Understanding Lamport Timestamps with Python's multiprocessing library

Steven Van Dorpe
Sep 29, 2018 · 11 min read

Everyone who has been working with distributed systems or logs from such a systems, has directly or indirectly encountered **Lamport Timestamps**. Lamport Timestamps are used to (partially) order events in a distributed system. The algorithm is based on **causal ordering** of events and is the foundation of more advanced clocks such as Vector Clocks and Interval Tree Clocks (ITC).

In this article we will first briefly introduce the concept of logical clocks, explain why ordering of events in distributed systems is needed and discuss some alternatives. Then we'll go over the algorithm of Lamport Timestamps and work an example with three processes. Next, we'll implement this example in easy-to-understand code using Python's *multiprocessing* library. To top it all off, we'll transform our code into an implementation with **Vector Clocks**.

Logical clocks

To understand why logical clocks are needed, it is important to understand what a distributed system is. A **distributed system** is a system whose components (here called **processes**) are located on different networked computers, which then coordinate their actions by passing **messages** to one other.

One of the main properties of a distributed system is that it **lacks a global clock.** All the processes have their own local clock, but due to clock skew and clock drift they have no direct way to know if their clock is in check with the local clocks of the other processes in the system, this problem is sometimes referred to as the **problem of clock synchronization.**

Solutions to this problem consist of using a central time server (Cristian's Algorithm) or a mechanism called a **logical clock.** The problem with a central time server is that its error depends on the round-trip time of the message from process to time server and back.

Logical clocks are based on capturing chronological and causal relationships of processes and ordering events based on these relationships. The first implementation, the Lamport timestamps, was proposed by **Leslie Lamport** in 1978 and still forms the foundation of almost all logical clocks.

Lamport Timestamps algorithm

A Lamport logical clock is an incrementing counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality).

The algorithm of Lamport Timestamps can be captured in a few rules:

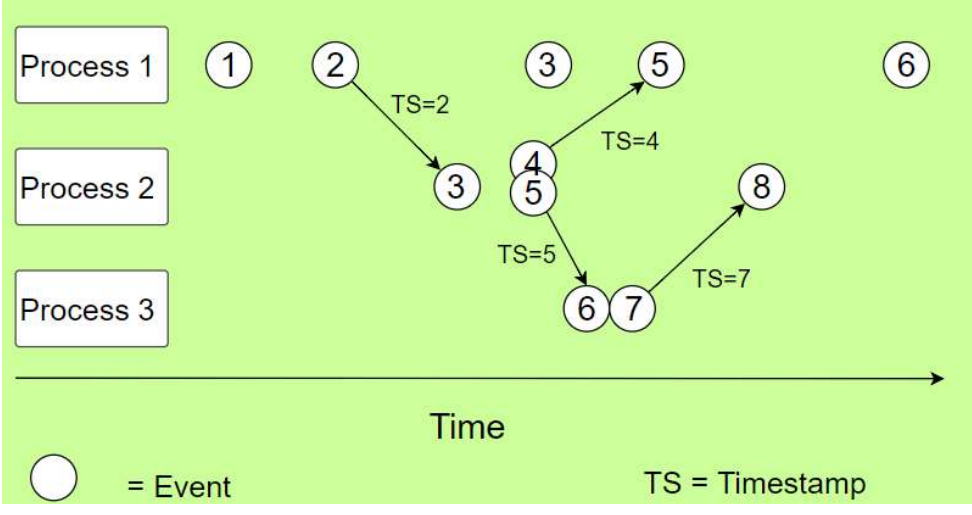
- All the process counters start with value 0.
- A process increments its counter for each event (internal event, message sending, message receiving) in that process.
- When a process sends a message, it includes its (incremented) counter value with the message.
- On receiving a message, the counter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

Looking at these rules, we can see the algorithm will create a minimum overhead, since the counter consists of just one integer value and the messaging piggybacks on inter-process messages.

One of the shortcomings of Lamport Timestamps is rooted in the fact that they only partially order events (as opposed to total order). **Partial order** indicates that not every pair of events need be comparable. If two events can't be compared, we call these events **concurrent**. The problem with Lamport Timestamps is that they can't tell if events are concurrent or not. This problem is solved by Vector Clocks.

Example

If the descriptions above seem a bit fuzzy to you, take a look at the following example. All processes start with their internal counter (clock) at zero. For each event, represented by a circle, the counter is increased by one. When a process receives a message it sets its counter to the greater of its internal counter and the timestamp included in the message.



Lamport Timestamps example

For example, consider the first event in process 2, where it receives a message from process 1. The local counter of process 2 is at this moment at 0, but incremented to 1 because receiving a message is an event. The timestamp included in the message will be 2: the local time of process 1 incremented by 1 (1+1). Process 1 will at the same time of sending the message also increment its own clock with 1.

To set the new time in process 2, take the maximum between the received timestamp and its own local time (*max(2,1)*) and increment it with 1. This will result in a new timestamp with value 3. This makes sense, since the message can never be received before or at the same time of sending it.

Implementation

Our implementation will run **three separate processes** that can communicate with each other through messages. **Each process has its own internal counter** that will be updated with each event. Our script will print a line for each event together with the updated internal counter and the time on the machine running the processes.

Before we get started, we need to import some functions from standard modules: *Process* and ***Pipe*** from *multiprocessing* to run multiple Python processes with one script, *getpid* from *os* to get the process id of each process, and *datetime* from *datetime* to get the current time.

If you need more information on the multiprocessing library, please read the documents, watch this great tutorial series or take a look at this (free) webinar.

```
from multiprocessing import Process, Pipe
from os import getpid
from datetime import datetime
```

Then we create some helper functions. The first one simply **prints the local Lamport timestamp** and the actual time on the machine executing the processes. Note that printing the 'actual' time doesn't make sense in a real distributed system, since the local clocks won't be synchronized with each other.

```
def local_time(counter):
    return ' (LAMPORT_TIME={}, LOCAL_TIME={})'.format(counter,
                                                      datetime.now())
```

The second one calculates the **new timestamp when a process receives a message.** The function takes the maximum of the received timestamp and its local counter, and increments it with one.

```
def calc_recv_timestamp(recv_time_stamp, counter):
    return max(recv_time_stamp, counter) + 1
```

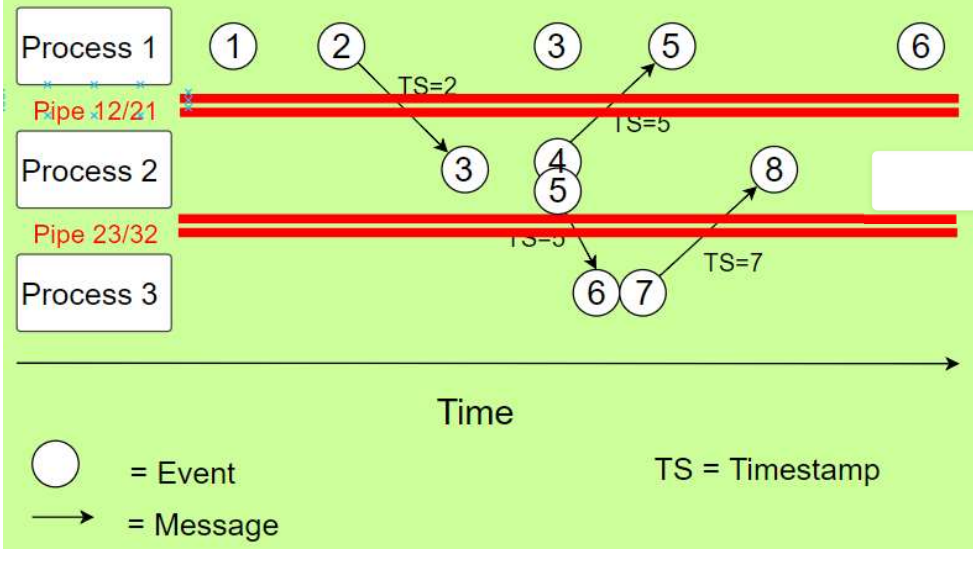
Next, we want to create a function for every event that may occur. In our example they are three events: *event* (any local event), *message sent* and *message received*. To make our code easy to read, **the event functions will return the updated timestamp** for the process where the event takes place.

The *event* event takes the local *counter* and the process id (*pid*), increments the counter by one, prints a line so we know the event took place and returns the incremented counter.

```
def event(pid, counter):
    counter += 1
    print('Something happened in {} !'.\
          format(pid) + local_time(counter))
    return counter
```

The *send_message* event also takes the *pid* and *counter* as input, but additionally requires a *pipe*. A **Pipe** is an object from the multiprocessing library that represents a two-way connection between two processes. Every pipe consists of two **connection objects**, one in each direction. To send or receive a message we need to call the *send* or *recv* function on these connection object.

If you take a look at our example, you can see we only need two message pipes. One between process 1 and process 2, and one between process 2 and process 3. So, our example will have four connection objects: *pipe12*, *pipe21*, *pipe23* and *pipe32* (see below).



Example with two pipes

Our *send_message* event first increments the counter by one, then sends an actual message (content is not important here) and its incremented timestamp, and prints a

short statement including the new local Lamport time and the actual time on the machine. Just like all our event functions it returns the new local timestamp.

```
def send_message(pipe, pid, counter):
    counter += 1
    pipe.send(('Empty shell', counter))
    print('Message sent from ' + str(pid) + local_time(counter))
    return counter
```

The *recv_message* event takes the same three arguments as *send_message*. It receives both the actual message and the timestamp by invoking the *recv* function on the ***pipe***. Then it calculates the new timestamp with our previously created *calc_recv_timestamp* function, and prints a line including the updated counter and the actual time on the machine.

```
def recv_message(pipe, pid, counter):
    message, timestamp = pipe.recv()
    counter = calc_recv_timestamp(timestamp, counter)
    print('Message received at ' + str(pid) + local_time(counter))
    return counter
```

We're all set up to start creating the definitions of our three processes. Every process starts with **getting its unique process id** (this is the actual process id of the process running on our machine) and **setting its own counter to 0**. Then the counter gets updated by invoking the different event functions according to our previously illustrated example and passing the returned value to the counter.

```
def process_one(pipe12):
    pid = getpid()
    counter = 0
    counter = event(pid, counter)
    counter = send_message(pipe12, pid, counter)
    counter = event(pid, counter)
    counter = recv_message(pipe12, pid, counter)
    counter = event(pid, counter)

def process_two(pipe21, pipe23):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe21, pid, counter)
    counter = send_message(pipe21, pid, counter)
    counter = send_message(pipe23, pid, counter)
    counter = recv_message(pipe23, pid, counter)


def process_three(pipe32):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe32, pid, counter)
    counter = send_message(pipe32, pid, counter)
```

Note that thus far nothing happens if you execute the code. This is because none of the processes are actually created, let alone started. In the *main* part of our script we will create the two **pipes** (***Pipe()***) and three processes (*Process()*), needed to run our script

successfully. To start the processes, we need to call *start* and *join* on each process. *Join* assures us that *all* processes will be completed before quitting.

```
if __name__ == '__main__':

    oneandtwo, twoandone = Pipe()
    twoandthree, threeandtwo = Pipe()

    process1 = Process(target=process_one,
                       args=(oneandtwo,))
    process2 = Process(target=process_two,
                       args=(twoandone, twoandthree))
    process3 = Process(target=process_three,
                       args=(threeandtwo,))

    process1.start()
    process2.start()
    process3.start()

    process1.join()
    process2.join()
    process3.join()
```

Now, try running the code. If everything went right you get an output very similar to the one below. Notice that every process has its own unique process id (these id's will change every time you run the code):

- Process 1 has id 16112
- Process 2 has id 18968
- Process 3 has id 9584

Each event has indeed printed a line, including the type of event and the updated Lamport timestamp. Take a minute to compare the timestamps with stamps in our example, you'll see they all agree with each other. You can test new examples by drawing a timeline and change the process definitions accordingly.

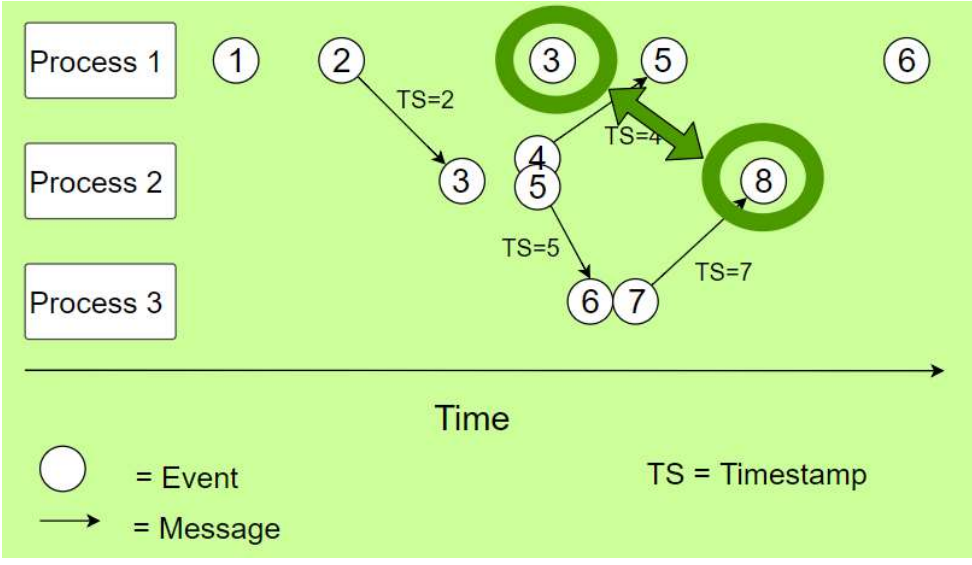
```
Something happened in 16112 ! (LAMPORT_TIME=1, LOCAL_TIME=2018-09-22 19:25:14.613288)
Message sent from 16112 (LAMPORT_TIME=2, LOCAL_TIME=2018-09-22 19:25:14.614286)
Something happened in 16112 ! (LAMPORT_TIME=3, LOCAL_TIME=2018-09-22 19:25:14.614286)
Message received at 18968 (LAMPORT_TIME=3, LOCAL_TIME=2018-09-22 19:25:14.620268)
Message sent from 18968 (LAMPORT_TIME=4, LOCAL_TIME=2018-09-22 19:25:14.620268)
Message sent from 18968 (LAMPORT_TIME=5, LOCAL_TIME=2018-09-22 19:25:14.620268)
Message received at 16112 (LAMPORT_TIME=5, LOCAL_TIME=2018-09-22 19:25:14.620268)
Something happened in 16112 ! (LAMPORT_TIME=6, LOCAL_TIME=2018-09-22 19:25:14.620268)
Message received at 9584 (LAMPORT_TIME=6, LOCAL_TIME=2018-09-22 19:25:14.628248)
Message sent from 9584 (LAMPORT_TIME=7, LOCAL_TIME=2018-09-22 19:25:14.628248)
Message received at 18968 (LAMPORT_TIME=8, LOCAL_TIME=2018-09-22 19:25:14.628248)
```

Script output with Lamport Timestamps

Vector Clocks

As mentioned before, Lamport timestamp have one big shortcoming: **they can't tell you when two events are concurrent.** Going back to our example, by just checking the

timestamps, we could conclude that event 3 in process 1 has happend before event 8 in process 3, but this isn't necessarily true.



Compare event 3 in process 1 with event 8 in process 2

If event 3 takes a few seconds, this won't influence event 8, and thus event 8 will be executed before event 3. You can easily verify this by adding a little bit of delay before executing event 3 in process 1.

```
from time import sleep

def process_one(pipe12):
    pid = getpid()
    counter = 0
    counter = event(pid, counter)
    counter = send_message(pipe12, pid, counter)
    sleep(3)
    counter = event(pid, counter)
    counter = recv_message(pipe12, pid, counter)
    counter = event(pid, counter)
```

Note that the order of our events has changed, while our timestamps have stayed the same.

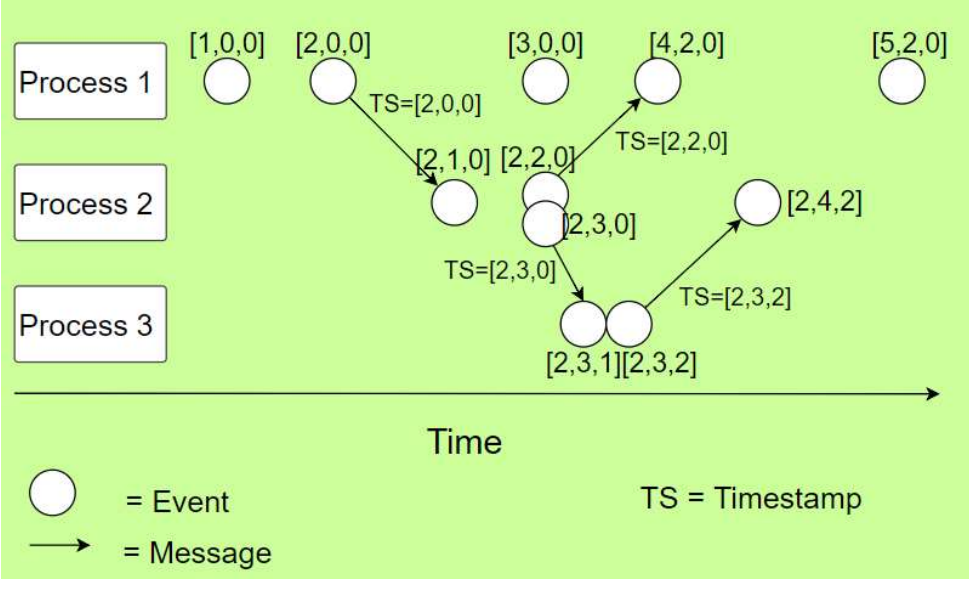
```
Something happened in 15104 ! (LAMPORT_TIME=1, LOCAL_TIME=2018-09-23 11:43:40.366728)
Message sent from 15104 (LAMPORT_TIME=2, LOCAL_TIME=2018-09-23 11:43:40.366728)
Message received at 7292 (LAMPORT_TIME=3, LOCAL_TIME=2018-09-23 11:43:40.366728)
Message sent from 7292 (LAMPORT_TIME=4, LOCAL_TIME=2018-09-23 11:43:40.366728)
Message sent from 7292 (LAMPORT_TIME=5, LOCAL_TIME=2018-09-23 11:43:40.366728)
Message received at 13144 (LAMPORT_TIME=6, LOCAL_TIME=2018-09-23 11:43:40.369720)
Message sent from 13144 (LAMPORT_TIME=7, LOCAL_TIME=2018-09-23 11:43:40.369720)
Message received at 7292 (LAMPORT_TIME=8, LOCAL_TIME=2018-09-23 11:43:40.369720)
Something happened in 15104 ! (LAMPORT_TIME=3, LOCAL_TIME=2018-09-23 11:43:43.368499)
Message received at 15104 (LAMPORT_TIME=5, LOCAL_TIME=2018-09-23 11:43:43.368499)
Something happened in 15104 ! (LAMPORT_TIME=6, LOCAL_TIME=2018-09-23 11:43:43.368499)
```

Output with delay in process 15104

Vector Clocks solve this problem by using **a vector counter** instead of an integer counter. The vector clock of a system with N processes is a vector of N counters, one counter per process. Vector counters have to follow the following update rules:

- Initially, all counters are zero (*[0,0,0]* in our example)
- Each time a process experiences an event, it increments its own counter in the vector by one.
- Each time a process sends a message, it includes a copy of its own (incremented) vector in the message.
- Each time a process receives a message, it increments its own counter in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector counter and the value in the vector in the received message.

When applying vector clocks to our previous example, we get a vector for each event as showed in the images below.



Vector Clocks example

Consider our previous example, where we compared the third event in process 1 with the fourth event in process 2. The vectors read *[3,0,0]* and *[2,4,2]*. **For one event to have happened before another event, all the elements of its vector need to be smaller or equal to the matching elements in the other vector.**

Here, there is clearly a conflict (*3>2 and 0<4*), and thus we can conclude these events are concurrent and have not influenced each other.

To convert our code to vector clocks, we need to make some minimal adjustments. The biggest one lays in how we calculate our new counter when we receive a message. Since we're going to work with **arrays**, we need to calculate the maximum between the counter in the message and the process's counter **for each element**.

```
def calc_recv_timestamp(recv_time_stamp, counter):
    for id in range(len(counter)):
        counter[id] = max(recv_time_stamp[id], counter[id])
    return counter
```

Next, again because we're working with arrays, we need to replace

```
counter += 1
```

by

```
counter[pid] += 1
```

in all our events.

Lastly, we want to change our process definitions by replacing our initial counter by a vector with length three, and manually setting our process id according to its position in the vector. For process one

```
pid = getpid()
counter = 0
```

should be replaced by

```
pid = 0
counter = [0,0,0]
```

If you change this accordingly for all the processes and run your code, you get the following output. Compare the output with our example and conclude that it's working fine.

```
Something happened in 0 ! (LAMPORT_TIME=[1, 0, 0], LOCAL_TIME=2018-09-24 15:32:20.992569)
Message sent from 0 (LAMPORT_TIME=[2, 0, 0], LOCAL_TIME=2018-09-24 15:32:20.992569)
Something happened in 0 ! (LAMPORT_TIME=[3, 0, 0], LOCAL_TIME=2018-09-24 15:32:20.992569)
Message received at 1 (LAMPORT_TIME=[2, 1, 0], LOCAL_TIME=2018-09-24 15:32:21.001545)
Message sent from 1 (LAMPORT_TIME=[2, 2, 0], LOCAL_TIME=2018-09-24 15:32:21.001545)
Message sent from 1 (LAMPORT_TIME=[2, 3, 0], LOCAL_TIME=2018-09-24 15:32:21.001545)
Message received at 0 (LAMPORT_TIME=[4, 2, 0], LOCAL_TIME=2018-09-24 15:32:21.001545)
Something happened in 0 ! (LAMPORT_TIME=[5, 2, 0], LOCAL_TIME=2018-09-24 15:32:21.001545)
Message received at 2 (LAMPORT_TIME=[2, 3, 1], LOCAL_TIME=2018-09-24 15:32:21.004537)
Message received at 1 (LAMPORT_TIME=[2, 4, 2], LOCAL_TIME=2018-09-24 15:32:21.004537)
Message sent from 2 (LAMPORT_TIME=[2, 3, 2], LOCAL_TIME=2018-09-24 15:32:21.004537)
```

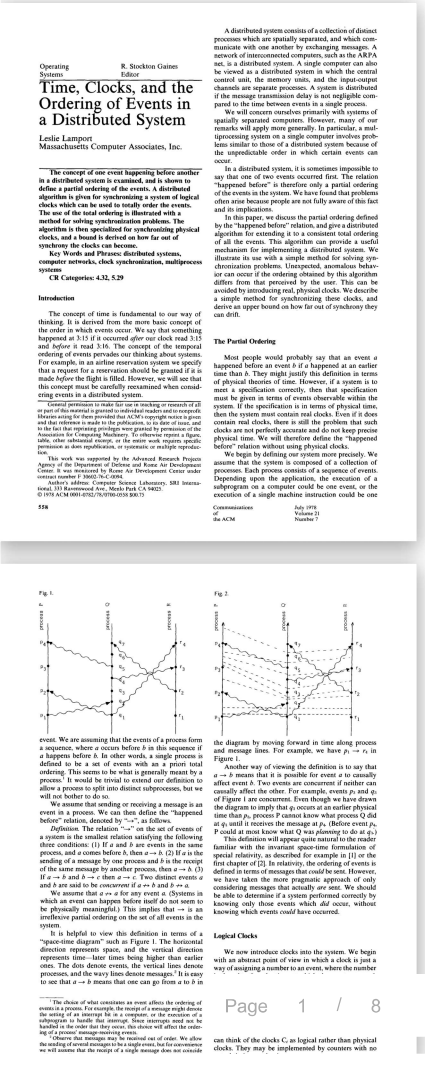
Output of Vector Clocks script

Once you properly understand how Lamport Timestamps and Vector Clocks work, it might be interesting to look into some other logical clocks such as **Matrix Clocks**,

Version Vectors or **Interval Tree Clocks**. Or, other mechanisms for tracking changes to data in a distributed system such as **Hash Histories.**

— Please feel free to bring any inconsistencies or mistakes to my attention in the comments or by leaving a private note. —

Sources



Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems
CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center under Contract No. F30602-76-C-0094 with the Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.

Fig. 2.

event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.[1] It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the "happened before" relation, denoted by "$\rightarrow$", as follows.

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a. (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a "space-time diagram" such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.[2] It is easy to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_3$ in Figure 1.

Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3, process P cannot know what process Q did at q_3 until it receives the message at p_4. (Before event p_4, P could at most know what Q was *planning* to do at q_3.)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no

[1] The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process' message-receiving events.

[2] Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide

Page 1 / 8

Time, Clocks, and the Ordering of Events in a Distributed System, Leslie Lamport, 1978

Cloud Computing Concepts, Part 1 | Coursera

About this course: Cloud computing systems today, whether open-source or used inside companies, are built using a...

www.coursera.org

Partially ordered set - Wikipedia

In mathematics, especially order theory, a partially ordered set (also poset) formalizes and generalizes the intuitive...

en.wikipedia.org

Lamport Timestamps Python Multiprocessing Vector Clocks Distributed Systems

About Help Legal