READ MODE · ANSWER FIRST

Distributed systems time, causality, and ordering explained

Learn happens before, Lamport and vector clocks, causal consistency, clock uncertainty, HLCs, and TrueTime style APIs.

CHEAT SHEET · 15 ESSENTIAL IDEAS

The whole story in 15 lines

Distributed time is a toolbox: track causes logically, expose uncertainty physically, and pay for stronger order only when the promise...

1. Happens before is reachability through local order and messages, not a comparison of wall clock readings.
2. A total order may be useful for agreement but its tie breaks are policy rather than newly discovered causality.
3. If x happens before y then the Lamport value of x is smaller than the Lamport value of y.
4. The implication works only from causality to clock order, never automatically in reverse.
5. On receive, take the componentwise maximum then increment the receiver own component.
6. Opposing components reveal concurrency that a scalar clock collapses into an arbitrary sequence.
7. The metadata grows with the participant set even when one event changes only one component.
8. Concurrent versions must survive until application logic reconciles their values.
9. A comment written after reading a post depends on that post even if the network later reorders both values.
10. Effects wait in a buffer while concurrent operations may still flow in either order.
11. Overlapping time intervals cannot prove which real event occurred first.
12. The physical part follows wall time when it can while the logical part advances when physical values tie.
13. The HLC pair keeps rising along a causal path even when the operating system clock moves backward.
14. Waiting until a commit timestamp is definitely in the past makes later real-time transactions receive later timestamps.
15. Use the weakest guarantee that proves the user-visible promise because stronger guarantees add metadata, waiting, or coordination.

Setup

DISTRIBUTED TIME TOOLBOX

KEY TERMS

EVENT	One local action
HAPPENS BEFORE	A can influence B
CONCURRENT	No causal path either way
LOGICAL TIME	Causal progress, not seconds
PHYSICAL TIME	An estimate of wall time
UNCERTAINTY	The clock error bound

THE JOURNEY

01 SETUP

Imagine three replicas recording events without one perfect shared clock. How can we recover the order that matters without pretending we know more than we do?

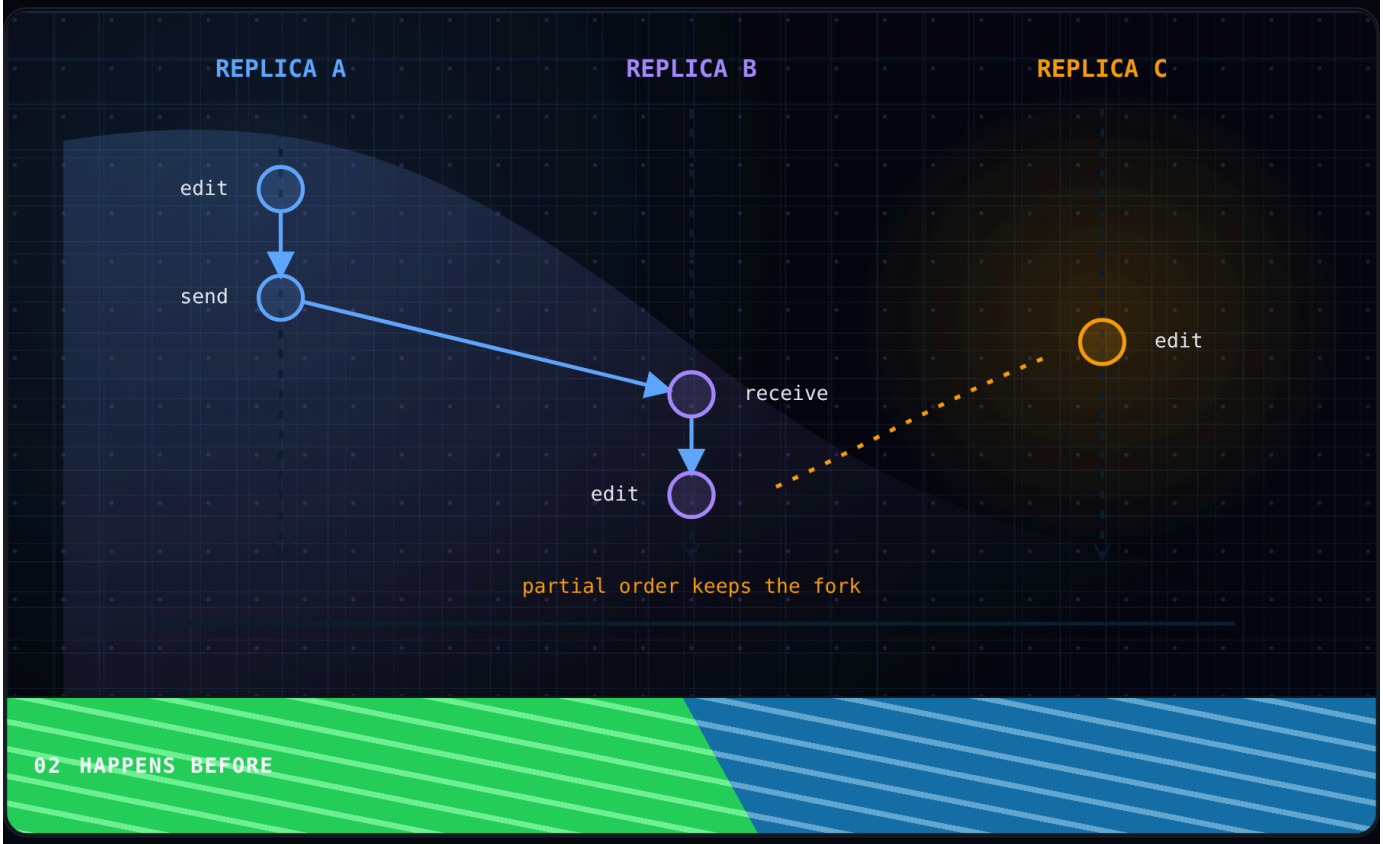
An event is one local action and “Happens before” means one event could affect another. Concurrent events have no path of cause and effect between them.

Logical time tracks causal progress and physical time estimates real clock time, and uncertainty tells us how far that estimate might be from reality.

We will build each kind of clock, then test its limits. This way, every clock gets one clear job instead of becoming a vague idea of time.

Each tool answers a different ordering question, so let us begin with the order that the execution itself can actually prove.

Happens Before



We separated cause from clock time and now look at the three lanes for replicas A, B, and C. Moving down a lane follows that replica's program.

Replica A edits some data and then sends a message because its program did those steps in order, the edit happens before the send.

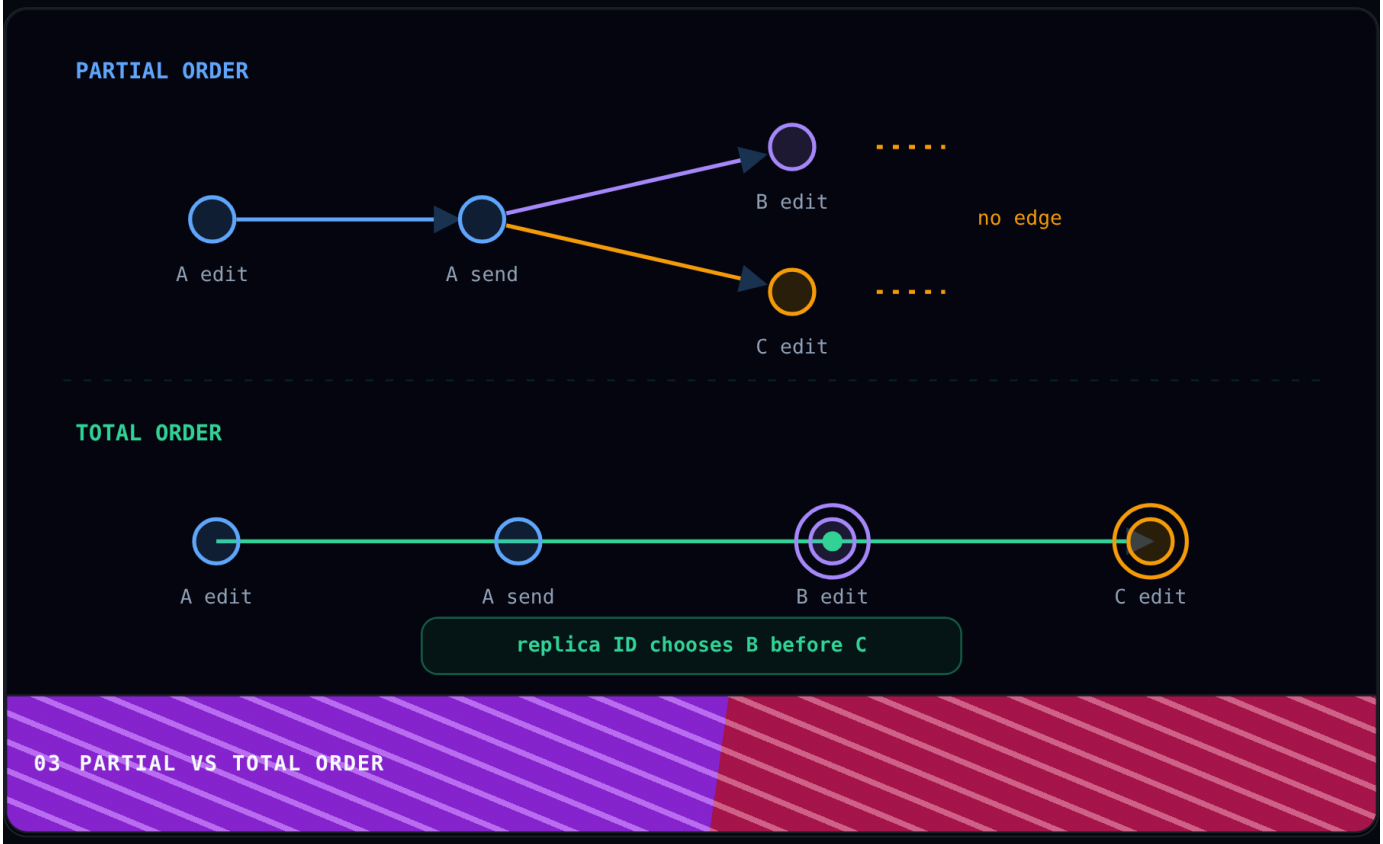
Replica B receives that message and then edits. The message connects the lanes, creating one causal path from A's edit to B's edit.

Replica C edits while disconnected, so can you find any local step or message that proves whether B's edit or C's edit came first?

You cannot prove an order because neither edit can reach the other through local steps or messages, which means the two edits are concurrent.

This “happens before” relation gives us a partial order. Next, we will compare it with a total order that forces every event into one sequence.

Partial vs Total Order



“Happens before” left one fork unresolved and let us show the same events in two ways: an honest partial order and a chosen total order.

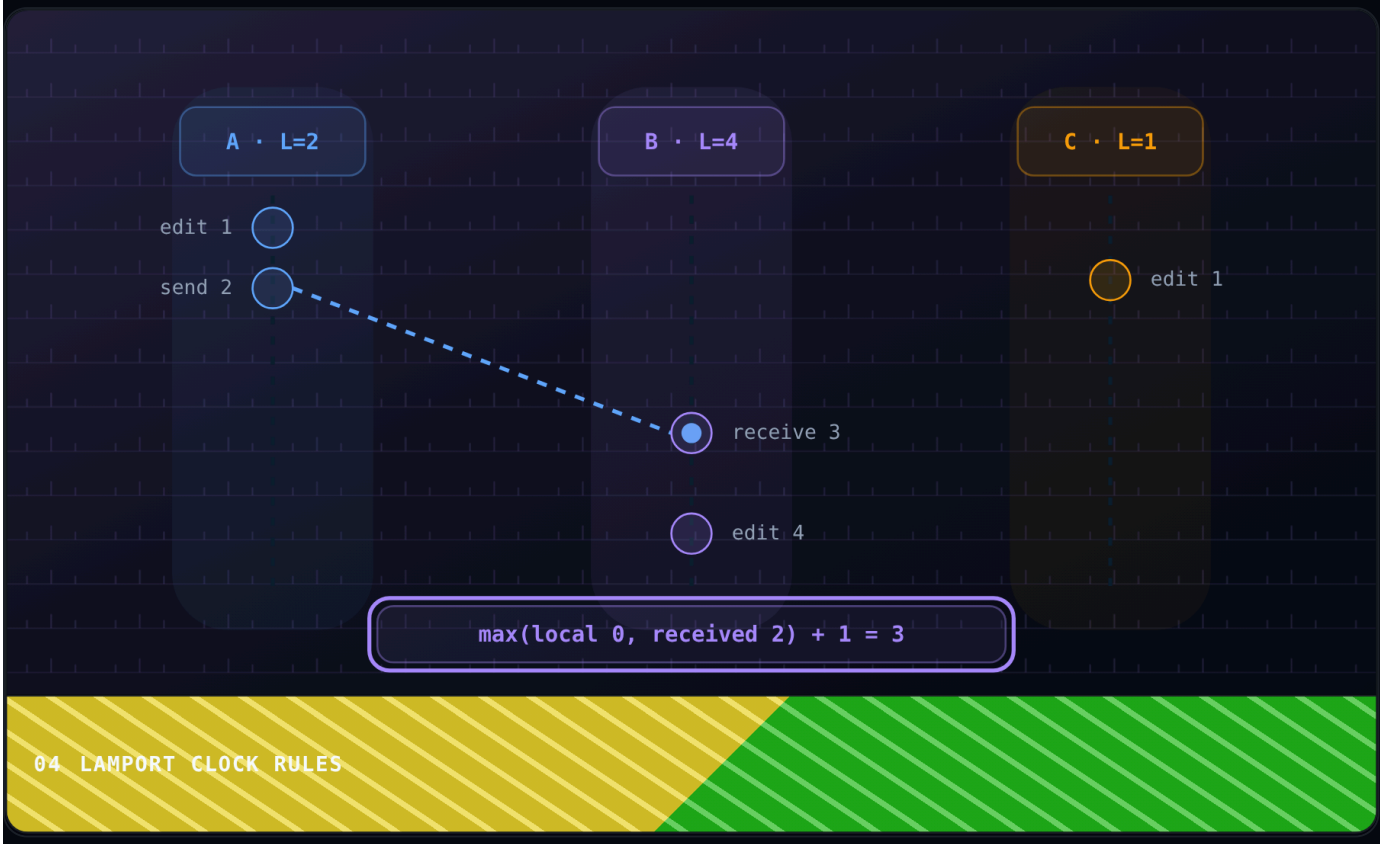
The partial graph keeps every relationship we can prove and its two open branches matter because those events may move forward independently.

Some protocols still need one exact replay sequence, so what extra relationship must they add between the two concurrent tips?

The total order adds a tie-breaking edge. Switch Tie break and a different valid sequence wins, even though the causal graph does not change.

A total order gives us an extra guarantee, and that guarantee has a cost, so next, we will build a small number that preserves causal direction.

Lamport Clock Rules



The causal graph is accurate, but it is bulky and a Lamport clock gives each replica one integer and increases it for every event.

Replica A labels its edit 1 and its send 2 and the message carries timestamp 2 across the network.

Change Message delay to move the receive event and the delay changes when the message arrives, but the message still carries timestamp 2.

Replica B's clock is 0 when timestamp 2 arrives, so what value will keep the receive event later than the send event?

B takes the larger value, 2, and adds 1 and the receive becomes 3, then B's next edit becomes 4.

Now we know the one-way rule: causes get smaller timestamps than their effects, so next, we will test the tempting reverse claim.

Lamport Clock Blind Spot

05 LAMPORT CLOCK BLIND SPOT

Lamport gave us a guarantee in one direction. Here two events carry different scalars and we ask what the numbers alone can prove.

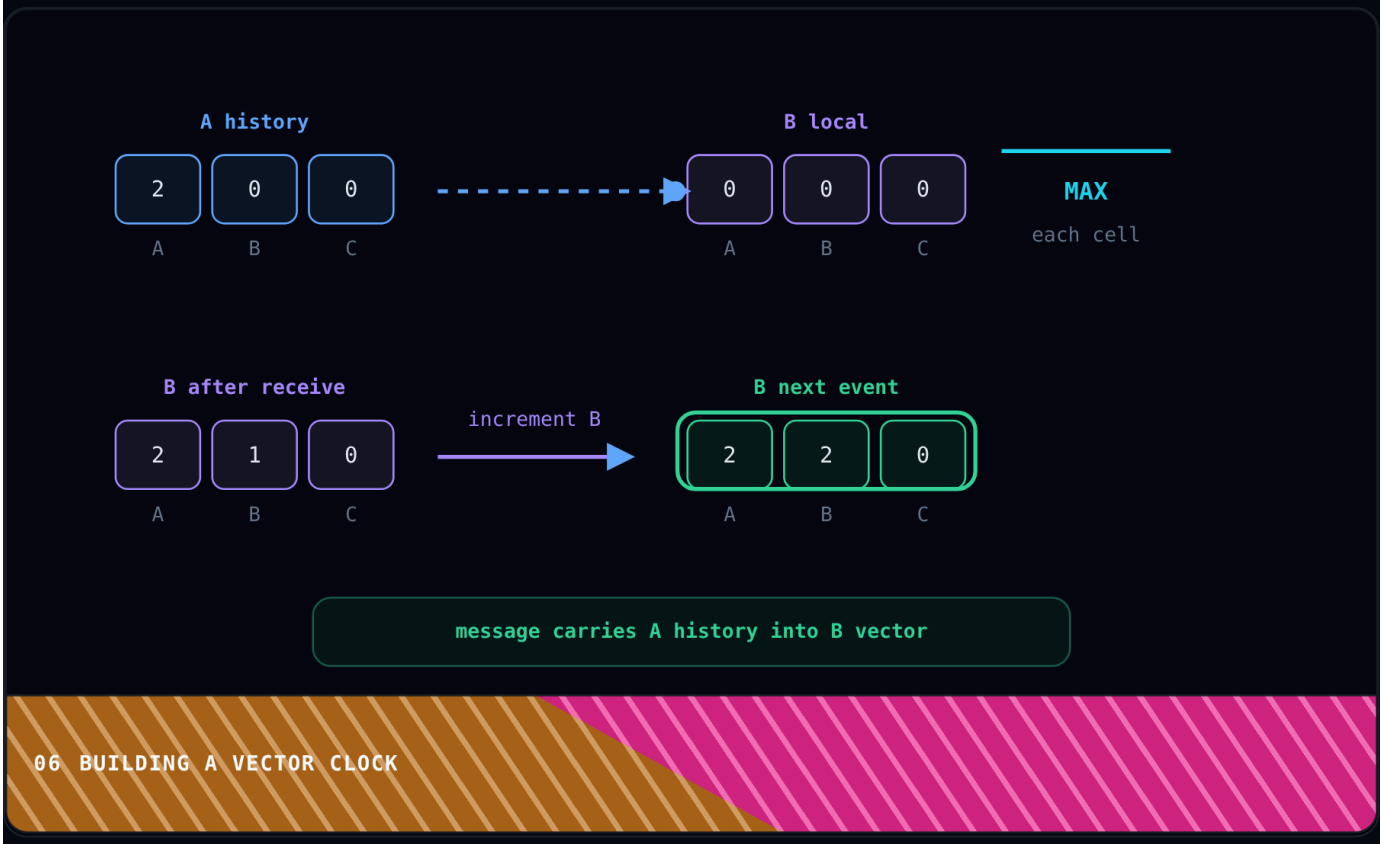
With Pair set to Causal the visible message path and the scalar order agree and the path is the proof of influence.

Switch Pair to Concurrent and notice that the numbers still compare after the message path disappears, so does 1 being less than 4 prove causality?

No, because the scalar order survives while the causal path disappears, which means Lamport clocks cannot distinguish concurrency from causality.

To recover the missing information we need more than one counter, so next each replica gets its own component.

Building a Vector Clock



06 BUILDING A VECTOR CLOCK

The Lamport blind spot came from compressing every history into one scalar and a vector keeps separate counters for A, B, and C.

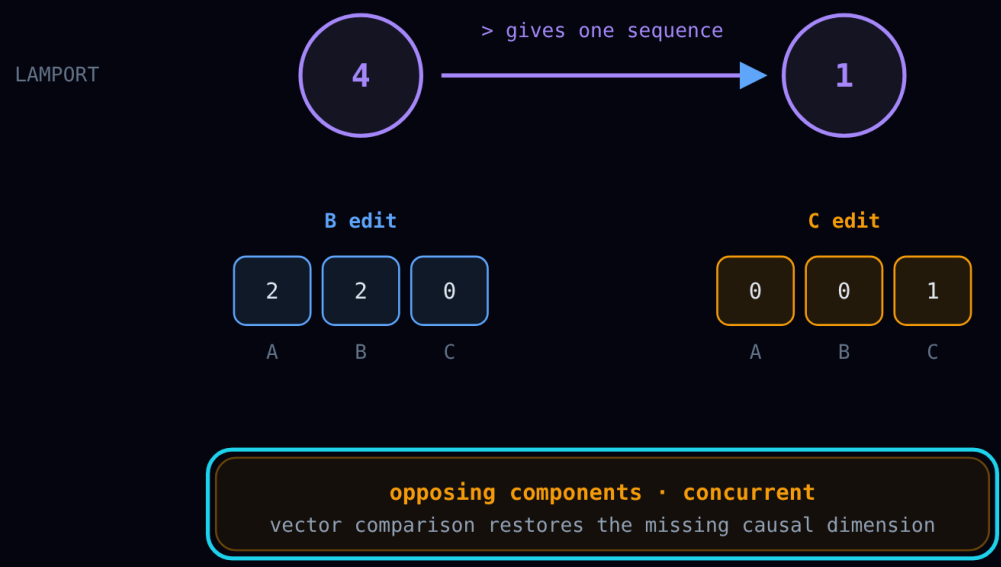
A increments only its own component and its edit becomes [1,0,0] and its send becomes [2,0,0].

B has its own local vector, so if the message arrives what should B keep from each side?

B takes the componentwise maximum then increments B and switch Message to Lost and the A history vanishes from B vector.

A vector is a compact causal history, so next we will compare two histories and let opposing components reveal concurrency.

Comparing Vector Clocks



07 COMPARING VECTOR CLOCKS

We built vectors by merging history and now B edit and C edit use the same encoding so their relationship is visible cell by cell.

B knows about A and B while C knows only about itself, so neither vector contains the complete history stored by the other.

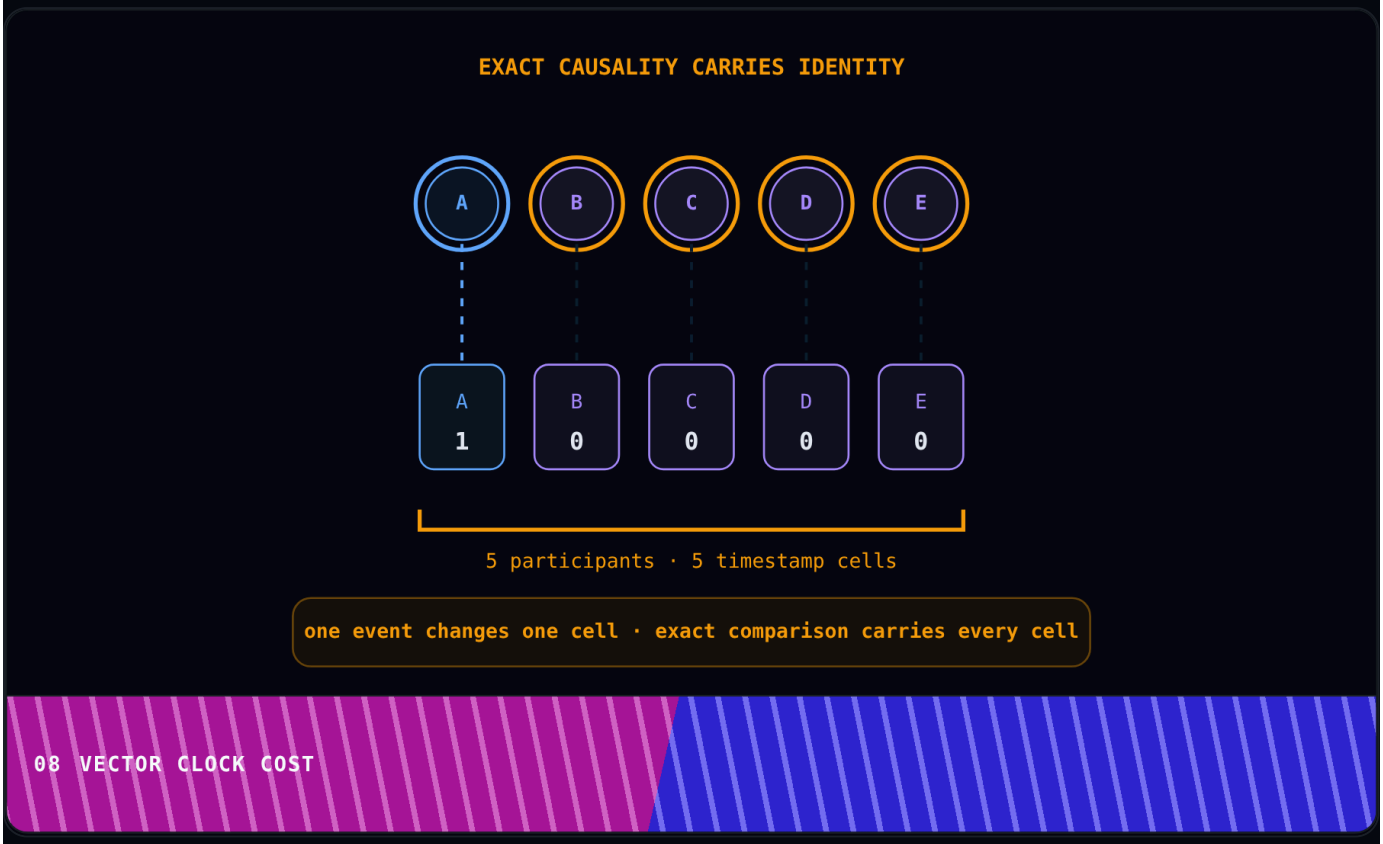
The scalar row still puts one event first, so does that single sequence match the component evidence below?

Compare every cell and you will see one history win in A and B while the other history wins in C.

Opposing directions mean the events are concurrent, but switching Pair to Causal makes every vector component point in the same direction.

Vectors recover exact causal comparison, but that accuracy costs metadata, so next we will watch the timestamp grow with the participant set.

Vector Clock Cost



08 VECTOR CLOCK COST

Exact causal knowledge is not free and each participant needs a stable cell in every full vector timestamp.

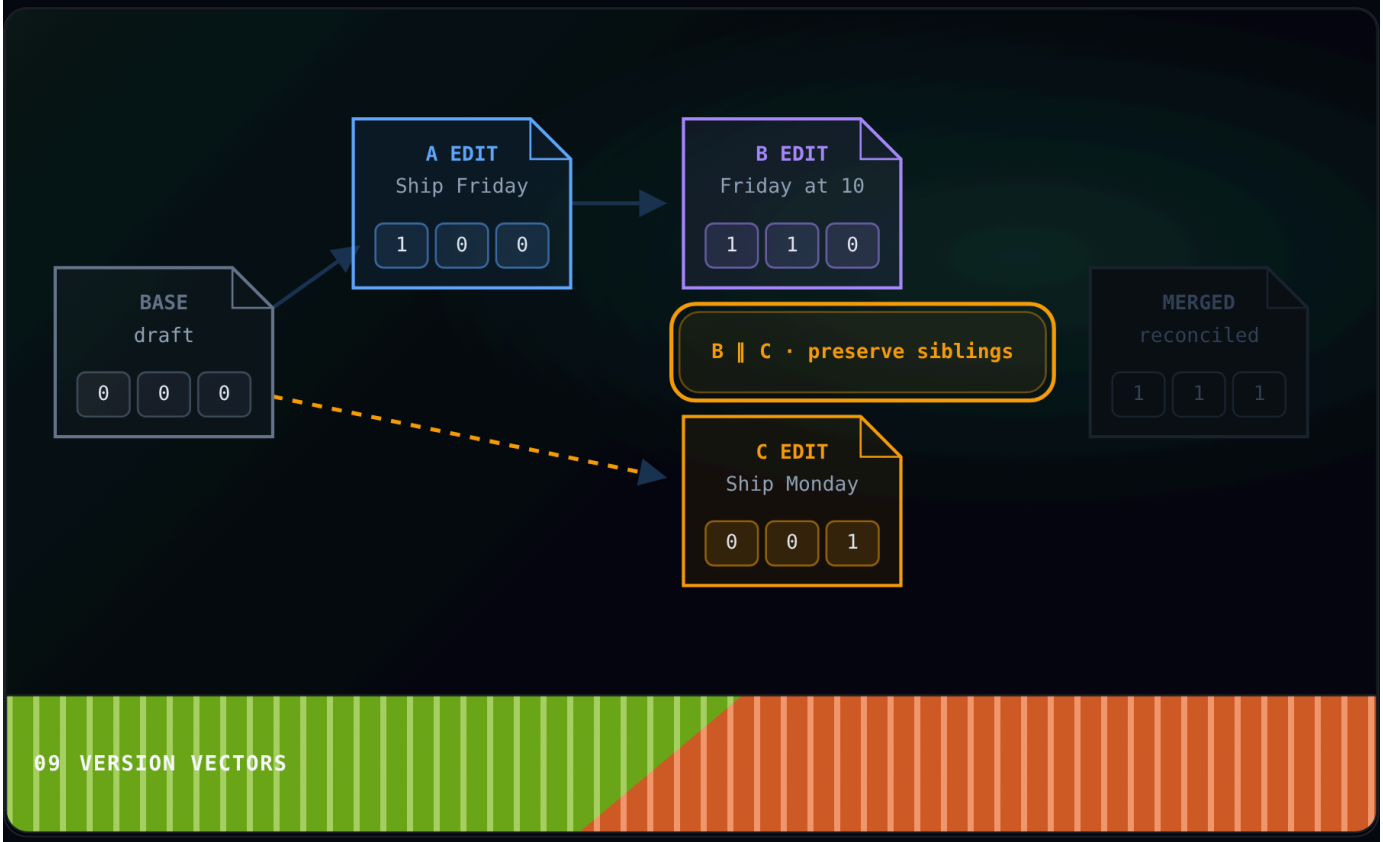
One event increments one cell but the timestamp still carries all known cells so another replica can compare histories.

What happens to the timestamp when the participant set grows from a few replicas to many identities?

The timestamp widens linearly, so increase Participants and watch the same event carry a longer strip even though only one component changes.

Vectors buy exact causality with growing metadata, so next version vectors apply that comparison to one replicated object.

Version Vectors



Vector clocks label events and a version vector labels the history of one object so a replicated store can compare actual values.

A creates [1,0,0], then B receives that history and edits it to [1,1,0], which means B safely supersedes A.

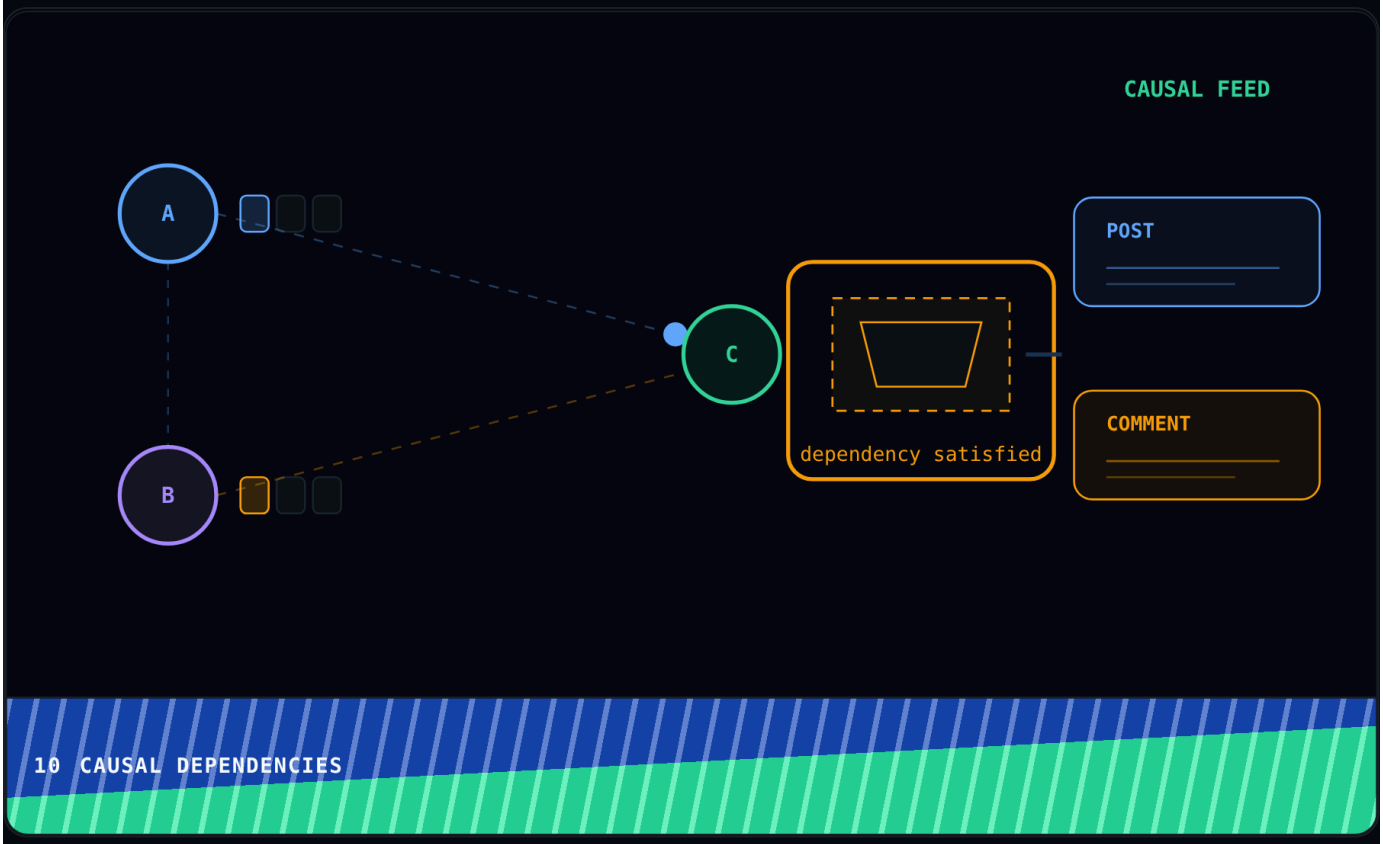
C edits its local copy and with Network partitioned it creates [0,0,1] without learning the A to B branch.

The replicas reconnect and compare both branches, so which version can they discard without losing an edit?

Neither version can be discarded because they are concurrent siblings, but switching Network to Connected turns C into one causal successor.

Version vectors preserve conflicting histories instead of losing an edit, so next we will turn one observed history into an explicit dependency.

Causal Dependencies



10 CAUSAL DEPENDENCIES

Version vectors told us which histories contain others and causal consistency starts when a new operation records the history it observed.

A publishes a post, then B reads it before writing a comment, which means the comment causally depends on the post.

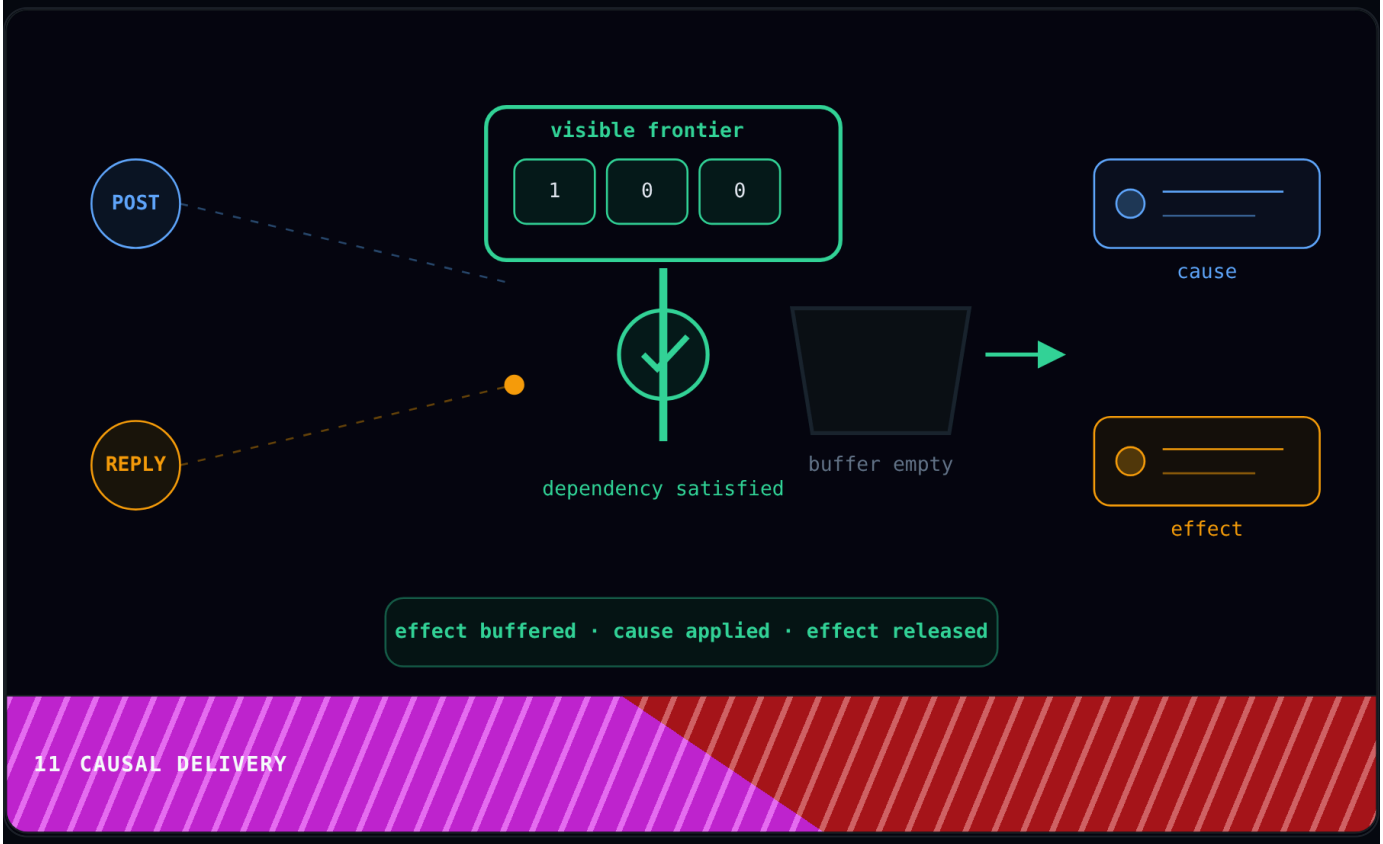
The network can deliver the small comment before the post and arrival order now disagrees with the dependency.

A reader asks for the feed while only the comment has arrived, so may the effect appear before its cause?

Causal delivery says the effect must wait for its cause, but switching Delivery to Arrival order exposes the broken feed that this rule prevents.

The dependency tells us what must wait, so next we will open the replica and watch its causal frontier enforce that rule.

Causal Delivery



The last stage named a dependency and this replica tracks applied history in a frontier vector.

When the effect arrives first its required A component is ahead of the frontier so the operation enters the holding buffer.

What changes when the missing cause advances the frontier to satisfy every required component?

The gate opens and the effect drains into the visible log and switch Arrival to Cause first and the buffer is never needed.

Causal delivery preserves dependencies without globally ordering concurrent work, so next we will ask physical clocks what they truly know.

Clock Uncertainty

Causal clocks avoid wall time and systems still need near-real timestamps for expiry, snapshots, and external ordering.

Replica clocks drift and synchronization takes time and each displayed reading is an illustrative estimate around one real timeline.

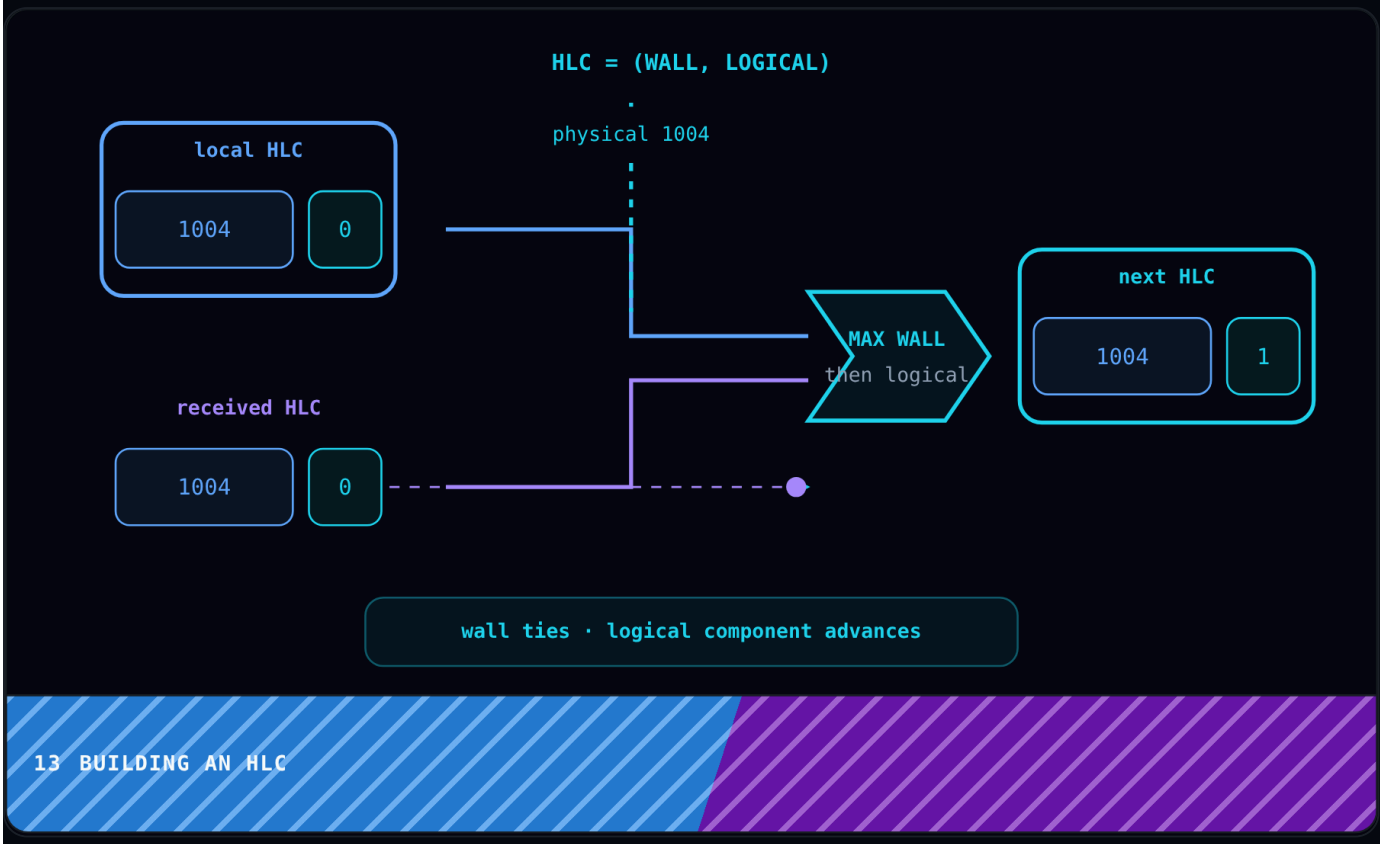
An honest API pairs each reading with plus or minus epsilon and use Uncertainty to widen the possible intervals.

Two center points look ordered but their intervals overlap, so can the clock prove which event happened first?

No, because either real order still fits the overlapping evidence, while only separated intervals can prove a physical before relationship.

Uncertainty is useful because it shows what the clock cannot prove, so next we will build a timestamp that also preserves causal progress.

Building an HLC



13 BUILDING AN HLC

Clock intervals describe uncertainty but many storage paths still need one sortable timestamp and HLC uses a pair instead of raw wall time.

When physical time advances the wall component follows it and the logical component resets.

A received timestamp has the same wall component as the local clock, so how can the next event remain strictly later?

The logical component increments when physical time cannot, so switch Physical clock between Advances and Ties to see which half carries progress.

HLC stays near wall time and preserves causal monotonicity, so next we will disturb the clock more violently by moving it backward.

HLC Under Clock Jumps

14 HLC UNDER CLOCK JUMPS

We just built the two-part timestamp and now A sends its HLC pair to B while the physical clock is disturbed.

A local and send events follow physical time and the message carries the complete pair across the network.

B receives that pair and chooses the greatest wall component before updating its logical component.

The operating system corrects B clock backward. Will the next HLC timestamp retreat too?

No, because setting Clock jump to Backward makes the logical part rise while the wall part holds, which keeps the causal path from retreating.

HLC repairs causal order inside a fixed-size timestamp, so next an interval API will expose uncertainty directly to transaction logic.

TrueTime Style APIs

HLC hides clock trouble inside a sortable pair and a TrueTime style API exposes uncertainty as [earliest, latest] for transaction logic.

Transaction one chooses the latest bound as its commit timestamp and that timestamp may still be slightly ahead of real time.

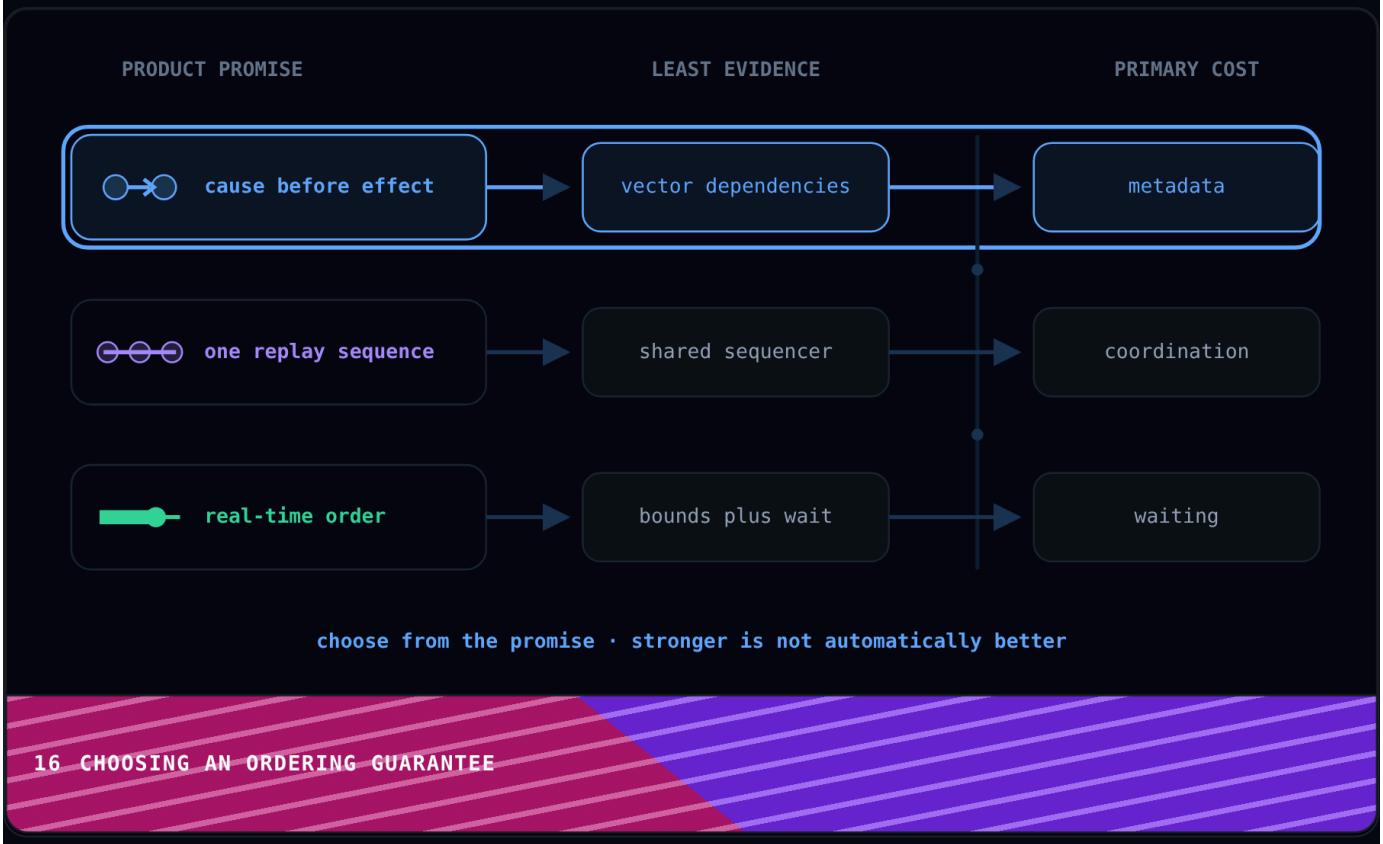
The transaction waits while the earliest possible current time is not past the chosen timestamp.

Transaction two starts after transaction one returns, so what stops their possible timestamp intervals from crossing?

Commit wait does and toggle Commit wait off to expose overlap and turn it on to make the later transaction start beyond the first timestamp.

The API spends bounded waiting to guarantee external order, so next we will choose which guarantee deserves that coordination cost.

Choosing an Ordering Guarantee



16 CHOOSING AN ORDERING GUARANTEE

We now have logical clocks and bounded physical time and the final design question starts from the behavior users require.

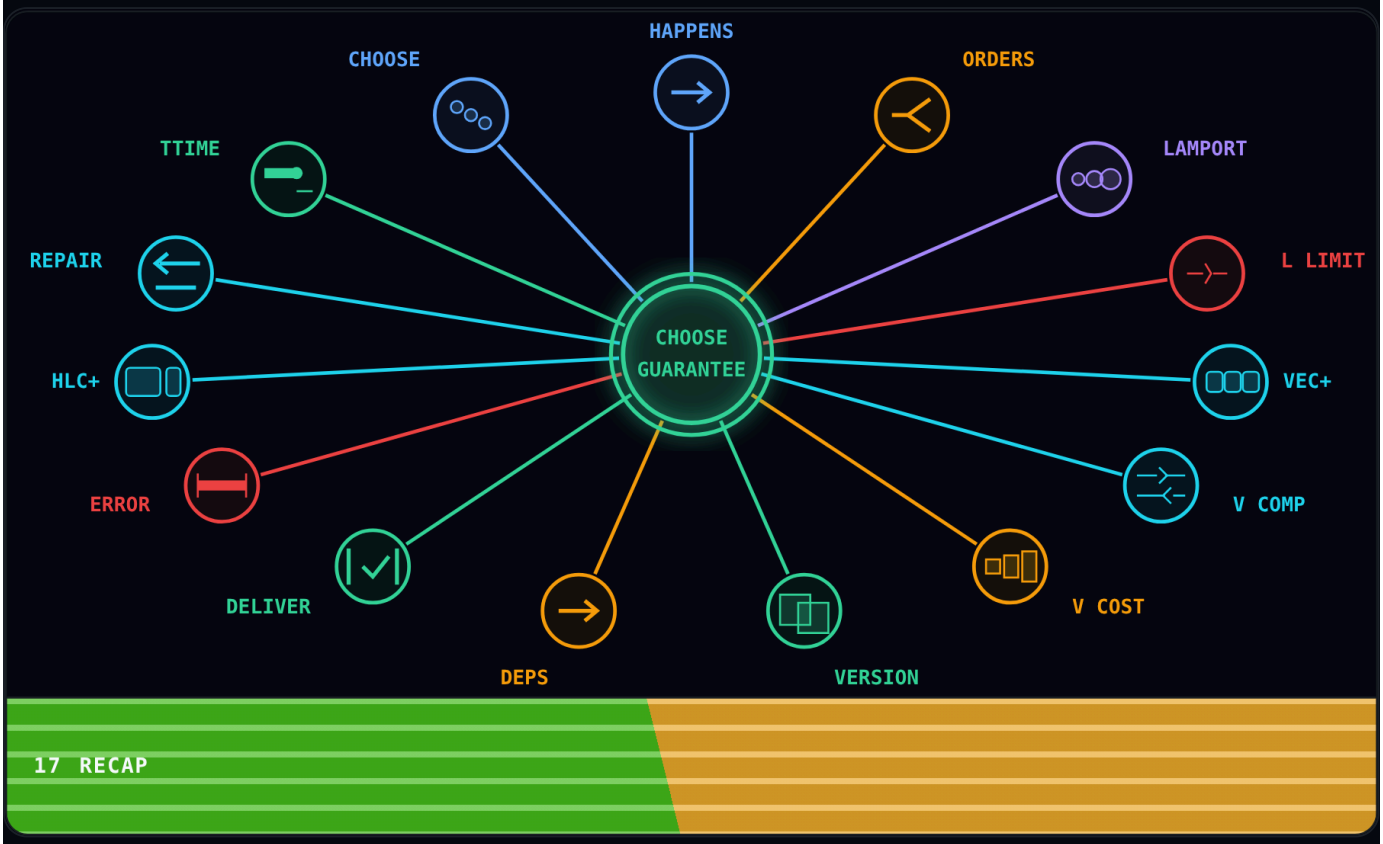
Causal order protects dependencies, total order supports deterministic replay and replicated decisions and external order follows real-time completion.

Should every system pay for the strongest guarantee even when the product never exposes it?

No, because changing Guarantee lets the map select the least costly clock evidence that can still prove the chosen product promise.

Ordering is a design choice rather than one universal clock and now let us reconnect all fifteen lessons into one toolbox.

Recap



We started with happens before and kept only relationships proven by local order or messages.

Then partial order preserved concurrency while total order inserted deterministic tie breaks.

Lamport rules compressed causal direction into one rising scalar.

Its blind spot showed that unequal scalars do not prove a causal path.

Vector construction restored separate counters and merged known history component by component.

Vector comparison used opposing components to identify concurrent events.

The cost stage made one component per participant visible as growing metadata.

Version vectors applied the comparison to object versions and preserved concurrent siblings.

Causal dependencies recorded the history an effect had already observed.

Causal delivery buffered the effect until the visible frontier contained its cause.

Clock uncertainty turned one suspicious physical point into an honest interval.

HLC construction paired near-wall time with a logical counter.

HLC repair kept causal timestamps rising through a backward clock jump.

A TrueTime style API exposed bounds and waited until external order was safe.

The decision map chose causal, total, or external order from the product promise.

The unified lesson is simple: preserve causes, admit concurrency, expose uncertainty, and pay for stronger order only when users need it.