

Deep Dive:
Logical Clocks: from Lamport to Bloom Clocks

Is this plausible?



At 8:30am, Susie receives her shipping order



At 9:45am, John drives to Susie's house to deliver her shipping order

Assume these events happen on the same day! 😁

At 10:23am, Martha dispatches Susie's shipping order



Is this plausible?



At 8:30am, Susie receives her shipping order



At 9:45am, John drives to Susie's house to deliver her shipping order

Assume these events happen on the same day! 😁

The Future cannot influence the Past!

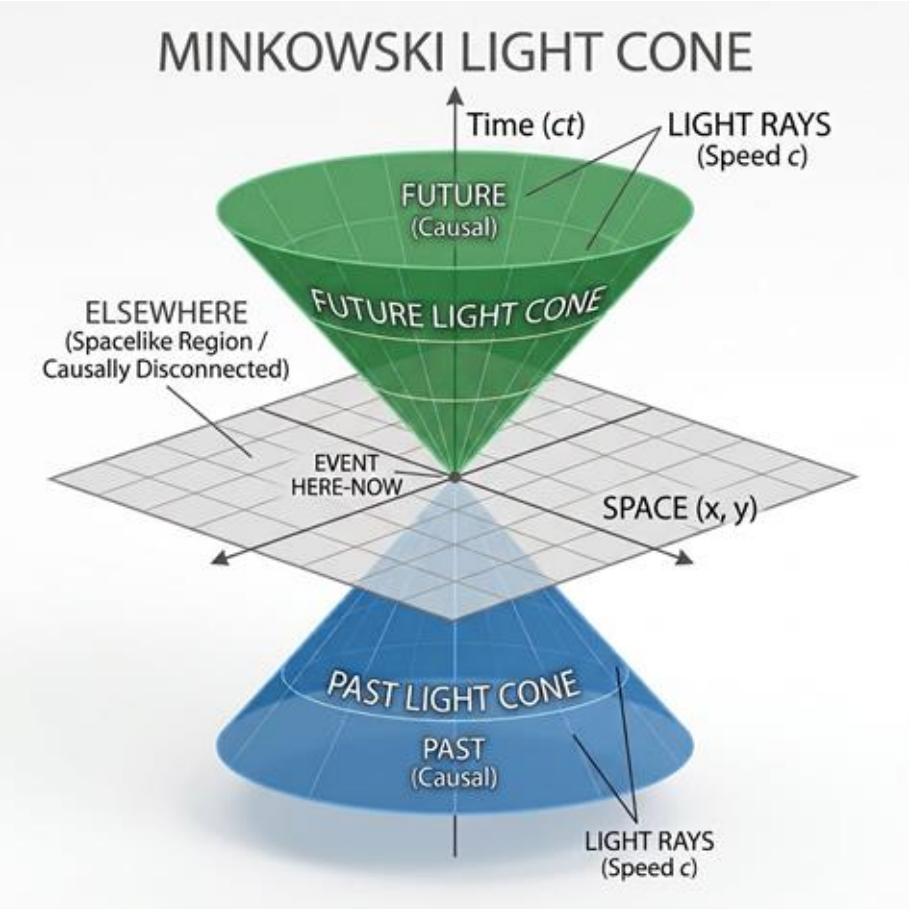
At 10:23am, Martha dispatches Susie's shipping order



Meanwhile in the online world…



Ana's friend never received her message saying that she doesn't like Pepperoni. 😢

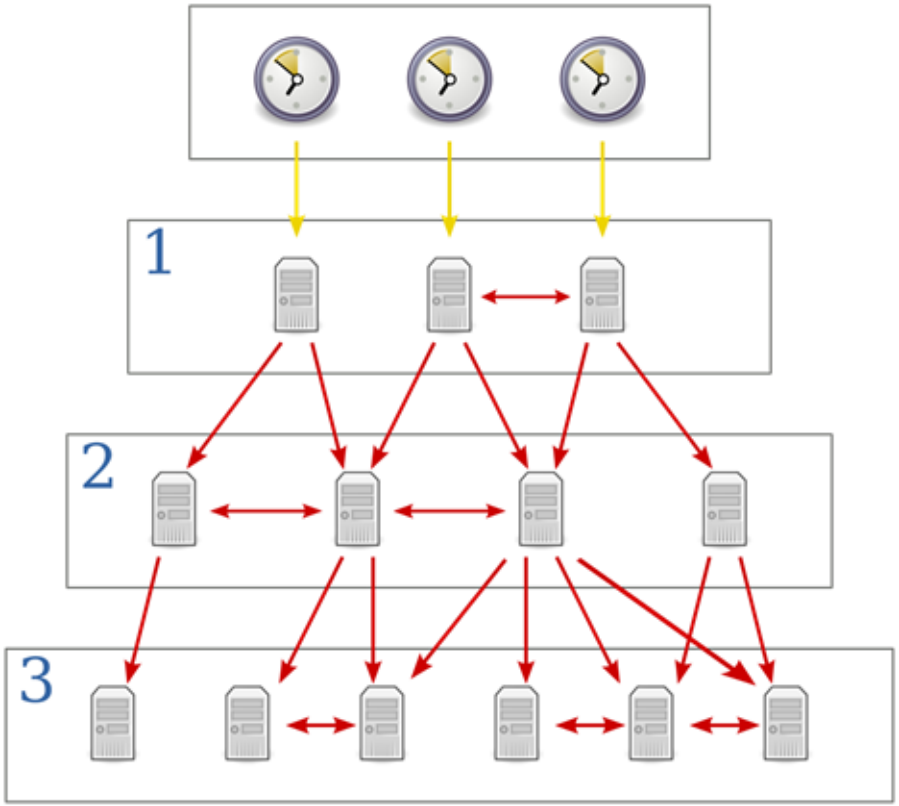


- A geometric representation of spacetime fabric where events unfold;
- ***Every event*** in space-time has a light cone that divides the universe into two regions: the future and the past;
- **Future cone:** all the events that ***can be*** influenced by the event at the center;
- **Past cone:** all the events that ***could have*** influenced the event at the center;
- ***Events outside the light cone cannot be causally connected to the event at the center;***

Physical Clocks

- Physical Clocks are a discrete approximation of continuous time;
- **Skew (offset):** two clocks disagree at the same instant (A reads 10:00:00, B reads 10:00:05);
- **Drift:** the rate at which a clock gains or loses time (temperature, voltage, aging crystals), drift makes skew grow;
- **Network latency and jitter** make reading a remote clock inherently uncertain;
- **Time zones and representation** complicate interpretation across regions (store UTC);

Network Time Protocol (NTP)

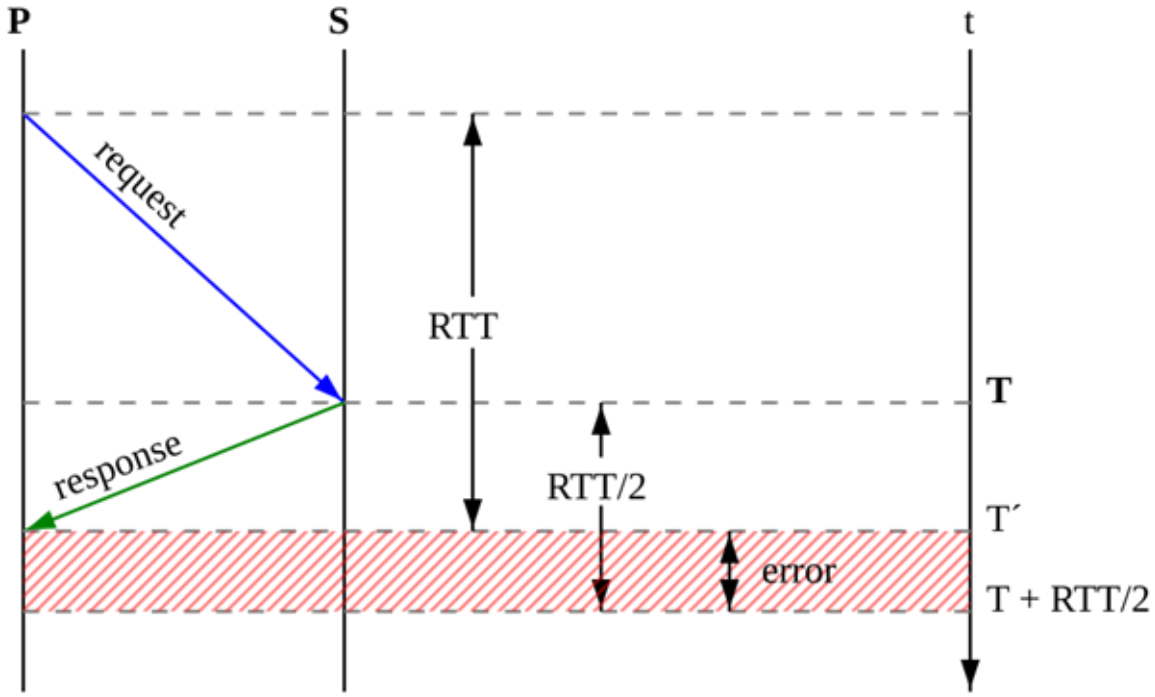


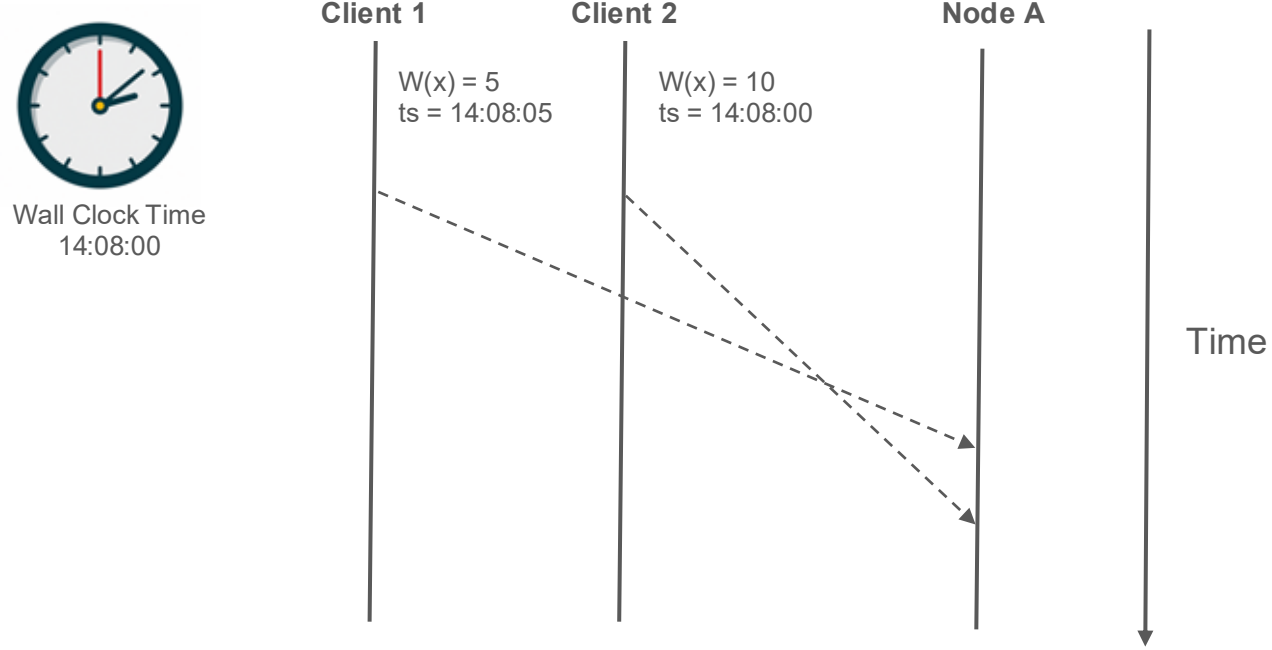
- NTP synchronizes clocks over the network through a hierarchy of **strata;**
- **Stratum 0:** reference clocks (atomic, GPS);
- **Stratum 1:** servers directly attached to a stratum-0 source;
- **Stratum 2, 3, …:** synchronize from the stratum above;
- Current version: NTPv4 (RFC 5905);
- Good sync reduces skew, but it never eliminates it;

Cristian’s algorithm

- A simple method for synchronizing a client’s local clock with a single, highly accurate time server (client-server model);
- Client periodically polls the time server for its time *T*;
- Client measures the round-trip time RTT and sets its clock to $T + \frac{RTT}{2}$**;**
- Assumes the server is accurate and network delay is **symmetric;**
- Error is bounded by about $\pm\frac{\mathrm{RTT}}{2}$, asymmetry and jitter remain;

Cristian’s algorithm





- Both writes are issued at the same real time; Client 1's clock is 5 s fast, and Client 2's write arrives last;
- **Last-arrival wins:** $x = 10$ (Client 2); decided by network delay;
- **Last-write-wins by timestamp:** $x = 5$ (Client 1); decided by a skewed clock; Client 2's update is silently lost;
- The writes are **concurrent**: neither policy can say which is "right", it just picks one;

The system model

- Nodes share no memory and no global clock; they communicate only by **message passing;**
- Messages can be **delayed**, **reordered**, **duplicated**, or **dropped;**
- Networks partition;
- Network latency and jitter add variability;

Causality

Events

*Events are the fundamental entities that represent a **discrete moment when the state changes or something occurs** within a system;*

Properties:

1. *Atomic actions;*
2. *Instantaneous;*
3. *All events are irreflexive:* $\forall a \in E: \neg(a \to a)$*;*
4. *Immutable;*
5. *Happen at one specific point in time (we can assign a timestamp);*
6. *Usually asynchronous;*

Causality

- Determining the order of events is a fundamental problem in distributed systems;
- ***Causality is the relationship between events where one event (the cause) influences or leads to another event (the effect);***
- Causality cannot form loops: $a \rightarrow b \Longrightarrow \neg(b \rightarrow a)$ (the future cannot influence the past);
- *The events can be partially ordered by the causality relation (→);*

Totally ordered set vs Partially ordered set

In a **totally ordered set**, any two elements can always be compared: $\forall a, b: a \leq b \vee b \leq a$;

e.g.: {1, 3, 2, 7} under ≤ (any two numbers can be compared);

In a **partially ordered set (poset)**, there may be elements a and b where neither a ≤ b nor b ≤ a, i.e., $\exists a, b: \neg(a \leq b) \wedge \neg(b \leq a)$. When this happens, we say the elements are independent, or concurrent, with respect to their ordering;

Happened-before Relation

- **Setup:**
  - E is the set of all events;
  - $p(e)$ is the process where event e happens;
  - $<_i$ is the local order of events within process P_i (a process is sequential, so this order is total);
  - M is the set of messages; for each $m \in M$, $\text{send}(m)$ and $\text{recv}(m)$ are its send and receive events;
- The happened-before relation ($\rightarrow$), defined by Lamport, is the smallest relation that satisfies:
  - **Same process (direct causal relationship):** $\forall a, b \in E: p(a) = p(b) = P_i \wedge a <_i b \Longrightarrow a \rightarrow b$;
  - **Sender and receiver:** $\forall m \in M: \text{send}(m) \rightarrow \text{recv}(m)$, i.e., if $a = \text{send}(m)$ and $b = \text{recv}(m)$ for some m, then $a \rightarrow b$;
  - **Transitivity:** $\forall a, b, c \in E: (a \rightarrow b \wedge b \rightarrow c) \Longrightarrow a \rightarrow c$;
- It does not depend on any physical clock;

Causal Path

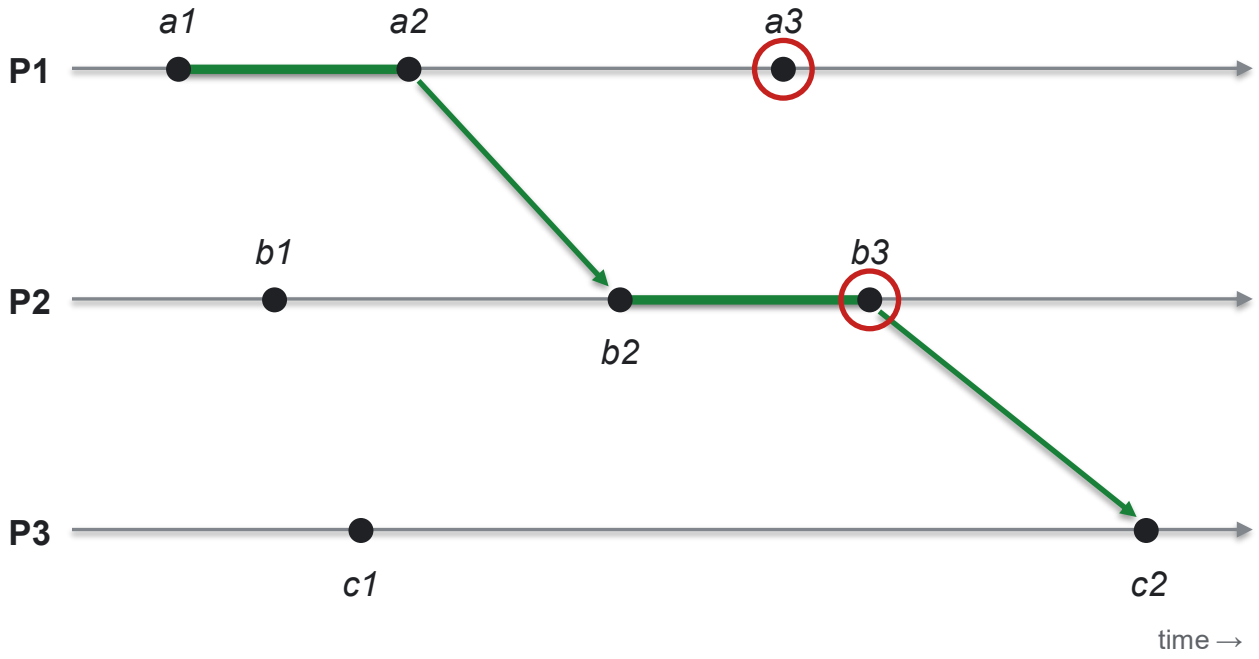
A causal path is a sequence of events connected by causal dependencies, where each event in the sequence influences, directly or indirectly, the next one;

As happened-before is a strict partial order, causal paths cannot form cycles (the future cannot influence the past);

That is, an event can only be reached from another event in one direction;

$a \rightarrow b$ holds exactly when there is a causal path from a to b: each step is either local order or a message, and transitivity chains the steps;

One execution, three processes



Causal path

a1 → a2 → b2 → b3 → c2

so a1 → c2 (transitivity)

Concurrent

there is no path between a3 and b3 in either direction: $\neg(a3 \rightarrow b3) \wedge \neg(b3 \rightarrow a3) \Longrightarrow a3 \parallel b3$

We reuse this execution for every clock in the lecture;

Concurrent Events

- Because events are only partially ordered, two events may not be comparable when there is no causal path between them. We call them **concurrent** events (that is, the happened-before relation cannot be established!);
- $E \parallel F \iff \neg(E \rightarrow F) \wedge \neg(F \rightarrow E)$: neither event happened before the other;
- Concurrent does not mean simultaneous: it means that neither event could have influenced the other;
- Any total order of events (e.g., ordering by timestamp) must invent an order between concurrent events;

Causal Histories

- A causal history is a set* of unique event identifiers;
- Give every event a unique name: **process id + local counter** (a1, a2, b1, …);
- The **causal history H(e)** is the set of all events that precede-or-equal e: $H(e) = \{e' \in E | e' \rightarrow e\} \cup \{e\}$;
- Simple, exact, and the mental model for every clock today;
- **Causal histories are a simple way of tracking causality;**

Causal Histories

Figure 2. Causal histories.

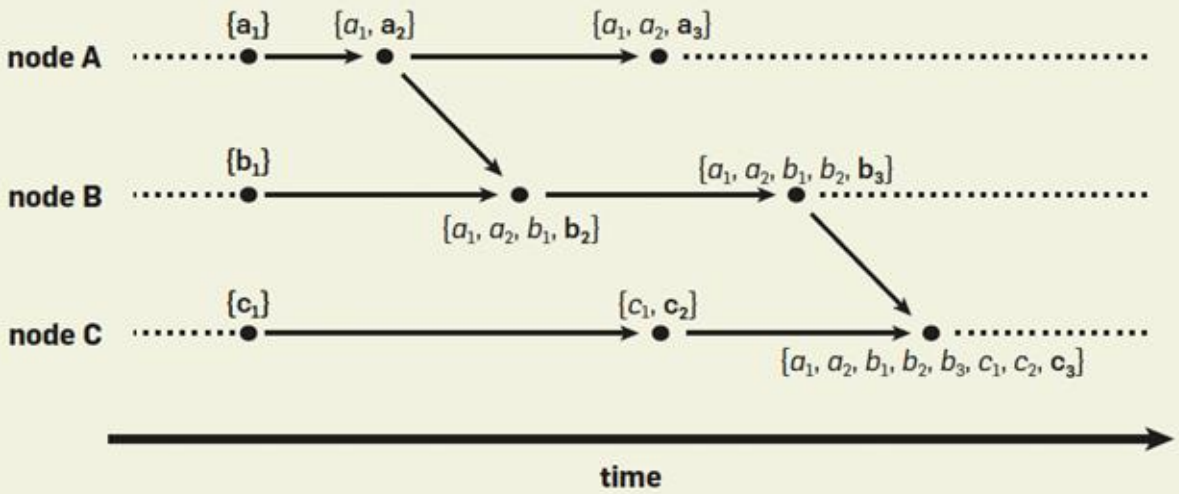


Figure: Baquero & Preguiça "Why Logical Clocks Are Easy" CACM 59(4) 2016

Causal Histories in pseudocode

Algorithm 1: Causal History

Data: Node N_i with history $H_i \leftarrow \emptyset$ **and** event counter $n_i \leftarrow 0$

function Tick()
- $n_i \leftarrow n_i + 1$
- $H_i \leftarrow H_i \cup \{(i, n_i)\}$ *// name the new event*
- **return** H_i

function LocalEvent()
- **return** Tick()

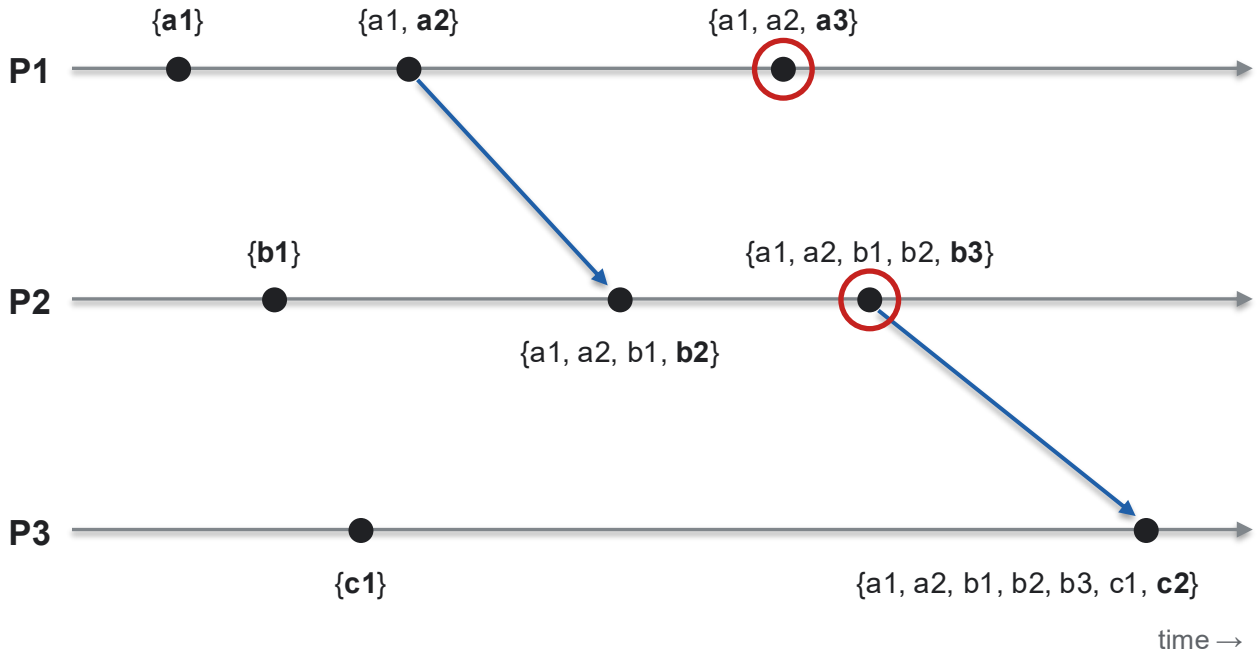
function Send(m, N_j)
- $t \leftarrow$ Tick()
- transmit $\langle m, t \rangle$ to N_j
- **return** t

function Receive($\langle m, H_M \rangle$)
- $H_i \leftarrow H_i \cup H_M$ *// merge*
- $t \leftarrow$ Tick()
- deliver m
- **return** t

function HappenedBefore(x, H_y) *// $x \rightarrow y, x \neq y$*
- **return** $x \in H_y$

Notation: ← assignment · ⟨m, t⟩ message m carrying timestamp t · transmit / deliver are network primitives · each function runs atomically at its node

Testing causality with histories



$x \rightarrow y \Leftrightarrow x \in H(y) \wedge x \neq y$

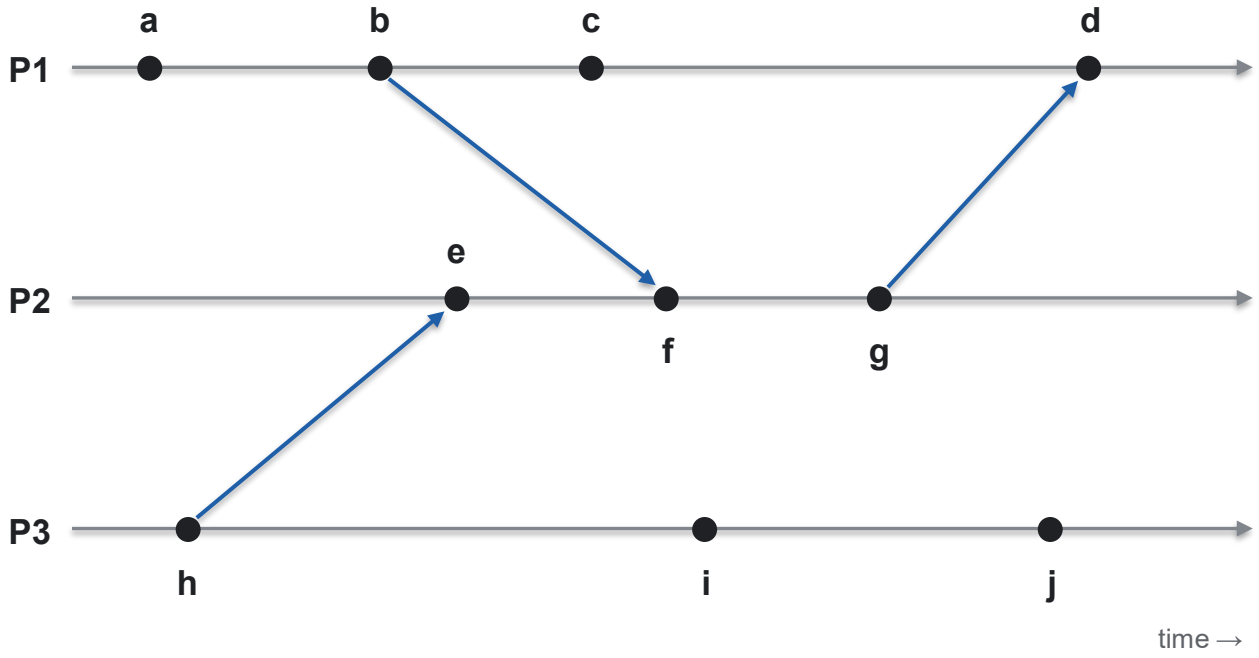
(equivalently H(x) ⊂ H(y), strict)

$a2 \in H(b3) \Rightarrow a2 \rightarrow b3$

$a3 \notin H(b3) \wedge b3 \notin H(a3) \Longrightarrow a3 \parallel b3$

Exact, but H grows with every event;

Exercise 1: happened-before

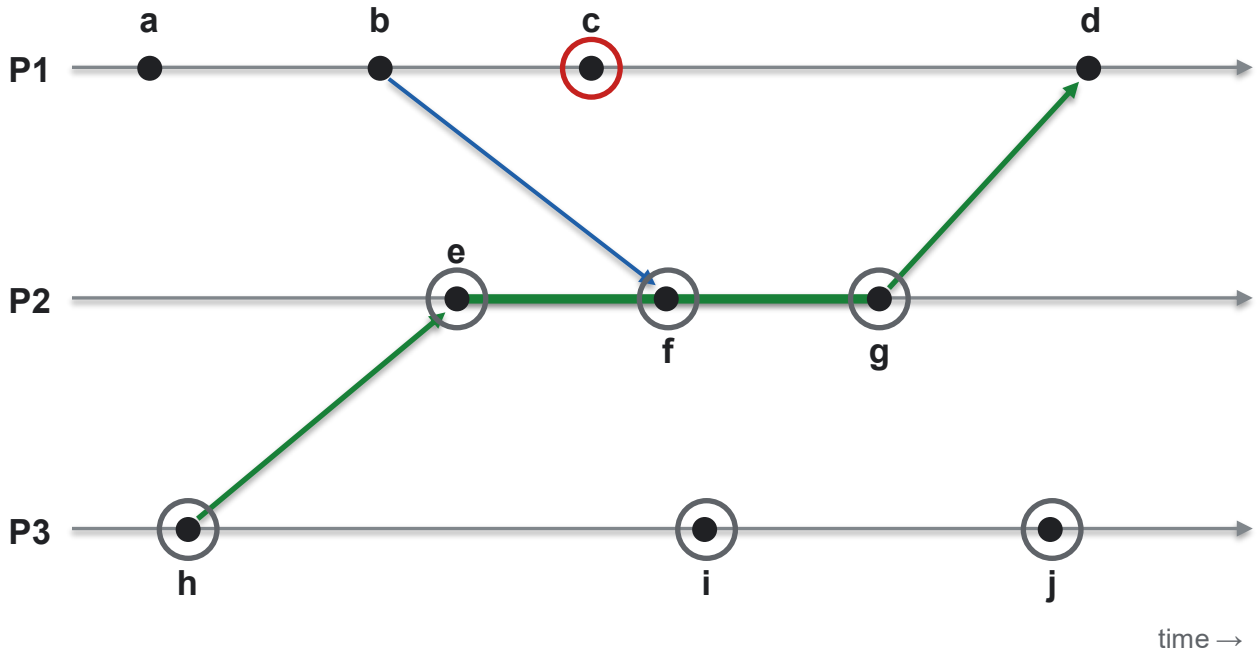


1. Does h happen before d?

2. Which events are concurrent with c?

3. Is a → j?

Exercise 1: answers



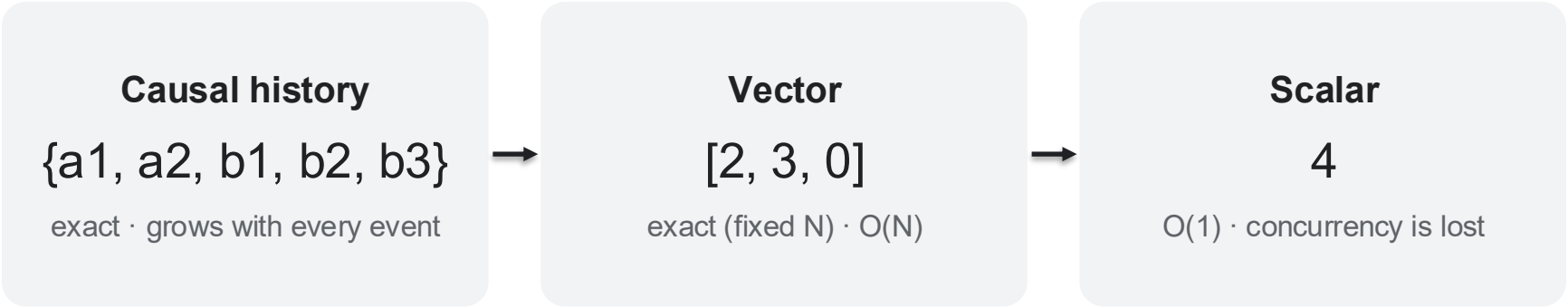
1. Yes: h → e → f → g → d;

2. e, f, g, h, i, and j (six events!): no causal path connects them to c;

3. No: a ∥ j, since P3 never hears from P1;

The compression dilemma

Histories are exact, but they grow forever. **What survives if we keep less?**



Each box describes event **b3** from our execution.

Lamport keeps one number; vector clocks keep the last event per process; version vectors keep only the updates.

Lamport Clocks

One number per event

Lamport Clocks

- Introduced in Lamport's 1978 paper "Time, Clocks, and the Ordering of Events in a Distributed System";
- Orders events **without synchronized physical clocks;**
- Each process keeps one integer counter: **logical time;**
- Representation: **one scalar per event:** $O(1)$ metadata per message;

Lamport clock

Algorithm 2: Lamport Clock

Data: Node N_i with logical clock $LC_i \leftarrow 0$

function LocalEvent()
    $LC_i \leftarrow LC_i + 1$
    return LC_i

function Send(m, N_j)
    $LC_i \leftarrow LC_i + 1$
    transmit $\langle m, LC_i \rangle$ to N_j
    return LC_i

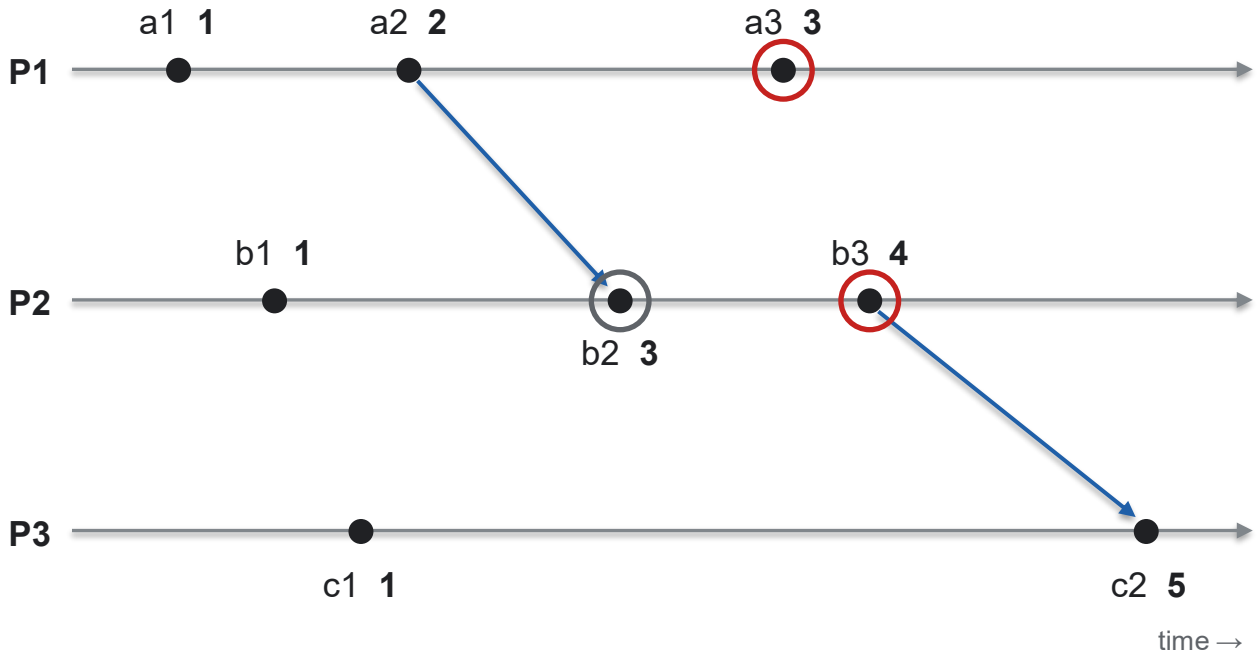
function Receive($\langle m, t \rangle$)
    $LC_i \leftarrow \max(LC_i, t) + 1$
    deliver m
    return LC_i

function Order(e) *// total order: (clock, node id)*
    return (LC(e), i)

The Lamport clock condition

- **If $a \to b$ then $L(a) < L(b)$;**
- Why: L increases along local order (C1: $a <_i b \Longrightarrow L_i(a) < L_i(b)$), and a receive is always greater than its send (C2: $L_i(\text{send}(m)) < L_j(\text{recv}(m))$), so L increases along every causal path (transitivity needs no rule of its own, since < on integers is already transitive);
- Equivalently: if $L(a) \geq L(b)$, then *a* did not happen before *b*;

Lamport clocks on the same execution



$a2 \rightarrow b2 : 2 < 3$ ✓

clock condition holds

$a3 \parallel b3$, yet $L(a3) = 3 < L(b3) = 4$

$a3 \parallel b2$ and $L = 3 = 3$ (a tie)

Numbers alone cannot tell you which pairs are concurrent

The converse does not hold

- $L(a) < L(b) \not\Rightarrow a \rightarrow b$ (example: a3 and b3);
- Causality is preserved in **one direction only;**
- Lamport clocks **cannot detect concurrency;**
- L induces a total **preorder** consistent with → (ties possible), not the causal partial order;
- Counters only grow; in practice 64 bits never run out;

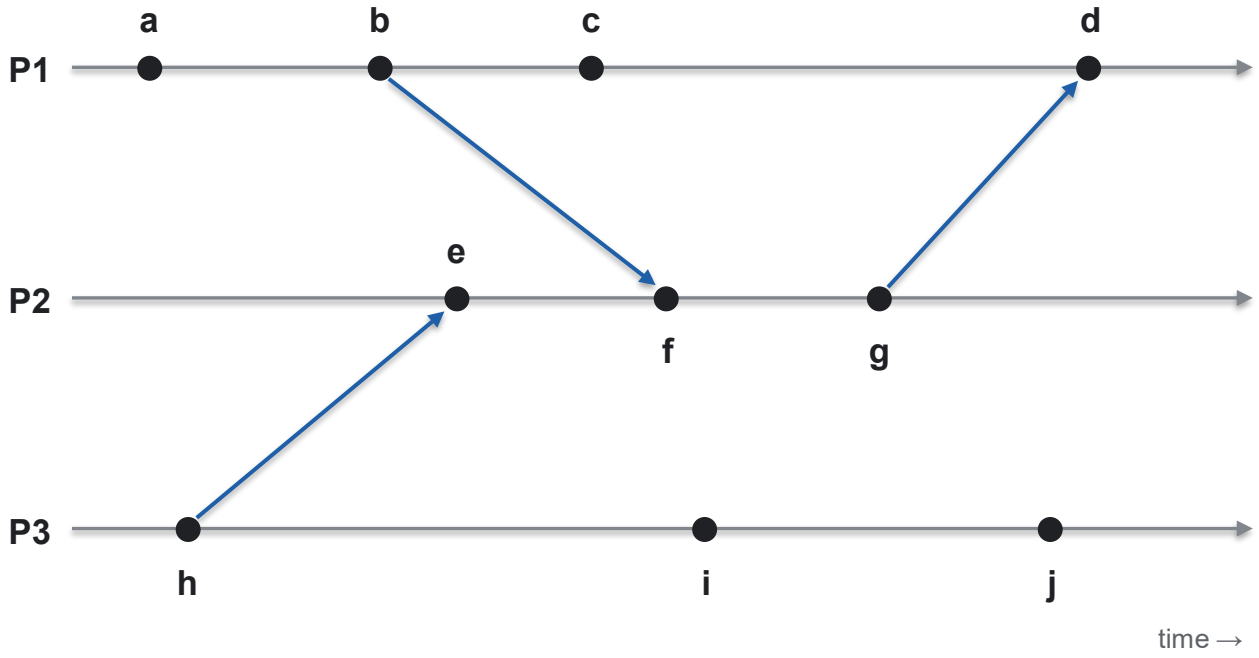
From Lamport timestamps to a total order

- Break ties with the process id: compare **(L, PID)** lexicographically, clock first: $(L(a), i) < (L(b), j) \Leftrightarrow L(a) < L(b) \vee (L(a) = L(b) \wedge i < j)$;
- Example: $a3 = (3, P1)$ and $b2 = (3, P2) \Rightarrow$ a3 ordered before b2;
- Result: a strict **total order that extends happened-before (→);**
- Between concurrent events this order is **arbitrary but consistent:** it does not claim causality;
- Good enough for mutual exclusion, total-order broadcast, deterministic tie-breaking;

Where Lamport clocks are used

- **Distributed mutual exclusion** (Lamport's algorithm: grant requests in (L, pid) order);
- **Total-order broadcast / replicated state machines**: a consistent order every node agrees on;
- **Logs and debugging:** ordering events that are consistent with causality;
- Building block for **hybrid logical clocks** (later);

Exercise 2: Lamport clocks

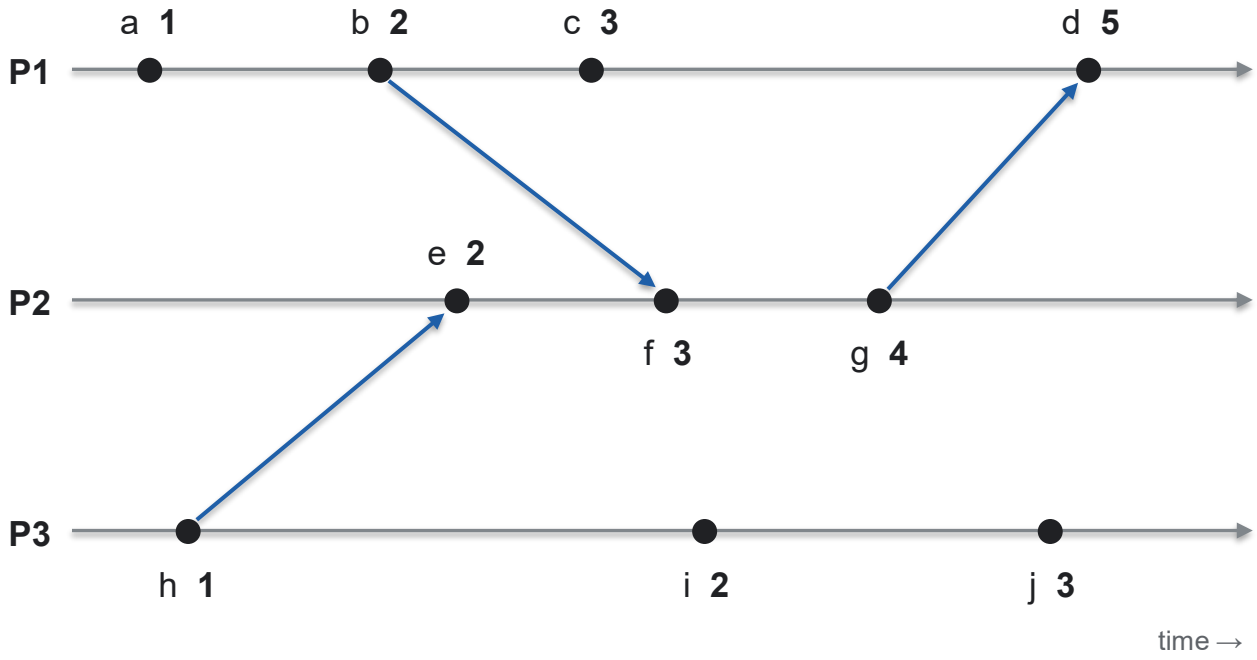


1. Compute the Lamport timestamp of every event;

2. Find two concurrent events x and y with L(x) < L(y);

3. Which events have the same timestamp as c?

Exercise 2: answers



1. a 1, b 2, c 3, d 5; e 2, f 3, g 4; h 1, i 2, j 3;

2. e.g., i ∥ f, yet L(i) = 2 < L(f) = 3;

3. f and j (L = 3). With (L, PID): (3, P1) < (3, P2) < (3, P3);

Vector Clocks

One counter per process

From causal histories to vectors

- $b_3 \in H \implies (b_1 \in H \wedge b_2 \in H)$: histories are **prefix-closed per process;**
- So keep only the **last event seen from each process;**
- In Baquero's figure, the causal history {a1, a2, b1, b2, b3, c1, c2, c3} is compacted to {a: 2, b: 3, c: 3}, or simply the vector [2, 3, 3];
- $H(b_3) = \{a_1, a_2, b_1, b_2, b_3\} \Rightarrow [2,3,0]$
- Same information, size $O(N)$ instead of $O(\text{events})$;
- Vector clocks are **losslessly compressed causal histories** (for a fixed set of N processes);

Vector clock in pseudocode

Algorithm 3: Vector Clock

Data: Node N_i with vector $V_i[1..n] \leftarrow [0, \ldots, 0]$

function Tick()
    $V_i[i] \leftarrow V_i[i] + 1$
    return V_i

function LocalEvent()
    return Tick()

function Send(m, N_j)
    $t \leftarrow$ Tick()
    transmit $\langle m, t \rangle$ to N_j
    return t

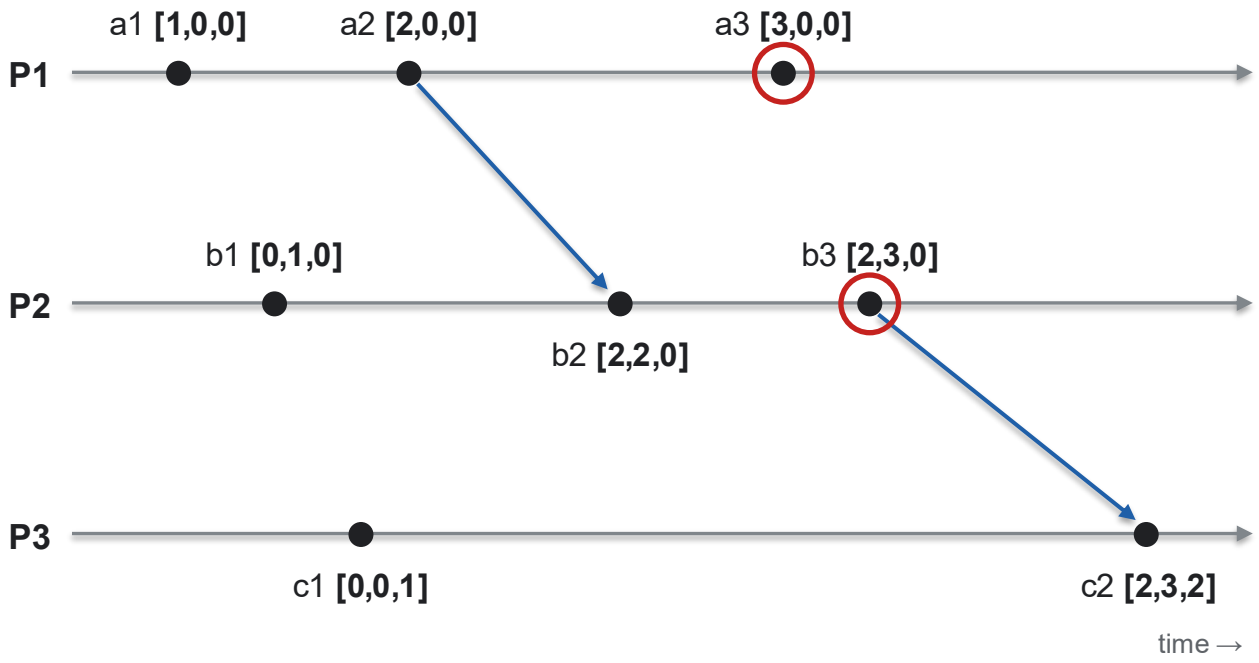
function Receive($\langle m, V_M \rangle$)
    for $k \leftarrow 1$ **to** n **do**
        $V_i[k] \leftarrow \max(V_i[k], V_M[k])$ *// merge*
    $t \leftarrow$ Tick()
    deliver m
    return t

function Leq(V, W)
    return $\forall k$: $V[k] \leq W[k]$
function HappenedBefore(V, W) *// $V \rightarrow W$*
    return Leq(V, W) **and** $V \neq W$
function Concurrent(V, W) *// neither happened before*
    return not Leq(V, W) **and not** Leq(W, V)

Comparing vectors

- $V \leq W \Leftrightarrow V[i] \leq W[i]$ for every *i*;
- $V < W \Leftrightarrow V \leq W \wedge V \neq W$ (strict: at least one component smaller);
- $V \parallel W \Leftrightarrow \neg(V \leq W) \wedge \neg(W \leq V)$ (some component is larger in each);
- Examples: $[2,0,0] < [2,3,0]$; $[3,0,0] \parallel [2,3,0]$
- Unequal vectors are **not** automatically ordered;

Vector clocks on the same execution



$a2 \rightarrow b3$: $[2,0,0] < [2,3,0]$ ✓

a3 [3,0,0] vs b3 [2,3,0]: $3 > 2$ but $0 < 3 \Rightarrow a3 \parallel b3$

Lamport said 3 < 4; the vectors reveal the concurrency

What vector clocks guarantee

- $a \to b \Leftrightarrow V(a) < V(b)$: the clock **characterizes** causality;
- $a \parallel b \Leftrightarrow V(a) \parallel V(b)$: concurrency is detected exactly;
- If a happened on P_i and $a \neq b$, a single comparison is enough: $a \to b \Leftrightarrow V(a)[i] \leq V(b)[i]$, since $V(b)[i]$ counts the events of P_i that b can reach backwards;
- Never imposes a false order on concurrent events;
- Assumes the standard algorithm with a known, fixed set of N processes;

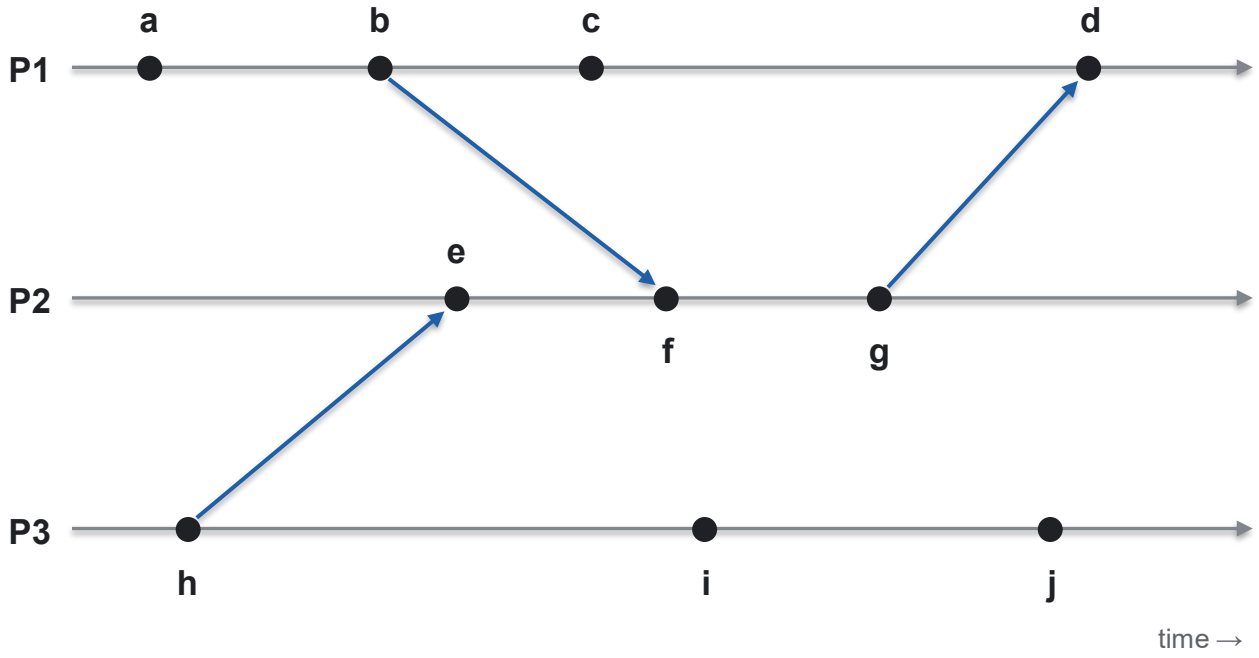
Precision versus metadata

Representation	Metadata size	Decides a → b exactly?	Can detect concurrency?
Explicit causal history	Grows with events	Yes	Yes
Lamport clock	O(1)	No	No
Vector clock	O(N)	Yes	Yes

The cost of vector clocks

- $O(N)$ integers per event and per message, where N is the number of processes;
- **Dynamic membership** is hard: processes join, leave, crash; entries are never safely removed;
- Mitigations:
  - Send only changed entries (Singhal–Kshemkalyani differential technique; requires FIFO channels);
  - Sparse / compressed vectors;
  - Dynamic identities: Interval Tree Clocks (see 'Beyond vectors');

Exercise 3: vector clocks

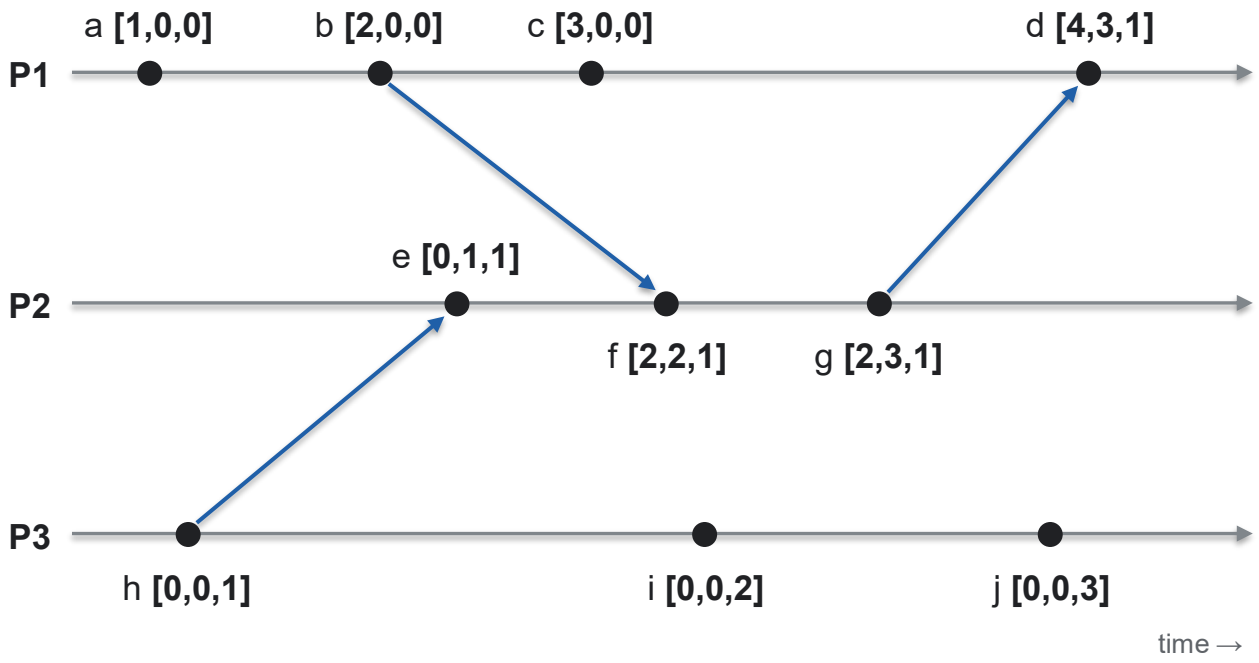


1. Compute the vector timestamps of f, g, and d;

2. Are c and f concurrent? Use the vectors;

3. Does h happen before d?

Exercise 3: answers



1. f [2,2,1], g [2,3,1], d [4,3,1];

2. c [3,0,0] vs. f [2,2,1]: 3 > 2 but 0 < 2, so c ∥ f;

3. Yes: h [0,0,1] < d [4,3,1];

Hybrid Logical Clocks

Logical time that stays close to wall-clock time

Why hybrid time?

- Logical clocks have **no relation to wall-clock time**: you cannot ask "what changed at 10:05?"
- Physical clocks relate to real time but **can violate causality** under skew;
- Hybrid logical clocks (HLC) combine both:
  - a timestamp that **respects causality** like a Lamport clock;
  - and stays **within a bounded distance of physical time;**
- Requires loosely synchronized clocks (NTP) with **bounded skew ε;**

HLC representation: (l, c)

- **pt:** the node's physical clock reading (from NTP-disciplined hardware);
- **l:** the largest physical time this node has **heard of** (its own or from messages), not necessarily its own pt;
- **c:** a small counter that orders events sharing the same l;
- Timestamps compare lexicographically: (l, c);
- Fits in 64 bits in practice (e.g. 48 bits of l + 16 bits of c);

Hybrid logical clock in pseudocode

Algorithm 6: Hybrid Logical Clock

Data: Node N_i with $l_i \leftarrow 0$ **and** $c_i \leftarrow 0$; pt_i reads N_i's physical clock

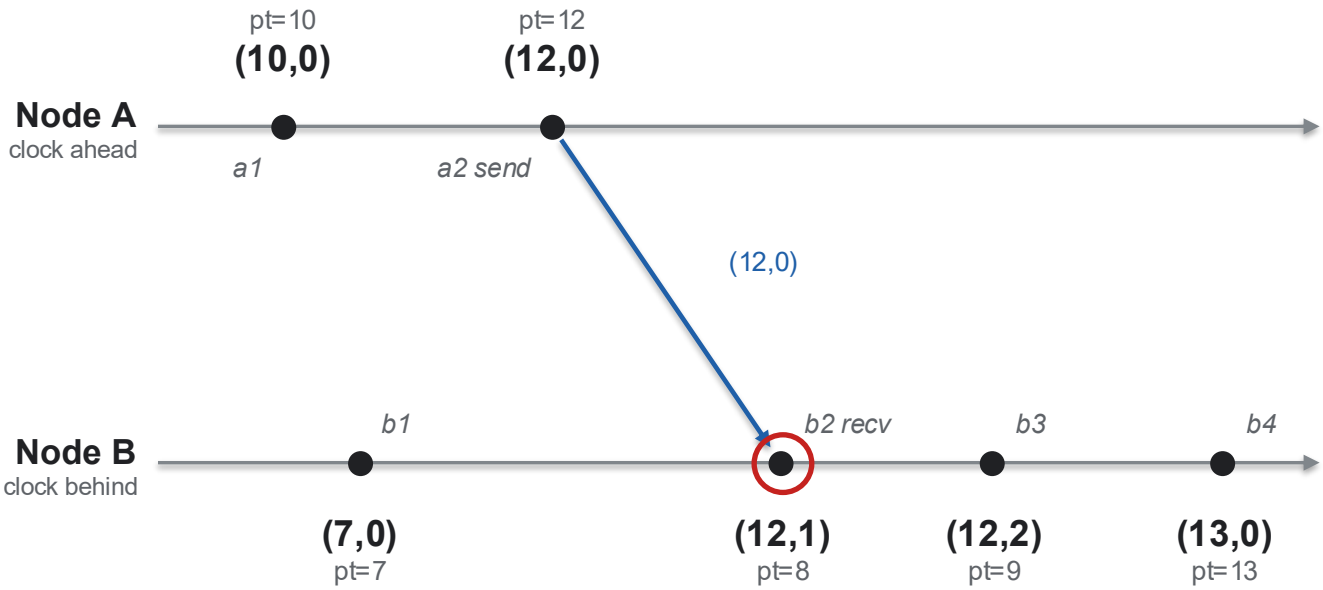
function Tick() *// local event or send*
- $l' \leftarrow l_i$
- $l_i \leftarrow \max(l', pt_i)$
- **if** $l_i = l'$ **then** $c_i \leftarrow c_i + 1$
- **else** $c_i \leftarrow 0$
- **return** (l_i, c_i)

function LocalEvent()
- **return** Tick()

function Send(m, N_j)
- $t \leftarrow$ Tick()
- transmit $\langle m, t \rangle$ to N_j
- **return** t

function Receive($\langle m, (l_M, c_M) \rangle$)
- $l' \leftarrow l_i$
- $l_i \leftarrow \max(l', l_M, pt_i)$
- **if** $l_i = l' = l_M$ **then** $c_i \leftarrow \max(c_i, c_M) + 1$
- **else if** $l_i = l'$ **then** $c_i \leftarrow c_i + 1$
- **else if** $l_i = l_M$ **then** $c_i \leftarrow c_M + 1$
- **else** $c_i \leftarrow 0$
- deliver m
- **return** (l_i, c_i) *// compare (l, c) lexicographically*

HLC under clock skew



b2: l = max(7, 12, 8) = 12 = l.m ⇒ c = 0 + 1 = **1**

b3: pt = 9 < l ⇒ l stays 12, c = **2**

b4: pt = 13 > l ⇒ l = 13, c resets to **0**

Wall clock alone: pt(b2) = 8 < pt(a2) = 12, yet a2 → b2 ✗

HLC: (12,0) < (12,1) ✓

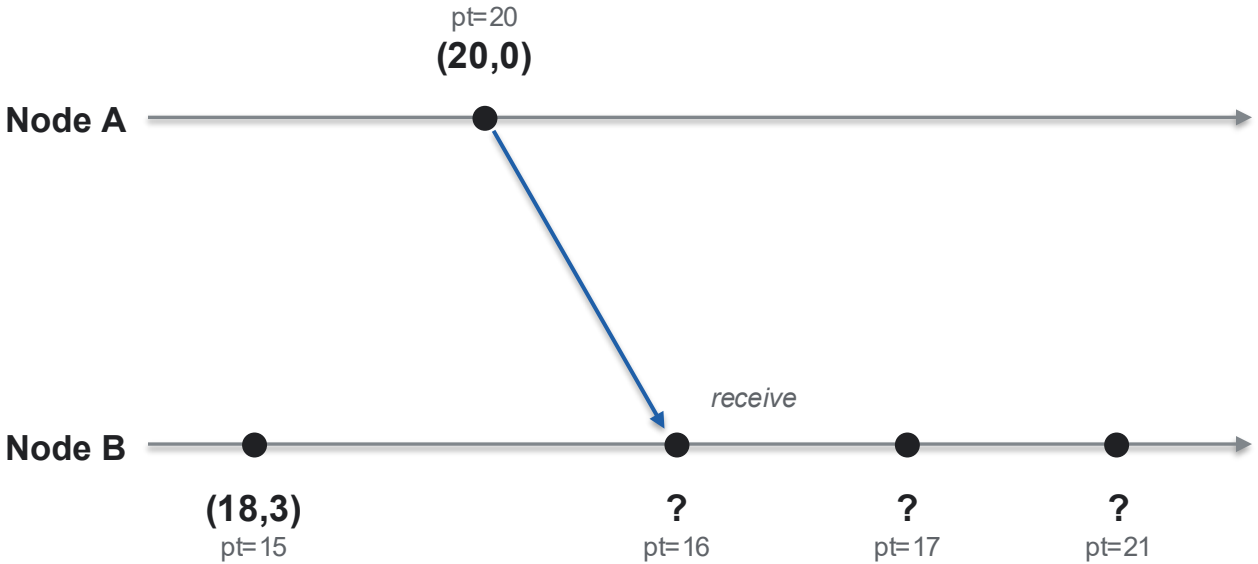
HLC guarantees and their assumptions

- $a \to b \Rightarrow \mathrm{HLC}(a) < \mathrm{HLC}(b)$: the Lamport clock condition;
- $l \geq \mathrm{pt}$ always, and $l - \mathrm{pt} \leq \varepsilon$ when clock skew is bounded by ε;
- **c is bounded** under the same assumption: it does not grow without limit;
- Like Lamport: the converse fails, so HLC **cannot detect concurrency;**
- Risk: one node with a clock far in the future drags everyone's l forward → reject or alarm on offsets > ε;

HLC in practice

- Used for transaction and MVCC timestamps in **CockroachDB**, and in **YugabyteDB** and **MongoDB** (cluster time);
- CockroachDB: max clock offset defaults to **500 ms**; uncertainty intervals handle skew below it; a node whose clock is off by > 80% of the max (vs. a majority) self-terminates;
- HLC is **not** TrueTime: Spanner's TT.now() returns an interval and **commit-wait** gives external consistency;
- A timestamp alone provides **neither isolation nor linearizability**; the protocol around it does;

Exercise 4: hybrid logical clocks



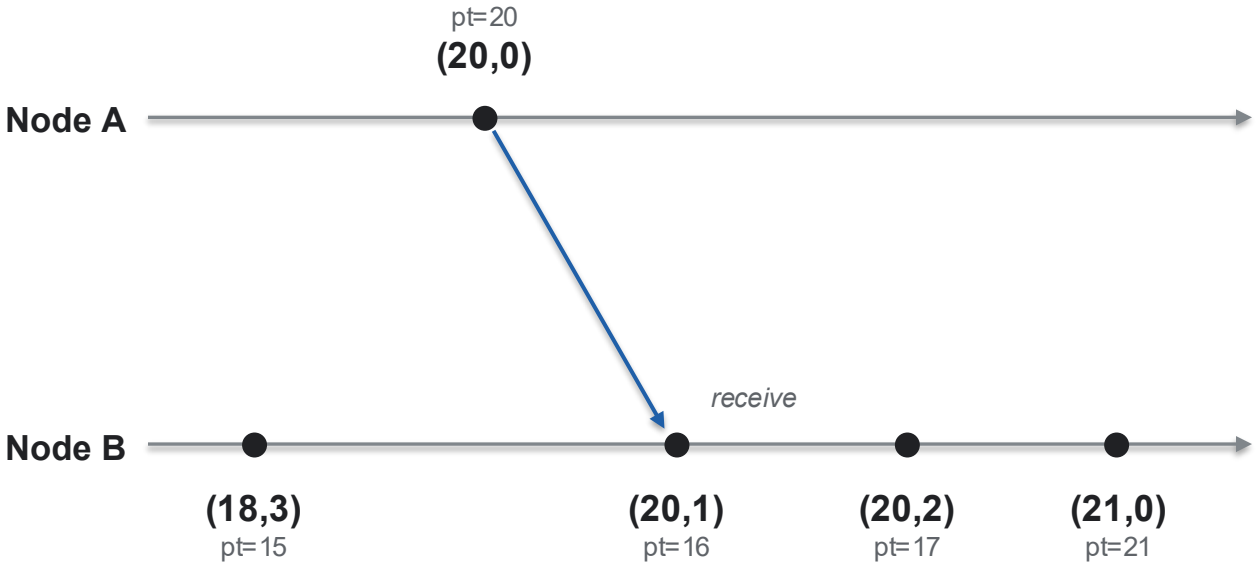
B's clock is (18, 3). A message stamped (20, 0) arrives when B's pt = 16;

1. What is B's HLC after the receive?

2. And after two local events at pt = 17 and pt = 21?

3. Which branch of Receive did you use?

Exercise 4: answers



1. $l = \max(18, 20, 16) = 20 = l_M$, so $c = c_M + 1$: **(20, 1)**;

2. pt = 17 < l: (20, 2); then pt = 21 > l: (21, 0);

3. elif $l_i = l_M$: $c_i \leftarrow c_M + 1$;

Three clocks, three questions

Lamport

Answers

"Give me an order consistent with causality, cheaply."

Cost

O(1)

Cannot tell you

whether two events are concurrent

Vector

Answers

"Are a and b causally related, or concurrent?"

Cost

O(N)

Cannot tell you

anything about wall-clock time

HLC

Answers

"A causality-respecting timestamp close to real time."

Cost

O(1): (l, c)

Cannot tell you

concurrency; needs bounded skew

Not a ladder: different design objectives.

Break (15 minutes)

Next: from events to data, versioning replicated objects

Version Vectors

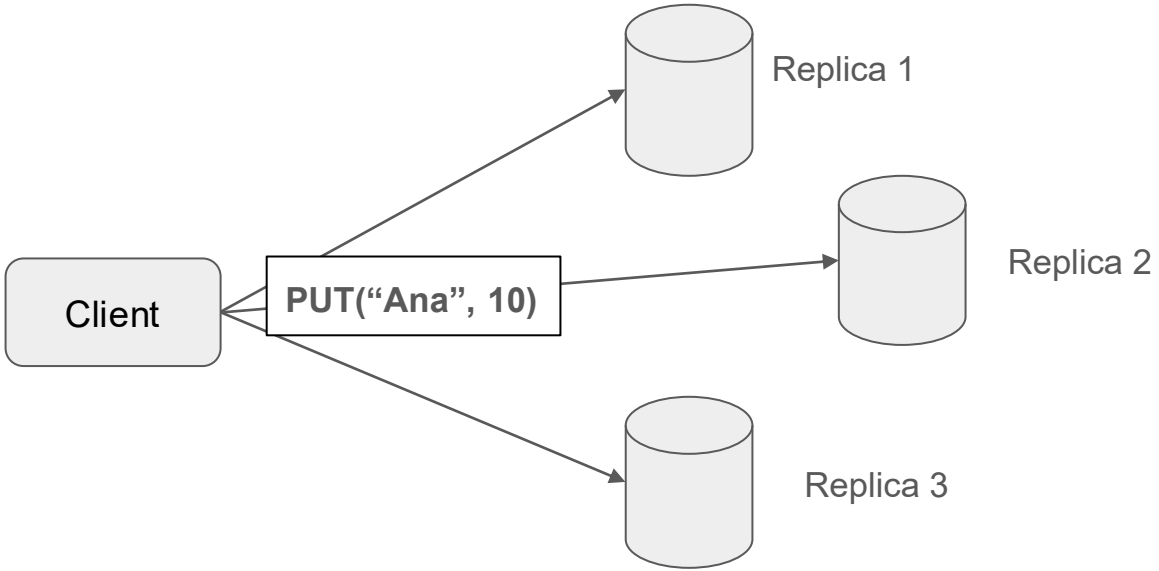
Causality for replicated data

From processes to data

- So far: **process-centric**: we order events in a computation;
- Now: **data-centric**: we track the versions of a replicated object;
- Key-value API: **GET(key)**, **PUT(key, value)**, DELETE(key);
- With concurrent writes, a PUT may not overwrite; it may create a sibling version;
- Exposed in the **Dynamo paper** (§4.4), **Riak**, Voldemort. Not every store: Cassandra uses per-cell last-write-wins timestamps; DynamoDB (the product) does not expose vector clocks;

Replicated Data

- Each key is replicated for availability, fault tolerance and latency (DDIA ch. 6)
- Replication can be synchronous or asynchronous



Sequential or concurrent?

Scenario A: sequential writes

Client 1: PUT(K, "red") at replica A;

Client 1: GET(K) returns "red";

Client 1: PUT(K, "blue");

The second write observed the first, so it causally supersedes it: K = "blue";

Scenario B: concurrent writes

Client 1: PUT(K, "red") at replica A;

Client 2: PUT(K, "blue") at replica B;

Replication runs asynchronously;

Neither write observed the other: K = {"red", "blue"};

How can the database distinguish these two situations without relying on physical timestamps?

Version vectors

- $\mathrm{VV}(x) = [v_1, \ldots, v_n]$: v_i counts the updates to object x that originated at replica (or client) i;
- Same structure as a vector clock, but only **updates to one object** count, not every event;
- A version vector represents causal history, not physical time;
- Parker et al., 1983, before vector clocks (Fidge / Mattern, 1988–89);
- A version vector is a compressed causal history of the object's **update events;**

Only updates count: histories …

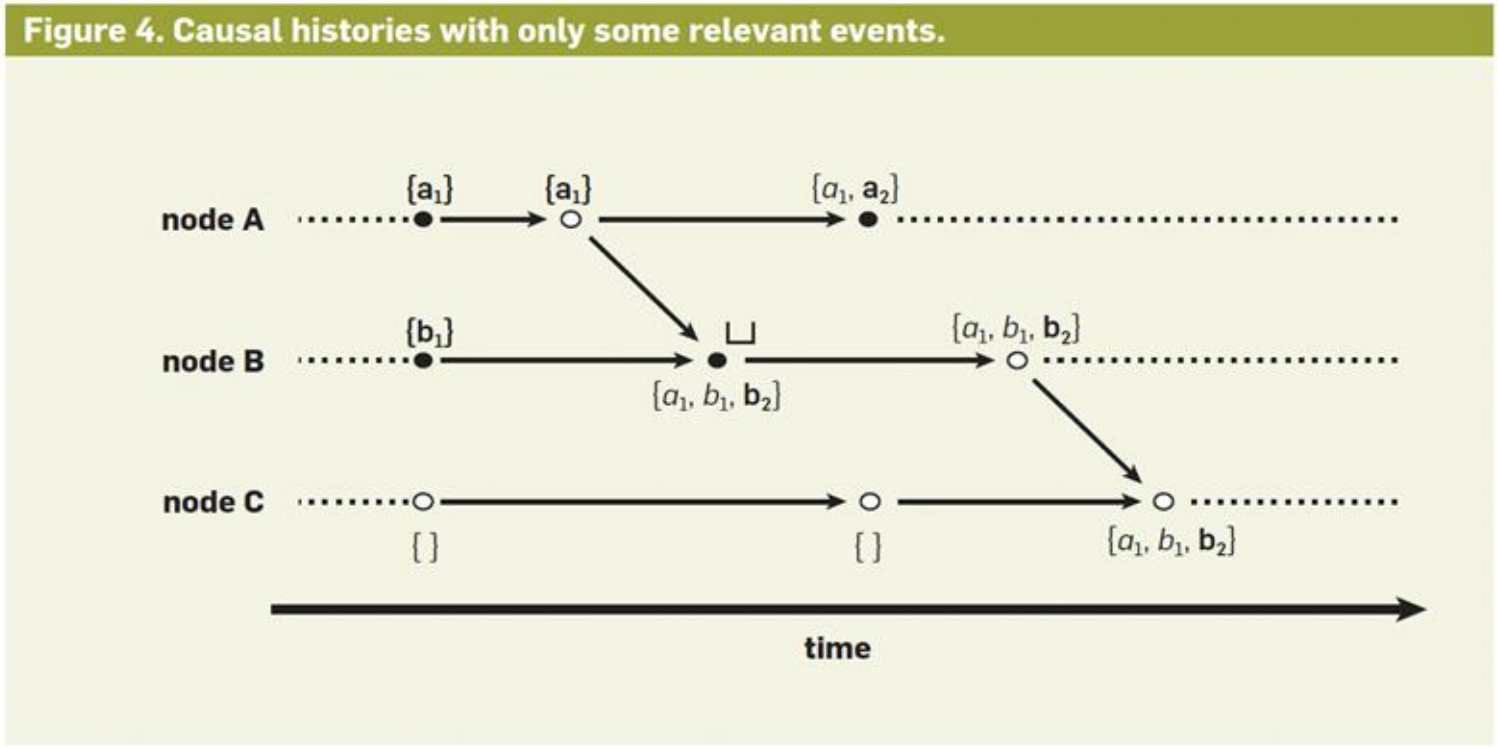


Figure: Baquero & Preguiça, "Why Logical Clocks Are Easy", CACM 59(4), 2016

… compressed into version vectors

Figure 5. Version vectors with only some relevant events.

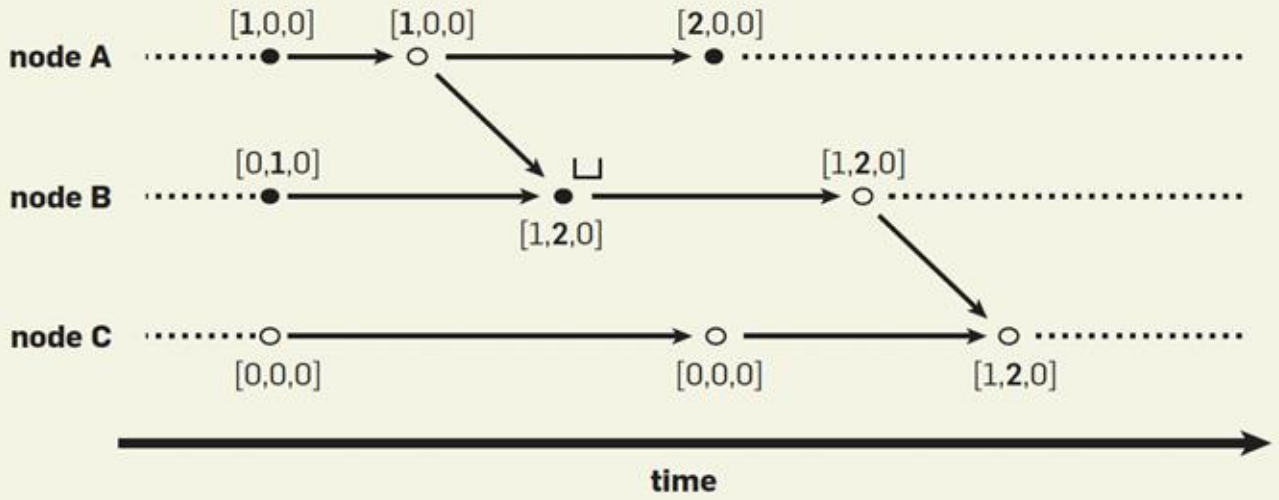
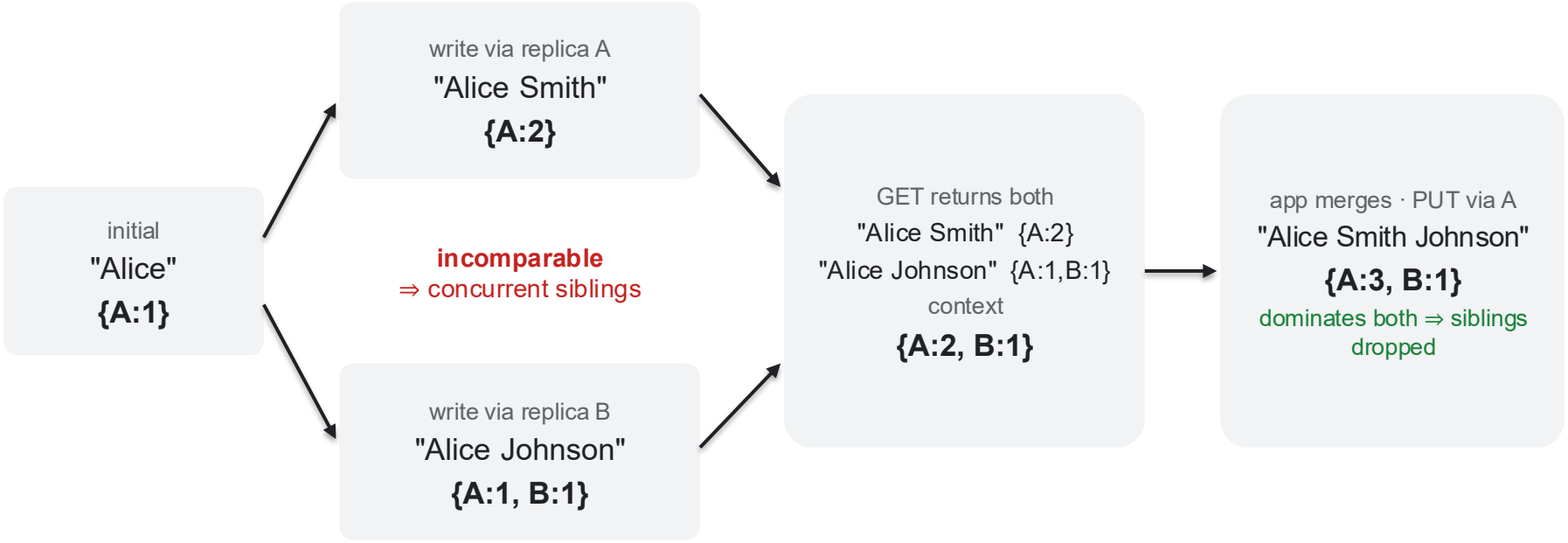


Figure: Baquero & Preguiça, "Why Logical Clocks Are Easy", CACM 59(4), 2016

Comparing versions

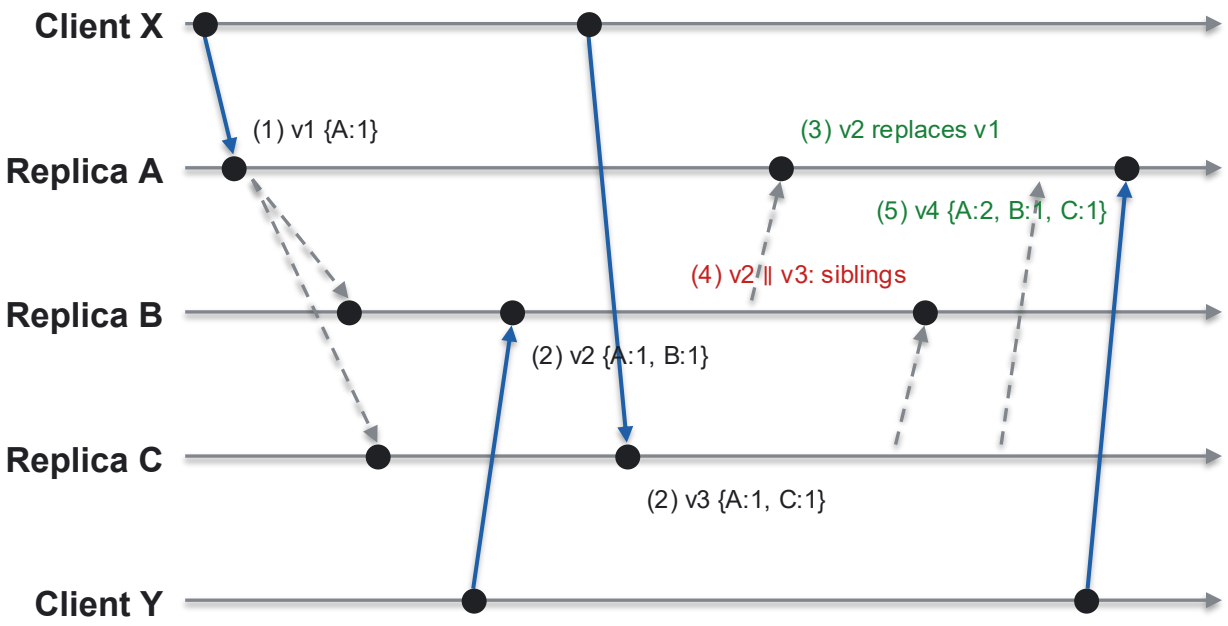
- Compare version vectors component-wise, exactly like vector clocks;
- **VV1 dominates VV2** ($VV_2 < VV_1$): VV1 includes every update VV2 has, plus more ⇒ VV2 is obsolete;
- e.g., $[2,1,0] < [2,2,0]$: the first history is contained in the second;
- **Equal:** same set of updates ($VV_1 = VV_2$);
- **Incomparable:** concurrent updates ($VV_1 \parallel VV_2$) ⇒ **conflict**, so keep both as siblings;
- e.g., $[2,1,0] \parallel [1,2,0]$: neither vector dominates the other, so the versions are concurrent;
- The vector detects the conflict; it does **not** decide how to merge the values;

Siblings: concurrent writes at different replicas



The vector detects the conflict; **the application decides the merge**. Systems differ in how GET/PUT expose the context.

Version vectors with three replicas



1. X writes v1 at A: {A:1}; it is replicated to B and C;

2. Partition! Y writes v2 at B with ctx {A:1} → {A:1, B:1}; X writes v3 at C with ctx {A:1} → {A:1, C:1};

3. A and B sync: {A:1, B:1} dominates {A:1}, so A drops v1;

4. B and C sync: v2 || v3, so both keep two siblings;

5. C and A sync. Y reads both at A (ctx {A:1, B:1, C:1}) and writes v4: {A:2, B:1, C:1} dominates both;

Solid arrows: client calls. Dashed arrows: replication and anti-entropy;

The write path: GET returns a context, PUT sends it back

- **GET:** returns the value(s) plus an opaque **causal context** summarizing what the client saw;
- **PUT:** the client sends the new value **with that context;**
- The replica creates a new version: context + its own increment;
- Stored versions **dominated** by the new one are discarded; the rest remain siblings;
- A PUT **without** a context claims to have seen nothing, so it should only add a sibling; whether it does depends on the versioning scheme (next);

Version vectors in pseudocode (1/2): Get and Put

Algorithm 7: Version Vectors (per-server replica)

Data: Replica R_i with counter $n_i \leftarrow 0$ **and** versions $S_i \leftarrow \emptyset$ *// S_i: (vv, value) pairs; concurrent ones are siblings*

```
function Get()
    ctx ← [0, ..., 0]
    values ← ∅
    for each (vv, value) in S_i do
        ctx ← Max(ctx, vv)    // merge causal contexts
        values ← values ∪ {value}
    return (values, ctx)

function Max(u, v)    // n = number of replicas
    for k ← 1 to n do
        out[k] ← max(u[k], v[k])
    return out
```

```
function Put(value, ctx)
    n_i ← n_i + 1
    vv ← copy(ctx)
    vv[i] ← n_i    // allocate a new version
    for each (u, v) in S_i do
        if Dominates(vv, u) then
            remove (u, v) from S_i    // superseded
    S_i ← S_i ∪ {(vv, value)}    // keep concurrent siblings
```

A new version supersedes only the versions whose causal history it dominates; the others stay as siblings;

Per-server ids: n_i also orders independent writes through R_i, so a write its client never saw can be discarded (next section);

Simplification: n_i must be durable and never reused (e.g., after a crash), so version ids stay unique and monotonic. Dominates is on the next slide.

Version vectors in pseudocode (2/2): Sync

Algorithm 7: Version Vectors (per-server replica), continued

```
function Sync(S_j)    // anti-entropy with R_j
    S ← S_i ∪ S_j
    for each (u, v) in S do
        for each (w, x) in S do
            if Dominates(w, u) then
                remove (u, v) from S    // w has seen u
                break
    S_i ← S
```

```
function Dominates(u, v)    // u has seen all of v
    for k ← 1 to n do
        if u[k] < v[k] then
            return false
    return u ≠ v    // ... and at least one more update
```

```
function Concurrent(u, v)    // siblings
    return u ≠ v
        and not Dominates(u, v)
        and not Dominates(v, u)
```

Dominates(u, v): u includes every update in v, and more, so v can be discarded;

Concurrent(u, v): neither includes all of the other's updates, so both are kept as siblings;

Sync keeps exactly the versions that no other version dominates. "for each" iterates over the set as it was when the loop began, so removing an element never skips another;

Exercise 5: version vectors

- A replica stores two siblings: x {A:2, B:1} and y {A:1, B:2}. Per-server counters: A = 2, B = 2;
- Starting from this state each time, which versions remain after:
  - 1. a PUT via A with ctx {A:2, B:1}?
  - 2. a PUT via B with ctx {A:2, B:2}?
  - 3. a PUT via B with no context?

Exercise 5: answers

- 1. The new version {A:3, B:1} dominates x only: y and the new version remain;
- 2. {A:2, B:3} dominates both: only the new version remains;
- 3. {B:3} is concurrent with x and with y: three siblings!
- The client's context decides what is replaced, not the server that receives the write;

Dotted Version Vectors

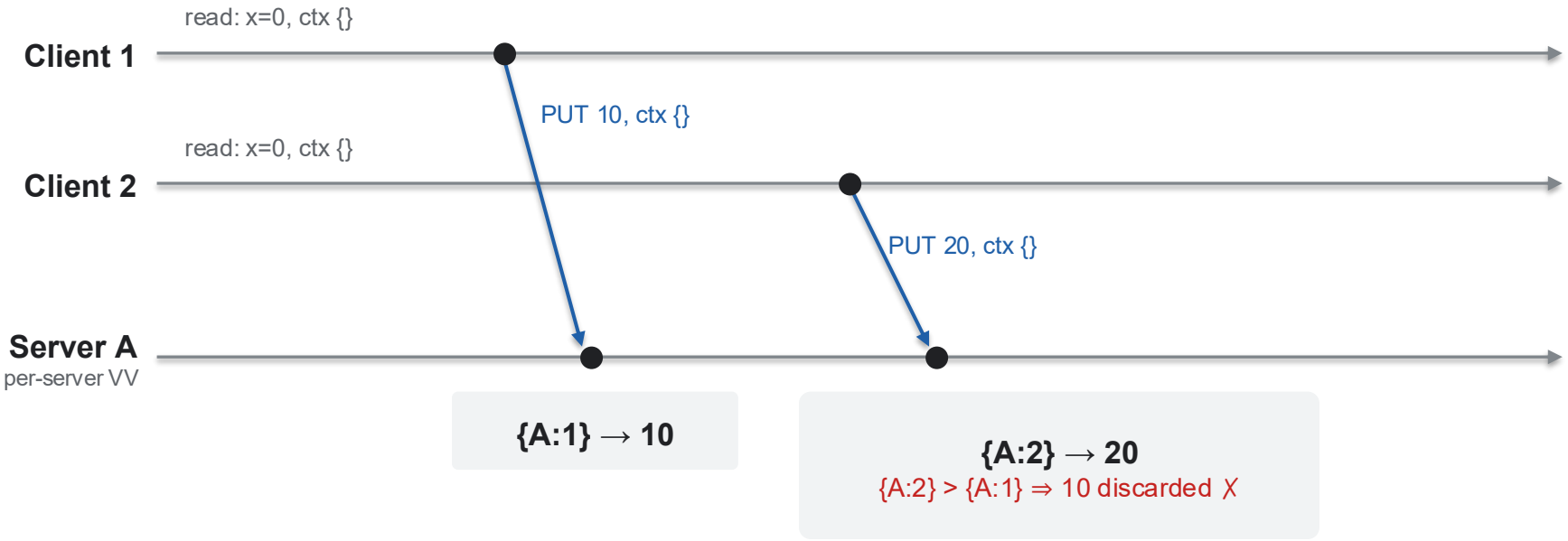
Separating an update's identity from its causal history

A question to start

Two clients independently update the same key through the same replica. How can the database know that neither write supersedes the other, even though the server processed them sequentially?

The order in which a server processes writes is not necessarily the causal order of the client operations that produced them;

The problem: two clients, one server



The server's counter order (A:1, then A:2) was used **as if** Client 2 had seen Client 1's write.
It had not: both PUTs carried ctx {}. The writes are concurrent, and one of them was lost.

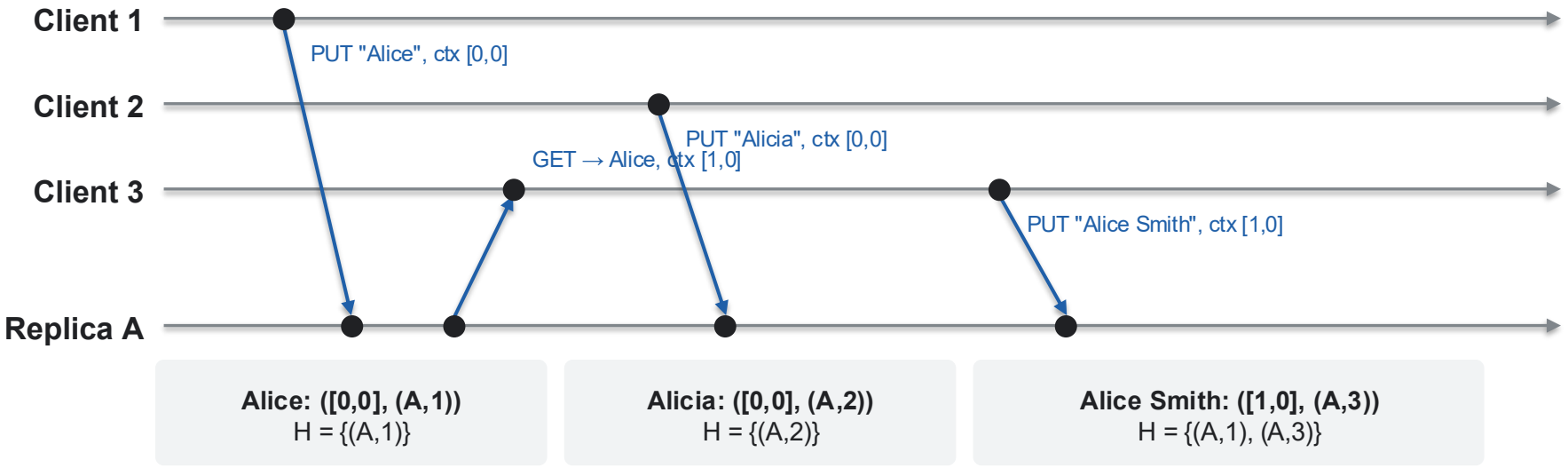
The problem is the server's aggregate history

- Version vectors themselves are not incorrect: they can detect concurrency, provided each version has an accurately maintained causal history;
- The problem arises when the server's aggregate version vector is used as the causal history of an individual version;
- The server's history may include events the client has never observed, which creates false causal dependencies and suppresses genuine conflicts;
- With per-client entries, version vectors stay exact, but they grow with the number of clients;
- Dotted version vectors offer a compact way to keep server ids and still preserve this distinction;

Dotted version vectors: a dot and its causal history

- A dotted version vector is a pair $\mathrm{DVV} = (\mathrm{VV}, d)$:
  - VV represents the causal predecessors of a version;
  - $d = (i, n)$, the dot, uniquely identifies the event that created that version (actor i, counter n);
- Its complete causal history is $H(\mathrm{VV}, d) = H(\mathrm{VV}) \cup \{d\}$, where $H(\mathrm{VV}) = \{(i, k) | 1 \leq k \leq \mathrm{VV}[i]\}$;
- **This separates the identity of an update from the causal history it depends on: the dot is not merely another counter!**
- The dots (A,1) and (A,2) are ordered in A's counter sequence, but that does not imply that the client updates they identify are causally related. The version's causal context determines that;
- This assumes that each actor numbers its events sequentially, and that VV describes a contiguous prefix of observed events;

Comparing DVVs: an example

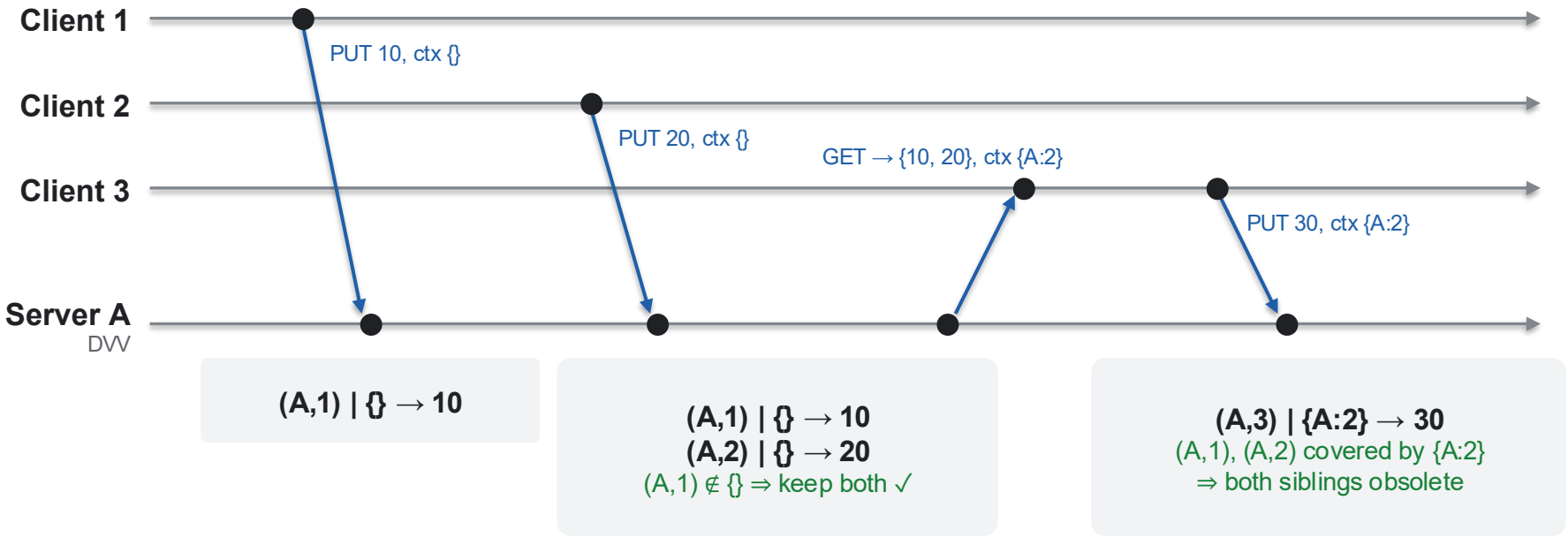


Alice → Alice Smith, since {(A,1)} ⊂ {(A,1), (A,3)};

Alicia ‖ Alice and Alicia ‖ Alice Smith: neither history contains the other, even though A processed the three writes in sequence;

Key customer:123; vectors list A, then B. Each version is (VV, dot), and H is its complete causal history;

The same writes with dotted version vectors



(dot | context) is a teaching abstraction, not the exact wire format; DVV sets share one context across siblings.

DVVs in pseudocode (1/3): Get and Put

Algorithm 8: Dotted Version Vectors (one DVV per version)

Data: Replica R_i with counter $n_i \leftarrow 0$ **and** versions $S_i \leftarrow \emptyset$ *// S_i: (X, value) pairs; X = (VV, d)*

```
function Get()
    ctx ← [0, ..., 0]
    values ← ∅
    for each ((VV, d), value) in S_i do
        ctx ← Max(ctx, VV)    // Max as in Algorithm 7
        (r, k) ← d
        ctx[r] ← max(ctx[r], k)    // the dot was seen too
        values ← values ∪ {value}
    return (values, ctx)
```

```
function Put(value, ctx)
    n_i ← n_i + 1
    d ← (i, n_i)    // new dot: unique update id
    X ← (copy(ctx), d)    // only what the client saw
    for each (Y, v) in S_i do
        if HappenedBefore(Y, X) then
            remove (Y, v) from S_i    // superseded
    S_i ← S_i ∪ {(X, value)}    // keep concurrent siblings
    return X
```

The dot names the new update; the context names its causal predecessors;

A new write does not inherit every update R_i has processed, only what its client saw;

n_i must be durable, so dots stay unique. HappenedBefore is on the next slide. "for each" iterates over the set as it was when the loop began, so removing an element never skips another;

DVVs in pseudocode (2/3): Sync

Algorithm 8: Dotted Version Vectors, continued

```
function Sync(S_j)    // anti-entropy with R_j
    S ← S_i ∪ S_j
    for each (X, v) in S do
        for each (Y, w) in S do
            if HappenedBefore(X, Y) then
                remove (X, v) from S    // Y has seen X
                break
    S_i ← S
```

```
function HappenedBefore(X, Y)    // X → Y
    H_X ← History(X)
    H_Y ← History(Y)
    return ProperSubset(H_X, H_Y)
```

```
function Concurrent(X, Y)    // neither saw the other
    H_X ← History(X)
    H_Y ← History(Y)
    return not Subset(H_X, H_Y)
        and not Subset(H_Y, H_X)
```

HappenedBefore(X, Y): Y's history contains X's, and more, so X can be discarded;

Concurrent(X, Y): neither history contains the other, even when both dots come from the same replica;

Sync keeps exactly the versions that no other version has seen. "for each" iterates over the set as it was when the loop began, so removing an element never skips another;

DVVs in pseudocode (3/3): histories

Algorithm 8: Dotted Version Vectors, continued

```
function History((VV, d))
    H ← {d}    // include this update
    for each replica r do
        for k ← 1 to VV[r] do
            H ← H ∪ {(r, k)}
    return H

function Subset(A, B)    // is every e in A also in B?
    for each e in A do
        if e ∉ B then
            return false
    return true

function ProperSubset(A, B)    // subset, and not equal
    return Subset(A, B) and A ≠ B
```

Two clients, one replica (A)

Client 1: Put(10, {}) → X = ({}, (A,1))

Client 2: Put(20, {}) → Y = ({}, (A,2))

History(X) = {(A,1)}, History(Y) = {(A,2)}

Neither history contains the other: Concurrent(X, Y) = true, so both versions survive;

Get() = ({10, 20}, {A:2}); Put(30, {A:2}) → ({A:2}, (A,3)) supersedes both;

With per-server version vectors, {A:2} would dominate {A:1}, and 10 would be lost (Algorithm 7);

History() expands the context for explanation only; real implementations compare vectors and dots directly. Assumes VV is a contiguous prefix.

Why dotted version vector sets?

- If a key has several concurrent values, we could store a complete DVV for each value, but this duplicates causal-history information;
- A dotted version vector set (DVVS) represents the object's state with a shared causal context and a set of active dot-value pairs: $S = (C, E)$;
  - C records the events known to the object (its causal context);
  - E identifies the versions that remain active (the siblings);
- e.g., $C = [2,0]$ and $E = \{((A, 1), \text{Alice}), ((A, 2), \text{Alicia})\}$;
- Invariant: every active dot belongs to the causal context, $\forall (d, v) \in E : d \in H(C)$. But being in C does not mean that a value was superseded!
- So the database can answer two separate questions: which updates have been observed (C)? Which values are still active (E)?

DVV sets in pseudocode (1/2): Get and Put

Algorithm 9: Dotted Version Vector Sets (one key)

Data: Replica R_i with context $C_i \leftarrow [0, \ldots, 0]$ **and** active versions $E_i \leftarrow \emptyset$
Data: C_i = the updates observed; E_i = the (dot, value) pairs that are still active

```
function Get()
    values ← ∅
    for each (d, v) in E_i do
        values ← values ∪ {v}
    return (values, C_i)

function Known(C, d)    // is d in H(C)?
    (r, k) ← d
    return k ≤ C[r]
```

```
function Put(value, ctx)    // ctx: what the client read
    for each (d, v) in E_i do
        if Known(ctx, d) then
            remove (d, v) from E_i    // superseded
    C_i ← Max(C_i, ctx)
    C_i[i] ← C_i[i] + 1
    d ← (i, C_i[i])    // allocate a new dot
    E_i ← E_i ∪ {(d, value)}
    return d
```

A client write supersedes only the versions in its causal context;

Versions unknown to the client remain active as siblings;

Max is component-wise (next slide). $C_i[i]$ must be durable, so dots stay unique and monotonic. "for each" iterates over the set as it was when the loop began, so removing an element never skips another;

DVV sets in pseudocode (2/2): Sync

Algorithm 9: Dotted Version Vector Sets (one key), continued

```
function Sync(C_j, E_j)    // anti-entropy with R_j
    E ← ∅
    // keep local versions still active or unknown remotely
    for each (d, v) in E_i do
        if Contains(E_j, d) or not Known(C_j, d) then
            E ← E ∪ {(d, v)}
    // add remote versions not previously observed locally
    for each (d, v) in E_j do
        if not Known(C_i, d) then
            E ← E ∪ {(d, v)}
    C_i ← Max(C_i, C_j)    // preserve causal knowledge
    E_i ← E
```

```
function Contains(E, d)
    for each (dot, value) in E do
        if dot = d then
            return true
    return false
```

```
function Max(u, v)    // n = number of replicas
    for k ← 1 to n do
        out[k] ← max(u[k], v[k])
    return out
```

A local version survives if (1) R_j still stores it, or (2) R_j has never observed it;
If neither holds, R_j observed it and superseded it: keeping it would resurrect an obsolete value;

Simplified from Almeida et al. (DAIS 2014): no quorums, hinted handoff, or actor retirement.

Merging DVV sets: two examples

Concurrent writes

Both replicas start with C = [0,0] and E = {};

A: Put("Alice", [0,0]) B: Put("Alicia", [0,0])

C_A = [1,0], E_A = {((A,1), Alice)}

C_B = [0,1], E_B = {((B,1), Alicia)}

A runs Sync(C_B, E_B): (A,1) is not known by B, and (B,1) is not known by A:

C_A = [1,1], E_A = {((A,1), Alice), ((B,1), Alicia)}

Both versions survive: they are concurrent;

A superseded version

C_A = [1,0], E_A = {((A,1), Alice)}

C_B = [1,1], E_B = {((B,1), Alicia)}

B knows (A,1) but no longer stores it: a client ran Get at B, saw Alice, and Put Alicia;

A runs Sync(C_B, E_B): (A,1) is known by B and not in E_B, so it is dropped:

C_A = [1,1], E_A = {((B,1), Alicia)}

(A,1) must not reappear;

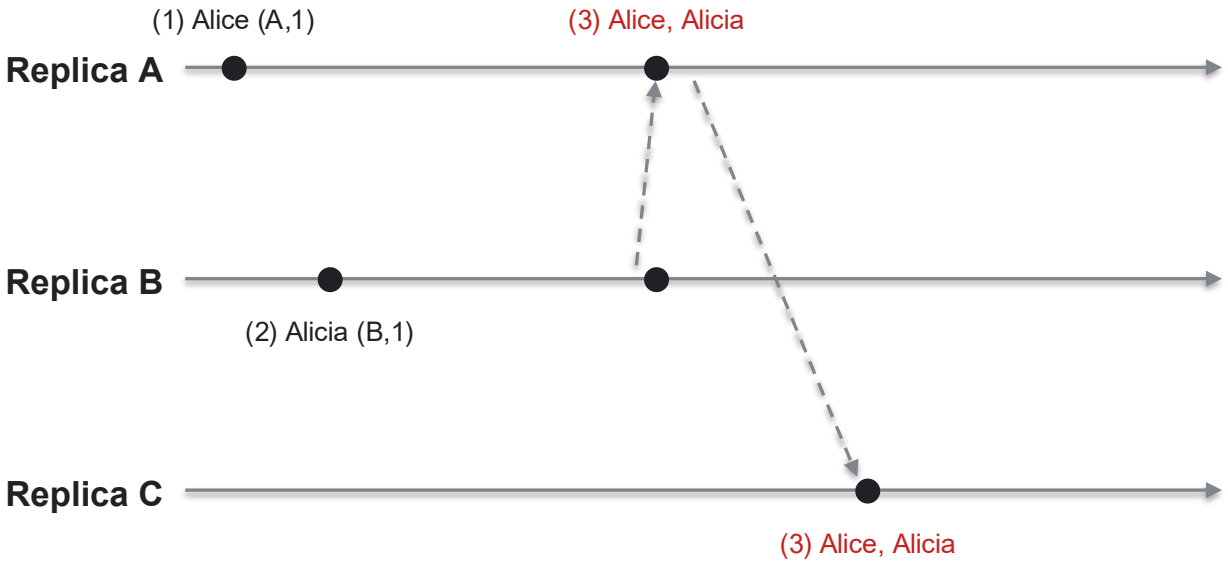
Causal context: updates that have been observed. **Active versions:** updates whose values have not been superseded;

Vectors list A, then B. Verified by code/v8_pseudocode_check.py; without the Known(C, d) test, Alice would come back.

DVV set operations

- **Read:** returns the active values and the causal context needed for a subsequent write;
- **Write(value, ctx):** creates a new dot and removes only the values the client has observed, i.e., those whose dots are in $H(\text{ctx})$;
- **Merge:** combines the causal contexts, and keeps a version if the other replica still has it active or has never observed it. Otherwise, the other replica saw it and dropped it, so it was superseded;
- Caveats:
  - the dot must be unique: the coordinator needs a safe counter per actor, including across crashes and restarts;
  - the client context must be trusted causal metadata, usually an opaque token issued by the database;
  - the coordinator must not treat all locally stored siblings as observed by the client, or it loses concurrent updates;
  - synchronization must carry the causal context, or an old sibling could reappear;

DVV sets in action (1/2): writes and merge



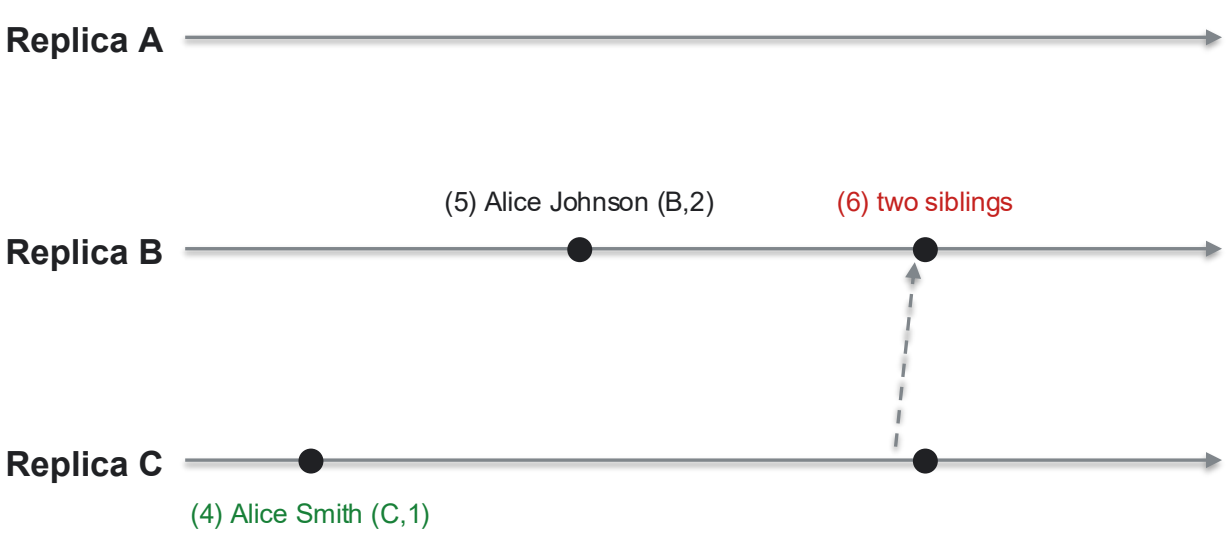
1. Client 1 PUTs "Alice" at A with ctx [0,0,0]: dot (A,1). A: C = [1,0,0], E = {((A,1), Alice)};

2. Client 2 PUTs "Alicia" at B with ctx [0,0,0]: dot (B,1). B: C = [0,1,0], E = {((B,1), Alicia)};

3. A and B merge: neither dot is in the other's context, so both survive: C = [1,1,0], E = {((A,1), Alice), ((B,1), Alicia)}. C receives the same state;

Key customer:123; vectors list A, B, C. Both values survive as siblings, without physical timestamps;

DVV sets in action (2/2): a stale client



4. Client 3 reads {Alice, Alicia} with ctx [1,1,0] and PUTs "Alice Smith" at C: both dots are in H(ctx), so both are removed; new dot (C,1): C = [1,1,1], E = {((C,1), Alice Smith)};

5. A stale client still holds ctx [1,1,0] and PUTs "Alice Johnson" at B: dot (B,2). (C,1) is not in its context, so it cannot supersede Alice Smith;

6. B and C merge: C = [1,2,1], E = {((C,1), Alice Smith), ((B,2), Alice Johnson)}: two siblings, as expected;

The database cannot simply use its current local state to decide which siblings to discard: only the client's context can;

VV, DVV, and DVV sets compared

	Version vectors (VV)	Dotted version vectors (DVV)	DVV sets (DVVS)
Main innovation	Compact representation of causal history using per-actor counters	Adds a unique dot to distinguish an individual update from its causal history	Associates individual dots with values, and keeps a shared causal context for a set of siblings
Representation	VV	(VV, dot)	(causal context, set of (dot, value))
Concurrent writes	Detects concurrency between independently tracked versions	Distinguishes concurrent updates even when they share a server actor	Preserves concurrent values as separate siblings
Conflict resolution	Application or storage-layer logic	Application or storage-layer logic	Merge rules discard superseded siblings; the application resolves genuine conflicts
Main limitation	One VV for several versions can lose information about their individual histories	Describes one version; several siblings need extra bookkeeping	More complex state management, merging, and dot allocation
Metadata	O(n) per vector	O(n) plus one dot per version	O(n + s) for s active siblings
When to use	Causality tracking where each version has an accurately maintained VV	Precise causality tracking for individual versions in optimistic replication	Multi-version replicated KV stores with siblings and client-driven conflict resolution

n = number of tracked actors; s = number of active siblings. Metadata bounds exclude application values;

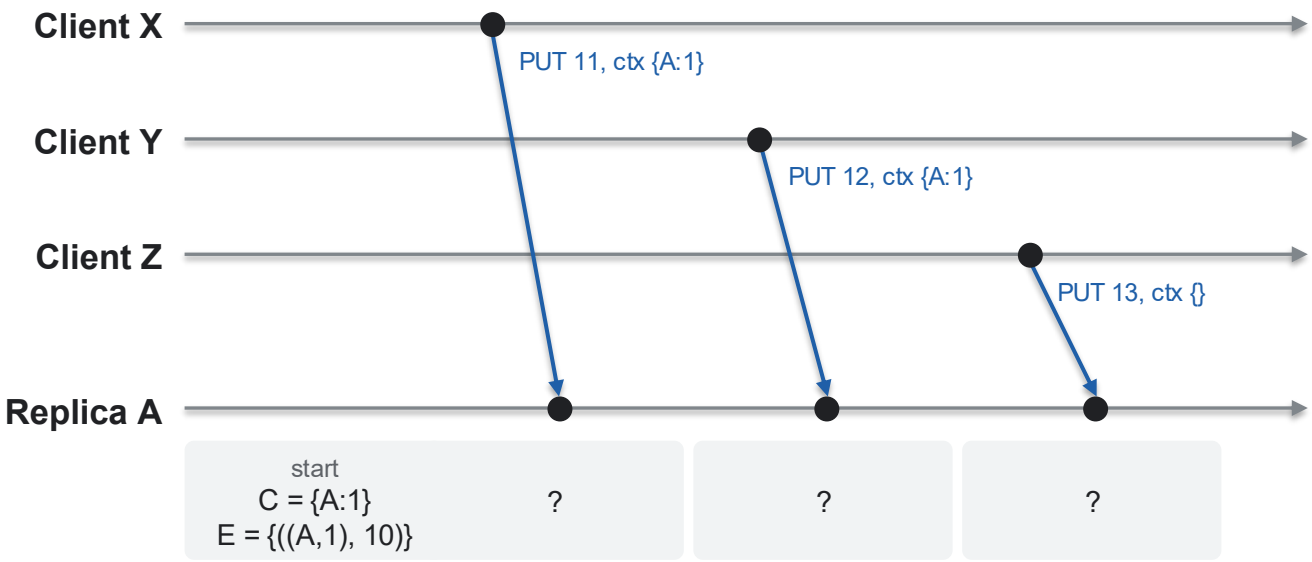
What dotted version vectors buy us

- "VV tracks causal history. DVV identifies an individual update precisely. DVVS uses that precision to maintain and reconcile concurrent versions without inventing causal dependencies;"
- DVVs do not eliminate genuine write conflicts: they eliminate false conflicts caused by insufficient causal information, so the system can identify genuine concurrency accurately;
- The order in which a server processes writes is not necessarily the causal order of the client operations that produced them!
- DVV and DVVS do not replace version vectors: they build on them to solve a more specific problem in optimistic replication;

Dotted version vectors in practice

- **Riak** (2.0+) can use DVVs to track siblings precisely with server ids;
- On conflict Riak **keeps siblings** instead of picking a winner;
- Resolution is a separate choice: application/client merge, last-write-wins, or CRDT data types;
- Read repair and anti-entropy **propagate** versions; they do not resolve conflicts;
- **DVV sets** compact all siblings of a key under one shared context (Almeida et al., DAIS 2014);
- "…identifiers have to be globally unique." Replica ids or client ids both work;

Exercise 6: dotted version vector sets



A final read at A returns: ?

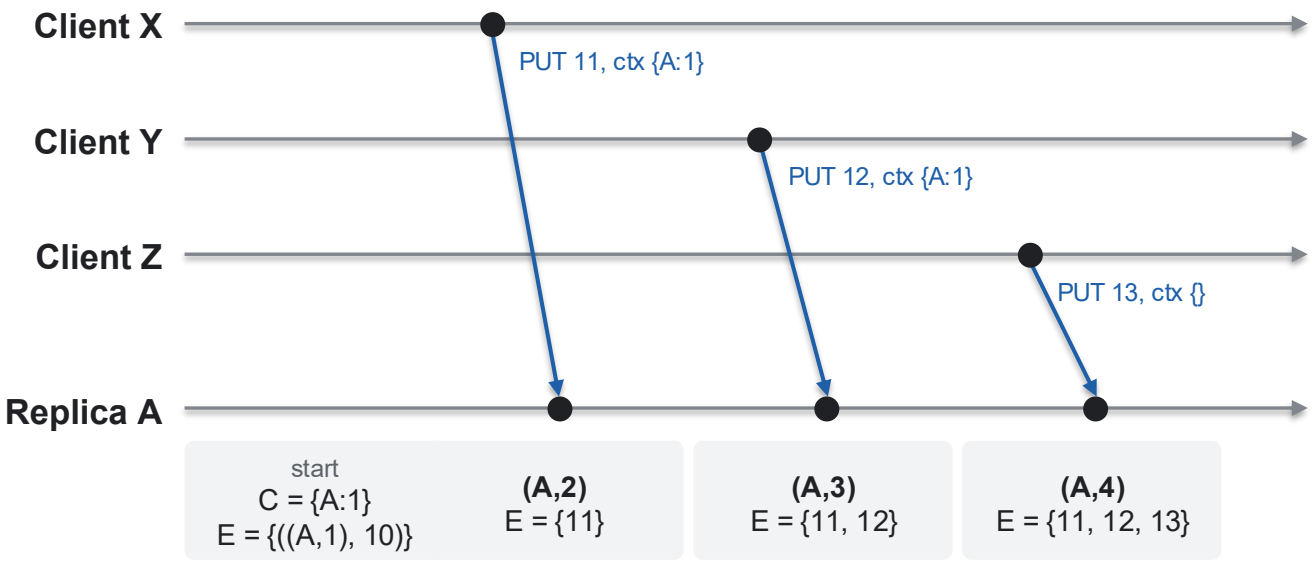
Replica A starts with C = {A:1} and E = {((A,1), 10)};

1. Which dot does each write get?

2. Which values does A keep after each write?

3. What does the final read return?

Exercise 6: answers



A final read at A returns: 11, 12, 13 with C = {A:4}

X: (A,1) ∈ H({A:1}), so 10 is removed; dot (A,2);

Y: (A,2) ∉ H({A:1}), so 11 stays; dot (A,3);

Z: nothing is in H({}); dot (A,4);

The read returns 11, 12, and 13 with C = {A:4}: three siblings;

Matrix Clocks

What do the others know?

Matrix clocks: knowing what others know

- A vector clock tells N_i what **it** knows. Sometimes N_i must know what **every other node** knows;
- Example: a log entry or message can be discarded once **everyone** has seen it;
- Each node keeps an $n \times n$ matrix M_i:
  - row i: N_i's own vector clock
  - row j: N_i's latest knowledge of N_j's vector clock
  - $M_i[j, k]$: how many of N_k's events N_i knows that N_j has seen
- Michael & Fischer (1982); used by Wuu & Bernstein and Sarin & Lynch to discard obsolete data in replicated databases;

Matrix clock in pseudocode

Algorithm 4: Matrix Clock

Data: Node N_i with matrix $M_i[1..n, 1..n] \leftarrow 0$ *// row i is N_i's own vector clock*

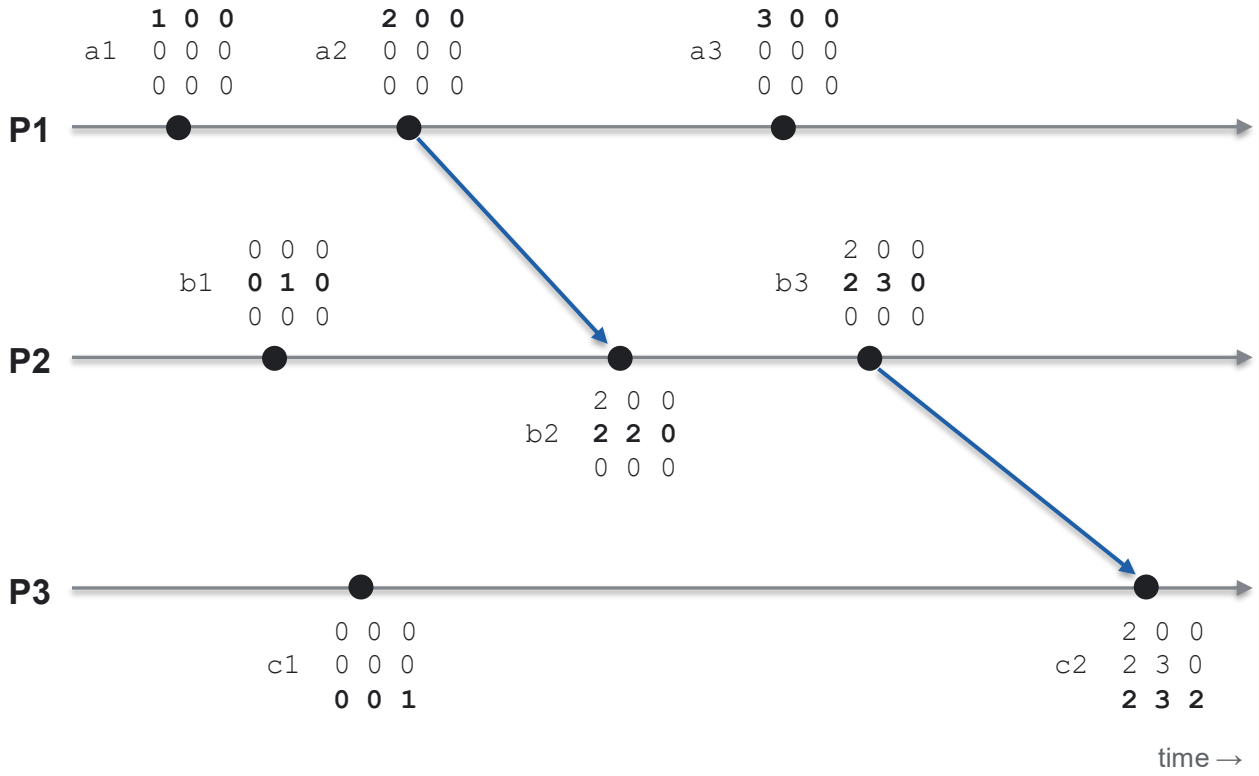
function LocalEvent()
- $M_i[i, i] \leftarrow M_i[i, i] + 1$
- **return** M_i

function Send(m, N_j)
- $M_i[i, i] \leftarrow M_i[i, i] + 1$
- transmit $\langle m, M_i \rangle$ to N_j
- **return** M_i

function KnownByAll(l, t) *// all have seen N_l's first t?*
- **return** $\min_k M_i[k, l] \geq t$

function Receive($\langle m, M_M \rangle$, j) *// m sent by N_j*
- **for** k ← 1 **to** n **do** *// own row ← sender's row*
  - $M_i[i, k] \leftarrow \max(M_i[i, k], M_M[j, k])$
- **for** k ← 1 **to** n **do**
  - **for** l ← 1 **to** n **do**
    - $M_i[k, l] \leftarrow \max(M_i[k, l], M_M[k, l])$
- $M_i[i, i] \leftarrow M_i[i, i] + 1$
- deliver m
- **return** M_i

Matrix clocks on the same execution



Bold row = own vector clock

b2: P2 copies P1's row from m1 ⇒ row P1 = 2 0 0

c2: column P1 = (2, 2, 2) ⇒ P3 knows **everyone** has seen a1 and a2

column P2 min = 0 ⇒ P3 cannot assume P1 has seen b1

What matrix clocks give you

- Everything a vector clock gives (row i);
- $\min_k M_i[k,l] \geq t \Rightarrow$ N_i knows every node has seen N_l's first t events;
- Use: garbage-collect logs and buffered messages, detect **stable** messages;
- Cost: $O(n^2)$ per node and per message: use only when this second-order knowledge pays off;
- For plain causality checks, compare rows; they are ordinary vector clocks;

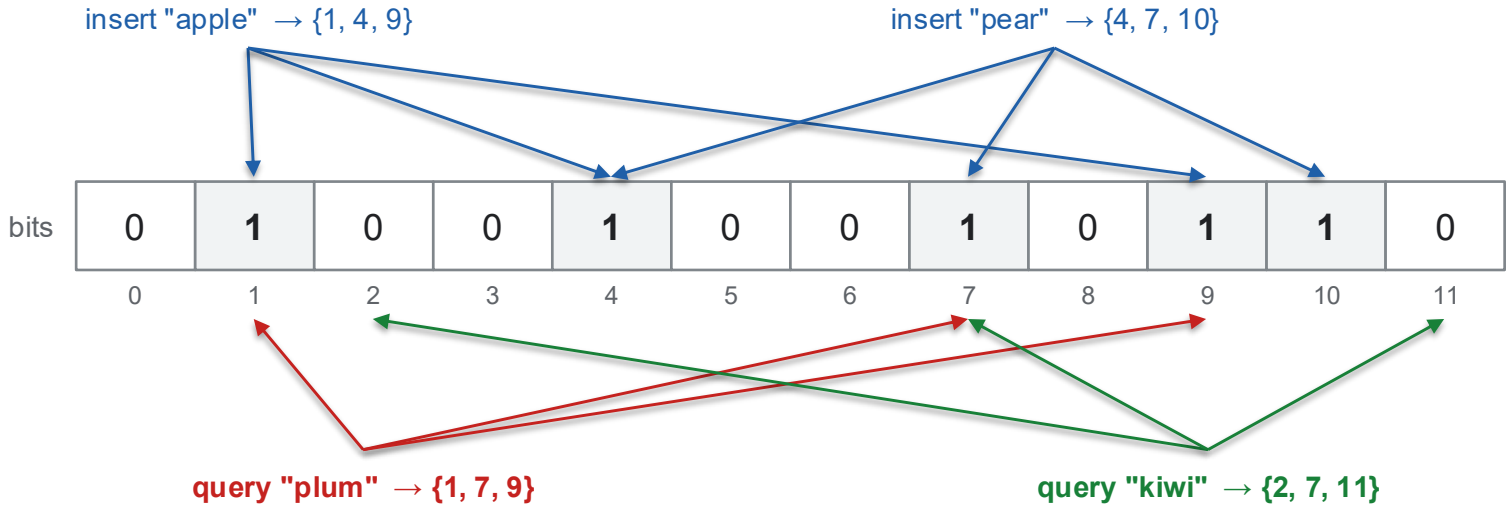
Bloom Clocks

Constant size, probabilistic answers

Recap: Bloom filters

- Probabilistic **set membership** in a fixed-size array;
- Insert: hash the key with k hash functions and set those positions;
- Query: all k positions set ⇒ **maybe** present; any position 0 ⇒ **definitely** absent;
- No false negatives, some **false positives;**
- **Counting** Bloom filter: counters instead of bits;

A Bloom filter, step by step



Counting Bloom filter: position 4 would hold 2, because both apple and pear hash to it. Removing pear decrements it back to 1, which a plain bit array cannot do;

Bloom clocks: representation and rules

- Each node keeps a counting Bloom filter B of **s** counters (s is fixed and independent of the number of nodes);
- Every event gets a unique id (node id + local counter), hashed with k hash functions;
- **Local event or send:** increment the k counters of the event's id; a send carries B;
- **Receive B′:** $B \leftarrow \max(B, B')$ component-wise, then count the receive event;
- Like a vector clock whose entries are **shared** by many events;
- Also called Bloom vector clocks (Gakhov, Beyond Clocks);

Bloom clock in pseudocode

Algorithm 5: Bloom Clock

Data: Node N_i with counters $B_i[0..s-1] \leftarrow 0$ **and** event counter $n_i \leftarrow 0$
Data: k hash functions $h_1, \ldots, h_k$: event id $\rightarrow 0..s-1$; $A \leq B$ means $A[x] \leq B[x]$ **for** every x

function Tick()
    $n_i \leftarrow n_i + 1$
    $e \leftarrow (i, n_i)$ *// unique event id*
    for $r \leftarrow 1$ **to** k **do**
        $B_i[h_r(e)] \leftarrow B_i[h_r(e)] + 1$
    return B_i

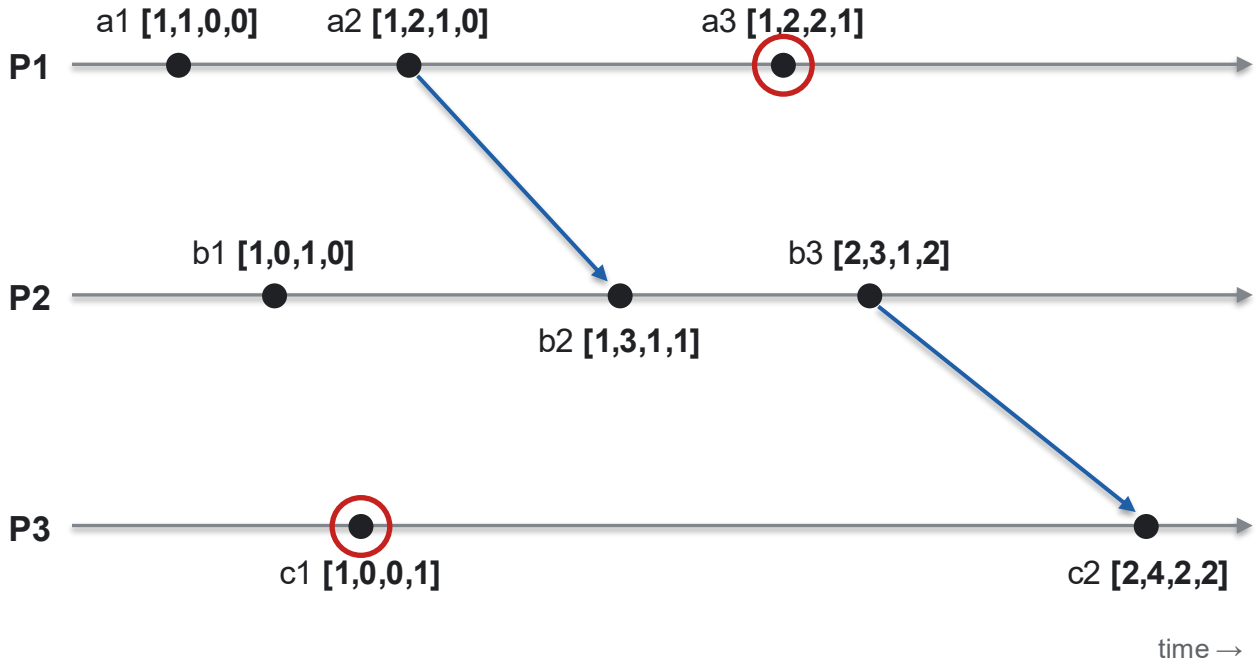
function LocalEvent()
    return Tick()

function Send(m, N_j)
    $t \leftarrow$ Tick()
    transmit $\langle m, t \rangle$ to N_j
    return t

function Receive($\langle m, B_M \rangle$)
    for $x \leftarrow 0$ **to** $s - 1$ **do**
        $B_i[x] \leftarrow \max(B_i[x], B_M[x])$ *// merge*
    $t \leftarrow$ Tick() *// the receive is an event too*
    deliver m
    return t

function MaybeBefore(A, B) *// may be a false positive*
    return $A \leq B$ **and** $A \neq B$
function SurelyConcurrent(A, B) *// never a false negative*
    return not $(A \leq B)$ **and not** $(B \leq A)$

Bloom clocks on the same execution



$a2 \rightarrow b3$: $[1,2,1,0] \leq [2,3,1,2]$ ✓

a3 vs b3 incomparable ⇒ **surely** concurrent ✓

$c1[1,0,0,1] \leq a3[1,2,2,1]$ looks like $c1 \rightarrow a3$, but $c1 \parallel a3$: a **false positive**

With s = 4, 6 of the 12 concurrent pairs look ordered

Illustrative hashes (s = 4, k = 2): a1→{0,1} a2→{1,2} a3→{2,3} b1→{0,2} b2→{1,3} b3→{0,3} c1→{0,3} c2→{1,2}

What a Bloom clock can and cannot tell you

- $a \to b \Rightarrow B(a) \leq B(b)$, always: **no false negatives;**
- $B(a) \parallel B(b) \Longrightarrow a \parallel b$ **for certain;**
- $B(a) \leq B(b) \Rightarrow$ **probably** $a \to b$ (a false positive when $a \parallel b$);
- False positives grow as counters fill up; larger s lowers the rate (Ramabaja gives an estimate; Misra and Kshemkalyani analyze it);
- Size $O(s)$, fixed and independent of the number of nodes;
- Use when nodes are many or dynamic and approximate answers are acceptable;

Beyond Vectors

A quick survey: exact vs approximate, fixed vs dynamic

Variations on vector clocks

Mechanism	Main idea	Motivation
Sparse vector clocks	Store only nonzero or relevant components	Reduce metadata for sparse causal histories
Differential vector clocks	Communicate changes to vectors rather than entire vectors	Reduce communication overhead
Interval Tree Clocks (ITC)	Combine causal histories with dynamically allocated identities	Support processes that fork and terminate
Plausible clocks	Map multiple processes to fewer vector components	Trade precision for smaller metadata
Bloom clocks	Use Bloom-filter-inspired structures to encode causality	Approximate causality with bounded metadata
Tree clocks	Organize causal information in a tree structure	Accelerate clock operations in concurrency analysis
Epoch-based clocks	Represent common cases with a single actor-counter pair	Avoid full vector operations when possible
Matrix clocks	Track information about what other processes know	Support higher-order knowledge and distributed garbage collection

Other compact and dynamic clocks

- **Plausible clocks:** map many processes onto fewer entries; always consistent with →, but may order concurrent events;
- **Hybrid vector clocks:** vectors of physical-time entries, compressed when entries are within ε;
- **Interval tree clocks:** identities that fork and join as nodes come and go;
- **Tree clocks, FastTrack epochs:** faster joins for dynamic race detection;
- **Replay clocks (RepCl):** combine ideas from HLC and vector clocks to replay a distributed computation offline, without forcing an order on concurrent events;

Computing causal paths

- Number the events $1 \ldots n$ and build the adjacency matrix of the one-step relation: $A_{\mathrm{ij}} = 1$ if $e_i \rightsquigarrow e_j$, and 0 otherwise;
- Counting paths: entry (i, j) of A^k is the number of causal paths of length exactly k from e_i to e_j;
- Reachability: $R = A \vee A^2 \vee \cdots \vee A^{n-1}$, and $e_i \to e_j \Leftrightarrow R_{\mathrm{ij}} = 1$. Stopping at $n - 1$ is enough: happened-before has no cycles, so no path visits more than n events;
- Warshall's recurrence computes R in $O(n^3)$: start from $R^{(0)} = A$ and allow one more intermediate event e_k per round: $R_{\mathrm{ij}}^{(k)} = R_{\mathrm{ij}}^{(k-1)} \vee (R_{\mathrm{ik}}^{(k-1)} \wedge R_{\mathrm{kj}}^{(k-1)})$;
- To recover the path itself, record a predecessor for each event during a BFS or DFS over A, then walk back from b to a;
- At run time we usually don't have the whole graph of events: vector clocks store this path information in each event's timestamp;

Summary: what each mechanism keeps and loses

Mechanism	Guarantee	Loses / limitation	Metadata	Typical use
Causal history	$x \rightarrow y \Leftrightarrow H(x) \subset H(y)$	nothing, but unbounded	O(events)	reference model
Lamport	$a \rightarrow b \Rightarrow L(a) < L(b)$	concurrency (converse fails)	O(1)	total order, mutual exclusion
Vector clock	$a \rightarrow b \Leftrightarrow V(a) < V(b)$	nothing, for a fixed N	O(N)	causal delivery, race detection
Matrix clock	row i = vector clock; column min $\geq t \Rightarrow$ all have seen	nothing; adds what others know	$O(n^2)$	garbage collection, stable messages
Bloom clock	$a \rightarrow b \Rightarrow B(a) \leq B(b)$	certainty: false positives	O(s), fixed	large or dynamic systems
HLC	$a \rightarrow b \Rightarrow hlc(a) < hlc(b)$; $l - pt \leq \varepsilon$	concurrency; needs bounded skew	O(1): (l, c)	DB / MVCC timestamps
Version vector	dominance $\Leftrightarrow$ includes the other's updates	same-server concurrency (per-server ids)	O(replicas) per object	detect conflicting versions
Dotted VV	dot = this write; context = what its writer saw	more complex representation	O(replicas) + dot	precise siblings (Riak)

Takeaways

- Wall-clock timestamps order by **clock reading**, not by **cause;**
- Happened-before is a **partial** order: concurrency is normal, not an error;
- Every clock is a **compression of causal histories**: ask what it throws away;
- Lamport: cheap, one direction · Vector: exact, $O(n)$ · Matrix: what others know, $O(n^2)$ · Bloom: fixed size, false positives · HLC: causal and near real time;
- Replicated data: version vectors detect conflicting versions; dotted VVs keep same-server concurrent writes apart;
- A clock is a **representation**, not a consistency model, and not a merge policy;

References

- L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM, 1978;
- C. Fidge (1988) and F. Mattern (1989): vector clocks. D. S. Parker et al. (1983): version vectors;
- M. Raynal, M. Singhal. Logical Time: Capturing Causality in Distributed Systems. IEEE Computer, 1996 (matrix clocks);
- L. Ramabaja. The Bloom Clock. arXiv:1905.13064, 2019;
- A. Misra, A. D. Kshemkalyani. The Bloom Clock for Causality Testing. arXiv:2011.11744, 2020;
- I. Lagwankar, S. S. Kulkarni. Replay Clocks. arXiv:2311.07842, 2023;
- I. Lagwankar. Tracing Distributed Algorithms Using Replay Clocks. MSc thesis, Michigan State University, 2024 (arXiv:2407.00069);
- C. Baquero, N. Preguiça. Why Logical Clocks Are Easy. CACM 59(4), 2016;
- S. Kulkarni, M. Demirbas et al. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases, 2014;
- N. Preguiça, C. Baquero et al. Dotted Version Vectors: Logical Clocks for Optimistic Replication. arXiv:1011.5808, 2010;
- P. S. Almeida et al. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. DAIS 2014;
- G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. SOSP 2007 (§4.4);
- R. Taft et al. CockroachDB (SIGMOD 2020); J. Corbett et al. Spanner (OSDI 2012, TOCS 2013);
- P. S. Almeida, C. Baquero, V. Fonte. Interval Tree Clocks. OPODIS 2008;
- Books: Tanenbaum & van Steen, Distributed Systems 4e, §5.2 (pp. 260–274), §7.4.4; Kleppmann & Riccomini, DDIA 2e; Gakhov, Beyond Clocks (p. 89, Bloom vector clocks); Coulouris et al. 5e, ch. 14;

Obrigado!

Deep Dive: from Lamport to Bloom Clocks