CS 60002: Distributed Systems

T11: Concurrency in Distributed Applications

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY
KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

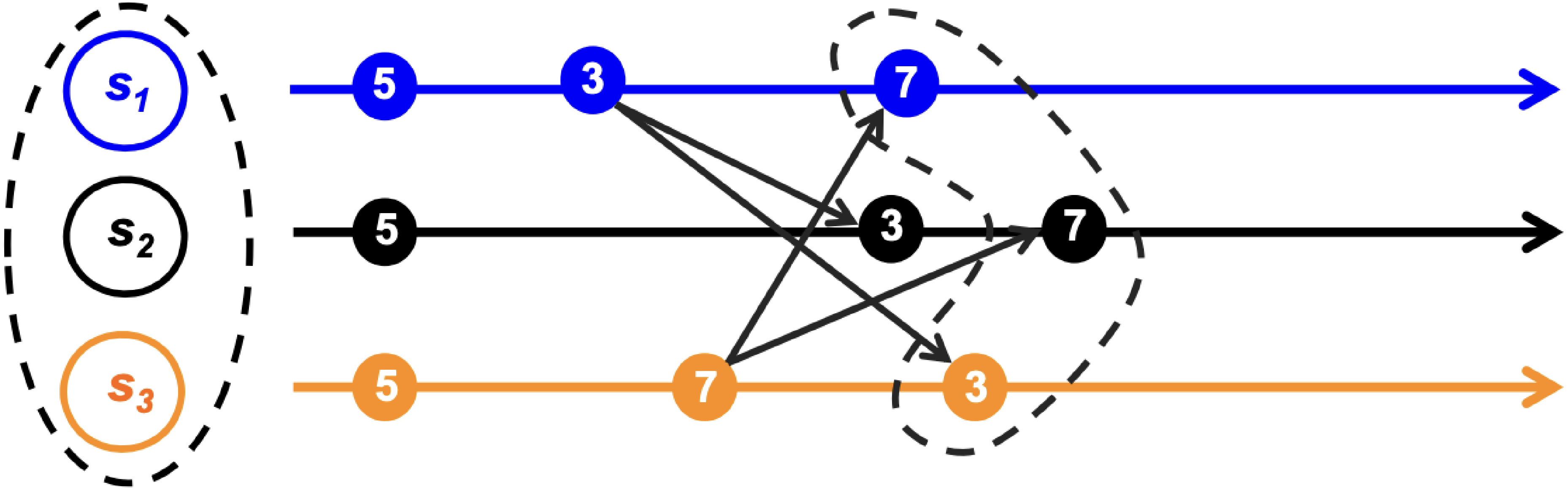
Content courtesy: Lecture slides by Prof. Vijay Chidambaram, UTexas
Course Link: https://github.com/vijay03/cs380d-s21

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?

Conflicts

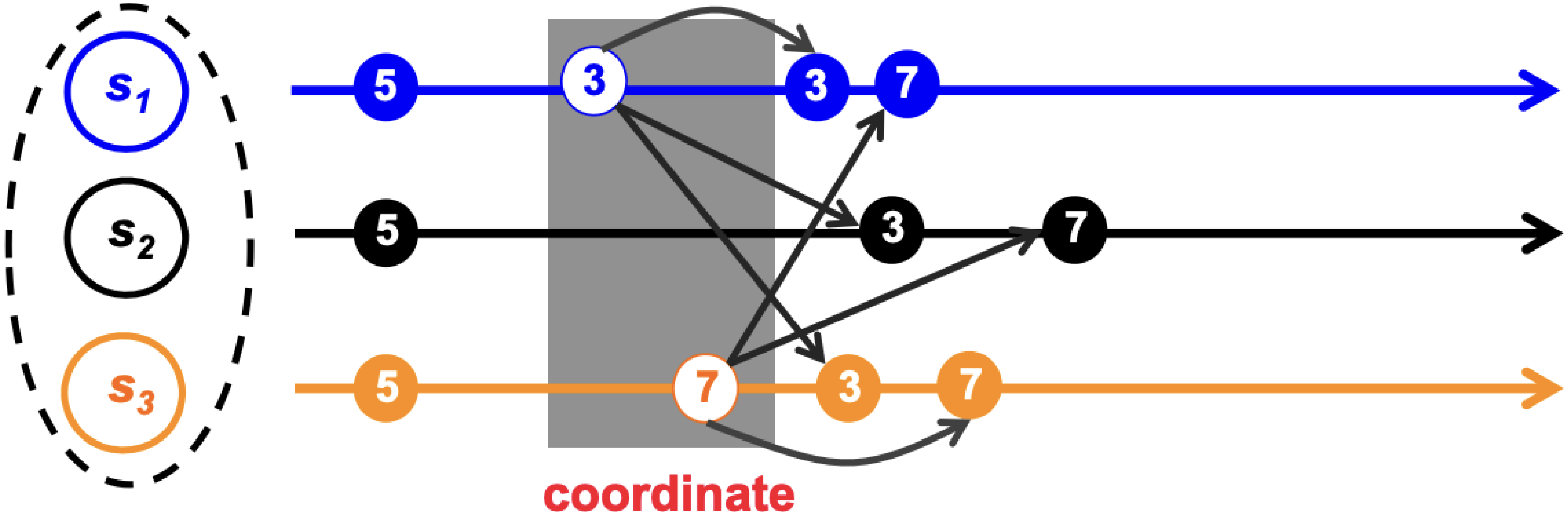
- Updating replicas may lead to different results, thus making the data inconsistent



https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F21/lectures/lec09-CRDT.pdf

Strong Consistency

- All replicas execute updates in the same total order
  - **Ensuring deterministic updates: Same update on the same objects -> same results**



https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F21/lectures/lec09-CRDT.pdf

Strong Consistency

- All replicas execute updates in the same total order
  - **Ensuring deterministic updates: Same update on the same objects -> same results**

- Requires coordination and consensus to decide on the total order of operations (Raft, PBFT)
  - **N-way agreement:** Serializable updates – very expensive
  - Consensus is good as long as the network is not partitioned (remember the 2f+1 requirement in CFT or 3f + 1 requirement in BFT) -- we need a large number of replicas – costly from CAPEX as well as OPEX
  - What if the network gets partitioned?

https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F21/lectures/lec09-CRDT.pdf

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

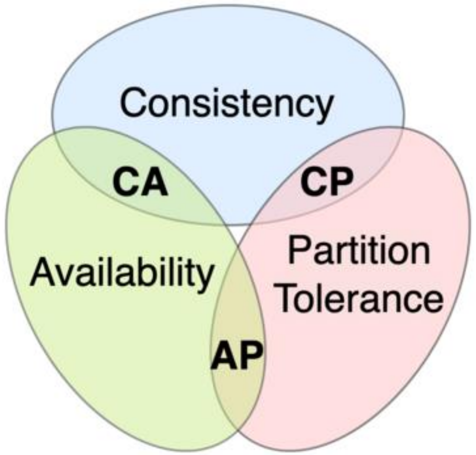


Image source: Wikipedia

- **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 that do not need strong consensus: **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

User A

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

Network Partition

User B

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

Conflicts due to Concurrent Updates

User A

Network Partition

User B

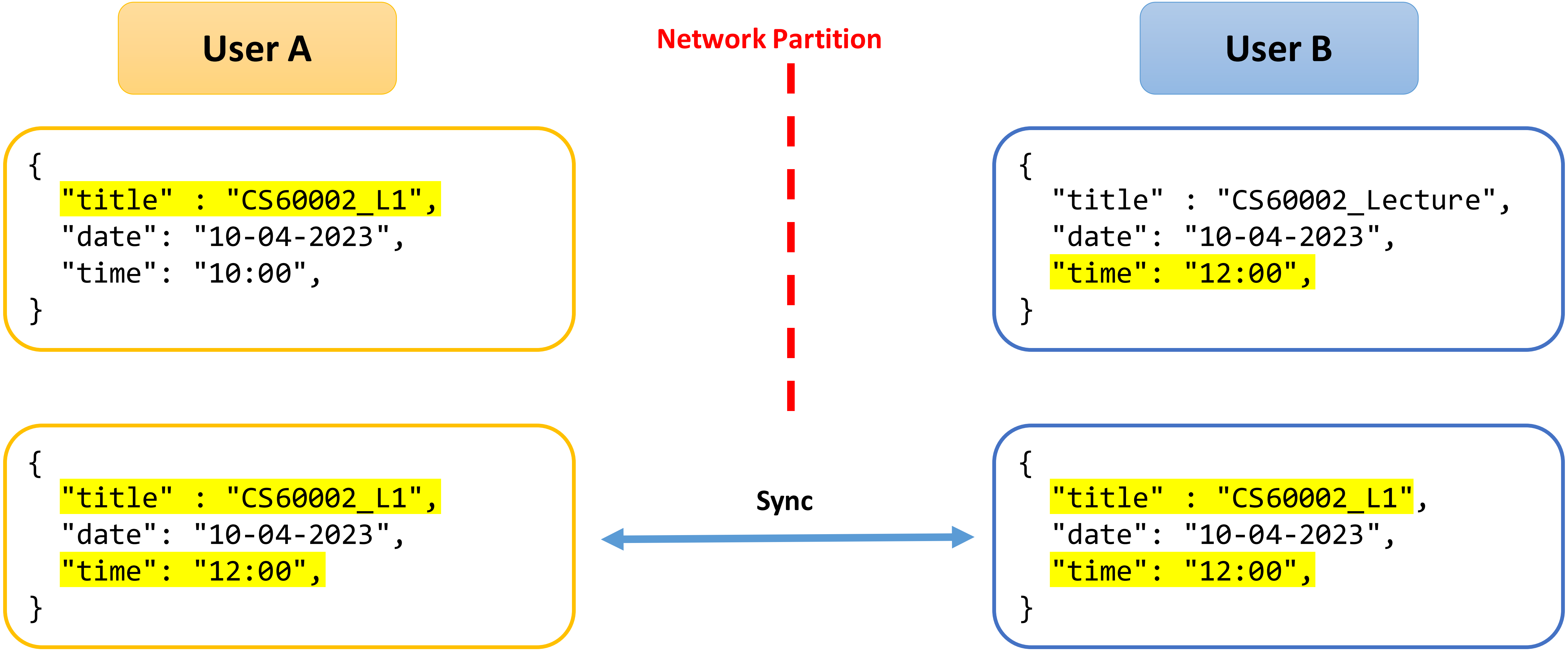
```
{
  "title" : "CS60002_Lecture",
  "date": "10-04-2023",
  "time": "10:00",
}
```

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

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

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

Conflicts due to Concurrent Updates



User A:

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

User B:

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

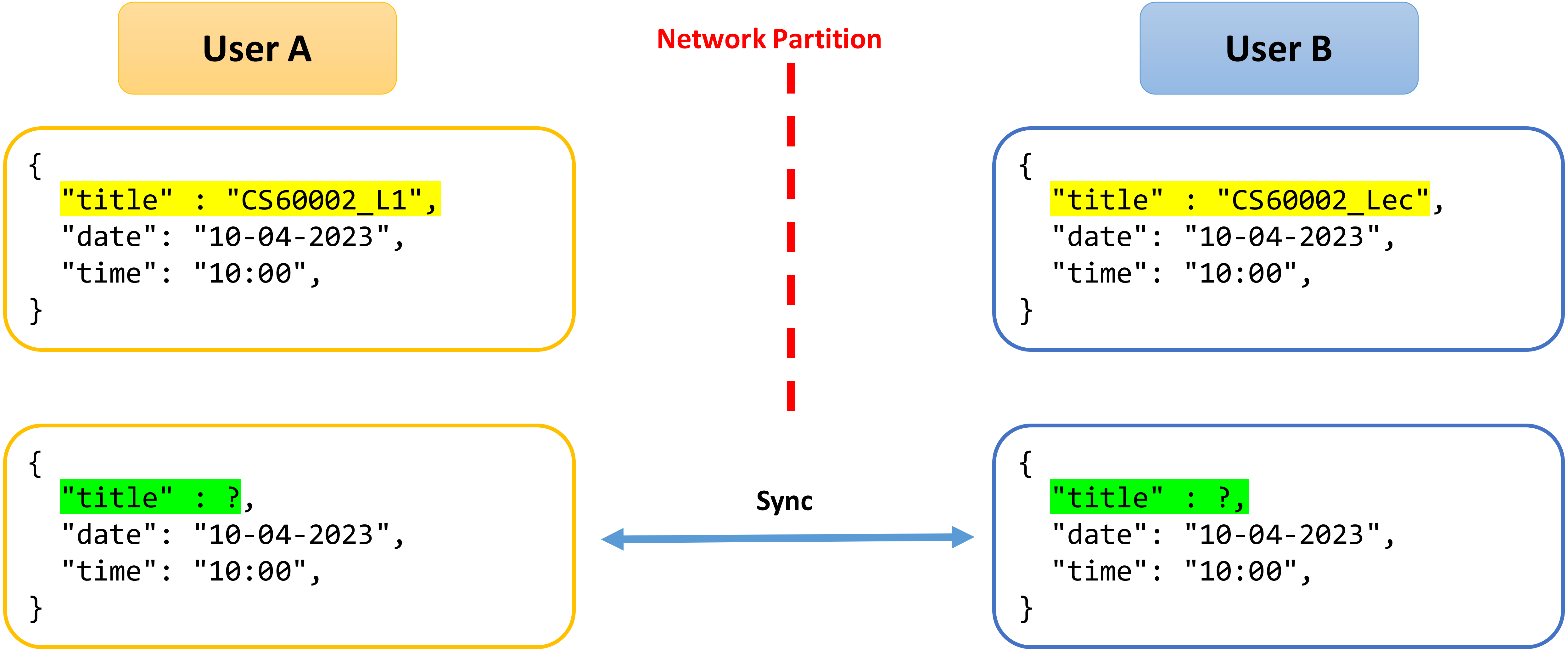
After Sync, User A:

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

After Sync, User B:

```
{
  "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",
}
```

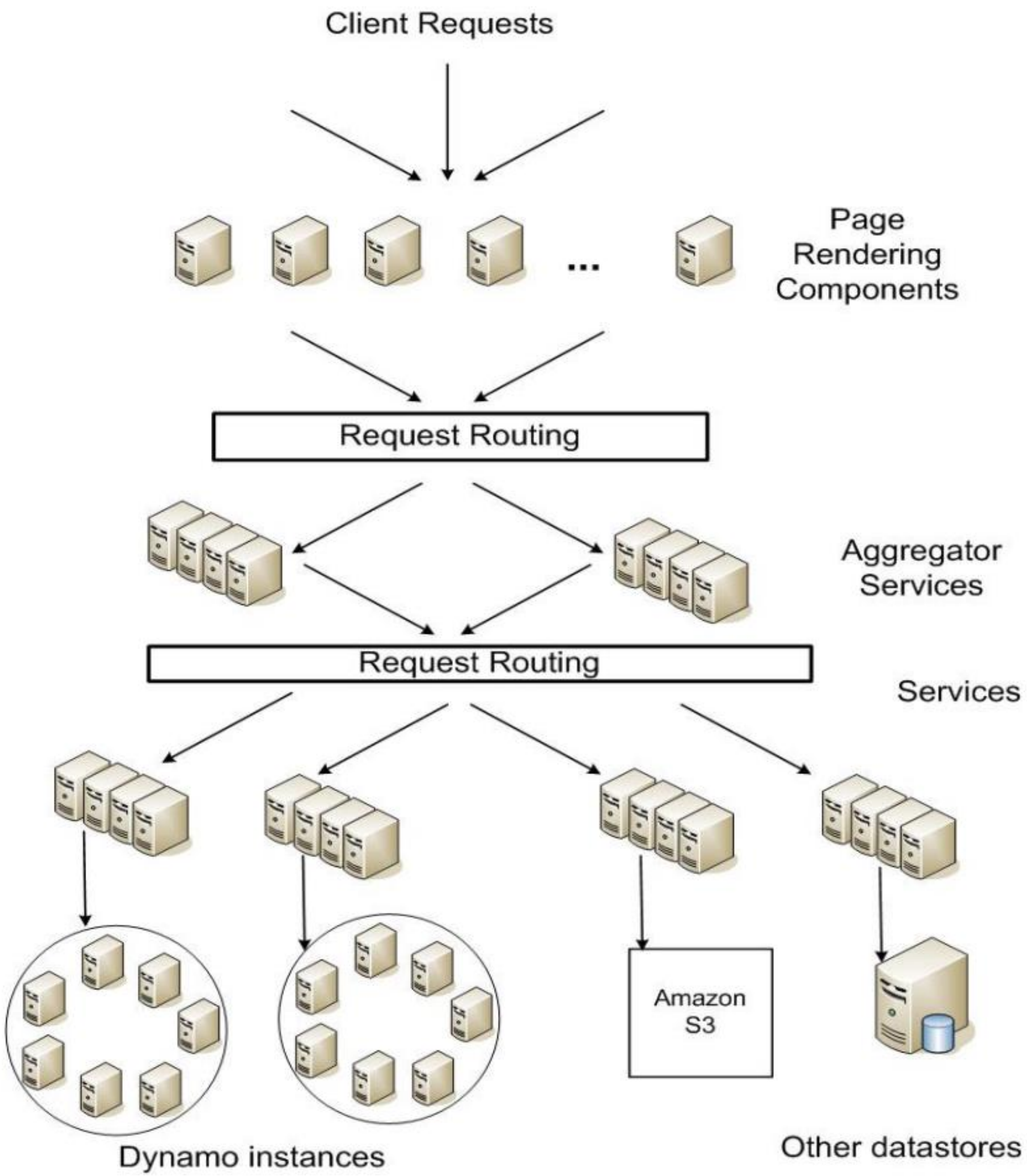
```
{
  "title" : "CS60002_Lec",
  "date": "10-04-2023",
  "time": "10:00",
}
```

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

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

Dynamo: Amazon's highly available key-value store

- An example when you do not need strong consistency



Dynamo: Amazon's highly available key-value store

- "*Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.*"

- "*a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios.* "

Dynamo: Amazon's highly available key-value store

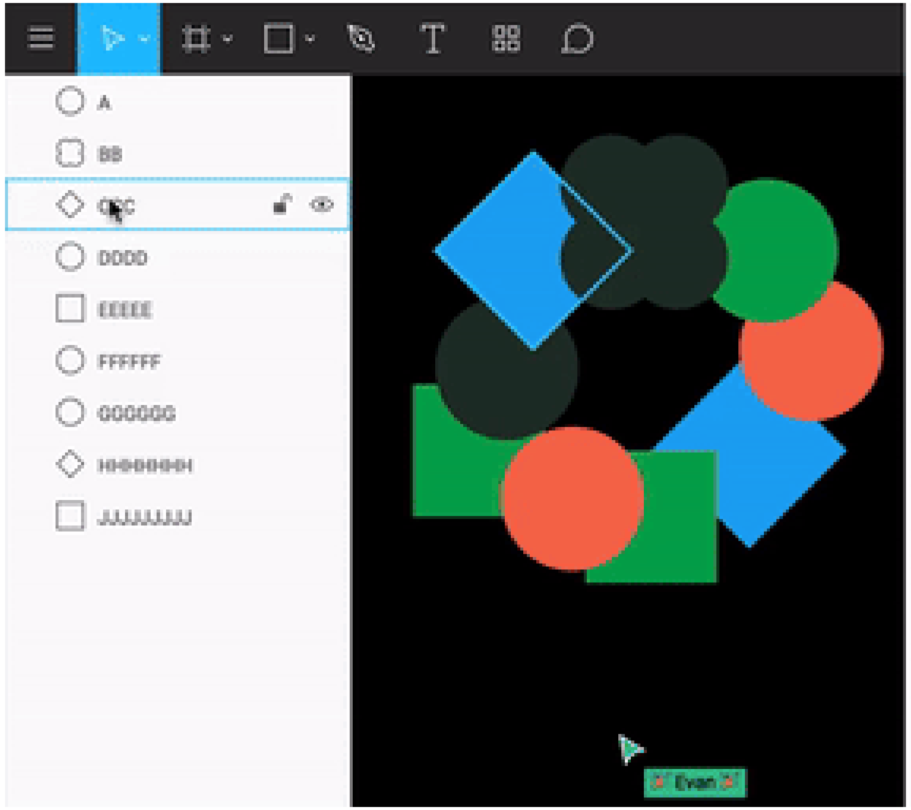
- **Must be available when partition happens**
  - *“For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.”*

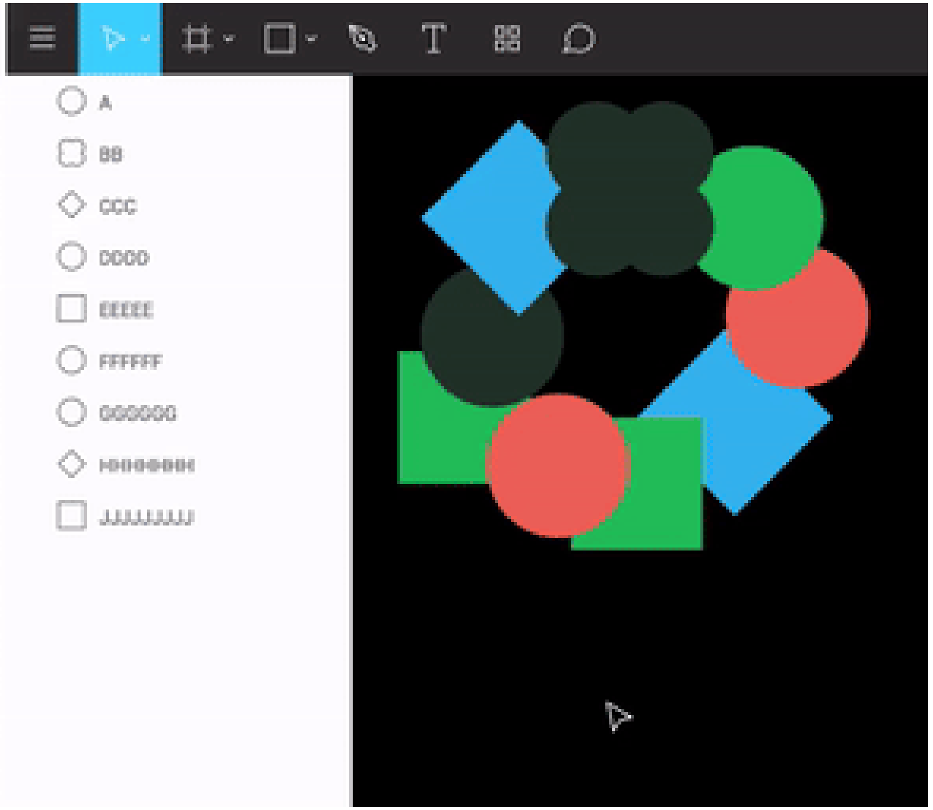
Dynamo: Amazon's highly available key-value store

- **Tradeoff with consistency**
  - "*There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions.*
    *For example, the shopping cart application requires that an "**Add to Cart**" operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. Note that both "**Add to Cart**" and "**Delete item from Cart**" operations are translated into **put** requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later."*

Another Example: Real-time Collaborative Editing

- https://www.figma.com/blog/realtime-editing-of-ordered-sequences/



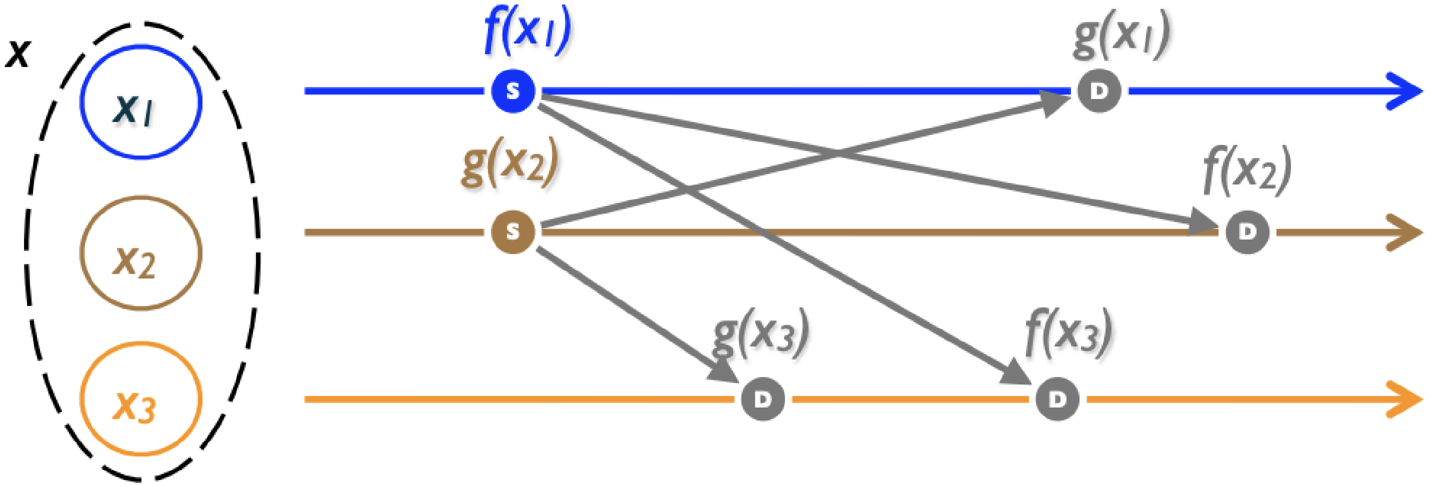


Algorithms for Conflict Resolution

- **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

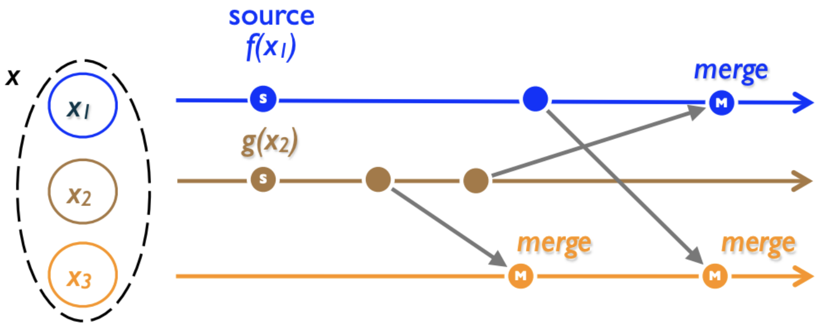
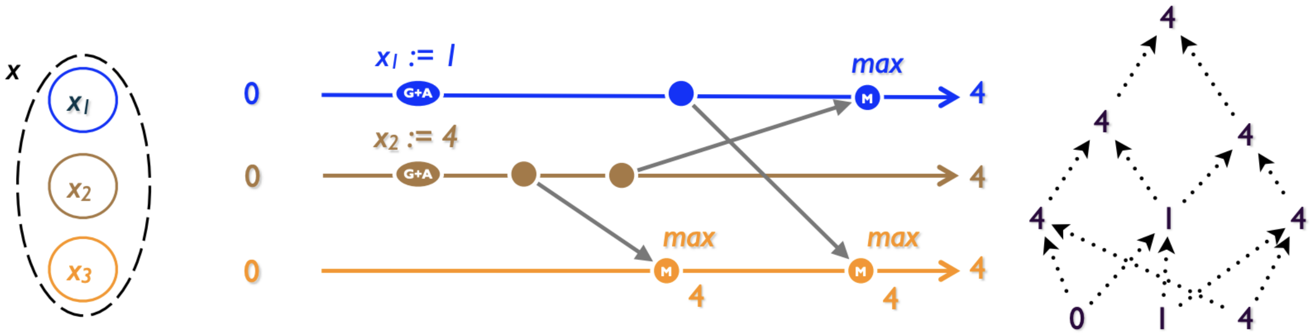


Figure 4: State-based replication



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

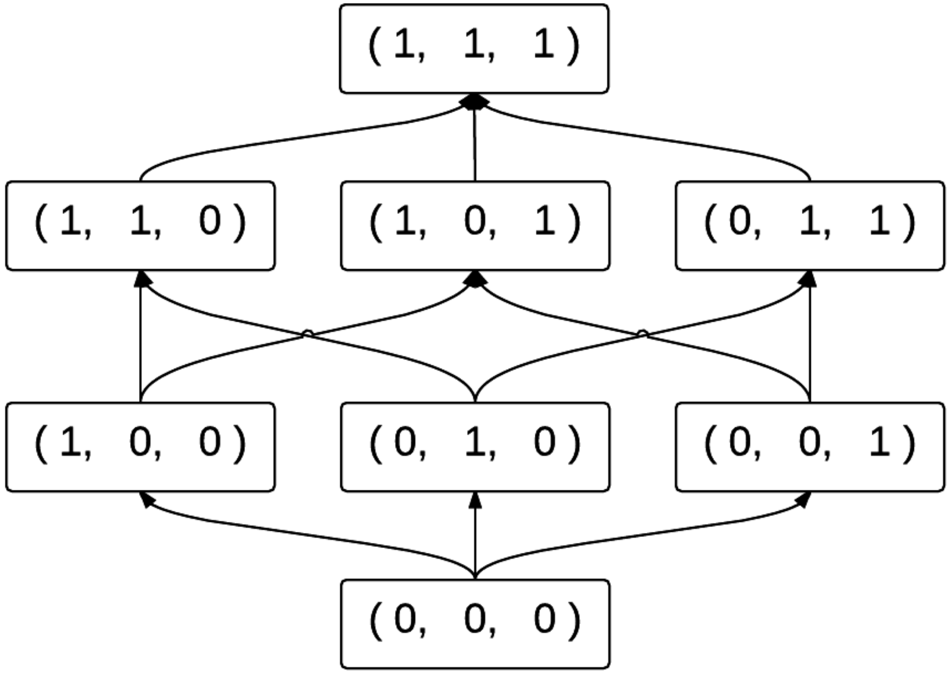
- The merge operator $\sqcup$ merges to states s_1 and s_2 as follows:

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

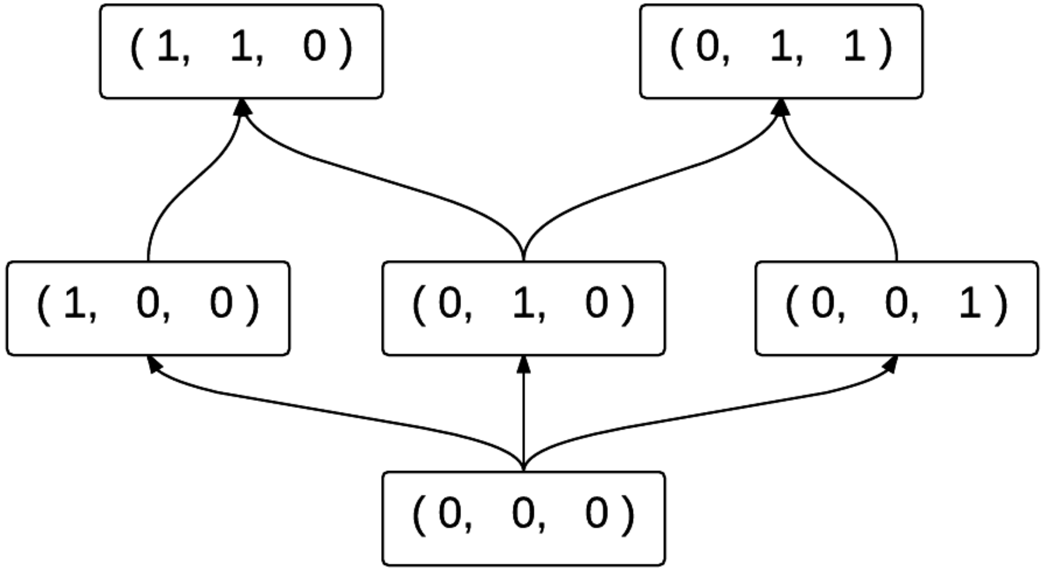
- **What should be the properties of the merge function?**
  - **Commutative:** $s_1 \sqcup s_2 = s_2 \sqcup s_1$
  - **Associative:** $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$
  - **Idempotent:** $s_1 \sqcup s_1 = s_1$

- The merge function provides a join for any pair of replica states, so the set of all states forms a **semilattice** (a partially ordered set that has a join (a least upper bound) for any nonempty finite subset).

Join Semilattice



A Join Semilattice



(1, 1, 0) v (0, 1, 1)
does not exist

NOT a Join 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
    $t :=$ 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

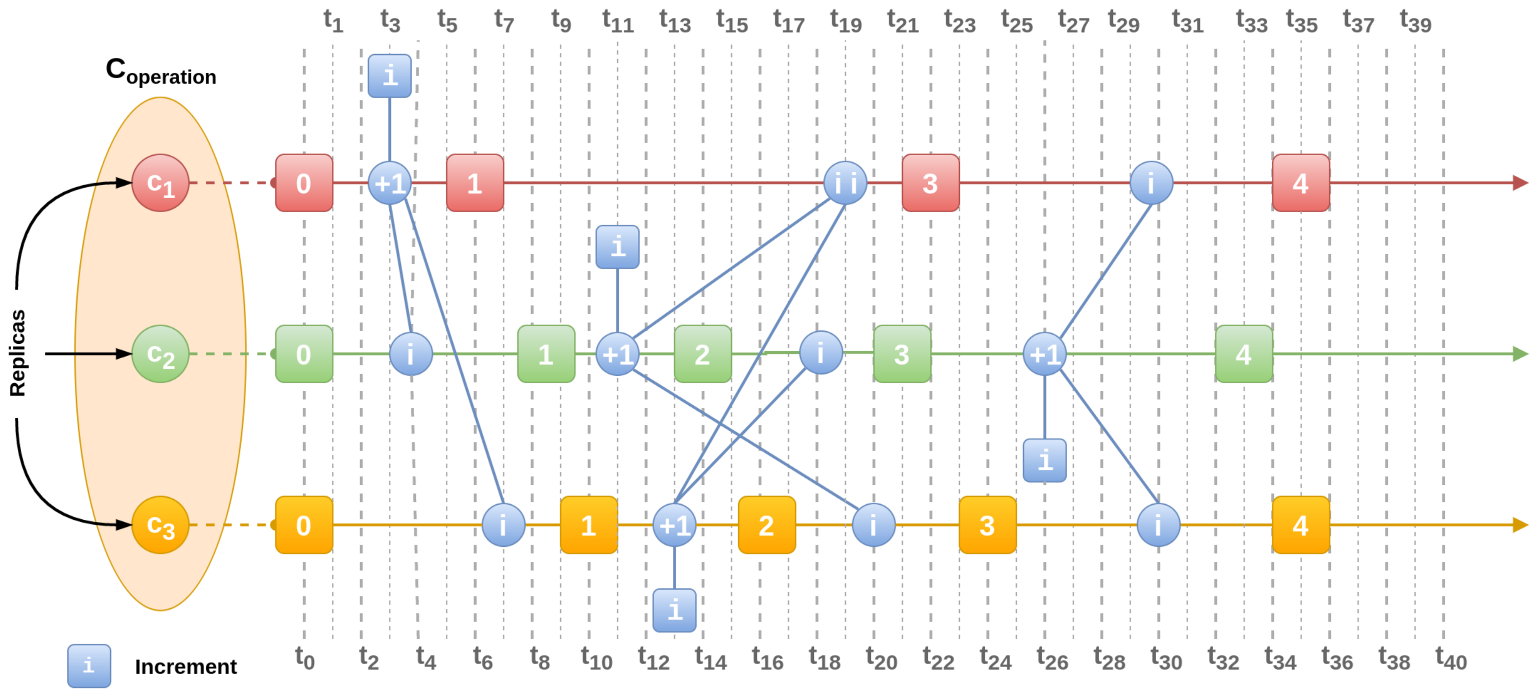
- These counters only support increment operations
  - Can be used to implement the like button in the social media

- **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
```



t1 t3 t5 t7 t9 t11 t13 t15 t17 t19 t21 t23 t25 t27 t29 t31 t33 t35 t37 t39
C operation
Replicas
c1
c2
c3
0
+1
1
i
ii
3
4
2
i
Increment
t0 t2 t4 t6 t8 t10 t12 t14 t16 t18 t20 t22 t24 t26 t28 t30 t32 t34 t36 t38 t40

Operation based (CmRDT) G-Counter

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

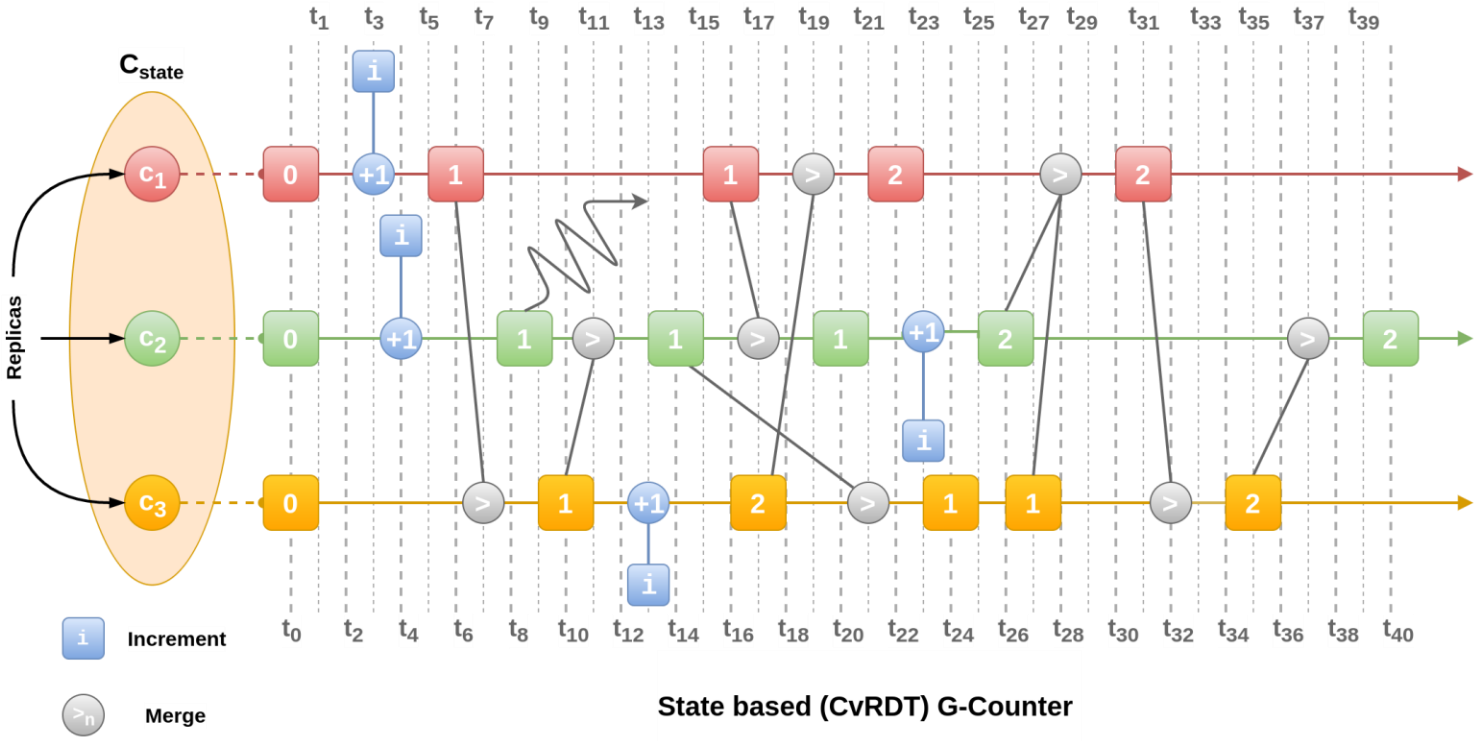
G-Counter - CvRDT

```
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
```



t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
C_{state}
Replicas
c_1
c_2
c_3
i
+1
0
1
2
>
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}
i Increment
$>_n$ Merge

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

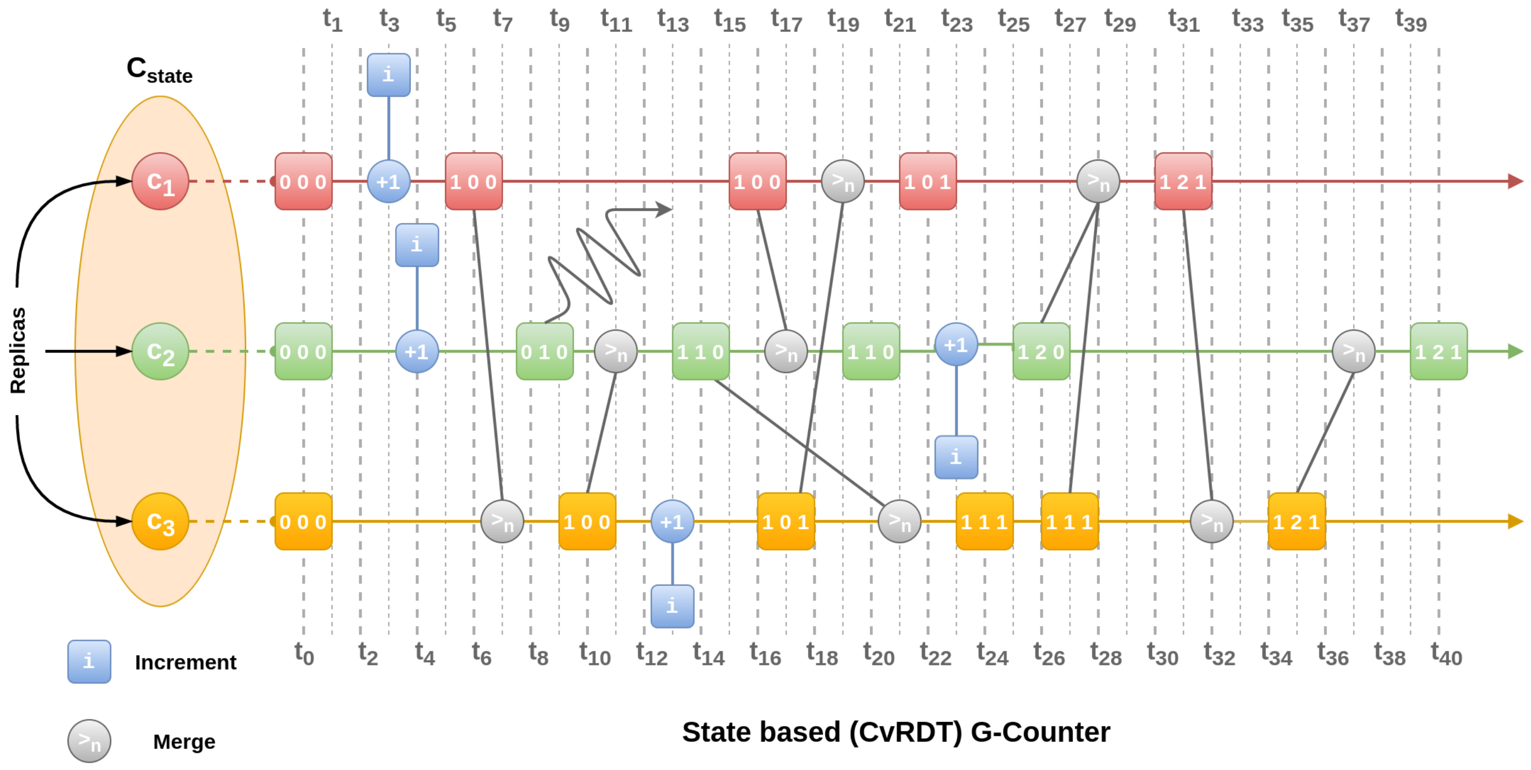
- The problem is with the representaton of the states. **How do we represent the states correctly so that the merge operation works fine?**

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
```



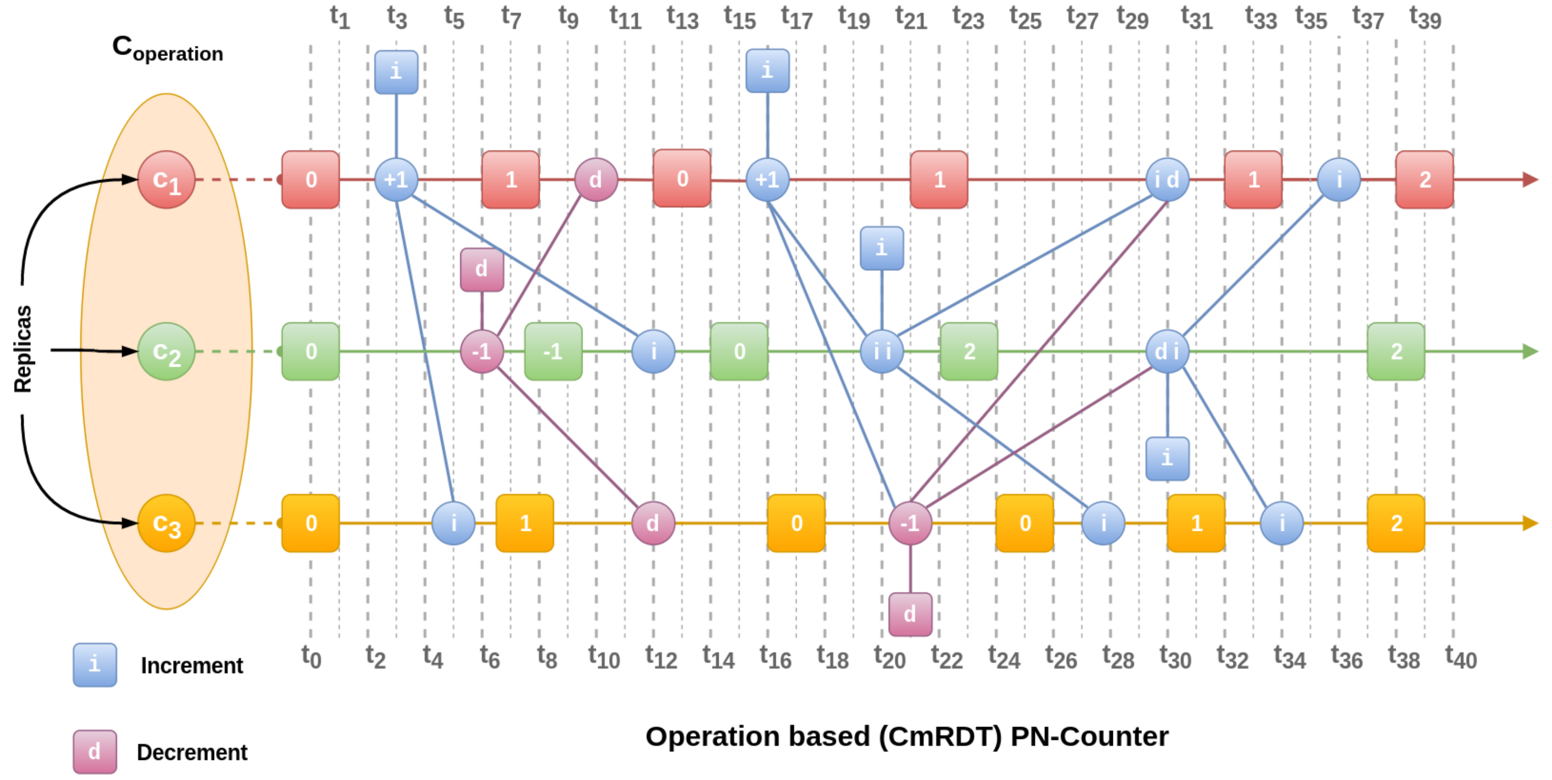
C_{state}
Replicas
c_1
c_2
c_3
t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}
0 0 0
i
+1
1 0 0
1 0 0
1 0 1
1 2 1
0 0 0
0 1 0
1 1 0
1 1 0
1 2 0
1 2 1
0 0 0
1 0 0
1 0 1
1 1 1
1 1 1
1 2 1
Increment
Merge

State based (CvRDT) G-Counter

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

PN Counter – Increment and Decrement Counter: CmRDT

```
class Counter(CmRDT):
    def __init__(self):
        self._count = 0

    def value(self):
        return self._count

    def increment(self):
        self._count += 1
        for replica in self.replicas():
            self.transmit("increment", replica)

    def decrement(self):
        self._count -= 1
```

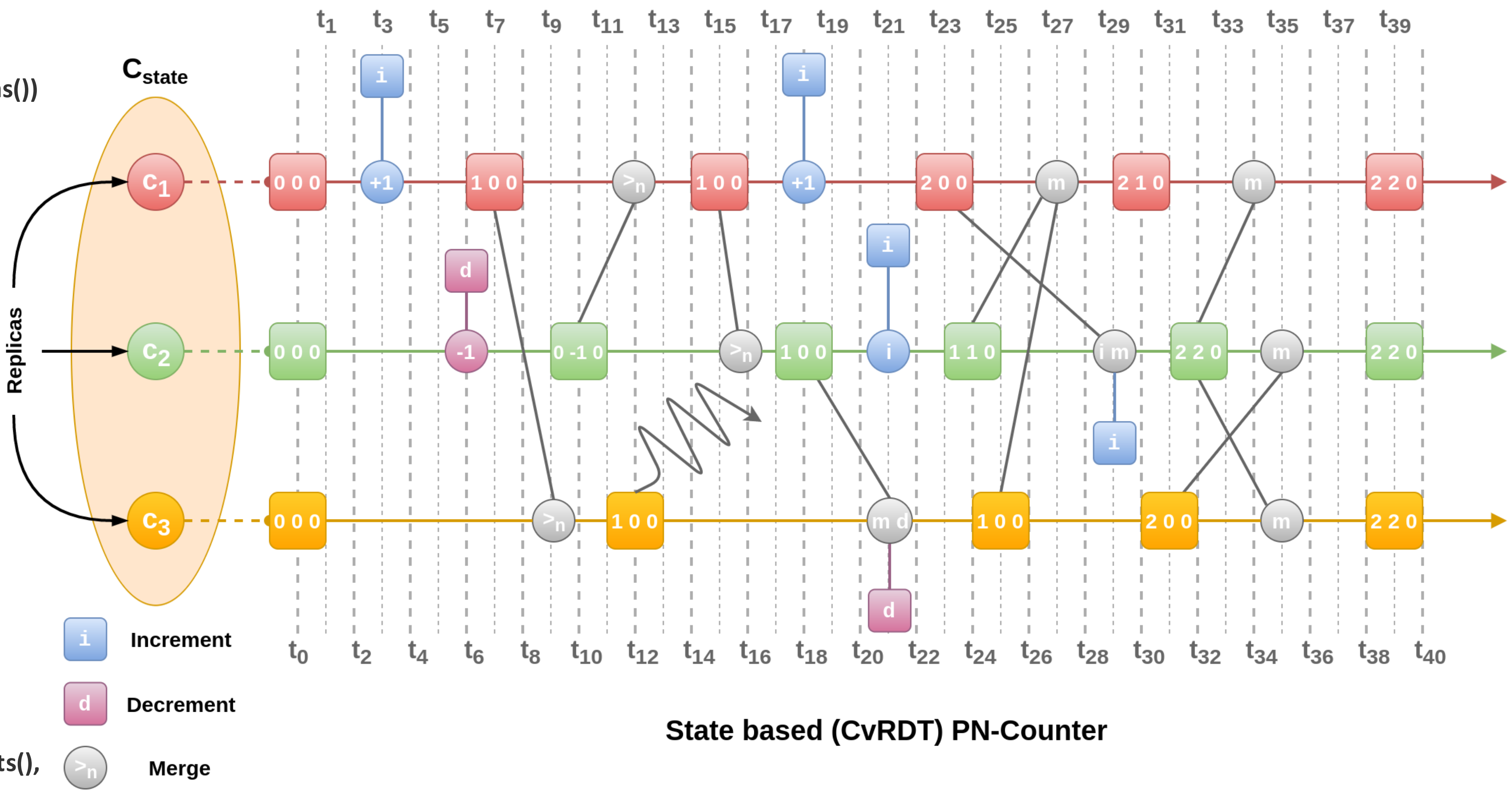


$C_{operation}$
Replicas
c_1
c_2
c_3
t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}
i
i
0 +1 1 d 0 +1 1 i d 1 i 2
d
i
0 -1 -1 i 0 i i 2 d i 2
i
0 i 1 d 0 -1 0 i 1 i 2
d
i Increment
d Decrement

Operation based (CmRDT) PN-Counter

PN Counter – Increment and Decrement Counter: CvRDT

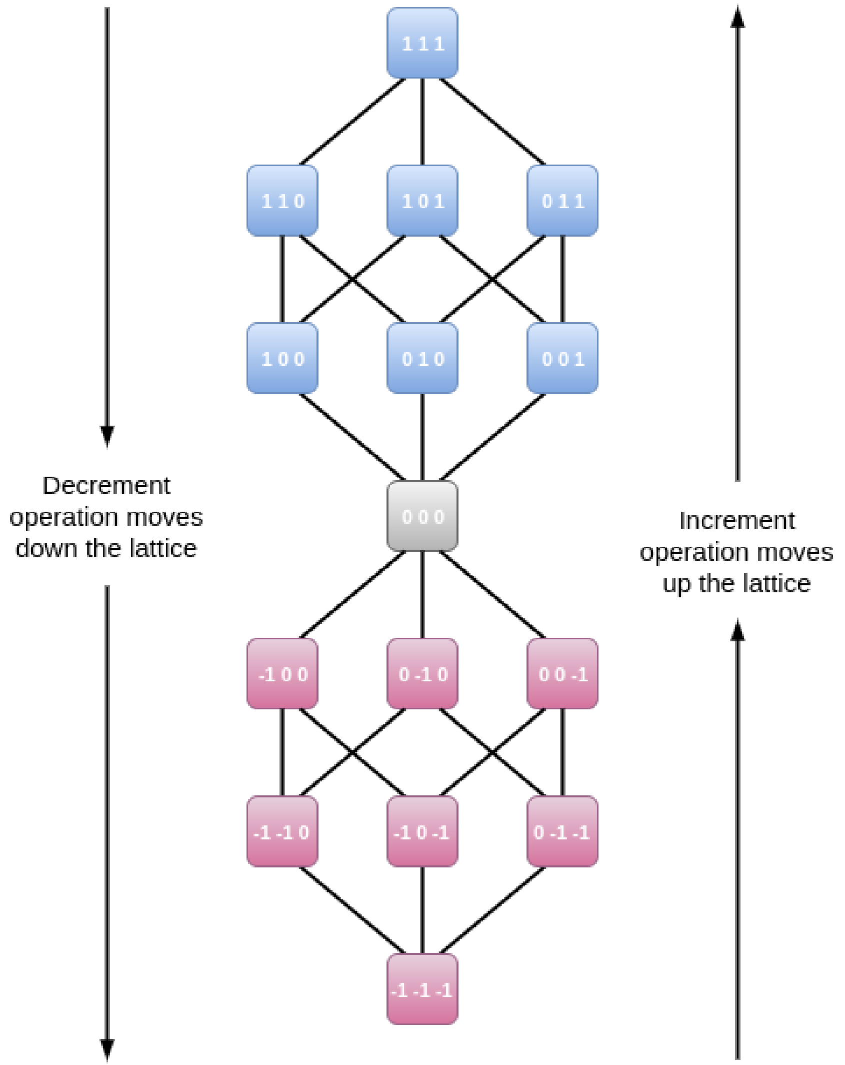
```
class Counter(CvRDT):

    def __init__(self, counts = None):
        if counts is None:
            self._counts = [0]  * length(self.replicas())
        else:
            self._counts = counts

    def value(self):
        return sum(self._counts)

    def counts(self):
  # return a copy of the counts
        return list(self._counts)

    def increment(self):
        self._counts[self.replicaId()] += 1

    def decrement(self):
        self._counts[self.replicaId()] -= 1

    def compare(self, other):
        return all(v1 <= v2 for (v1, v2) in
                zip(self.counts(),
                    other.counts()))

    def merge(self, other):
        return Counter(map(max, zip(self.counts(),
                        other.counts())))
```



C_{state}
Replicas
c_1
c_2
c_3
t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
i
i
0 0 0
+1
1 0 0
>n
1 0 0
+1
2 0 0
m
2 1 0
m
2 2 0
d
i
0 0 0
-1
0 -1 0
>n
1 0 0
i
1 1 0
i m
2 2 0
m
2 2 0
i
0 0 0
>n
1 0 0
m d
1 0 0
2 0 0
m
2 2 0
d
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}
i Increment
d Decrement
>n Merge

State based (CvRDT) PN-Counter

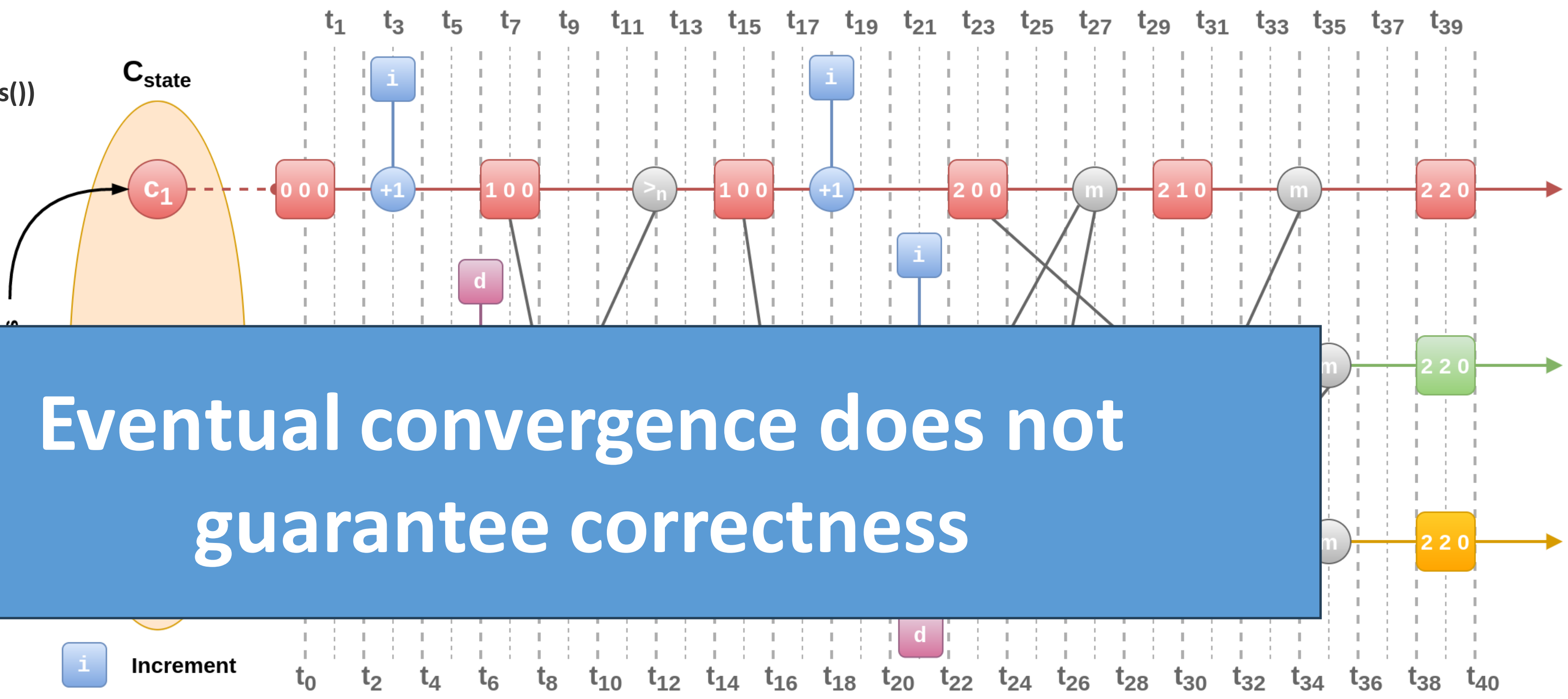
PN Counter – Increment and Decrement Counter: CvRDT



State lattice diagram (partial)

PN Counter – Increment and Decrement Counter: CvRDT

```
class Counter(CvRDT):

      def __init__(self, counts = None):
         if counts is None:
            self._counts = [0]  * length(self.replicas())
         else:
            self._counts = counts

      def value(self):
         return sum(self._counts)

      def counts(self):
   # return a copy of the counts
         return list(self._counts)

      def increment(self):
         self._counts[self.replicaId(

      def decrement(self):
         self._counts[self.replicaId(

      def compare(self, other):
         return all(v1 <= v2 for (v1, v2) in
                 zip(self.counts(),
                    other.counts()))

      def merge(self, other):
         return Counter(map(max, zip(self.counts(),
                        other.counts())))
```



C_{state}
c_1
t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
0 0 0
i
+1
1 0 0
d
$>_n$
1 0 0
i
+1
i
2 0 0
m
2 1 0
m
2 2 0
m
2 2 0
m
2 2 0
d
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}

Eventual convergence does not guarantee correctness

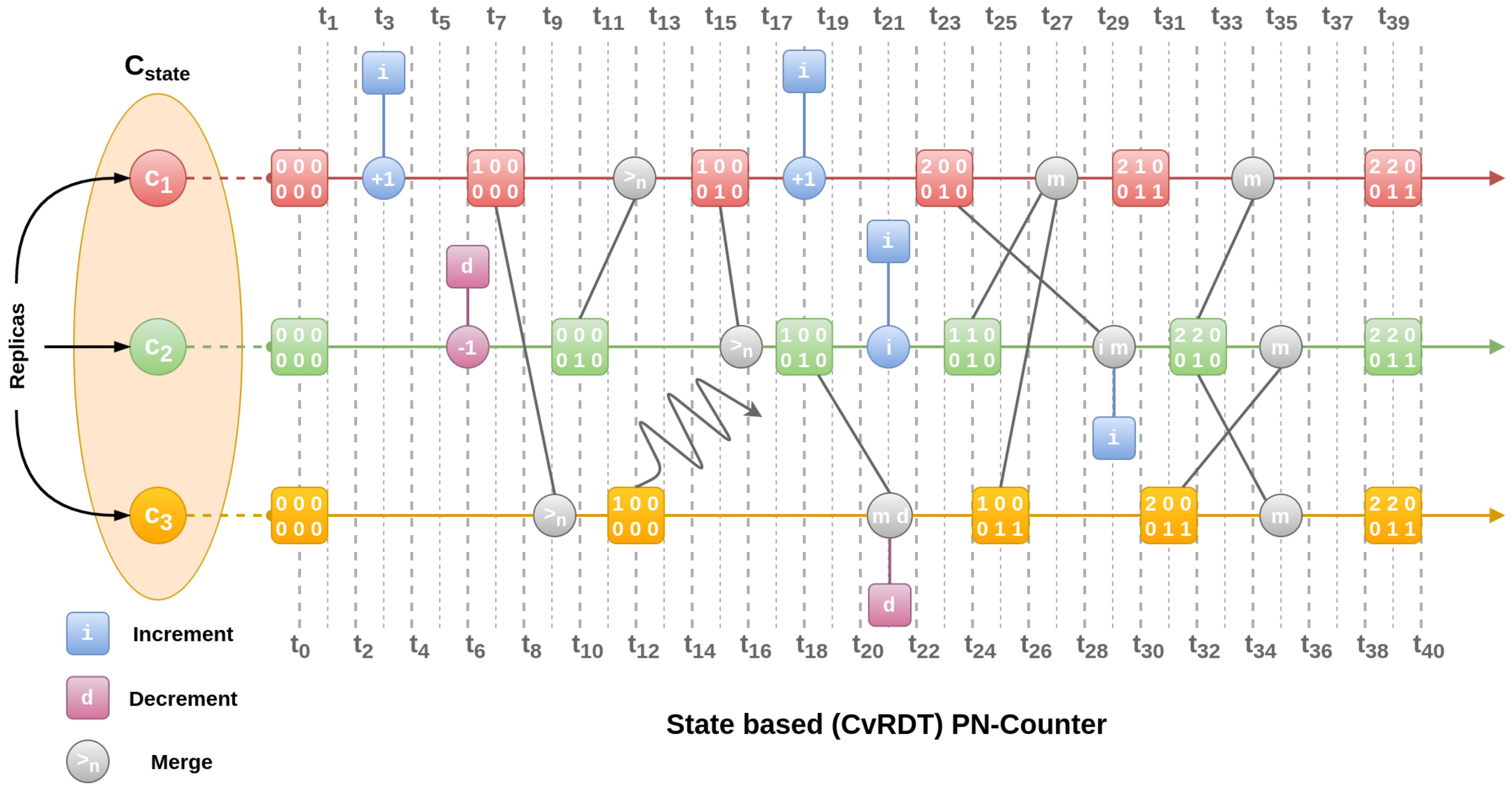
i Increment

d Decrement

$>_n$ Merge

State based (CvRDT) PN-Counter

PN Counter – Increment and Decrement Counter: CvRDT



C_{state}
Replicas
c_1
c_2
c_3
t_1 t_3 t_5 t_7 t_9 t_{11} t_{13} t_{15} t_{17} t_{19} t_{21} t_{23} t_{25} t_{27} t_{29} t_{31} t_{33} t_{35} t_{37} t_{39}
t_0 t_2 t_4 t_6 t_8 t_{10} t_{12} t_{14} t_{16} t_{18} t_{20} t_{22} t_{24} t_{26} t_{28} t_{30} t_{32} t_{34} t_{36} t_{38} t_{40}
0 0 0 / 0 0 0
i
+1
1 0 0 / 0 0 0
>n
1 0 0 / 0 1 0
i
+1
2 0 0 / 0 1 0
m
2 1 0 / 0 1 1
m
2 2 0 / 0 1 1
0 0 0 / 0 0 0
d
-1
0 0 0 / 0 1 0
>n
1 0 0 / 0 1 0
i
i
1 1 0 / 0 1 0
i m
i
2 2 0 / 0 1 0
m
2 2 0 / 0 1 1
0 0 0 / 0 0 0
>n
1 0 0 / 0 0 0
m d
d
1 0 0 / 0 1 1
2 0 0 / 0 1 1
m
2 2 0 / 0 1 1
i
Increment
d
Decrement
>n
Merge

State based (CvRDT) PN-Counter

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