Lab 2: Simulation for Clock Synchronization in Distributed System using Lamport's Algorithm.

Time is an important practical issue in distributed system. For example, we require computers around the world to timestamp electronic commerce transactions consistently. Time is also an important theoretical construct in understanding how distributed executions unfold. But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate.

Clock skew and clock drift rate: Computer clocks, like any others, tend not to be in perfect agreement. The instantaneous difference between the readings of any two clocks is called their **skew**. Also, the crystal-based clocks used in computers are, like any other clocks, subject to **clock drift**, which means that they count time at different rates, and so diverge. A **clock's drift rate** is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock. For ordinary clocks based on a quartz crystal this is about $\mathbf{10^{-6}}$ seconds/second, giving a difference of **1 second** every **1,000,000 seconds**, or **11.6 days**. The drift rate of 'high-precision' quartz clocks is about $\mathbf{10^{-7}}$ or $\mathbf{10^{-8}}$**.**

Coordinated Universal Time: Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in 1013. The output of these atomic clocks is used as the standard for elapsed real time, known as International Atomic Time. Since 1967, the standard second has been defined as 9,192,631,770 periods of transition between the two hyper-fine levels of the ground state of Caesium-133 (Cs^{133}). Coordinated Universal Time – abbreviated as UTC – is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. UTC signals are synchronized and broadcast regularly from land- based radio stations and satellites covering many parts of the world. For example, Satellite sources include the Global Positioning System (GPS).

Logical time and logical clocks: From the point of view of any single process, events are ordered uniquely by times shown on the local clock. However, as Lamport [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it.

The sequence of events within a single process $\mathbf{p_i}$ can be placed in a single, total ordering, which we denote by the relation $\rightarrow_i$ between the events. That is, $\mathbf{e} \rightarrow_i \mathbf{e'}$ if and only if the event **e** occurs before **e'** at $\mathbf{p_i}$.

We can use a scheme that is similar to physical causality but that applies in distributed systems to order some of the events that occur at different processes. This ordering is based on two simple and intuitively obvious points:

- If two events occurred at the same process $\mathbf{p_i}$ (i = 1 2 … N) then they occurred in the order in which $\mathbf{p_i}$ observes them – this is the order $\rightarrow_i$ that we defined above.
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these two relationships the happened-before relation. We can define the happened-before relation, denoted by →, as follows:

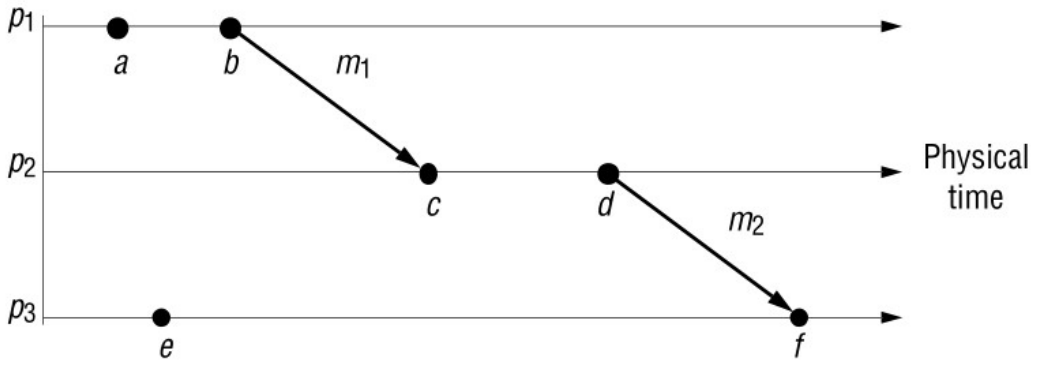
HB1: If ∃ process $\mathbf{p_i}$: $\mathbf{e} \rightarrow_i \mathbf{e'}$, then $\mathbf{e} \rightarrow \mathbf{e'}$

HB2: For any message m, **send(m) → receive(m)**

where send(m) is the event of sending the message, and receive(m) is the event of receiving it.

HB3: If **e**, **e'** and **e''** are events such that $\mathbf{e} \rightarrow \mathbf{e'}$ and $\mathbf{e'} \rightarrow \mathbf{e''}$ then $\mathbf{e} \rightarrow \mathbf{e''}$

Events occurring at three processes



It can be seen that $\mathbf{a} \rightarrow \mathbf{b}$, since the events occur in this order at process $\mathbf{p_1}$($\mathbf{a} \rightarrow_i \mathbf{b}$), and similarly $\mathbf{c} \rightarrow \mathbf{d}$. Furthermore, $\mathbf{b} \rightarrow \mathbf{c}$, since these events are the sending and reception of message $\mathbf{m_1}$, and similarly $\mathbf{d} \rightarrow \mathbf{f}$. Combining these relations, we may also say that, for example, $\mathbf{a} \rightarrow \mathbf{f}$.

We must also keep in mind that not all events are related by the relation →. For example, $\mathbf{a} \not\rightarrow \mathbf{e}$ and $\mathbf{e} \not\rightarrow \mathbf{a}$, since they occur at different processes, and there is no chain of messages intervening between them. We say that events such as **a** and **e** that are not ordered by → are concurrent and write this as **a||e**.

Lamport invented a simple mechanism by which the happened- before ordering can be captured numerically, called a logical clock. A Lamport logical clock is a monotonically increasing software

counter, whose value need bear no particular relationship to any physical clock. Each process $\mathbf{p_i}$ keeps its own logical clock, $\mathbf{L_i}$, which it uses to apply so-called Lamport timestamps to events.

We denote the timestamp of event e at $\mathbf{p_i}$ by $\mathbf{L_i(e)}$, and by **L(e)** we denote the timestamp of event e at whatever process it occurred at.

To capture the happened-before relation →, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

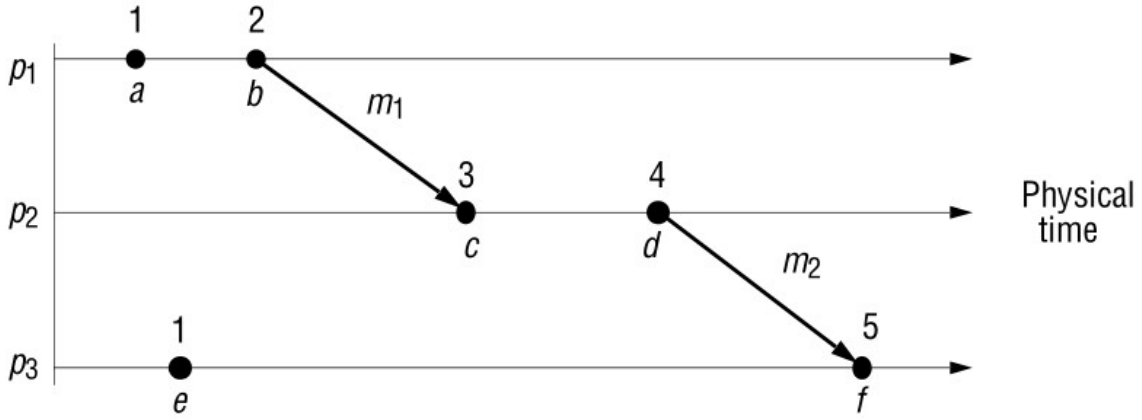
LC1: $\mathbf{L_i}$ is incremented before each event is issued at process $\mathbf{p_i}$: $\mathbf{L_i := L_i + 1}$.

LC2: (a) When a process $\mathbf{p_i}$ sends a message **m**, it piggybacks on **m** the value $\mathbf{t = L_i}$.

(b) On receiving **(m, t)**, a process $\mathbf{p_j}$ computes $\mathbf{L_j := max(L_j, t)}$ and then applies LC1 before time-stamping the event **receive(m)**.

Although we increment clocks by 1, we could have chosen any positive value. It can easily be shown, by induction on the length of any sequence of events relating two events **e** and **e′**, that **e → e′** implies **L(e) < L (e′)**.

Lamport timestamps for the events



Totally ordered logical clocks: Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set of events – that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur. If e is an event occurring at $\mathbf{p_i}$ with local timestamp $\mathbf{T_i}$, and **e′** is an event occurring at $\mathbf{p_j}$ with local timestamp $\mathbf{T_j}$, we define the global logical timestamps for these events to be $\mathbf{(T_i, i)}$ and $\mathbf{(T_j, j)}$, respectively. And we define $\mathbf{(T_i, I) < (T_j, j)}$ if and only if either $\mathbf{T_i < T_j}$, or $\mathbf{T_i = T_j}$ and **i < j**.

Vector clocks: Mattern and Fidge developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from **L(e) < L(e′)** we cannot conclude that **e → e′**. A vector clock for a system of **N** processes is an array of **N** integers. Each process keeps its own vector clock $\mathbf{V_i}$, which it uses to timestamp local events. Like Lamport timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

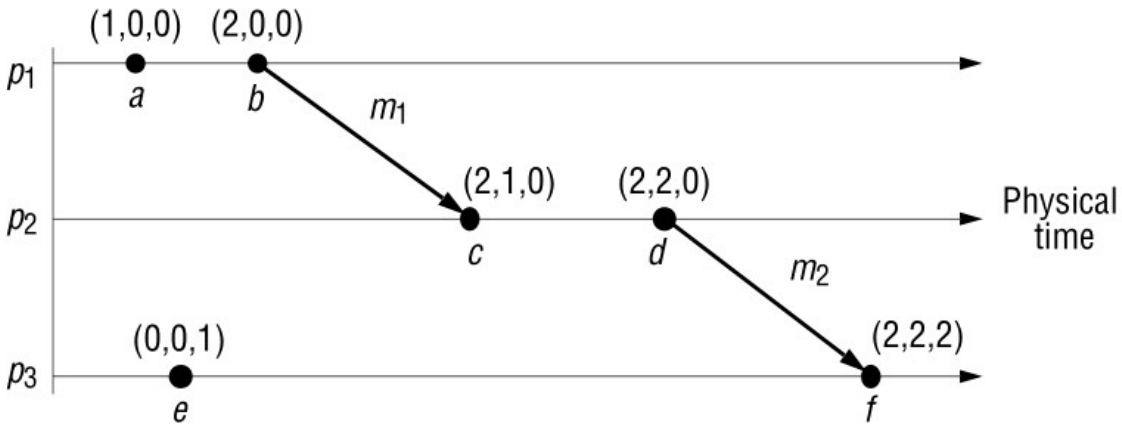
VC1: Initially, $\mathbf{V_i[j]} = 0$, for i, j = 1,2… N.

VC2: Just before $\mathbf{p_i}$ timestamps an event, it sets $\mathbf{V_i[i] := V_i[i] + 1}$.

VC3: $\mathbf{p_i}$ includes the value $t = V_i$ in every message it sends.

VC4: When $\mathbf{p_i}$ receives a timestamp t in a message, it sets $\mathbf{V_i[j] := max(V_i[j], t[j])}$, for j = 1,2...N. Taking the component-wise maximum of two vector timestamps in this way is known as a merge operation.

Vector timestamps for the events



We may compare vector timestamps as follows:

V = V′ iff V[j] = V′[j] for j = 1, 2… ,N

V ≤ V′ iff V[j] ≤ V′[j] for j = 1, 2… ,N

V < V′ iff V ≤ V′ ∧ V ≠ V′

Vector timestamps have the disadvantage, compared with Lamport timestamps, of taking up an amount of storage and message payload that is proportional to N, the number of processes.

Source Code:

```
import sys
import pprint
processList =[]
logicalClock = {}
TimeStamp = {}

def addProcess():
  pName = input("Enter Processes Name seperated by space ")
  processList = pName.split()
  for process in processList:
    logicalClock[process] = 0

def sendMessage(t):
  eName = input("Enter the Event which will receive the message ")
  pName = input("Enter the process on which this event will occur ")
  if t>logicalClock[pName]:
    logicalClock[pName] = t
  TimeStamp[eName] = logicalClock[pName] + 1
  logicalClock[pName] += 1

def addEvent():
  pName = input("Enter the Process for which you want to add an event ")
  eName = input("Enter Event Name ")
  eType = input("Enter the type of event(normal/message) ")
  if eType == "normal":
    TimeStamp[eName] = logicalClock[pName] + 1
    logicalClock[pName] += 1
  if eType == "message":
    TimeStamp[eName] = logicalClock[pName] + 1
    logicalClock[pName] += 1
    sendMessage(TimeStamp[eName])

def display():
  pprint.pprint(TimeStamp)

if __name__ == "__main__":
  addProcess()
  while(1):
    print("1.ADD EVENT\n2.DISPLAY TIMESTAMP\n3.EXIT")
    n = int(input("Enter your choice "))
    if n==1:
```

```
      addEvent()
    elif n==2:
      display()
    else:
      sys.exit("BYE")
```

Output:

```
Enter Processes Name seperated by space P1 P2 P3
1.ADD EVENT
2.DISPLAY TIMESTAMP
3.EXIT
Enter your choice 1
Enter the Process for which you want to add an event P1
Enter Event Name a
Enter the type of event(normal/message) normal
1.ADD EVENT
2.DISPLAY TIMESTAMP
3.EXIT
Enter your choice 1
Enter the Process for which you want to add an event P3
Enter Event Name e
Enter the type of event(normal/message) normal
1.ADD EVENT
2.DISPLAY TIMESTAMP
3.EXIT
Enter your choice 1
Enter the Process for which you want to add an event P1
Enter Event Name b
Enter the type of event(normal/message) message
Enter the Event which will receive the message c
Enter the process on which this event will occur P2
1.ADD EVENT
2.DISPLAY TIMESTAMP
3.EXIT
Enter your choice 2
{'a': 1, 'b': 2, 'c': 3, 'e': 1}
```