بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

جلسه‌ی نوزدهم

الگوریتم‌هایی در سیستم‌های توزیع‌شده

جلسه‌ی گذشته

Consistency Guarantees

- Eventual C

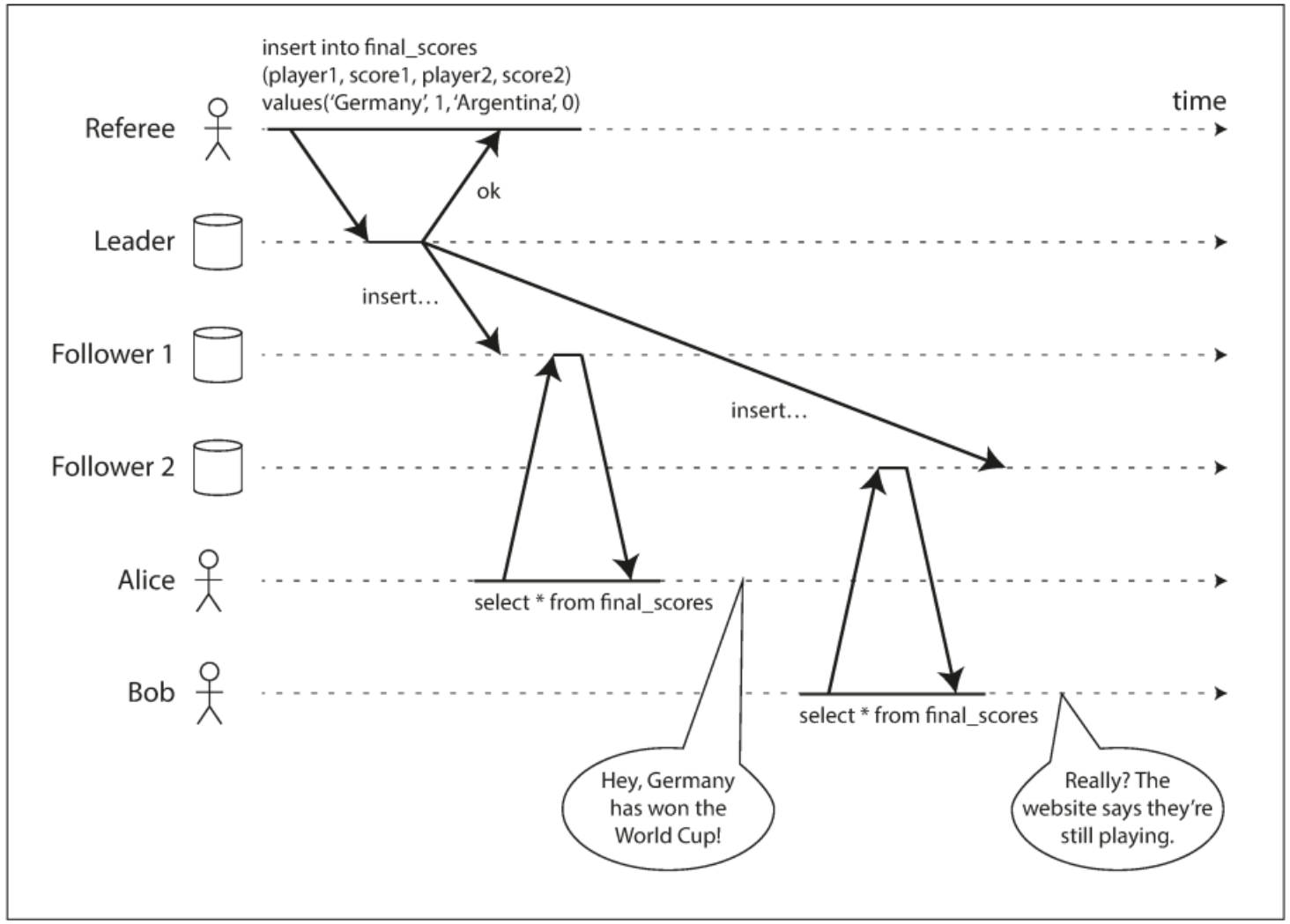

Figure 9-1. This system is not linearizable, causing football fans to be confused.

Linearizability

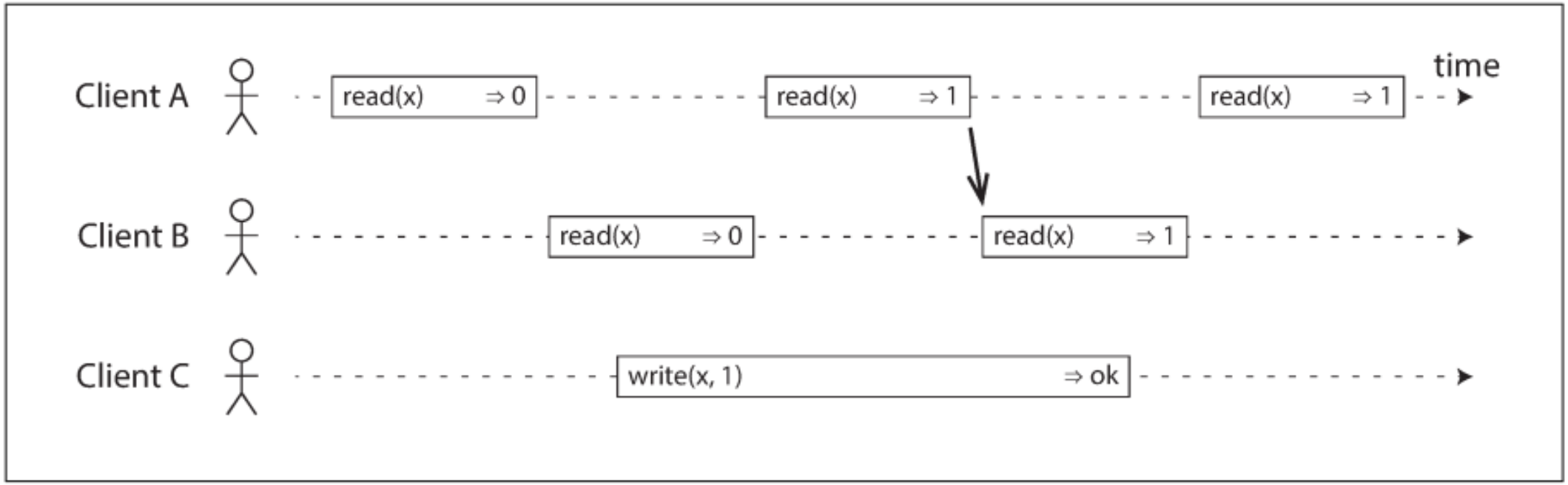

Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

Linearizability

- New operation:
  - *cas(x, v_old, v_new) ⇒ r*
  - *Compare and set*

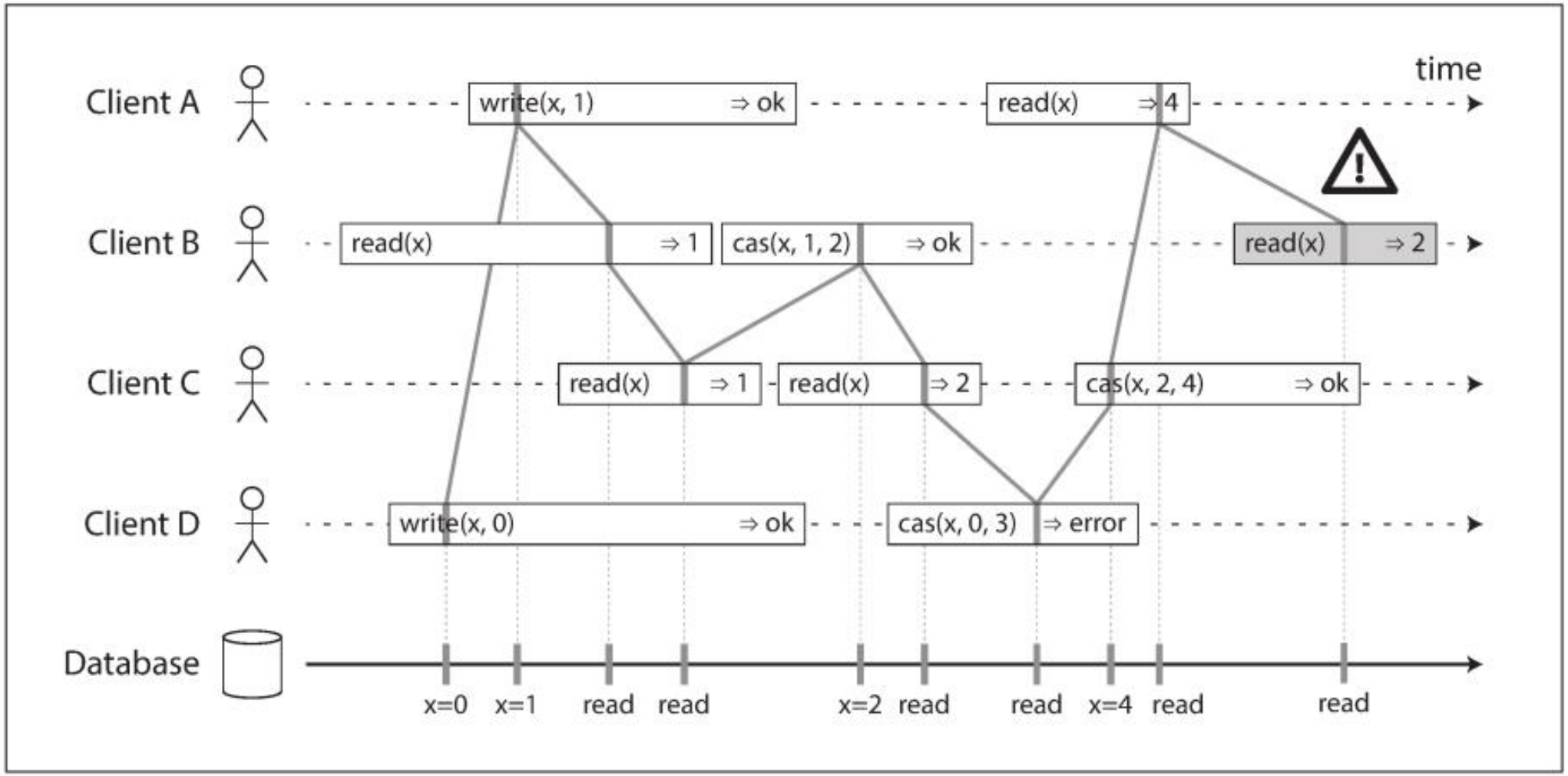

Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

Linearizability ->

- Linearizability -> Leader Election
- Linearizability -> Consensus
- Linearizability -> uniqueness Constraint

Implementing Linearizable Systems

- Without replication
- Single Leader replication
- Implement with consensus algorithms
- Multi leader replication
- Leader-less replication

CAP Theorem

- Consistency
- Availability
- Partition Tolerance

- you can only guarantee two out of these properties

Costs of Linearizability

- Single computers may be not linearizability
- Many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance [46].
- Linearizability is slow—and this is true all the time, not only during a network fault.

ORDERING GUARANTEES

Causality

- Consistent Prefix Reads
- Detecting Concurrent Writes
- Snapshot Isolation
- …

Causal Consistency

- If some process performs a write operation A, and some (the same or another) process that observed A then performs a write operation B
  - *then it is possible that A is the cause of B*
  - *we say that A "potentially causes" or "causally precedes" B.*
- Causal Consistency guarantees that if A causally-precedes B, then every process in the system observes A before observing B.

Causal Consistency = session guarantees

- Read Your Writes: If a process performs a write, the same process later observes the result of its write.
- Monotonic Reads: the set of writes observed (read) by a process is guaranteed to be monotonically non-decreasing.
- Writes Follow Reads: if some process performs a read followed by a write, and another process observes the result of the write, then it can also observe the read (unless it has been overwritten).
- Monotonic Writes: If some process performs a write, followed some time later by another write, other processes will observe them in the same order.

Causal Consistency

- linearizability implies causality

Try to implement causal consistency

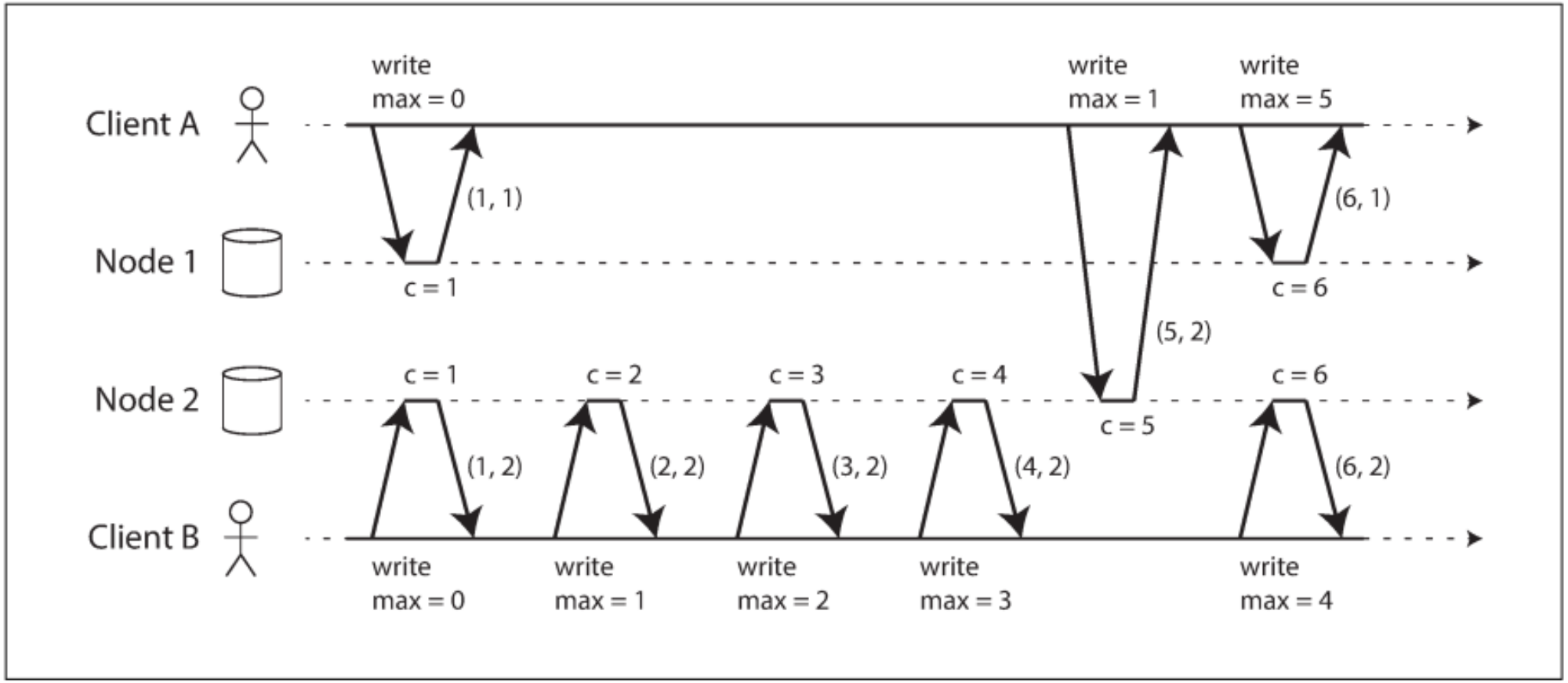

Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

Is timestamp ordering is enough?

- Username uniqueness constraint
  - *it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized. If you have an operation to create a username, and you are sure that no other node can insert a claim for the same username ahead of your operation in the total order, then you can safely declare the operation successful.*
- Try to solve with total order broadcast

جلسه‌ی جدید

TOTAL ORDER BROADCAST

Total order broadcast (Atomic Broadcast)

- Reliable delivery
  - *No messages are lost: if a message is delivered to one node, it is delivered to all nodes.*
- Totally ordered delivery
  - *Messages are delivered to every node in the same order.*

- Validity:
  - *if a correct participant broadcasts a message, then all correct participants will eventually receive it.*
- Uniform Agreement:
  - *if one correct participant receives a message, then all correct participants will eventually receive that message.*
- Uniform Integrity:
  - *a message is received by each participant at most once, and only if it was previously broadcast.*
- Uniform Total Order:
  - *the messages are totally ordered in the mathematical sense; that is, if any correct participant receives message 1 first and message 2 second, then every other correct participant must receive message 1 before message 2.*

Uniqueness constraint with total order broadcast

Total order broadcast like WAL

Total order broadcast Vs. Linearizability

- Total order broadcast is asynchronous:
  - *messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about when a message will be delivered (so one recipient may lag behind the others).*
- By contrast, linearizability is a recency guarantee:
  - *a read is guaranteed to see the latest value written*

معادل بودن مسئله‌ها

Total order broadcast => Linearizability

- implement such a linearizable compare-and-set operation using total order broadcast

Linearizability => Total order broadcast

- Assume we have an atomic increment-and-get operation with linearizable register

Linearizability => Consensus

DISTRIBUTED TRANSACTION

Atomicity on a single node

Atomicity with multiple nodes

- o simply send a commit request to all of the nodes and independently commit the transaction on each one?
  - *it could easily happen that the commit succeeds on some nodes and fails on other nodes*

Two phase commit

- A coordinator node

Two phase commit

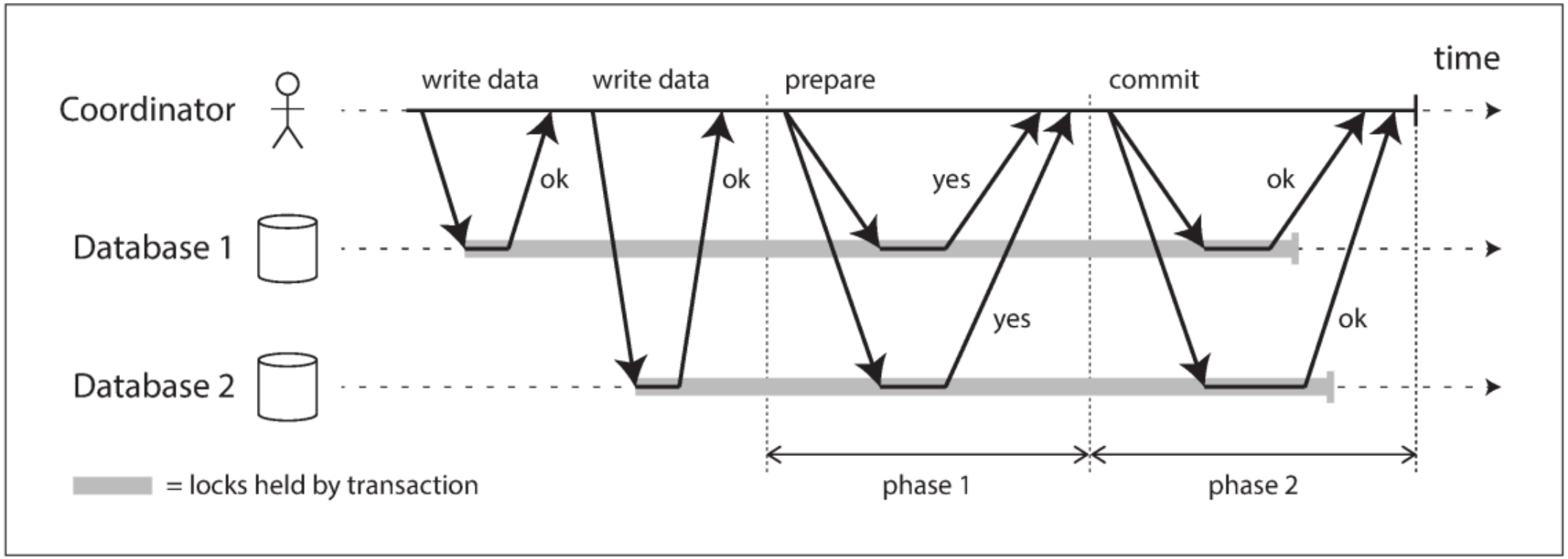

Figure 9-9. A successful execution of two-phase commit (2PC).

Two phase commit - promise

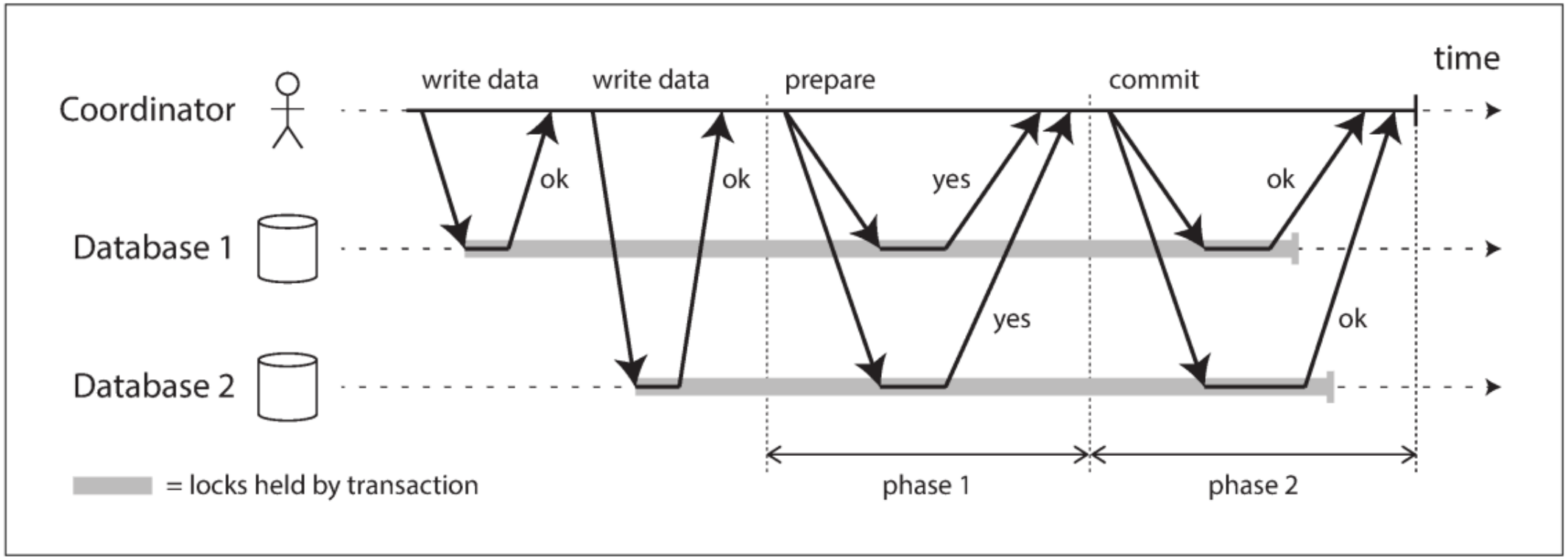

Figure 9-9. A successful execution of two-phase commit (2PC).

Two phase commit

1. When the application wants to begin a distributed transaction, it requests a transaction ID from the coordinator. This transaction ID is globally unique.
2. The application begins a single-node transaction on each of the participants, and attaches the globally unique transaction ID to the single-node transaction. All reads and writes are done in one of these single-node transactions. If anything goes wrong at this stage (for example, a node crashes or a request times out), the coordinator or any of the participants can abort.
3. When the application is ready to commit, the coordinator sends a prepare request to all participants, tagged with the global transaction ID. If any of these requests fails or times out, the coordinator sends an abort request for that transaction ID to all participants.

Two phase commit

4. When a participant receives the prepare request, it makes sure that it can definitely commit the transaction under all circumstances. This includes writing all transaction data to disk (a crash, a power failure, or running out of disk space is not an acceptable excuse for refusing to commit later), and checking for any conflicts or constraint violations. By replying “yes” to the coordinator, the node promises to commit the transaction without error if requested. In other words, the participant surrenders the right to abort the transaction, but without actually committing it.
5. When the coordinator has received responses to all prepare requests, it makes a definitive decision on whether to commit or abort the transaction (committing only if all participants voted “yes”). The coordinator must write that decision to its transaction log on disk so that it knows which way it decided in case it subse quently crashes. This is called the commit point.
6. Once the coordinator’s decision has been written to disk, the commit or abort request is sent to all participants. If this request fails or times out, the coordinator must retry forever until it succeeds. There is no more going back: if the decision was to commit, that decision must be enforced, no matter how many retries it akes. If a participant has crashed in the meantime, the transaction will be committed when it recovers—since the participant voted “yes,” it cannot refuse to commit when it recovers.

Two phase commit – coordinator failure

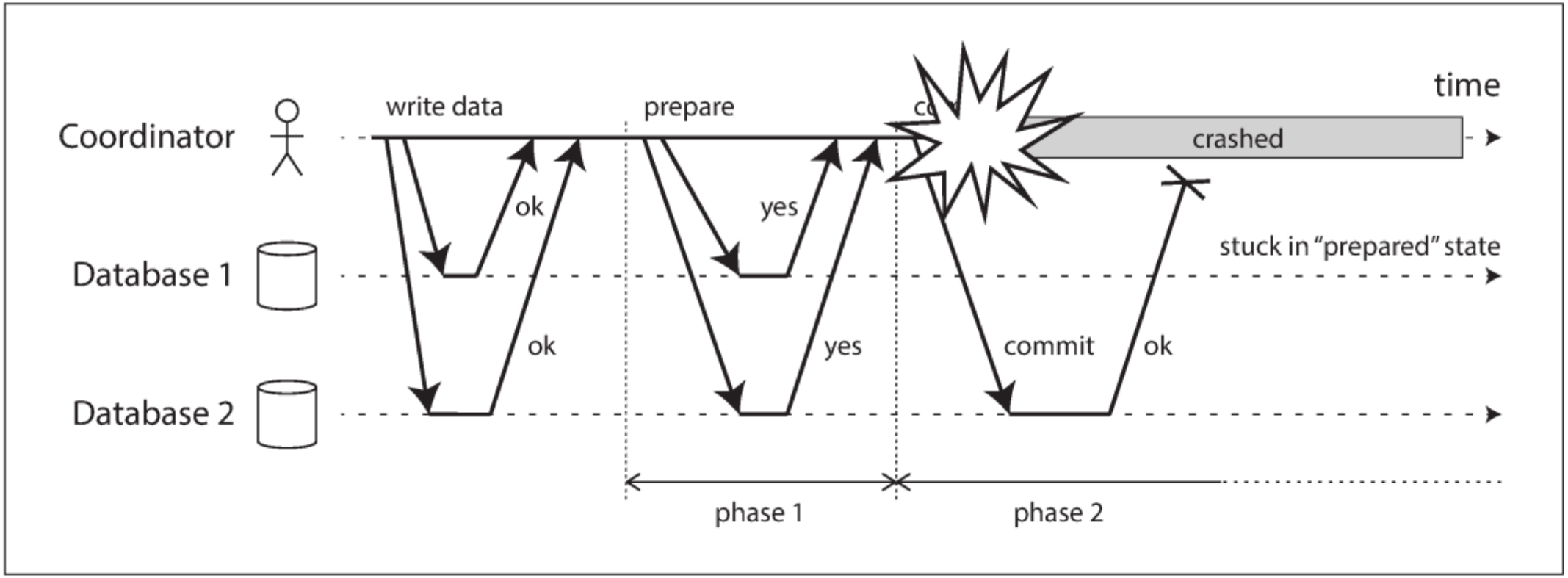

Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

Why two phase commit coordinator failure is an issue

- Holding locks while in doubt

Three phase commit

- Two phase commit -> blocking
- If we know that if a packet not receiving from a timeout is really failed
  - *We can implement a non-blocking distributed transaction*
  - *Not practically used due to unbounded timeouts.*

Distributed Transactions in Practice

- Database-internal - all on same software
- Heterogeneous distributed transactions - compatible with other systems, and harder to support

Distributed Transactions in Practice

- Exactly-once message processing

CONSENSUS

Definition

- Consensus is generalized as a problem where node(s) may propose values, and consensus decides on one of those values
- Uniform agreement - same decision
- Integrity - only decide once
- Validity - choice must be one of the proposals
- Termination - every node that doesn’t crash eventually decides
  - *2PC does not meet this requirement*

- Best known consensus algorithms are:
  - *Viewstamped Replication (VSR)*
  - *Paxos (and Multi-Paxos)*
  - *Raft*
  - *Zab*

Consensus = Total Order Broadcast

- Most of these implement total order broadcast, which is equivalent to repeated rounds of consensus

With Single Leader

- In single leader replication, the leader provides a total order broadcast, which is consensus. But if a leader fails, you need consensus to elect a new leader. So what do you do?
- Epoch numbering
  - *These consensus algorithms track a monotonically increasing epoch number, and only ensure that within each epoch number, the leader is unique*
  - *If two would-be leaders disagree, the one with the higher epoch number wins*
  - *Before a leader can make a decision, it needs to collect votes from a quorum of nodes*

The cost of Consensus

- The voting process for proposals is a form of synchronous replication - and potentially slow
- Since consensus requires a majority, a network partition will make minority nodes unable to proceed
- Most consensus algorithms don’t allow adding or removing nodes
- If you have highly variable network delays, timeouts can cause frequent leader elections, which consumes many resources
- Edge cases exist, such as where Raft got stuck bouncing the leader back and forth between two nodes forever

Membership and Coordination Services

- ZooKeeper or etcd are often described as “distributed key-value stores” but you wouldn’t use them as application databases. Typically they hold system configuration settings.
- They’re designed to hold small amounts of data, all in memory. This small set is kept in sync using total ordered broadcast.
- Typically you wouldn’t run ZooKeeper on thousands of nodes. Rather you’d have 3 or 5 that handle key coordination, and all other nodes would look to those handful of nodes.

Membership and Coordination Services

- Allocating work to nodes - ZooKeeper is good for choosing a single node, like a leader, and it is also good for allocating resources
  - *Job schedulers*
  - *Assigning and rebalancing partitions to nodes*
  - *Can be done using atomic operations, ephemeral nodes, and notifications. But it's still complicated enough that higher level libraries like Apache Curator can be helpful*

Membership and Coordination Services

- Service discovery
  - *ZooKeeper, etcd, and Consul are often used for service discovery*
  - *Service recovery may not really require consensus (DNS works pretty well), but if you're electing a leader anyway, perhaps help with discovery. An additional service is read-only caching replicas that can serve results of all decisions, but don't participate in voting.*

Membership and Coordination Services

- Membership services
  - *A membership service tracks which nodes are alive and active*