Clock Synchronization

August 20th 2019

In distributed systems, memory is not shared, and so the state of the system may not be easily known by any computer in particular.

Ordering of events may not be globally known, so various strategies exist to ensure that the ordering of events can be determined by all nodes in the system.

This can be needed for situations like stream updates, or crash recovery, and others, by replaying the messages.

The local clocks for each computer isn't guaranteed to be the same as each other, and over time, can be pretty significant. There's 2 values to keep track of:

Clock skew: Relative difference between the value of two processes

Clock drift: Relative difference between clock frequencies of two processes.

The larger the clock drift, the faster the clock skew increases. The larger the clock skew, the more unsynchronized the system is.

To maintain the "true" time you'll either need:
1) an atomic clock
2) a GPS receiver, which gets time from atomic clocks (from satellites).

Option 1) is impractical, and option 2) has some limitations: (discussed later).

August 21st 2019

There are 2 types of synchronization strategies: external synchronization, and internal synchronization

External sync. is when processes attempt to reach a dedicated time server and synchronize with it.

Internal sync is when the processes attempt to just minimize the clock skew between themselves.

If a system is externally synchronized, then it's also internally synchronized. The reverse is not true though, as the system may drift away from the real external time over time.

Typical quartz clocks have a drift rate of about 31.5 seconds per year. However, processes may operate in millisecond slopes so it might be really important in some time-sensitive apps to be precisely synchronized (like airline flights)

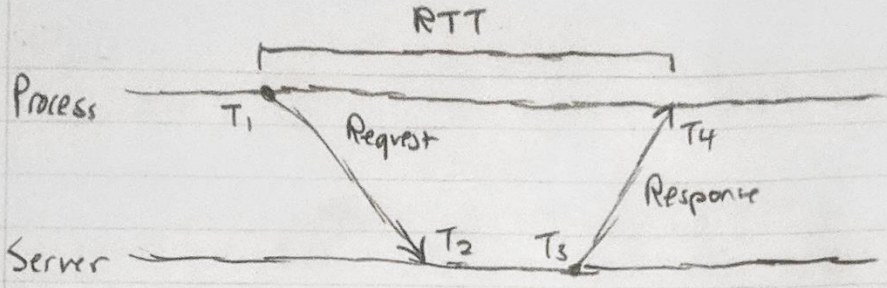
External Synchronization: Cristian's Algorithm

The problem is that contacting an external server itself takes time, so it's outdated by the time it gets back.

Inaccuracy is proportional to message latency, which is unbounded in an asynchronous system.

The idea behind Cristian's Algorithm is to measure the Round-Trip Time (RTT) and adjust the time accordingly.

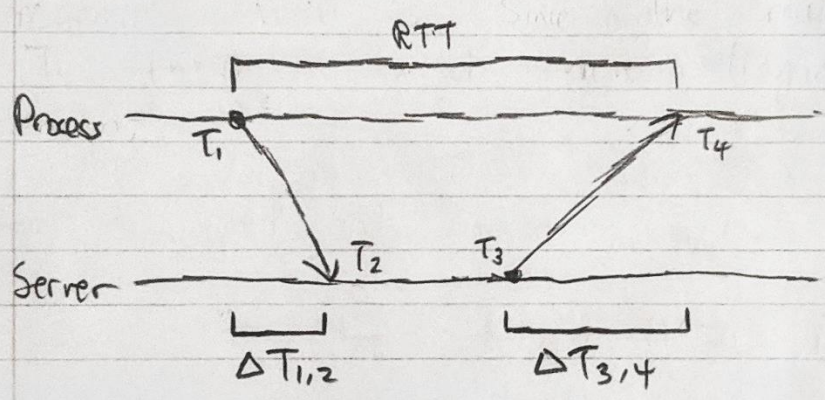
August 21st 2019



The latencies between T_1 and T_2, and T_3 and T_4 depend on a few factors:

- → Operating system overhead,
- → TCP message queuing
- → routing and transmission delays
- → etc.

To understand the problem, let's identify the ideal scenario:



Let $\Delta T_{1,2}$ be the real UTC time difference between the events T_1, and T_2
Similar for $\Delta T_{3,4}$

If the time server reported its local time at T_3, and the process just added $\Delta T_{3,4}$ to it, then you're done.

But, since the processes local time isn't guaranteed, and also that it can't know when T_3 happened on its own local clock, $\Delta T_{3,4}$ cannot be known.

All the process knows is the RTT, measured on its local clock, along with the received T_3 response.

A decent estimate is to just divide RTT by 2, which assumes that:

- → $\Delta T_{1,2} \approx \Delta T_{3,4}$
- → Time between T_3 and T_2 is minimal.

These are reasonable assumptions on a LAN connection

August 21st 2019

The algorithm can then be made more accurate by:

- repeating it multiple times
- adjusting it if the minimum amount of time it takes to send it is known.

Suppose that $\Delta T_{1,2}$ is at minimum equal to min_1, and $\Delta T_{3,4} \rightsquigarrow min_2$.

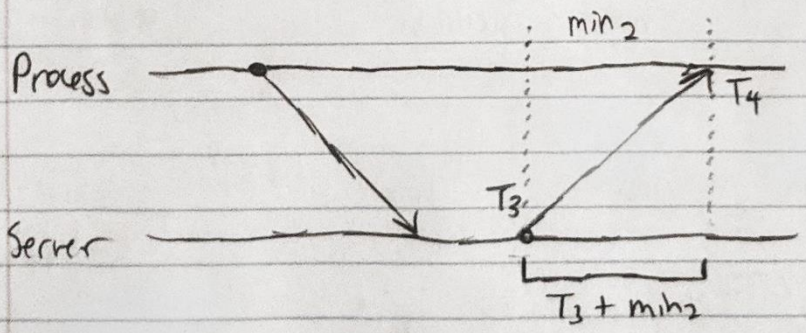
We can bound the actual time when the process actually receives the time to:

$T_3 + min_2$, and
$T_3 + RTT - min_1$

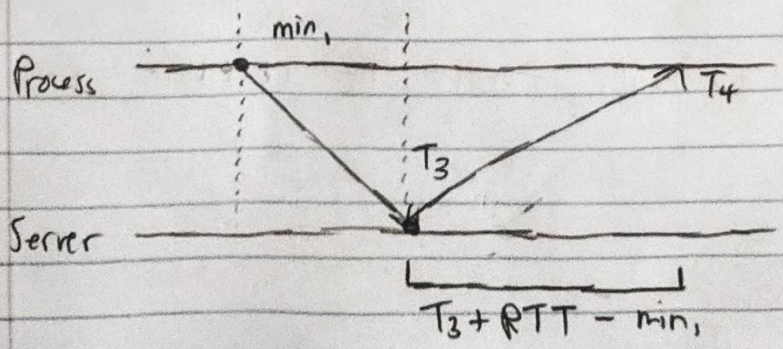
$T_3 + min_2$ would be the earliest possible time that the process could've received the response, and

$T_3 + RTT - min_1$ is the max:

Illustration of the two scenarios:



This means that the response took the absolute minimum time necessary to respond.



This assumes that the response is sent instantly after it was received.

So $T_2 = T_3$, and
$T_2 - T_1 = min_1$

August 21st 2019

Some caveats to Cristians Algorithm:
→ Don't decrease clock value, only increase. May violate event ordering.
→ Allowed to increase / decrease the speed of clock (only available in software clocks).
→ Abnormal readings may occur (possibly due to network hiccups), so discard them if it's not within expected error range.

This algorithm will only synchronize if the RTT is small compared to the required accuracy.

Usually, network latency over LAN is less than 1ms, on Ethernet. Since the time() UNIX syscall is non-blocking it's generally extremly tiny.

So Cristians algorithm works pretty well over LAN, but not on WAN, where latencies are much higher

Internal Synchronization: The Berkeley Algorithm

Sometimes, you might not require every node to know some external time (usually UTC). In this situation, the Berkeley Algorithm can be used.

Unlike Cristians which attempts to reach out to a dedicated time service, Berkely flips it and has a time service asking all the nodes for their times

The core idea is to average out all the times among the system, and adjust them to match the average.

Let's call the time service the master, and the rest slaves.

August 21st 2019

Periodically, the master polls each machine and asks for their time, Cristian's algorithm can be used to compensate for latency.

Then, calculate the average time (including its own local time).

Send an offset to each slave. The offset, as opposed to just sending the average timestamp avoids the latency problem.

Over time, the average will reduce the skew of the system overall.

If the master dies, then a new master can be promoted through leader election (later).

Logical Synchronization

Clock synchronization is pretty hard, so if actual timestamps aren't needed, then you'll only require the relative ordering of events.

We can call this type of clock a logical clock

Lamport Timestamps (Logical Timestamps)

Used by almost all distributed systems, especially cloud computing. Any time logical event ordering is needed, this is typically what's used.

To write logical ordering, we denote $\rightarrow$ to mean "Happens before".

August 21st 2019

There are 3 rules to follow:

1) $a \rightarrow b$, if time(a) < time(b) and both a and b occur in the same process (and thus have the same local clock).

2) If a message m is sent from P_1 to P_2, then send(m) → receive(m).

This means that the sending of a message always happens before it's received.

3) If $a \rightarrow b$, and $b \rightarrow c$, then $a \rightarrow c$.

Note that → doesn't imply that b is caused by a, or is related at all.

An example is if no messages are passed at all, in which case there is no relation between processes.

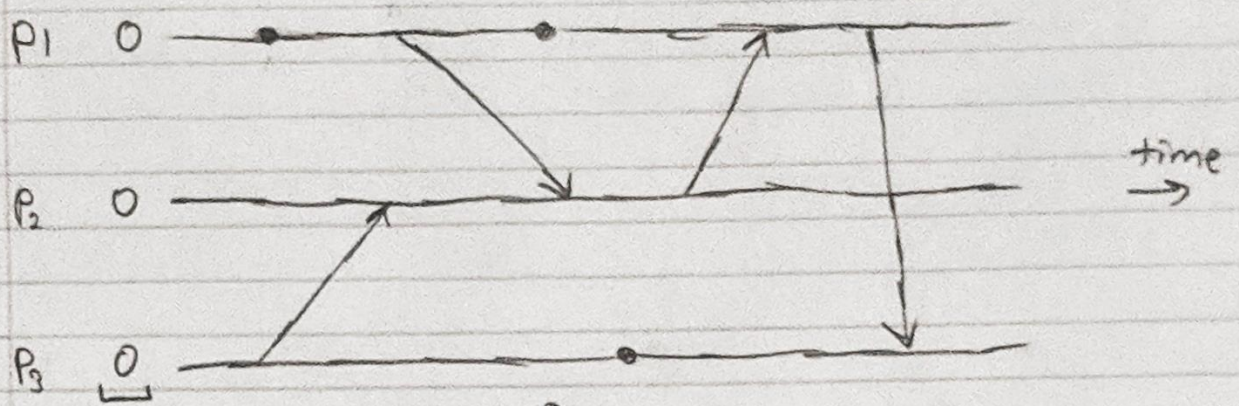
With this ordering, we can track the sequence of events even if all the local clocks in the system are all completely skewed.

To do this, we assign logical (Lamport) timestamps to each event in a particular way:

Suppose each process has its own local counter:
- Starts at 0
- Increments when local event or send occurs
- Sent messages carry the timestamp
- Received messages update the clock by:
  max(local counter, message timestamp) + 1.

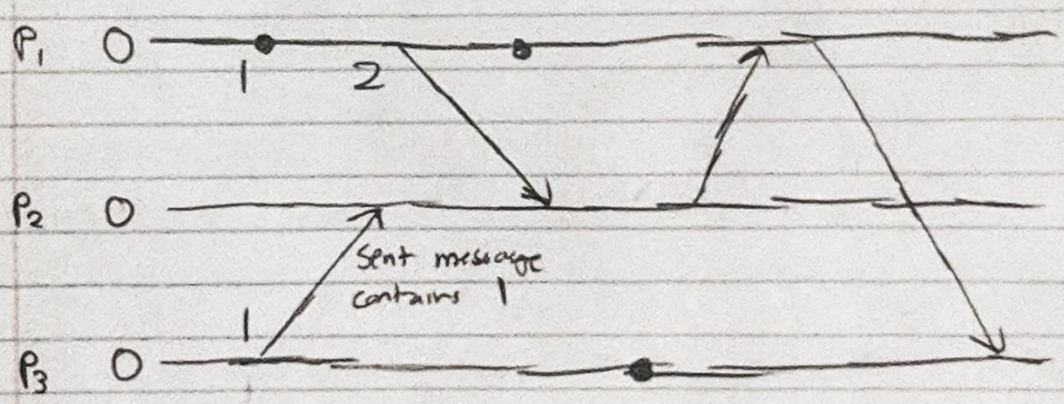
August 21st 2019

Suppose we've got the following scenario:



clocks all initialized to 0.

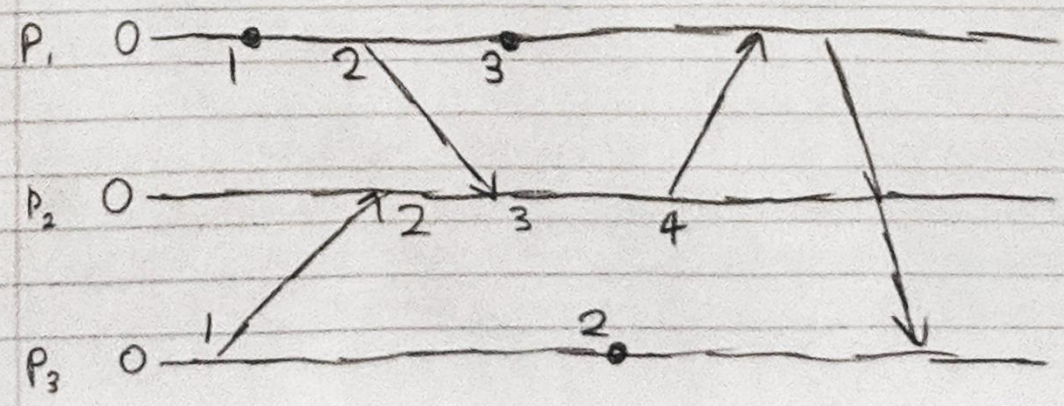
Applying local increments and sends, we can fill out the timestamps for the first few events in P_1 and P_3.



For P_2, the timestamp of the receive message will be: max(local clock, message timestamp) + 1

$= \max(0, 1) + 1$

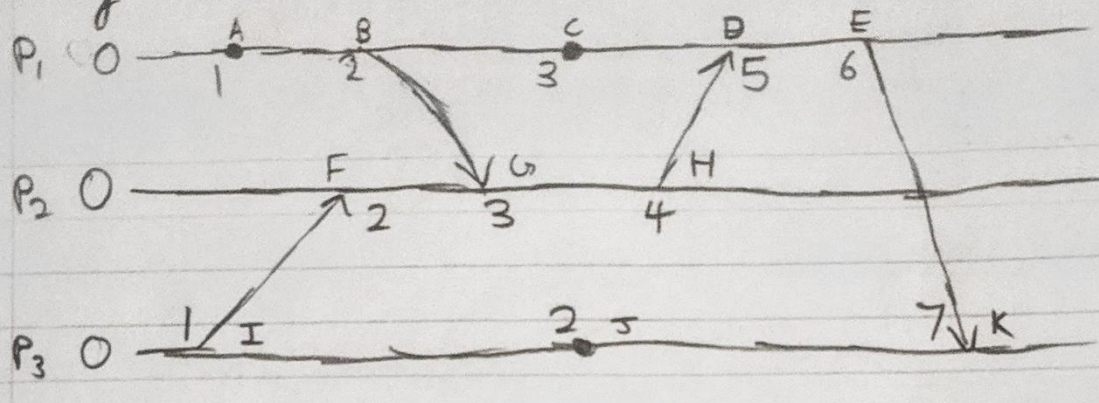
$= 2$



Continuing on...

August 21st 2019

Filling out the rest of the diagram, and labelling the events,



Let's look at some causality relationships:

$A \to B$
$B \to G$
$A \to G$
} relates to {
$1 < 2$
$2 < 3$
$1 < 3$

Here, we see that for some events $a \to b$, we can say that timestamp(a) < timestamp(b).

However, the timestamps alone don't necessarily imply a causal relationship, since the counting is still local:

For example, take C and G. Despite being stamped with 3, the relationship $C \to G$ does not exist as there isn't a guarantee that C happens before G.

This can also be seen as having no path that contains both C and G.

Thus, C and G are a pair of parallel events. The same can be seen with I and C.

So, Lamport timestamps are not guaranteed to be ordered or unequal for parallel events.

$a \to b \Rightarrow$ timestamp(a) < timestamp(b)

BUT

timestamp(a) < timestamp(b) $\Rightarrow a \to b$ OR a, b are parallel

August 21st 2019

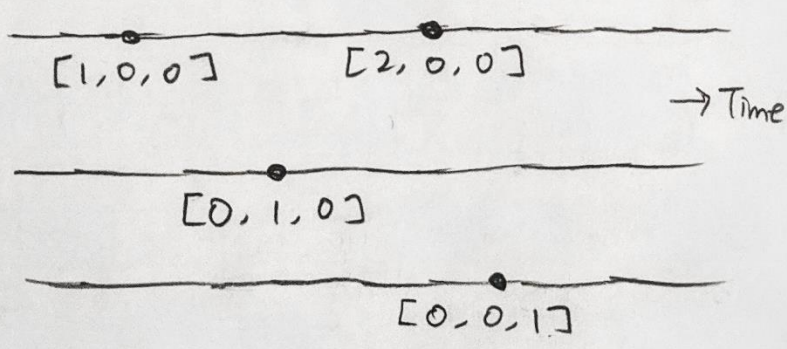
How do we modify the system to allow us to determine whether events occur in parallel or causally, given their timestamps?

Vector Timestamps

The main reason why determining whether events are occurring concurrently or causally in the Lamport system is that on message receives, the local ordering of events may effectively be overriden. Basically, the events got mixed together.

The core idea of a vector timestamp is to separate the local histories for each process, and just keep track of external clocks only when receiving a message.

This is done by time stamping each event as a series of Lamport stamps:



(only local events shown for now).

The increment rules for the local counters are very similar to that of the Lamport clock:

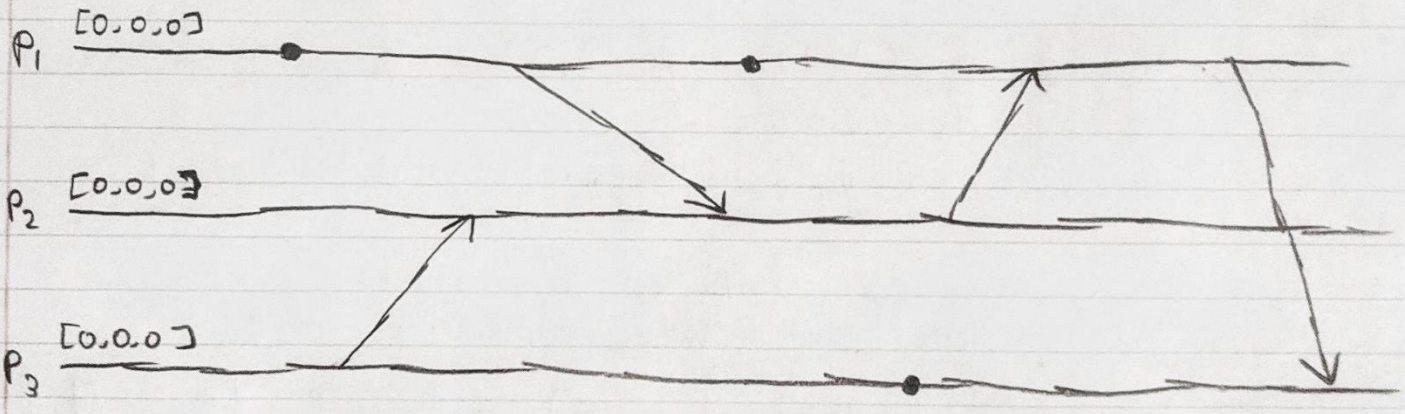
- Starts at $[0, \ldots, 0]$
- For the i^{th} process, on a local event or send, increment the i^{th} element by one.
- Each message carries the send-event's vector timestamp $V_{msg}[\ldots]$

August 22nd 2019

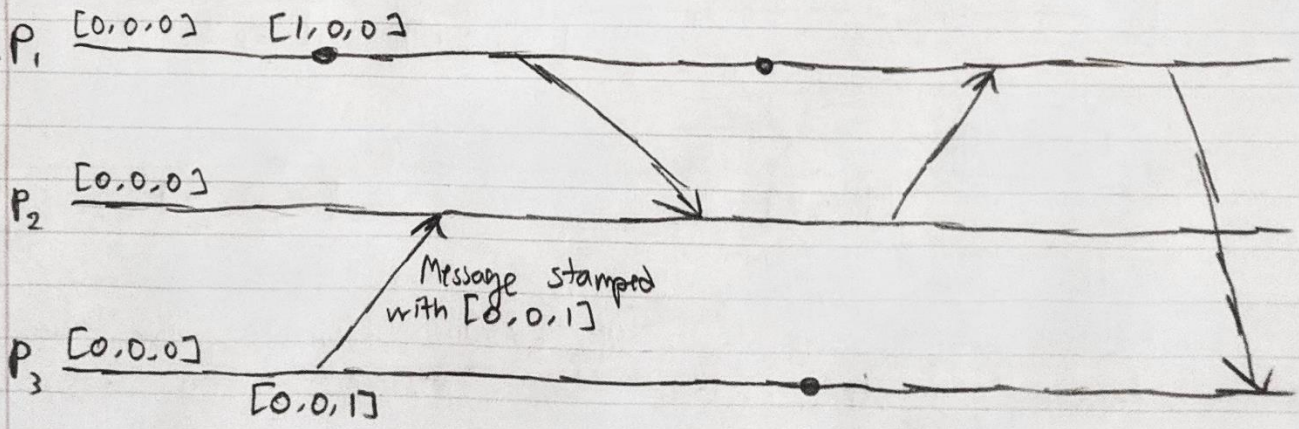
- Received messages update the counter by:
  (for the i^{th} process)
  1) Increment the i^{th} element by one: $V_i[i] = V[i] + 1$
  2) For all other entries in V_i,
     $V_i[j] = max(V_{msg}[j], V_i[j])$ for $j \neq i$

It's similar to the lamport counter, but the received message rule is modified to increment its own element separately from the others.

Using the previous example,



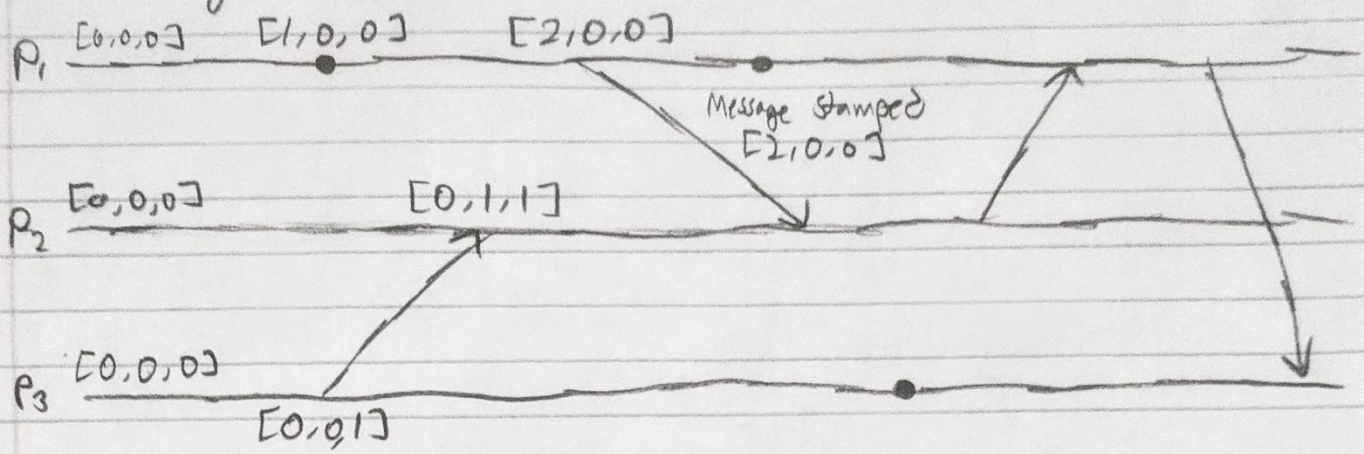
Iterating forward, P_1 has a local event, and P_3 sends a message to P_2. The stamps look like this:



The resulting increment when P_2 receives the message, is that it will increment $V_2[2]$ by 1 and get the max values from the message stamp and its own local clock.

August 22nd 2018

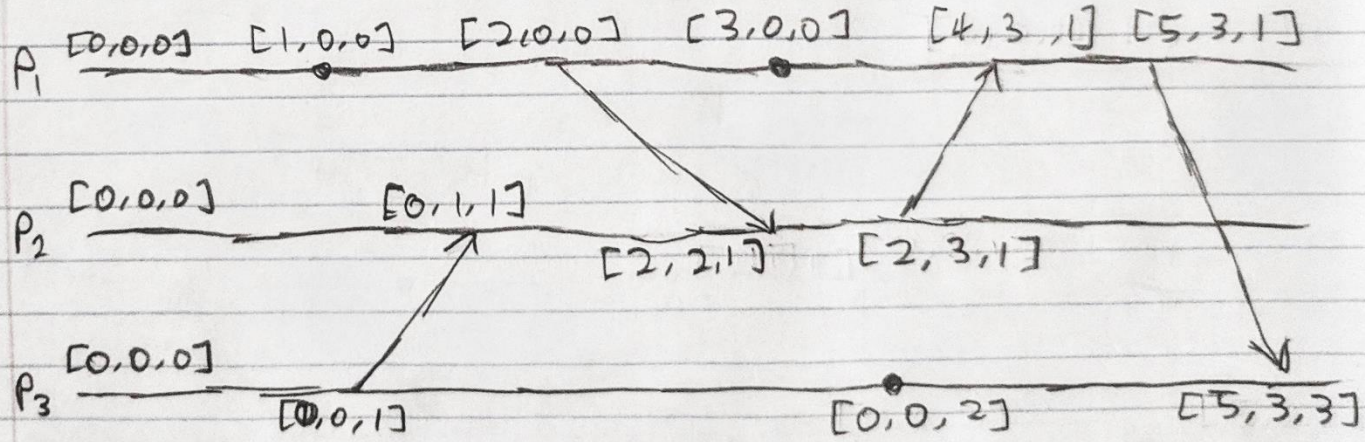
The resulting stamp for P_3's receive is [0,1,1].



For the second message P_2 receives, the stamp will be:

$V_2[1] = \max(V_2[1], V_{msg}[1]) = V_{msg}[1] = 2$
$V_2[2] = V_2[2] + 1 = 2$
$V_2[3] = \max(V_2[3], V_{msg}[1]) = V_2[3] = 1$

So the stamp will be [2,2,1]
The rest of the values are as follows:



With these timestamps, we can track down the causal history for all events, unlike the Lamport timestamps, where only the reverse was true.

So why do vector timestamps let us do this?

August 22nd 2019

1) $V_i[i]$ is essentially tracking the # of events that have occured at process i. So it acts like the local logical clock.

2) $V_i[j]$ for $j \neq i$ indicates the number of events that have occurred at process j that process i knows about. It acts as process i's into about process j's local logical clock

If we look at two timestamps T_1 and T_2, we can observe that T_1 cannot be parallel to T_2 if all elements of T_1 are $\leq T_2$'s elements. T_1 happens before if the previous is true, and at least one element is less than the corresponding element in T_2.

Defined mathematically,

$T_1 = T_2 \iff T_1[i] = T_2[i]$ for all $i = 1, \ldots, N$

$T_1 \leq T_2 \iff T_1[i] \leq T_2[i]$ for all $i = 1, \ldots, N$

T_1 and T_2 are causally related given the following relationship:

$$T_1 < T_2 \iff T_1 \leq T_2 \wedge \exists j \cdot 1 \leq j \leq N \wedge T_1[j] < T_2[j].$$

For example, $[2,3,1] < [4,3,1]$, and in the diagram, this send event occurs before the receive.

Two events are parallel given the following relationship:

$$\underbrace{T_1 \parallel T_2}_{T_1 \text{ and } T_2 \text{ are parallel}} \iff \neg(T_1 \leq T_2) \wedge \neg(T_2 \leq T_1).$$

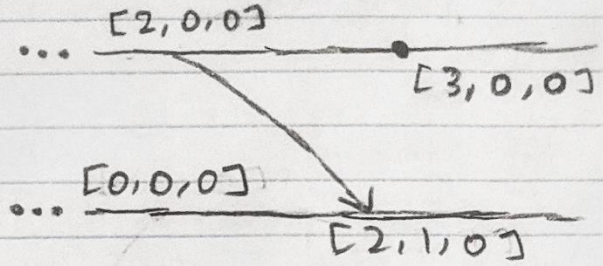
In other words, both T_1 and T_2 have elements that are larger than the corresponding element in the other.

ex $[3,0,0]$ and $[0,0,2]$.

August 22nd 2019

This works because at every event, at least one element in the vector is always incremented.

The intuition would then be that if events diverged, then so would the timestamps:



Here, the last two events [3,0,0] and [2,1,0] separately had their elements incremented, resulting in an element in each that is higher than the other:

[2,0,0] became [3,0,0] and
[2,0,0] / [0,0,0] became [2,1,0]

So they must occur in parallel.

Leader Election

In a lot of distributed systems, having a leader/coordinator/orchestrator is useful, and comes up a lot.

In particular, a leader managing replicas is a common pattern seen when replicating data.

If all processes are the same, then election algorithms try to designate a coordinator among them.

August 22nd 2019

A general strategy is to do the following:
- Assign all processes with a unique id, id(P).
- Locate the process with the highest id

Assume that every process knows the id of all other processes. But processes don't know which other ones are down.

The Bully Algorithm

Suppose there are N processes, and $id(P_k) = k$.

The algorithm starts when a process finds that the coordinator is down. An election is then held:

1) P_k sends an ELECTION message to all processes with higher identifiers, $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$

2) If there's no response, P_k wins and becomes coordinator.

3) If there is a response, it takes over and P_k is done. Wait for VICTORY message to be received until timeout.

At any time, a process can suddenly receive an ELECTION message, and should send an OK message to indicate it is taking over.

The receiver holds another election (unless in the middle of one), and eventually, the highest id process will emerge as the victor.

That process will broadcast to everyone indicating it is the new VICTORY message.

August 22nd 2019

If a process that was previously down comes back up, it will also hold an election. By the process of the algorithm, if it was the highest valve process, it'll be the new coordinator.

The algorithm has a $\Theta(N^2)$ relation to the number of messages sent over the network.

This happens if the lowest valued process holds an election, and all other processes are alive. Then $N-1$, $N-2$, ... messages are sent, leading to $\Theta(N^2)$.

Note that the Bully algorithm has some pitfalls, where there are certain cases where it may not terminate, or will have instances where multiple coordinators exist at once.

These are discussed in further detail in the Consensus chapter.

Ring Election

Suppose now that N processes are organized logically in a ring. This may be enforced through the network topology, but can also just be organized like that to make it easier to apply these algorithms.

There's several algorithms for leader election in this kind of setup, so we'll only focus on one particular one.

August 23rd 2019

Here, assume that:

- i^{th} process P_i has communication channel to $P_{(i+1) \bmod N}$
- All messages are sent clockwise around the ring

If P_i sees that the coordinator is down, it will start an election by sending an ELECTION message that contains $id(P_i)$.

If the sucessor is down, P_i will contact $P_{(i+2 \bmod N)}$, and so on until a running process is found.

Then, if a process P_j receives that ELECTION, will compare its id to the one in the message. Afterwards,

- If P_j's id is higher, it'll forward its own
- If P_j's id is lower, it'll forwoid the received message.
- If P_j's id was in the message, then it must be the greatest and thus sends a VICTORY message around in a circle so everyone knows.

Assuming failures don't occur during the election, the process is guaranteed to end in $\Theta(n)$.

Worst case: $3N-1$ messages,
Best case: $2N$ messages,

What if multiple elections are started at the same time? A small change needs to be made:

Each process will cache the initiator of each election message received. The cache should be updated if any higher id initiator's message comes by.

August 23rd 2019

Now, suppress any election/elected messages of any lower-id initiators. This means any duplicate election will be eventually squashed.

Ring Elections: Signup

similar to Candidate replacement, but the rules for when P_i receives a message:

- If the message doesn't contain P_j's id, then append it on, and send it forwards.
- If the message contains P_j's id, then we know it's gone in a full circle, and thus we can determine who the highest id of the alive bunch.

Once we know who should be coordinator, P_i circulates to everyone a VICTORY message containing the member that should be coordinator.

If multiple elections are initiated at the same time, then they'll both complete with the circulation of the exact same coordinator, so it won't break. But, it may waste bandwidth

A similar squashing strategy employed in the candidate replacement Strategy may also solve this issue.

The best & worst cases are $2N$, which is less than the Candidate replacement strategy. But, the size of the messages are much larger.