Linearizability and Ordering

Linearizability

Linearizability = Strong Consistency

Once a write is committed to one replica for a given object, all reads (regardless of replica) of that object afterwards will show the results of the write (not be stale).
To a user, it seems as if there is just one copy of the data.

Do not confuse linearizability with serializability, this does not have anything to do with transactions!

When is linearizability useful?

When we want just ONE of something:

- One node can hold a lock (no read to the lock should be stale)
- Uniqueness constraints in database (Only I get to have the username goldenshowerlover100)

Pros and Cons of Linearizability

Pros:

- Always up to date results

Cons:

- Huge performance hit (once we see how to implement linearizability this will make more sense)

Linearizability and Ordering

If we are to act as if there is only one copy of the data, this means that there should be a total order of the operations on the data:

The state of the database should be able to be expressed by all the operations in some set order (no operations happen concurrently, but rather one at a time, the order needs to be the same for all replicas)

This order preserves causality: if write A happens before write B, A should be before B in the total order

Similarly, if writes A and B are concurrent, it doesn't matter where A and B are in the total order relative to one another, but each replica must have them in the same order

Total Order vs Partial Order

Total Order: A < B < C < D < E

Partial Order: A, B < C < D, E

You can express causality with just a partial order (see version vectors, concurrent operations do not need to be ordered)! However, in this case, the replicas may have inconsistent data:

- Replica 1 does A then B
- Replica 2 does B then A

How to create an ordering

Single Leader Replication:

- Write ahead log does this for you

Leaderless or Multileader Replication:

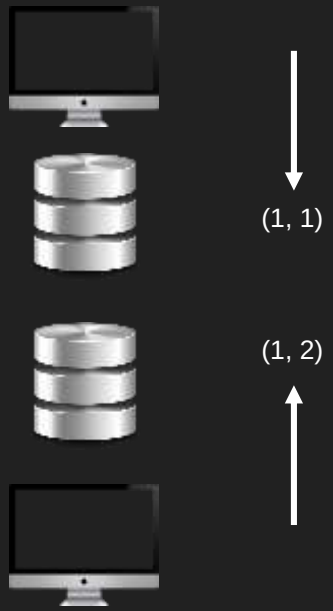
- Lamport Timestamps
- Version vectors work too (but can take a lot of space if many nodes)

Lamport Timestamp

(Counter, NodeId) - gives us a total order

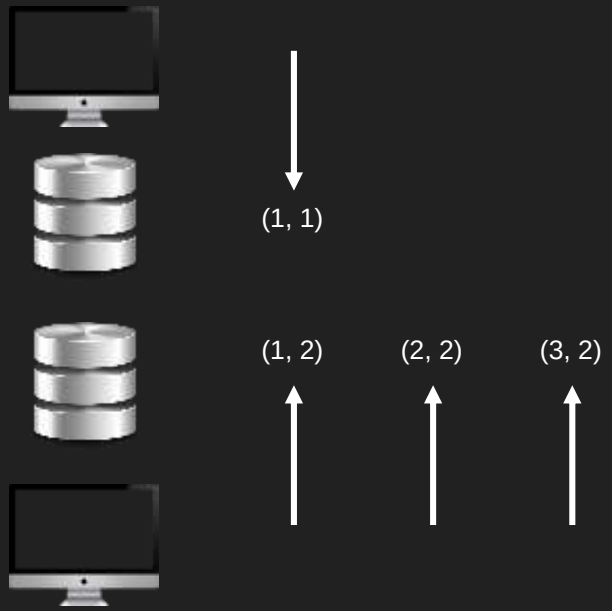
Lamport Timestamp

(Counter, NodeId) - gives us a total order



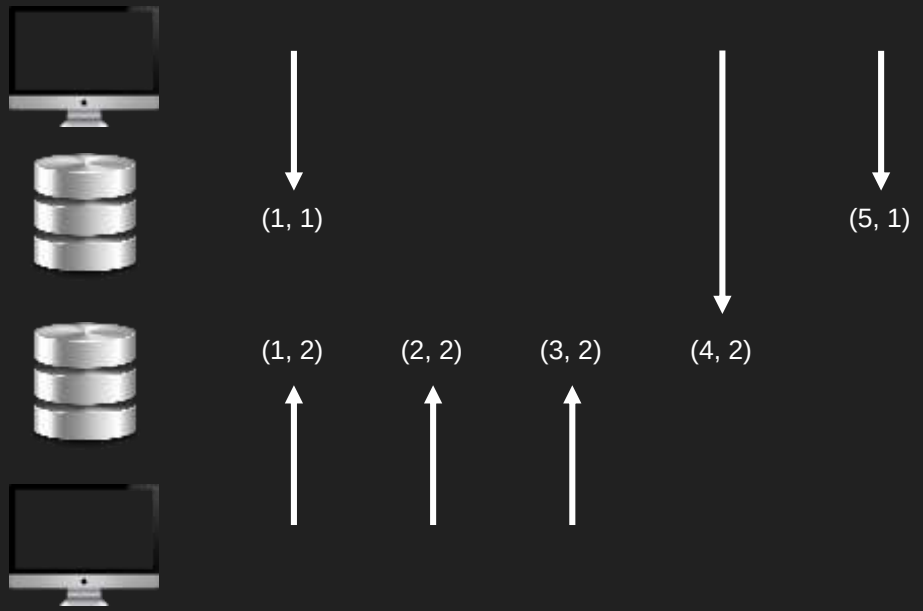
Lamport Timestamp

(Counter, NodeId) - gives us a total order



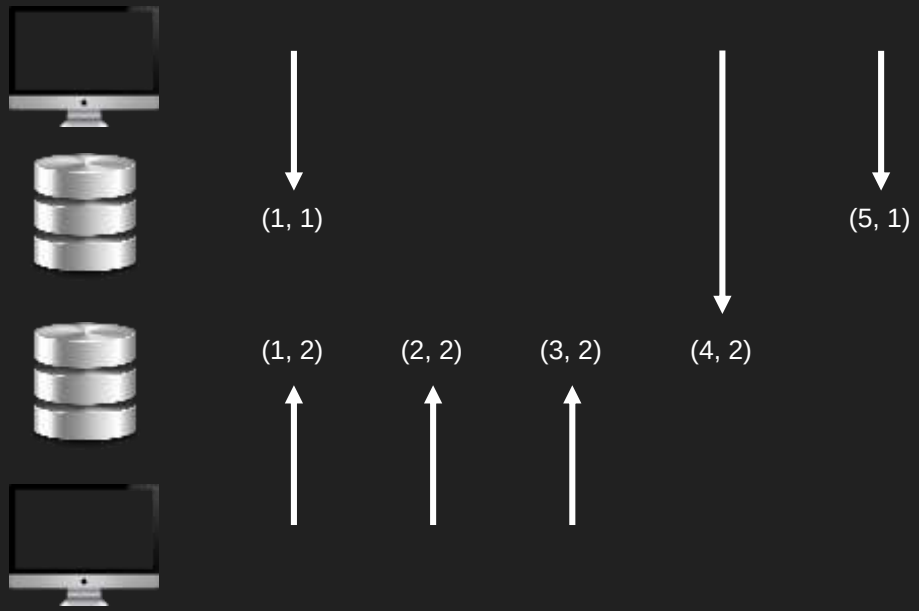
Lamport Timestamp

(Counter, NodeId) - gives us a total order



Lamport Timestamp

(Counter, NodeId) - gives us a total order



Every client and replica keeps track of the highest counter it has seen and if it sees a higher one, it skips to that number. Use an arbitrary tiebreaking order for nodeIds if the counters are the same.

Final order:
(1, 1)
(1, 2)
(2, 2)
(3, 2)
(4, 2)
(5, 1)

Lamport Timestamps Tradeoffs

Lamport Timestamps vs. Version Vectors:

- Lamport Timestamps take less space (O(1) vs. O(n))
- Version Vectors can express concurrent operations since they are a partial ordering, whereas Lamport Timestamps do not since they are a total order
  - Leads to data loss since one operation arbitrarily kept over another
  - No ability for custom merging/sibling logic

Lamport Timestamp Pitfalls

Although Lamport Timestamps provide us with a total ordering for operations, they are insufficient as they only give us this order retroactively:

Imagine my friend and I both want to claim the username fupa_sniffer_21, and both make writes to different replicas concurrently - using Lamport Timestamps both replicas will converge to the same state, but they will not know which operation “comes first” until all of the operations and their corresponding timestamps are sent to them. In the moment, both me and my friend will receive success messages on write.

Total Order Broadcast

It is not enough to order operations, rather each replica needs to receive every operation in the correct order as it happens:

- Messages delivered to every node without being lost (keep retrying if it is lost)
- Messages delivered in the same order to every node

A protocol that satisfies this properties is known as total order broadcast, useful for creating a replicated log

Total Order Broadcast and Linearizability

Writes:

- Add them in the log, replicas perform the writes in the correct order
  - Uniqueness constraint example: database performs first write to username, has error for any subsequent writes to username in log

Reads:

- Add them in the log, database will read up to date value consistent with timing of read

Consensus

While total order broadcast can be used to create linearizable storage, and linearizable storage can be used to create total order broadcast, we have not said how to create either of these.

Ultimately, doing so is equivalent to solving the consensus problem, we will discuss some ways of doing so in future videos.

Linearizability and Ordering Summary

Linearizability: As if there were only one copy of the data among replicas, strong consistency

Ordering: Easy in single leader replication (assuming no partitions), requires lamport timestamps otherwise. However, even Lamport Timestamps can only describe order of operations retroactively. To do so, we need Total Order Broadcast.

While Total Order Broadcast and Linearizability are not the same, they can both be reduced to the consensus problem.

If you can avoid having to deal with strong consistency, eventual consistency often suffices and takes a far lesser toll on performance.