Cloud-Based Data Processing

Consistency

Jana Giceva

Consistency guarantees

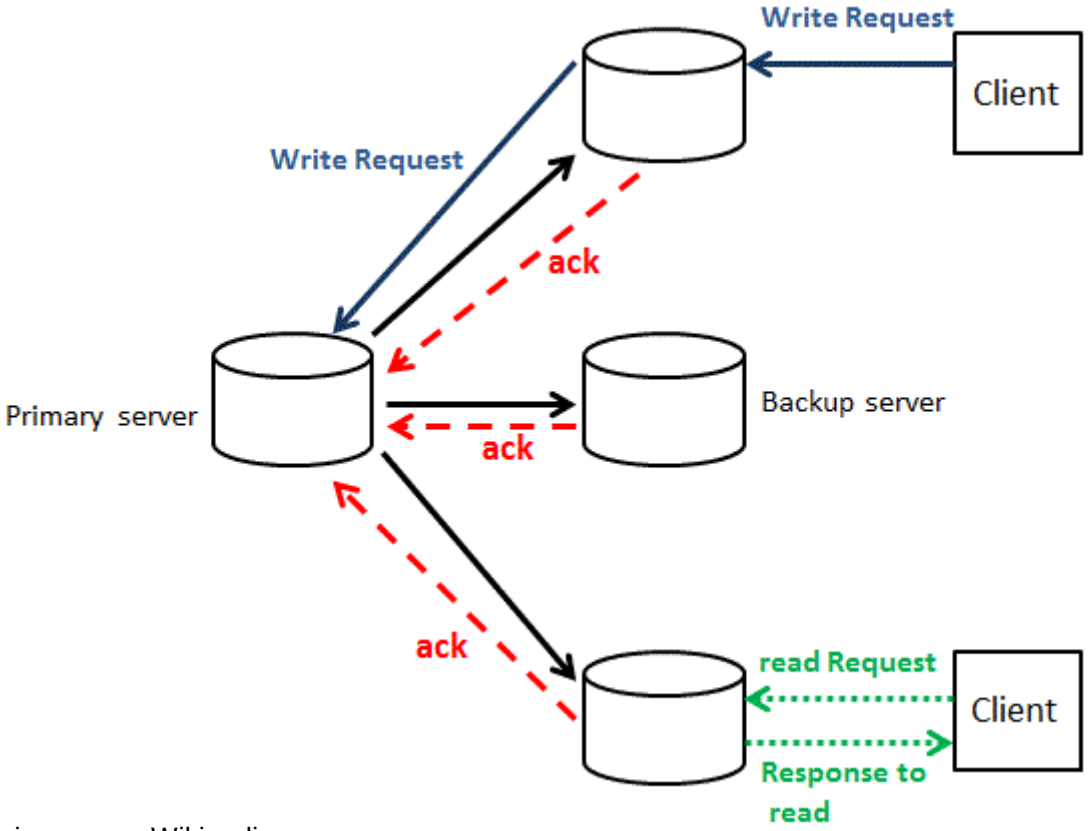


image src: Wikipedia

- Consistency guarantees needed when data is replicated!
- Inconsistencies occur regardless of the replication method.
- Most systems provide at least eventual consistency – all replicas eventually converge to the same value/state.
- The edge cases of eventual consistency only become apparent when there is a fault in the system or at high concurrency.

Consistency models vs isolation levels

- **Consistency models vs isolation levels:**

Transaction isolation – avoiding race conditions due to concurrently executing transactions.

- Serializable
- Repeatable read
- Read committed
- Read uncommitted

Distributed consistency – coordinating the state of replicas in the face of delays and faults.

- Strict
- Linearizability
- Causal
- Monotonic reads/writes
- Read-your-writes
- etc.

Consistency

Consistency

A word that means many different things in different contexts:

- **ACID:** a transaction transforms the database from one **consistent** state to another

  Here **consistent** = satisfying application specific invariants
  e.g., every course with students enrolled must have at least one lecturer

- **Read-after-write consistency** (Lecture 3: Replication)

- **Replication:** replica should be **consistent** with other replicas
  - **Consistent =** in the same state (when exactly?)
  - **Consistent =** read operations return same result?

- **Consistency model:** many to choose from

Distributed Transactions

- Recall **atomicity** in the context of ACID transactions
  - A transaction either **commits** or **aborts**
  - If it commits, its updates are durable
  - If it aborts, it has no visible side-effects
  - ACID consistency (preserving invariants) relies on atomicity

- If the transaction updates data on multiple nodes, this implies
  - Either all nodes must commit, or all must abort
  - If any node crashes, all must abort

- Ensuring this is the **atomic commitment** problem.
  Looks a bit similar to **consensus?**

Atomic commit versus consensus

Consensus

- One or more nodes propose a value
- Any one of the proposed values is decided
- Crashed nodes can be tolerated, as long as a quorum is working.

Atomic commit

- Every node votes whether to commit or abort
- Must commit if all nodes vote to commit; must abort if 1+ nodes vote to abort
- Must abort if a participating node crashes

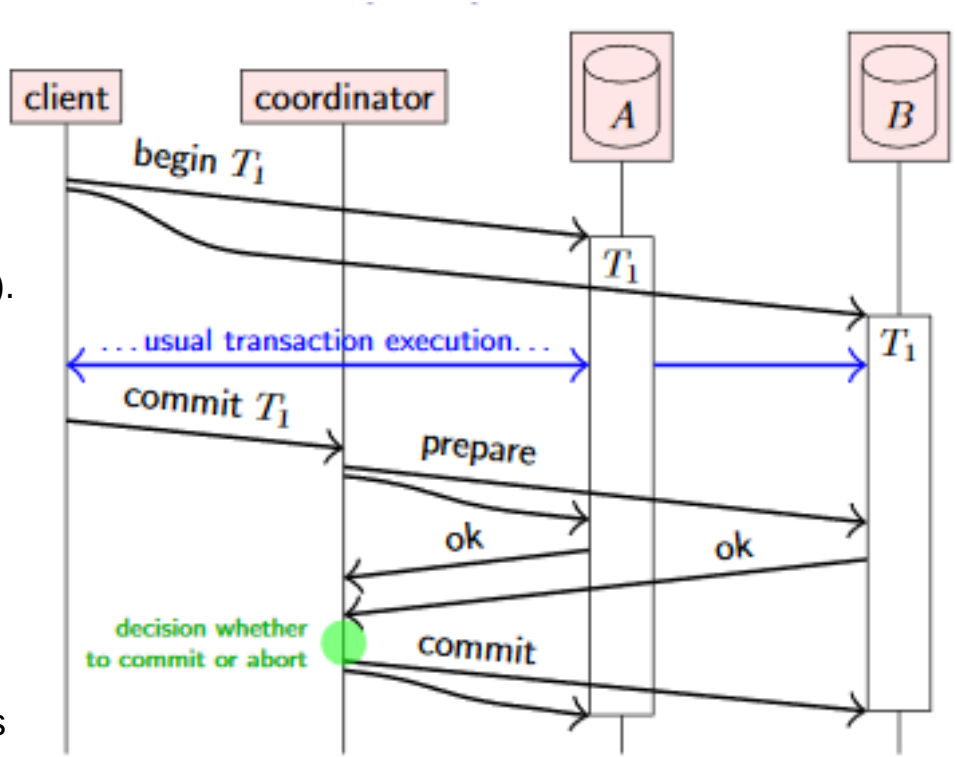
Two-phase commit (2PC)

- **Two-phase commit protocol**
  most common algorithm to ensure atomic commitment across multiple nodes.
  [Gray, 1978]

- Not to be confused with two-phase locking (2PL).

- **What if the coordinator crashes?**
  - Coordinator writes the decision to disk
  - When it recovers, it reads decision from disk and sends it to replicas
  - Problem if coordinator crashes after prepare, but before broadcasting decision
  - Algorithm is blocked until coordinator recovers

- **Solution: consensus (total order broadcast).**



Fault-tolerant 2PC (1/2)

- **Fault tolerant 2PC** based on **Paxos Commit** [Gray and Lamport, 2006]

- Every node that participates in the transaction uses total order broadcast to disseminate its vote (commit or abort).

- If node A suspects that node B has failed, then A may try to vote to abort on behalf of B.

on initialisation for transaction T **do**
  $commitVotes[T] := \{\}$; $replicas[T] := \{\}$; $decided[T] := \text{false}$
end on

on request to commit transaction T with participating nodes R **do**
  for each $r \in R$ **do** send $(\text{Prepare}, T, R)$ to r
end on

on receiving $(\text{Prepare}, T, R)$ at node $replicaId$ **do**
  $replicas[T] := R$
  $ok =$ "is transaction T able to commit on this replica?"
  total order broadcast $(\text{Vote}, T, replicaId, ok)$ to $replicas[T]$
end on

on a node suspects node $replicaId$ to have crashed **do**
  for each transaction T in which $replicaId$ participated **do**
    total order broadcast $(\text{Vote}, T, replicaId, \text{false})$ to $replicas[T]$
  end for
end on

Fault-tolerant 2PC (2/2)

- **Fault tolerant 2PC** based on **Paxos Commit** [Gray and Lamport, 2006]
- Potential race condition if two (conflicting) votes received from the same node (e.g., by node A voting on B's behalf).
- Resolved due to total order broadcast + counting only the first vote to arrive.

on delivering $(\text{Vote}, T, replicaId, ok)$ by total order broadcast **do**
  if $replicaId \notin commitVotes[T] \ \wedge \ replicaId \in replicas[T] \ \wedge$
      $\neg decided[T]$ **then**
    if $ok = \text{true}$ **then**
      $commitVotes[T] := commitVotes[T] \cup \{replicaId\}$
      if $commitVotes[T] = replicas[T]$ **then**
        $decided[T] := \text{true}$
        commit transaction T at this node
      end if
    else
      $decided[T] := \text{true}$
      abort transaction T at this node
    end if
  end if
end on

Linearizability

Linearizability

- **Multiple nodes concurrently accessing replicated data.**

  How do we define **consistency** here?

- The strongest option: **linearizability** [Herlihy and Wing, 1990]
  - Informally: every operation takes effect **atomically,** sometime after it started and before it finished
  - All operations behave as if executed on a **single copy** of the data
    even if there are in fact multiple replicas

  - Consequence: every operation returns an “up to date” value (a.k.a. “strong consistency”)

    Not just for distributed systems, also in shared memory concurrency
    (memory on multicore CPUs is not linearizable by default)

- Note: linearizability ≠ serializability!

Linearizability vs Serializability

- **Serializability** – isolation property of **transactions,** where every transaction may read and write multiple objects. It guarantees that transactions will behave the same as if they had executed in **some** serial order.

- **Linearizability** – recency guarantee on reads and writes of a register (an individual object). It does not group operations together into transactions, so it does not prevent problems such as write skew or phantoms, unless you take additional measures such as materializing conflicts.

- A database may provide **both** Serializability and Linearizability, and this combination is known as **strict serializability** or **strong one-copy serializability (strong 1SR).**

  Implementations of 2PL or actual serial execution are typically linearizable.
  However, SSI (serializable snapshot isolation) is not, by design.

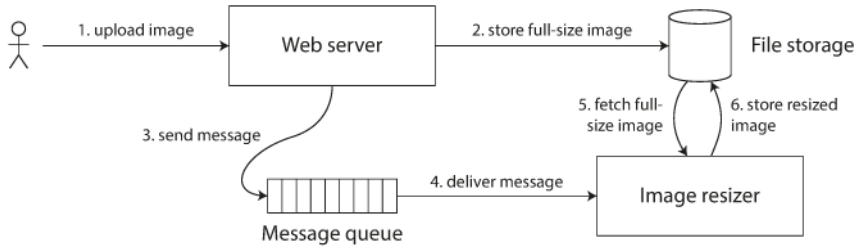
When do we want linearizability?

- In what circumstances is linearizability useful?

- **Locking and leader election**
  Coordination services like ZooKeeper and etc are often used to implement distributed locks and leader election. Libraries like Apache Curator provide higher-level recipes on top of ZooKeeper (e.g., how to actually implement locks).

- **Constraints and uniqueness guarantees**
  Uniqueness constraints are common in databases. If you want to enforce it, you need linearizability. The operation is similar to an atomic compare-and-swap.

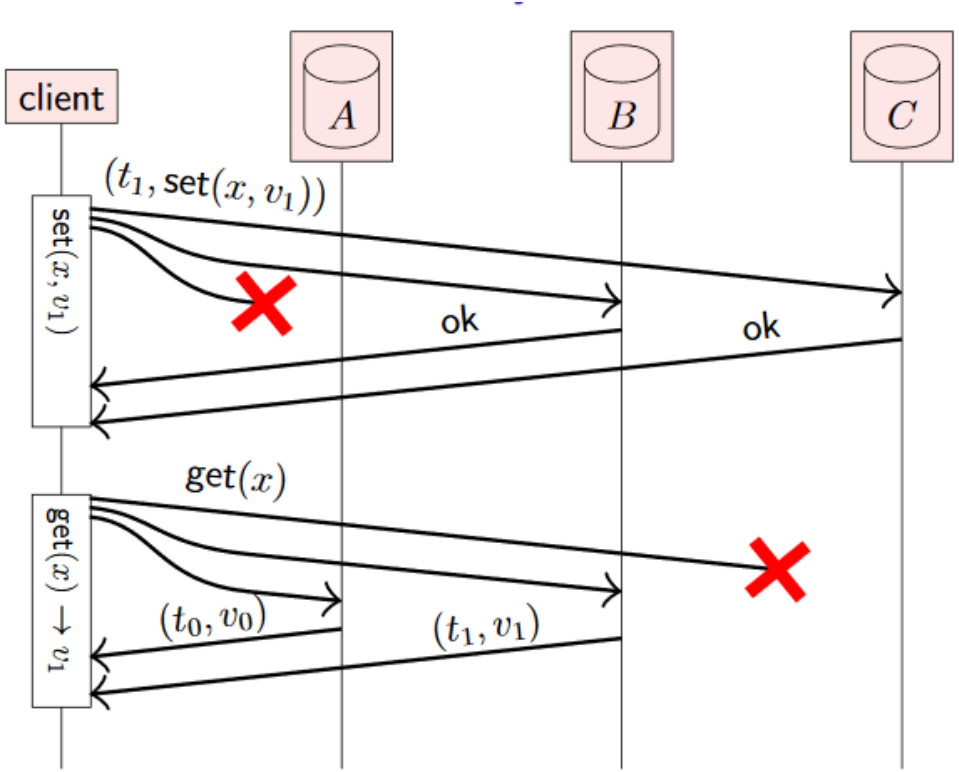
- **Cross-channel timing dependencies**
  Coordinating message delivery through more than one communicating channel.



Read-after-write consistency revisited

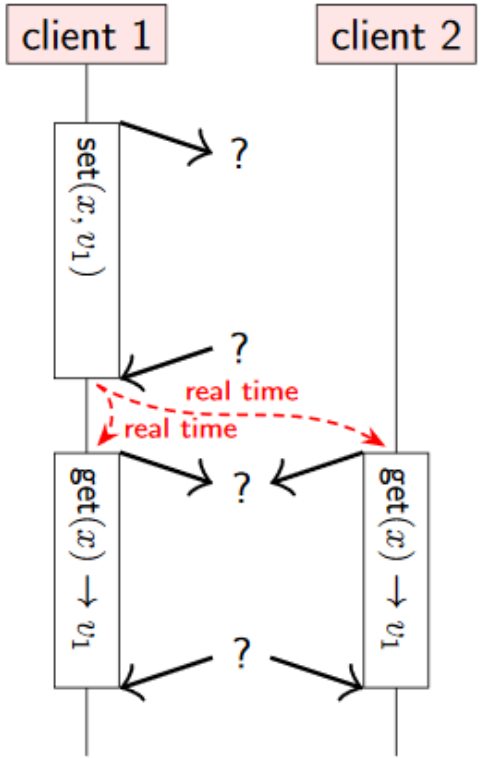
- The main purpose is to guarantee that nodes observe the system in an “up-to-date” state.
- We saw it before with **read-after-write consistency,** but it defines consistency model for reads and writes by the same node.

  Linearizability generalizes this idea to operations made concurrently by different nodes.
- From the client’s side, an operation takes some amount of time:
  - starts when the request is sent,
  - Finishes when the result is returned.



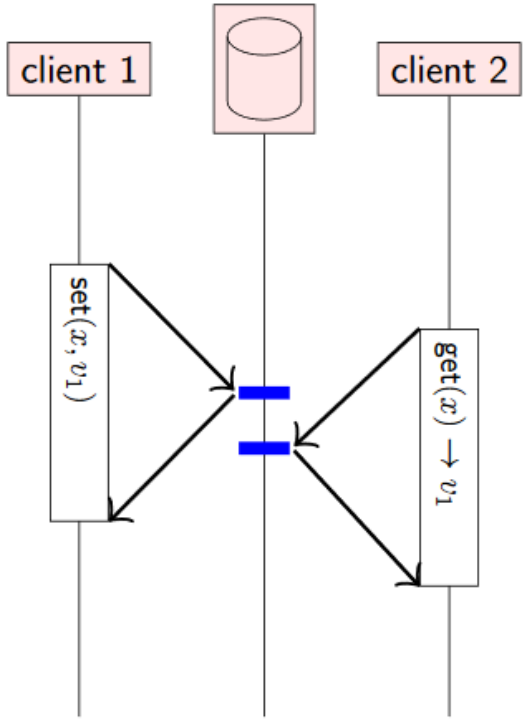
From the client's point of view

- Focus on **client-observable behavior**:
  when and what an operation returns.
- Ignore how the replication system is implemented.
- **Did operation A finish before operation B started?**
- Even if the operations are on different nodes.
- This is not happens-before:
  we want client 2 to read value written by client 1,
  even if the clients have not communicated!



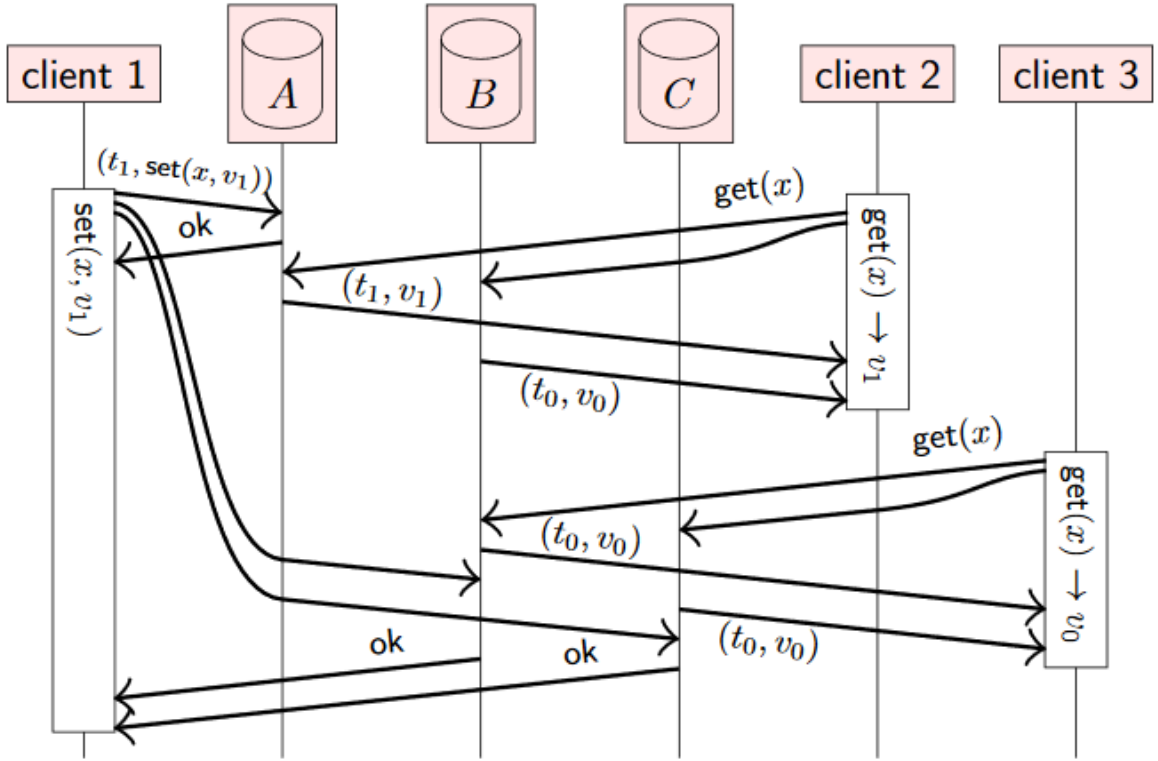
Operations overlapping in time

- Client 2's `get` operation overlaps in time with client 1's `set` operation.
  - Maybe the `set` operation takes effect first?
  - But also, the `get` operation may be executed first.
- Either outcome is fine in this case.
- Note that, operation A finishing before operation B started is not the same as A happened before B.
  - It is possible that two operations do not overlap in time, but are still concurrent according to the happens-before relation
  - because no communication has occurred between the operations.

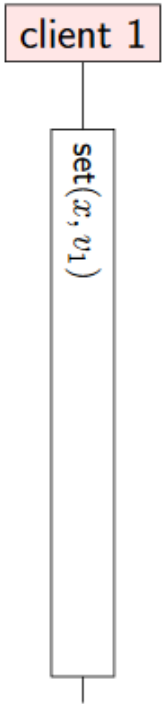


Not linearizable, despite quorum reads/writes

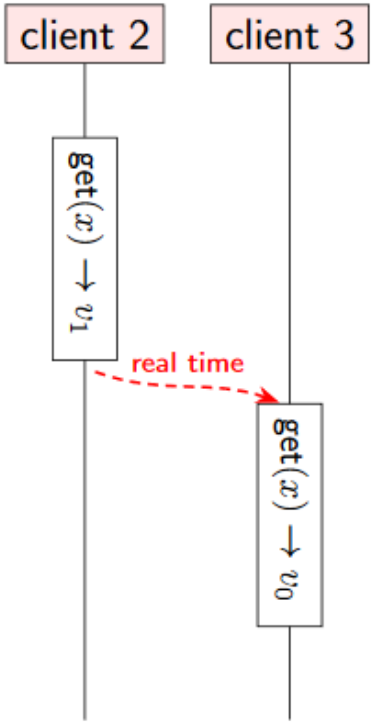
- Linearizability is not only a relationship between a `set` and a `get`, but also among multiple `get` operations.
- Example: replica A gets the update value to x quickly, but replicas B and C are delayed.



Not linearizable, despite quorum reads/writes



- Client 2's operation finishes before client 3's operation starts.
- Linearizability therefore requires client 3's operation to observe a state no older than client 2's operation.
- This example violates linearizability because v_0 is older than v_1.

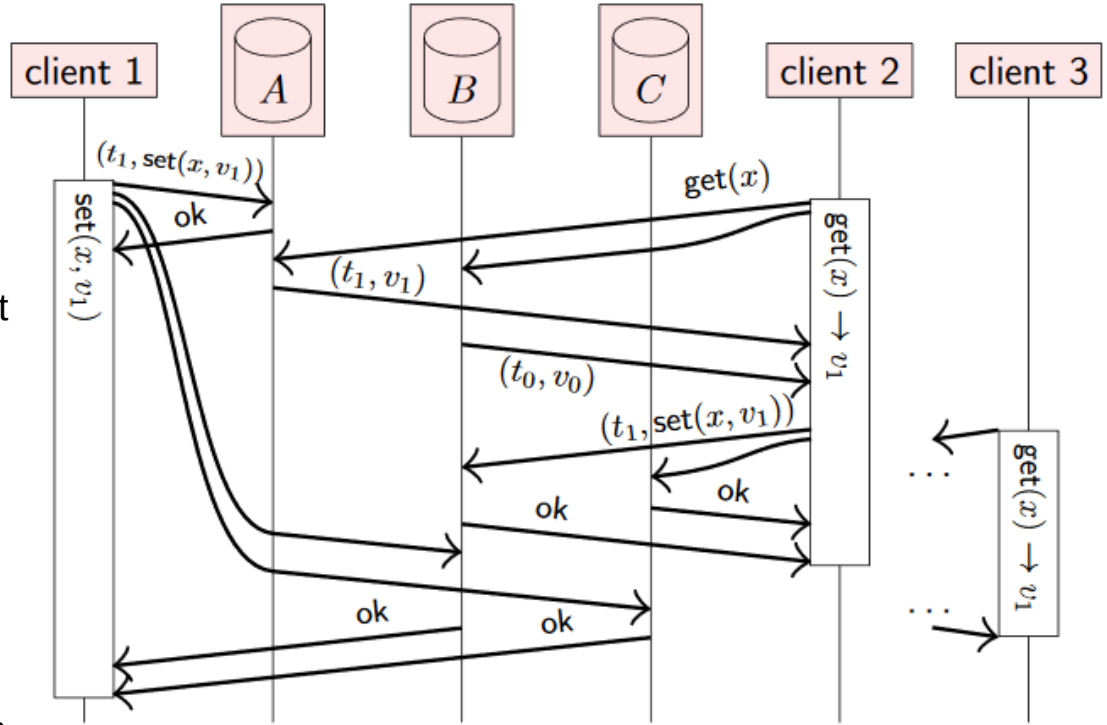


Making quorum reads/writes linearizable

- Can make quorum operations linearizable
  [ABD algorithm, Attiya et al. 1995]

- As before, send the updates to all replicas, and wait for acknowledgement from a quorum of replicas.

- For a read, a client must first send the request to all replicas and wait for responses from a quorum.
  - If some include a more recent value then the client must write back the most recent value to all stale replicas (like in read-repair).
  - The get operation finishes only after the most recent value is stored on a quorum of replicas.



Linearizability for different types of operations

- To ensure linearizability of `get` (quorum read) and `set` (**blind write** to quorum):
  - When an operation finishes, the value read/written is stored on a quorum of replicas.
  - Every subsequent quorum operation will see that value.
  - Multiple concurrent writes may overwrite each other.

- What about an atomic **compare-and-swap** operation?
  - $CAS(x, oldValue, newValue)$ sets x to $newValue$ iff current value of x is $oldValue$.
  - Can we implement **linearizable** `compare-and-swap` in a distributed system?

  - Not, with the ABD algorithm.
  - But, **yes** with consensus (total order broadcast to the rescue again!).

Linearizable compare-and-swap (CAS)

- **Broadcast every** operation we want to perform.
- **Execute** the operation **when** it is **delivered**.
- Like in state-machine replication, this ensures that an operation has the same effect and outcome on every replica.

on request to perform get(x) **do**
  total order broadcast (get, x) and wait for delivery
end on

on request to perform CAS(x, old, new) **do**
  total order broadcast (CAS, x, old, new) and wait for delivery
end on

on delivering (get, x) by total order broadcast **do**
  return $localState[x]$ as result of operation get(x)
end on

on delivering (CAS, x, old, new) by total order broadcast **do**
  $success$:= false
  if $localState[x] = old$ **then**
    $localState[x] := new$; $success$:= true
  end if
  return $success$ as result of operation CAS(x, old, new)
end on

Eventual consistency

Eventual consistency

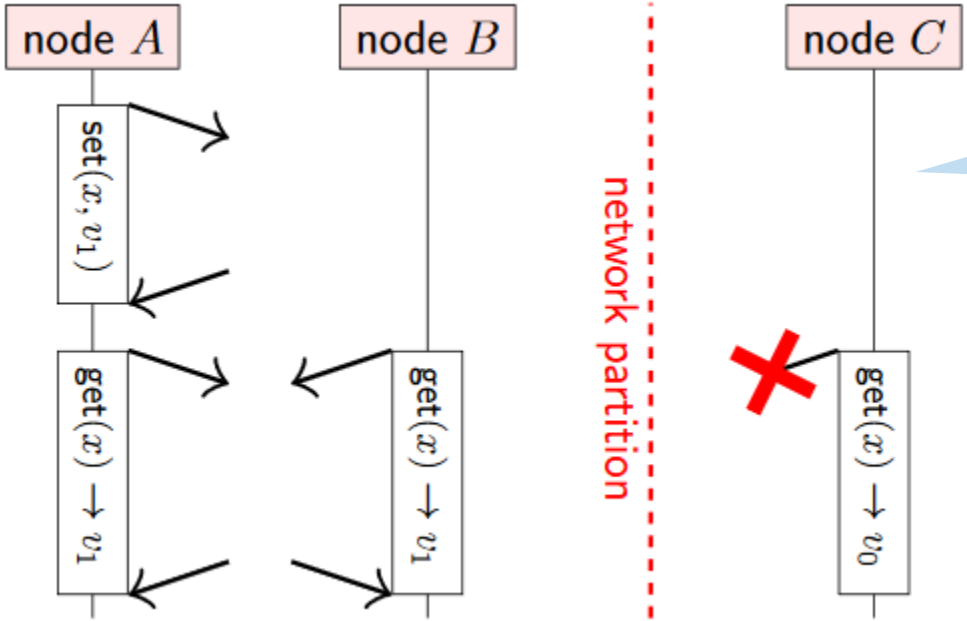
- Linearizability advantages:
  - Makes a distributed system behave as if it were non-distributed.
  - Simple for applications to use.

- Downsides:
  - **Performance** cost: lots of messages and waiting for responses
  - **Scalability** limits: leader can be a bottleneck.
  - **Availability** problems: if you cannot contact a quorum of nodes, you cannot process any operation.

- **Eventual consistency:** a weaker model than linearizability. Different trade-off choices.

The CAP theorem

- A system can be either strongly **Consistent** (linearizable), or **Available** in the presence of a network **Partition** [Gilbert and Lynch, 2002].



C must either wait indefinitely for the network to recover, or return a potentially stale value.

A system can be both linearizable and available as long as there is no network partition [Kleppmann, 2015].

Eventual consistency

- **Optimistic replication:** Replicas process operations based only on their local state.

- **Eventual consistency:** If there are no more updates, **eventually** all replicas will be in the same state [Vogels, 2009] -- no guarantees how long it may take.

- **Strong eventual consistency** [Shapiro et al., 2011].
  - **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
  - **Convergence:** any two replicas that have processed the same set of updates are in the same state (even if updates where processed in a different order).

- Properties:
  - Does not require waiting for network communication
  - Causal broadcast (or weaker) can disseminate updates
  - Concurrent updates → **conflicts** need to be resolved.

Causal consistency

- Many examples where **causality is important.**

- 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.

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

- For example, snapshot isolation provides causal consistency:
  when you read from the database, and you see some piece of data, then you must also be able to see any data that causally preceded it.

Causality vs. linearizability

- The difference between a total order and a partial order is reflected in the two consistency models:
- **Linearizability**: total order of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic. For any two operations we can always say which one happened first.
- **Causality**: we say that two operations are concurrent if neither happened before the other – i.e., they are incomparable if they are concurrent. This means causality defines a partial but not total order.
- **Linearizability implies causality, but comes at a cost.**
- 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 partitions.
- Many systems that appear to require linearizability in fact only really require causal consistency, so researchers are actively working on that topic.

Minimum system model requirements

Problem	Must wait for communication	Requires synchrony
Atomic commit	All participating nodes	Partially synchronous
Consensus, total order broadcast, linearizable CAS	Quorum	Partially synchronous
Linearizable get/set	Quorum	Asynchronous
Eventual consistency, causal broadcast, FIFO broadcast	Local replica only	Asynchronous

Collaboration and conflict resolution

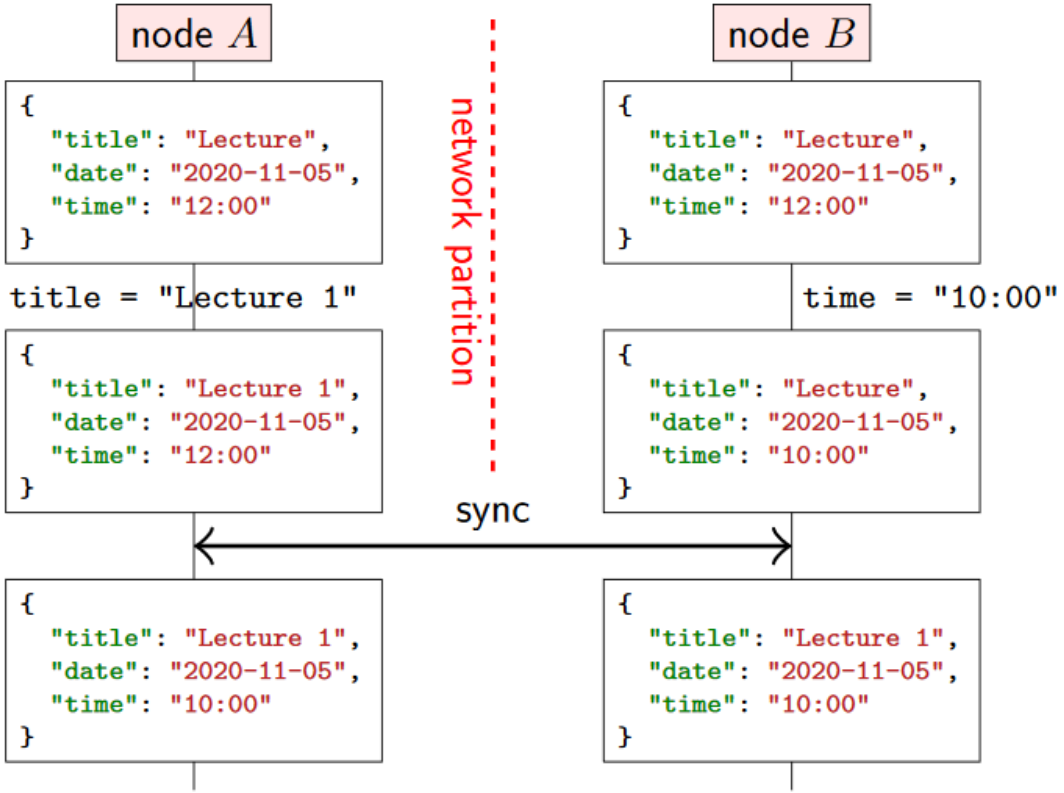
Collaboration and conflict resolution

- Nowadays, we use a lot of **collaboration software**
  - Examples: calendar sync, Google docs, etc.
  - Several users/devices working on a shared file / documents
  - Each user device has local replica of the data
  - Update local replica anytime (even while offline), sync when network available
  - **Challenge:** how to reconcile concurrent updates?

- **Families of algorithms:**
  - Conflict-free Replicated Data Types (**CRDTs**)
    - Operation-based
    - State-based
  - Operational Transformations (**OT**)

Conflicts due to concurrent updates

- Example: two nodes initially start with the same calendar entry.
  - Node A changes the title from “Lecture” to “Lecture 1”
  - Node B changes the time from 12:00 to 10:00.
  - The updates happen while A and B are temporarily unable to communicate.
  - Once the connection is restored, the final calendar entry reflects both the change in the title and to the time.



Operation-based CRDT: map

- Conflict-free Replicated Data Types (CRDTs) are a family of algorithms that perform such conflict resolution [Shapiro et al., 2011].

- Replicated object that an application accesses through the object-oriented interface of an abstract datatype: set, list, map, tree, etc.

- Example – map from keys to values
  - Local state consists of the set of values (timestamp, key, value) triples.
  - On read, return the local value
  - On write, create a globally unique timestamp and do reliable broadcast of the triple.
  - Resolve conflicts using the last-writer-wins (LWW) approach.

on initialisation **do**
  $values := \{\}$
end on

on request to read value for key k **do**
  if $\exists t, v.\ (t, k, v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
  $t := \text{newTimestamp}()$ ▷ globally unique, e.g. Lamport timestamp
  broadcast (set, t, k, v) by reliable broadcast (including to self)
end on

on delivering (set, t, k, v) by reliable broadcast **do**
  $previous := \{(t', k', v') \in values \mid k' = k\}$
  if $previous = \{\} \ \vee\ \forall (t', k', v') \in previous.\ t' < t$ **then**
    $values := (values \setminus previous) \cup \{(t, k, v)\}$
  end if
end on

Operation-based CRDTs

- Reliable broadcast may deliver updates in any order:
  - `broadcast`(set, t_1, “title”, “Lecture 1”)
  - `broadcast`(set, t_2, “time”, “10:00)

- Recall **strong eventual consistency:**
  - **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica
  - **Convergence:** any two replicas that have processed the same set of updates are in the same state

- CRDT algorithm implements this:
  - Reliable broadcast ensures every operation is eventually delivered to even (non-crashed) replica
  - Applying an operation is **commutative:** order of delivery does not matter

State-based map CRDT

An alternative CRDT algorithm for the same map datatype.

- The definition of values and the function for reading the value of a key is as before.
- The update: instead of broadcasting each operation, we directly apply the value locally and then broadcast the whole of values.
- On delivery of the message to another replica, we merge together the two replica states using a merge function.
  - The function compares the timestamp of entries with the same key.

The operator $\sqcup$ merges two states s_1 and s_2 as follows:

$$s_1 \sqcup s_2 = \{(t,k,v) \in (s_1 \cup s_2) \mid \nexists (t',k',v') \in (s_1 \cup s_2).\ k' = k \wedge t' > t\}$$

on initialisation **do**
  $values := \{\}$
end on

on request to read value for key k **do**
  if $\exists t, v.\ (t,k,v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
  $t := \text{newTimestamp}()$ ▷ globally unique, e.g. Lamport timestamp
  $values := \{(t',k',v') \in values \mid k' \neq k\} \cup \{(t,k,v)\}$
  broadcast $values$ by best-effort broadcast
end on

on delivering V by best-effort broadcast **do**
  $values := values \sqcup V$
end on

State-based CRDTs

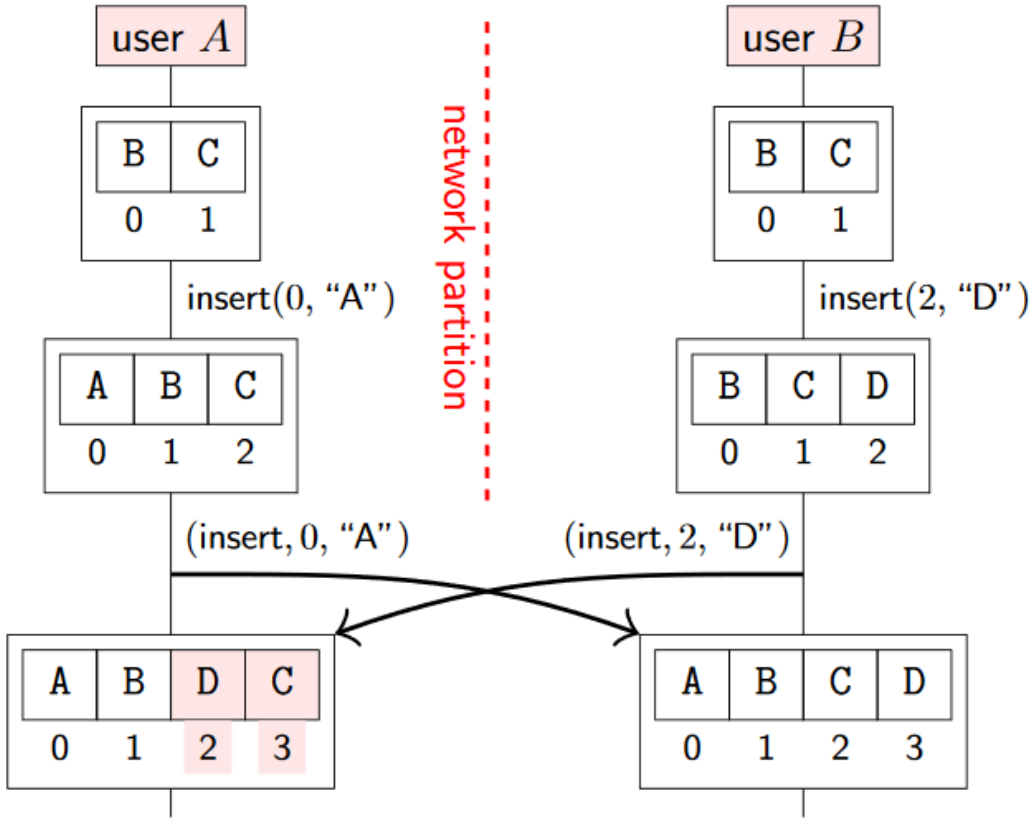
- Merge operation $\sqcup$ must satisfy: $\forall\ s_1, s_2, s_3, \ldots$
  - **Commutative:** $s_1 \sqcup s_2 = s_2 \sqcup s_1$
  - **Associative:** $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$
  - **Idempotent:** $s_1 \sqcup s_1 = s_1$

- State-based versus operation-based:
  - Operation-based CRDT typically has smaller messages
  - State-based CRDT can tolerate message loss/duplication

- Not necessarily uses broadcast
  - Can also merge concurrent updates to replicas e.g., in quorum replication, etc.

Collaborative text editing: the problem

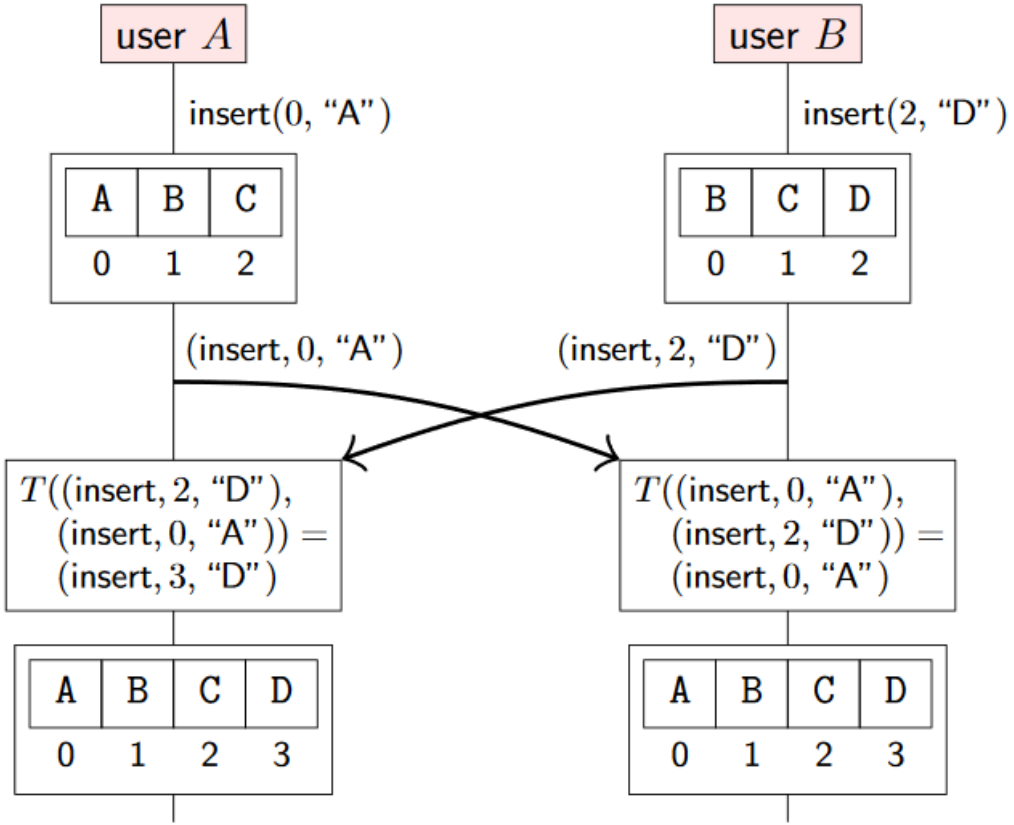
- Set-up: think of a collaboratively editable text document as a list of characters.
- When several users may concurrently update a text document: if not careful, their operations may clash on more complex data structures like lists.
- Instead of reaching to a state ABCD, user A gets ABDC.



Operational transformation

Operational transformation – class of algorithms specifically targeting such conflicts

- A node keeps track of the history of operations it has performed.
- When it receives another node's operation that is concurrent to its own operations, it **transforms** the incoming operation relative to its own concurrent operations.
- $T(op_1, op_2)$ takes two operations: incoming and local and returns a transformed operation op_1', such that applying op_1' on the local state has the same effect as originally intended.

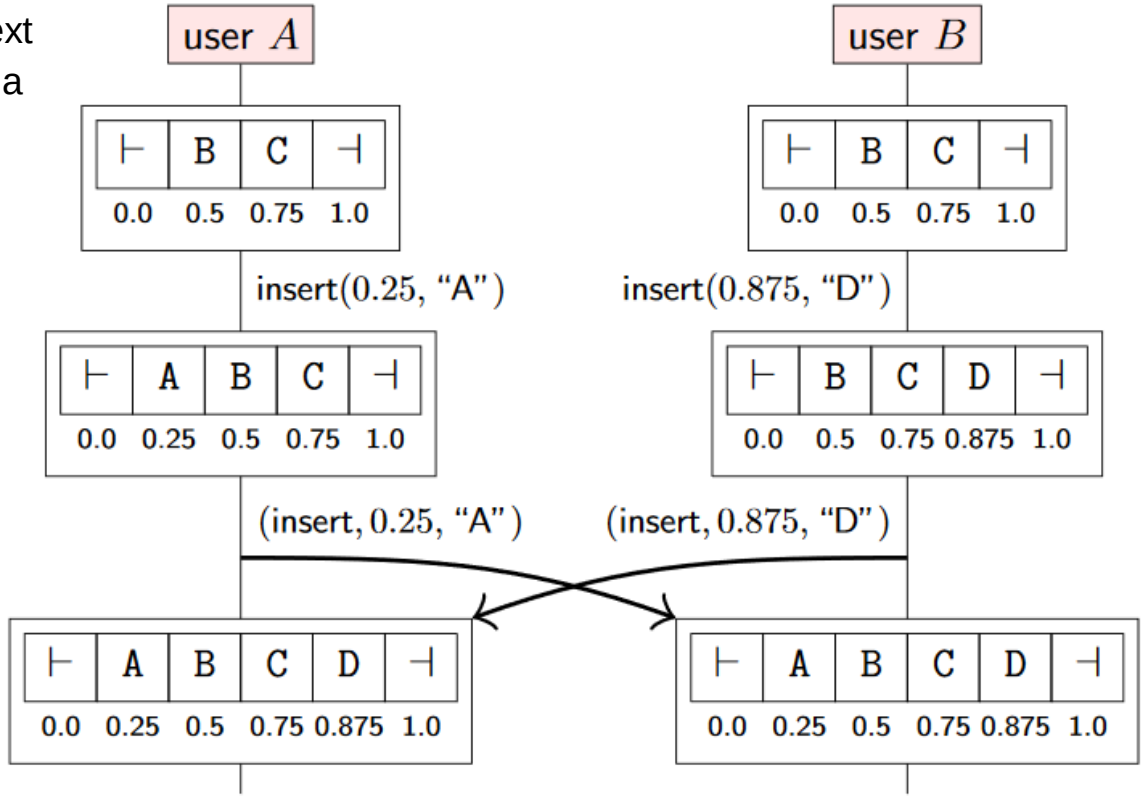


Text editing CRDT

Rather than identifying the positions in text using indexes, label each character with a unique identifier (e.g., a rational number between 0 and 1).

- 0 represents the beginning of a text, 1 the end.
- When inserting a new character between two existing ones, we assign the character with a position number between i and j, i. e., $^{i+j}/_{2}$.

- This way, conflict resolution becomes much simpler.



Operation-based text CRDT (1/2)

function ElementAt($chars, index$)
  $min =$ the unique triple $(p, n, v) \in chars$ such that
    $\nexists (p', n', v') \in chars.\ p' < p \vee (p' = p \wedge n' < n)\}$
  if $index = 0$ **then return** min
  else return ElementAt($chars \setminus \{min\}, index - 1$)
end function

on initialisation **do**
  $chars := \{(0, \text{null}, \vdash), (1, \text{null}, \dashv)\}$
end on

on request to read character at index $index$ **do**
  let $(p, n, v) :=$ ElementAt($chars, index + 1$); **return** v
end on

on request to insert character v at index $index$ at node $nodeId$ **do**
  let $(p_1, n_1, v_1) :=$ ElementAt($chars, index$)
  let $(p_2, n_2, v_2) :=$ ElementAt($chars, index + 1$)
  broadcast $(\text{insert}, (p_1 + p_2)/2, nodeId, v)$ by causal broadcast
end on

Operation-based text CRDT (2/2)

on delivering (insert, p, n, v) by causal broadcast **do**
    $chars := chars \cup \{(p, n, v)\}$
end on

on request to delete character at index $index$ **do**
    let $(p, n, v) := \textsc{ElementAt}(chars, index + 1)$
    broadcast (delete, p, n) by causal broadcast
end on

on delivering (delete, p, n) by causal broadcast **do**
    $chars := \{(p', n', v') \in chars \mid \neg(p' = p \wedge n' = n)\}$
end on

- Use causal broadcast so that insertion of a character is delivered before its deletion
- Insertion and deletion of different characters commute

Overview of systems

Taxonomy of existing KVS systems

System	Scale	Memory Model	Per-Key Consistency	Multi-key Consistency
Masstree	M	SM	Linearizable	None
Bw-tree	M	SM	Linearizable	None
PALM	M	SM	Linearizable	None
MICA	M	SM	Linearizable	None
Redis	S	N/A	Linearizable	Serializable
COPS, Bolt-on	D	MP	Causal	Causal
Bayou	D	MP	Eventual, Monotonic Reads/Writes, Read Your Writes	Eventual
Dynamo	D	MP	Linearizable, Eventual	None
Cassandra	D	MP	Linearizable, Eventual	None
PNUTS	D	MP	Linearizable Writes, Monotonic Reads	None
CouchDB	D	MP	Eventual	None
Voldemort	D	MP	Linearizable, Eventual	None
HBase	D	MP	Linearizable	None
Riak	D	MP	Eventual	None
DocumentDB	D	MP	Eventual, Session, Bounded Staleness, Linearizability	None
Memcached	M & D	SM & MP	Linearizable	None
MongoDB	M & D	SM & MP	Linearizable	None
H-Store	M & D	MP	Linearizable	Serializable
ScyllaDB	M & D	MP	Linearizable, Eventual	None
Anna	**M & D**	**MP**	**Eventual, Causal, Item Cut, Writes Follow Reads Monotonic Reads/Writes, Read Your Writes, PRAM**	**Read Committed, Read Uncommitted**

References

The material covered in this class is mainly based on:

- The book *"Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems"* by Martin Kleppmann (Chapter 9) (link)
- Slides from "*Distributed Systems"* course from University of Cambridge (link)

Papers:

- Consistency, Availability, and Convergence – Mahajan et al. (TR from UT Austin, 2011)
- Highly Available Transactions: Virtues and Limitations – Bailis et al. (VLDB'14)
- Don't settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS – Lloyd et al. (SOSP'11)
- A comprehensive study of Convergent and Commutative Replicated Data Types – Shapiro et al. (2011)
- Anna: a KVS for Any Scale – Wu et al. (ICDE'18)