CS 60002: Distributed Systems

T12: Eventual Concurrency in Applications

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Content courtesy: Lecture slides by Prof. Vijay Chidambaram, UTexas
Course Link: https://github.com/vijay03/cs380d-s21

Collaboration in Distributed Applications

- We use several collaborative software now a days
  - Google calendar
  - Docs, Office software
  - Splitwise
- Multiple devices or users work on a common file or document
- Each user/device maintains the local replica of the data
  - Users can update the local replica anytime, even when offline
  - The updates need to be synced when the device comes online
- **Challenge:** How to reconcile the current updates?

CAP Theorem (Brewer's theorem)

- Any distributed data store can support only two of the following three requirements
  - **Consistency:** Every read receives the most recent write or an error (**Sequential consistency**)
  - **Availability:** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
  - **Partition Tolerance:** The system continues to operate despite an arbitrary number of messages being dropped (or arbitrarily delayed) by the network between nodes.

CAP Theorem (Brewer's theorem)

- In the presence of network partition,
  - **Cancel the operation:** Decrease availability but ensure consistency
  - **Proceed with the operation:** Ensure availability but data may get inconsistent

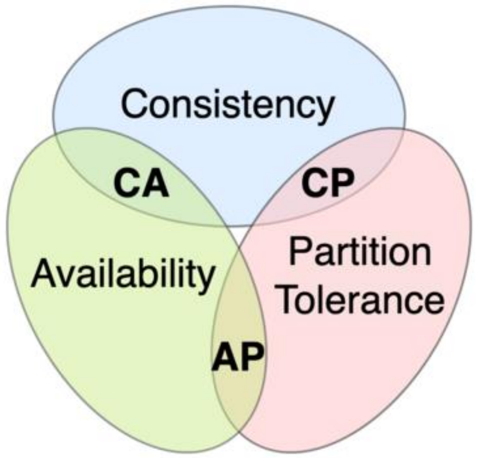


Image source: Wikipedia

- **NB:** Consistency in CAP theorem is different from Consistency in ACID transactions

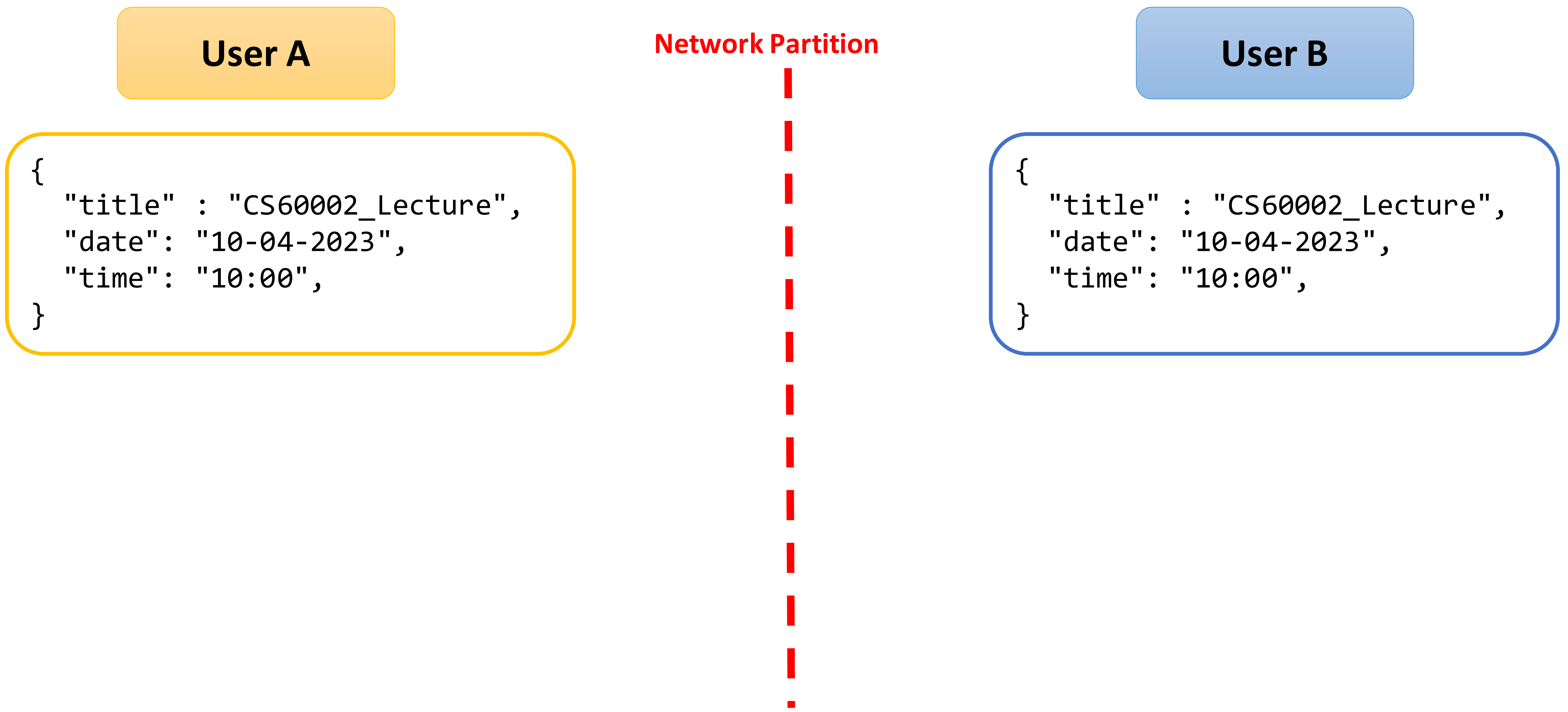
Consistency vs Reconciliation

- CAP theorem prevents achieving both consistency (sequential consistency) and availability in the presence of partition

- Consistency guarantee for practical applications: **Eventual consistency**
  - A replica may execute an operation without synchronizing a priori with other replicas
  - The operation is sent asynchronously to other replicas
  - Every replica eventually applies all updates, possibly in different orders
  - A background consensus algorithm reconciles any conflicting updates

- However, reconciliation is not always easy!

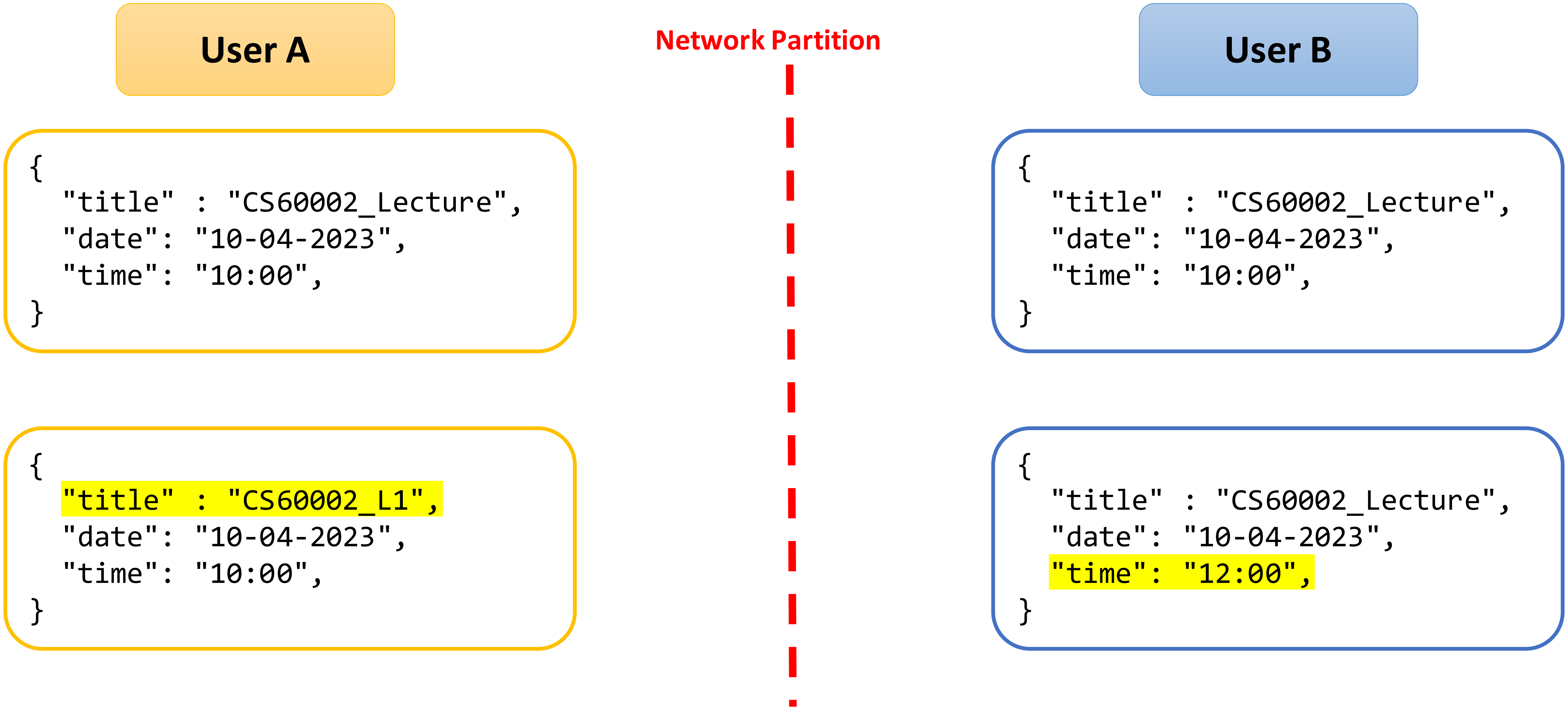
Conflicts due to Concurrent Updates



```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "10:00",
}
```

```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "10:00",
}
```

Conflicts due to Concurrent Updates



User A:

```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "10:00",
}
```

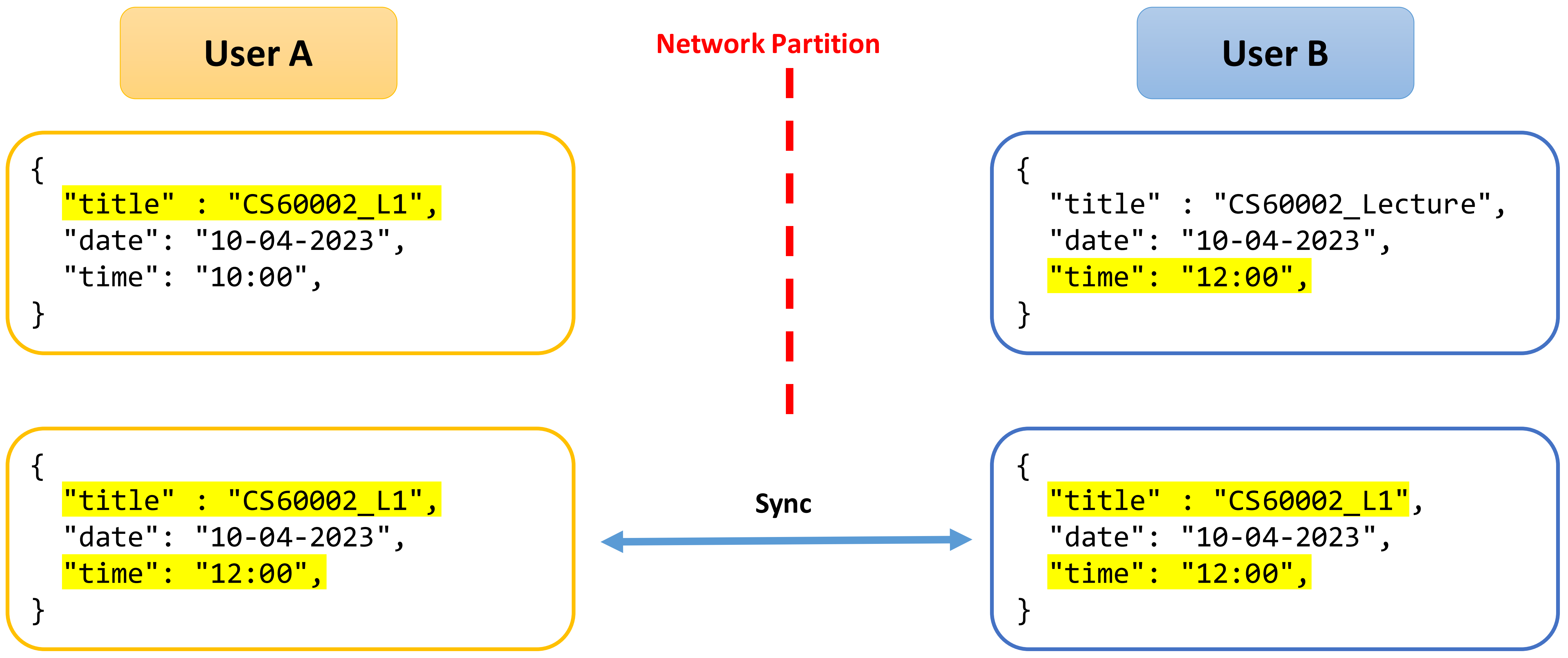
```
{
  "title" : "CS60002_L1",
  "date": "10-04-2023",
  "time": "10:00",
}
```

User B:

```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "10:00",
}
```

```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "12:00",
}
```

Conflicts due to Concurrent Updates



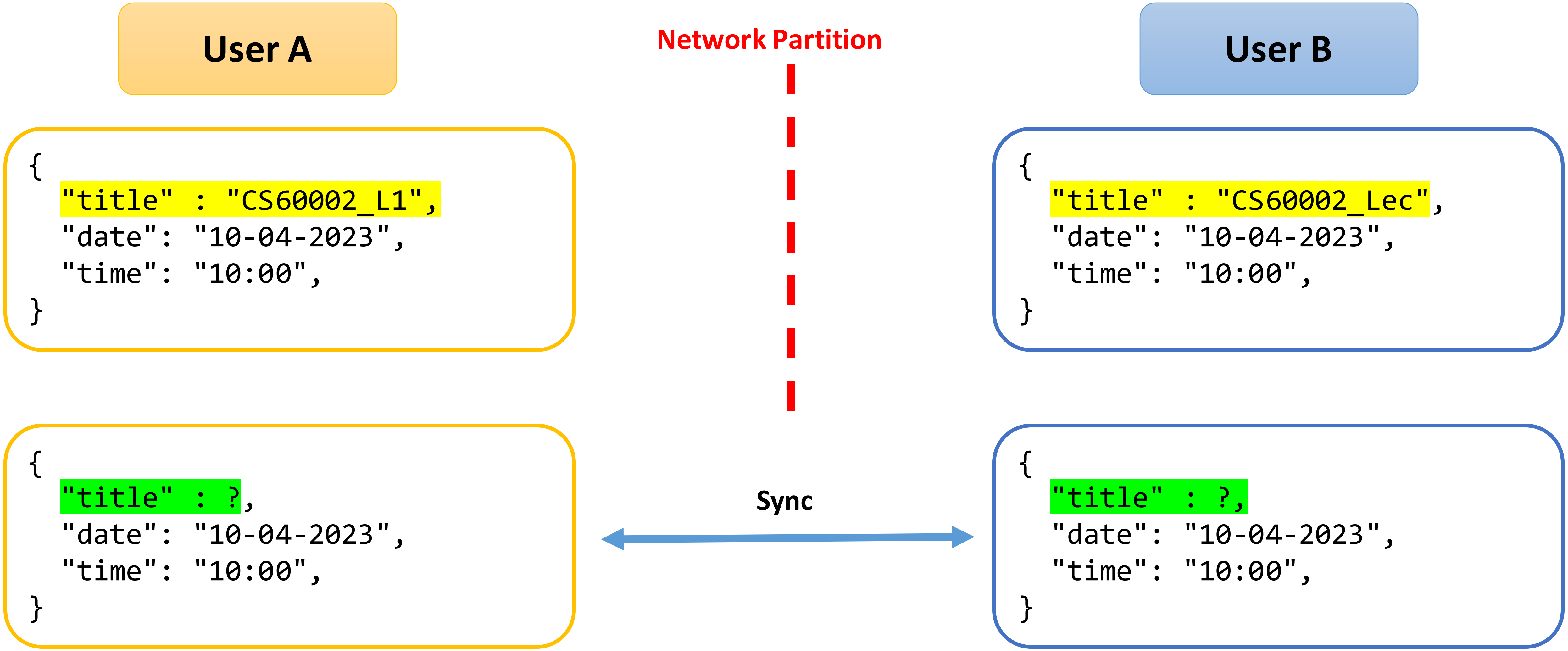
```
{
  "title" : "CS60002_L1",
  "date": "10-04-2023",
  "time": "10:00",
}
```

```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "12:00",
}
```

```
{
  "title" : "CS60002_L1",
  "date": "10-04-2023",
  "time": "12:00",
}
```

```
{
  "title" : "CS60002_L1",
  "date": "10-04-2023",
  "time": "12:00",
}
```

Conflicts due to Concurrent Updates



```
{
  "title" : "CS60002_L1",
  "date": "10-04-2023",
  "time": "10:00",
}
```

```
{
  "title" : "CS60002_Lec",
  "date": "10-04-2023",
  "time": "10:00",
}
```

```
{
  "title" : ?,
  "date": "10-04-2023",
  "time": "10:00",
}
```

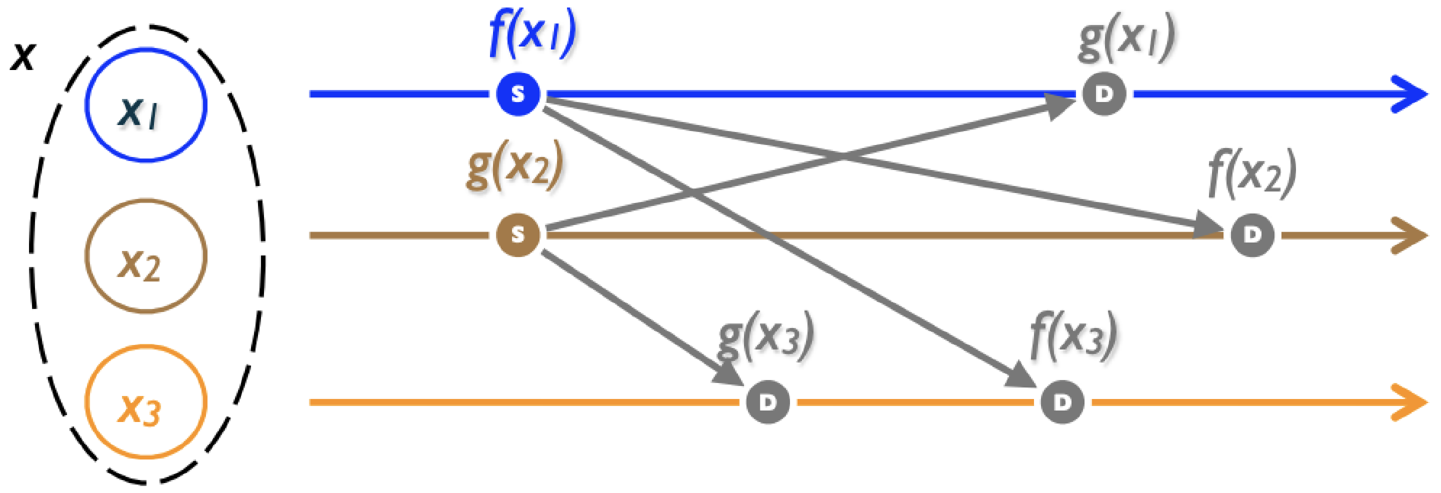
```
{
  "title" : ?,
  "date": "10-04-2023",
  "time": "10:00",
}
```

Algorithms for Conflict Resolution

- **Conflict-free Replicated Data Types** (CRDTs) -- also sometimes known as "convergent replicated data type" or "commutative replicated data type"
    - Some simple mathematical properties ensure eventual consistency
    - Operation-based
    - State-based
    - A trivial example – replicated counter with increment and decrement operations – the operations are commutative

- **Operational Transformation** (OT)

Operation based CRDT



- Consistency is ensured if the operations are commutative
  - Replicated counters with increment and decrement operations

Operation-based CRDT

- Also called **"Commutative Replicated Data Types"** (CmRDTs)
- Propagates states by transmitting only the update operations
  - a CmRDT of a single integer might broadcast the operations (+10) or (−20)
- Replicas receive the updates and apply them locally.
  - The operations are **commutative**

Operation-based CRDT

- Use reliable broadcast to deliver the updates to other replicas
  - However, with reliable broadcast, updates can be delivered in any order
    - `R-broadcast (set, t1, "title", "CS60002_L1")`
    - `R-broadcast (set, t2, "time", "10:00")`
  - Note that the message timestamps need to be globally unique across processes, say, Lamport timestamp

Operation-based CRDT

- Use reliable broadcast to deliver the updates to other replicas
  - However, with reliable broadcast, updates can be delivered in any order
    - `R-broadcast (set, t1, "title", "CS60002_L1")`
    - `R-broadcast (set, t2, "time", "10:00")`
  - Note that the message timestamps need to be globally unique across processes, say, Lamport timestamp

- 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 CRDT

- CRDT algorithm implements this
  - Reliable broadcast ensures that every operation is eventually delivered to every non-faulty replicas
  - Apply a commutative operation – order of delivery doesn't matter

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** $(\mathsf{set}, t, k, v)$ by reliable broadcast (including to self)
**end on**

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

State-based CRDT

- Also known as **Convergent Replicated Data Types (CvRDTs)**
- CvRDTs send their full local states to other replicas
  - States are merged by a function

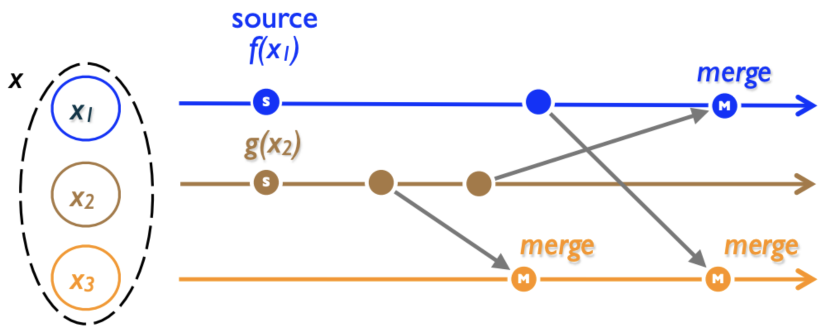
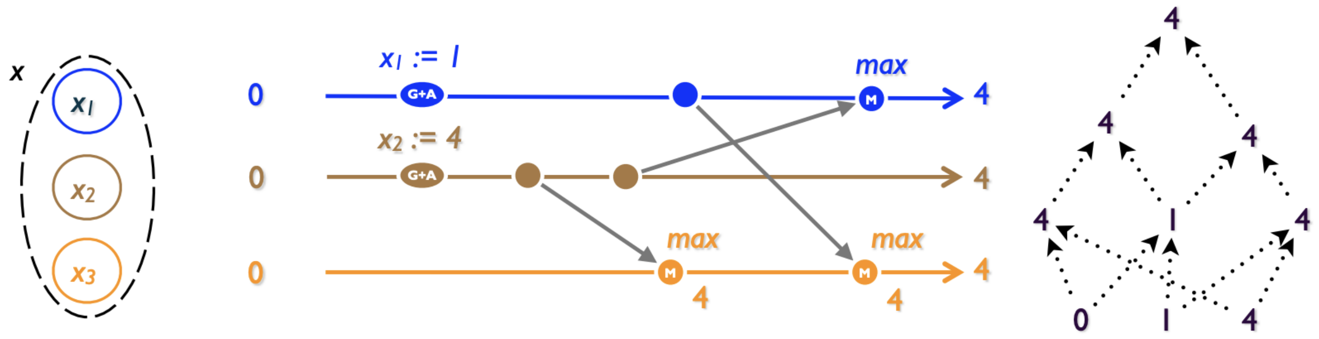


Figure 4: State-based replication



State-based (key, value)-map CRDT

- The merge operator $\sqcup$ merges to states s_1 and s_2 as follows:

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

- **What should be the properties of the merge function?**

State-based (key, value)-map CRDT

- The merge operator $\sqcup$ merges to states s_1 and s_2 as follows:

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

- **What should be the properties of the merge function?**
  - **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:** $s_1 \sqcup s_1 = s_1$

- The merge function provides a join for any pair of replica states, so the set of all states forms a **semilattice**.

State-based (key, value)-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
    $values := \{(t', k', v') \in values \mid k' \neq k\} \cup \{(t, k, v)\}$
    broadcast $values$ by best-effort broadcast
end on

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

Operation-based vs State-based CRDT

- Operation-based CRDTs typically have smaller number of messages
- State-based CRDTs can tolerate message loss or duplication

G-Counter

- These counters only support increment operations
  - Can be used to implement the like button in the social media
- **CmRDT**: The increment operation is transmitted to all other replicas
  - There is only one update in each "like" click – easy to implement
- **CvRDT**: The counter state is transmitted to all other replicas
  - Counter's count is the state
  - Since the count always increases, modeling the state as count automatically makes it a join semilattice (through the *max* function).

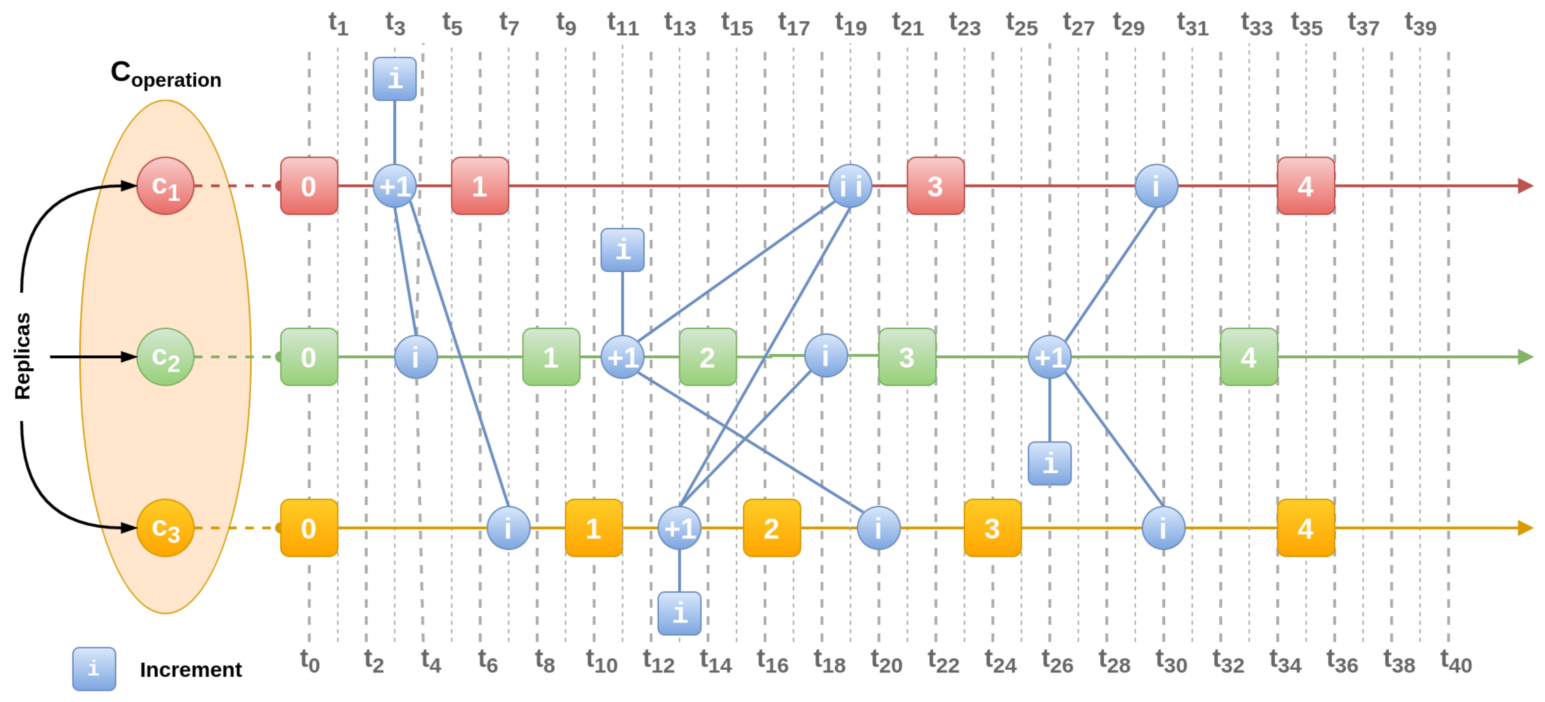
G-Counter - CmRDT

```
On initialization do
        Count = 0
End on

On request to read value do
        Return Count
End on

On request to update value do
        Broadcast (replica, "increment") through
        reliable broadcast, including self
End on

On delivering by reliable broadcast do
        Count += 1
End on
```



Operation based (CmRDT) G-Counter

Example image: https://www.cs.utexas.edu/~rossbach/cs380p/papers/Counters.html

G-Counter - CvRDT

```
On initialization with value v do
    Count = v
End on

On request to read value do
    Return Count
End on

On request to update value do
    Count += 1
    Broadcast Count by best effort broadcast
End on

On delivering value v by reliable broadcast do
    Count = max (Count, v)
End on
```

Is this a correct design?

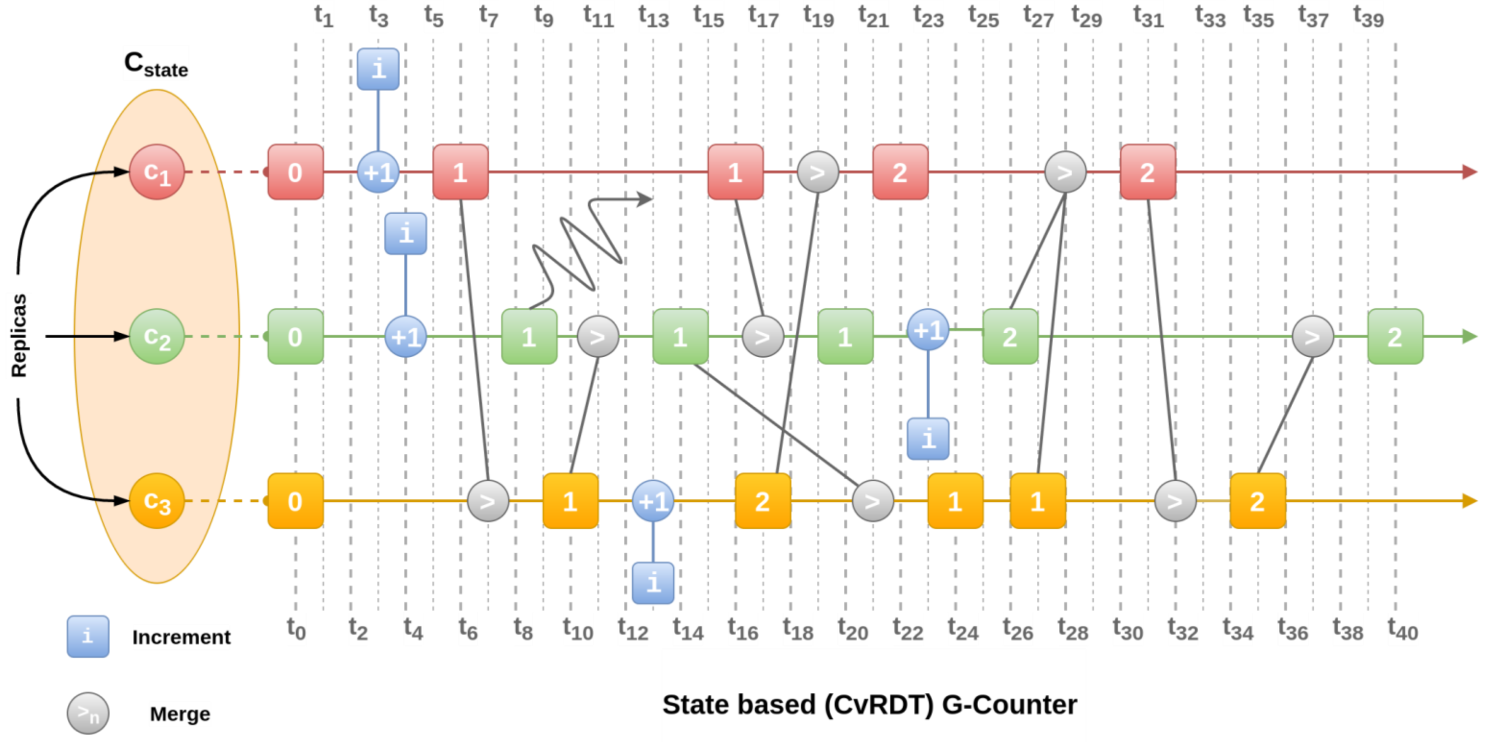
G-Counter - CvRDT

```
On initialization with value v do
    Count = v
End on

On request to read value do
    Return Count
End on

On request to update value do
    Count += 1
    Broadcast Count by best effort broadcast
End on

On delivering value v by reliable broadcast do
    Count = max (Count, v)
End on
```



State based (CvRDT) G-Counter

Total four increments, but eventual value is 2 – the design is incorrect!

Example image: https://www.cs.utexas.edu/~rossbach/cs380p/papers/Counters.html

G-Counter - CvRDT

- Why the previous design is incorrect?
  - The merge operation simply compares the state of the replicas and returns the bigger of the two counts
  - What we really need is to add the two counts as we need the total counts issued by all clients across all replicas

G-Counter - CvRDT

- Why the previous design is incorrect?
  - The merge operation simply compares the state of the replicas and returns the bigger of the two counts
  - What we really need is to add the two counts as we need the total counts issued by all clients across all replicas

- Let the merge operation add the two counts – **is this correct?**

G-Counter - CvRDT

- Why the previous design is incorrect?
  - The merge operation simply compares the state of the replicas and returns the bigger of the two counts
  - What we really need is to add the two counts as we need the total counts issued by all clients across all replicas
- Let the merge operation add the two counts – **is this correct?**
  - The merge operation is no longer **idempotent** -- repeated merging of the same values will not return the same result
  - If a message gets duplicated, we'll get incorrect result

G-Counter - CvRDT

- Why the previous design is incorrect?
  - The merge operation simply compares the state of the replicas and returns the bigger of the two counts
  - What we really need is to add the two counts as we need the total counts issued by all clients across all replicas

- Let the merge operation add the two counts – **is this correct?**
  - The merge operation is no longer **idempotent** -- repeated merging of the same values will not return the same result
  - If a message gets duplicated, we'll get incorrect result

- The problem is with the representaton of the states. **How do we represent the states correctly so that the merge operation works fine?**

G-Counter - CvRDT

- Represent the state as a sequence of counts – each value in the sequence corresponds to the count of a replica
  - Number of values in the state equals to the number of replicas
  - merge operation computes the index-wise maximum of the state.

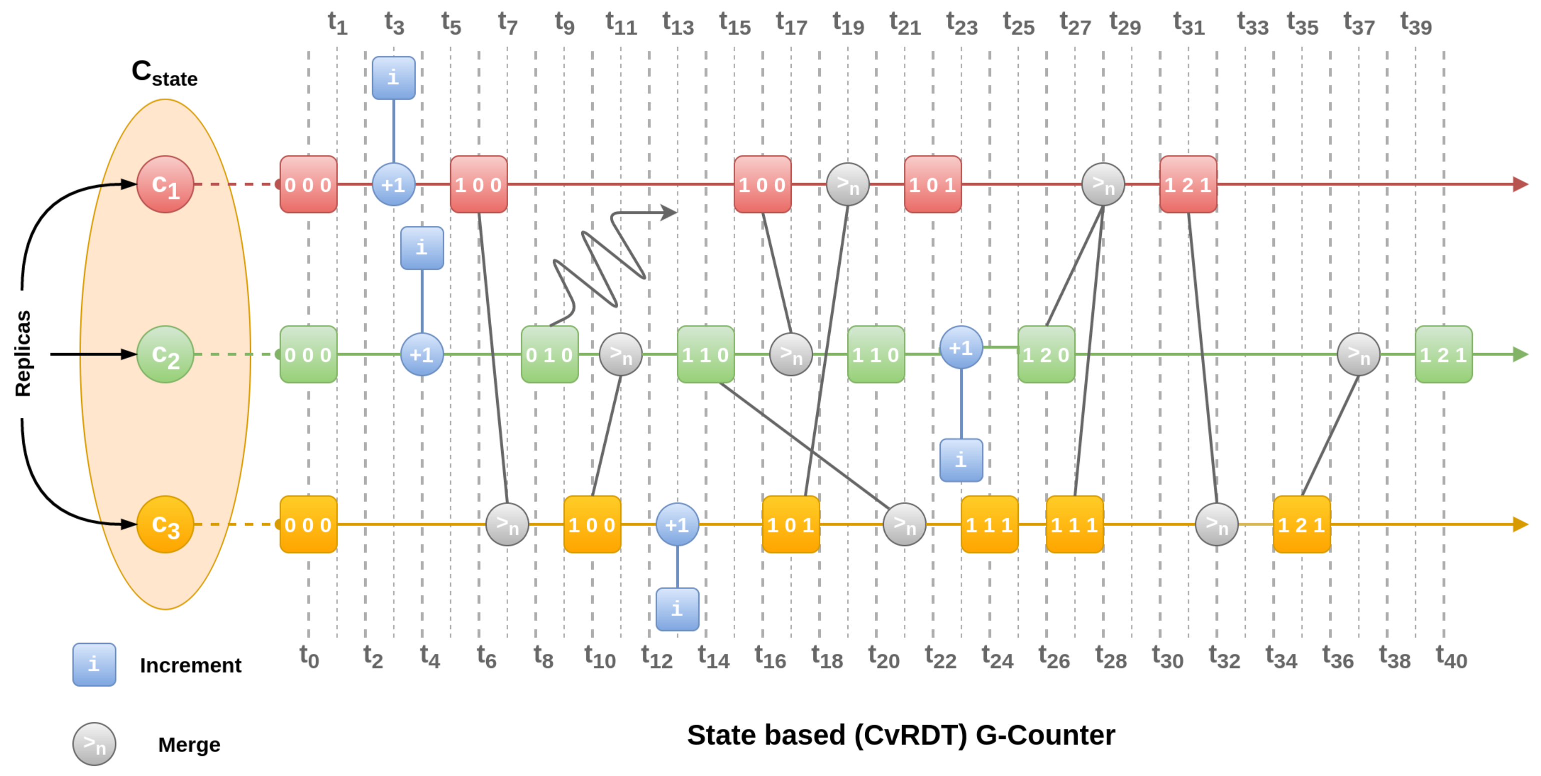
G-Counter - CvRDT

```
On initialization do
    Count = { } with number of entries equal to
                the number of replicas
End on

On request to read value do
    Return Sum(Count)
End on

On request to update value do
    Count[self.value] +=1
    Broadcast Count by best effort broadcast
End on

On delivering value $C_v$ by reliable broadcast do
    Count[self.value] = max (Count[value],
                               $C_v$[value])
End on
```



State based (CvRDT) G-Counter

Example image: https://www.cs.utexas.edu/~rossbach/cs380p/papers/Counters.html

Related Readings

- Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). A comprehensive study of Convergent and Commutative Replicated Data Types. In *[Research Report] RR-7506, Inria – Centre Paris-Rocquencourt* (p. 50).

- Marc Shapiro, C. B., Nuno Preguiça, & Zawirski, M. (2011). Conflict-free Replicated Data Types. In *[Research Report] RR-7687, INRIA* (p. 18).

Happy
Learning