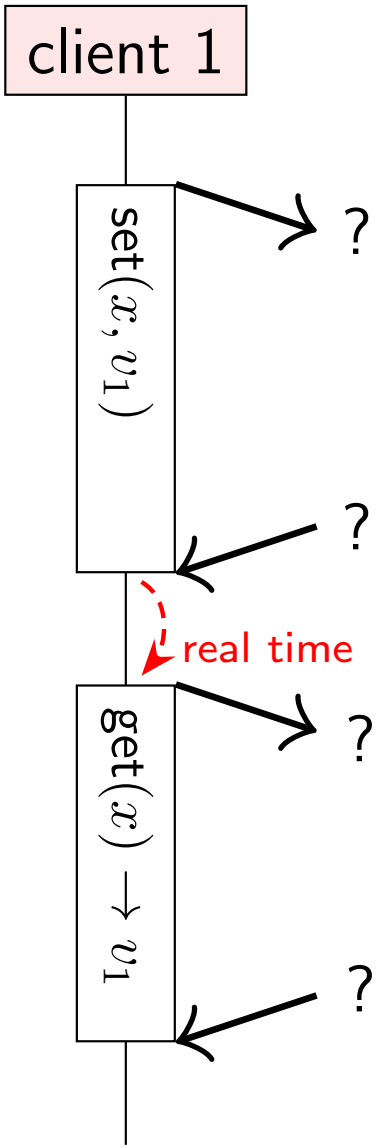
- Focus on client-observable behaviour: when and what an operation returns

From the client's point of view



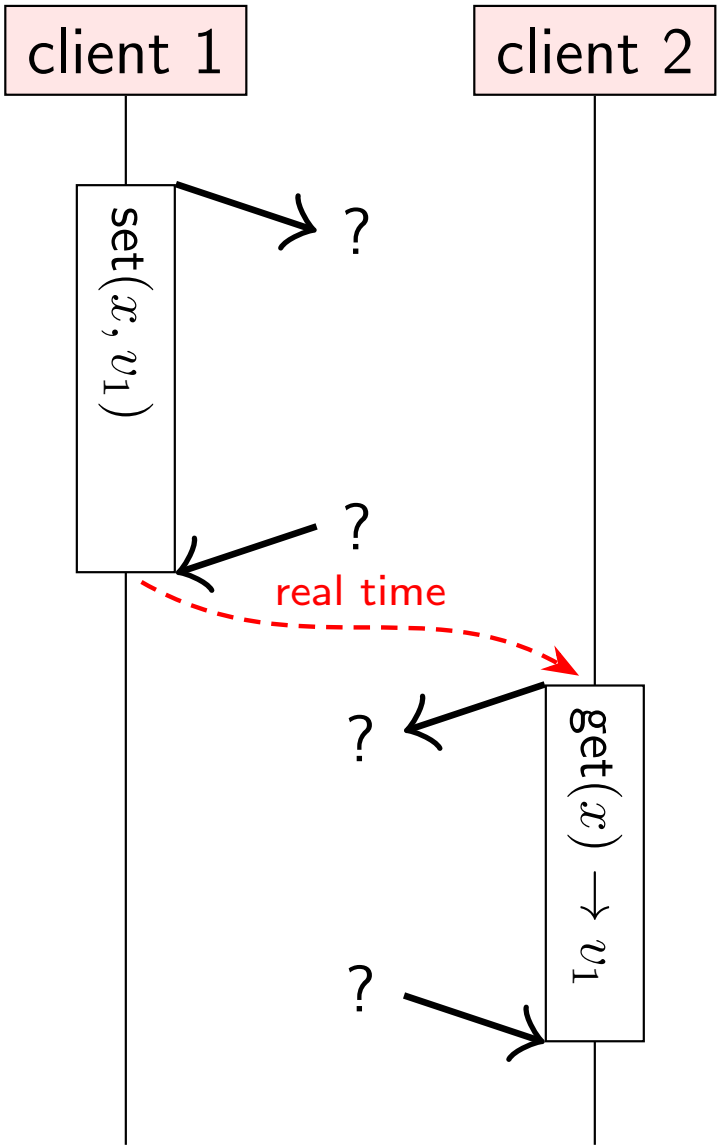
- Focus on client-observable behaviour: when and what an operation returns
- Ignore how the replication system is implemented internally

From the client's point of view



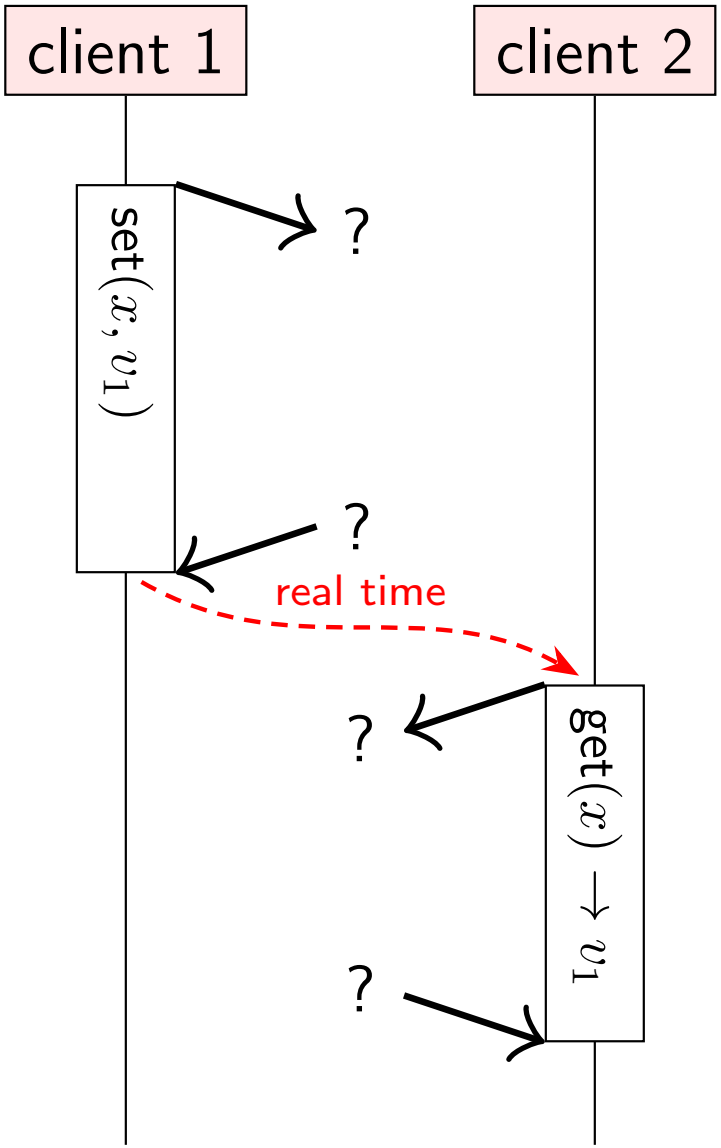
- Focus on client-observable behaviour: when and what an operation returns
- Ignore how the replication system is implemented internally
- Did operation A finish before operation B started?

From the client's point of view



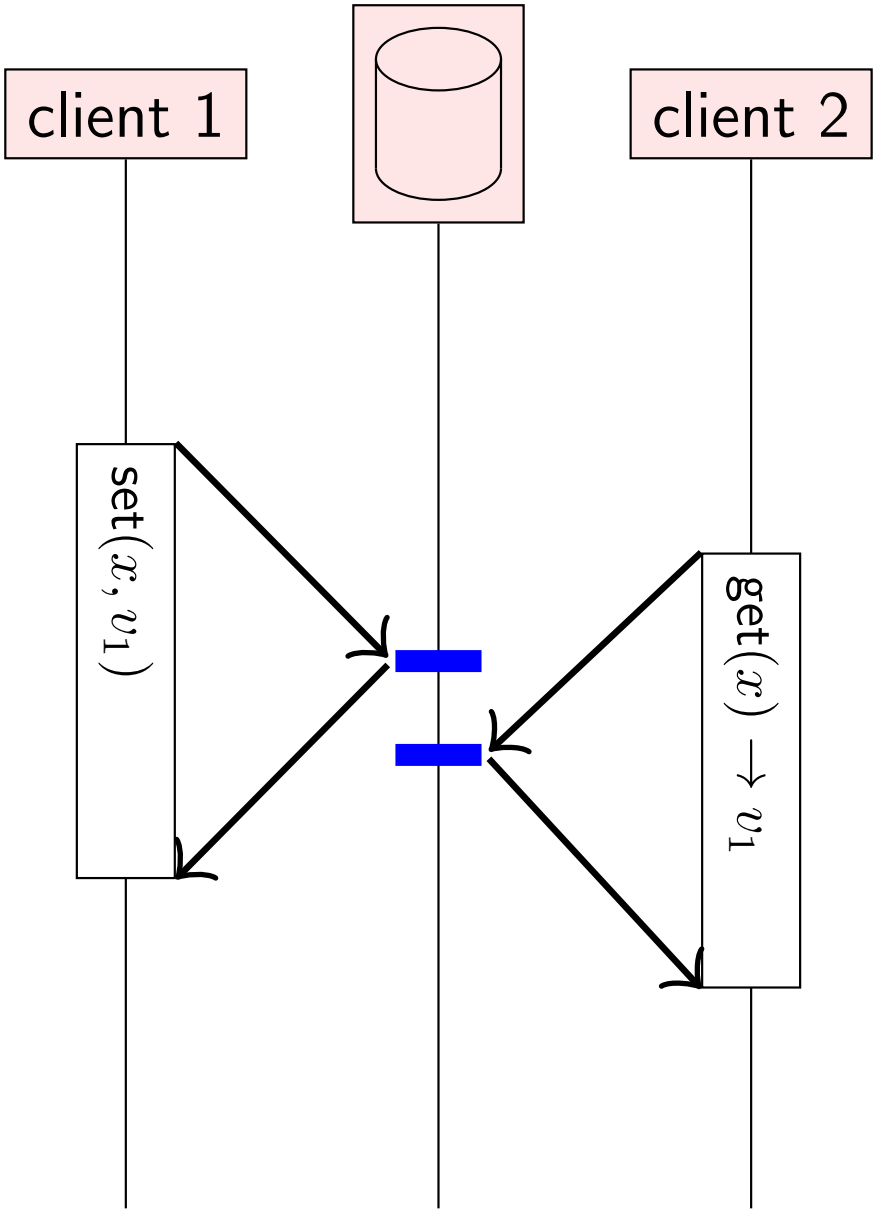
- Focus on client-observable behaviour: when and what an operation returns
- Ignore how the replication system is implemented internally
- Did operation A finish before operation B started?
- Even if the operations are on different nodes?

From the client's point of view



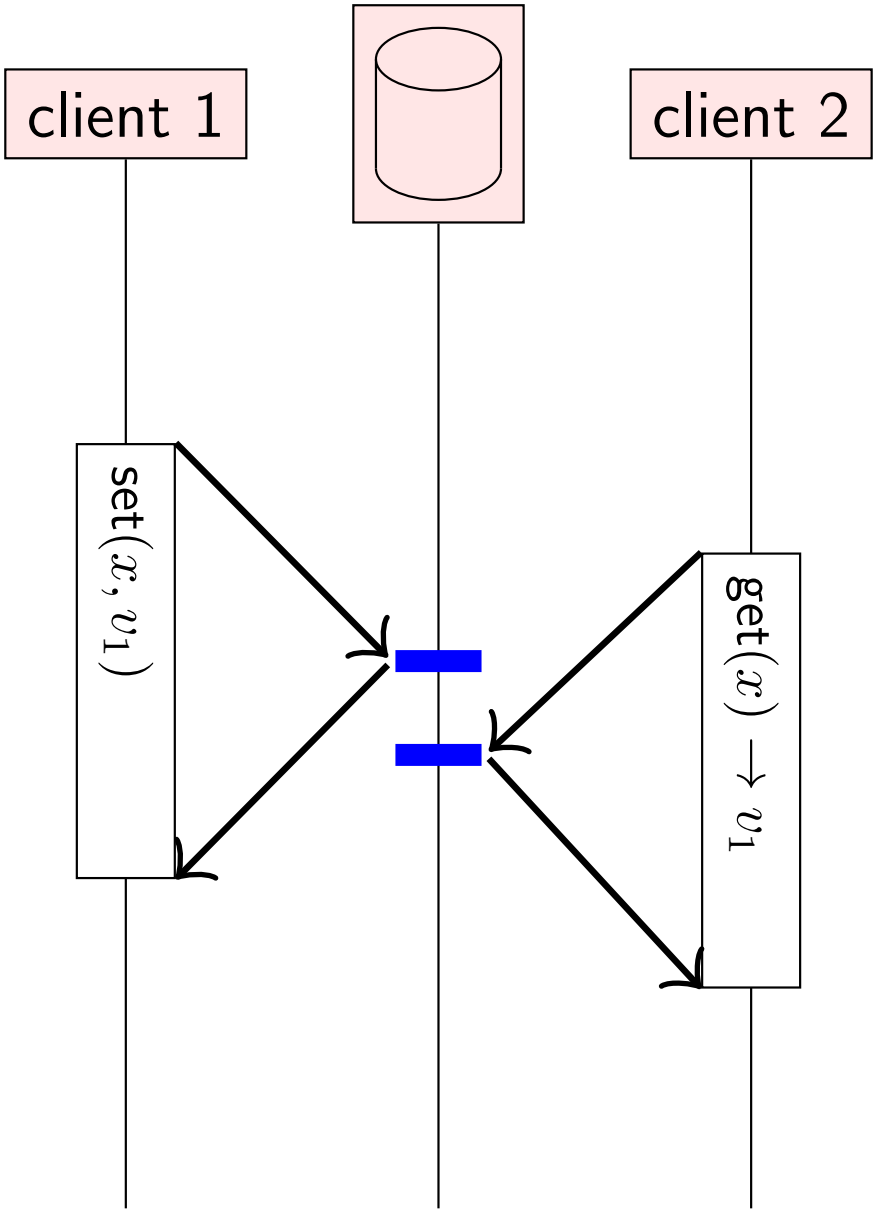
- Focus on client-observable behaviour: when and what an operation returns
- Ignore how the replication system is implemented internally
- Did operation A finish before operation B started?
- Even if the operations are on different nodes?
- **This is not happens-before**: we want client 2 to read value written by client 1, even if the clients have not communicated!

Operations overlapping in time



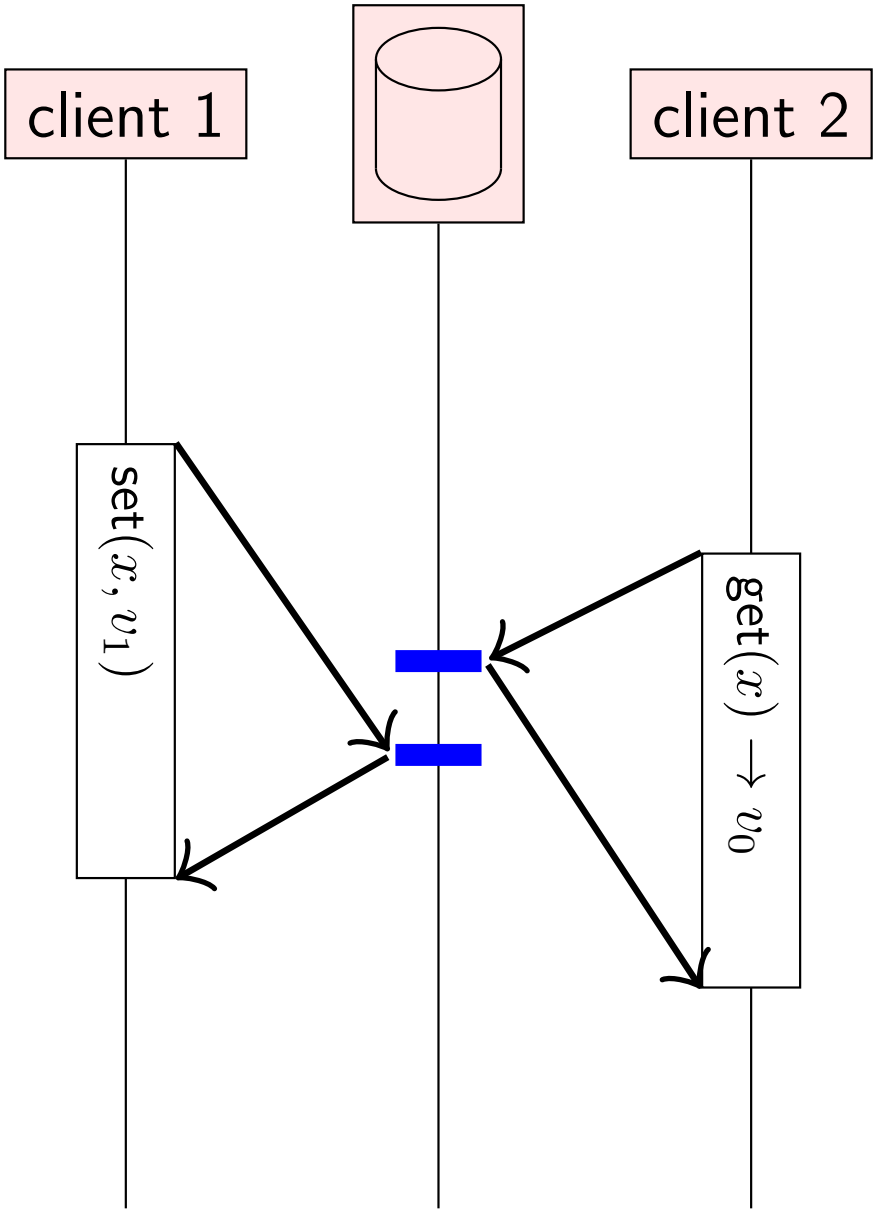
- Client 2's get operation overlaps in time with client 1's set operation

Operations overlapping in time



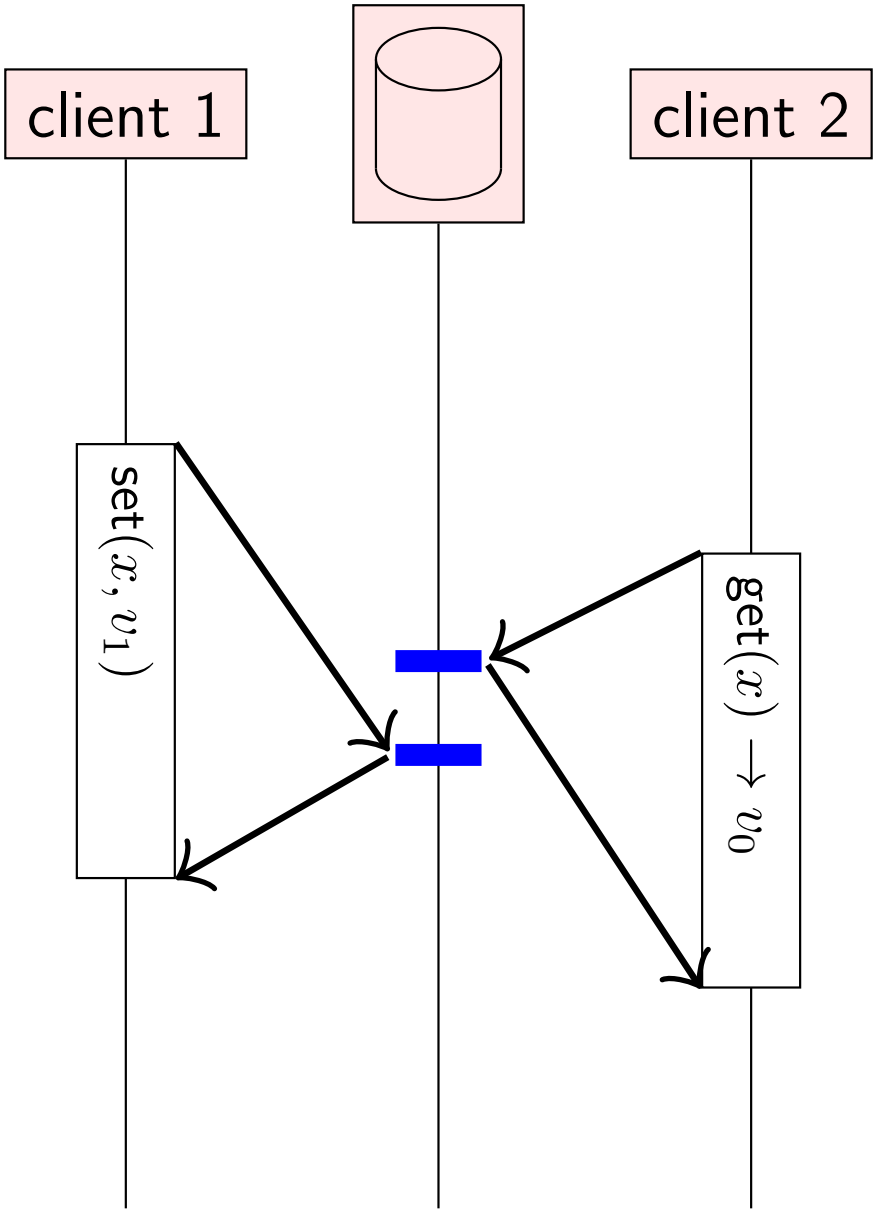
- Client 2's get operation overlaps in time with client 1's set operation
- Maybe the set operation takes effect first?

Operations overlapping in time



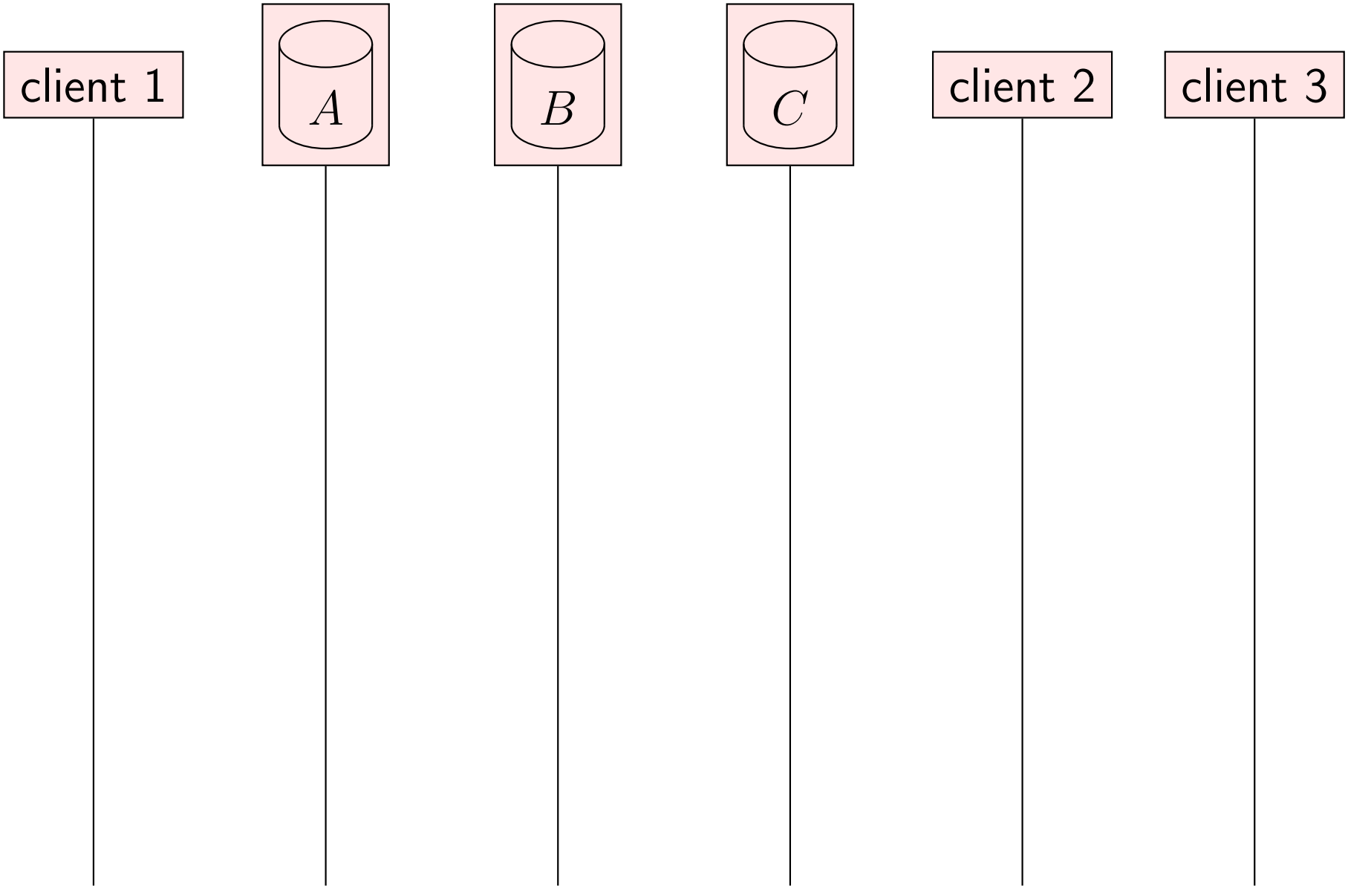
- Client 2's get operation overlaps in time with client 1's set operation
- Maybe the set operation takes effect first?
- Just as likely, the get operation may be executed first

Operations overlapping in time

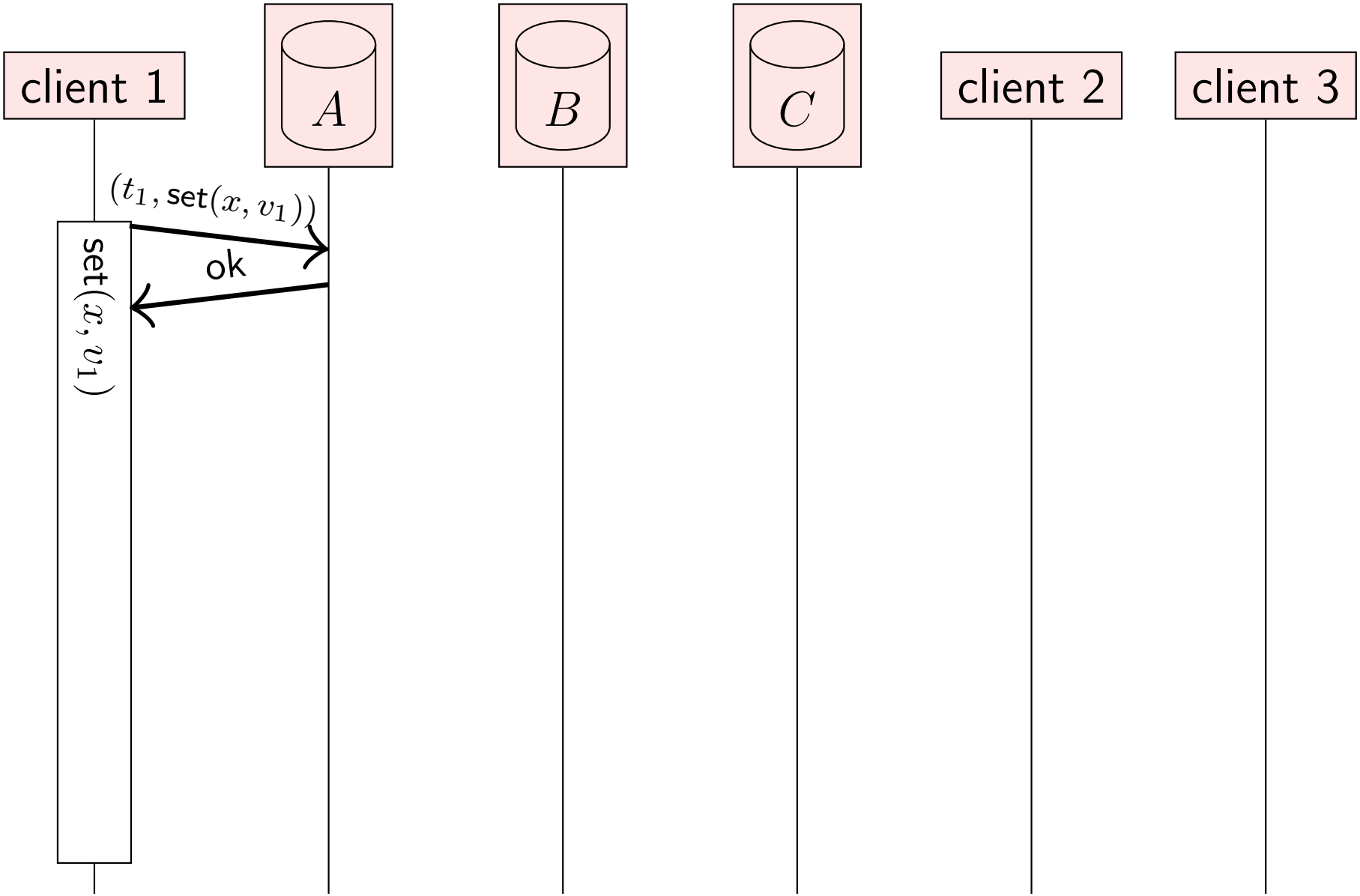


- Client 2's get operation overlaps in time with client 1's set operation
- Maybe the set operation takes effect first?
- Just as likely, the get operation may be executed first
- Either outcome is fine in this case

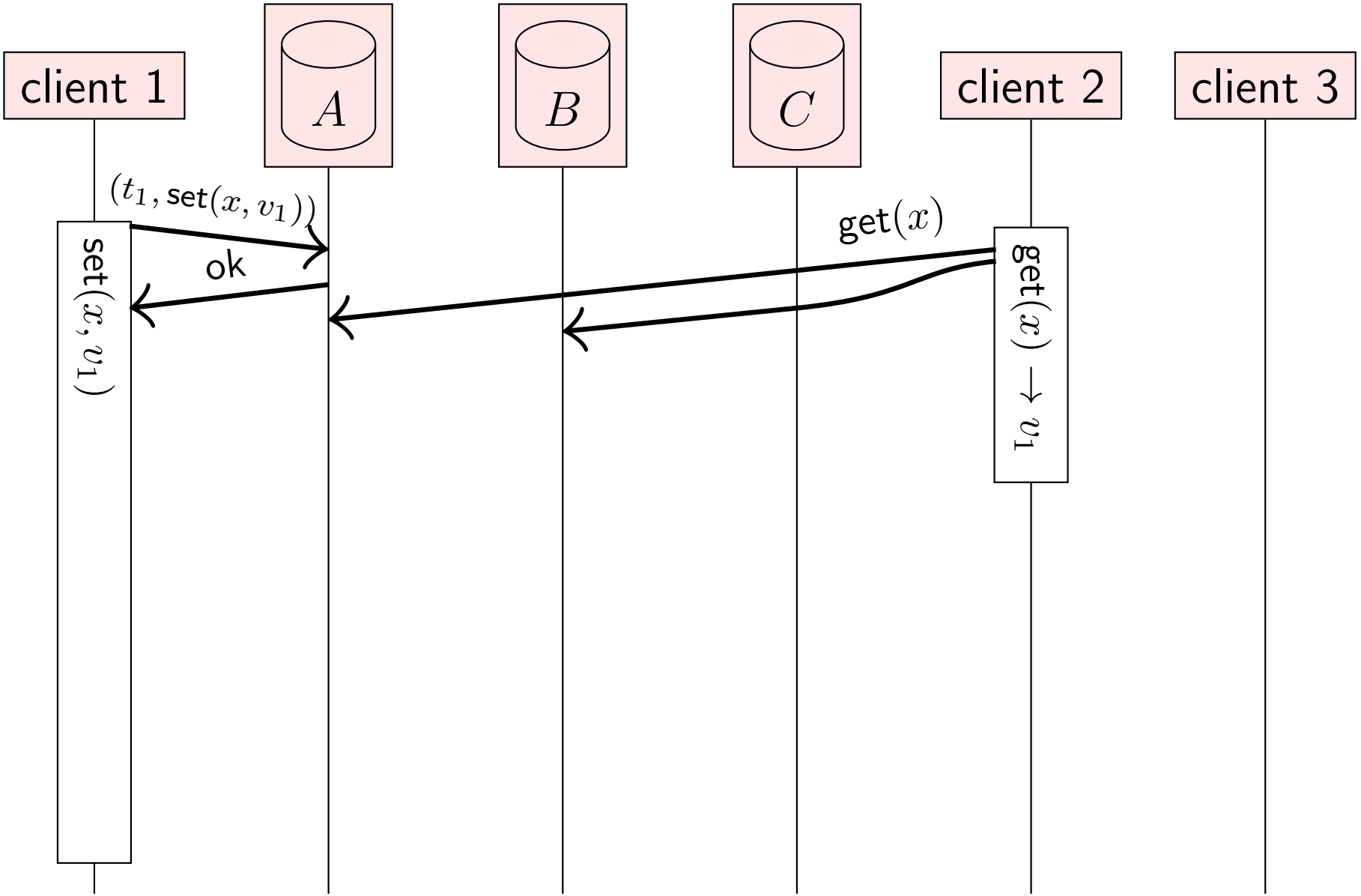
Not linearizable, despite quorum reads/writes



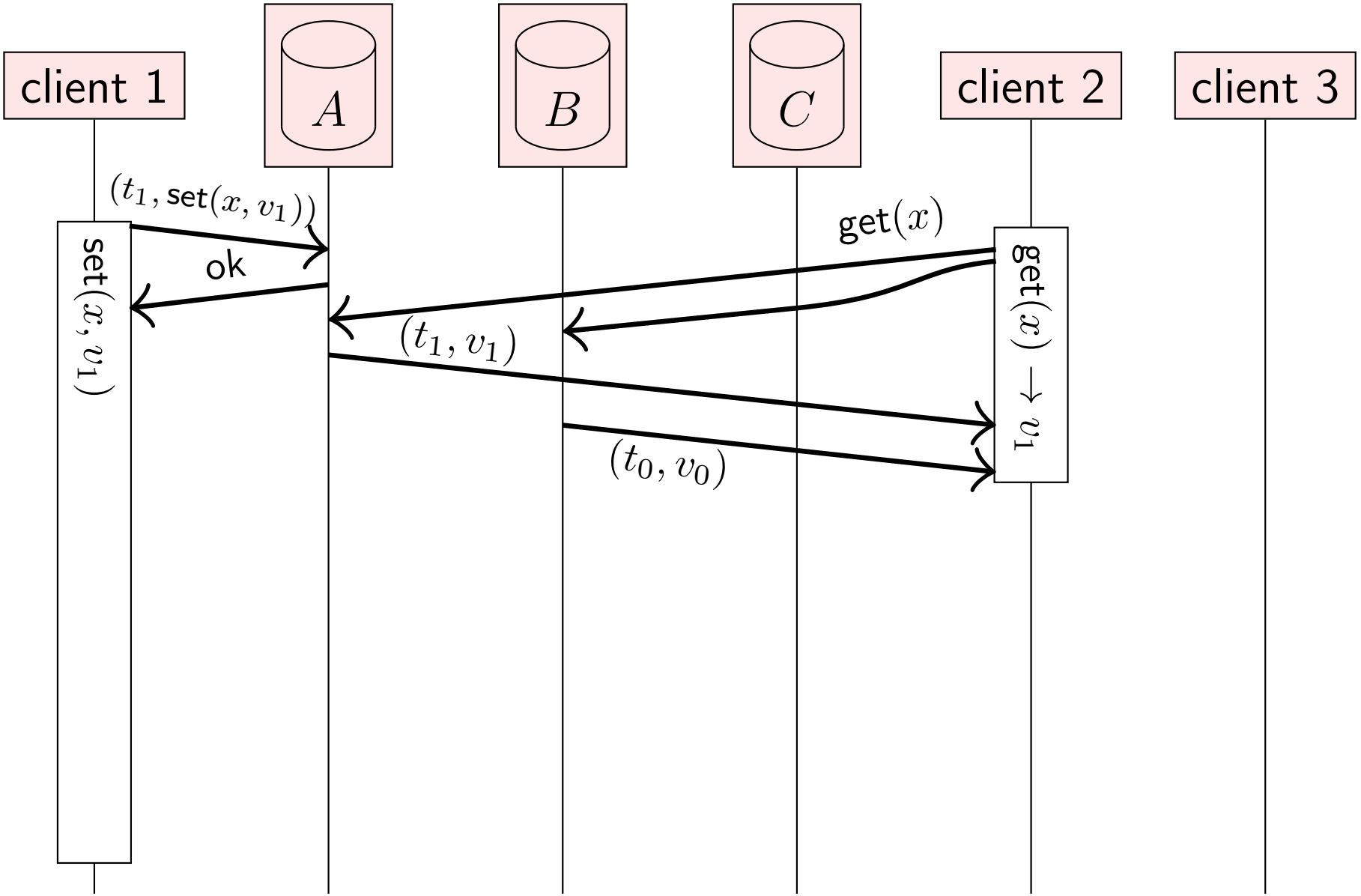
Not linearizable, despite quorum reads/writes



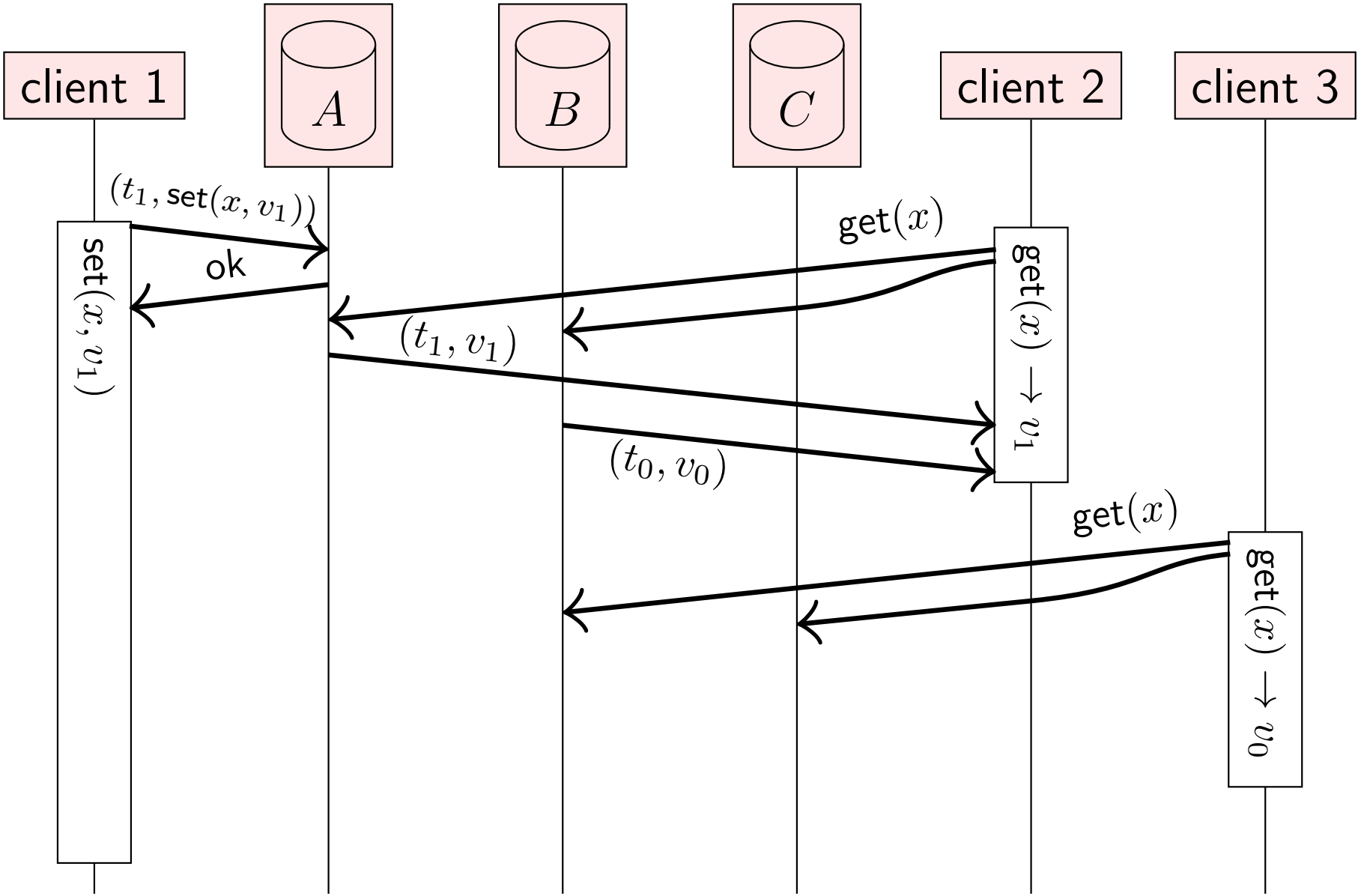
Not linearizable, despite quorum reads/writes



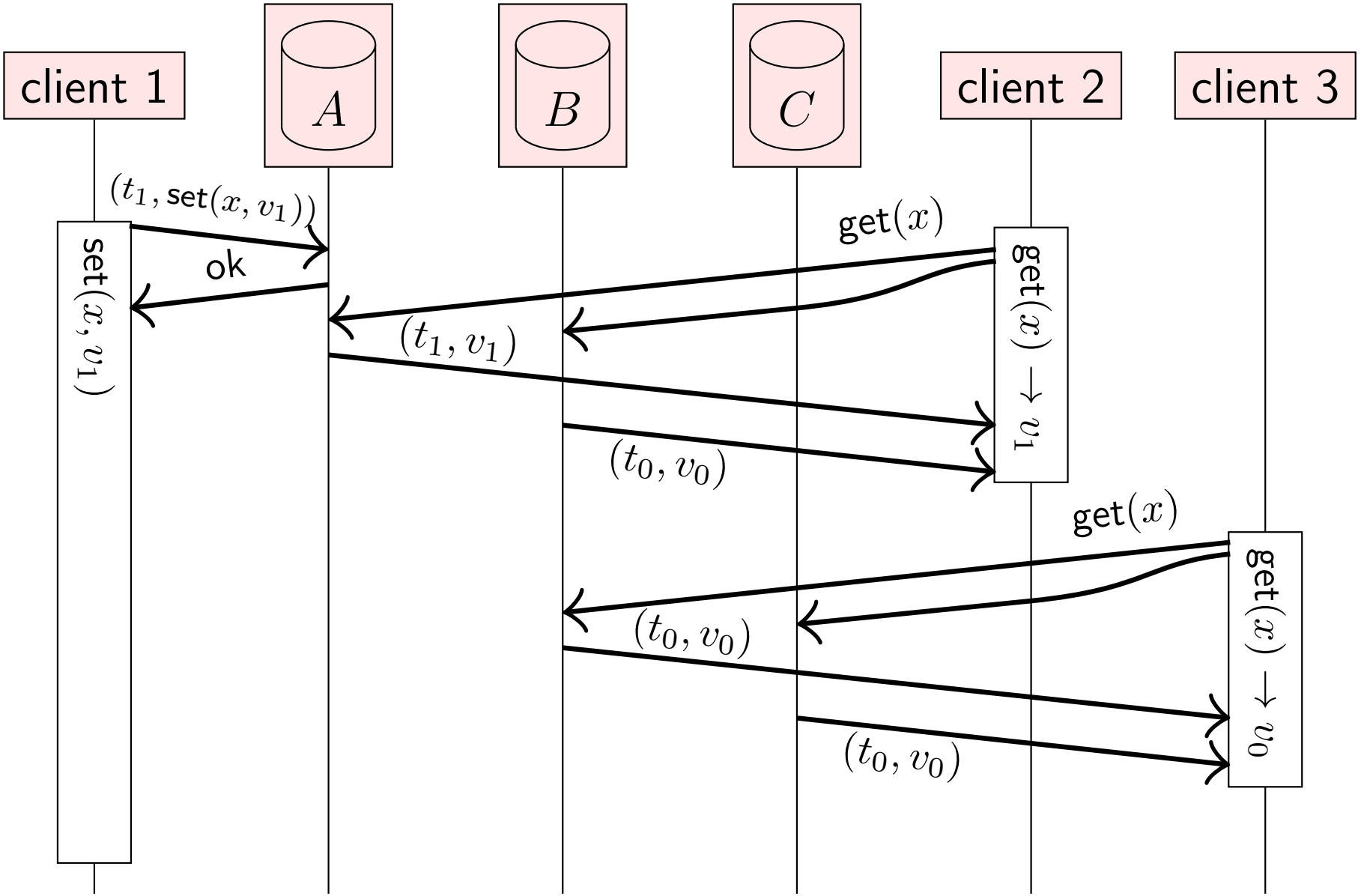
Not linearizable, despite quorum reads/writes



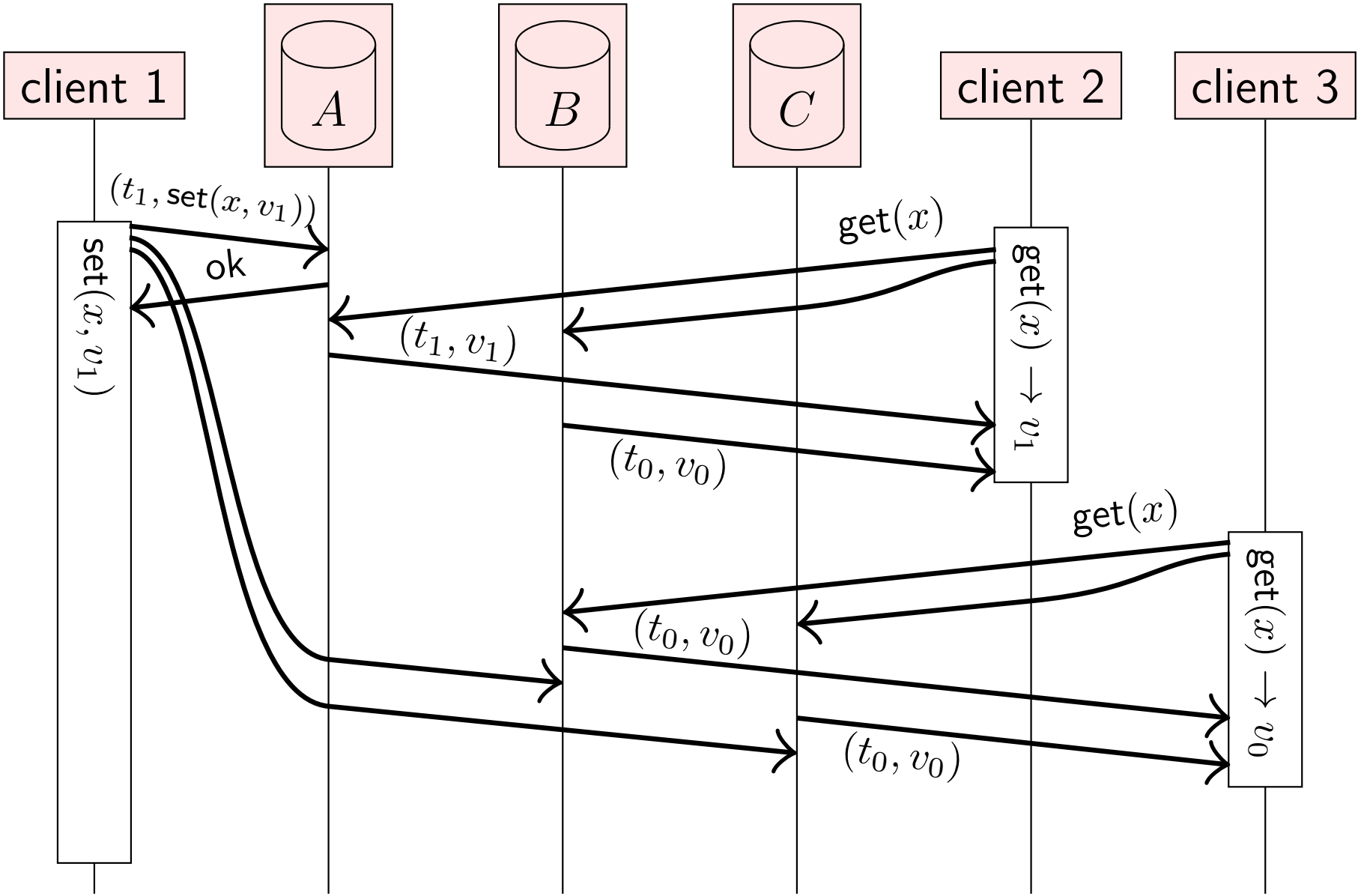
Not linearizable, despite quorum reads/writes



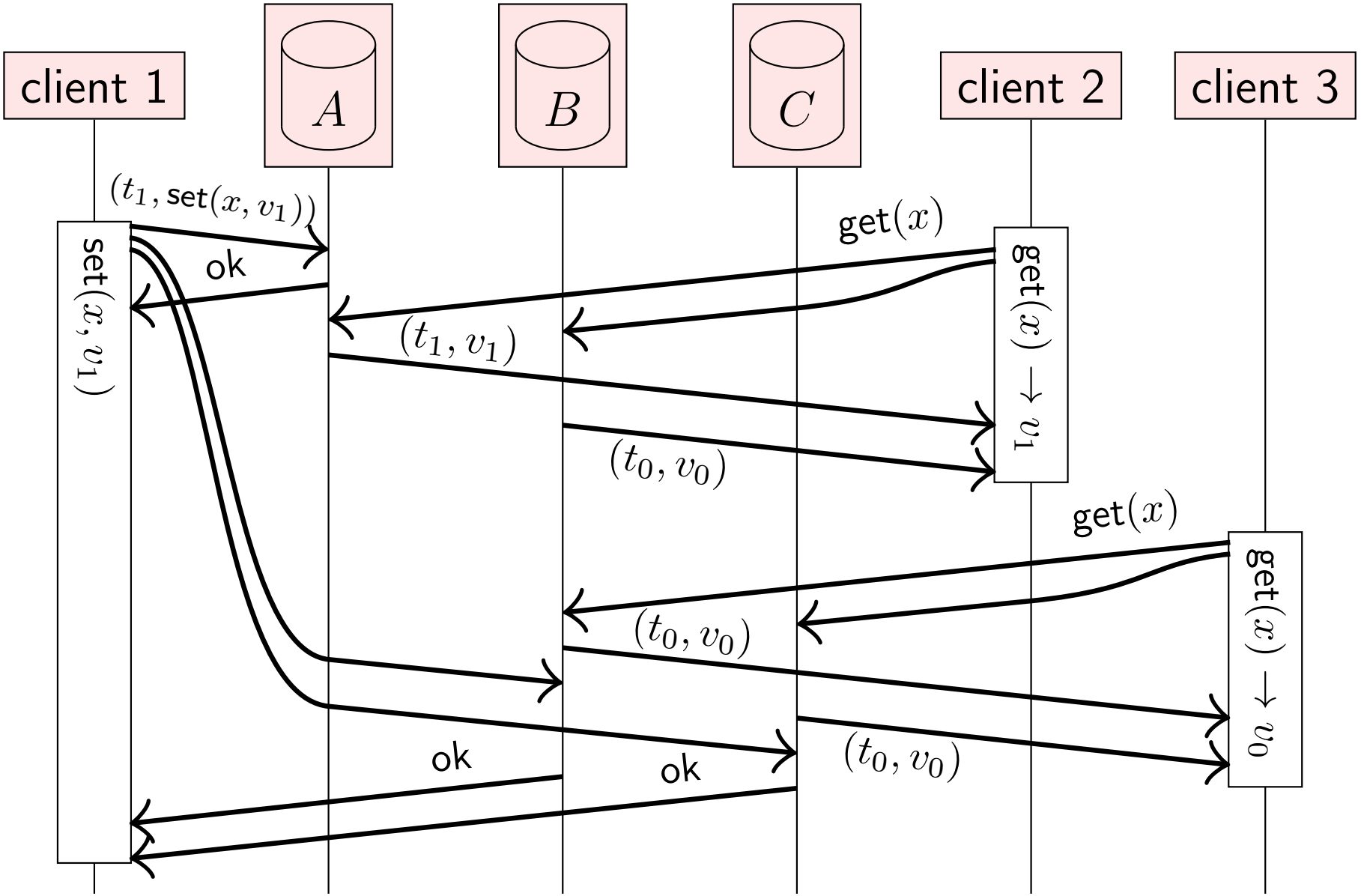
Not linearizable, despite quorum reads/writes



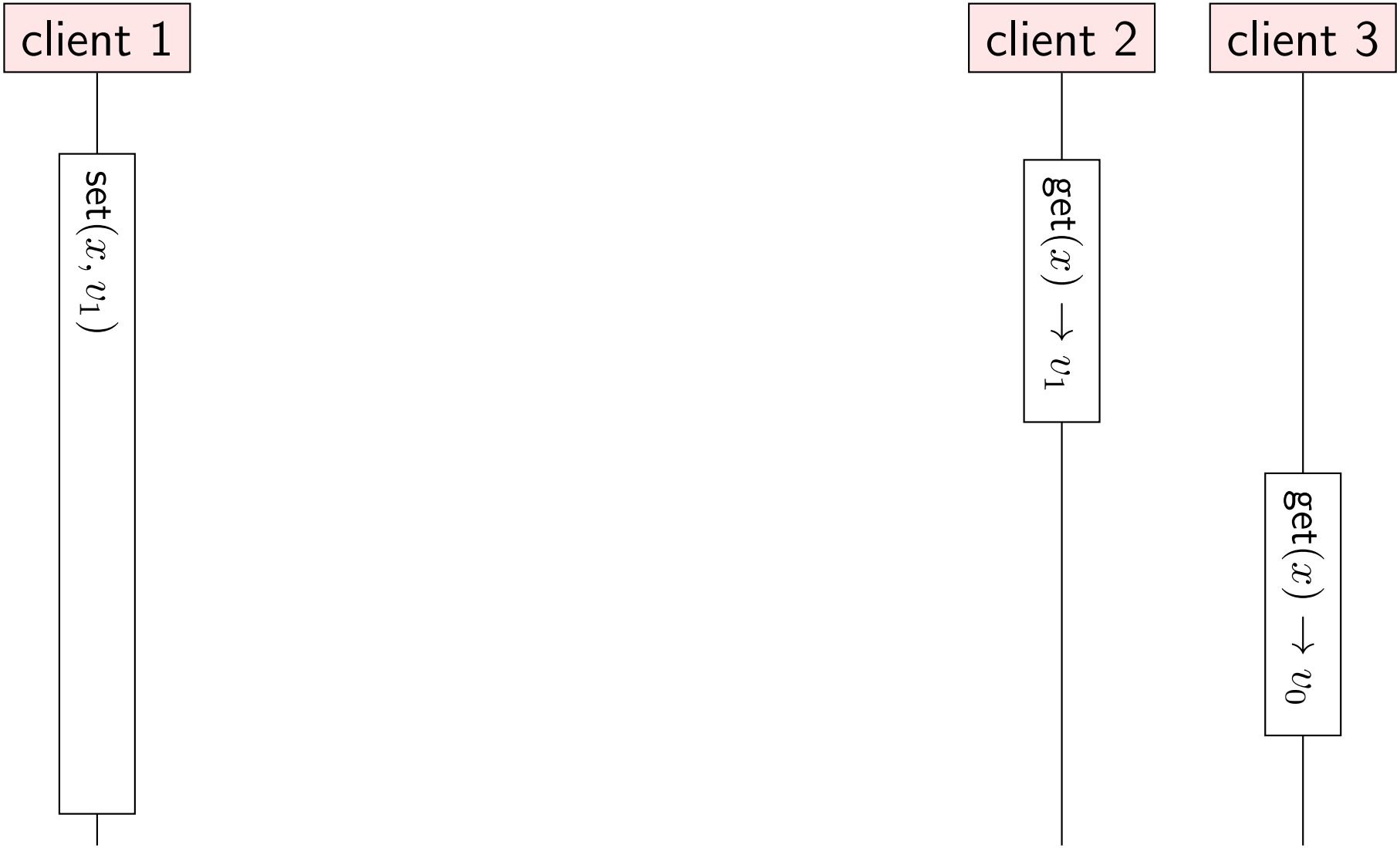
Not linearizable, despite quorum reads/writes



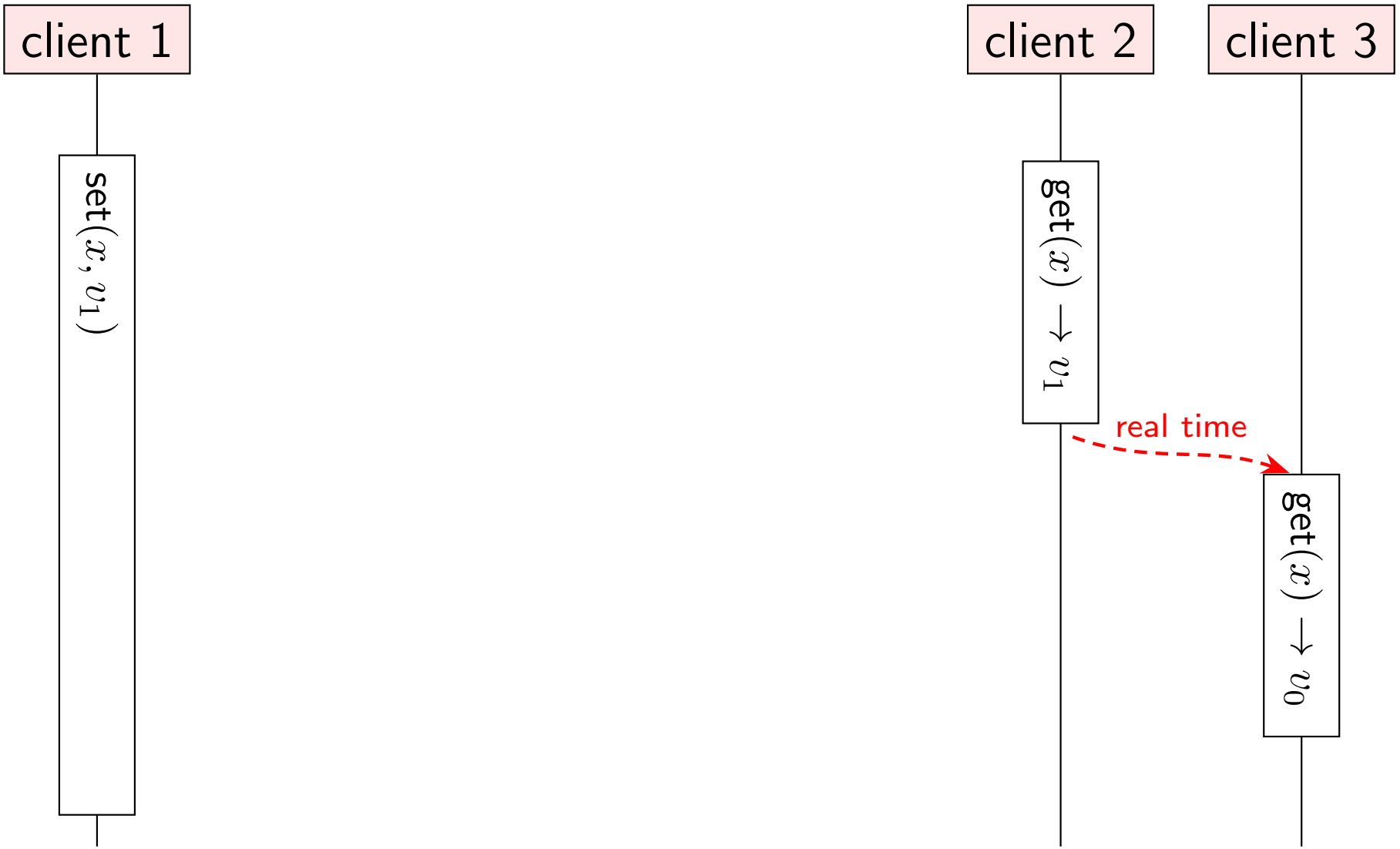
Not linearizable, despite quorum reads/writes



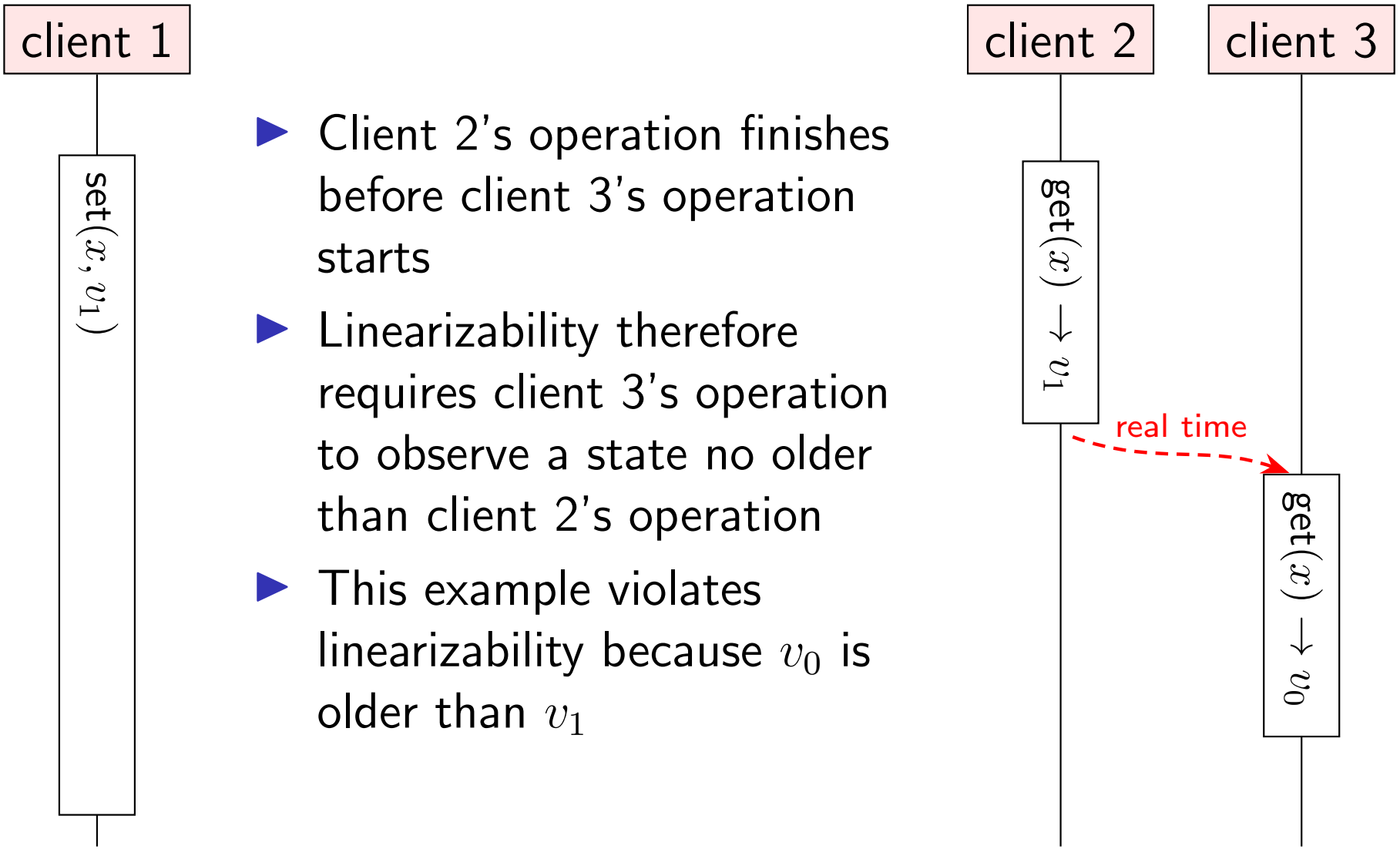
Not linearizable, despite quorum reads/writes



Not linearizable, despite quorum reads/writes

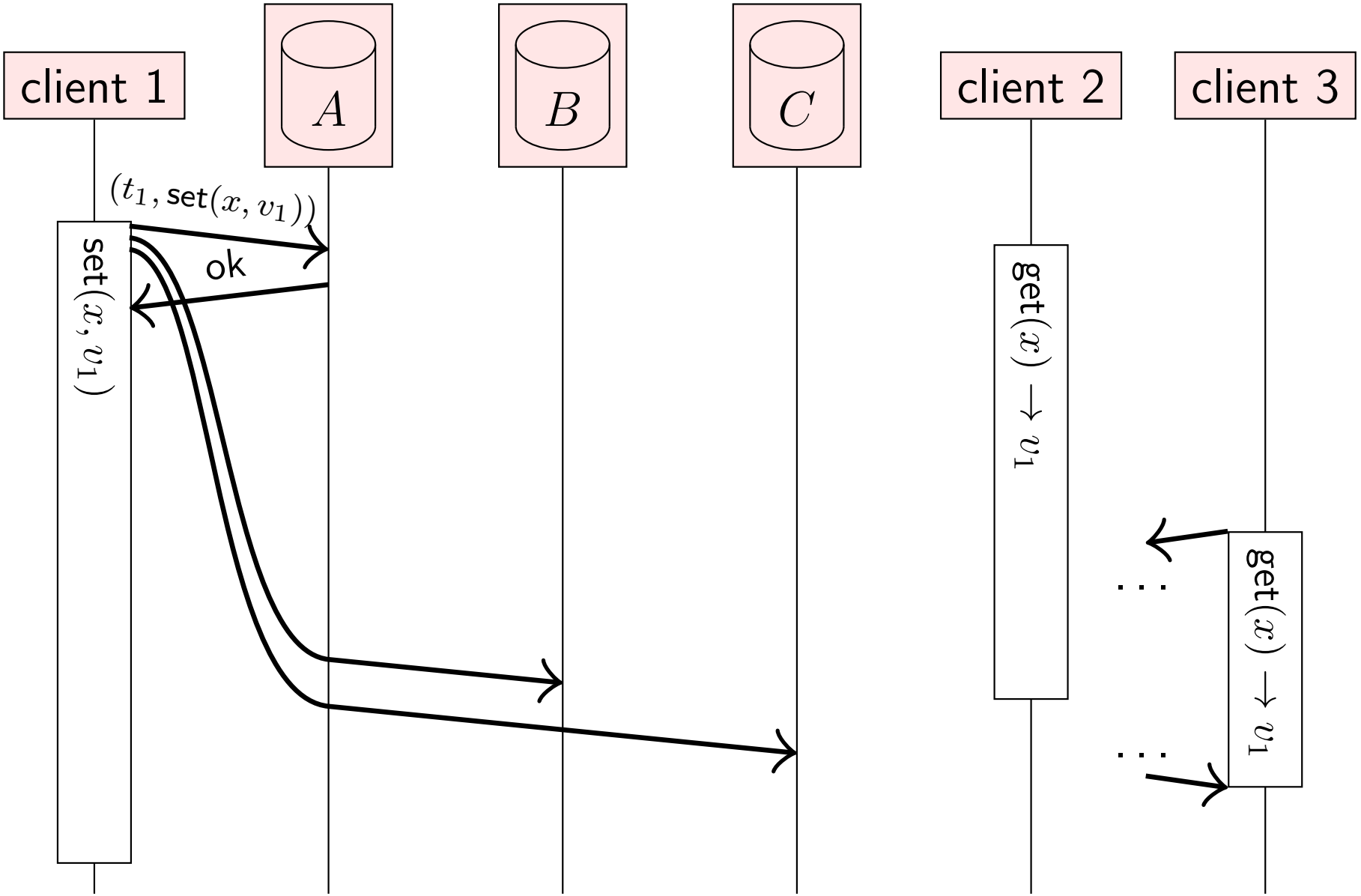


Not linearizable, despite quorum reads/writes

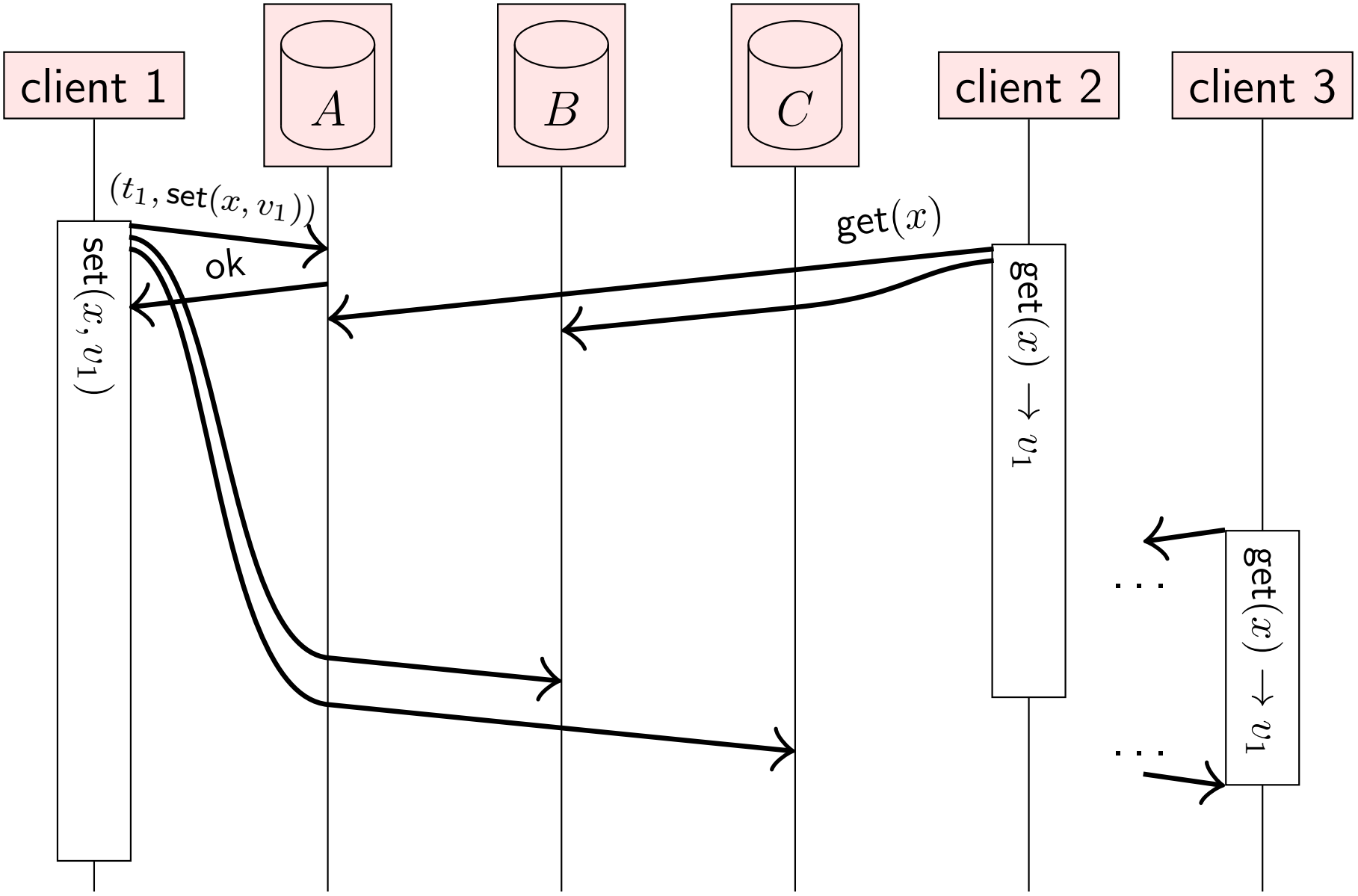


- Client 2's operation finishes before client 3's operation starts
- Linearizability therefore requires client 3's operation to observe a state no older than client 2's operation
- This example violates linearizability because v_0 is older than v_1

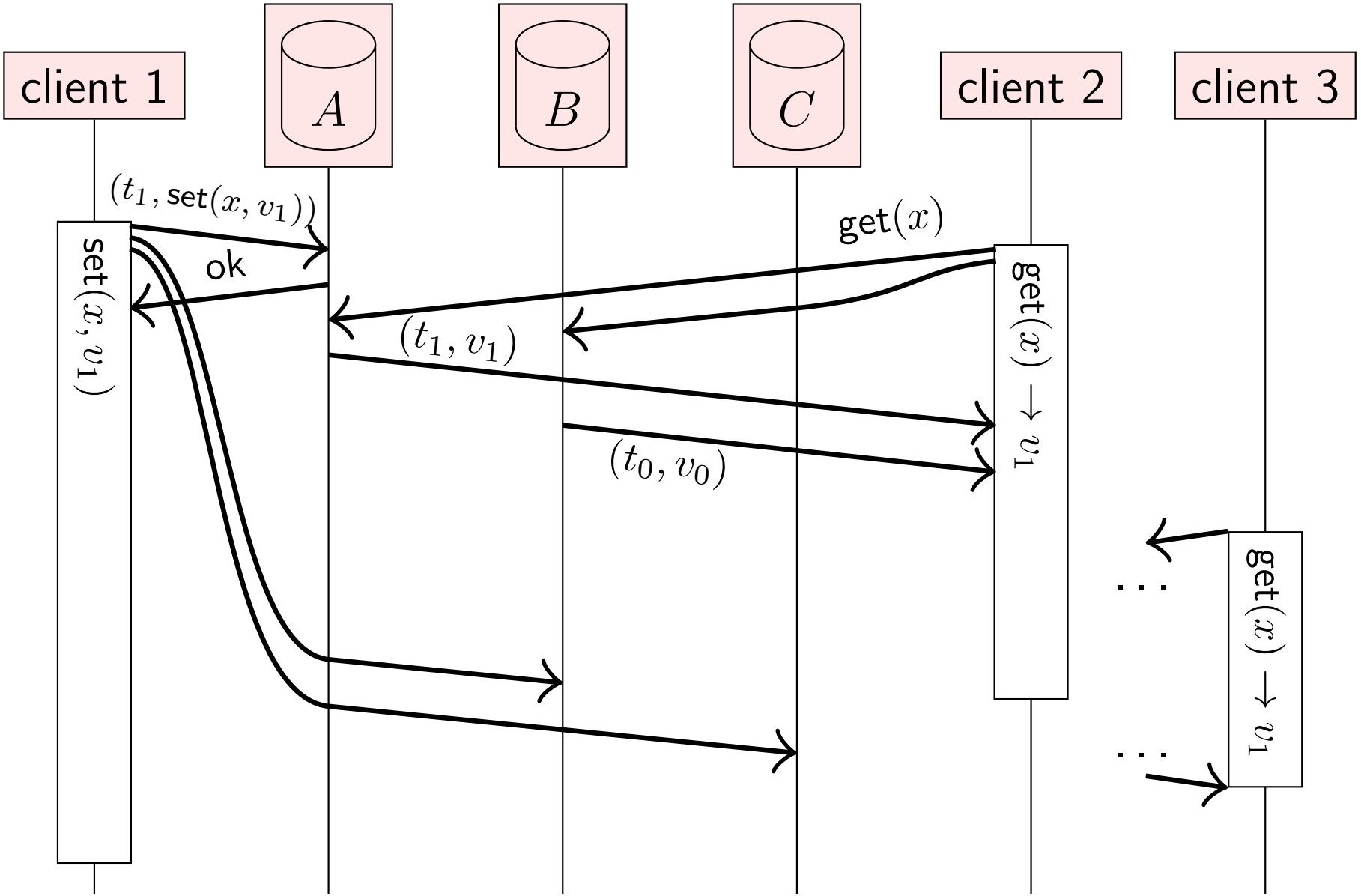
Making quorum reads/writes linearizable



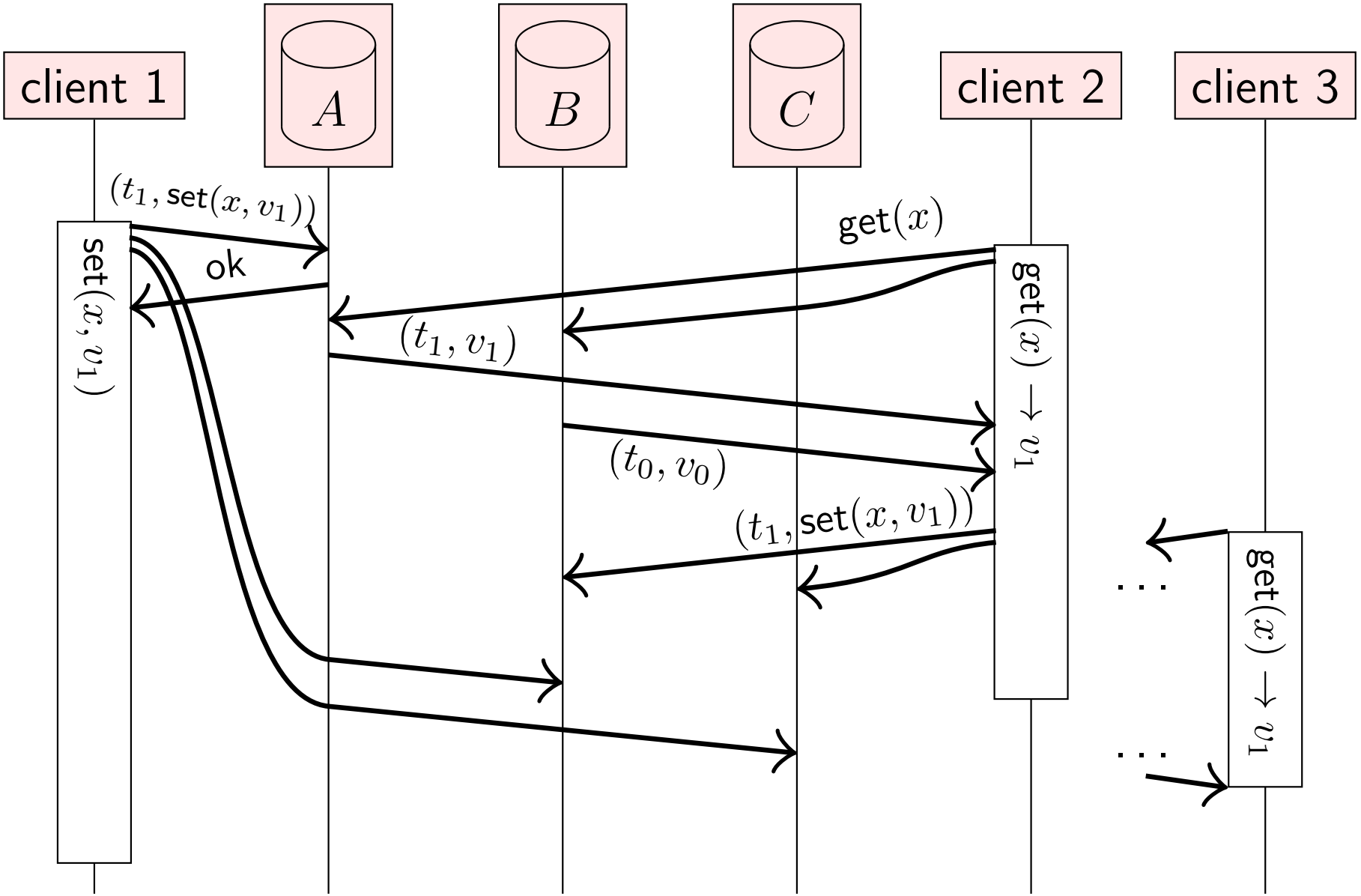
Making quorum reads/writes linearizable



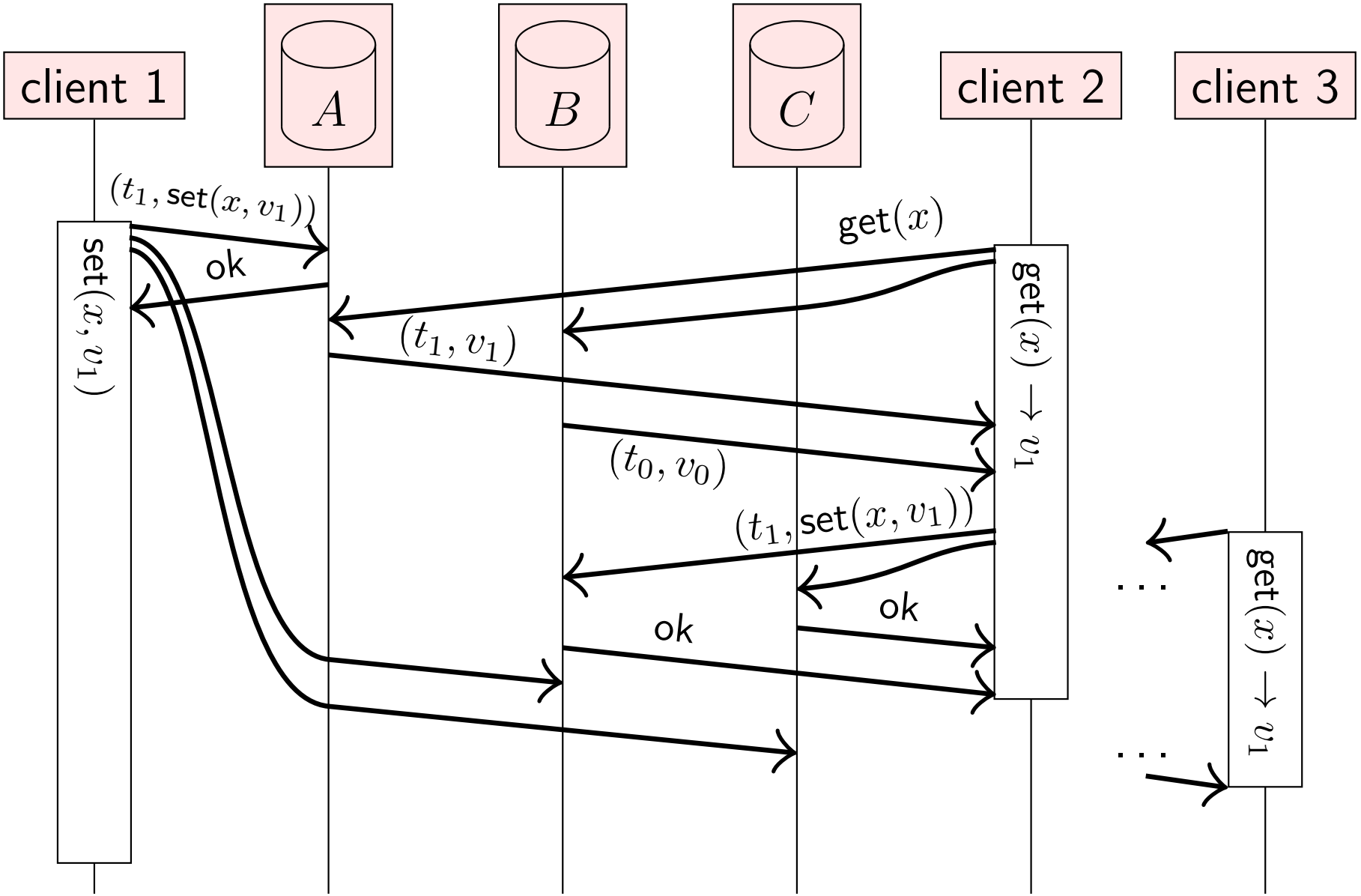
Making quorum reads/writes linearizable



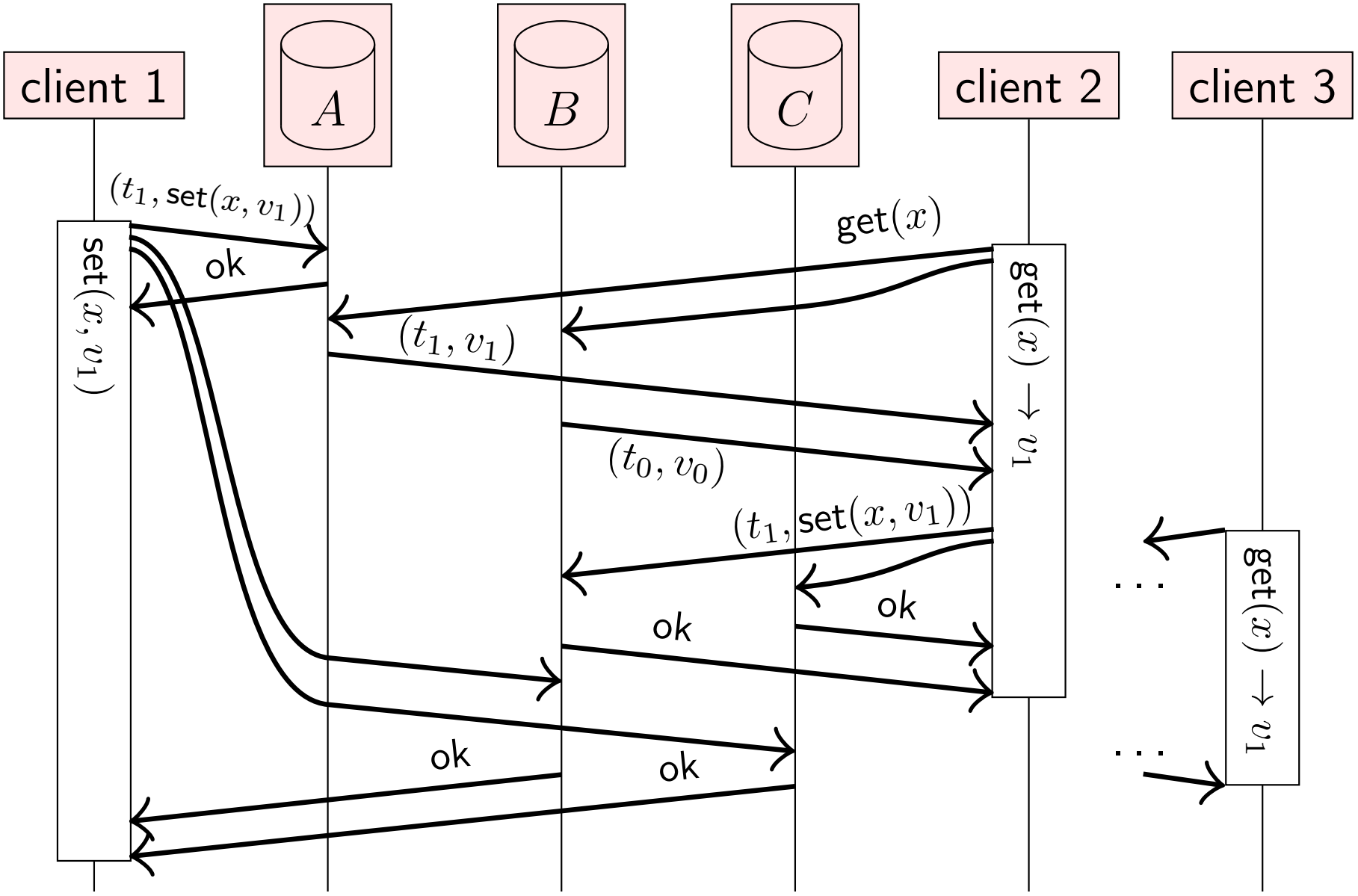
Making quorum reads/writes linearizable



Making quorum reads/writes linearizable



Making quorum reads/writes linearizable



Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- When an operation finishes, the value read/written is stored on a quorum of replicas
- Every subsequent quorum operation will see that value

Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- When an operation finishes, the value read/written is stored on a quorum of replicas
- Every subsequent quorum operation will see that value
- Multiple concurrent writes may overwrite each other

Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- When an operation finishes, the value read/written is stored on a quorum of replicas
- Every subsequent quorum operation will see that value
- Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- CAS$(x, oldValue, newValue)$ sets x to $newValue$ iff current value of x is $oldValue$
- Previously discussed in shared memory concurrency

Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- When an operation finishes, the value read/written is stored on a quorum of replicas
- Every subsequent quorum operation will see that value
- Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- CAS$(x, oldValue, newValue)$ sets x to $newValue$ iff current value of x is $oldValue$
- Previously discussed in shared memory concurrency
- Can we implement **linearizable** compare-and-swap in a distributed system?

Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- When an operation finishes, the value read/written is stored on a quorum of replicas
- Every subsequent quorum operation will see that value
- Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- CAS$(x, oldValue, newValue)$ sets x to $newValue$ iff current value of x is $oldValue$
- Previously discussed in shared memory concurrency
- Can we implement **linearizable** compare-and-swap in a distributed system?
- **Yes:** total order broadcast to the rescue again!

Linearizable compare-and-swap (CAS)

on request to perform get(x) **do**
    total order broadcast (get, x) and wait for delivery
end on

on request to perform CAS(x, old, new) **do**
    total order broadcast $(\text{CAS}, x, old, new)$ and wait for delivery
end on

on delivering (get, x) by total order broadcast **do**
    return $localState[x]$ as result of operation get(x)
end on

on delivering $(\text{CAS}, x, old, new)$ by total order broadcast **do**
    $success :=$ false
    if $localState[x] = old$ **then**
        $localState[x] := new;\ success :=$ true
    end if
    return $success$ as result of operation CAS(x, old, new)
end on

Eventual consistency

Linearizability advantages:

- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

Eventual consistency

Linearizability advantages:

- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

Downsides:

- **Performance** cost: lots of messages and waiting for responses

Eventual consistency

Linearizability advantages:

- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

Downsides:

- **Performance** cost: lots of messages and waiting for responses
- **Scalability** limits: leader can be a bottleneck

Eventual consistency

Linearizability advantages:

- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

Downsides:

- **Performance** cost: lots of messages and waiting for responses
- **Scalability** limits: leader can be a bottleneck
- **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

Eventual consistency

Linearizability advantages:

- Makes a distributed system behave as if it were non-distributed
- Simple for applications to use

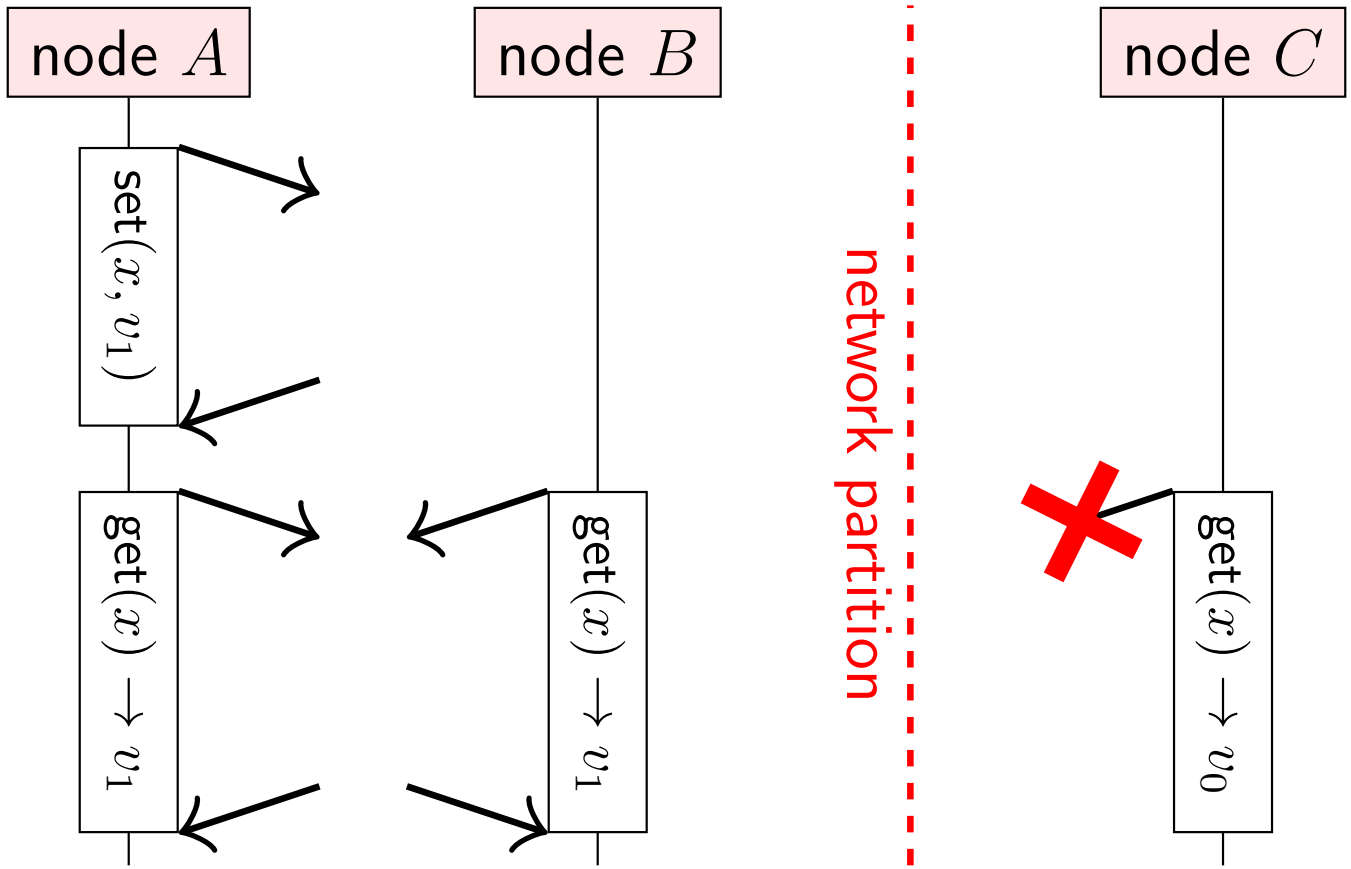
Downsides:

- **Performance** cost: lots of messages and waiting for responses
- **Scalability** limits: leader can be a bottleneck
- **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

Eventual consistency: a weaker model than linearizability. Different trade-off choices.

The CAP theorem

A system can be either strongly **Consistent** (linearizable) or **Available** in the presence of a network **Partition**



C must either wait indefinitely for the network to recover, or return a potentially stale value

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency:

- **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency:

- **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- **Convergence:** any two replicas that have processed the same set of updates are in the same state
  (even if updates were processed in a different order).

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency:

- **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- **Convergence:** any two replicas that have processed the same set of updates are in the same state (even if updates were processed in a different order).

Properties:

- Does not require waiting for network communication
- Causal broadcast (or weaker) can disseminate updates

Eventual consistency

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency:

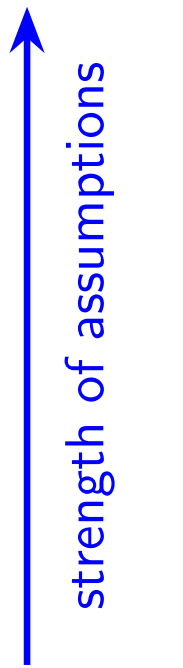
- **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- **Convergence:** any two replicas that have processed the same set of updates are in the same state (even if updates were processed in a different order).

Properties:

- Does not require waiting for network communication
- Causal broadcast (or weaker) can disseminate updates
- Concurrent updates $\Longrightarrow$ **conflicts** need to be resolved

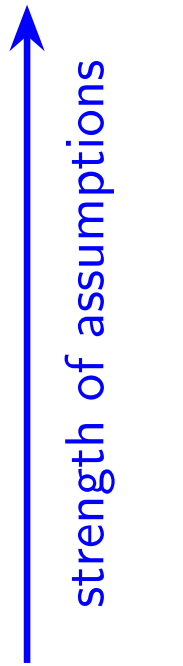
Summary of minimum system model requirements

Problem	**Must wait for communication**	**Requires synchrony**
atomic commit	all participating nodes	partially synchronous



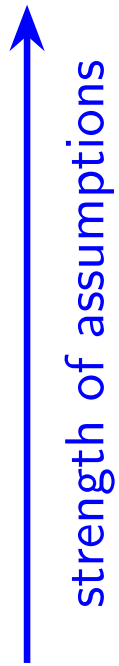
Summary of minimum system model requirements

Problem	**Must wait for communication**	**Requires synchrony**
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous



Summary of minimum system model requirements

Problem	**Must wait for communication**	**Requires synchrony**
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous
linearizable get/set	quorum	asynchronous



Summary of minimum system model requirements

Problem	**Must wait for communication**	**Requires synchrony**
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous
linearizable get/set	quorum	asynchronous
eventual consistency, causal broadcast, FIFO broadcast	local replica only	asynchronous