Consistency and Conflict Resolution

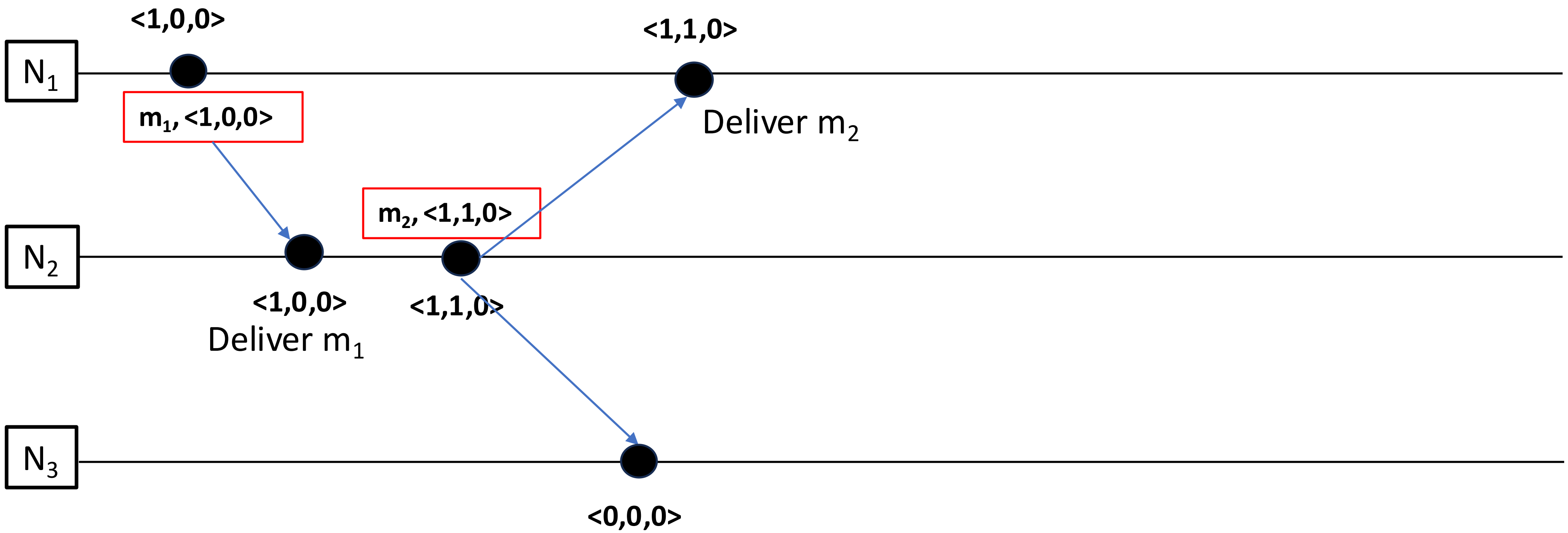
Agenda

- **Time and ordering wrap-up**
- Eventual Consistency

Causally Ordering Multicast Messages

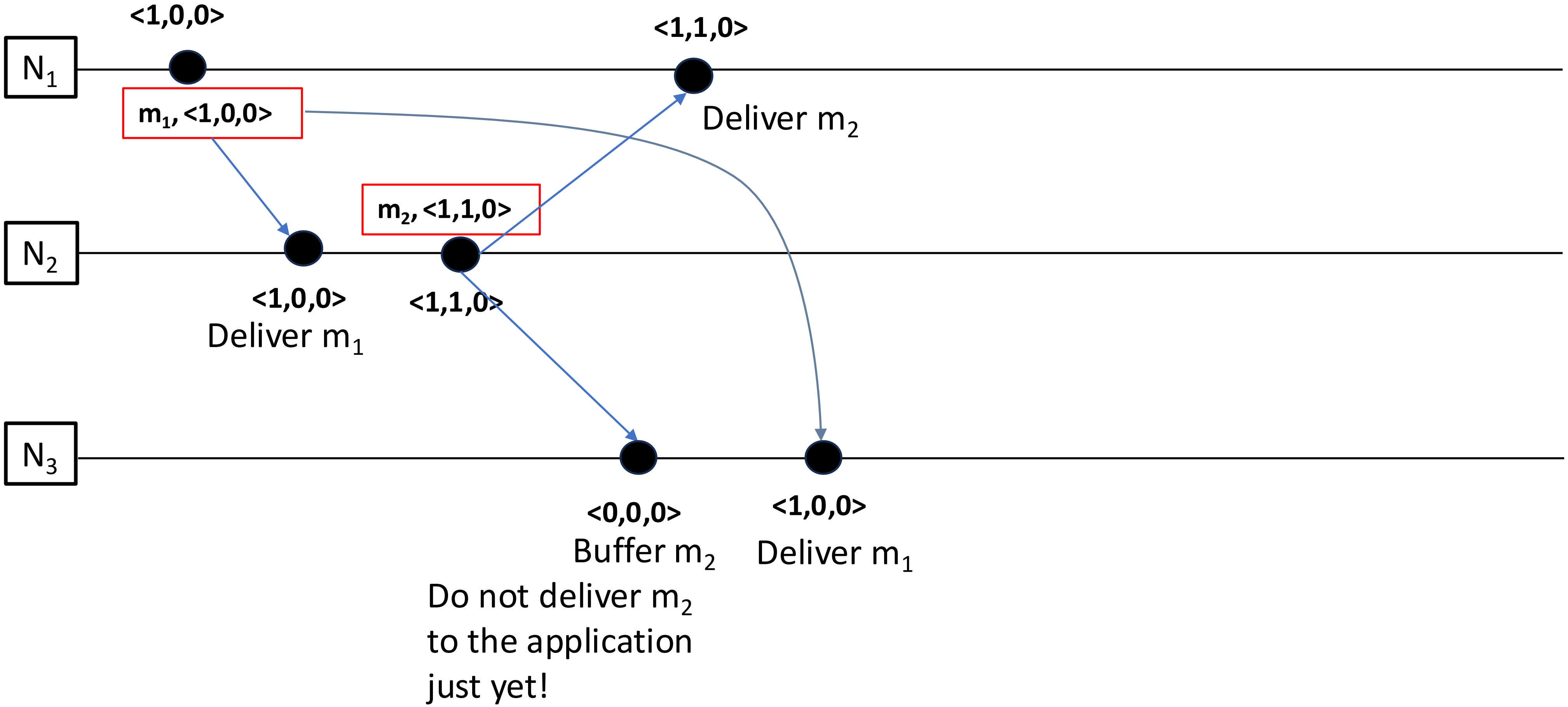
- Remember causal ordering? (Week 5, Lecture 2)
- Can we leverage Vector clocks to causally order group communication messages? Yes! (see below)
- Let's assume that clocks are adjusted only when:
  - sending messages (at the sender) – same as before
  - **delivering messages** to the application (at the receiver) – **new rule**
  - messages are buffered until they can be delivered (see below) – **new rule**
- Vector clocks are updated as follows:
  - Upon sending a message, a node N_i increments the i^{th} index of its vector clock $VC_i[i]$ by one (as discussed earlier)
  - When a Node N_j delivers a message m with timestamp TS(m), N_j will update $VC_j[i]$ to $\max\{VC_i[i], TS(m)[i]\}$ for each i
- The delivery of a message m (sent by N_i) to the application layer at N_j is delayed until:
  - $TS(m)[i] = VC_j[i]+1$
  - $TS(m)[k] \leq VC_j[k]$ for all $k \neq i$

Causally Ordering Multicast Messages

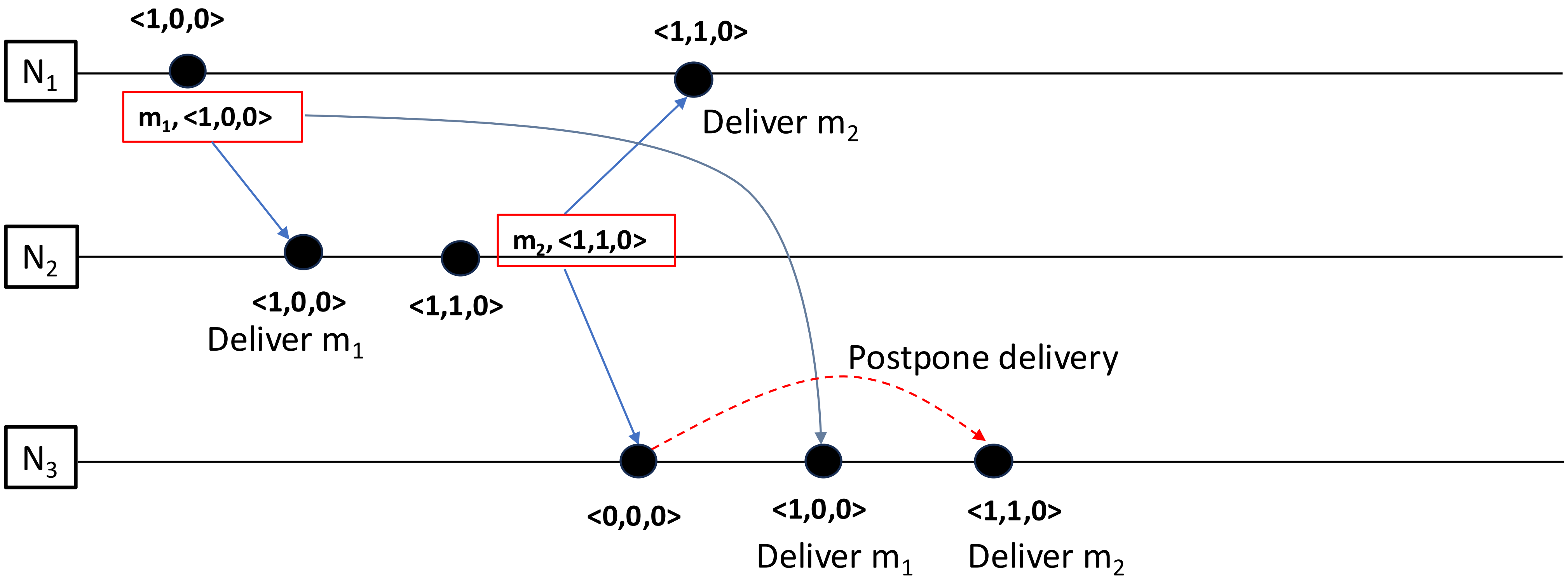


$(TS(m)[1] = 1) \leq (VC_3[1] = 0)$ does not hold! This means I am missing a message from N_1 – Place m_2 in a buffer

Causally Ordering Multicast Messages



Causally Ordering Multicast Messages

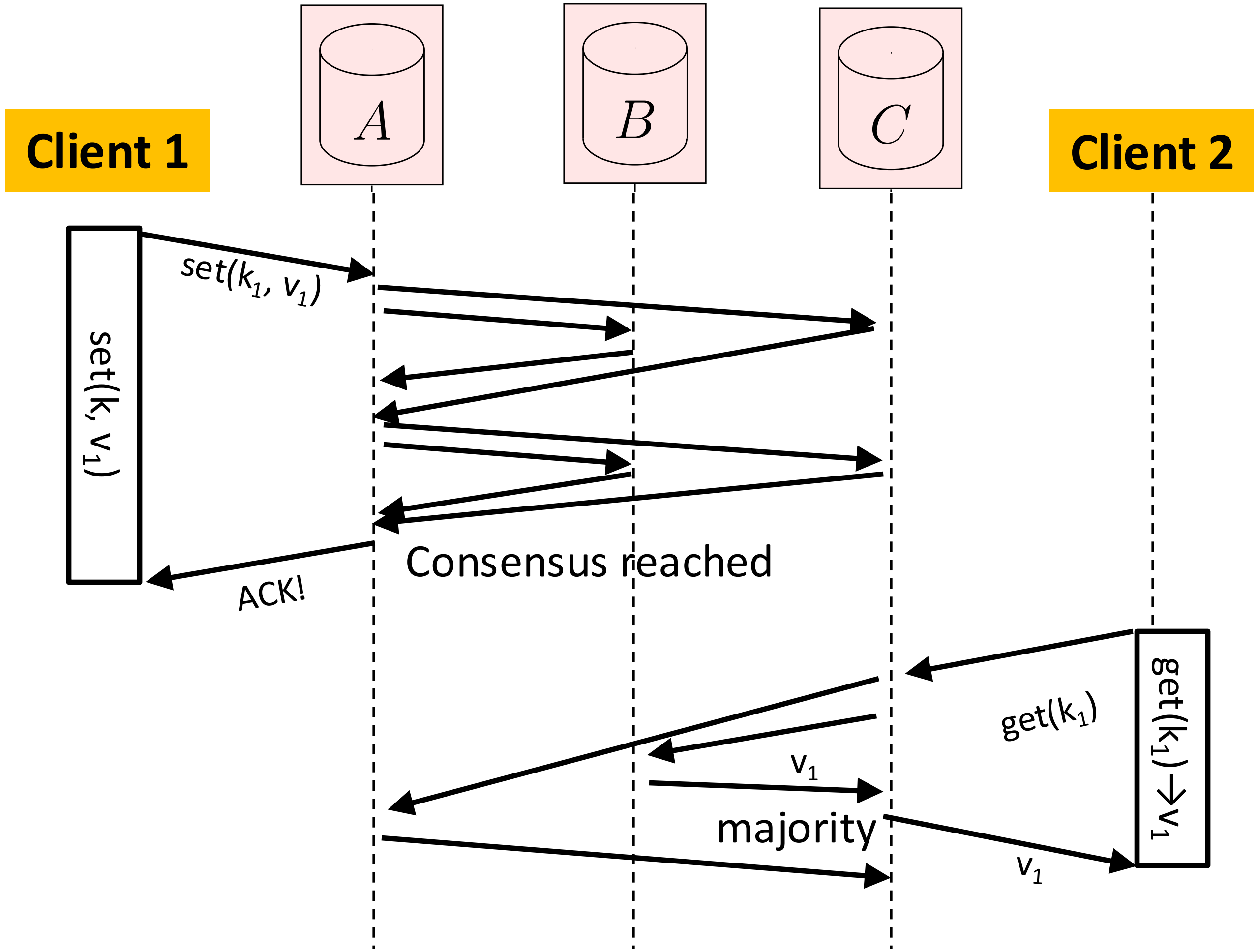


Agenda

- Time and ordering wrap-up
- **Eventual Consistency**

Consistency vs. Availability: The Trade-off During a Partition

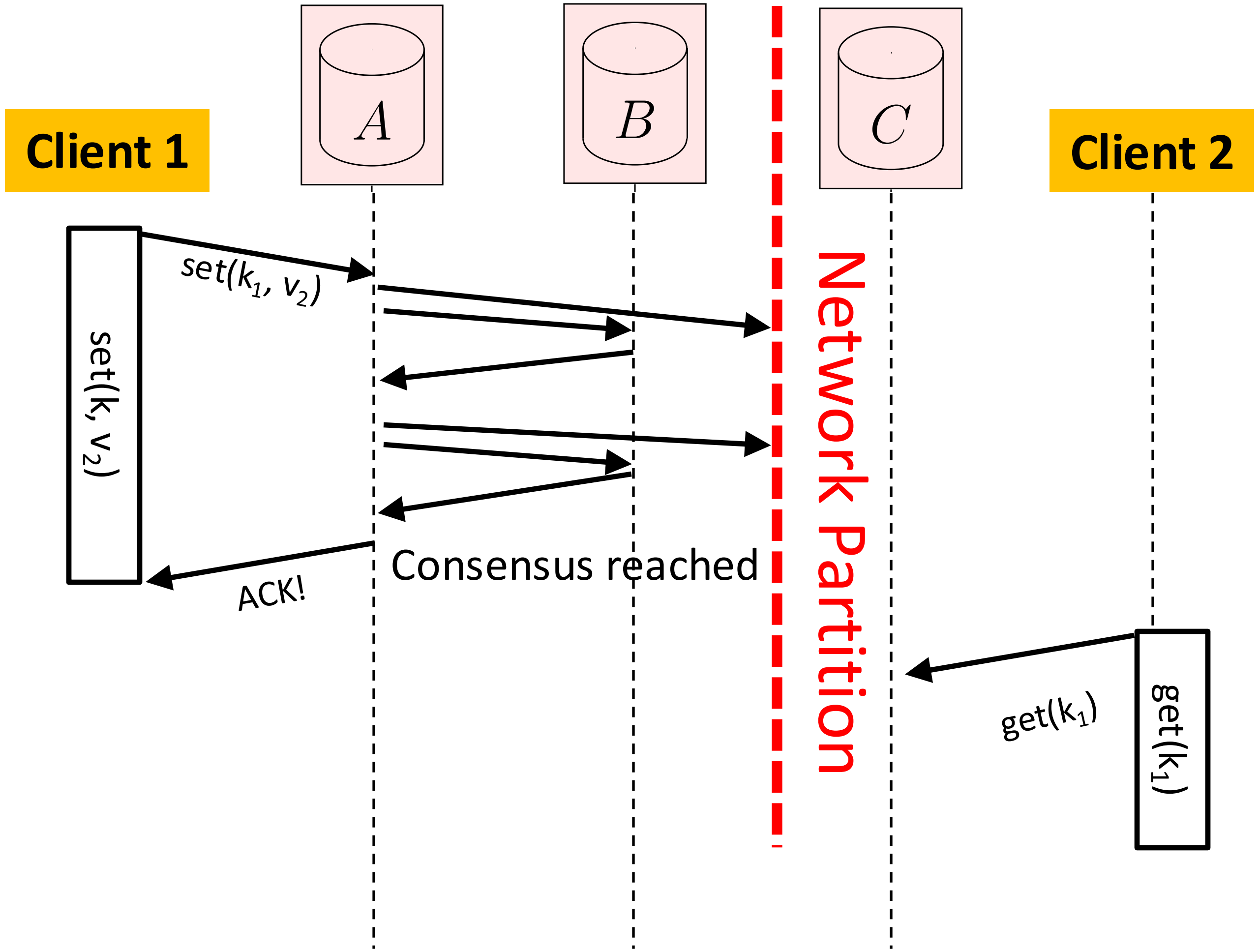
- **Set operation**: Achieves strong consistency by requiring consensus among all replicas.
- **Get operation**: Ensures consistency by querying a majority of replicas to retrieve the latest (up-to-date) state.



Consistency vs. Availability: The Trade-off During a Partition

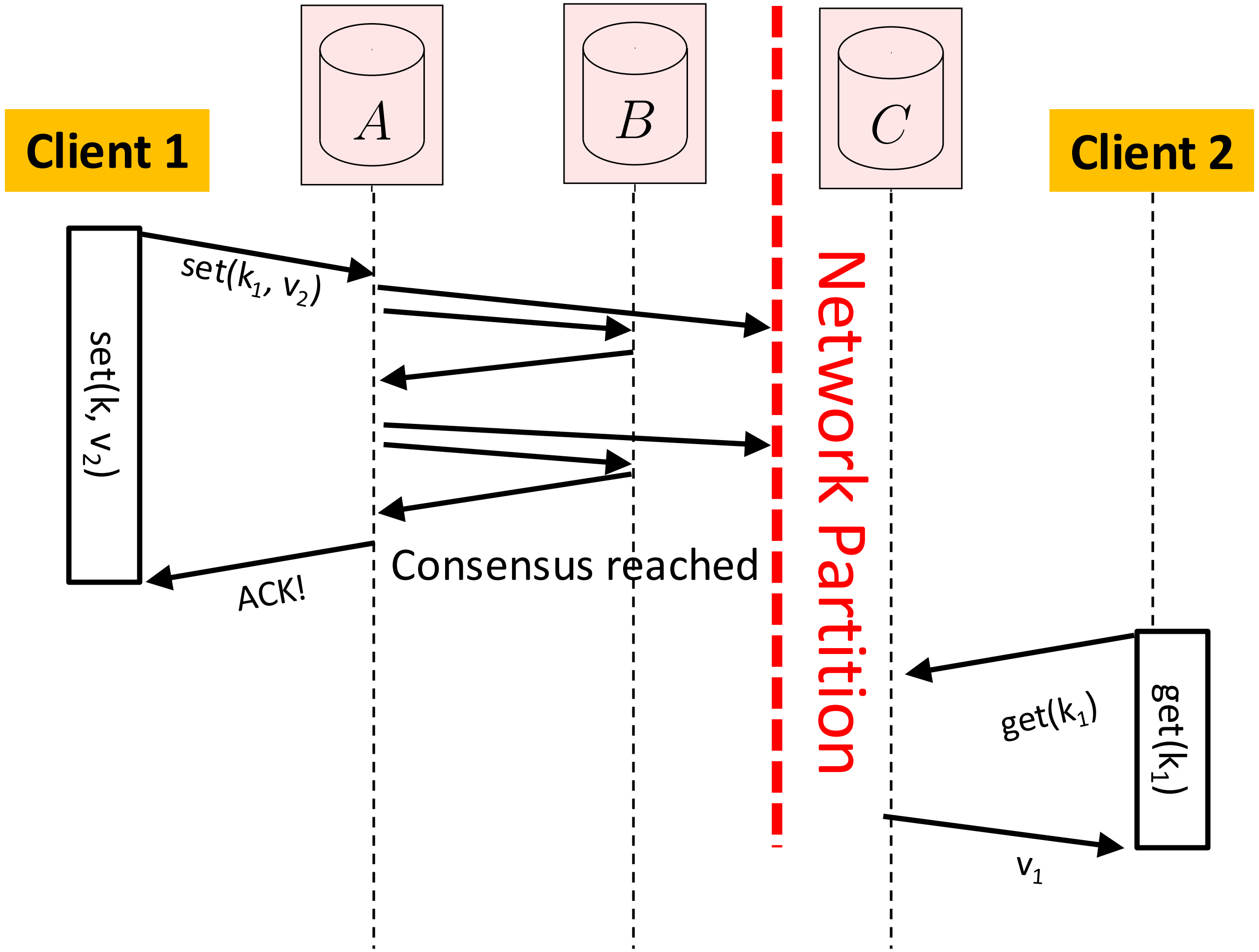
- **Scenario**: Replica C receives a get request while the other replicas are unreachable due to a network partition.

- **What should Replica C do upon receiving the get request?**
    - **Choice 1 – Maintain availability** (i.e., liveness): respond to the query, potentially sacrificing consistency
    - **Choice 2 – Maintain consistency**: suspends the operation, ensuring no inconsistent data is served, sacrificing availability



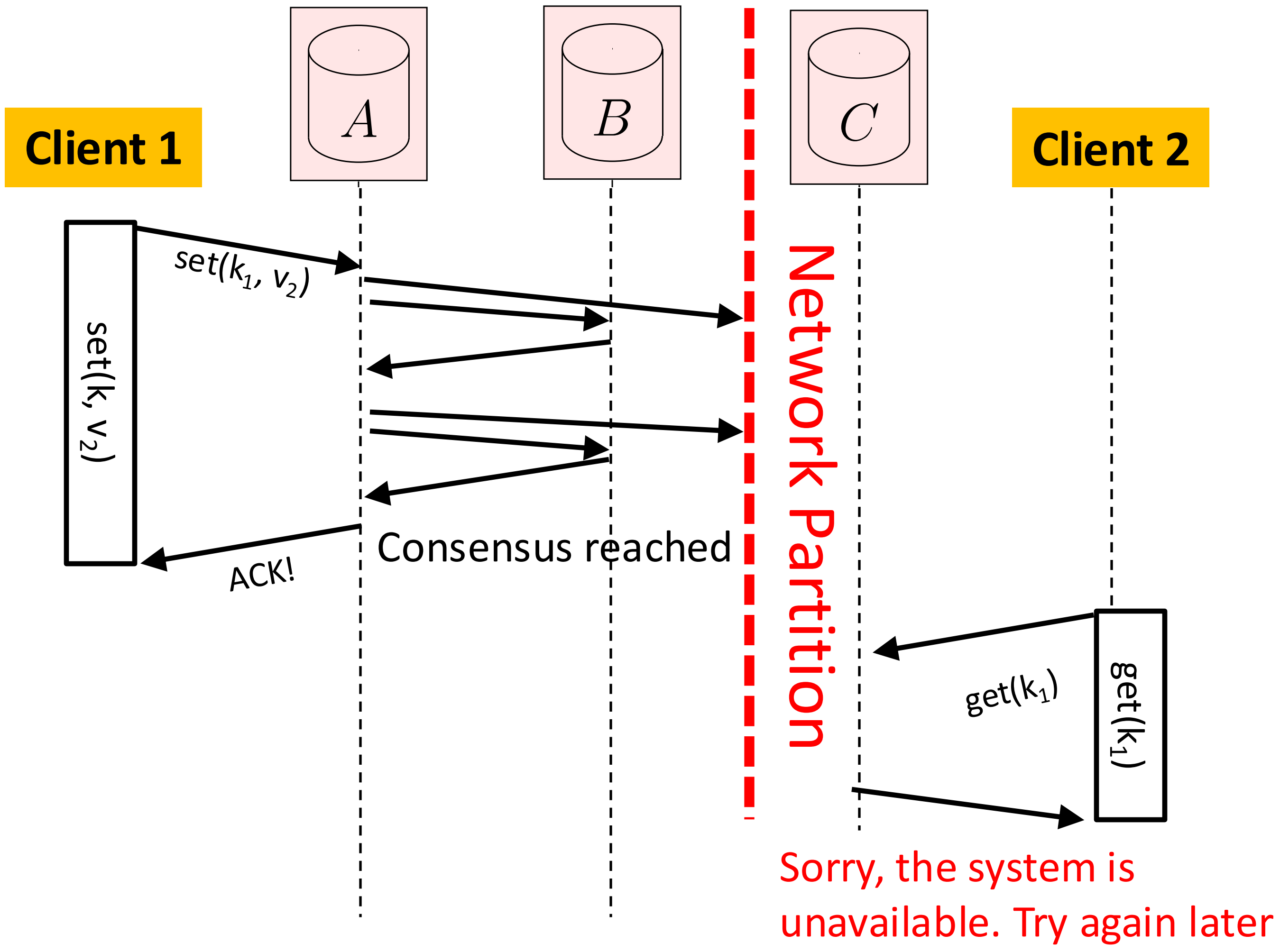
Choice 1: Availability

- **What should Replica C do upon receiving the get request?**
  - **Choice 1 – Maintain availability** (i.e., liveness): respond to the query
    - Replica C responds to the query using local state (v_1)



Choice 2: Consistency

- **What should Replica C do upon receiving the get request?**
  - **Choice 2 – Maintain consistency**: suspend and do not respond until the network is healed
    - Return an error message

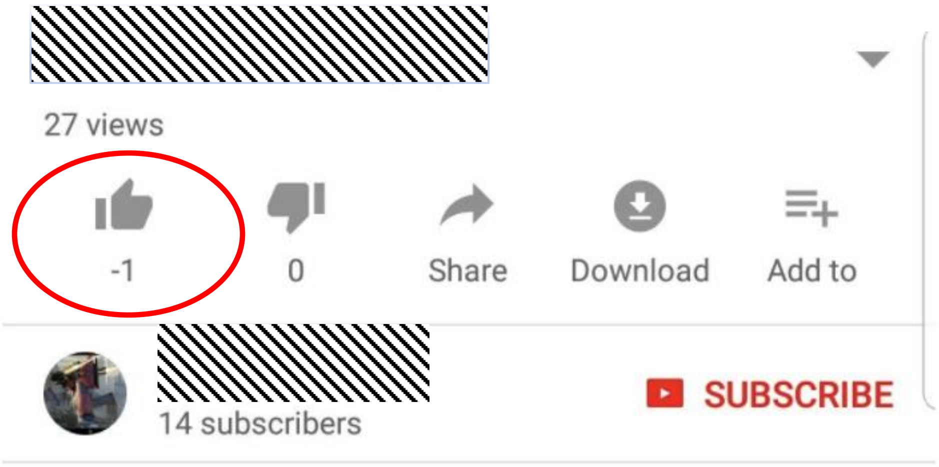


Brewer's CAP Theorem

- 3 fundamental requirements for distributed design:
  - ➔ Consistency: system has the same state everywhere
  - ➔ Availability: system operates if one node is responsive
  - ➔ Partition tolerance: system operates despite network interruptions (i.e. partition between nodes)
- **Theorem**: It is impossible for a distributed system to simultaneously provide all 3 guarantees. You must compromise.
- CAP becomes painfully noticeable as an application grows.
  - ➔ Need for scalability
  - ➔ Need for high availability
  - ➔ Network is unreliable

Brewer's CAP Theorem

- What to do?
  - ➔ Wait until all heals (sacrifice availability)
  - ➔ Deal with issues later (sacrifice consistency)
- In a distributed system, partitions are inevitable.
- Therefore, you must give up either availability or consistency.
- **Eventual consistency** is the compromise



Levels of consistency (1)

- Strong Consistency:
  - An update committed at one replica is propagated to all other (functioning) replicas before any subsequent operation occurs
  - A read operation always reflects the result of the most recent write operation
  - The system **behaves as though it has a single instance of the service**, with no visible replicas
  - Achieving strong consistency often involves significant overhead

Levels of consistency (2)

- Weaker consistency models can improve availability and performance compared to strong consistency
- Strong consistency is not ideal for all applications, especially those prioritising high availability or low latency
- A compromise is **"eventual consistency"**, where:
  - Updates executed at one replica will eventually propagate to all functioning replicas
  - The system guarantees that all replicas will converge to the same state over time, assuming no further updates
  - Temporary inconsistencies must be acceptable for applications that prioritise availability

Eventual consistency

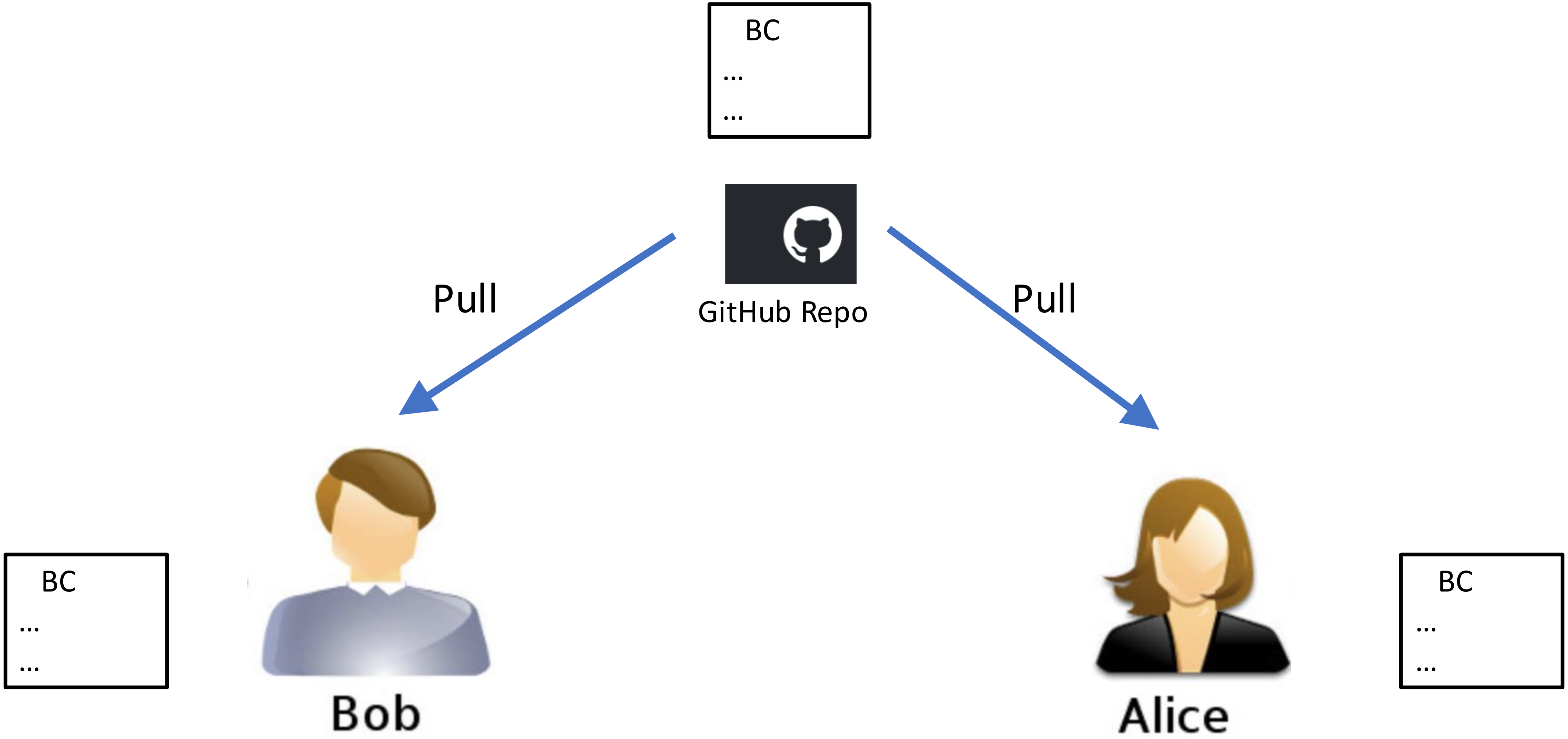
- **The order of updates** at each replica can differ, leading to temporary inconsistencies
- The system must handle **temporary inconsistencies** gracefully to ensure eventual consistency
- **Convergence rule**: Any two replicas that execute the **same set of updates** must arrive at the **same state**
- **Conflict resolution**:
  - Convergence requires the application to resolve **conflicts** arising from concurrent updates (e.g., concurrent writes to the same data)
  - Common mechanisms include **last-writer-wins**, **merge functions**, or **application-specific rules**
- Despite inconsistencies, replicas must **eventually converge**:
  - Certain events can be **locally ordered** (e.g., causal ordering) using tools like **vector clocks** or **Lamport timestamps**.
  - **Conflict resolution mechanisms** ensure replicas achieve a consistent final state

Conflicting operations

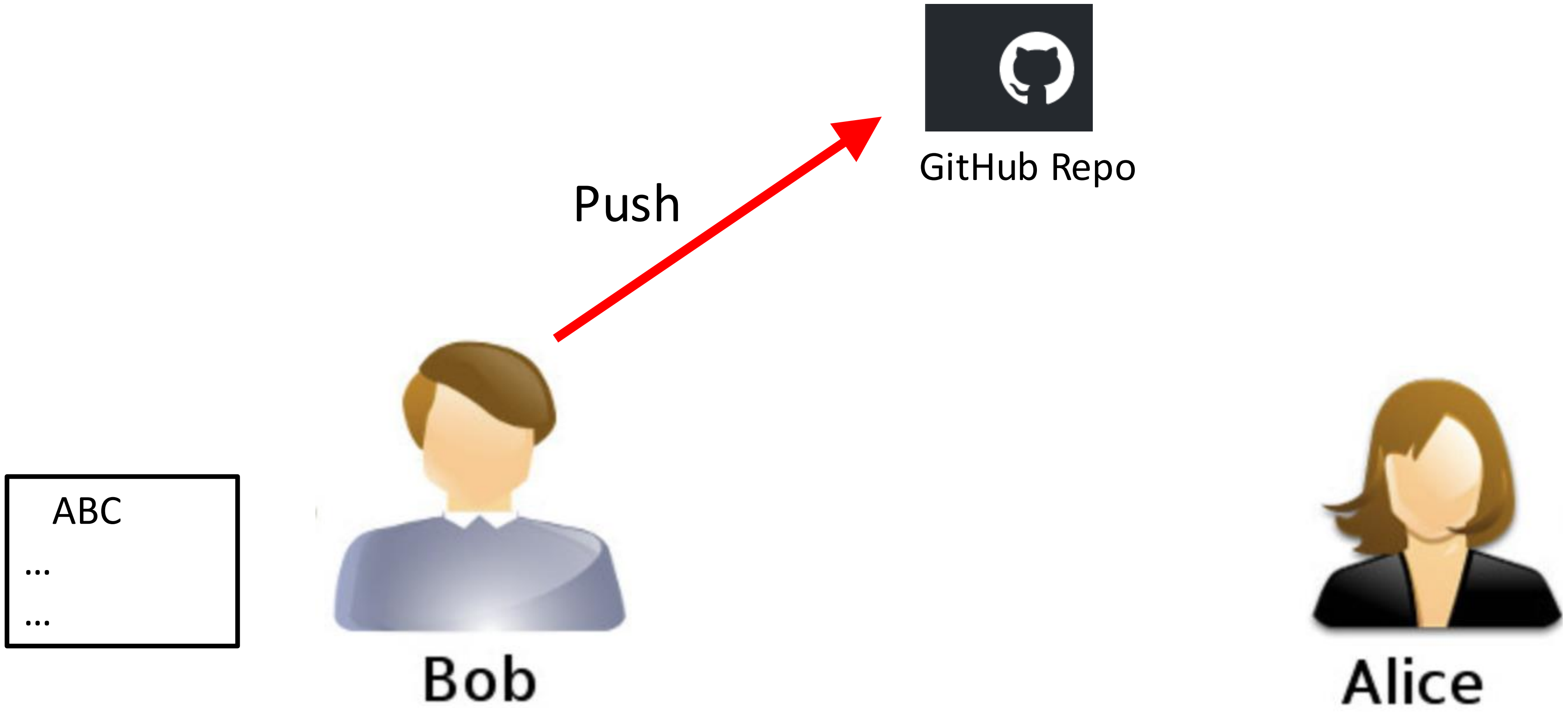
- With eventual consistency, the system must somehow resolve conflicting operations

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

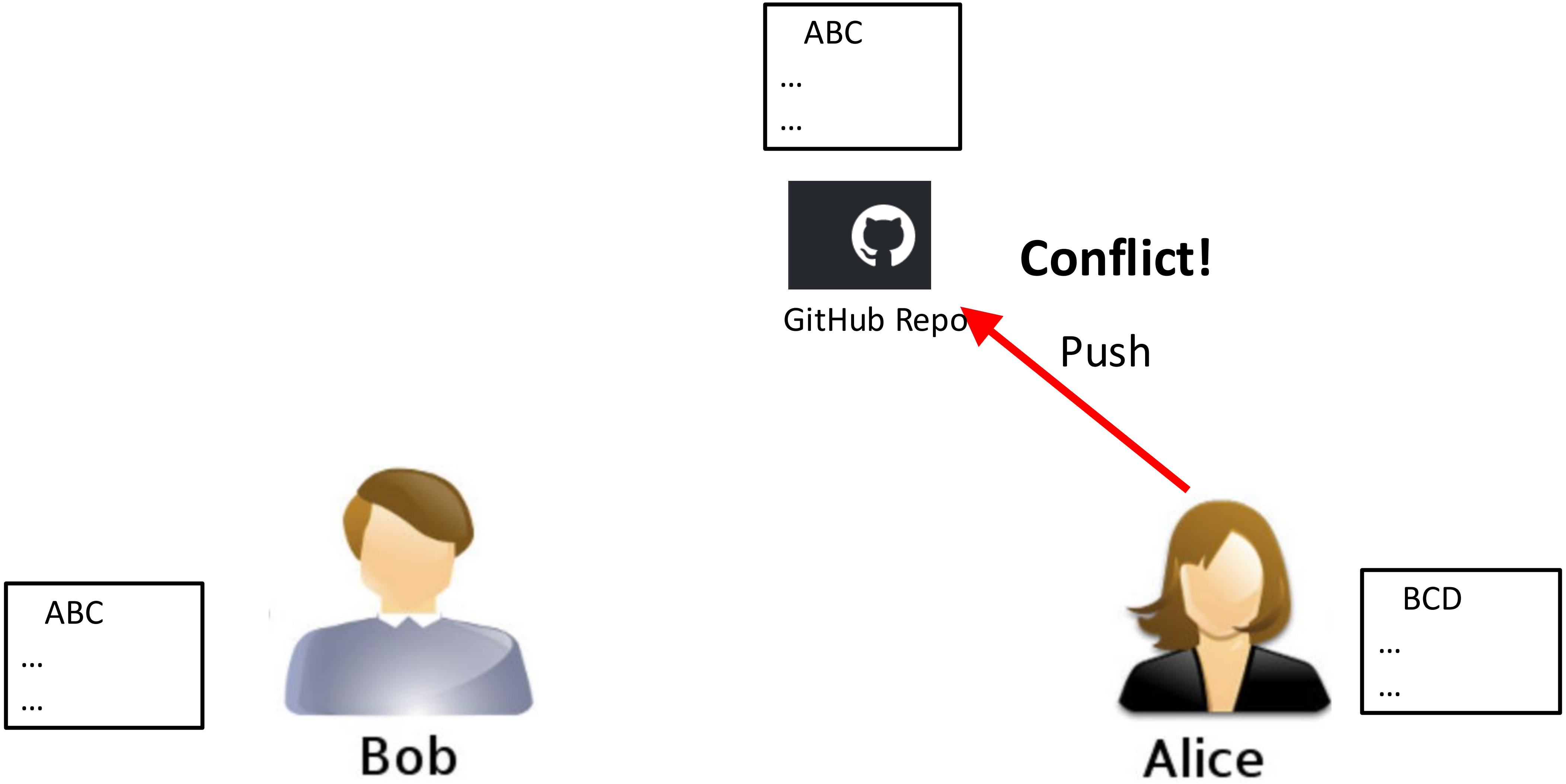
Example: Conflicting Operations



Example: Conflicting Operations



Example: Conflicting Operations



A Use-case for Eventual Consistency: Collaboration Software

- Multiple users sharing documents, spreadsheets, presentations, etc.,
  - Office365, Google Docs, Overleaf
- Note-taking apps
  - OneNote, Notion, Evernote, Slite, etc.
- Collaborating on a whiteboard
  - Miro, mural, MS whiteboard, etc.
- Shared calendars
  - Asana, Google calendar, etc.
- Collaboration is another example of **replication**
  - Each device where a document is opened is another replica
  - Any updates made to one replica must be sent to others

Resolving Conflicts

- A simple approach to resolve write-write conflicts is to use a “**Last Writer Wins**” policy
  - When two operations update to the same data, the system will apply both operations – one after the other
  - The second one (i.e. last) will overwrite the result of the first one
- We will ignore the read-write conflicts in the following discussion
  - Assume that such conflicts are tolerable by the application

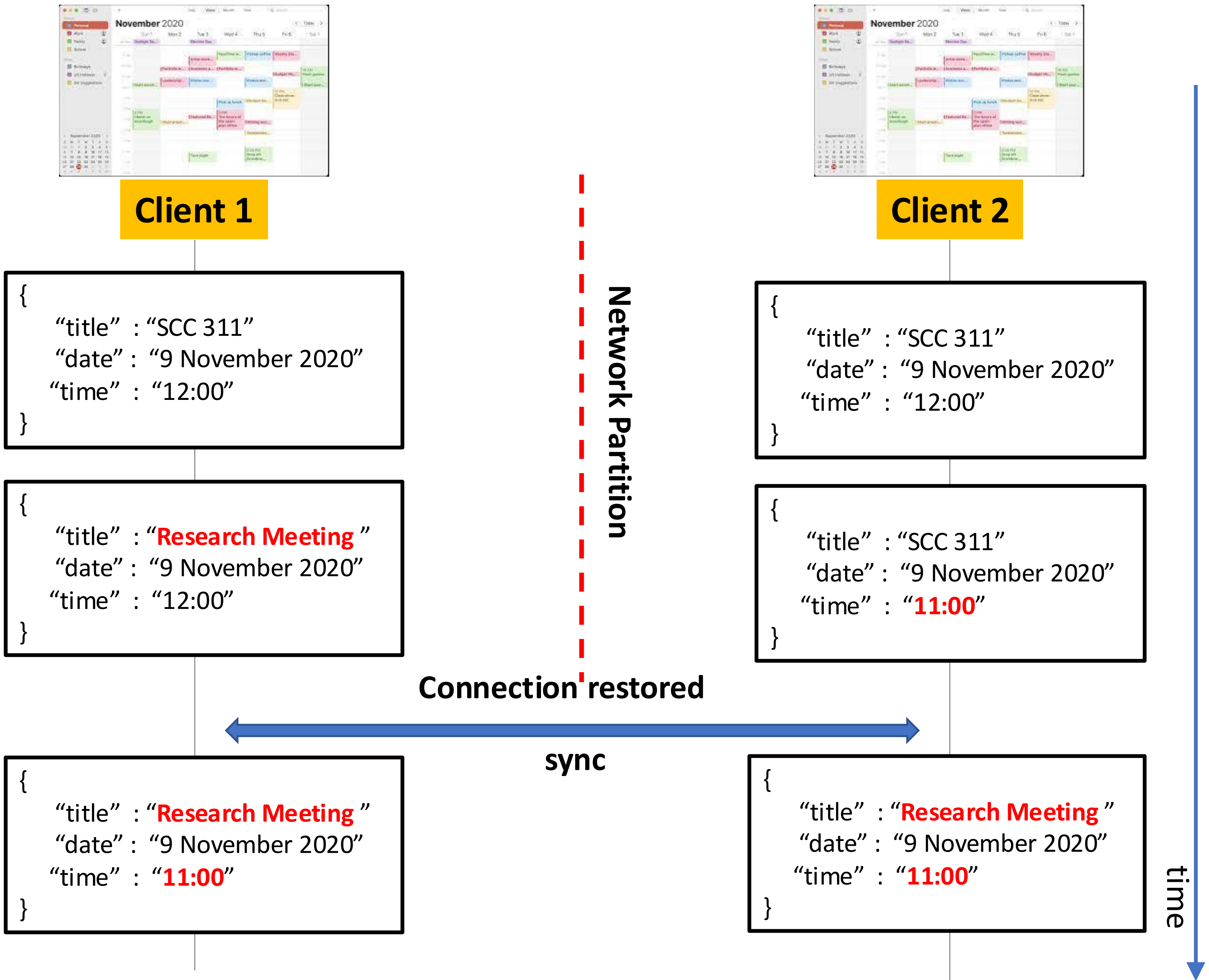
Resolving Conflicts

- Let's look at an example of a Map data structure

```
def set (k:key,  v: value):
    t  = newTimestamp()              # globally unique
    reliable_broadcast (t, k, v)     # send to everyone reliably
```

```
def onReceiveSetRequest (t, k, v):
    previousTime, value  = getLocal(k)
    if previousTime is None or previousTime < t:
        setLocal(k, v, t)
```

- Calendar App example with two client devices sharing a calendar
    - Each device is a replica
- Network gets partitioned
- Both clients make updates to the same calendar object
- Updates merged after network heals



- Calendar App example with two client devices sharing a calendar
  - Each device is a replica
- Network gets partitioned
- Both clients make updates to the same calendar object
- Updates merged after network heals

Client 1

```
{
    "title"  : "SCC 311"
    "date" :  "9 November 2020"
   "time"  :  "12:00"
}
```

```
{
    "title"  : "Research Meeting "
    "date" :  "9 November 2020"
   "time"  :  "12:00"
}
```

Network Partition

Client 2

```
{
    "title"  : "SCC 311"
    "date" :  "9 November 2020"
   "time"  :  "12:00"
}
```

```
{
    "title"  : "Staff Meeting"
    "date" :  "9 November 2020"
   "time"  :  "12:00"
}
```

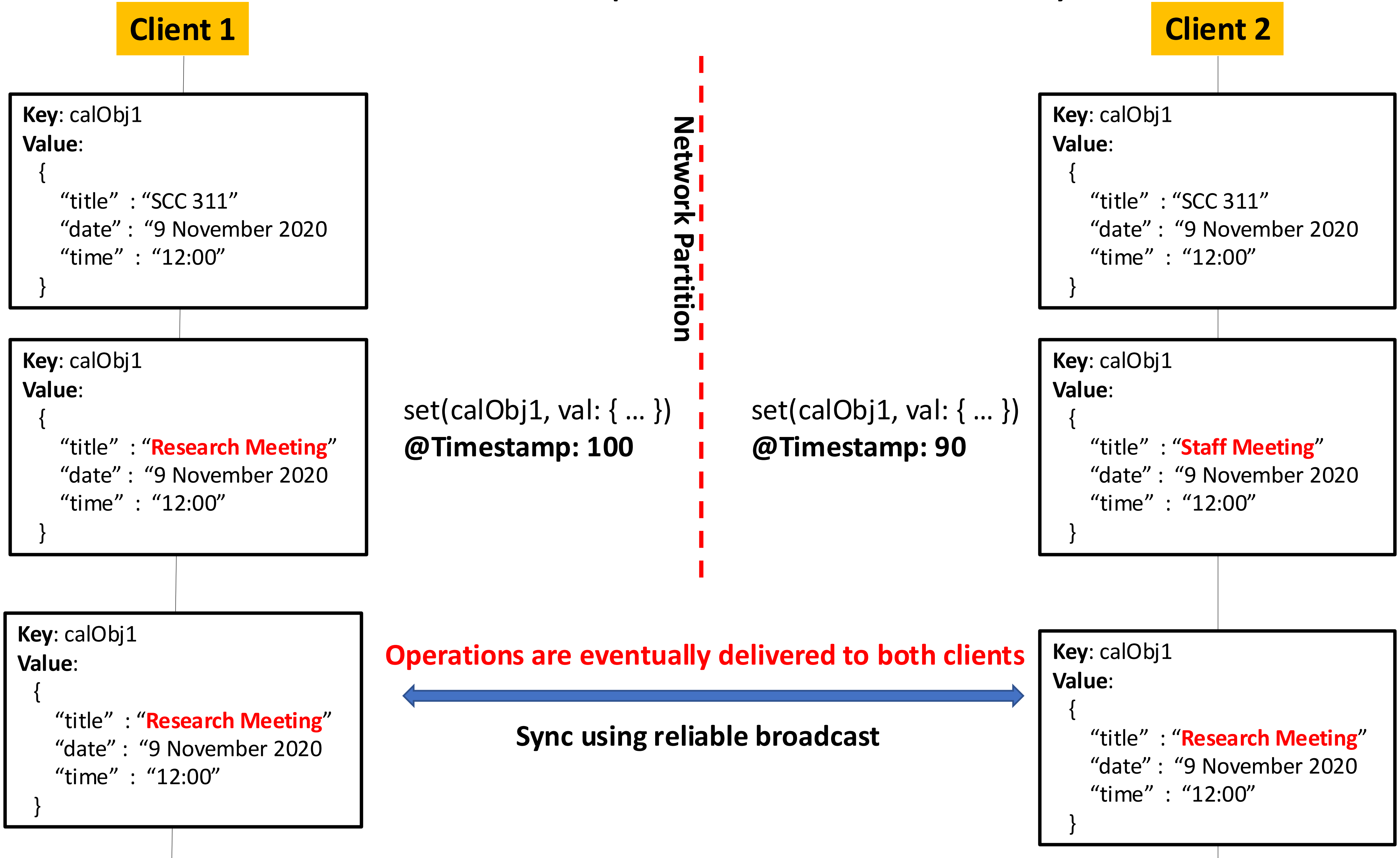
Connection restored

Sync

Conflicting titles!

time

Introduce Timestamps and the “Last Writer Wins” Policy



Resolving Conflicts

- Instead of broadcasting updates, replicas can also broadcast their state (after applying updates)

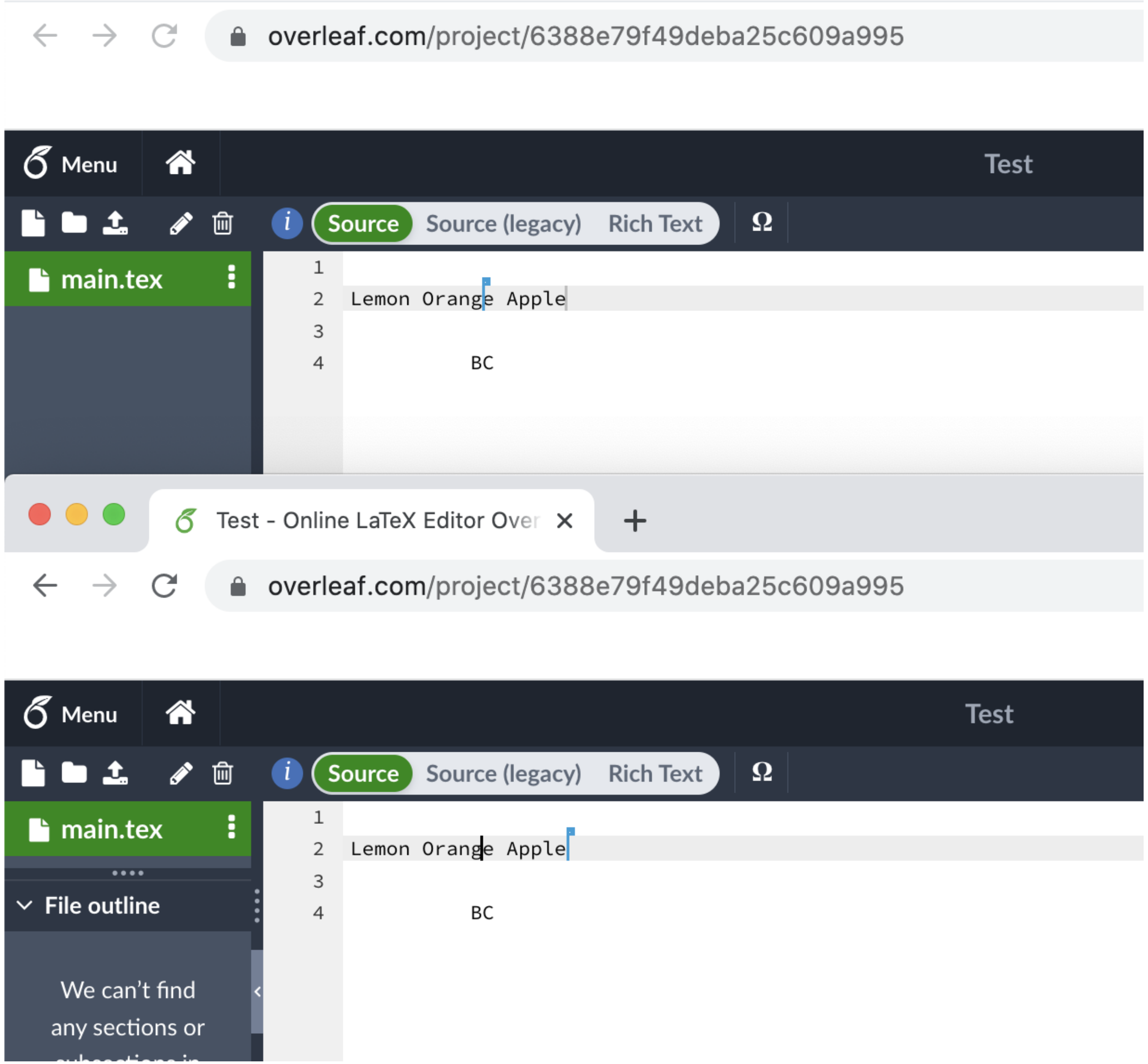
```
def set (k:key,  v: value):
    t  = newTimestamp()                          # globally unique
    localState[k] = (“value”: v, “time”: t)      # update local state
    best_effort_broadcast(localState)            # send to everyone

def onReceiveSetRequest (state):                 # merge – local-state ∪ remote- state
    for time, key, val in state.pairs:
        if key in localState.keys():
            previousTime = localState.get(k)[time]
            if previousTime < time:
                setLocal(k, v, t)
        else:
            localState.append({time,key,val)
```

Resolving Conflicts via State Update

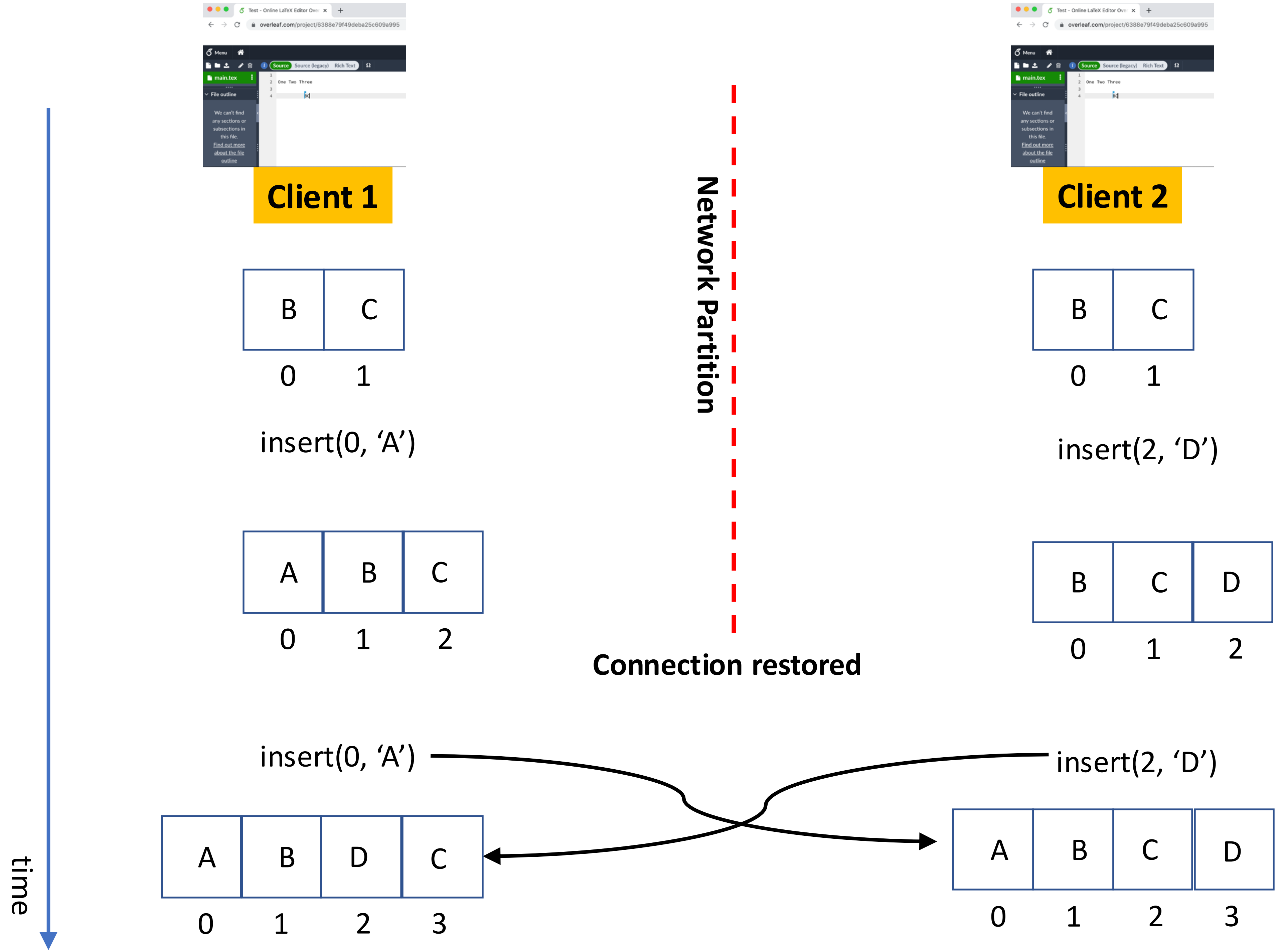
- Broadcast messages can be large
- It can tolerate loss of messages
  - A replica can become up-to-date using the latest broadcast messages
    - Hence, the use of a lightweight, best effort broadcast (instead of the more heavy-weight, reliable broadcast)
- State merge function must be:
  - Commutative: $s1 \cup s2 = s2 \cup s1$.
  - Associative: $(s1 \cup s2) \cup s3 = s1 \cup (s2 \cup s3)$

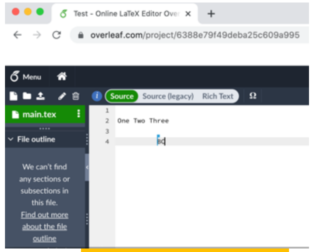
Example Application: Collaborative Text Editing



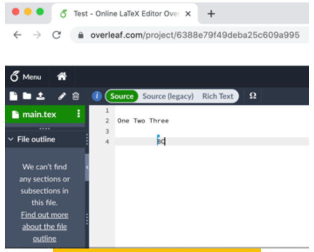
Example Application: Collaborative Text Editing

- The key strokes immediately apply to the local **view** (i.e., to the local replica) as you type
- Copies of the documents temporarily differ from each other
- In case of a network partition, the copies can diverge significantly
- Eventually all replicas (server or user copies) must converge to same view
- Must deal with concurrent updates and the conflicts that results from those


Client 1
Client 2
Network Partition
B C
0 1
insert(0, 'A')
A B C
0 1 2
B C
0 1
insert(2, 'D')
B C D
0 1 2
Connection restored
insert(0, 'A')
insert(2, 'D')
A B D C
0 1 2 3
A B C D
0 1 2 3
time



Client 1



Client 2

insert(0, ‘A’)

Operational Transformation

insert(2, ‘D’)

Server

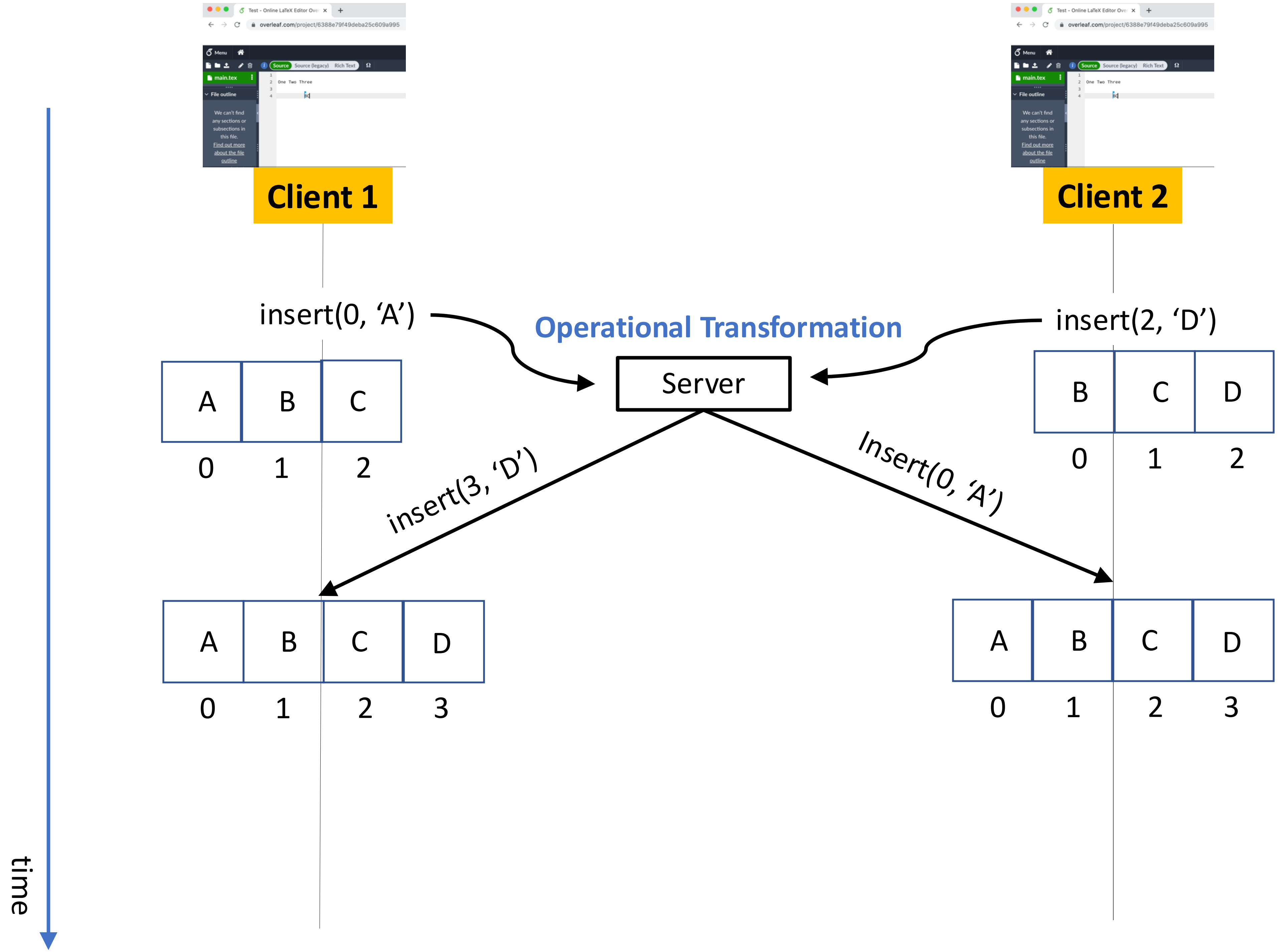
Transform (insert(0, ‘A’), insert(2, ‘D’))
→ insert(3, ‘D’)

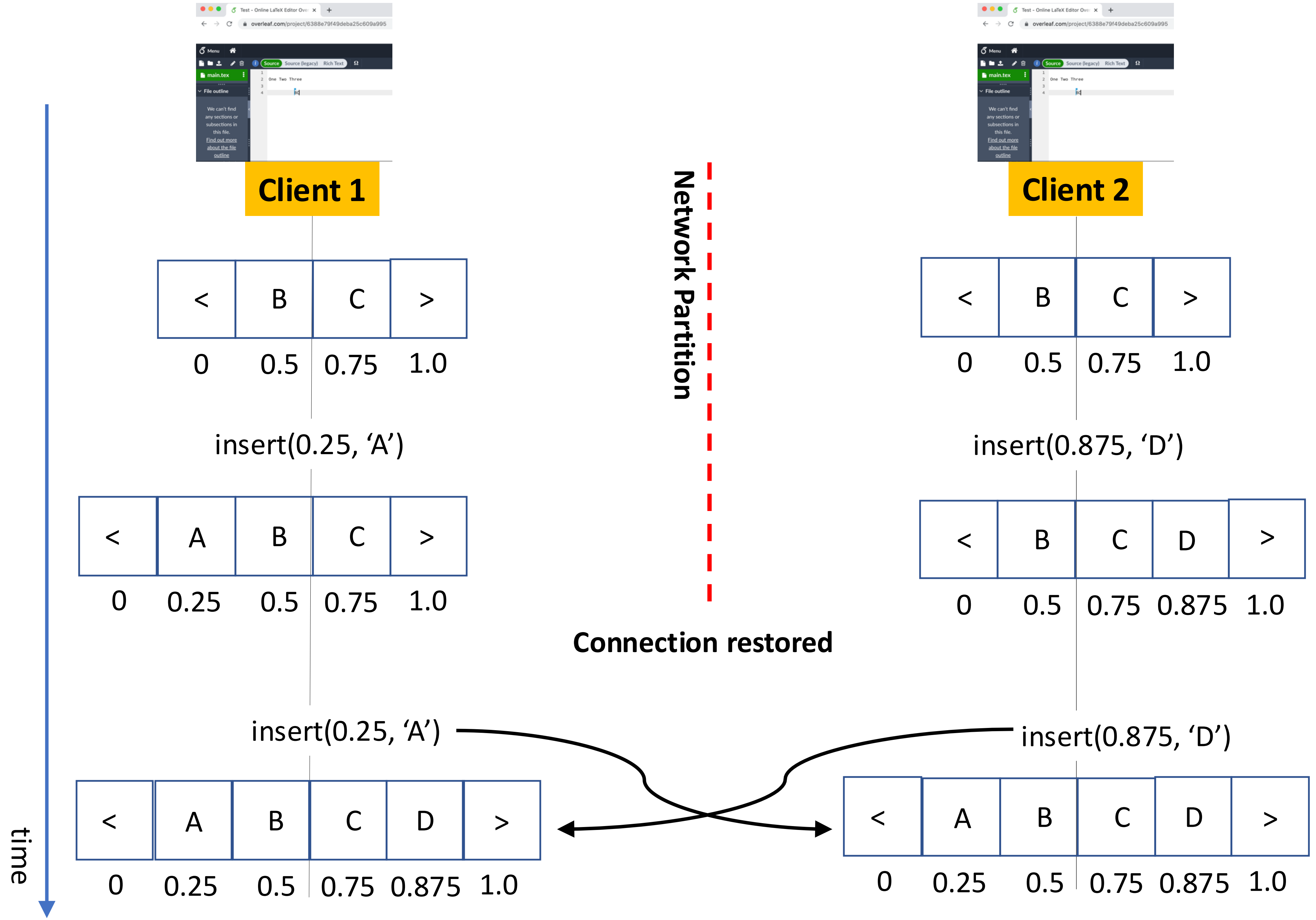
Transform (insert(2, ‘D’), insert(0, ‘A’))
→ insert(0, ‘A’)

A	B	C	D
0	1	2	3

A	B	C	D
0	1	2	3

time


Client 1
Client 2
insert(0, 'A')
Operational Transformation
insert(2, 'D')
Server
A B C
0 1 2
B C D
0 1 2
insert(3, 'D')
Insert(0, 'A')
A B C D
0 1 2 3
A B C D
0 1 2 3
time


Client 1
Client 2
Network Partition
< B C >
0 0.5 0.75 1.0
< B C >
0 0.5 0.75 1.0
insert(0.25, ‘A’)
insert(0.875, ‘D’)
< A B C >
0 0.25 0.5 0.75 1.0
< B C D >
0 0.5 0.75 0.875 1.0
Connection restored
insert(0.25, ‘A’)
insert(0.875, ‘D’)
< A B C D >
0 0.25 0.5 0.75 0.875 1.0
< A B C D >
0 0.25 0.5 0.75 0.875 1.0
time

Further reading

- Kleppmann, et al. 2019, October. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 154-178).
- Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee: "Replicated abstract data types: Building blocks for collaborative applications," Journal of Parallel and Distributed Computing, 71(3):354–368, 2011.