# Designing Data Intensive Applications

## CHAPTER-9

## Consistency and Consensus

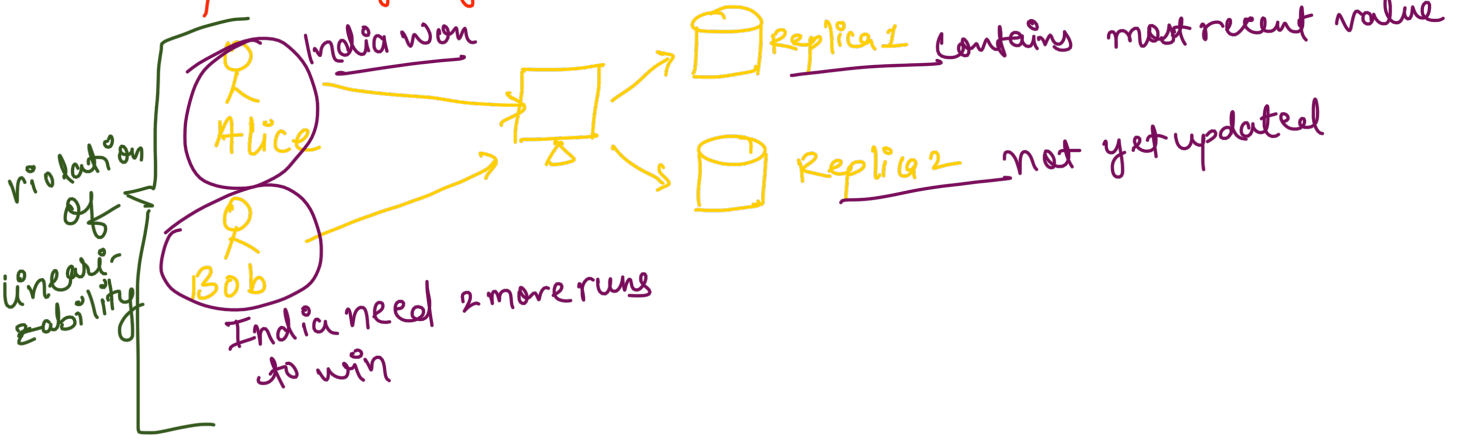
Subscribe for more such content...

YouTube – MsDeep Singh

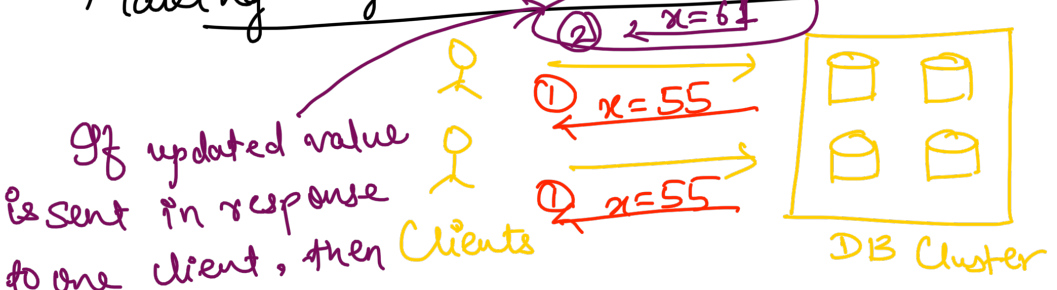
**Linearizability** → There could be multiple replicas in database but to client it appears like a single node.

provide strong/atomic consistency.

Example – ongoing cricket match




**Making system Linearizable**




x → current score of India

x=55 ↓ x=61

If updated value is sent in response to one client, then all clients should be sent updated score.

↑ even if write is in progress.

Comparing **Linearizability** with ~~Serializability~~

↑ recency guarantee on reads

↑ Isolation property of transactions

and writes of an object.

Transaction behaves as same if executed sequentially.

## why linearizability?

① Locking and leader election

↳ for single leader replication – should only be one leader, not more → split brain

a node can aquire lock on becoming leader.

linearizable. Example zookeeper, etcd.

② Uniqueness Gurantees → Example username should be unique for user.

should be linearizable operation.

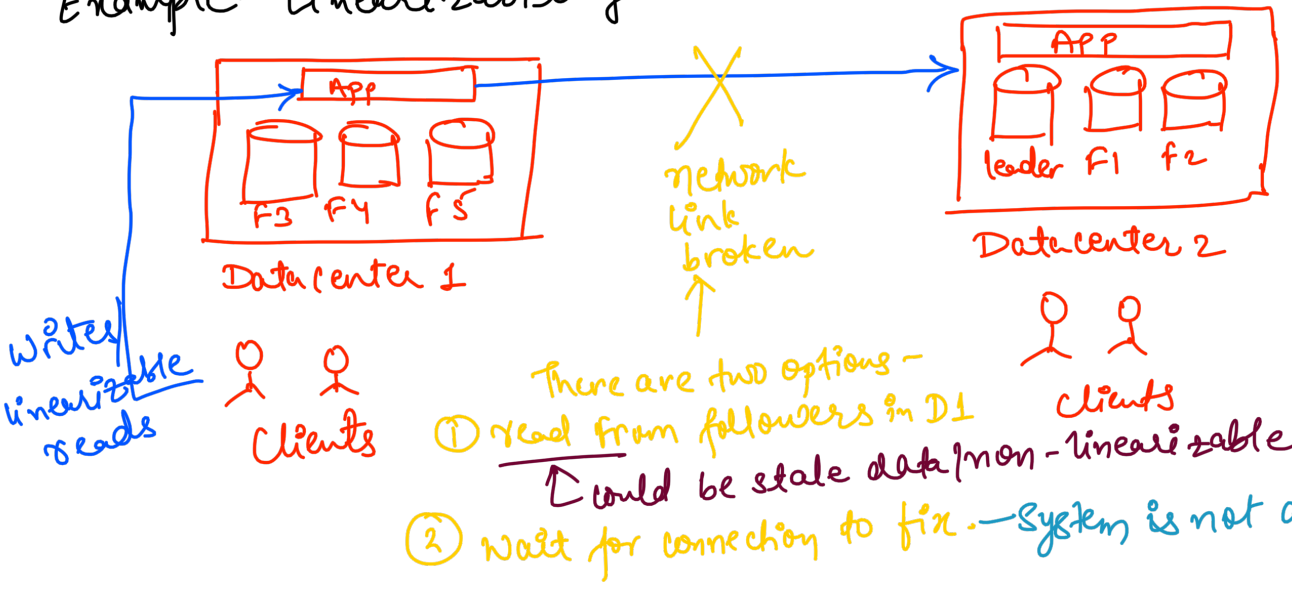
## Implementation

↳ keep only single copy → operation will be atomic by default.

System is not fault tolerant.

↳ Consensus Algorithms → can be used to implement linearizable systems

Example- Linearizability vs Availability. → Single leader replication




There are two options –

① read from followers in D1 ∴ could be stale data/non-linearizable – system is available

② wait for connection to fix. – system is not available

## CAP Theorem → Choose either 2 of Consistency, Availability and Partition Tolerance.

Unpractical, CA system is not possible.

## linearizability and network delays

↳ It is generally avoided (if our use-case permits) to improve performance.

It is slow even if there are no network faults.

# Ordering Guarantees → operations are executed in particular order

## ① Ordering and Causality

! → ordering helps in preserving causality.

→ Example

@ Consistent Prefix Reads → question should be asked before so as it can be answered. (cause → effect; causal dependency b/w que & ans)

Causality impose ordering in events
↑ cause comes first, then effect.

↳ if system obeys this ordering ↑ referred as causally consistent.

## Linearizability vs Causality

### Linearizability

Total order of operations

allows two elements to be compared. eg. $13 > 5$

we can always say which occurred first.

Database abstracts out that there are no concurrent events.

### Causality

Partial order

the elements can be incomparable or one could be greater than other.

eg. $\{a, b\}$ and $\{c, d\}$

↳ two operations are ordered if they're causally related. ↑ one happened before other.

↑ incomparable if concurrently occurred.

Example Git version control system.

↳ Any linearizable system will preserve causality correctly.

Stronger than causal system

hence performance can be slow.

## Sequence Number Ordering

↳ maintaining all causal dependencies could be large overhead.

→ for event ordering, sequence no or timestamps can be used. ↑ from logical clock.

assign seq no to each operation & these no can be compared. → use counters for each operation.

Sequence number for non-causal

example multi leader replication

1. Each node generate its own numbers.
   - node should have some identifier so as numbers are unique across clusters
   - eg one node generate odd and other even.
2. Attach Timestamp → used in last write wins algorithm.
3. Assign range of numbers to each node.

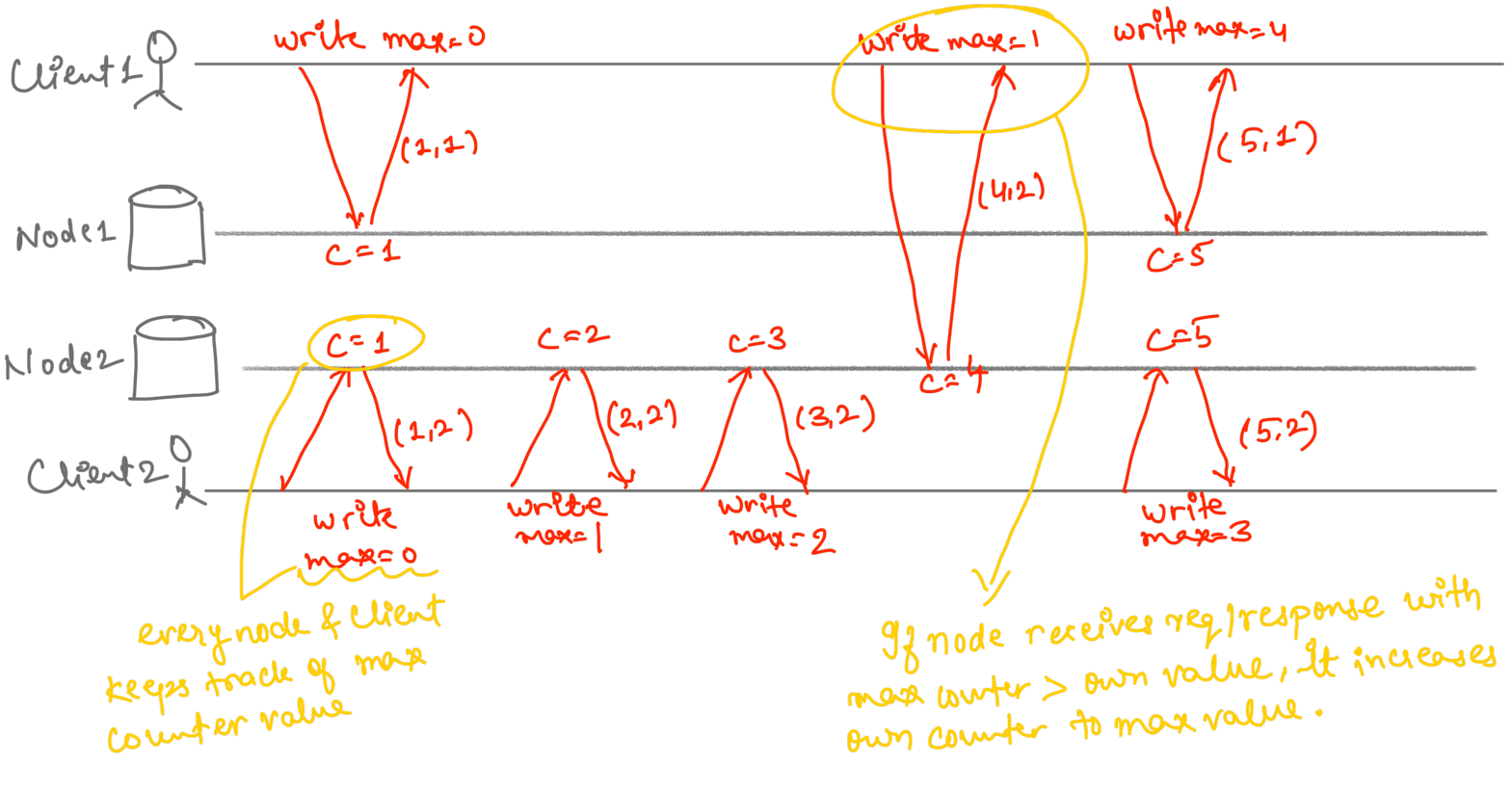
→ Generated seq numbers are not causally consistent:
they are not capturing order of events across the nodes.

Solution – Lamport Timestamps

{counter, nodeId}

One with greater timestamp is greater

if same time, greater nodeId is greater




The same username problem

Two users trying to create account with same username at same time.

↳ username creation operation is not collected yet, lamport timestamp will not help.

↳ A node should check with all other nodes if any request with same username.

↑ node has generated operation but don't know what they are?

↑ What if network fault between node communication.

To uniquely identify usernames, total ordering of operations is not sufficient ∑ you should when total order is finalized.

Total order Broadcast.

How can all nodes in cluster agree on same total ordering of operations

For single leader database? - one node as leader handles this.

what if leader fails?

Total order Broadcast → also used in consensus algorithms.

↳ protocol for exchanging msgs between nodes.
- → Reliable delivery
- → Totally ordered delivery → delivered to every node in same order

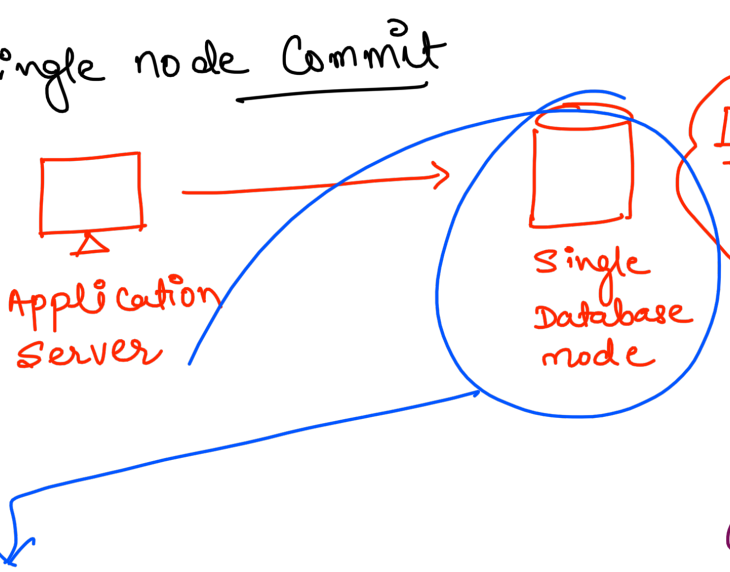
↳ If msg is already delivered, node is not allowed to insert it back in earlier position.

Distributed Transactions and Consensus → several nodes to agree on something

eg. leader selection

Atomic Commit and Two Phase Commit (2PC)

① Single node Commit




On transaction Commit
① Update WAL on disk
② append commit record to log on disk

what if DB crashes before #1 or after #1?
↳ rollback changes

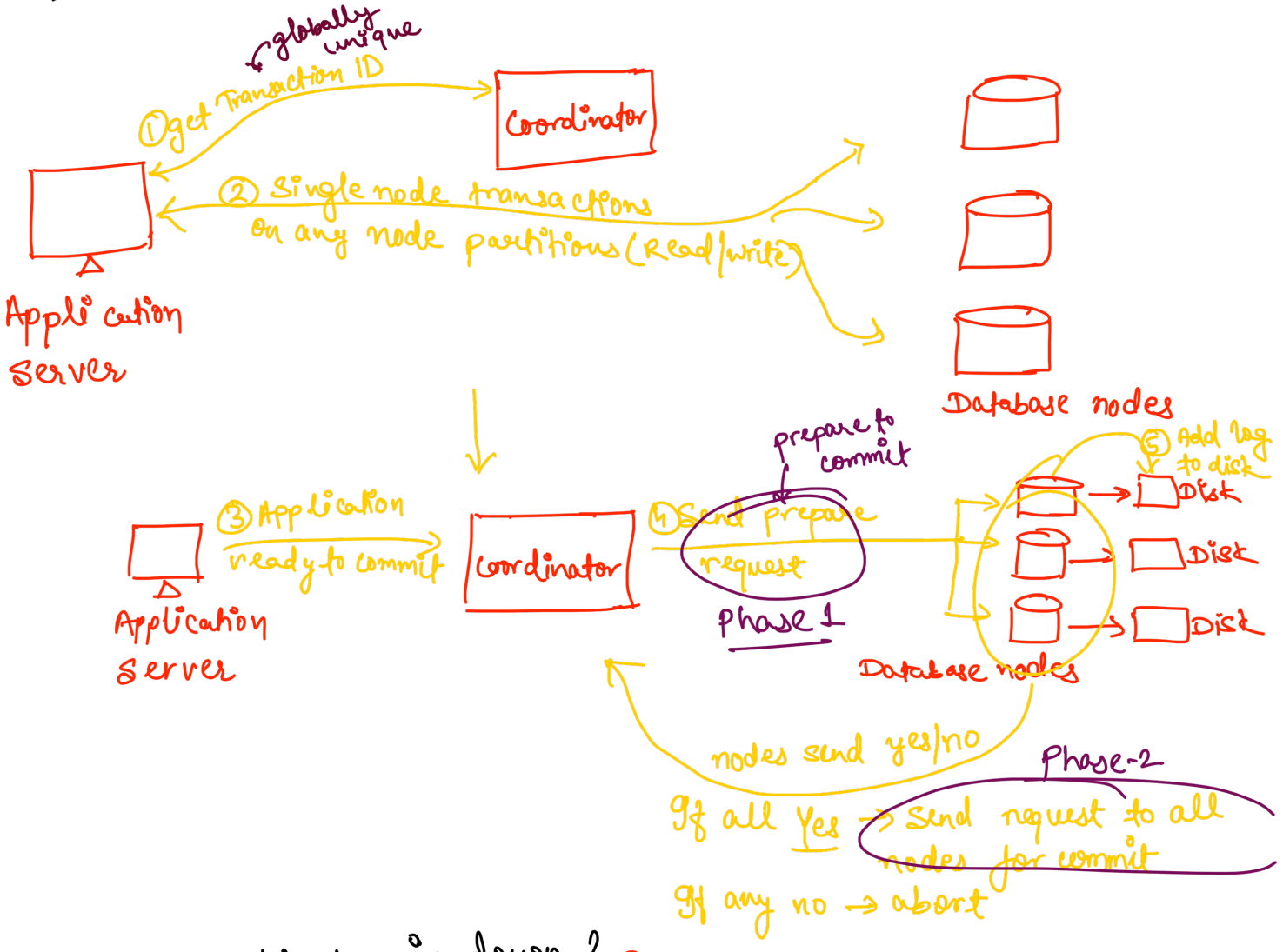
what if crashed after #2?
↳ recover from disk once node restarts.

What if multiple Nodes?

↳ we can't send commit msgs to all nodes. → what if it fails on few nodes? → network delay, node crash, etc.

Difficult to handle.

# Two Phase Commit (2PC) → the commit/abort is split into two phases.




## what if coordinator is down?

→ Coordinator is down after sending commit request to one node → other nodes will not know what to do with transaction but wait.

① Either database nodes interact with each to decide what to do with specific transaction. (not part of 2PC)

② Coordinator dumps all actions to disk so it can recover once back online.

→ what if disk is also corrupt?
↳ manual intervention by admin.

## Distributed Transaction Examples

① Database Internal → internal transactions among the nodes in cluster.

② Heterogeneous distributed transactions → different technologies
ex:
→ two databases from different vendors, message broker & database

extentended Architecture

## XA Transactions → 2PC implementation across heterogeneous tech

↳ C API for interfacing with transaction coordinator.

↳ in Java - Java Transaction API → network driver or client library

The driver can send required info to coordinator (prepare | commit | abort)

## Fault Tolerant Consensus → safety properties

1. Uniform agreement among nodes → no two nodes decide differently
2. Integrity → node decides only once
3. validity → nodes decides value v, then v was proposed by some node.
4. Termination → every node decides a value that doesn't crash → system should reach a decision even if node crashes.
   ↳ Liveness property

Consensus algo properties

## Single Leader Replication and consensus

How is leader chosen? (in case of failovers)
- manually by operation team
- automatic selection

Can split brain problem occur?
→ We need all nodes agree for leader.
↳ consensus

Consensus algo are based on total order broadcast.
like a single leader replication
↳ and it requires a leader

Spiral loop

Solution → Epoch numbering for quorum

→ protocols define epoch number → within each epoch, leader is unique.

If leader dead → election is given an incremented epoch number → monotonic increasing

helpful in resolving conflicts.

## Coordination services eg. zookeeper, etcd

1. Linearizable atomic operations → for concurrent operation on nodes, only one succeed using atomic compare and set operation.
   consensus

② Total ordering of messages
↳ fencing tokens are used to avoid conflicts on lock lease.
↳ Zookeeper use monotonically increasing transaction Id & version number
③ Failure Detection → via heartbeats.
④ Change notifications → new nodes added/removed

See you in next video ___

Happy Learning 

Don't forget to subscribe...