TDT4225
Chapter 9 – Consistency and Consensus

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Consistency Guarantees

- Replicated systems may have inconsistent values.
- Eventual consistency: convergence, as we expect all replicas to eventually converge to the same value.
- The edge cases of eventual consistency only become apparent when there is a fault in the system (e.g., a network interruption) or at high concurrency.
- Systems with stronger guarantees may have worse performance or be less fault-tolerant (available) than systems with weaker guarantees.
- Distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

Linearizability

- Aka atomic consistency, strong consistency, immediate consistency, or external consistency.
- The basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic.

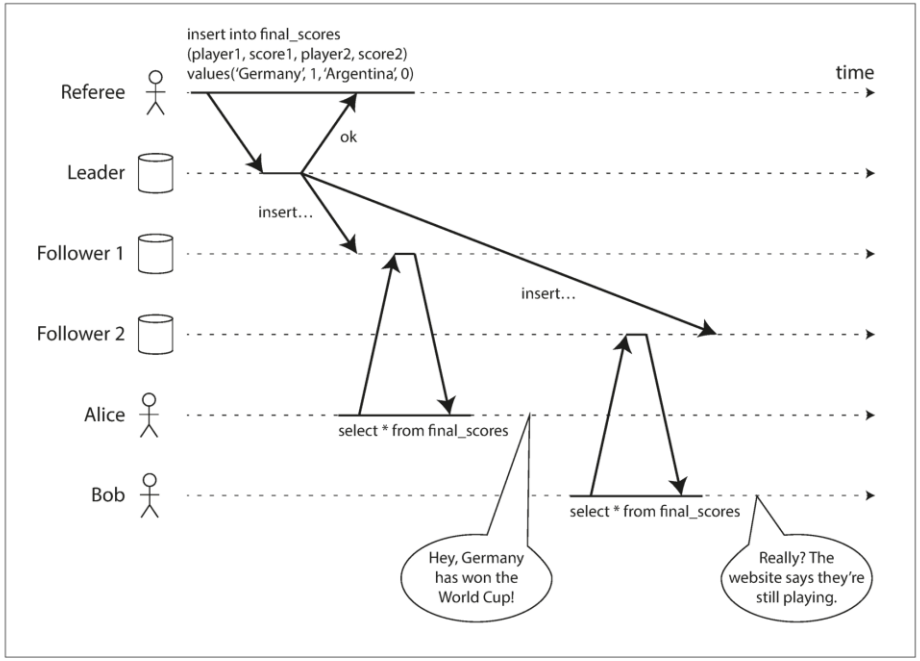

Figure 9-1. This system is not linearizable, causing football fans to be confused.

What Makes a System Linearizable?

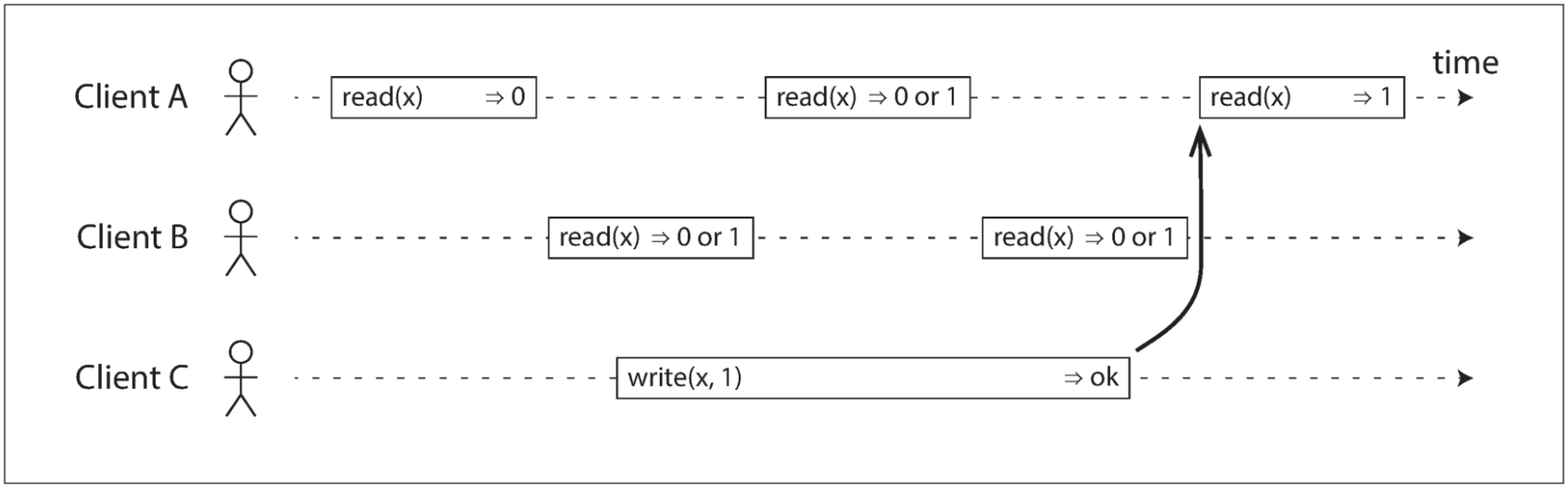

Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

Linearizable, Another Constraint

- An atomic point in time for the flip?

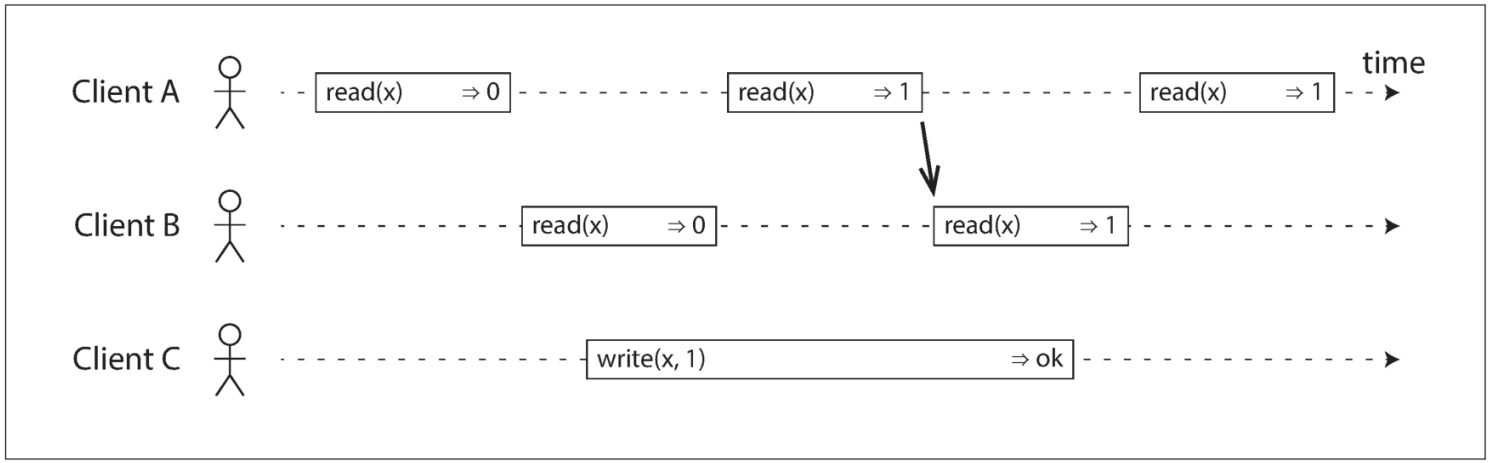

Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

Linearizable, CAS – Compare-and-Set

- cas(x, v_old , v_new) ⇒ r means the client requested an atomic compare-and-set on variable x.
- An atomic compare-and-set (cas) operation can be used to check the value hasn't been concurrently changed by another client.
- The requirement of linearizability is that the lines joining up the operation markers always move forward in time (from left to right), never backward.

Linearizable, CAS – Compare-and-Set (2)

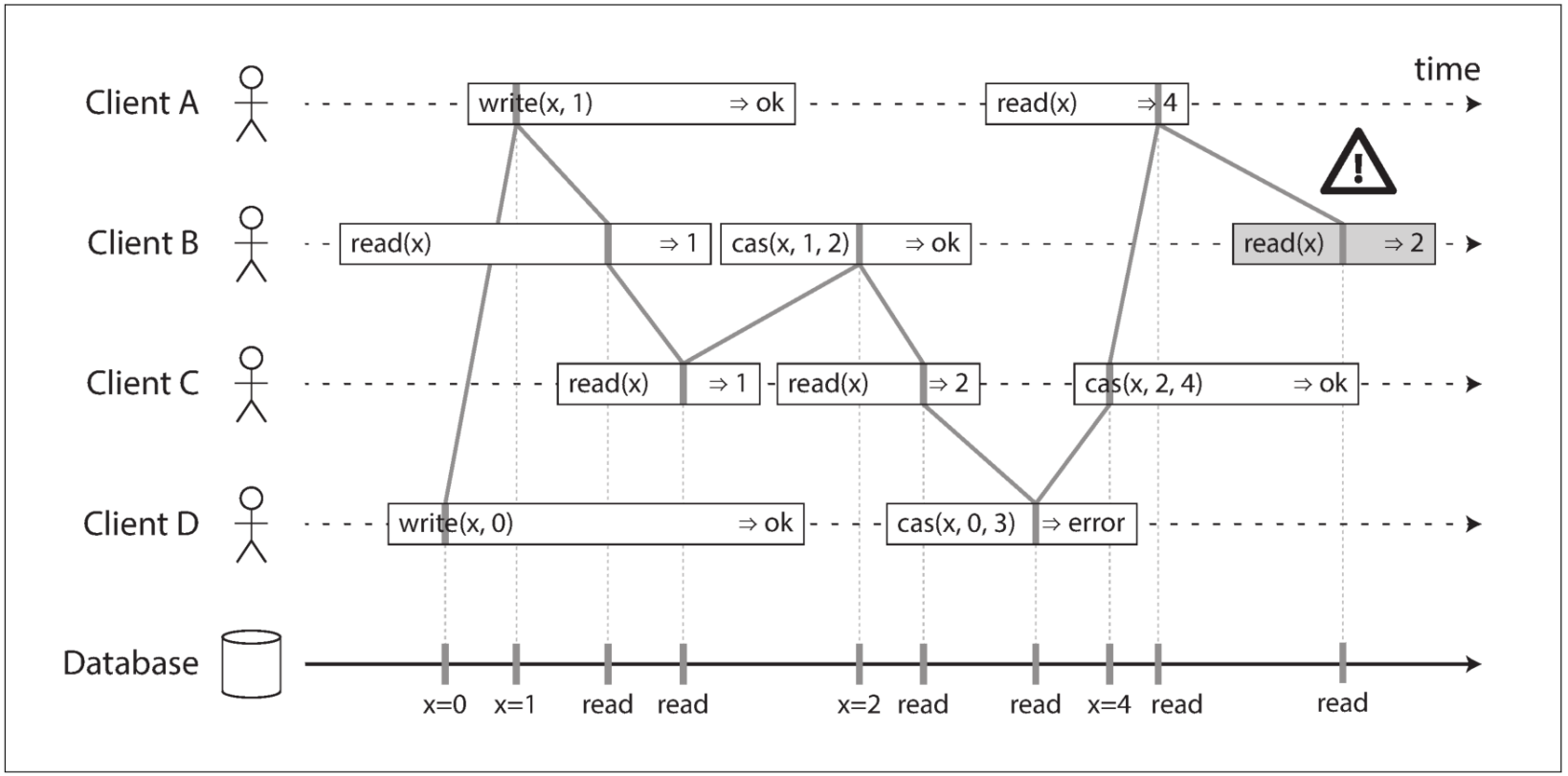

Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

Linearizable vs. serializable

- **Serializability**: Guarantees that transactions behave the same as if they had executed in some serial order.
- **Linearizability**: Recency guarantee on reads and writes.
- Implementations of serializability based on two-phase locking or actual serial execution are linearizable.
- Serializable snapshot isolation is by design not linearlizable. It hides concurrent writes.

Relying on Linearizability

- Locking and leader election: All nodes must agree which node owns the lock; otherwise it is useless.
- Constraints and uniqueness guarantees: A hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability.
- Cross-channel timing dependencies:

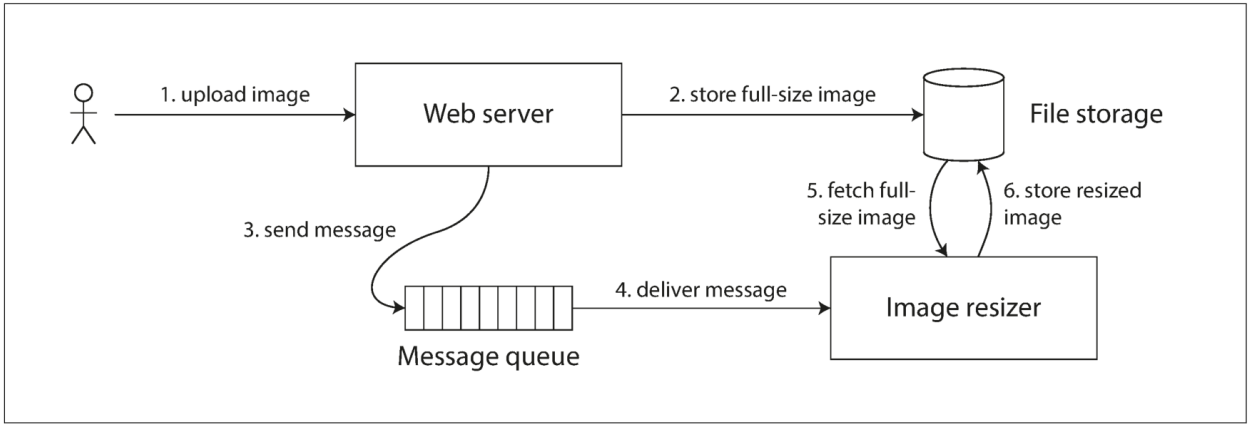

Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

Implementing Linearizable Systems

- Replication methods:
  - Single-leader replication (potentially linearizable)
  - Consensus algorithms (linearizable): ZooKeeper and etcd
  - Multi-leader replication (not linearizable)
  - Leaderless replication (probably not linearizable)

Linearizability and quorums

- It seems as though strict quorum reads and writes should be linearizable, it may not when we have variable network delays:

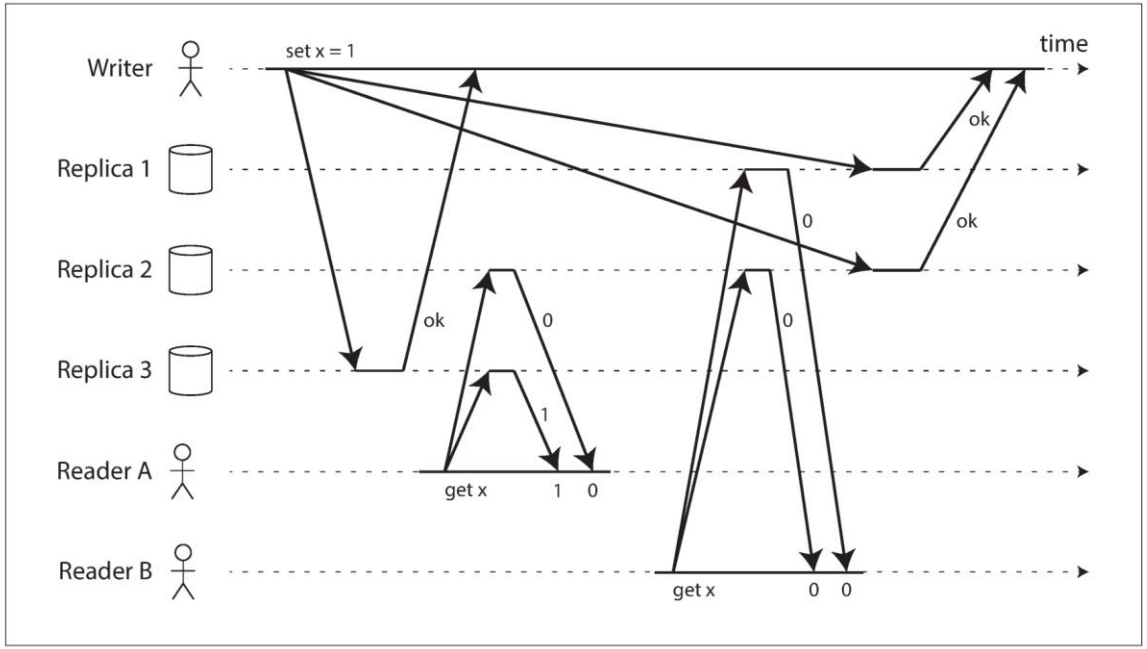

Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

The Cost of Linearizability

- If you need linearizability: Must be made unavailable

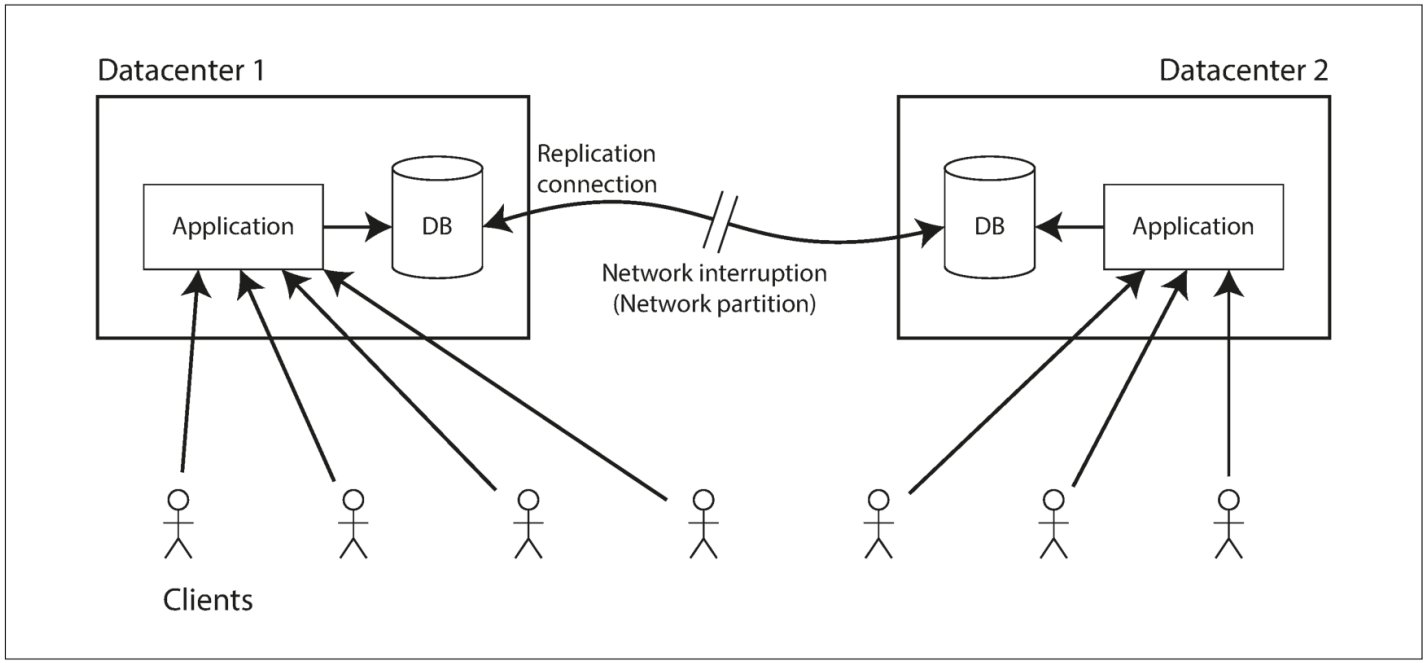

Figure 9-7. A network interruption forcing a choice between linearizability and availability.

The CAP theorem

- Eric Brewer, 2000: Consistency, Availability and Partition Tolerance. Choose any two.
- Applications that don’t require linearizability can be more tolerant of network problems.
- Either Consistent or Available when Partitioned.
- Partitioned isn’t something you choose
- CAP awakened the interest in other solutions than consistency
- CAP isn’t used so much anymore

Linearizability and network delays

- Every CPU core has its own memory cache and store buffer. Memory access first goes to the cache by default, and any changes are asynchronously written out to main memory.
- The reason for dropping linearizability is performance, not fault tolerance.
- The same is true of many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance.
- If you want linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network.

Ordering Guarantees

- There are deep connections between ordering, linearizability, and consensus.
- Ordering helps preserve causality, e.g. causal dependency between the question and the answer.
- E.g. a row must first be created before it can be updated.
- If a system obeys the ordering imposed by causality, we say that it is **causally consistent**.
- Linearizability: total order of operations
- Causality: defines a partial order, some operations are incomparable

Linearizability is stronger than causal consistency

- Linearizability implies causality.
- Performance considerations: Some distributed data systems have abandoned linearizability.
- Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures.
- We can use sequence numbers or timestamps to order events.
- However, physical clocks cannot be used

Lamport timestamps (1978)

- Every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request: Every causal dependency results in an increased timestamp.

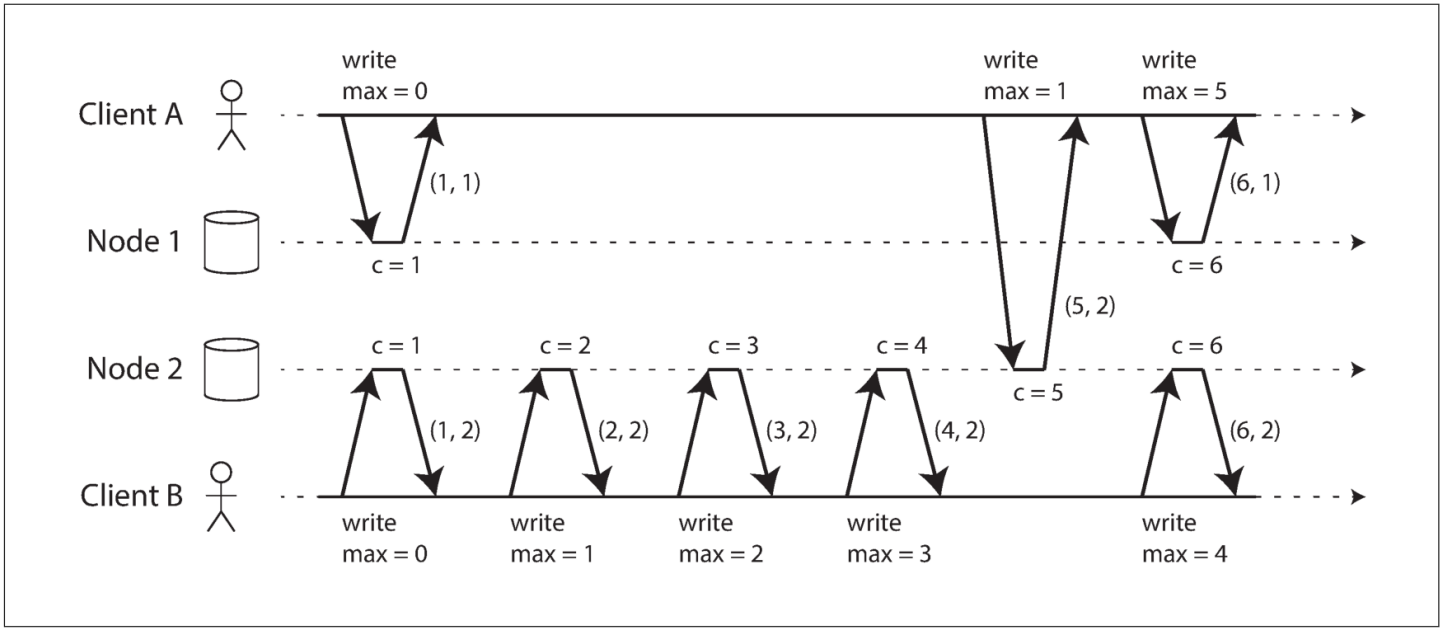

Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

Timestamp ordering is not sufficient

- To implement a uniqueness constraint for user-names, it's not sufficient to have a total ordering of operations – you need to know when that order is finalized among all nodes.
- Partitioned databases with a single leader per partition often maintain ordering only per partition, which means they cannot offer consistency guarantees across partitions
- *Total order broadcast*: a protocol for exchanging messages between nodes.
  - Reliable delivery
  - Totally ordered delivery: Every node receives the messages in the same order
  - May be used for state-machine replication
  - May be seen as a replication log, where all nodes can read the same sequence of messages

Distributed Transactions and Consensus

- Informally, the goal is simply to get several nodes to agree on something.
- Leader election
- Atomic commit
- On a single node, transaction commitment crucially depends on the order in which the log is durably written to disk: operations + commit log record
- A transaction commit must be irrevocable—you are not allowed to change your mind and retroactively abort a transaction after it has been committed.

Two-phase commit (2PC)

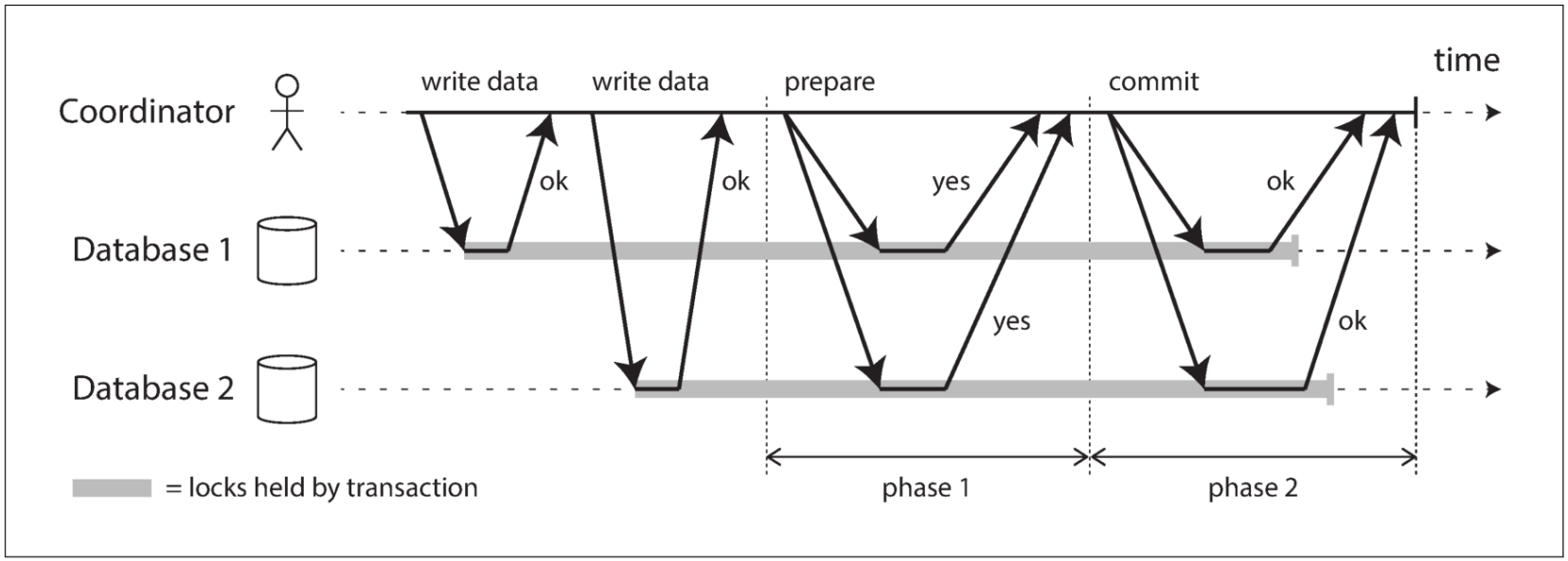

Figure 9-9. A successful execution of two-phase commit (2PC).

2PC (2)

- Coordinator
- Participants
- If all participants reply “yes,” indicating they are ready to commit, then the coordinator sends out a commit request in phase 2.
- If any of the participants replies “no,” the coordinator sends an abort request to all nodes in phase 2.
- Commitment:
  - It’s like bacon and eggs
  - The chicken participates
  - The pig is committed

2PC (3)

- Request a trancation ID from the coordinator
- Start execution on all nodes. Any node may decide to abort.
- When ready-to-commit, the coordinator sends a prepare-to-commit to all participants
- Participants receive PTC, and replies yes or no.
- Coordinator receives all, and decides. Logs the decision.
- The commit or abort message is sent to all participants.
- Done is returned from all participants

Coordinator failure

- If any of the prepare requests fail or time out, the coordinator aborts the transaction.
- If any of the commit or abort requests fail, the coordinator retries them indefinitely.
- **Uncertain / in doubt**: If the coordinator crashes, the participant can do nothing but wait until the coordinator recovers.

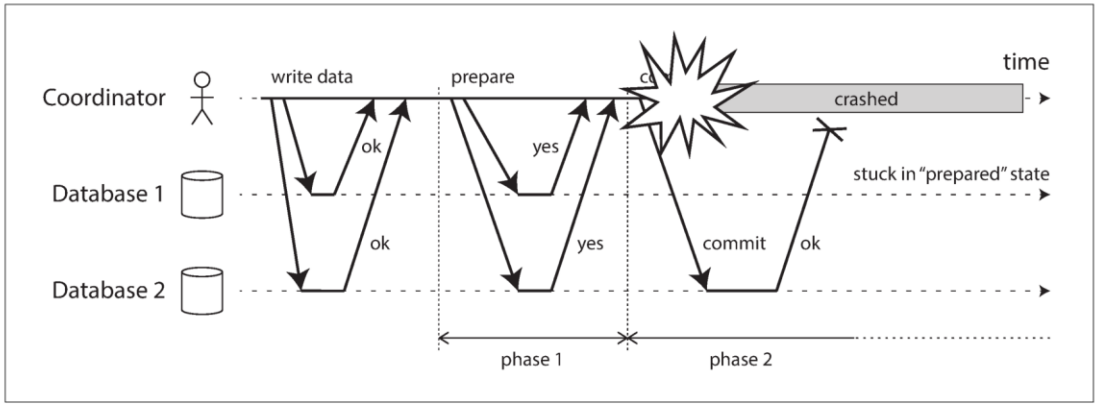

Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

Peformance of 2PC

- Too costly?: Due to the additional disk forcing (fsync) that is required for crash recovery, and the additional network round-trips.
- Two types of distributed transactions
  - Database-internal distributed transactions
  - Heterogeneous distributed transactions (TP monitors)
- XA Protocol: a C API for interfacing with a transaction coordinator. Standard 2PC support.
- The coordinator is usually a library that is loaded into the same process as the application issuing the transaction.

Holding locks while in doubt

- Problem when prepared-to-commit, but coordinator dies

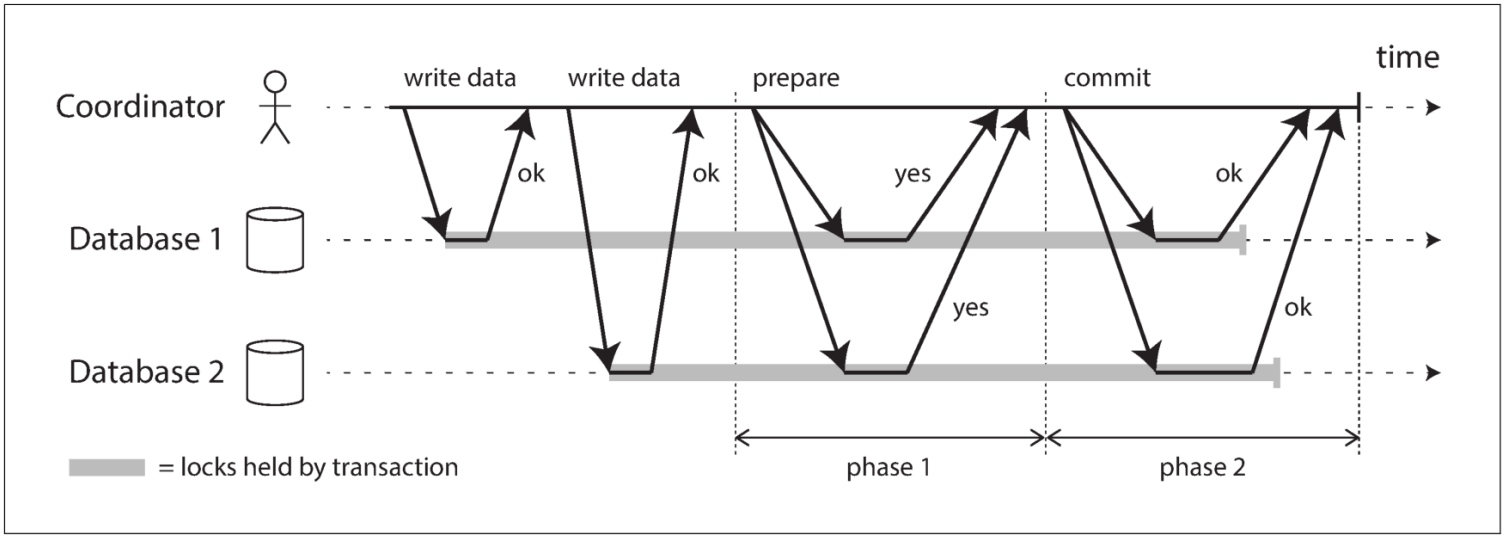

Figure 9-9. A successful execution of two-phase commit (2PC).

- Orphaned transactions must often be manually GCed (removing problematic locks after a long period)

Limitations of distributed transactions (XA)

- The coordinator must also be replicated (hot standby controller)
- XA cannot detect deadlocks in-between heterogenous participants.
- Problem when applications become parts of the transaction execution.
- If any part of the system is broken, the transaction also fails. Distributed transactions thus have a tendency of amplifying failures.

Fault-Tolerant Consensus

- One or more nodes may propose values, and the consensus algorithm decides on one of those values.
- Everyone decides on the same outcome, and once you have decided, you cannot change your mind.
- *Uniform agreement*: No two nodes decide differently.
- *Integrity*: No node decides twice.
- *Validity*: If a node decides value v, then v was proposed by some node.
- *Termination*: Every node that does not crash eventually decides some value.

Consensus algorithms and total order broadcast

- Viewstamped Replication (VSR)
- Paxos (and multi-Paxos)
- RAFT
- Zab
- They decide on a sequence of values: Total order broadcast algorithms.
- Messages to be delivered exactly once, in the same order, to all nodes.

Epoch numbering and quorums

- Epoch number: The algorithms guarantee that within each epoch, the leader is unique.
- Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader.
- The new leader must collect votes from a quorum of nodes to see that it has the highest epoch number.
- Some systems also allow nodes to have weights in quorums

Limitations of consensus

- Consensus algorithms were a breakthrough in distributed systems, however they are costly.
- Consensus systems always require a strict majority to operate. Network failure may be a problem.
- Consensus systems generally rely on timeouts to detect failed nodes. Variable network time may be a problem.
- RAFT: If one network link is problematic, the algorithm may be fluctuating between multiple new leaders. (sensitive to network problems)

Membership and Coordination Services

- ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory
- ZooKeeper is modeled after Google's Chubby lock service
  - Linearizable atomic operations
  - Total ordering of operations
  - Failure detection
  - Change notification
- ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients.
- Service discovery does not require consensus, leader election does.
- A membership service determines which nodes are currently active and live members of a cluster.

Summary

- Consistency and consensus.
- Similar solutions / problems
  - Linearizable compare-and-set registers
  - Atomic transaction commit
  - Total order broadcast
  - Locks and leases
  - Membership / coordination service
  - Uniqueness constraint