Physical and Logical Time

Content

Time properties

- *Time needs memory (since it is countable change).* There was a definition of time that was: time is collection of countable events.
- *Time is local (rate of change slow with acceleration).* There was an experiment with two atomic clocks. One clock was moved in a higher altitude and it was possible to check that they got a little difference of time.
- *Time synchronization is harder at distance.*
- *Time is a bad proxy for causality.*

Clock Drift

Refers to the drift between measured time and a reference time for the same measurement unit. It's basically the acceleration between the clock of reference.

External vs internal synchronization

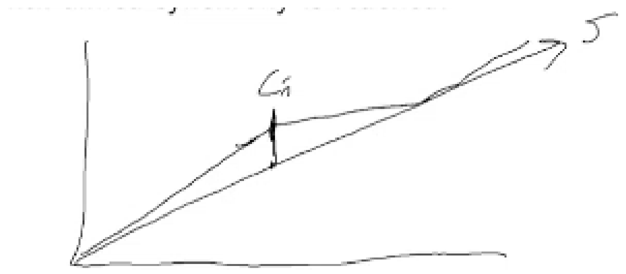
- **External synchronization** states the **precision** with respect an **authority -** it's like synchronizing the clock if the clock of the church. For a band D > 0 and UTC source S we have $|S(t) - C_i(t)| < D$.
- **Internal synchronization** states a **precision** between nodes. For a band D > 0 we have $|C_j(t) - C_i(t)| < D$.

An externally synchronized system at band D is necessarily also at 2D internal synchronization. Imagine that we have two people synchronizing their clocks with a church and they cannot have more than 1 minute of difference (less or more). We have that D = 1. Thus the difference between these two people can be at maximum 2 minutes: one is late 1 minute and the other is advanced 1 minute.

Monotonicity

One example is `make`. We have that $(t' > t \rightarrow C(t') > C(t))$. It's necessary to **reduce the time rate** until the aimed synchrony is reached.

> Our computer has two different types of clock: the **monotonic** and the **wall-clock.** The wall-clock is the one that tells the hour of the day. This clock is not really precise that time to time it tries to synchronize with NTP. This clock can be advanced or late. The **monotonic** clock by another hand just advances, it just computes the time that has passed. And the strategy to synchronize this kind of clock is to decrease the time rate until it reaches the synchronization. This synchronization is also made with NTP.



Adjusting the monotonic clock.

Synchronization

Synchronous system

- To we synchronize a node with another, we can simply send a package (from A) containing the current time (to B). In the end the time of B will be $t + t_{trans}$.
- However, the t_{trans} can vary between $[t_{min}, t_{max}]$. Thus, the final time will be between $[t + t_{min}, t + t_{max}]$. The uncertainty is $u = t_{max} - t_{min}$. So, what should be the time to adjust the clock? We can set that $B_{clock} = t + \frac{(t_{max} + t_{min})}{2}$. Because on this way the uncertainty will be $\frac{u}{2}$.

Asynchronous system

Te problem with this kind of system is that $t_{max} = +\inf$. How can we update clocks?

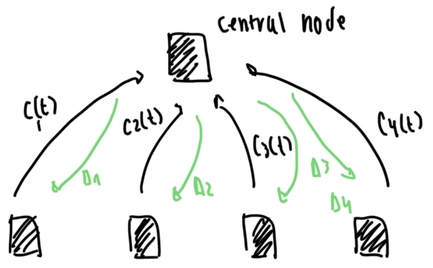
Cristian's algorithm

We can use the **Cristian's algorithm.** It's simply the node that has requested the synchronization calculates the RTT (round-trip-time) and set the clock as $t + t_r/2$. The network may be congestioned by the time of the send of the request, for this reason, many requests can be made and in the end the mean of the t_r is used.

Berkeley Algorithm

How can we synchronize many clocks?

- Every node sends the actual time in each of them. The central node will calculate the difference between him self and each node. There’re many ways to decide what should be the current time (e.g mean, max).



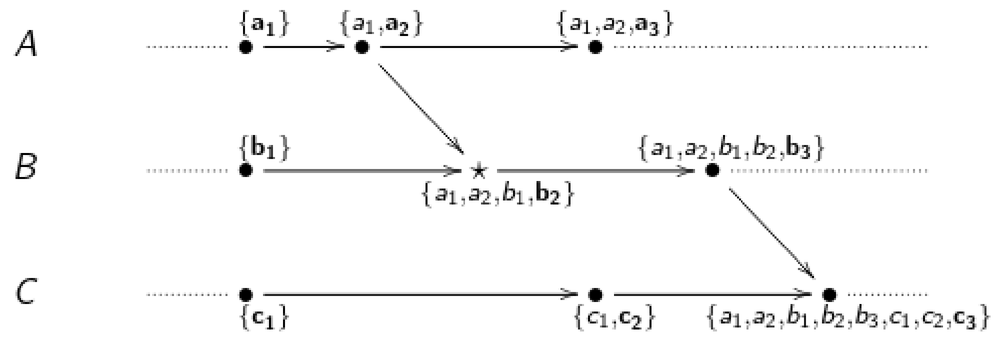
- In the end, the node that decides the time, will answer to each of the nodes, what is the amount of time that should be advanced or delayed in each one of them. This is the Δ time.

Causality

Causal histories

https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3E81EE1A30BAD99A180524218C5F0AFD?doi=10.1.1.9013&rep=rep1&type=pdf

- If the event is in the past, it means that we know the history.
- If we don’t know other’s history, this means that they happened in parallel.



$\star$ receive $\{a_1, a_2\}$ at node b with $\{b_1\}$ yields $\{b_1\} \cup \{a_1, a_2\} \cup \{b_2\}$

We have that:

$$\text{Check } \diamond \to \star \text{ iff } \{a_1, \mathbf{a_2}\} \subset \{a_1, a_2, b_1, \mathbf{b_2}\}$$

An optimization is:

$$\text{Check } \diamond \to \star \text{ iff } \mathbf{a_2} \in \{a_1, a_2, b_1, \mathbf{b_2}\}$$

In general terms:

$$\text{Note: } \{e_n\} \subset C_x \Rightarrow \{e_1 \ldots e_n\} \subset C_x$$

Due, to this optimization a lot of redundancy can be compressed.

Vector clocks

https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

Imagine that the causal history of a node is:

$$\{a_1, b_2, b_1, b_2, b_3, c_1, c_2, c_3\}$$

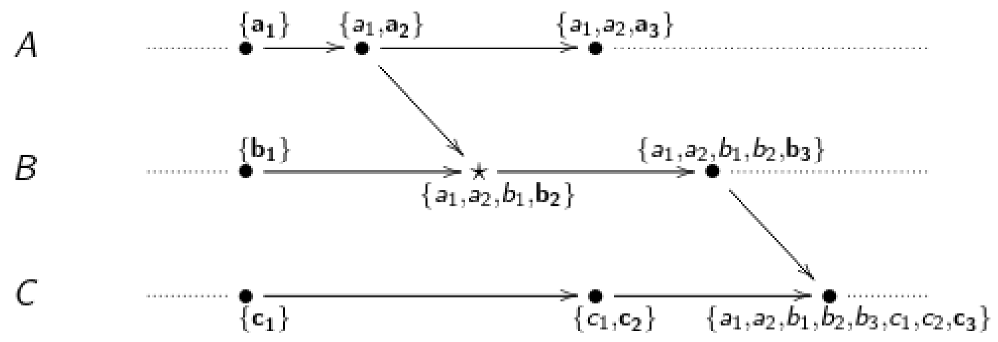
From the optimization explained in the previous section we can create:

$$\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$$

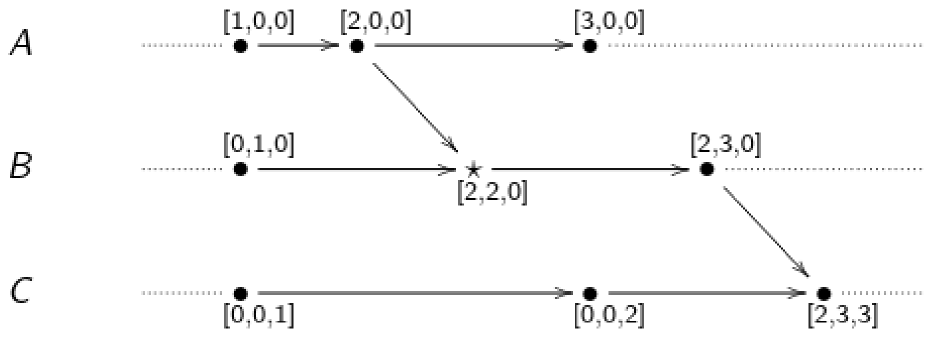
In the end, everything can be represented as a vector clock:

$$[2, 3, 3]$$

Now, the previous causal history can be represented as:

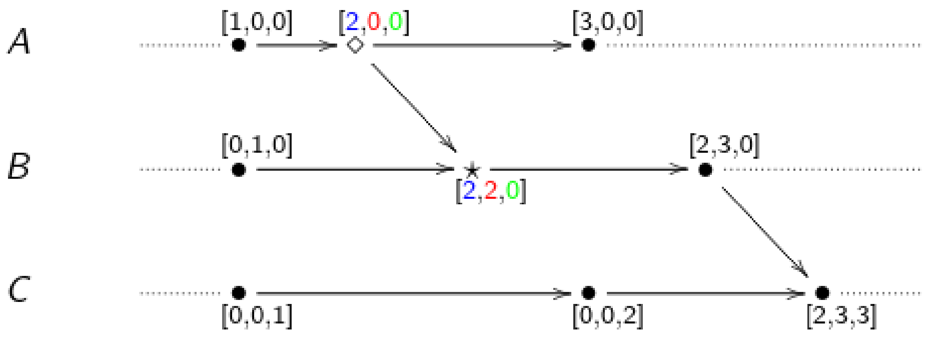


$\star$ receive $\{a_1, a_2\}$ at node b with $\{b_1\}$ yields $\{b_1\} \cup \{a_1, a_2\} \cup \{b_2\}$



$\star$ receive $[2,0,0]$ at b with $[0,1,0]$ yields $\mathrm{inc}_b(\sqcup([2,0,0],[0,1,0]))$

Then, how can we verify the causality?

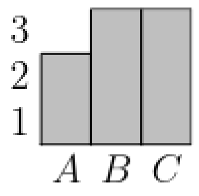


Check $\diamond \to \star$ iff point-wise check $2 \leq 2, 0 \leq 2, 0 \leq 0$

Graphical vision

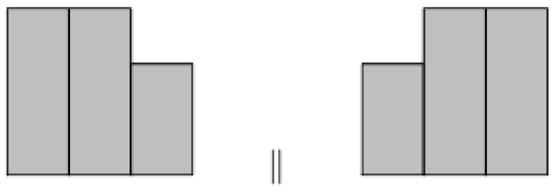
I prefer the arithmetic operations to check the causality, but actually there's the graphical relation:

$$[2, 3, 3] \equiv \{A \mapsto 2, B \mapsto 3, C \mapsto 3\} \equiv$$



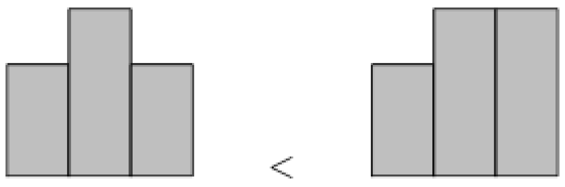
When one of the sides knows that something the other doesn't know, we have two **parallel events**:

$$[3, 3, 2] \parallel [2, 3, 3]$$



Yet, it's possible to see when an **event precedes other**:

$$[2, 3, 2] < [2, 3, 3]$$

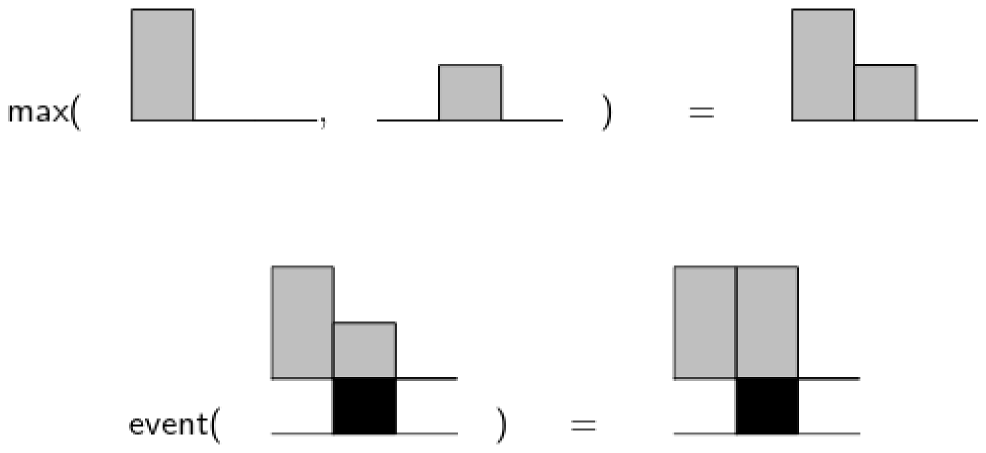


How to join two vectors?

If an event happened in node B (current vector clock was [0,1,0]) and received the event from A [2,0,0], the final clock will be:

$$\max([0,1,0],[2,0,0]) + [0,1,0] = [2,2,0]$$

Graphically speaking we should have something like this:



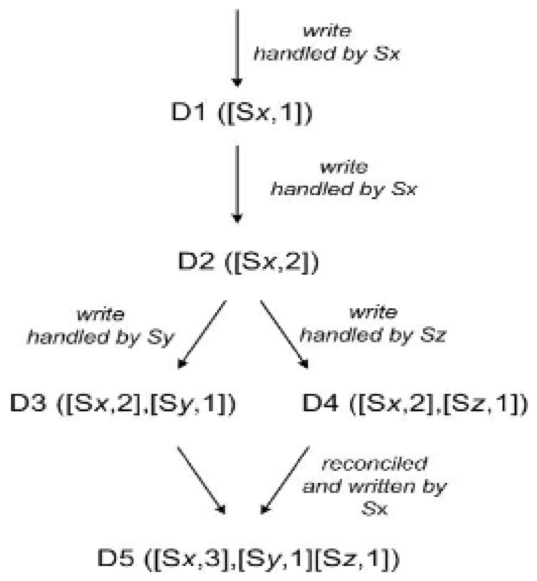
It’s also possible to specify who is the *owner* of the vector clock by representing the last event:

$$[2,0,0] \rightarrow [1,0,0]_{a2}$$
$$[2,2,0] \rightarrow [2,1,0]_{b2}$$

Dynamo

https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Amazon wanted the client to be able to put things in the Amazon’s cart and if the client desires to finish the purchase in a cellphone, for example, not necessarily the person will be accessing the same server. The original server and the current server needed to have some communication so that this effect could be achieved. But at some point it would be possible to the system have 2 copies of the same cart.

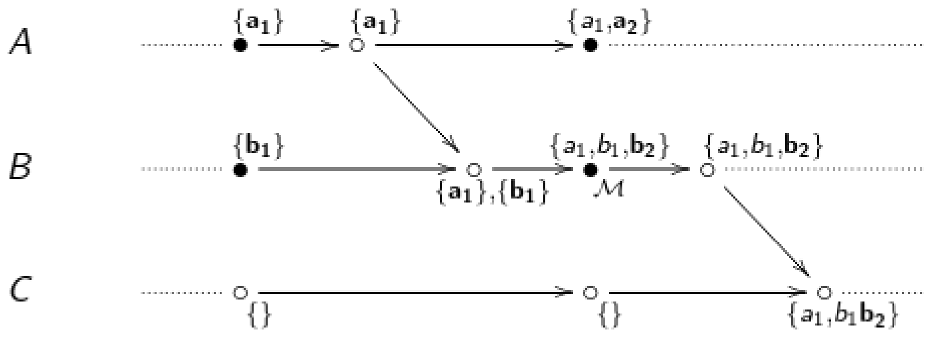


From Amazon’s article

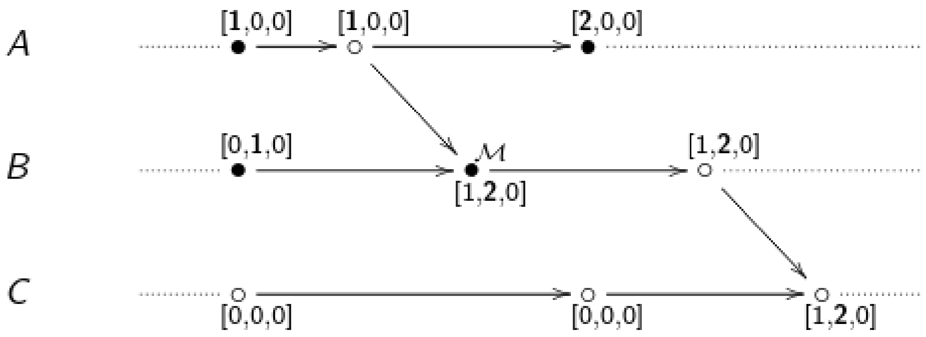
In the end the two copies needed to be merged in just one.

Version vectors vs vector clocks

It’s not always possible to merge automatically.



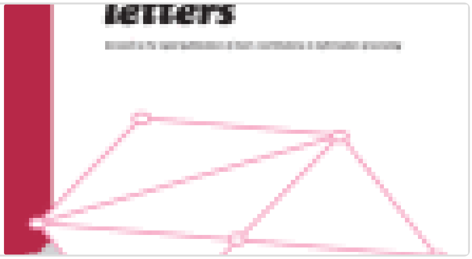
The difference between **version clocks** and **vector clocks** is that in the **version clocks** we might actually have two equal states.



Can causality scale?

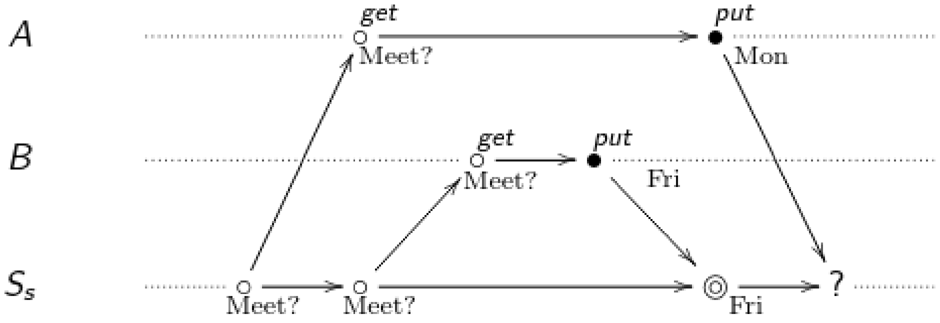
Concerning the size of logical clocks in distributed systems

https://www.sciencedirect.com/science/article/abs/pii/002001909190055M



One entry is needed per source of concurrency.

Imagine the following scenario:

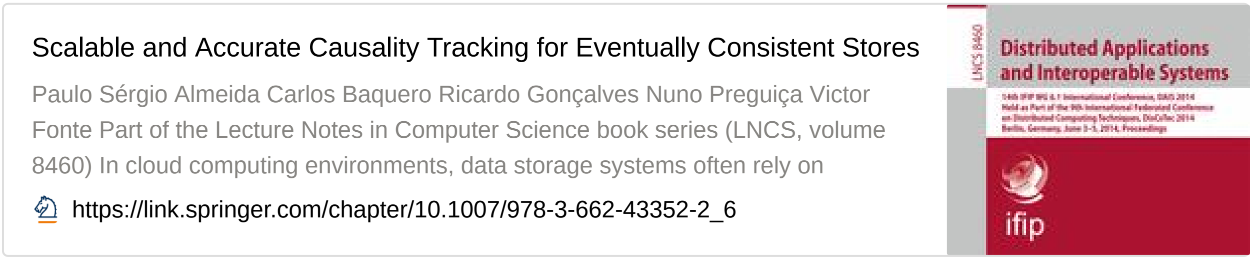


When **B** marks `Friday` , the system `Ss` set the choice as `Friday`. Later the **A** marks `Monday` and by this time we gonna have a conflict. The solutions are:

- Overwrite **B**s entry → Used in **cassandra**
- Conditional write (reject **A**s input)

- Use Multi-Value ([1,0] || [0,1]). One entry per client. → This is option chosen by dynamo.

Dotted version vectors (Scaling at the edge DVVs)

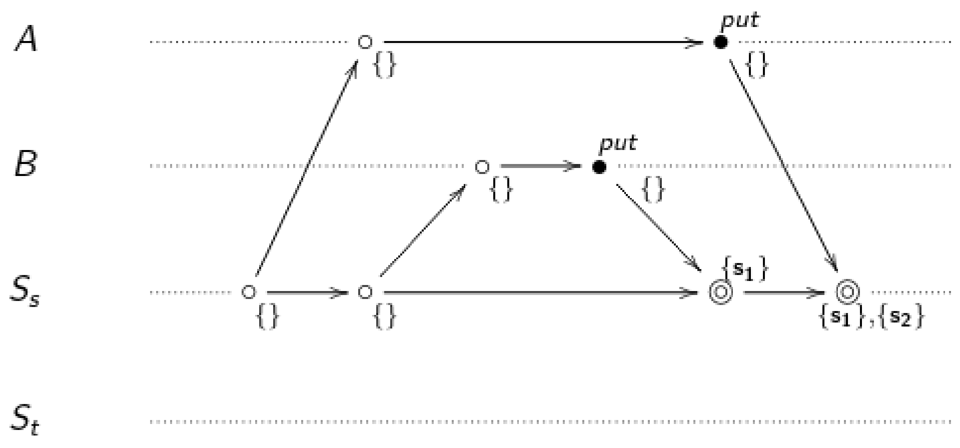


Scalable and Accurate Causality Tracking for Eventually Consistent Stores

Paulo Sérgio Almeida Carlos Baquero Ricardo Gonçalves Nuno Preguiça Victor Fonte Part of the Lecture Notes in Computer Science book series (LNCS, volume 8460) In cloud computing environments, data storage systems often rely on

https://link.springer.com/chapter/10.1007/978-3-662-43352-2_6

In the previous situation we gonna have a parallelism:



Where:

$$[0,0]_{s1}||[0,0]_{s2}$$

In the context of the **Amazon store cart** the Amazon could show the two types of cards to the user and let him decide what to do.

Advantages:

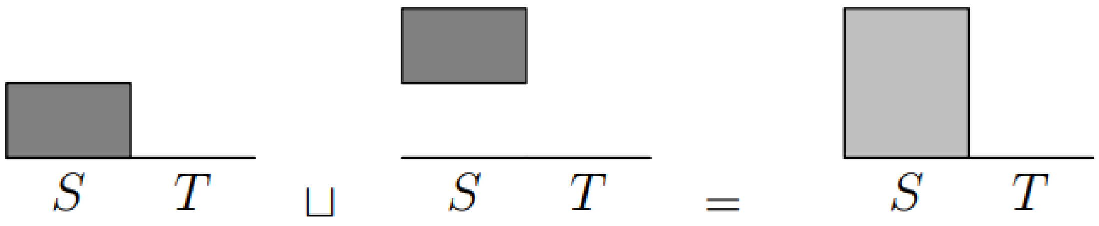
- Scales at the edge
- Bounds to the number of replicas/proxies instead of clients (like dynamo did) achieving more compact representation in comparison with dynamo, which did not

scale

- does not lose causality (nor overwrite, nor conditional write)

Representing causal past

$$[0,0]s_1 \sqcup [0,0]s_2 = [2,0]$$



$$[0,2]t_3 \parallel [2,0]t_4$$