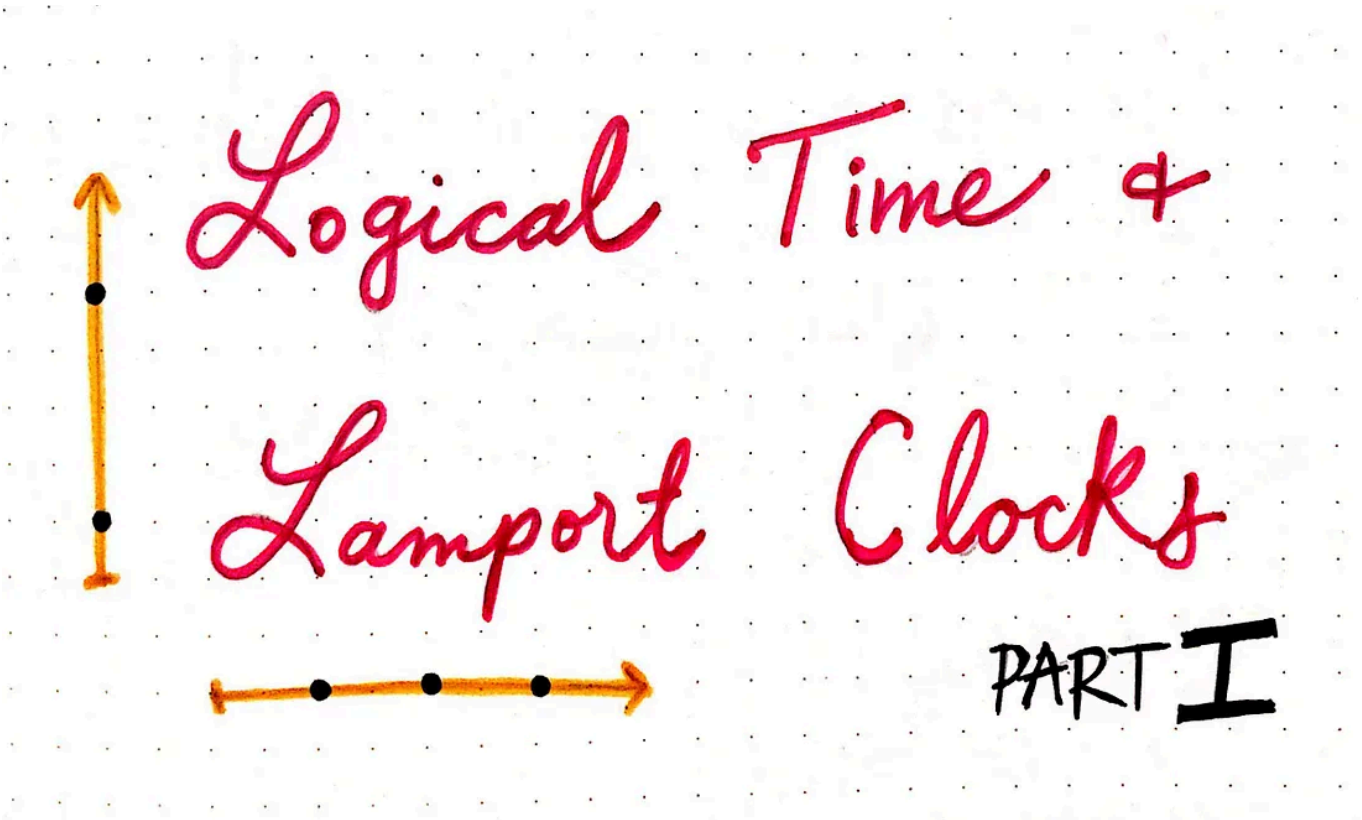
Logical Time and Lamport Clocks (Part 1)

Vaidehi Joshi · Follow

Published in baseds · 9 min read · Nov 15, 2019



Logical time and Lamport clocks (part 1)

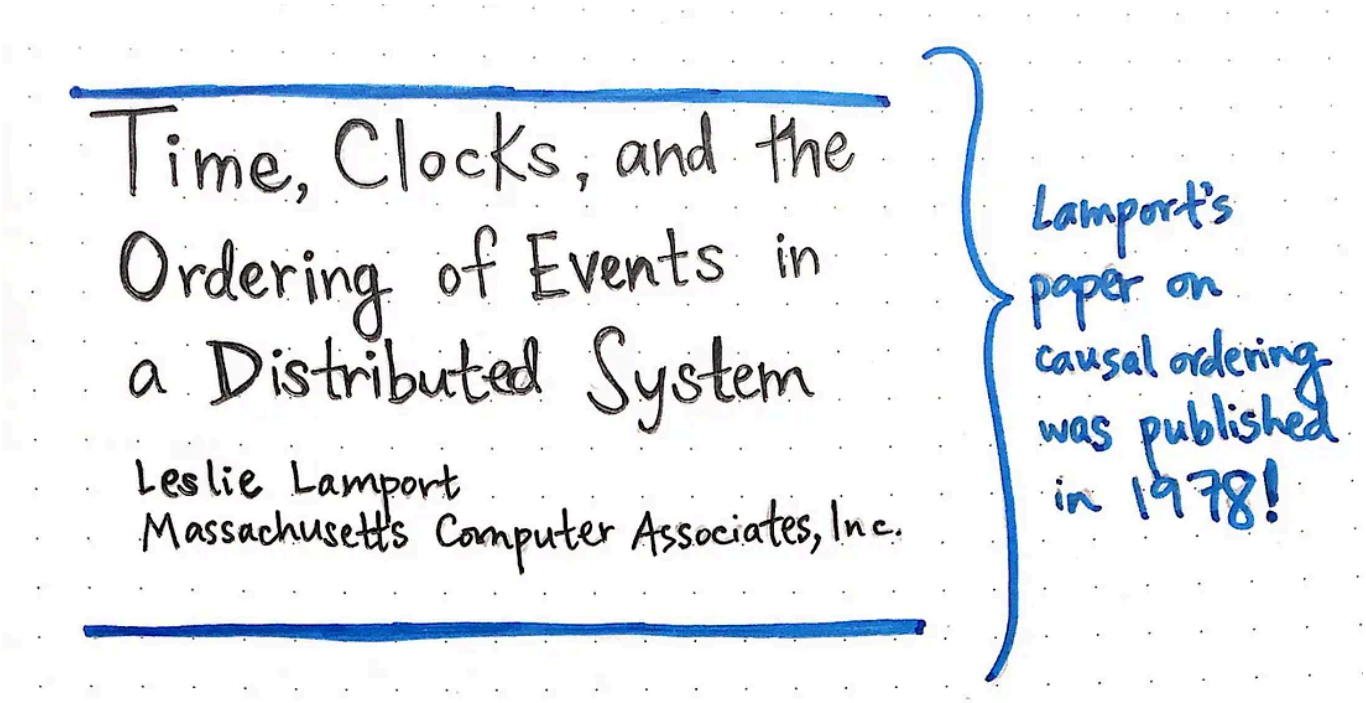
Over the course of this series, we've seen many instances of how things can more complicated than they seem. We saw this with failure, and we saw it with replication. More recently, we discovered that even the concept of time is more complex that we might have originally thought.

However, when the things that you thought you knew seem more convoluted than ever, sometimes the answer is to keep it simple. In other words, we can keep a problem simple by stripping out the confusing parts and trimming it down to its most essential parts. In fact, this approach is exactly what one computer scientist did in the late 70's when he was, just like us, trying to figure out how to make sense of time in a distributed system.

The previous two posts on clocks and the ordering of events in a distributed system have both been building up to the topic we're finally ready to uncover: logical time and Lamport clocks! There's a lot to cover, so we'll spread it out between two posts. Let's dive right into part one!

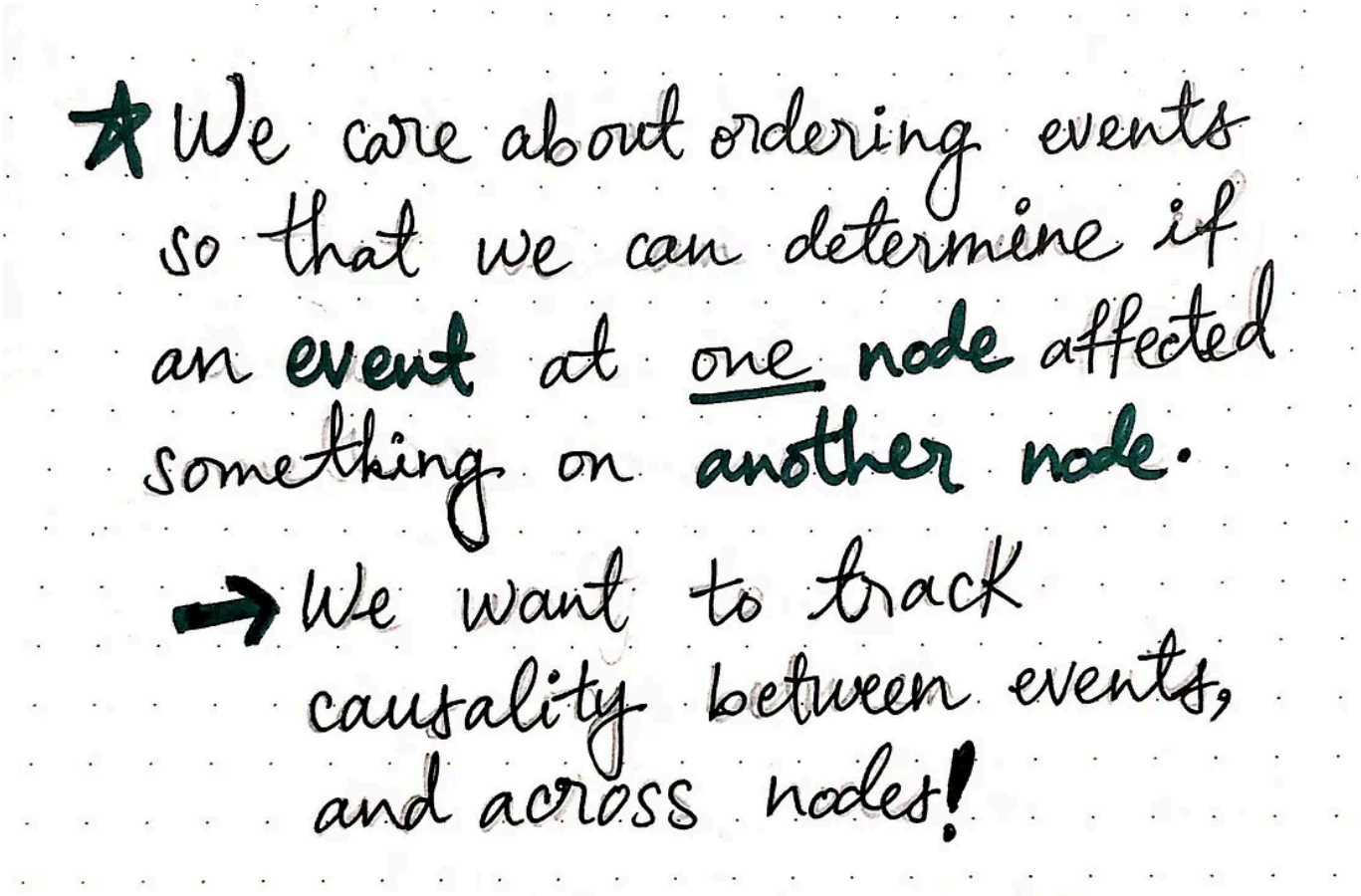
Causality and "happens before"

The story of logical clocks starts in 1978, when a paper was published in the *Communications of ACM* journal that would go down in history. Leslie Lamport, a computer scientist at Massachusetts Computer Associates, wrote about his research around ordering events in a distributed system. This paper, entitled "Time, Clocks, and the Ordering of Events in a Distributed System", would go on to be one of the most-cited papers in computer science, and would also win the Principles of Distributed Computing Influential Paper Award more than 20 years later.



Lamport's paper on causal ordering was published in 1978!

In the paper, Lamport outlines how we think about time as humans, and why we need to shift our paradigm when it comes to distributed systems, and the idea of partial ordering. He explains that, "In a distributed system, it is sometimes impossible to say that one of two events occurred first".

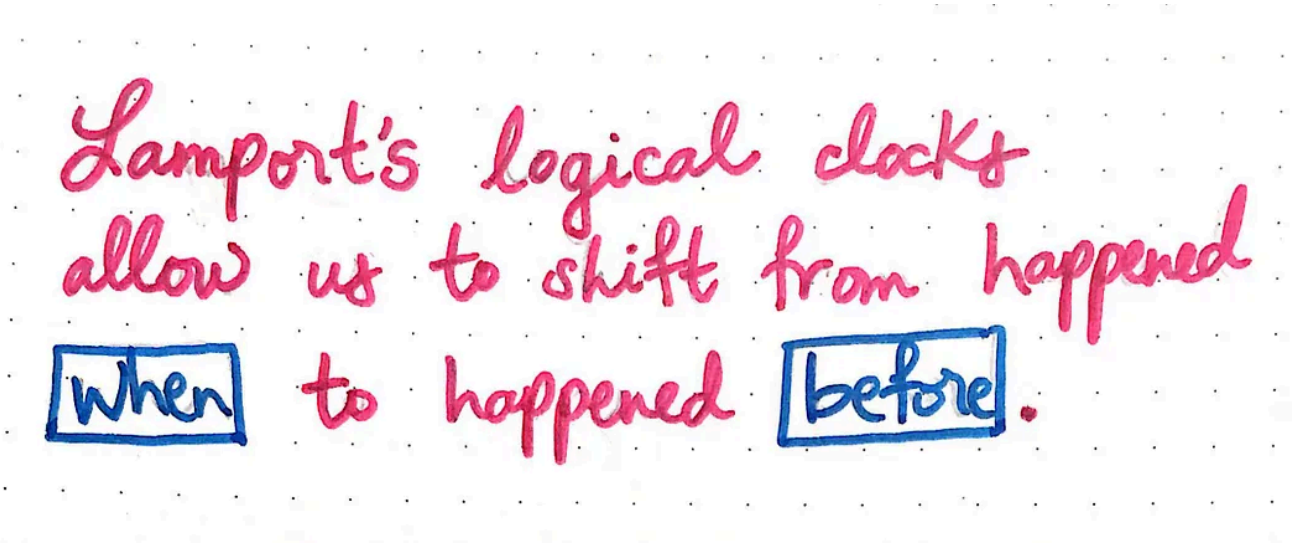


Why do we care about ordering events? So that we can track causality!

As Lamport cites in his paper, the reason that we care about time is so that we can figure out the order in which things happen. While this is certainly harder (or sometimes impossible!) to do in a distributed system, our reason for wanting to *know* the order of some events in a system all stems back to the same desire: we care about ordering events so we can determine how those events are connected. When dealing with events in *any* system, the reason we actually want to order them is so that we can see the chain of events within the system. Within a distributed system in particular, we this means that we are often trying to determine if an event at one node affects or causes an event at another node.

But, as we've seen in our own study of distributed systems, this task is no easy feat. When a system is distributed, there is no global clock, which means that we cannot depend on any central time source. This also means that the events in our system are not totally ordered, which means we can't be sure of exactly *when* every event in the system took place. Lamport's paper acknowledges all of these constrains and empathizes with how tricky this problem really is!

Lamport's solution is to shift our thinking. He presents a novel idea: we don't actually need to think about causality in the context of total ordering to start. Instead, he says that we can start with a partial ordering of events, and then just deal with figuring out which events happened before other events. Once we figure out a partial ordering, we can turn it into a consistent total ordering.



Lamport's logical clocks allow us to shift from happened "when" to happened "before".

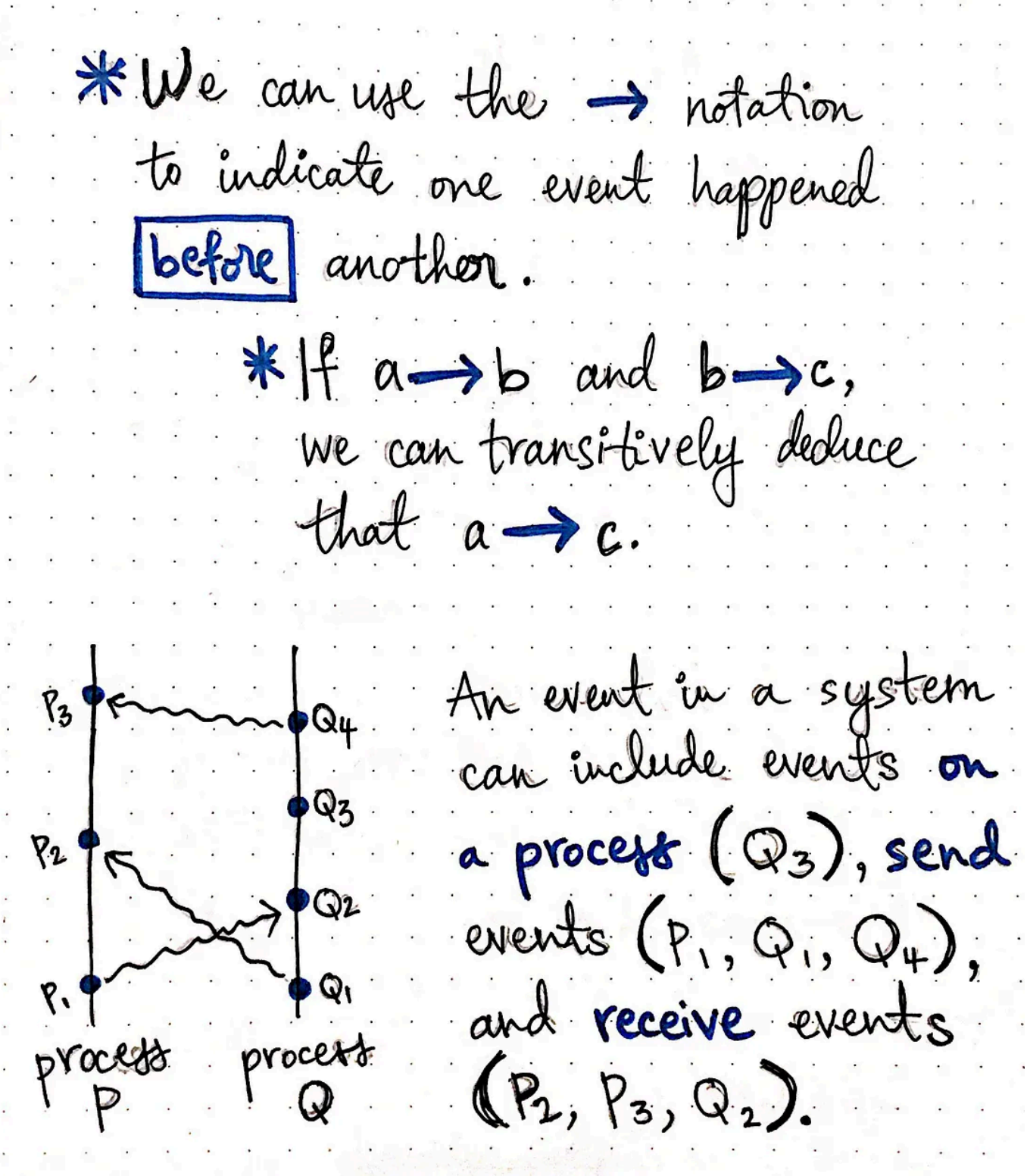
So how do we do this? Well, to start, we need to shift from thinking about *when* an event happened to what the event happened *before*.

Shifting from "when" to "before"

The idea of one event happening before another is central to Lamport's paper. He uses the → shorthand notation to indicate the ***happens before*** relationship, or the fact that one event happened before another. For

example, if we know that one event, a, happened before another event, b, then we can say that $a \rightarrow b$, or a happened before b.

The happens before relationship can also be applied transitively. In other words, we can create a chain of events where one event happened before another. If we say that $a \rightarrow b$ and $b \rightarrow c$, then by using the transitive property, we can say that $a \rightarrow c$, or a happened before c. As we might be able to imagine, we could very easily string together a chain of events, where one event happens before another, which happens before another, and so on and so forth.



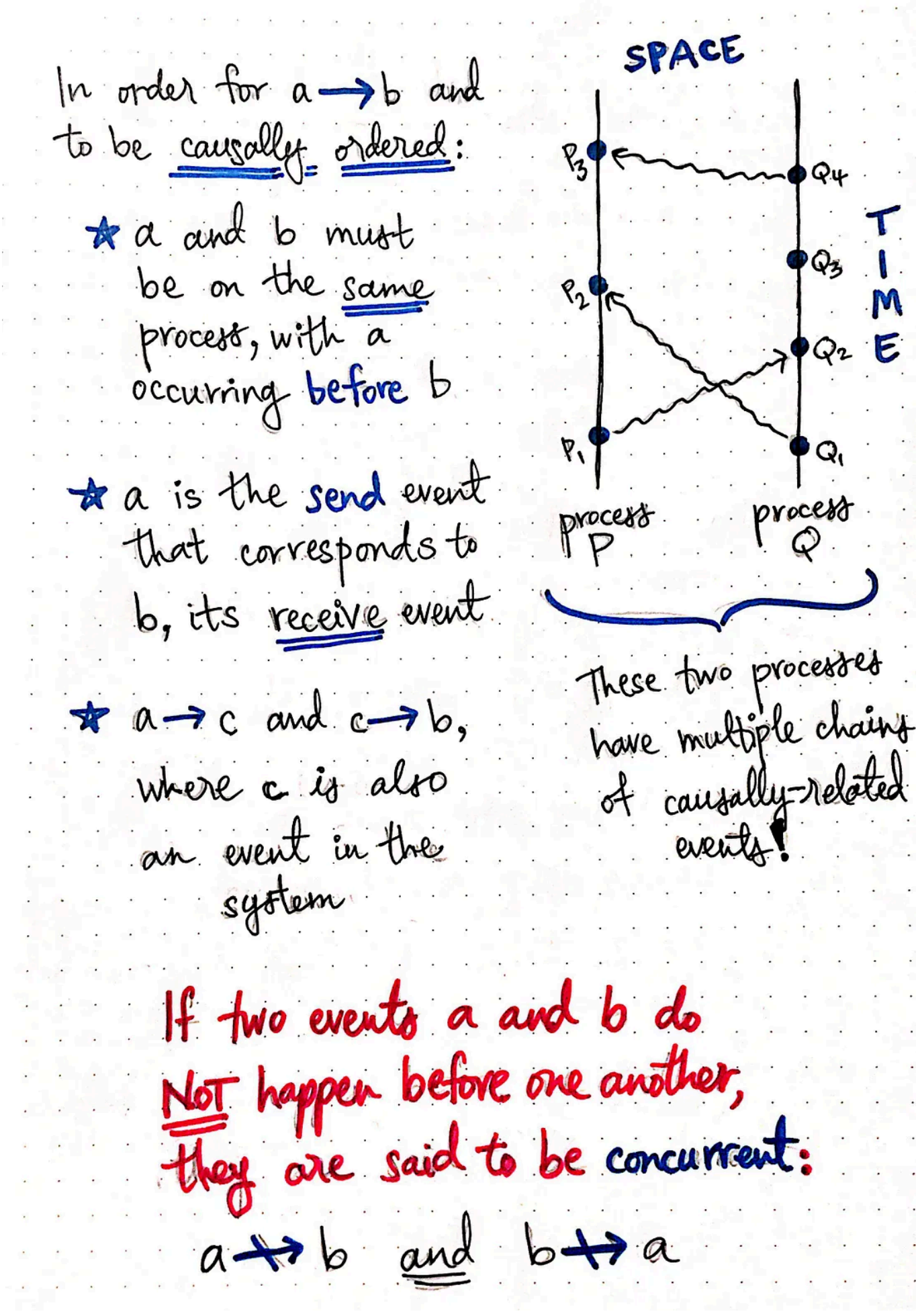
Understanding the “happened before” notation.

But wait a second — we keep talking about different events in the system, but we haven’t really clarified what an event could possibly be! As it turns out, an event in a distributed system can take different forms. As we know, there can be many different nodes in a distributed system. Each node has its own local system clock, and it is capable of processing its own tasks. However, the nodes can also communicate between one another, sending messages back and forth.

An ***event*** encompasses all of the different things that can happen within and between nodes in a system. An event could be something that occurs on a single process or node. An event also includes any ***send events***, where a node sends a message to another node or process. Conversely, we must also consider ***receive events***, when a node receives an incoming message from elsewhere in the system.

In the example above, we can see examples of all three of these events. We have two processes, `P` and `Q`. There is one event, `Q3`, which occurs on process `Q` that are not related to sending or receiving any messages. This is our basic event that indicates that something occurred on the node for process `Q`. However, we also have a few send events: `P1`, `Q1`, `Q4`. These are all events that indicate that we are sending messages out from a node to somewhere else in the system. On the other hand, `P2`, `P3`, and `Q2` are each receive events, which indicate that we have received some message from another node in the system.

Now that we understand what an event in a system could be, we can turn back to the happens before relationship. When we say that an event $a \rightarrow b$, we are asserting that event *a* occurred before *b*, because *a* happened before *b*. We can say that these two events are ***causally ordered***, where the ordering of the events is contingent upon one event causing another to happen.



Causally-ordered and concurrent events across two processes.

There are a few rules to causal ordering that are important for us to understand. In order for $a \rightarrow b$ to be causally ordered, one of the following

three situations must be true:

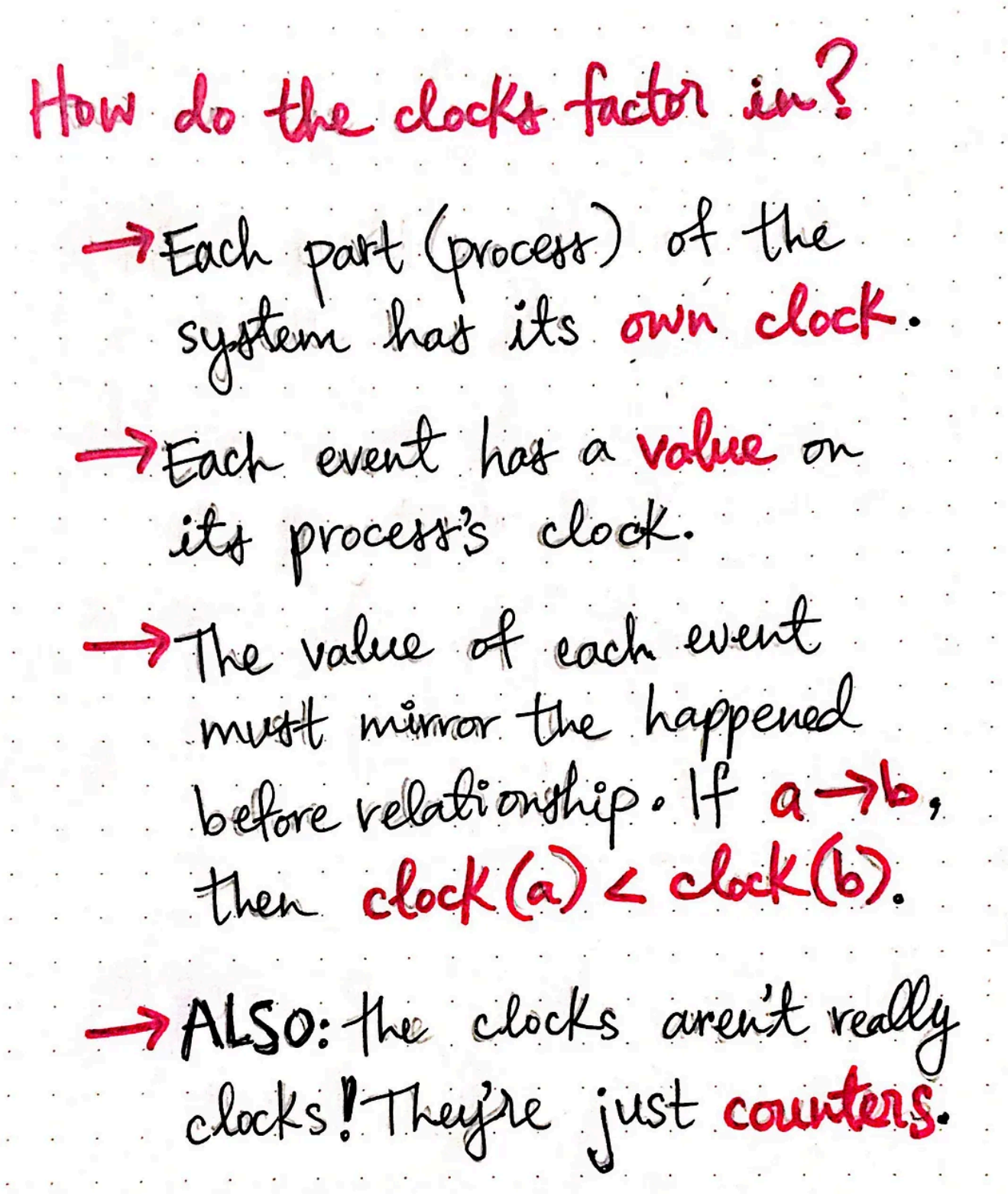
1. Events *a* and *b* must occur on the ***same process***, and *a* must occur before *b* occurs on the process.
2. The events can occur on ***different processes*** so long as *a* is the send event that corresponds to *b,* which must be its receive event.
3. The events are ***transitively linked*** with another event in the system, but *a* still happens before *b*. For example, if *a* happens before *c*, and *c* happens before *b*, then we know that *a* → *b*.

As messages travel through time and across space from one process to another and, we can start to construct chains of causally events (also called ***causal paths***) and see how different events across processes are connected to one another. For example, in processes `P` and `Q`, `Q1` → `P3` (through event `P2`), and `P1` → `Q4` (through events `Q2` and `Q3`).

Finally, it's worth mentioning that, if two events *a* and *b* do not happen before one another, than we can say that *a* ≠> *b* and *b* ≠> *a,* and that the two events are ***concurrent***. We will cover this in much more depth in part two of this post, but for now, we should just note that concurrent events do not have causal paths from one to another.

Logical clocks to the stage

In addition to the idea of "happens before", another core concept that Lamport introduces in his paper is the logical clock. As we already know, each node or process in a distributed system has its own concept of time, or its own local clock. However, Lamport's take on local clocks is different that what we've seen before.



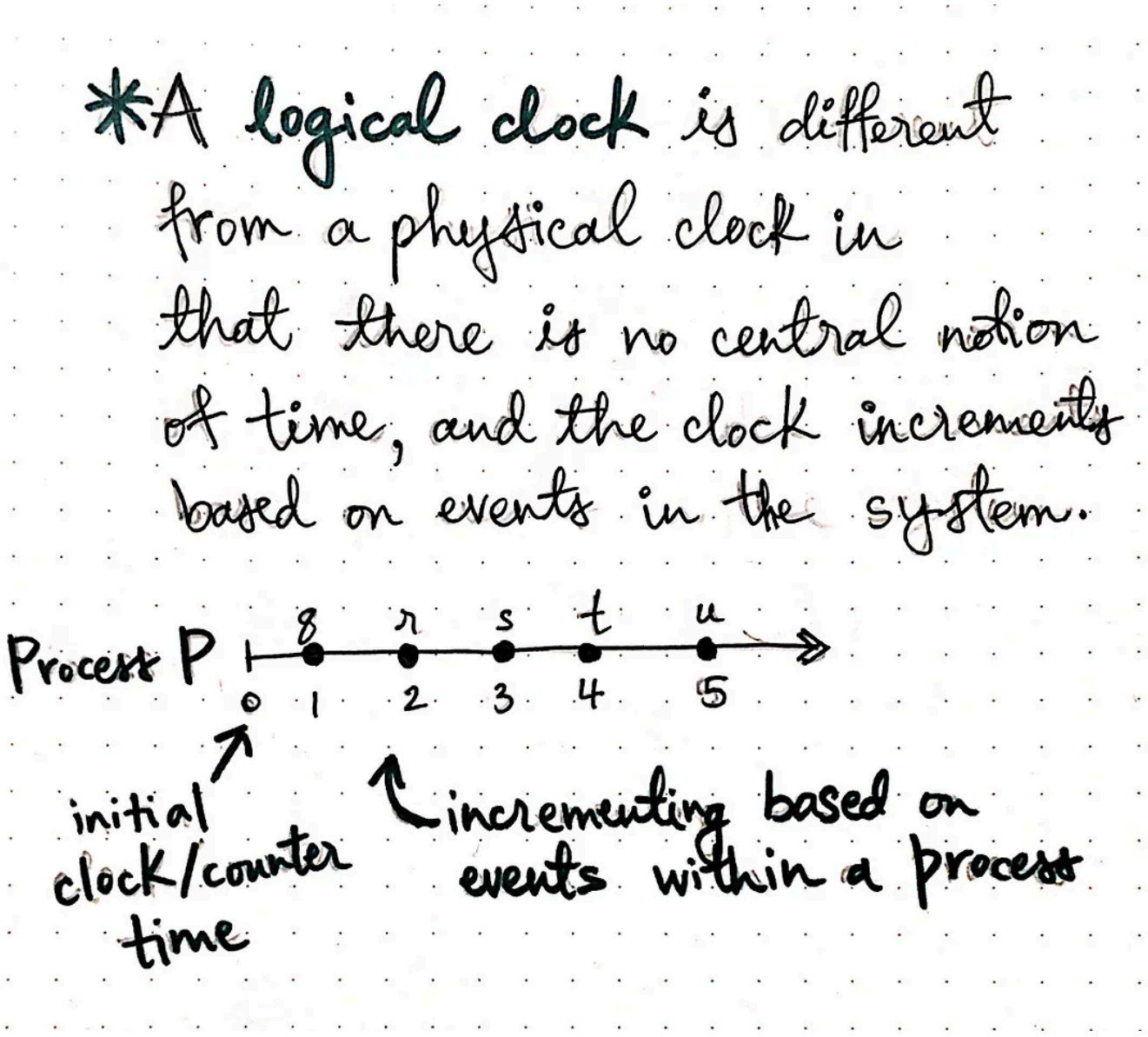
How do the clocks factor in?

Lamport suggests using something different from the typical physical clock that we all think of. Instead of using each process's physical clock to track the order of events, we can instead use a *counter*. The counter can start with an initial time (like 0), and we can treat that counter as the processes own local clock.

Lamport continues with this idea by proposing that, not only will every process within a distributed system have its own counter clock, but each

event that is recorded on a process should also have a ***value*** on that process's local clock. Furthermore, the value of each of these events on the clock must mirror any happened before relationships. For example, if event $a \rightarrow b$, then the clock time for when event a occurred must be less than the clock time for whenever event b occurred; in other words, `clock(a) < clock(b)`.

By using basic counters instead of physical clocks, Lamport simplifies clocks into something a little easier to deal with. These counter clocks are called logical clocks. A ***logical clock*** is quite different from a physical clock in that there is no central notion of time, and the clock is just a counter that increments based on events in the system.



Logical clocks: a definition.

Each process in a distributed system can use a logical clock to causally order all the events that are relevant to it. As events occur in a process — whether they are send or receive events — the process's clock counter is incremented by an arbitrary amount. We'll learn more about how this works in practice in part two of this post. We'll also be introduced to Lamport's algorithm for incrementing counters, and how to obey causality across processes. There's so much interesting stuff to learn; thankfully we have more time and another post to cover it all!

Resources

Conveniently, Lamport's work on logical clocks and causal ordering is well-taught and written about. There are a lot of great resources that introduce these topics, with varying complexities. If you'd like to do some further reading, check out some of my favorite resources, which I've listed below!

1. Time, Clocks, and the Ordering of Events in a Distributed System, Leslie Lamport
2. Time, Clocks and Ordering of Events in a Dist. System, Dan Rubenstein
3. Time and Ordering: Lamport Timestamps, Indranil Gupta
4. Lamport's Logical Clocks, Michael Whittaker
5. Logical Clocks, Professor Paul Krzyzanowski

Programming | Distributed Systems | Code | Computer Science | Software Development