Thanks for Reading!

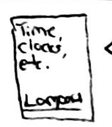
This Zine was brought to you by UCSC's CMPS 128!

Reader's Digest:

A Summarization of

Lesslie Lamport's Time, Clocks, and the Ordering of Events in a distributed System

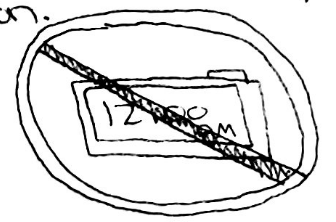
If you are a CS student who is interested in distributed systems, or have had to read the paper mentioned in the title, and didn't quite get it, then this Zine is for you!



← kind of hard to understand!

So, lets get into it! First, what is a distributed system? A distributed system is a group of distinct processes, that usually are separated by some distance, that communicate with eachother by passing messages. Both this Zine and the paper will focus on networks of connected computers, like ARPA net. But, that is not the only type of system our remarks apply to. For example, a multiprocessing CPU has many similarities due to the distinct individual processes having to communicate.

In distributed Systems, time is weird. We can no longer rely on our wall clock to correctly help us order events. To help us understand how we can order events in a distributed System, we will look at Lamport's "happens-before" relation.



this algorithm is a distributed algorithm. Each process follows the rules, and there is no central leader process or even a central type of storage. This approach is powerful because we can generalize it to implement any desired Synchronization for a distributed multi-process system.

Overall we have discussed most of the topics covered in Lamport's paper. What was not covered can be found under the following subsections:

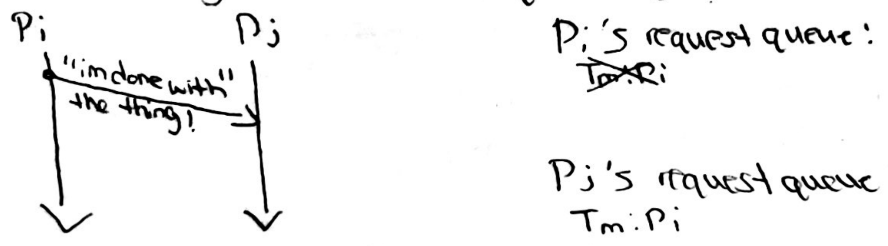
- Anomalous Behavior
- Physical Clocks

Citations for resources used in this Zine:

Lamport, L. (1978). "Time, Clocks, and the ordering of events in a distributed system" (PDF)

"Mutual Exclusion." Wikepedia, Wikimedia Foundation, 21 Apr. 2019,

(3) To give up the resource, process P_i removes any $T_m:P_i$ requests resource message from its request queue.



P_i's request queue: ~~Tm:~~Pi

P_j's request queue $T_m:P_i$

and sends a timestamped P_i releases resource message to every other process on the system

(4) When process P_j receives a P_i releases resource message, it removes any $T_m:P_i$ requests resource from its requests queue.

(5) Process P_i gets to grab the resource when the following two conditions are satisfied:

a. there is a $T_m:P_i$ requests resource message in its request queue which is ordered before any other request in its queue by our relation $\Rightarrow$

b. P_i received a message from all other processes timestamped later than T_m.

The reason real clocks suck in a distributed system, is that they are not totally accurate and don't even keep precise physical time! Because these problems exist, the "happens-before" ($\rightarrow$) relation is defined without using physical clocks.

- wait! What do you mean no physical clocks?!

you

For us to finally define the "happens-before" ($\rightarrow$) relation we need to talk about what an event is. There are three things we will consider an event.

(1) A process executing something is an event
(2) A process sending a message to another process is an event
(3) A process receiving a message from another process is an event

Now for the relation! The "happens-before" ($\rightarrow$) relation is the smallest relation such that:

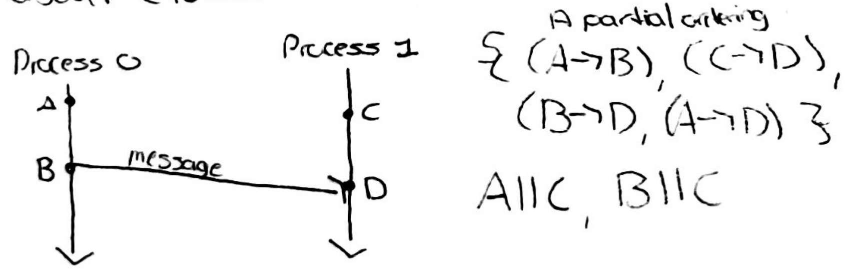
(1) If A and B are events on the same process and A is executed before B, then $A \rightarrow B$.

(2) If A is a send message event, and B is the corresponding receive message event, then $A \rightarrow B$.

(3) If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$

The "Smallest relation such that..." part just means that the set of event pairs e.g ($A \to B$) generated by this relation is finite and contains no arbitrary values.

Notice that <u>there are pairs of events that we cannot use this relation to describe</u>, because one did not happen-before the other. We say these events are parallel. Now that this relation is defined, we can talk about Clocks.

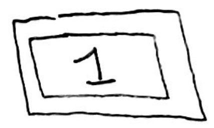


A partial ordering

$\{ (A \to B), (C \to D), (B \to D), (A \to D) \}$

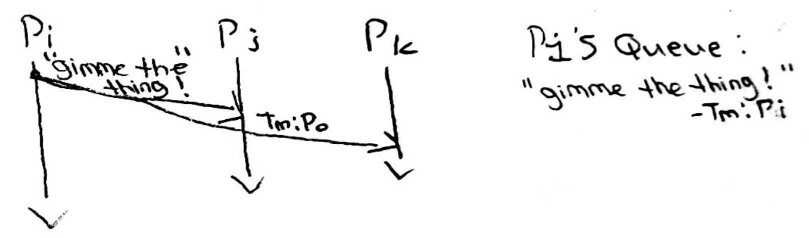
$A \| C$, $B \| C$

We will begin with the idea that a clock is just a way to assign a number to an event, like a counter. These assigned numbers have nothing to do with actual time, so we call them logical clocks. So what does it mean for a system of logical clocks to be correct?



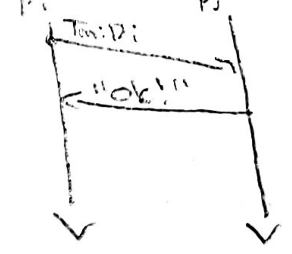
But, what time is it?

(1) To request the resource, process P_i sends the message $T_m:P_i$ requests resource to every other process in the system, and adds that message to its request queue where T_m is the time stamp of the message



P_1's Queue:
"gimme the thing!" - $T_m:P_i$

(2) When Process P_j receives the message $T_m:P_i$ requests resource, it places it on its own request queue and sends an acknowledgement back to P_i saying that P_j successfully received the message.



P_i's request queue:
$T_m:P_i$

P_j's request queue:
$T_m:P_i$

Whoops! I missed a page.

We can use our newly defined "happens-before" relation to help us. The best thing we can say is that if an event A occurs before an event B, then A should happen at an earlier time than B. So basically:

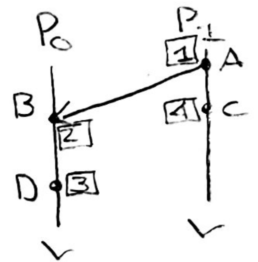
if $A \to B$, then $LC(A) < LC(B)$

LC stands for Logical Clock. The above statement is our Clock Condition. As long as this is satisfied, we can use logical clocks in our system.

- Nice!

- I get to use these all I want

Importantly, we cannot say the opposite or converse of our clock condition is true. That would mean all parallel events must happen at the same time. Check out the example below to see how that would be a problem.



$\{(A \to B), (B \to D), (A \to C)\}$

$B \parallel C$ $D \parallel C$

B and D are concurrent with C. If the converse of our Clock Condition was true, then both B and D would have to happen at the same time as C, but that contradicts our clock condition because $B \to D$.

To make sure that we satisfy our clock condition we need the following two conditions to hold:

(1) If A and B are on the same process, and $A \rightarrow B$, then $LC(A) < LC(B)$

(2) If A is a send event from a process P_1 and B is the corresponding recieve event on a process P_2, then $LC(A) < LC(B)$

Now with the "happens-before" relation and the clock condition defined, we can talk about creating a total ordering of events.

To do this, we are going to need to define a new relation, $\Rightarrow$. You can think of this relation as an upgrade to the "happens-before" relation. The "happens-before" relation gives us a partial ordering of events, with some being parallel. Our new relation will be able to totally order events. For this to happen we need some way to totally order the processes themselves, represented by '$<$'

What the heck does THAT mean?!

(4) A process can send messages directly to any other process.

(5) Each process maintains a request queue that is not seen by the other processes.

(6) Each request queue initially contains the message $T_0 : P_0$ requests resource, where P_0 is the process given the resource at first and T_0 is an initial clock value less than the initial value of any clock.

Now we can focus on the problem without all that pesky implementation!
Lamport

Given these assumptions, Lamport lays out an algorithm to solve our mutual exclusion problem within the context of a distributed system. The algorithm follows the following five rules:

What more lists?
-you

Tell me about it...
-me

So, to solve the problem we can implement our system of clocks following the two rules defined in our $\Rightarrow$ relation. This makes finding a solution to our problem much more straightforward. Before we move forward though, we are going to make some assumptions to simplify the problem, just as Lamport did in the paper. He says in the paper that these assumptions are not essential but help to avoid distracting implementation details.

We will assume:

(1) A process P_0 is initially given access to the resource

(2) For any two processes P_1 and P_2, messages sent from P_1 to P_2 are recieved in the order that they are sent.

(3) Any sent message is eventually received.

It means that we need a way to break ties when the clocks of two events are the same. If we can say a process came before another in some arbitrary way, like say having a lower process ID, then we can break the tie and maintain our total order. We will define our new relation such that:

If A is an event on process P_1, and B is an event on process P_2, then $A \Rightarrow B$ if and only if either

(1) $LC(A) < LC(B)$

- or -

(2) $LC(A) < LC(B)$ and $P_1 < P_2$

It is important to understand that depending on how you implement your logical clocks, and what way you decide to order your processes will yield different correct totally ordered outcomes.

However, the "happens-before" ($\rightarrow$) relation is <u>uniquely</u> determined by the events. No matter your implementation, the happens before relation will yield the same unique result.

By now, you must be wondering, "Why are we going through the trouble of defining these special clocks just to get this total order?"

I WAS wondering!

Lamport actually shows why implementing a total order is so useful in distributed systems, which brings us to a version of the mutual exclusion problem.

If you don't know what the mutual exclusion problem is, it is the problem of:

"How can a software system control multiple process' access to a shared resource, when each process needs exclusive control of that resource when using it?" (Wikipedia).

In the paper we are presented with the following hypothetical problem. A system composed of a fixed collection of processes must share a single resource that can only be used exclusively. The processes need to work together to synchronize themselves to avoid conflict.

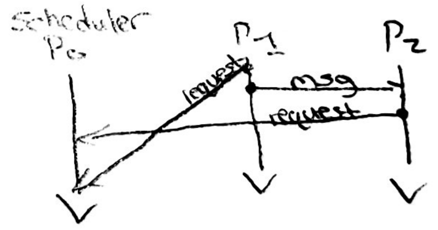
So, we need an algorithm that satisfies the following three conditions:

(1) A process that was given access to the resource must give it up before the resource can be given to another process

(2) Requests for the resource must be answered in the order that they are made.

(3) If every process that was given the resource eventually gives it up, then every request is eventually answered.

Lamport says it is important to realize that this is a non-trivial problem. You can't just designate a central process to delegate which process gets the resource at what time. For example:



requests can get crossed, violating condition (2).