Linearizability vs Serializability

Serializability
is an isolation property of transactions
It guarantees that transactions behave the same as if they had executed in some serial order (each transaction running to completion before the next transaction starts). It is okay for that serial order to be different from the order in which transactions were actually run.

Serializability focuses on the *final result* of transactions, without constraints on when exactly they appear to take effect relative to real time.

Linearizability

Linearizability is a recency guarantee on reads and writes of a register (an individual object).

Serializable snapshot isolation is not linearizable: by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

The key distinction is that linearizability preserves the real-time ordering between operations. Once an operation completes, all subsequent operations must observe its effects.

Cross-channel timing dependencies

Similar situations can arise in computer systems. For example, say you have a website where users can upload a photo, and a background process resizes the photos to lower resolution for faster download (thumbnails).

The image resizer needs to be explicitly instructed to perform a resizing job, and this instruction is sent from the web server to the resizer via a message queue (see Chap ter 11). The web server doesn’t place the entire photo on the queue, since most mes sage brokers are designed for small messages, and a photo may be several megabytes in size. Instead, the photo is first written to a file storage service, and once the write is complete, the instruction to the resizer is placed on the queue.

If the file storage service is linearizable, then this system should work fine. If it is not linearizable, there is the risk of a race condition: the message queue might be faster than the internal replication inside the storage service. In this case, when the resizer fetches the image (step 5), it might see an old version of the image, or nothing at all. If it processes an old version of the image, the full-size and resized images in the file storage become permanently

inconsistent. This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue. Without the recency guarantee of linearizability, race conditions between these two channels are possible.

Implementing Linearizable Systems

Single-leader replication (potentially linearizable) ->The leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes. If you make reads from the leader, or from synchronously updated followers, they have the potential to be linearizable.

Consensus algorithms (linearizable) ->Some consensus algorithms, which we will discuss later in this chapter, bear a resemblance to single-leader replication. However, consensus protocols contain measures to prevent split brain and stale replicas.

Multi-leader replication (not linearizable) -> Systems with multi-leader replication are generally not linearizable, because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes. For this reason, they can produce conflicting writes that require resolution.

Leaderless replication (probably not linearizable) -> For systems with leaderless replication , people sometimes claim that you can obtain “strong consistency” by requiring quorum reads and writes (w + r > n).

making a system linearizable can harm its performance and availability, especially if the system has significant network delays (for example, if it’s geographically distributed). For this reason, some distributed data systems have abandoned linearizability, which allows them to achieve better performance but can make them difficult to work with..

The CAP theorem

CAP stands for

either Consistent or Available when Partitioned.

The trade-off is as follows:

- If your application requires linearizability, and some replicas are disconnected from the other replicas due to a network problem, then some replicas cannot process requests while they are disconnected: they must either wait until the net work problem is fixed, or return an error (either way, they become unavailable).
- If your application does not require linearizability, then it can be written in a way that each replica can process requests independently, even if it is disconnected from other replicas

(e.g., multi-leader). In this case, the application can remain available in the face of a network problem, but its behavior is not linearizable.

Why make this trade-off? It makes no sense to use the CAP theorem to justify the multi-core memory consistency model: within one computer we usually assume reli able communication, and we don't expect one CPU core to be able to continue operating normally if it is disconnected from the rest of the computer. The reason for dropping linearizability is performance, not fault tolerance.

The same is true of many distributed databases that choose not to provide lineariza ble guarantees: they do so primarily to increase performance, not so much for fault tolerance. Linearizability is slow—and this is true all the time, not only during a network fault.

Weaker consistency models can be much faster, so this trade-off is important for latency-sensitive systems.

Ordering and Causality

If you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. This happened before relationship is another expression of causality: if A happened before B, that means B might have known about A, or built upon A, or depended on A. If A and B are concur rent, there is no causal link between them; in other words, we are sure that nei ther knew about the other.

Causality imposes an ordering on events: cause comes before effect; a message is sent before that message is received; the question comes before the answer. And, like in real life, one thing leads to another: one node reads some data and then writes some thing as a result, another node reads the thing that was written and writes something else in turn, and so on. These chains of causally dependent operations define the causal order in the system—i.e., what happened before what.

If a system obeys the ordering imposed by causality, we say that it is causally consistent.

The causal order is not a total order->
A total order allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller.

Linearizability In a linearizable system, we have a total order of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first.

Causality->
Two events are ordered if they are causally related (one happened before the other), but they

are incomparable if they are concurrent.
This means that causality defines a partial order, not a total order: some operations are ordered with respect to each other, but some are incomparable.

If you are familiar with distributed version control systems such as Git, their version histories are very much like the graph of causal dependencies. Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.

Linearizability is stronger than causal consistency

The answer is that linearizability implies causality: any system that is linearizable will preserve causality correctly.

In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently.

Capturing causal dependencies

In order to maintain causality, you need to know which operation happened before which other operation. This is a partial order: concurrent operations may be pro cessed in any order, but if one operation happened before another, then they must be processed in that order on every replica. Thus, when a replica processes an operation, it must ensure that all causally preceding operations (all operations that happened before) have already been processed; if some preceding operation is missing, the later operation must wait until the preceding operation has been processed.

In order to determine causal dependencies, we need some way of describing the “knowledge” of a node in the system. If a node had already seen the value X when it issued the write Y, then X and Y may be causally related.

Sequence Number Ordering

We can use sequence numbers or timestamps to order events. A timestamp need not come from a time-of-day clock. It can instead come from a logical clock, which is an algorithm to generate a sequence of numbers to identify operations, typically using counters that are incremented for every operation.

Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a total order: that is, every operation has a unique sequence number, and you can always compare two sequence numbers to determine which is greater (i.e., which operation happened later).

We can create sequence numbers in a total order that is consistent with causality we promise that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B).

Noncausal sequence number generators

The causality problems occur because these sequence number generators do not correctly capture the ordering of operations across different nodes:
• Each node may process a different number of operations per second. Thus, if one node generates even numbers and the other generates odd numbers, the counter for even numbers may lag behind the counter for odd numbers, or vice versa. If you have an odd-numbered operation and an even-numbered operation, you cannot accurately tell which one causally happened first.
• Timestamps from physical clocks are subject to clock skew, which can make them inconsistent with causality.
• In the case of the block allocator, one operation may be given a sequence number in the range from 1,001 to 2,000, and a causally later operation may be given a number in the range from 1 to 1,000. Here, again, the sequence number is inconsistent with causality.

Lamport timestamps

There is actually a simple method for generating sequence numbers that is consistent with causality. It is called a Lamport timestamp.

Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The Lamport timestamp is then simply a pair of (counter, node ID).

Two nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique.

A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering: if you have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp.

The key idea about Lamport timestamps, which makes them consistent with causality, is the following: every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.

As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because

every causal dependency results in an increased timestamp.

Lamport timestamps are sometimes confused with version vectors. Although there are some similarities, they have a different purpose: version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas Lamport timestamps always enforce a total ordering.

From the total ordering of Lamport timestamps, you cannot tell whether two operations are concurrent or whether they are causally dependent. The advantage of Lamport timestamps over version vectors is that they are more compact.

Although Lamport timestamps define a total order of operations that is consistent with causality, they are not quite sufficient to solve many common problems in dis tributed systems.

At first glance, it seems as though a total ordering of operations (e.g., using Lamport timestamps) should be sufficient to solve this problem: if two accounts with the same username are created, pick the one with the lower timestamp as the winner (the one who grabbed the username first), and let the one with the greater timestamp fail. Since timestamps are totally ordered, this comparison is always valid.

This is not sufficient when a node has just received a request from a user to create a username, and needs to decide right now whether the request should succeed or fail. At that moment, the node does not know whether another node is concurrently in the process of creating an account with the same username, and what timestamp that other node may assign to the operation.

The problem here is that the total order of operations only emerges after you have collected all of the operations. If another node has generated some operations, but you don’t yet know what they are, you cannot construct the final ordering of operations: the unknown operations from the other node may need to be inserted at various positions in the total order.

This idea of knowing when your total order is finalized is captured in the topic of total order broadcast.

Total Order Broadcast

However, in a distributed system, getting all nodes to agree on the same total ordering of operations is tricky.

Total order broadcast is usually described as a protocol for exchanging messages between nodes. Informally, it requires that two safety properties always be satisfied:
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.

Consensus services such as ZooKeeper and etcd actually implement total order broadcast. This fact is a hint that there is a strong connection between total order broadcast and consensus, which we will explore later in this chapter.

Total order broadcast is exactly what you need for database replication: if every message represents a write to the database, and every replica processes the same writes in the same order, then the replicas will remain consistent with each other (aside from any temporary replication lag). This principle is known as state machine replication.

Distributed Transactions and Consensus

Atomic Commit and Two-Phase Commit (2PC)

The outcome of a transaction is either a successful commit, in which case all of the transaction's writes are made durable, or an abort, in which case all of the transaction's writes are rolled back.

A transaction commit must be irrevocable—you are not allowed to change your mind and retroactively abort a transaction after it has been committed. The reason for this rule is that once data has been committed, it becomes visible to other transac tions, and thus other clients may start relying on that data; this principle forms the basis of read committed isolation.

2PC

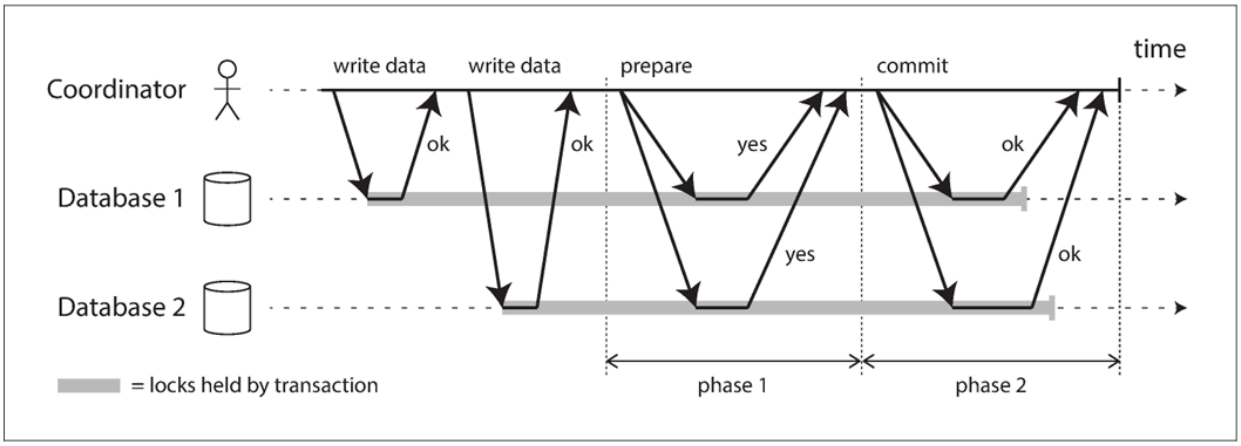

Figure 9-9. A successful execution of two-phase commit (2PC).

2PC uses a new component that does not normally appear in single-node transac tions: a coordinator (also known as transaction manager). The coordinator is often implemented as a library within the same application process that is requesting the transaction.

A 2PC transaction begins with the application reading and writing data on multiple database nodes, as normal.

We call these database nodes participants in the transaction.
When the application is ready to commit, the coordinator begins phase 1: it sends a prepare request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:
• If all participants reply “yes,” indicating they are ready to commit, then the coordinator sends out a commit request in phase 2, and the commit actually takes place.
• If any of the participants replies “no,” the coordinator sends an abort request to all nodes in phase 2.

If any of the prepare requests fail or time out, the coordinator aborts the trans action; if any of the commit or abort requests fail, the coordinator retries them indefinitely. However, it is less clear what happens if the coordinator crashes.

If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction. But once the participant has received a prepare request and voted “yes,” it can no longer abort unilaterally—it must wait to hear back from the coordinator whether the transaction was committed or aborted. If the coordinator crashes or the network fails at this point, the participant can do nothing but wait. A participant’s transaction in this state is called in doubt or uncertain.

Without hearing from the coordinator, the participant has no way of knowing whether to commit or abort. In principle, the participants could communicate among themselves to find out how each participant voted and come to some agreement, but that is not part of the 2PC protocol.

The only way 2PC can complete is by waiting for the coordinator to recover. This is why the coordinator must write its commit or abort decision to a transaction log on disk before sending commit or abort requests to participants: when the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log. Any transactions that don’t have a commit record in the coordinator’s log are aborted. Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.

Three-phase commit
Two-phase commit is called a blocking atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover. In theory, it is possible to make an atomic commit protocol nonblocking, so that it does not get stuck if a node fails. However, making this work in practice is not so straightforward.

Fault-Tolerant Consensus

The consensus problem is normally formalized as follows: one or more nodes may propose values, and the consensus algorithm decides on one of those values.

In this formalism, a consensus algorithm must satisfy the following properties:
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.

The uniform agreement and integrity properties define the core idea of consensus: everyone decides on the same outcome, and once you have decided, you cannot change your mind.

If you don’t care about fault tolerance, then satisfying the first three properties is easy: you can just hardcode one node to be the “dictator,” and let that node make all of the decisions. However, if that one node fails, then the system can no longer make any decisions. This is, in fact, what we saw in the case of two-phase commit: if the coordinator fails, in-doubt participants cannot decide whether to commit or abort.

It essentially says that a consensus algorithm cannot simply sit around and do nothing forever —in other words, it must make progress. Even if some nodes fail, the other nodes must still reach a decision.