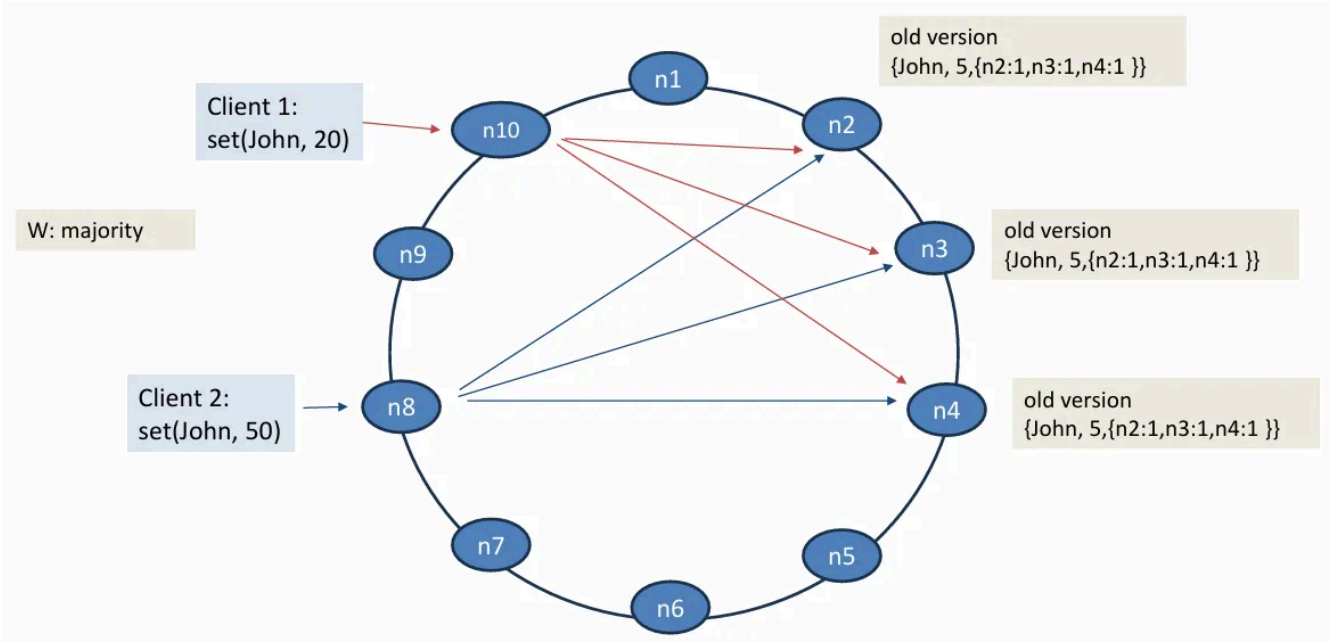
Lecture 12

Type	Lecture

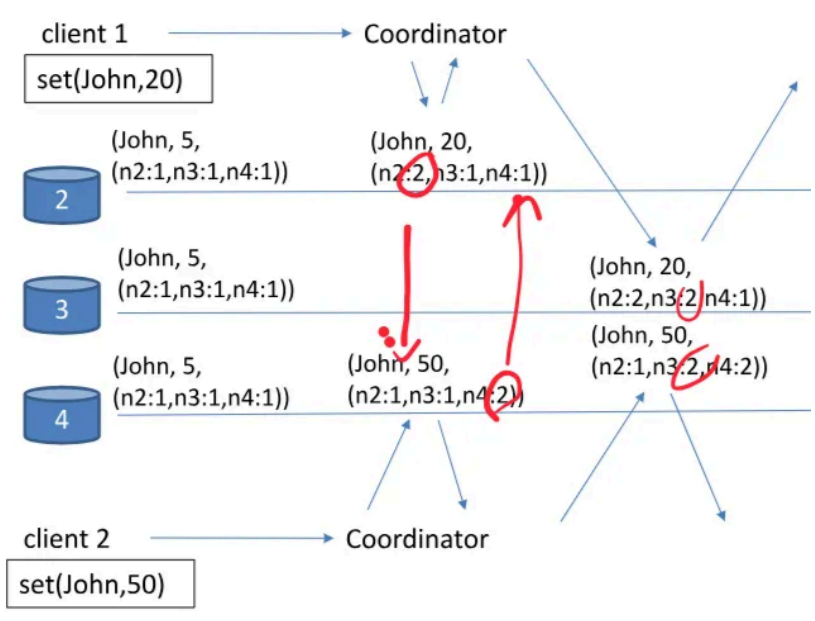
Concurrent Writes –P2P Distributed Databases



- 10 nodes in a distributed system: `n1` to `n10`.
- old version for nodes n2, n3, n4 → `{John, 5, {n2:1, n3:1, n4:1}}`
- Two clients, each sending a write request to update the value of `"John"`:
  - `Client 1: set(John, 20)`
    - coordinator node n10 sends the write operation to replication node and then it is replicated on other two nodes, so nodes n2, n3 and n4 have the same write operation.
  - `Client 2: set(John, 50)`
    - coordinator node n8 sends the write operation to replication node and then it is replicated on other two nodes, so nodes n2, n3 and n4 have the same write operation.

These requests are executed concurrently on the same base version of the data.

What happens:



- **Initial State**: All nodes have:

  `John = 5` , `VV = {n2:1, n3:1, n4:1}`

- Client 1 sends the set `set(John, 20)` operation and Client 2 send the set `set(John, 50)` operation, coordinator nodes send this operation to the respective nodes

- Client 1's coordinator sends to n2 and increments its counter → `VV_2 = {n2: 2, n3: 1, n4: 1}`

  Client 2's coordinator sends to n4 and increments its counter → `VV_4 = {n2: 1, n3: 1, n4: 2}`

The VV's are not comparable since $VV_2 > VV_4$ and $VV_2 < VV_4$ on different indexes → Concurrent writes.

- Client 1's coordinator sends to n3 and increments its counter → `VV_2 = {n2: 2, n3: 2, n4: 1}`

  Client 2's coordinator sends to n3 and increments its counter → `VV_4 = {n2: 1, n3: 2, n4: 2}`

Are both writes successul?

Both writes are successful, because both clients reached a quorum (2 out of 3 nodes: nodes 2, 3, 4).

Are the VVs comparable?

VV's are not comparable, because the version vectors are concurrent. Neither version is causally after the other ⇒ it is a true write conflict.

---

1. Sibling Values

What happens to the VVs?

If `.allow_mult(true)` is enabled, both VVs are stored as siblings with their respective values.

```
[
  { "value": 20, "VV": {n2:2, n3:2, n4:1} },
  { "value": 50, "VV": {n2:1, n3:2, n4:2} }
]
```

What is returned to the clients?

Both values are returned to the client on read

(Client 1 gets `{John, 20}` and Client 2 gets `{John, 50}`):

- The client application receives all siblings.
- The app must then merge or choose one value.

Once resolved, the application writes back a new value:

- The chosen value is written with an updated VV.
- All replicas replace the siblings with this new version.

In systems like Dynamo or Riak, which favor high availability and eventual consistency, write conflicts must not block progress:

- So rather than rejecting or forcing coordination, the system accepts both writes and defers resolution.
- This ensures availability, even during network partitions or concurrent writes.

---

2. Last-Write-Wins (LWW)

How LWW Works:

If `last_write_wins(true)` is enabled in a system like Riak:

- The system chooses one write as the "winner".
- It discards all other concurrent values, based on some internal logic.
- Criteria for "Last" Write:

- Timestamp of the write (most common).
- Or lexicographic ordering of node IDs as tiebreakers.
- Might even use vector clocks or system clocks.

What is returned to the clients?

In our case, If the system considers client 2's write as the latest (e.g., based on timestamp), it will return `"John = 50"` and discard `"John = 20"`.

What happens to the VVs?

The VV is discarded entirely:

- No sibling versions.
- No history of who wrote what or when.
- No causality tracking.

What LWW does:

- LWW simplifies resolution by refusing to track concurrency.
- It's a trade-off: simplicity vs correctness/precision.

❗ **Problems with LWW:**

- Data loss risk: Concurrent writes are silently dropped.
- No visibility: The client has no idea there was a conflict.
- Simplicity: No need for complex resolution logic.
- Prevents sibling growth, which simplifies storage and read logic.

Summary:

- LWW is useful for systems where simplicity and high availability are more important than full consistency or causality.
- It is not suitable where every write must be preserved, such as collaborative editing, financial records, or legal data.

---

3. CRDT (Conflict-Free Replicated Data Type)

CRDTs are data structures designed for replicated environments (like distributed databases) that can:

- Be updated concurrently on different nodes,
- Be merged automatically, and
- Guarantee eventual consistency without conflicts

Conflict-Free Replication Data Structures:

you never have to manually resolve conflicts because the data structure does it inherently.

Convergent Replicated Data Types (CvRDTs):

- Used in systems like Riak

- Nodes replicate state and merge entire states
- Merge function must be commutative, associative, and idempotent

Commutative Replicated Data Types (CmRDTs):

- Used in systems like Redis
- Instead of syncing full state, they sync operations
- Operations must commute, so order doesn't matter

> "A conflict-free replicated data type (CRDT) is a data structure that is replicated across multiple computers in a network, with the following features:

1. **Independent & Concurrent Updates:** Each replica (node) can be updated without coordination.
2. **Automatic Conflict Resolution:** The data type contains logic to merge any divergent states.
3. **Guaranteed Convergence:** Despite temporarily diverging, all replicas will eventually converge to the same state.

In traditional replication concurrent writes often lead to conflicts, requiring Version vectors (to detect), Siblings or LWW (to resolve).

In CRDTs:

- These conflicts never appear — the data type handles it internally.
- The system becomes simpler and more available, especially in AP systems (Availability + Partition Tolerance in CAP).

---

how Riak uses CRDTs to solve the problem of conflict resolution in distributed systems:

Riak's CRDTs are often implemented as operation-based CRDTs (CmRDTs). This means clients send operations (e.g., increment by 1), not full state. This is efficient and avoids large replication overhead.

Distributed systems deal with concurrent updates, network partitions, and replica divergence. Without CRDTs, Riak users had to manually handle:

- Version vectors
- Siblings (conflicting versions)
- Timestamps, vector clocks

CRDTs abstract away the complexity of:

- Detecting conflicts
- Merging state
- Ensuring convergence

This simplifies client-side code significantly and promotes developer productivity.

Each data type (like counters, sets, maps) has built-in merge logic. This ensures eventual consistency, meaning all nodes converge without human intervention.

No matter how good the system, concurrent updates or network partitions will happen. Riak accepts this reality and embraces conflict-tolerant design using CRDTs.

Traditional systems use:

- Timestamps → susceptible to clock drift

- Vector clocks → hard to manage as system scales
- Dotted version vectors → more accurate but complex

Riak handles everything internally using CRDTs. Developers are shielded from the complexities of distributed consistency and concurrency.

The core of CRDTs is that write operations must commute.

- Commutative: `a + b = b + a`
- This ensures that ordering doesn't matter, which is crucial in distributed systems where updates arrive out of order.

With this property:

- All nodes can apply changes in any order and still reach the same final state.
- CRDTs inherently support Strong Eventual Consistency (SEC).

Example for a CRDT data type: Grow-Only Counter (G-Counter):

- Each node maintains a local counter.
- The system maintains a vector with one slot per node:

```
(n1: 2, n2: 3, n3: 0)
```

- Each node can only increment its own slot.
- When nodes synchronize, they merge the vectors using:

```
merge(a, b) = element-wise max(a, b)
```

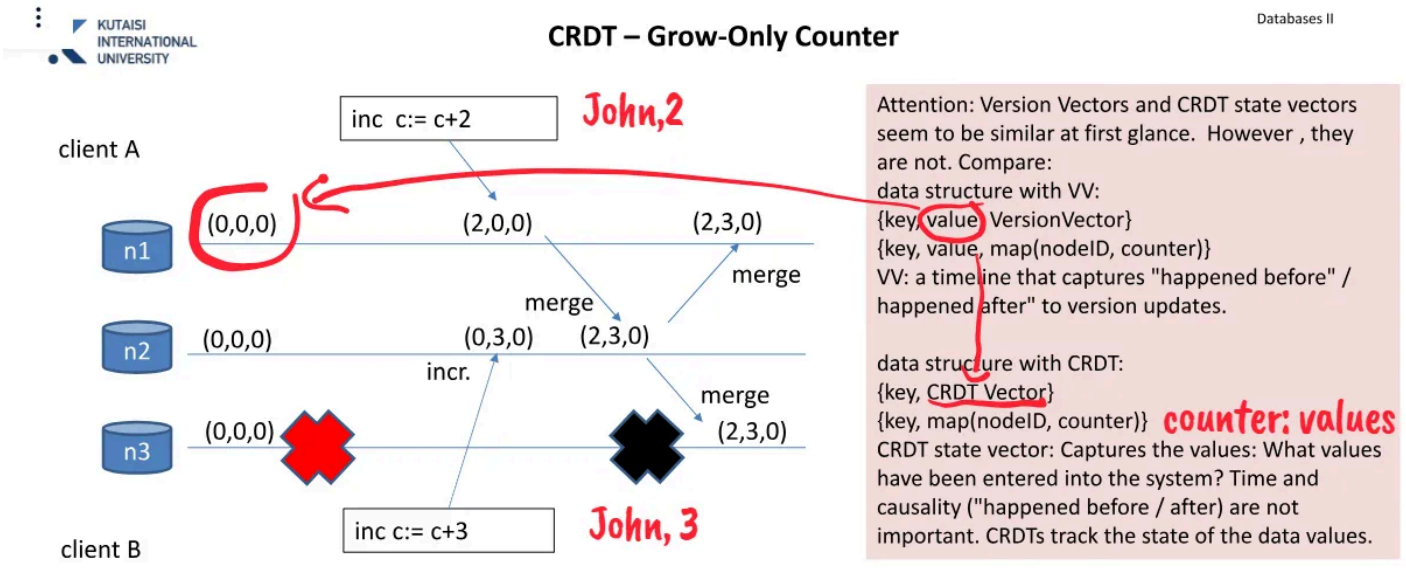
Merge Function Properties:

- **Commutative** → order doesn't matter
- **Associative** → grouping doesn't matter
- **Idempotent** → applying the same update multiple times doesn't change the result

Resulting value:

To get the actual count: **sum all elements of the vector** → `total = sum(n1 + n2 + n3)`

CRDT – Grow-Only Counter



All nodes (`n1`, `n2`, `n3`) start with: `(0, 0, 0)` Each slot corresponds to a node's local count.

- Client A updates n1:
  - Performs `inc: c := c + 2`
  - Node `n1` now has: `(2, 0, 0)` Only its own slot (index 0) is updated.

  Client B updates n2:
  - Performs `inc: c := c + 3`
  - Node `n1` now has: `(0, 3, 0)` Only its own slot (index 1) is updated.
- Nodes merge:
  - First Merge: `n1` and `n2` exchange state: `merge((2,0,0), (0,3,0)) = (2,3,0)`
  - Second Merge: `n1` and `n3` then sync: `merge((0,0,0), (2,3,0)) = (2,3,0)`
- all nodes converge to `(2,3,0)`

To retrieve the actual counter value (The current vector state): `sum(2,3,0) = 2 + 3 + 0 = 5`. So, across the distributed system, the total increment seen is 5.

State-Based CRDT

The merge method in State-Based CRDTs must have three critical properties:

Property	Meaning
Commutative	Order of merging doesn't matter: `merge(a, b) = merge(b, a)`
Associative	Grouping doesn’t matter: `merge(a, merge(b, c)) = merge(merge(a, b), c)`
Idempotent	Repeated merges don’t change result: `merge(a, a) = a`

These properties ensure that:

- Updates can arrive in any order.
- Merging is safe even with delayed or repeated messages.
- All replicas will eventually converge.

Replication Protocol requirement:

- No need for total order delivery or coordination.
- As long as updates eventually reach all replicas, the system is correct.
- This relaxes the consistency requirement to achieve high availability.

Client Behavior in State-Based CRDTs:

- Instead of sending operations (e.g., “increment by 1”), the client sends: `{key, current state}`
- This makes state-based CRDTs simpler to reason about, but possibly heavier in bandwidth since full state is shared.

Operation-based CRDTs **send only the operations**, not the full state.

Aspect	State-Based (CvRDT)	Operation-Based (CmRDT)
What is sent?	Full state	Individual operations
Reliability requirement	Eventual delivery	Reliable, exactly-once delivery
Idempotence required?	Yes	Not always – must ensure no duplicates
Bandwidth use	Higher (due to full state)	Lower (due to small ops)
Ease of implementation	Easier (merge only)	Harder (requires op logs, delivery guarantees)

Because ops may not be idempotent, care must be taken:

- Either enforce exactly-once delivery, or
- Track operations to avoid duplicates

Once received, however, ordering still doesn't matter thanks to CRDT design.

Summary:

- State-based CRDTs replicate entire state, merged using a function that is:
  - Commutative, Associative, and Idempotent
- This makes them robust against message loss, duplication, and out-of-order delivery.
- Client sends state, not operations.
- Operation-based CRDTs trade off bandwidth for complexity and delivery guarantees.

Strong Eventual Consistency

Strong Eventual Consistency (SEC) is a formal property in distributed systems that ensures not just eventual convergence, but correct convergence based on delivered updates.

- If two replicas receive the same set of updates (not necessarily in the same order),
- Then their final states must be identical.

A state-based CRDT satisfies SEC. A state-based CRDT ensures that if all replicas receive the same updates, their merge logic guarantees identical results.

Summary Replication

Single leader replication:

- Primary node (the leader) handles all write requests.
- Followers/secondaries replicate the data by applying the writes in order.
- If the leader fails, a failover process elects a new leader.

Advantages:

- Provides **strong consistency** if secondaries are read-only.

Disadvantages:

- Write availability is limited to the leader.
- Failover is complex and can be slow.
- May not scale well under heavy write loads.

Peer-to-peer replication:

- Any available replica can accept writes (no central leader).
- Writes are accepted if a quorum of nodes responds.
- Unavailable replicas may miss writes temporarily and catch up later.
- *"No failover"* → there's no single point of failure.
- Concurrent Write Conflicts:
  - Since any replica can write independently, concurrent writes to the same key can result in conflicts.
  - Requires conflict detection and resolution mechanisms (like version vectors or CRDTs).
- Eventual Consistency:
  - Eventually each node sees every operation. Eventually all nodes have the same state.
- Strong Eventual Consistency (SEC):

    - Using data structures that guarantee a conflict-resolution merge.

Finally:

- Single-leader replication is easier for strict consistency but limits availability.
- P2P replication prioritizes availability and fault tolerance, but requires mechanisms (like CRDTs) to ensure convergence.