Lecture 8

Concurrency control in applications

Collaboration and conflict resolution

Nowadays we use a lot of **collaboration software**:

- **Examples:** calendar sync (last lecture), Google Docs, . . .

Collaboration and conflict resolution

Nowadays we use a lot of **collaboration software**:

- **Examples:** calendar sync (last lecture), Google Docs, . . .
- Several users/devices working on a shared file/document
- Each user device has local replica of the data

Collaboration and conflict resolution

Nowadays we use a lot of **collaboration software**:

- **Examples:** calendar sync (last lecture), Google Docs, ...
- Several users/devices working on a shared file/document
- Each user device has local replica of the data
- Update local replica anytime (even while offline), sync with others when network available

Collaboration and conflict resolution

Nowadays we use a lot of **collaboration software**:

- **Examples:** calendar sync (last lecture), Google Docs, ...
- Several users/devices working on a shared file/document
- Each user device has local replica of the data
- Update local replica anytime (even while offline), sync with others when network available
- **Challenge:** how to reconcile concurrent updates?

Collaboration and conflict resolution

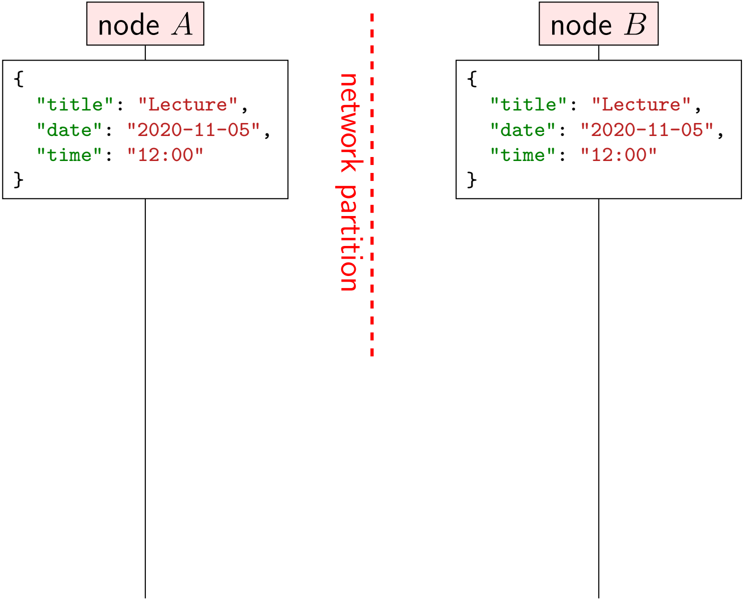
Nowadays we use a lot of **collaboration software**:

- **Examples:** calendar sync (last lecture), Google Docs, ...
- Several users/devices working on a shared file/document
- Each user device has local replica of the data
- Update local replica anytime (even while offline), sync with others when network available
- **Challenge:** how to reconcile concurrent updates?

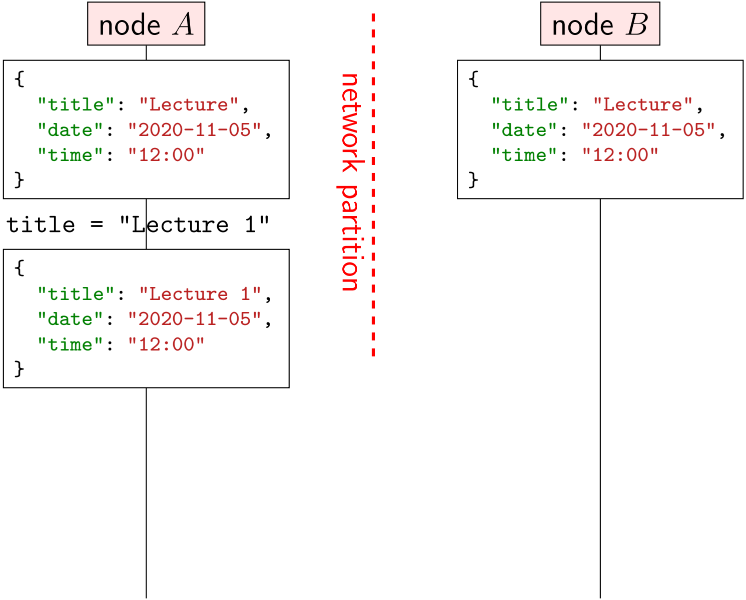
Families of **algorithms**:

- Conflict-free Replicated Data Types (**CRDTs**)
  - Operation-based
  - State-based
- Operational Transformation (**OT**)

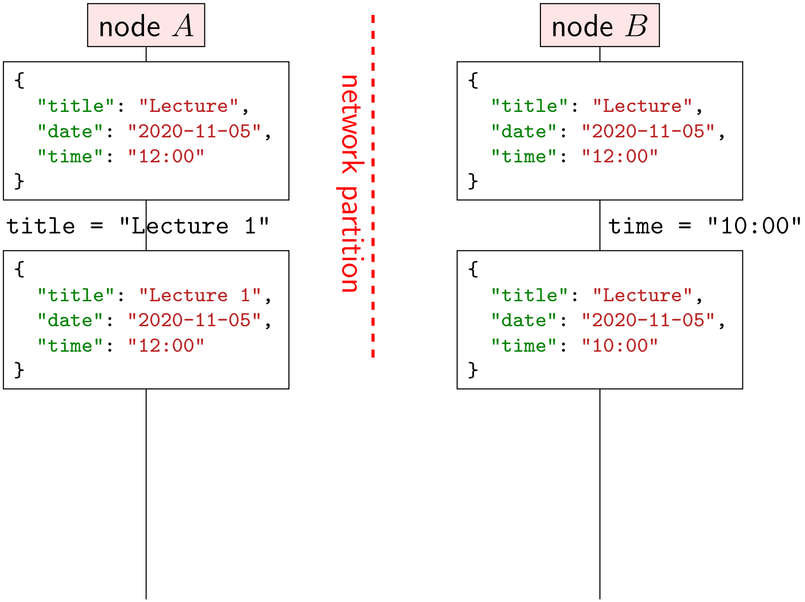
Conflicts due to concurrent updates



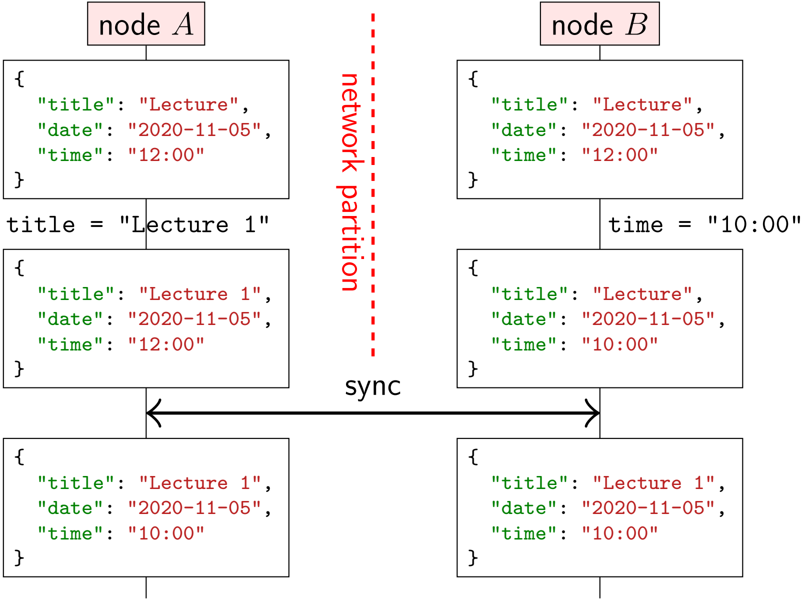
Conflicts due to concurrent updates



Conflicts due to concurrent updates



Conflicts due to concurrent updates



Operation-based map CRDT

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**
  $t := \mathrm{newTimestamp}()$ ▷ 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 $(\text{set}, t_1, \text{“title”}, \text{“Lecture 1”})$
- broadcast $(\text{set}, t_2, \text{“time”}, \text{“10:00”})$

Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- broadcast $(\text{set}, t_1, \text{“title”}, \text{“Lecture 1”})$
- broadcast $(\text{set}, t_2, \text{“time”}, \text{“10:00”})$

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

Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- broadcast $\big($set, t_1, "title", "Lecture 1"$\big)$
- broadcast $\big($set, t_2, "time", "10:00"$\big)$

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 every (non-crashed) replica

Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- broadcast $(\text{set}, t_1, \text{“title”}, \text{“Lecture 1”})$
- broadcast $(\text{set}, t_2, \text{“time”}, \text{“10:00”})$

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 every (non-crashed) replica
- Applying an operation is **commutative**: order of delivery doesn't matter

State-based map CRDT

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**
  $\mathit{values} := \{\}$
end on

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

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

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

State-based CRDTs

Merge operator $\sqcup$ 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 CRDTs

Merge operator $\sqcup$ 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:

- Op-based CRDT typically has smaller messages
- State-based CRDT can tolerate message loss/duplication

State-based CRDTs

Merge operator $\sqcup$ must satisfy: $\forall s_1, s_2, s_3 \ldots$

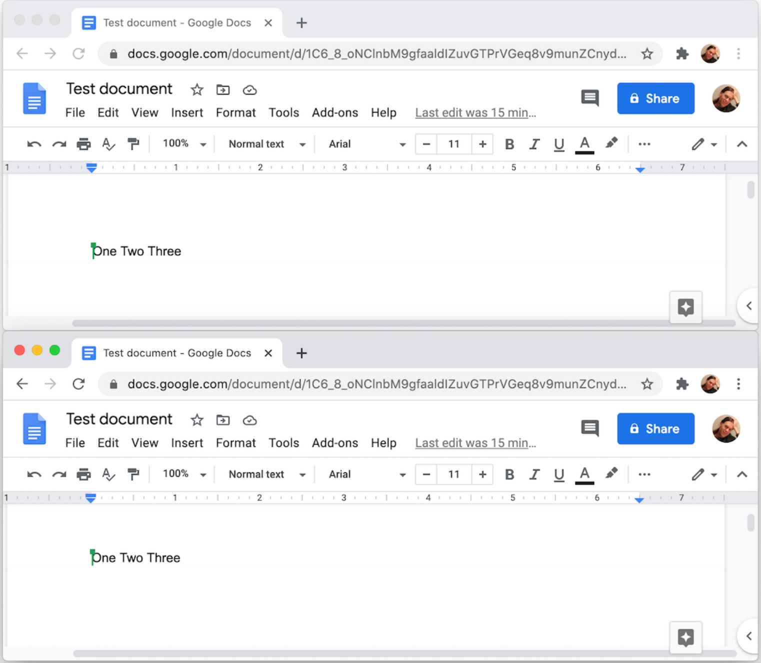
- **Commutative**: $s_1 \sqcup s_2 = s_2 \sqcup s_1$.
- **Associative**: $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$.
- **Idempotent**: $s_1 \sqcup s_1 = s_1$.

State-based versus operation-based:

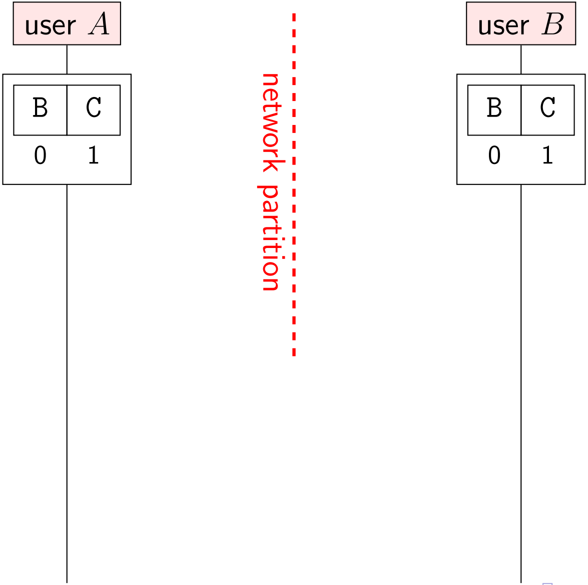
- Op-based CRDT typically has smaller messages
- State-based CRDT can tolerate message loss/duplication

Not necessarily uses broadcast:

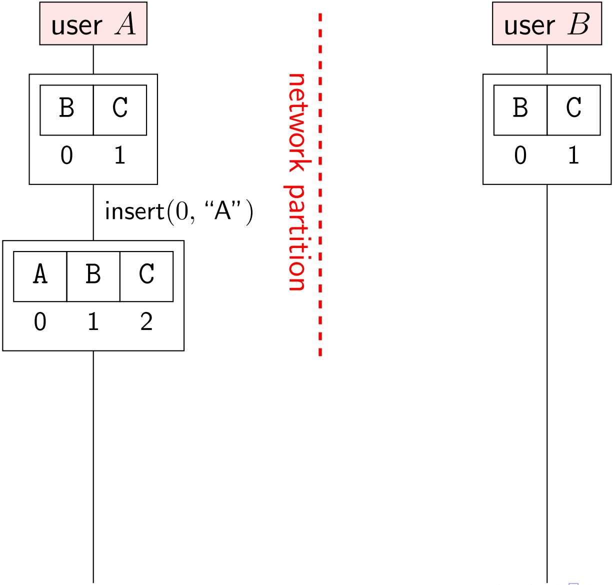
- Can also merge concurrent updates to replicas e.g. in quorum replication, anti-entropy, ...


Test document - Google Docs
docs.google.com/document/d/1C6_8_oNClnbM9gfaaldIZuvGTPrVGeq8v9munZCnyd...
Test document
File Edit View Insert Format Tools Add-ons Help
Last edit was 15 min...
Share
100%
Normal text
Arial
11
One Two Three
Test document - Google Docs
docs.google.com/document/d/1C6_8_oNClnbM9gfaaldIZuvGTPrVGeq8v9munZCnyd...
Test document
File Edit View Insert Format Tools Add-ons Help
Last edit was 15 min...
Share
100%
Normal text
Arial
11
One Two Three

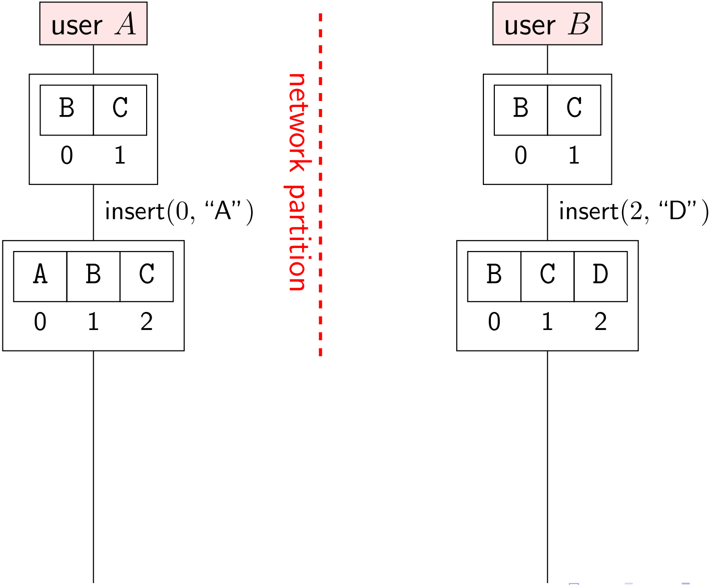
Collaborative text editing: the problem



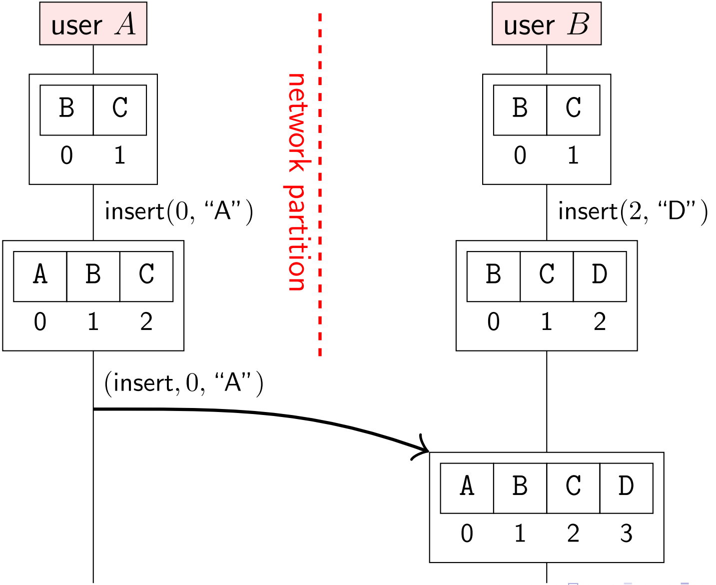
Collaborative text editing: the problem



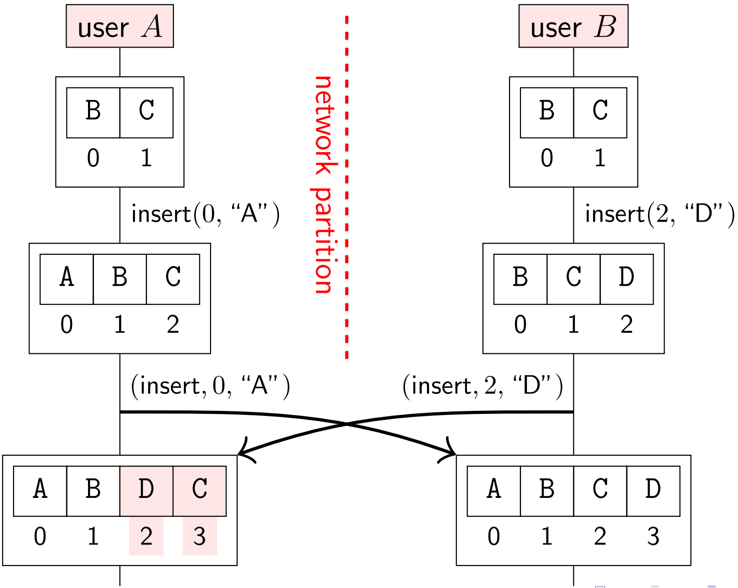
Collaborative text editing: the problem



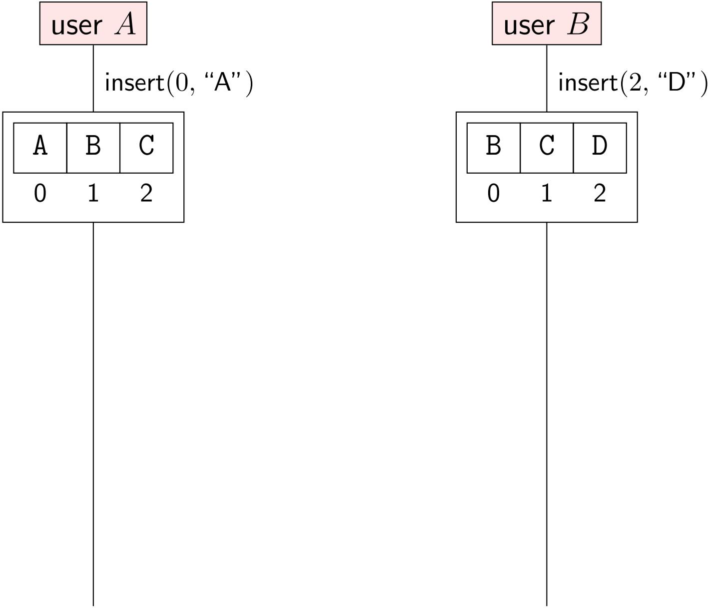
Collaborative text editing: the problem



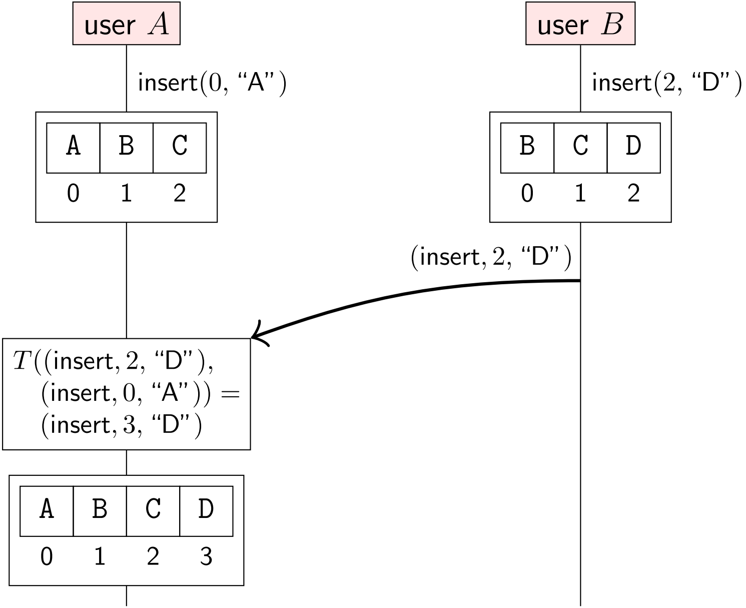
Collaborative text editing: the problem



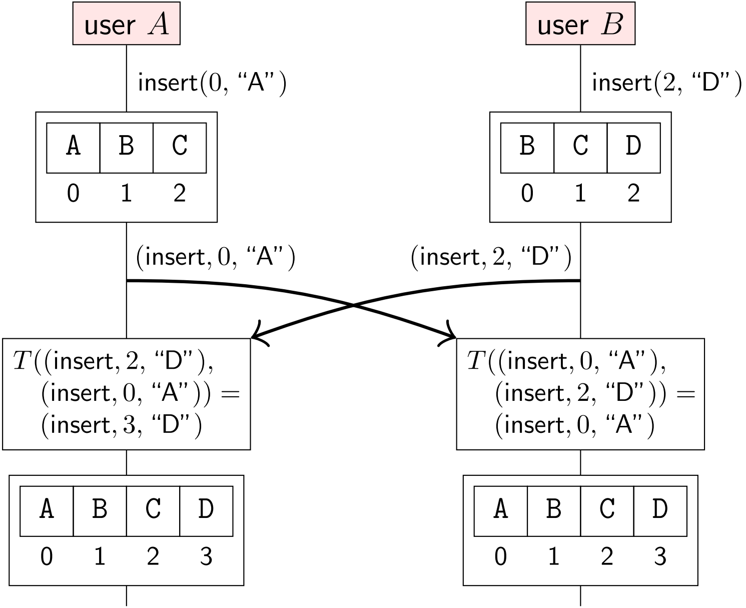
Operational transformation



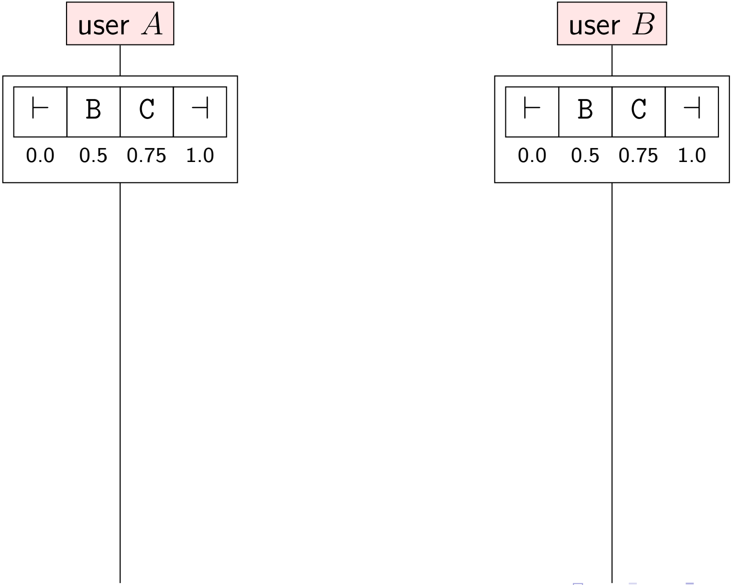
Operational transformation



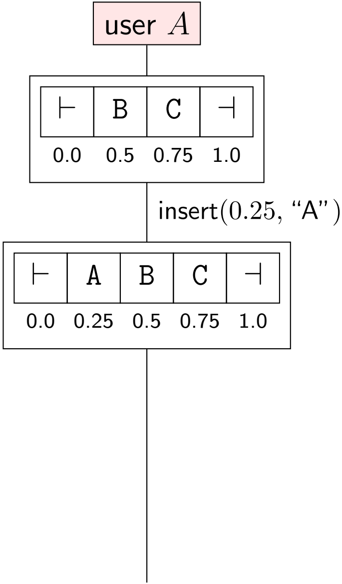
Operational transformation



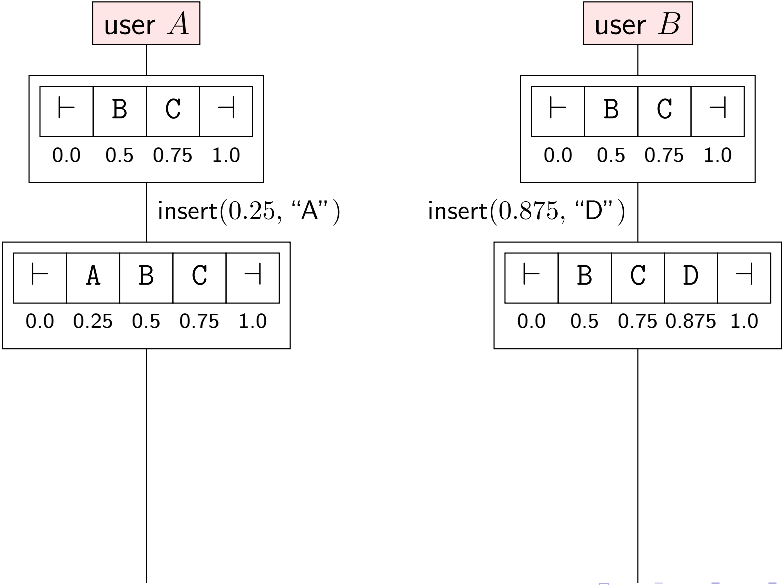
Text editing CRDT



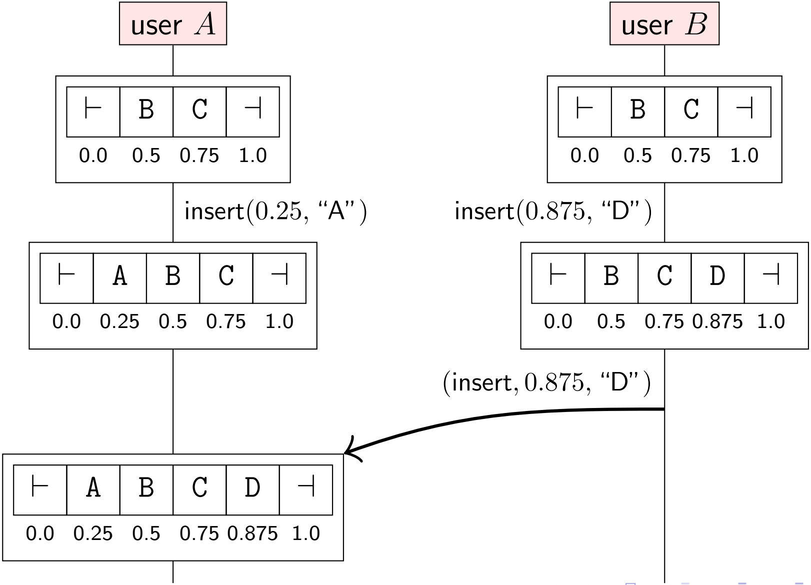
Text editing CRDT



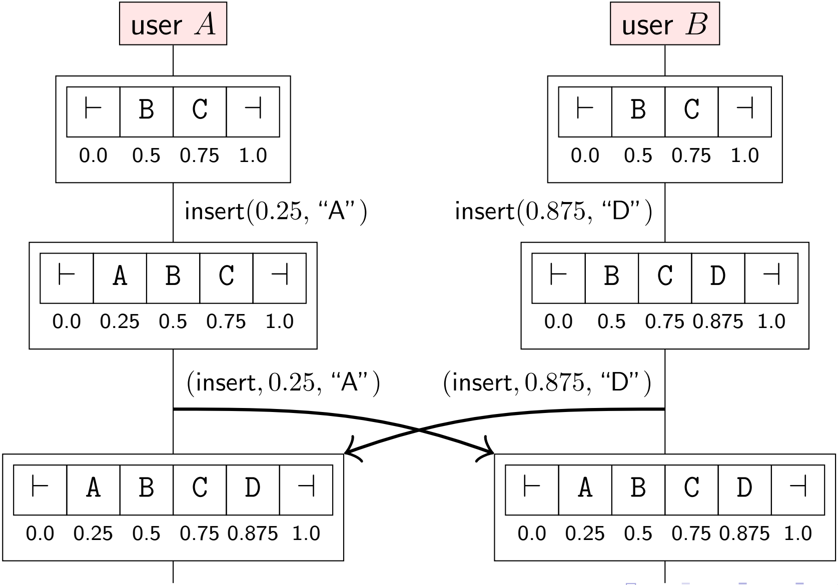
Text editing CRDT



Text editing CRDT



Text editing CRDT



Operation-based text CRDT (1/2)

function $\textsc{ElementAt}(chars, index)$
  $min =$ 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 $\textsc{ElementAt}(chars \setminus \{min\}, index - 1)$
end function

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

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

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

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Consistency properties:

- **Serializable** transaction isolation
- **Linearizable** reads and writes

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Consistency properties:

- **Serializable** transaction isolation
- **Linearizable** reads and writes
- Many **shards**, each holding a subset of the data; atomic commit of transactions across shards

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Consistency properties:

- **Serializable** transaction isolation
- **Linearizable** reads and writes
- Many **shards**, each holding a subset of the data; atomic commit of transactions across shards

Many standard techniques:

- State machine replication (Paxos) within a shard
- Two-phase locking for serializability
- Two-phase commit for cross-shard atomicity

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Consistency properties:

- **Serializable** transaction isolation
- **Linearizable** reads and writes
- Many **shards**, each holding a subset of the data; atomic commit of transactions across shards

Many standard techniques:

- State machine replication (Paxos) within a shard
- Two-phase locking for serializability
- Two-phase commit for cross-shard atomicity

The interesting bit: read-only transactions require **no locks**!

Consistent snapshots

A read-only transaction observes a **consistent snapshot**:
If $T_1 \rightarrow T_2$ (e.g. T_2 reads data written by T_1)...

- Snapshot reflecting writes by T_2 also reflects writes by T_1
- Snapshot that does not reflect writes by T_1 does not reflect writes by T_2 either

Consistent snapshots

A read-only transaction observes a **consistent snapshot**:
If $T_1 \rightarrow T_2$ (e.g. T_2 reads data written by T_1)...

- Snapshot reflecting writes by T_2 also reflects writes by T_1
- Snapshot that does not reflect writes by T_1 does not reflect writes by T_2 either
- In other words, snapshot is **consistent with causality**
- Even if read-only transaction runs for a long time

Consistent snapshots

A read-only transaction observes a **consistent snapshot**:
If $T_1 \to T_2$ (e.g. T_2 reads data written by T_1)...

- Snapshot reflecting writes by T_2 also reflects writes by T_1
- Snapshot that does not reflect writes by T_1 does not reflect writes by T_2 either
- In other words, snapshot is **consistent with causality**
- Even if read-only transaction runs for a long time

Approach: **multi-version concurrency control** (MVCC)

- Each read-write transaction T_w has commit timestamp t_w
- Every value is tagged with timestamp t_w of transaction that wrote it (not overwriting previous value)

Consistent snapshots

A read-only transaction observes a **consistent snapshot**:
If $T_1 \to T_2$ (e.g. T_2 reads data written by T_1)...

- Snapshot reflecting writes by T_2 also reflects writes by T_1
- Snapshot that does not reflect writes by T_1 does not reflect writes by T_2 either
- In other words, snapshot is **consistent with causality**
- Even if read-only transaction runs for a long time

Approach: **multi-version concurrency control** (MVCC)

- Each read-write transaction T_w has commit timestamp t_w
- Every value is tagged with timestamp t_w of transaction that wrote it (not overwriting previous value)
- Read-only transaction T_r has snapshot timestamp t_r
- T_r ignores values with $t_w > t_r$; observes most recent value with $t_w \leq t_r$

Obtaining commit timestamps

Must ensure that whenever $T_1 \rightarrow T_2$ we have $t_1 < t_2$.

- Physical clocks may be **inconsistent with causality**

Obtaining commit timestamps

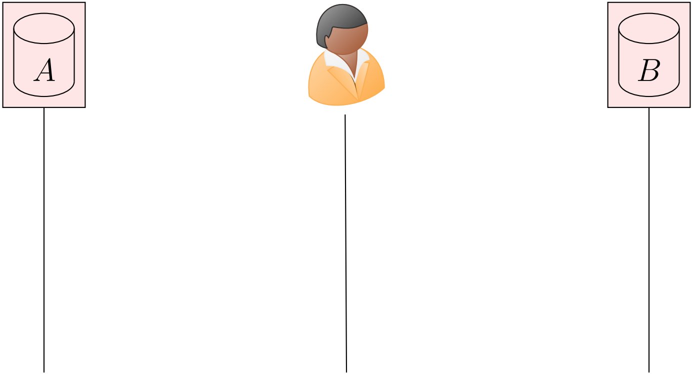
Must ensure that whenever $T_1 \rightarrow T_2$ we have $t_1 < t_2$.

- Physical clocks may be **inconsistent with causality**
- Can we use Lamport clocks instead?
- Problem: linearizability depends on **real-time order**, and logical clocks may not reflect this!

Obtaining commit timestamps

Must ensure that whenever $T_1 \rightarrow T_2$ we have $t_1 < t_2$.

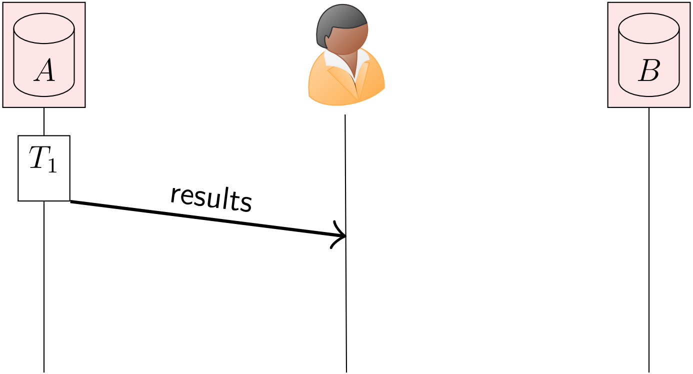
- Physical clocks may be **inconsistent with causality**
- Can we use Lamport clocks instead?
- Problem: linearizability depends on **real-time order**, and logical clocks may not reflect this!



Obtaining commit timestamps

Must ensure that whenever $T_1 \rightarrow T_2$ we have $t_1 < t_2$.

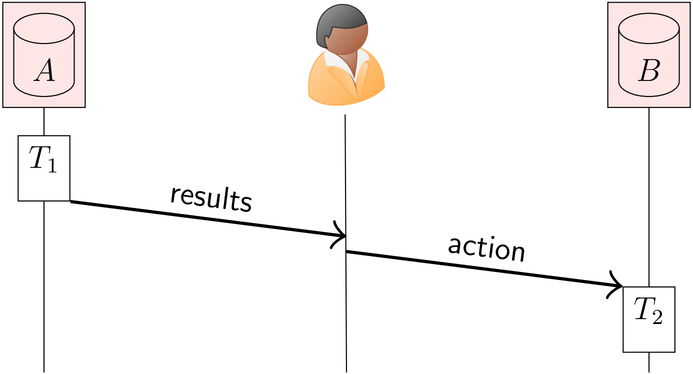
- Physical clocks may be **inconsistent with causality**
- Can we use Lamport clocks instead?
- Problem: linearizability depends on **real-time order**, and logical clocks may not reflect this!



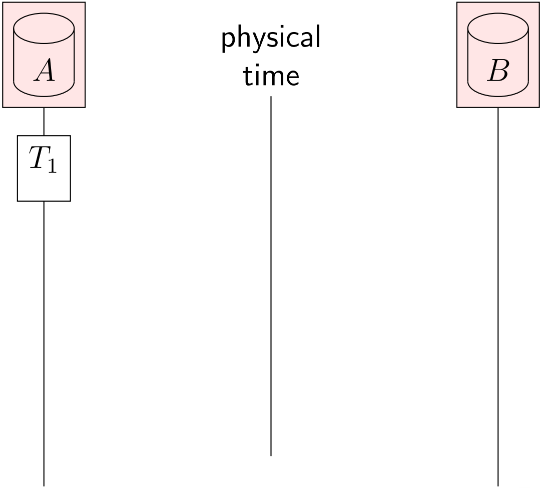
Obtaining commit timestamps

Must ensure that whenever $T_1 \rightarrow T_2$ we have $t_1 < t_2$.

- Physical clocks may be **inconsistent with causality**
- Can we use Lamport clocks instead?
- Problem: linearizability depends on **real-time order**, and logical clocks may not reflect this!

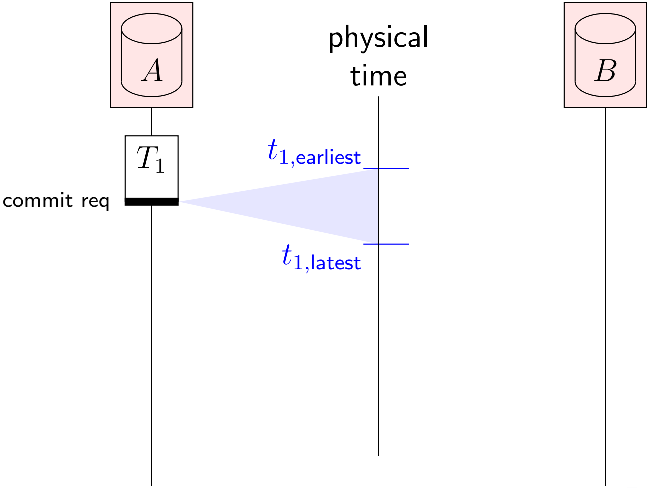


TrueTime: explicit physical clock uncertainty



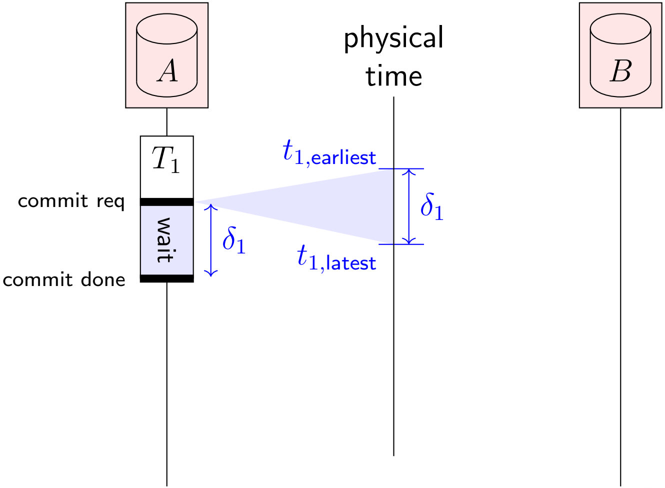
TrueTime: explicit physical clock uncertainty

Spanner's TrueTime clock returns $[t_{\text{earliest}}, t_{\text{latest}}]$.
True physical timestamp must lie within that range.



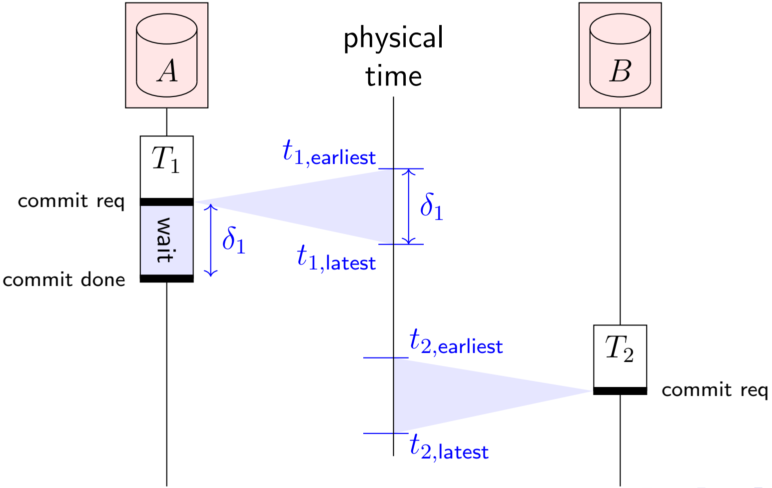
TrueTime: explicit physical clock uncertainty

Spanner's TrueTime clock returns $[t_{\text{earliest}}, t_{\text{latest}}]$.
True physical timestamp must lie within that range.
On commit, wait for uncertainty $\delta_i = t_{i,\text{latest}} - t_{i,\text{earliest}}$.



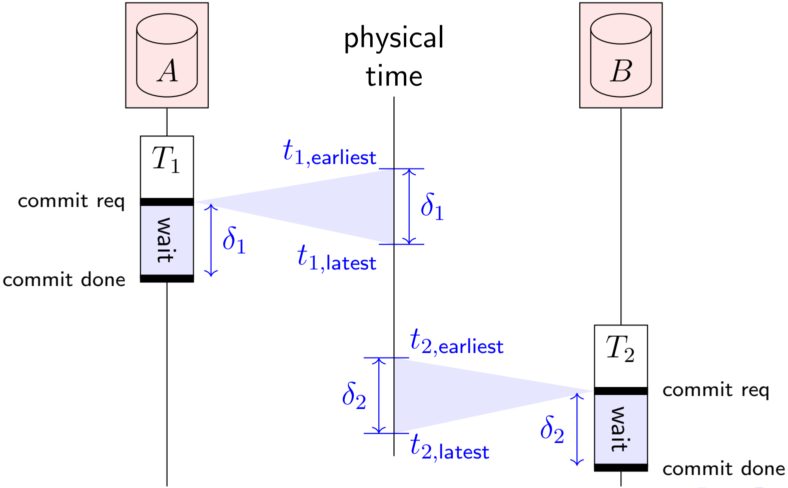
TrueTime: explicit physical clock uncertainty

Spanner's TrueTime clock returns $[t_{\text{earliest}}, t_{\text{latest}}]$.
True physical timestamp must lie within that range.
On commit, wait for uncertainty $\delta_i = t_{i,\text{latest}} - t_{i,\text{earliest}}$.



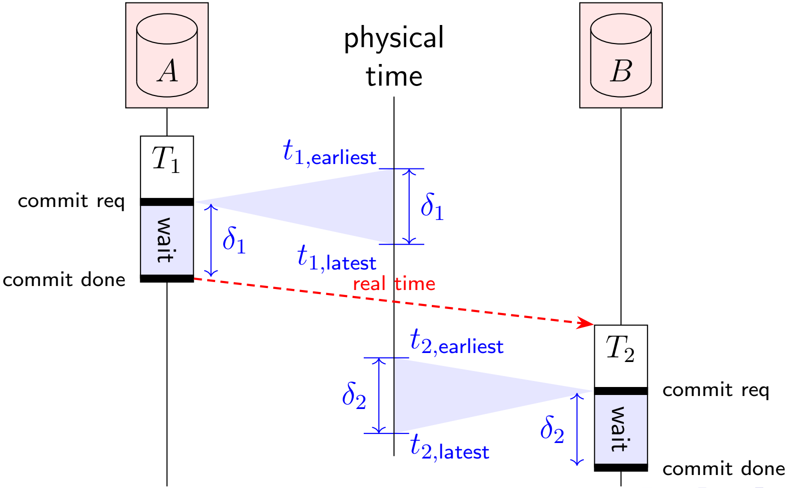
TrueTime: explicit physical clock uncertainty

Spanner's TrueTime clock returns $[t_{\text{earliest}}, t_{\text{latest}}]$.
True physical timestamp must lie within that range.
On commit, wait for uncertainty $\delta_i = t_{i,\text{latest}} - t_{i,\text{earliest}}$.



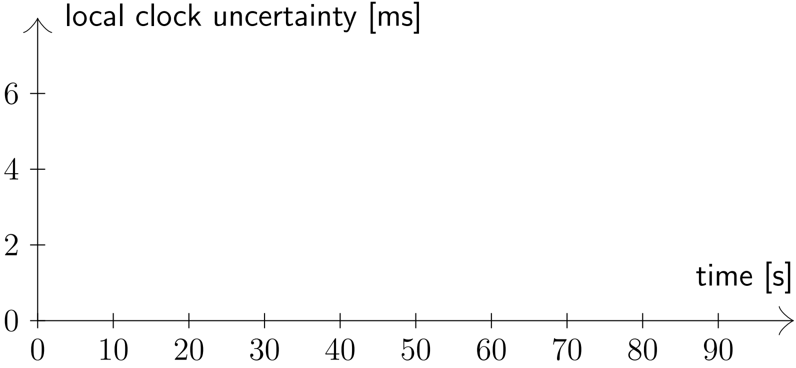
TrueTime: explicit physical clock uncertainty

Spanner's TrueTime clock returns $[t_{\text{earliest}}, t_{\text{latest}}]$.
True physical timestamp must lie within that range.
On commit, wait for uncertainty $\delta_i = t_{i,\text{latest}} - t_{i,\text{earliest}}$.



Determining clock uncertainty in TrueTime

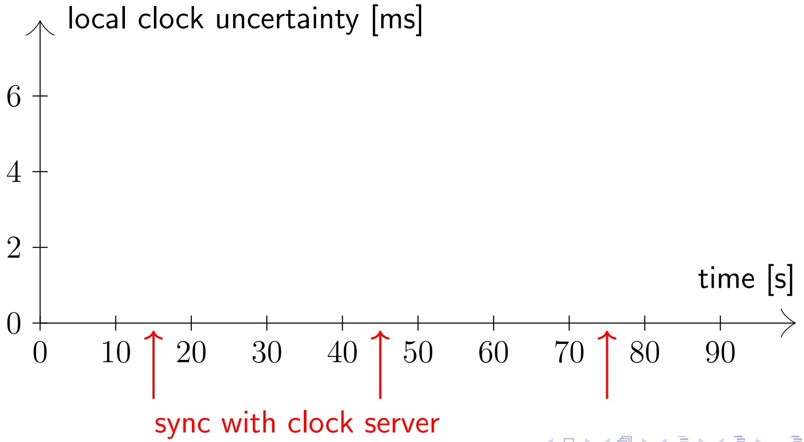
Clock servers with **atomic clock** or **GPS receiver** in each datacenter; servers report their clock uncertainty.



Determining clock uncertainty in TrueTime

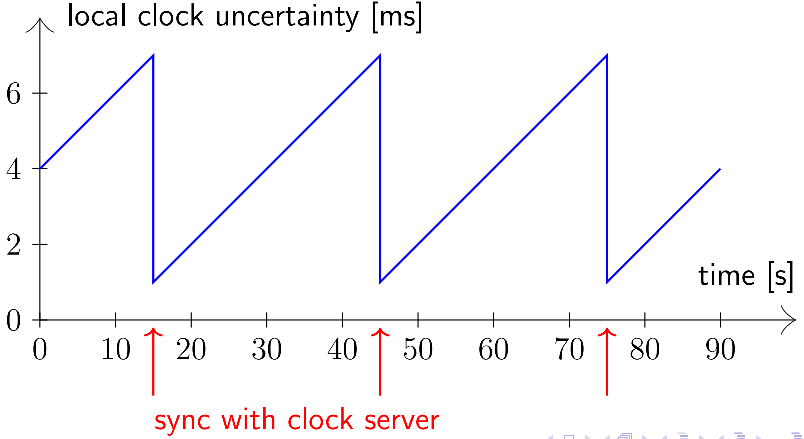
Clock servers with **atomic clock** or **GPS receiver** in each datacenter; servers report their clock uncertainty.

Each node syncs its quartz clock with a server every 30 sec.



Determining clock uncertainty in TrueTime

Clock servers with **atomic clock** or **GPS receiver** in each datacenter; servers report their clock uncertainty.
Each node syncs its quartz clock with a server every 30 sec.
Between syncs, assume worst-case drift of 200ppm.

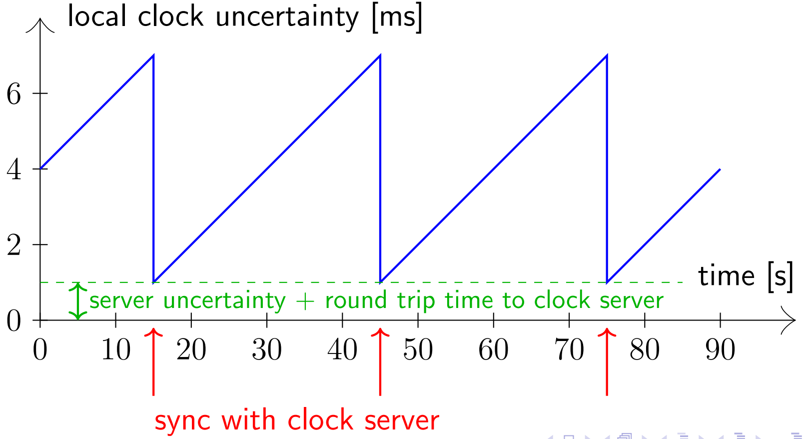


Determining clock uncertainty in TrueTime

Clock servers with **atomic clock** or **GPS receiver** in each datacenter; servers report their clock uncertainty.

Each node syncs its quartz clock with a server every 30 sec.

Between syncs, assume worst-case drift of 200ppm.



That's all, folks!

Any questions? Email mk428@cst.cam.ac.uk!

Summary:

- Distributed systems are everywhere
- You use them every day: e.g. web apps
- Key goals: availability, scalability, performance
- Key problems: concurrency, faults, unbounded latency
- Key abstractions: replication, broadcast, consensus
- No one right way, just trade-offs