Partial Ordering of Events in Distributed Systems

Introduction :

- In distributed systems, since many nodes are communicating with each other simultaneously and updating a state of some global resource (for example a database), it becomes important for us to know the order in which these events occurred to debug how we ended up in some particular state.
- This report discusses why traditional methods to calculate time does not apply to distributed systems and discusses the approach published by Leslie Lamport in his paper, Time, Clocks, and the Ordering of Events in a Distributed System.

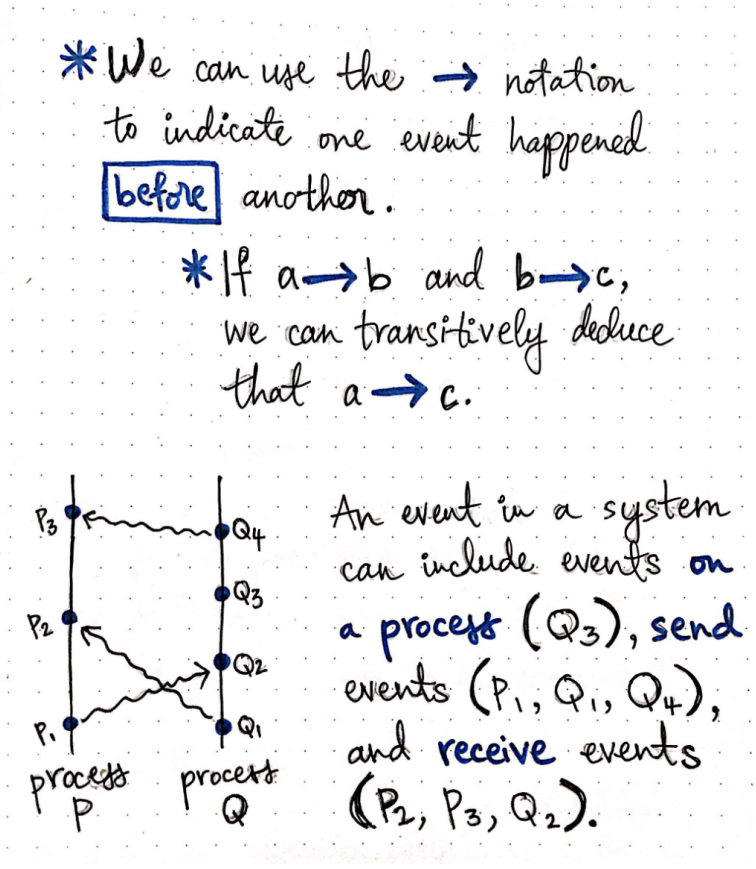
Motivation :

- The reason behind using another mechanism instead of using timestamps based on for ordering events in Distributed Systems
    - Usually, System clocks operate using Real Time Clock, or RTC for short.
    - As it turns out, deep inside the integrated circuit is a crystal, which vibrates or oscillates; it's known as a crystal oscillator.
    - As the crystal vibrates, the clock uses those vibrations to increase the counter and keep track of time.
    - Unfortunately, these clocks are inconsistent due to several imprecisions in the way the clock is constructed.
    - The preciseness of a clock could be affected by temperature, location, the clock's source of power, etc.
    - As a result, two clocks starting at the same time will drift apart gradually. This is known as **Clock Drift.**
    - Hence, we need to find another way to keep track of time in distributed systems.
- We could also try using some Global Time Service which tells our nodes in the system about time.
    - However, there are two limitations to that,
        i. A global time service constitutes a Single Point of Failure.
            - If we lose the time service, we won't be able to keep track of time.
        ii. Unreliable Communication link,

- If the communication link between our Node and the global time service breaks, we lose track of time.

Logical Time and Lamport Clocks :

- Leslie Lamport, a computer scientist at MCA, wrote about his research around ordering events in distributed systems in his paper, "Time, Clocks, and the Ordering of Events in a Distributed System"
- His paper highlights how we can deduce ordering of events using causality and sometimes how it is impossible in distributed systems to determine the order of events.
- 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 → b, or a happened before b.



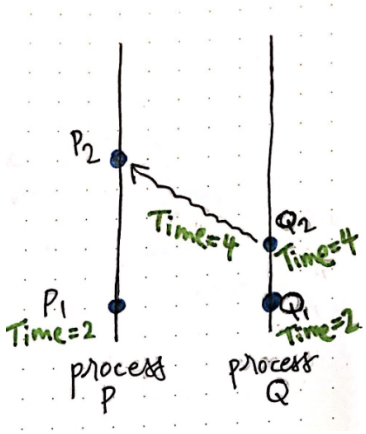
Understanding the "happened before" notation.

- Before moving on to Lamport's algorithm to determine order of events, let us first define what event is.
    - An event encompasses all of the different things that can happen within and between nodes in a system.
        i. It could be an internal process in the node.
        ii. A send event in case the node decides to send some message to another node, or
        iii. A receive event when a node receives some message from another node.
- Lamport suggested that instead of using timestamps of the system clocks, we can use a counter in the system.
- The counter can start with 0 and we can increment it by one if any of the above three events happen.
- Below is the Algorithm that he suggested

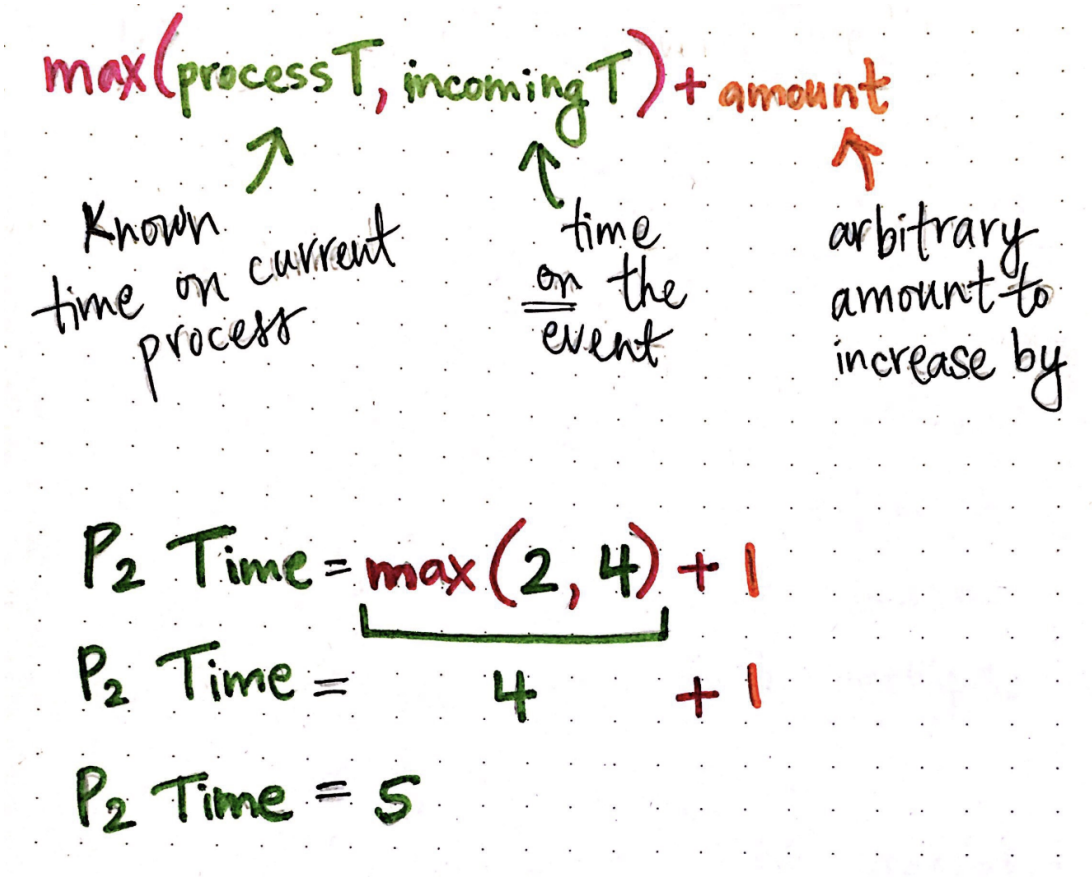
Lamport's Algorithm :

1. All the process counters start with value 0.
2. A process increments its counter for each event (internal event, message sending, message receiving) in that process.
3. When a process sends a message, it includes its (incremented) counter value with the message.
4. On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

Let us understand this by taking an example,

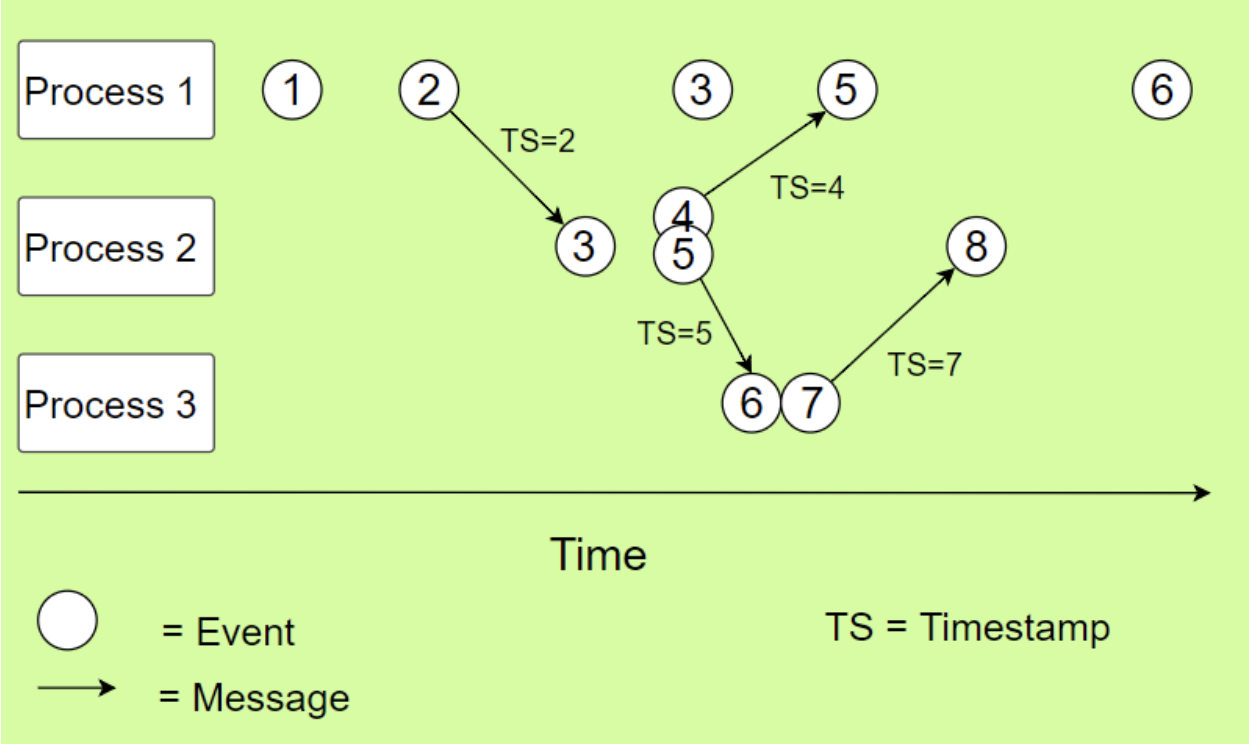


- Here, we have two processes, P and Q.
- Process P has an event on the process itself: P1. The time at P1 is 2.
- Process Q also has an event of its own, Q1, which has a time of 2. The next event on process Q is Q2, which is a send event and marks a message being sent from process Q to process P.
- When process Q sends a message to P, it also sends it's current time(counter) along with the message.
- process P receives that message, it marks it with a receive event, or P2.
- Then the logical clock at P2, compares it's current time with time it receives and takes the maximum of both and increments it by one.
- This can be summarized as follows
    - Time : max(processT, incomingT) + amount



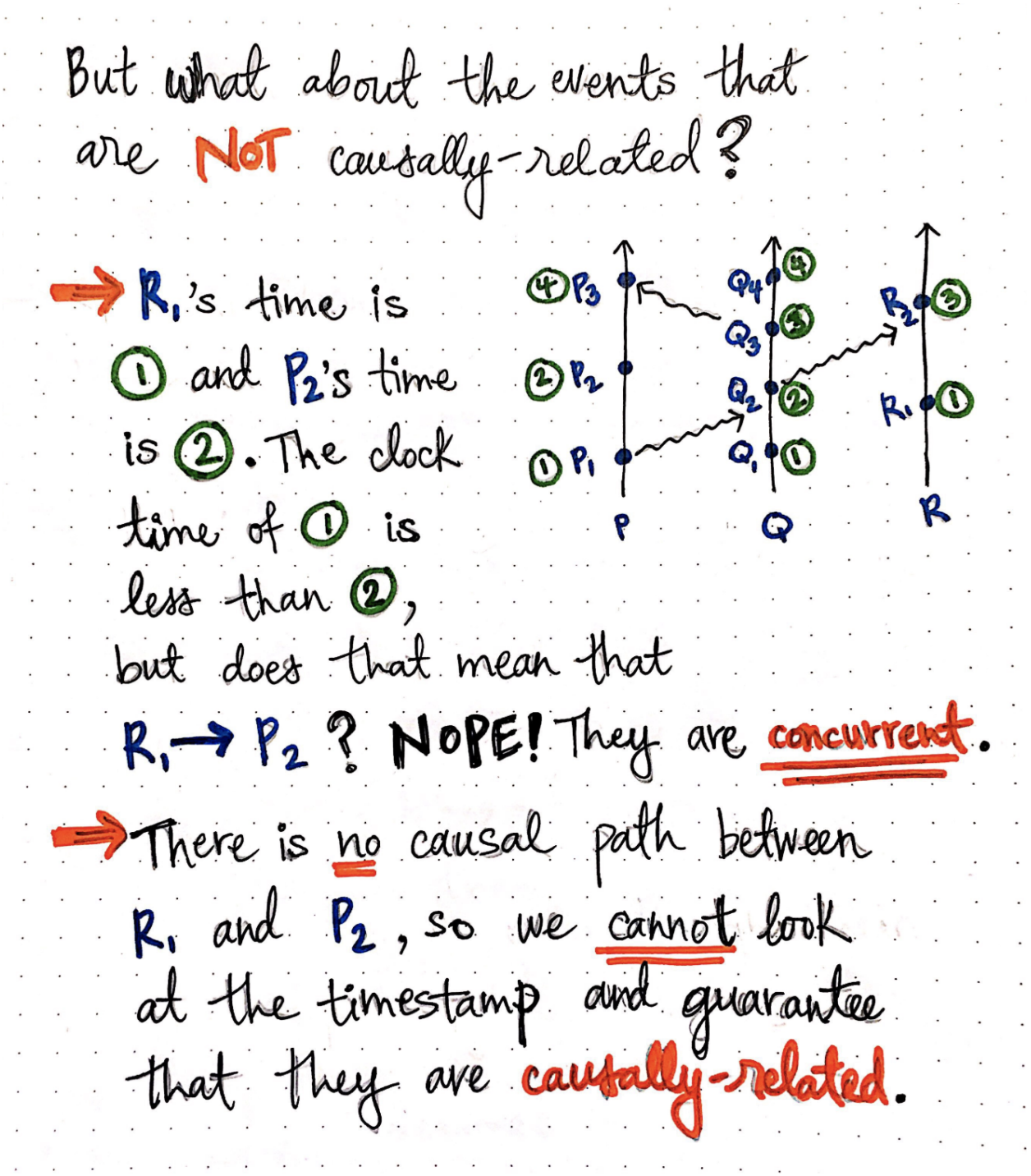
Lamport's algorithm for determining a timestamp.

Below is a more complex example for the same,



Limitations :

- For events that are not causally related, we cannot say that an event happens before a certain other event by comparing their timestamps.
- Let's look at an example below.



What happens when two events are not causally-related?

- Hence, in order to determine whether an event a happens before another event b, i.e a → b they should satisfy two conditions,
    a. They should be causally related,
    b. clock(a) < clock(b)

Conclusion :

- Hence we saw how we need to rethink how time works in distributed systems and how Lamport clocks helps us with partial ordering of events.
- Even though Lamport clocks are rarely used in Systems today, it is important to know that the patterns used in this paper serves as the fundamental design on which other clock synchronization mechanisms were built.

References :

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. Logical Time and Lamport Clocks (Part 1) - baseds, Medium blog
4. Logical Time and Lamport Clocks (Part 2) - baseds, Medium blog