IT4883: Distributed Systems

Spring 2020

Synchronization - 2

School of Information and Communication Technology
Hanoi University of Science and Technology

Today…

- Last Session
  - Synchronization: Introduction
  - Synchronization: Clock Synchronization and Cristian's Algorithm

- Today's session
  - Synchronization
    - Clock Synchronization: Berkeley's Algorithm and NTP
    - Logical Clocks: Lamport's Clock, Vector Clocks

Where do We Stand in Synchronization Chapter?

Previous lecture

Today's lecture

Time Synchronization

- Physical Clock Synchronization (or, simply, Clock Synchronization)
  Here, actual time on the computers are synchronized
- Logical Clock Synchronization
  Computers are synchronized based on the relative ordering of events

Mutual Exclusion

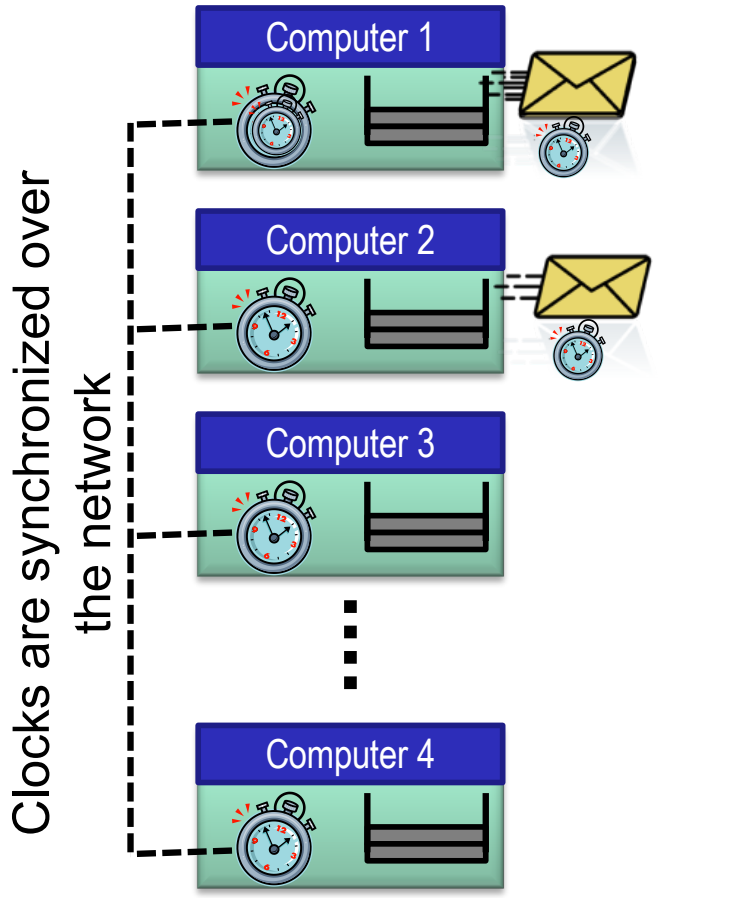
- How to coordinate between processes that access the same resource?

Election Algorithms

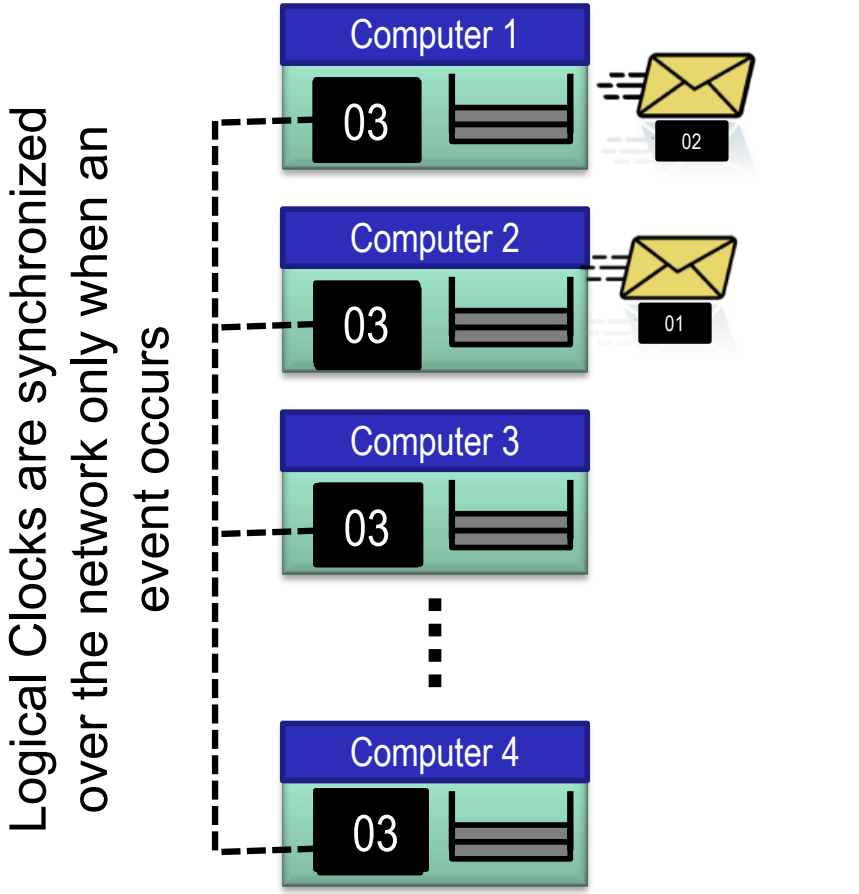
- Here, a group of entities elect one entity as the coordinator for solving a problem

Next lecture

Types of Time Synchronization



Clock-based Time Synchronization



Event-based Time Synchronization

Overview

Time Synchronization

- Clock Synchronization
- Logical Clock Synchronization

Mutual Exclusion

Election Algorithms

Clock Synchronization

Coordinated Universal Time

Tracking Time on a Computer

Clock Synchronization Algorithms

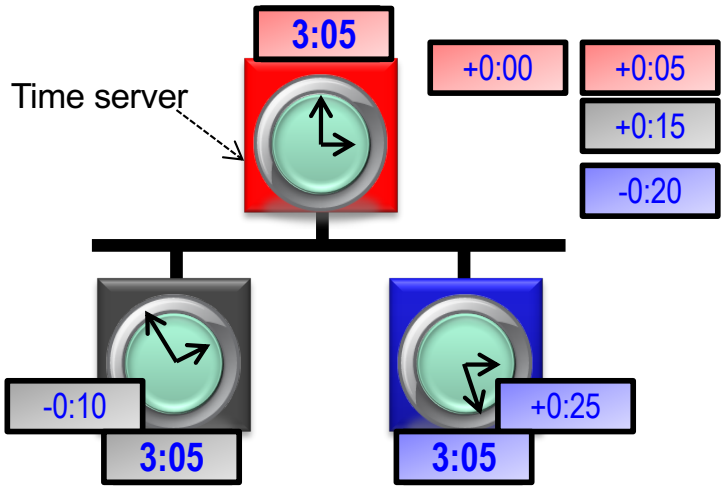
- Cristian's Algorithm
- Berkeley Algorithm
- Network Time Protocol

Berkeley Algorithm

+ Berkeley Algorithm is a distributed approach for time synchronization

Approach:

1. A time server periodically (approx. once in 4 minutes) sends its time to all the computers and polls them for the time difference
2. The computers compute the time difference and then reply
3. The server computes an average time difference for each computer
4. The server commands all the computers to update their time (by gradual time synchronization)



Berkeley Algorithm – Discussion

1. Assumption about packet transmission delays

- Berkeley's algorithm predicts network delay (similar to Cristian's algorithm)
- Hence, it is effective in intranets, and not accurate in wide-area networks

2. No UTC Receiver is necessary

- The clocks in the system synchronize by averaging all the computer's times

3. Time server failures can be masked

- If a time server fails, another computer can be elected as a time server

Clock Synchronization

Coordinated Universal Time

Tracking Time on a Computer

Clock Synchronization Algorithms

- Cristian's Algorithm
- Berkeley Algorithm
- Network Time Protocol

Network Time Protocol (NTP)

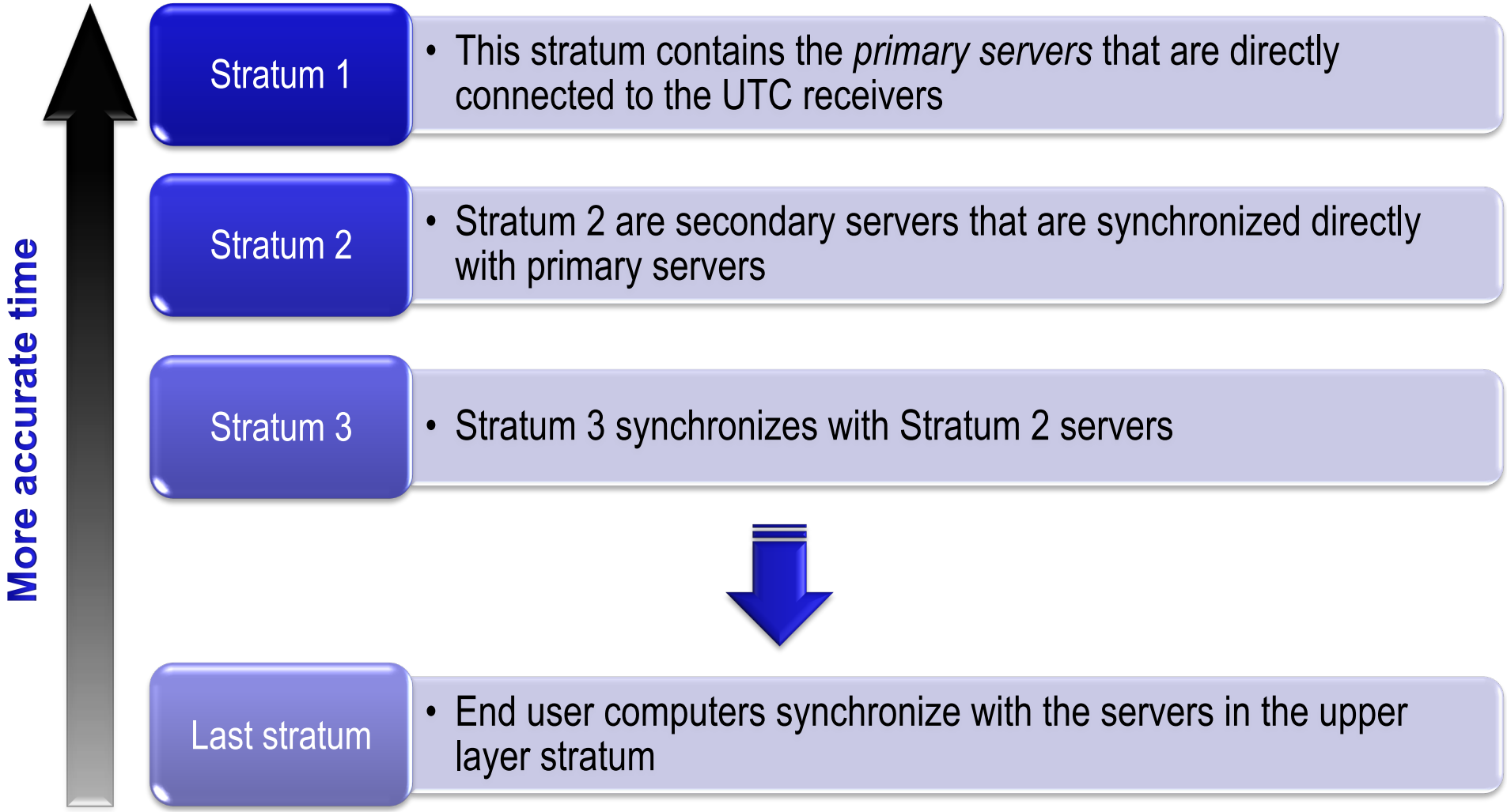
NTP defines an architecture for a time service and a protocol to distribute time information over the Internet

In NTP, servers are connected in a logical hierarchy called *synchronization subnet*

The levels of synchronization subnet is called *strata*

- Stratum 1 servers have most accurate time information (connected to a UTC receiver)
- Servers in each stratum act as time servers to the servers in the lower stratum

Hierarchical organization of NTP Servers



Operation of NTP Protocol

When a time server **A** contacts time server **B** for synchronization

- If **`stratum(A) <= stratum(B)`**, then **A** does not synchronize with B
- If **`stratum(A) > stratum(B)`**, then:

  Time server **A** synchronizes with **B**

  An algorithm similar to Cristian's algorithm is used to synchronize

  Time server **A** updates its stratum

  `stratum(A) = stratum(B) + 1`

Discussion of NTP Design

Accurate synchronization to UTC time

- NTP enables clients across the Internet to be synchronized accurately to the UTC
- Large and variable message delays are tolerated through statistical filtering of timing data from different servers

Scalability

- NTP servers are hierarchically organized to speed up synchronization, and to scale to a large number of clients and servers

Reliability and Fault-tolerance

- There are redundant time servers, and redundant paths between the time servers
- The architecture provides reliable service that can tolerate lengthy losses of connectivity
- A synchronization subnet can reconfigure as servers become unreachable. For example, if Stratum 1 server fails, then it can become a Stratum 2 secondary server

Security

- NTP protocol uses authentication to check of the timing message originated from the claimed trusted sources

Summary of Clock Synchronization

Physical clocks on computers are not accurate

Clock synchronization algorithms provide mechanisms to synchronize clocks on networked computers in a DS

- Computers on a local network use various algorithms for synchronization
  - Some algorithms (e.g, Cristian's algorithm) synchronize time with by contacting centralized time servers
  - Some algorithms (e.g., Berkeley algorithm) synchronize in a distributed manner by exchanging the time information on various computers
- NTP provides architecture and protocol for time synchronization over wide-area networks such as Internet

Overview

Time Synchronization

- Clock Synchronization
- Logical Clock Synchronization

Mutual Exclusion

Election Algorithms

Why Logical Clocks?

Lamport (in 1978) showed that:

- Clock synchronization is not necessary in all scenarios

  If two processes do not interact, it is not necessary that their clocks are synchronized

- Many times, it is sufficient if processes agree on the order in which the events has occurred in a DS

  For example, for a distributed *make* utility, it is sufficient to know if an input file was modified *before* or *after* its object file

Logical Clocks

Logical clocks are used to define an order of events without measuring the physical time at which the events occurred

We will study two types of logical clocks

1. Lamport's Logical Clock (or simply, Lamport's Clock)
2. Vector Clock

Logical Clocks

We will study two types of logical clocks

1. Lamport's Clock
2. Vector Clock

Lamport's Logical Clock

Lamport advocated maintaining logical clocks at the processes to keep track of the order of events

To synchronize logical clocks, Lamport defined a relation called "**happened-before**"

The expression **a→b** (read as "**a** happened before **b**") means that all entities in a DS agree that event **a** occurred before event **b**

Happened-before Relation

The **happened-before** relation can be observed directly in two situations:

1. If **a** and **b** are events in the same process, and **a** occurs before **b**, then **a→b** is true
2. If **a** is an event of message **m** being sent by a process, and **b** is the event of the message **m** being received by another process, the **a→b** is true.

The **happened-before** relation is transitive

- If **a→b** and **b→c**, then **a→c**

Time values in Logical Clocks

For every event **a**, assign a logical *time value* **C(a)** on which all processes agree

Time value for events have the property that

- If **a→b**, then **C(a)< C(b)**

Properties of Logical Clock

From **happened-before** relation, we can infer that:

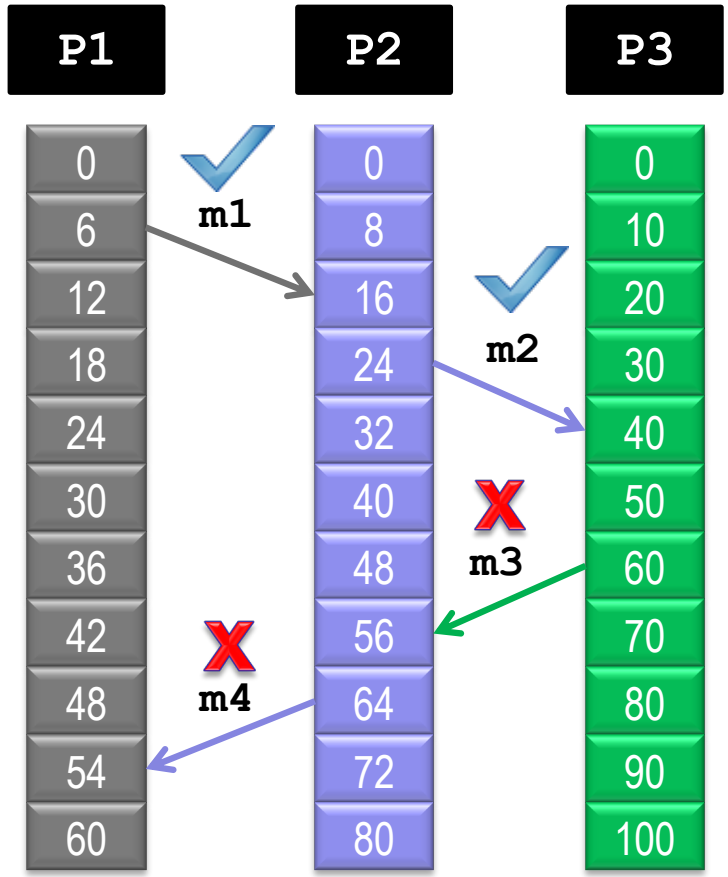
- If two events **a** and **b** occur within the same process and **a→b**, then assign **C(a)** and **C(b)** such that **C(a) < C(b)**
- If **a** is the event of sending the message **m** from one process, and **b** is the event of receiving the message **m**, then
  the time values **C(a)** and **C(b)** are assigned such that all processes agree that **C(a) < C(b)**
- The clock time **C** must always go forward (increasing), and never backward (decreasing)

Synchronizing Logical Clocks

Three processes **P1**, **P2** and **P3** running at different rates

If the processes communicate between each other, there might be discrepancies in agreeing on the event ordering

- Ordering of sending and receiving messages **m1** and **m2** are correct
- However, **m3** and **m4** violate the happens-before relationship



Lamport's Clock Algorithm

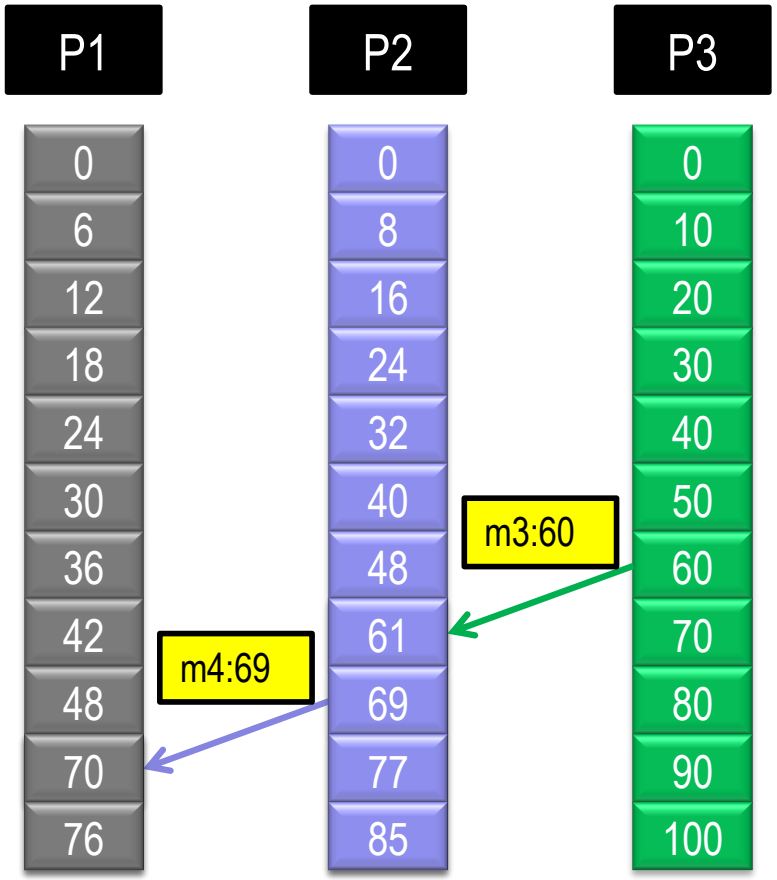
When a message is being sent:

Each message carries a **`timestamp`** according to the sender's logical clock

When a message is received:

If the receiver logical clock is less than message sending time in the packet, then adjust the receiver's clock such that

```
currentTime = timestamp + 1
```



Logical Clock Without a Physical Clock

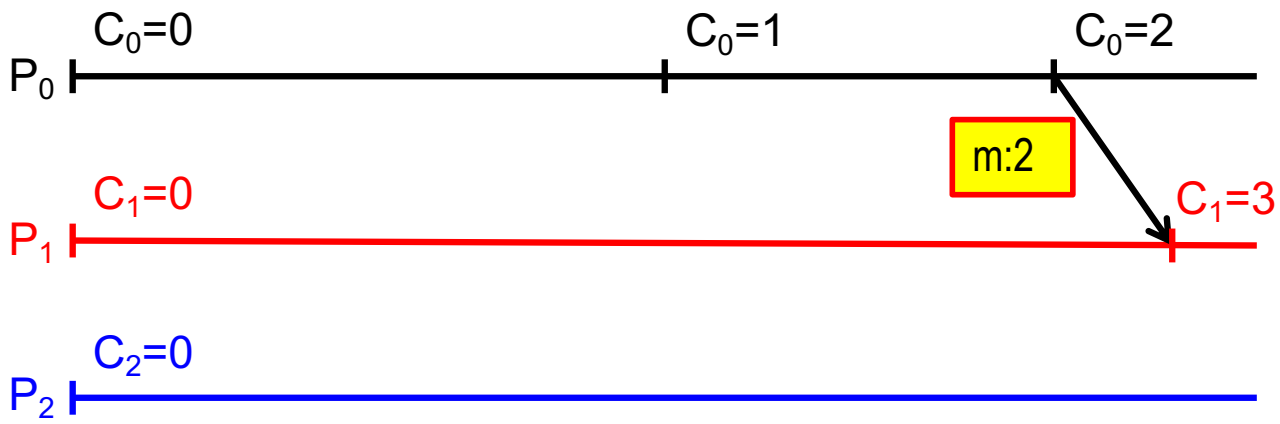
Previous examples assumed that there is a physical clock at each computer (probably running at different rates)

How to attach a time value to an event when there is no global clock?

Implementation of Lamport's Clock

Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1. For any two successive events that take place within P_i, C_i is incremented by 1
2. Each time a message m is sent by process P_i , the message receives a timestamp $ts(m) = C_i$
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\mathbf{max(C_j, ts(m)) + 1}$



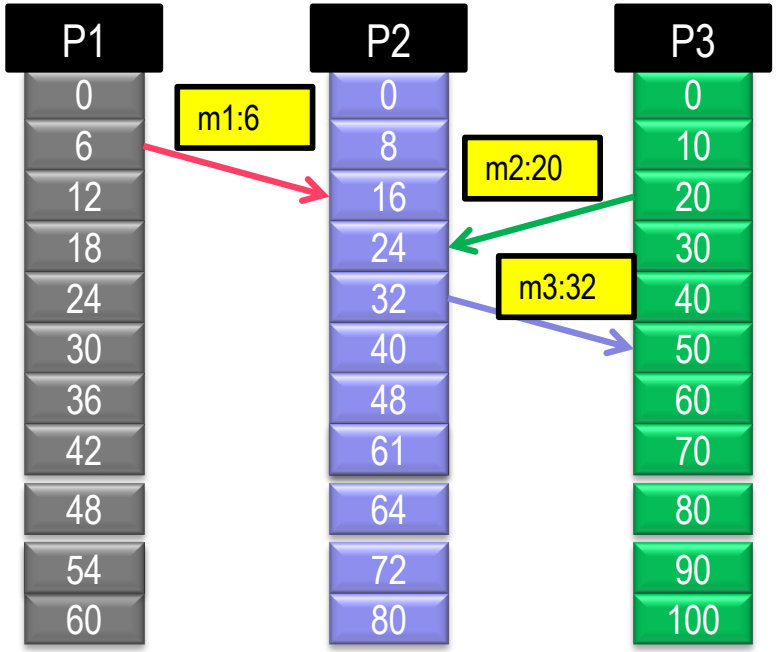
Limitation of Lamport's Clock

Lamport's Clock ensures that if **a→b**, then **C(a) < C(b)**

However, it does not say anything about any two events **a** and **b** by comparing their time values

- For any two events **a** and **b**, **C(a) < C(b)** does not mean that **a→b**

Example:



Compare m1 and m3

P2 can infer that **m1→m3**

Compare m1 and m2

P2 **cannot** infer that **m1→m2** or **m2→m1**

Summary of Lamport's Clock

Lamport advocated using logical clocks

- Processes synchronize based on their time values of the logical clock rather than the absolute time on the physical time

Which applications in DS need logical clocks?

- Applications with provable ordering of events

  Perfect physical clock synchronization is hard to achieve in practice. Hence we cannot provably order the events
- Applications with rare events

  Events are rarely generated, and physical clock synchronization overhead is not justified

However, Lamport's clock cannot guarantee perfect ordering of events by just observing the time values of two arbitrary events

Logical Clocks

We will study two types of logical clocks

1. Lamport's Clock
2. Vector Clocks

Vector Clock Update Algorithm

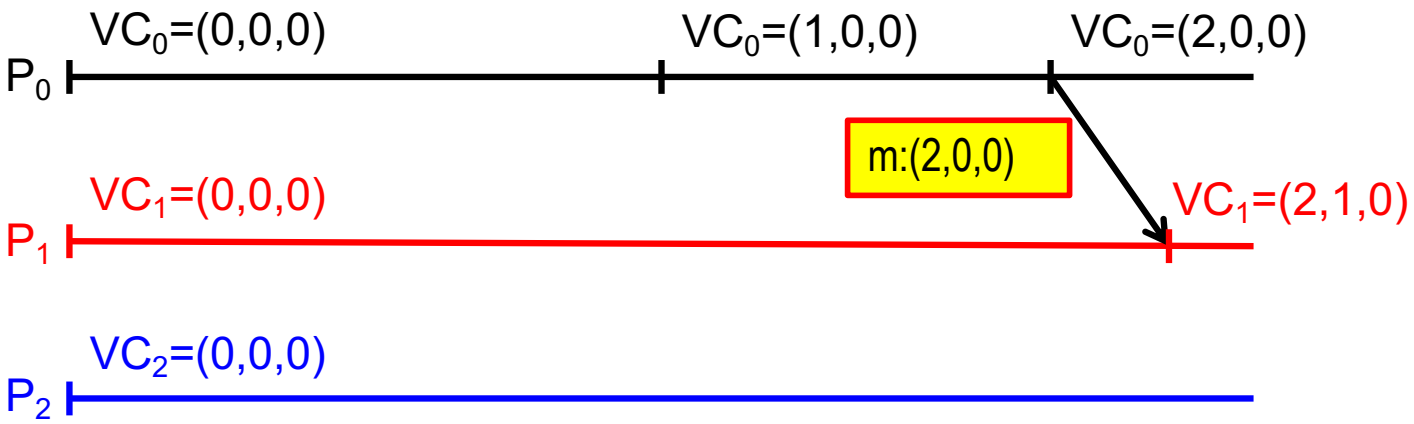
Whenever there is a new event at P_i, increment $VC_i[i]$

When a process P_i sends a message **m** to P_j:

- Increment $VC_i[i]$
- Set **m**'s timestamp **ts(m)** to the vector VC_i

When message m is received process P_j :

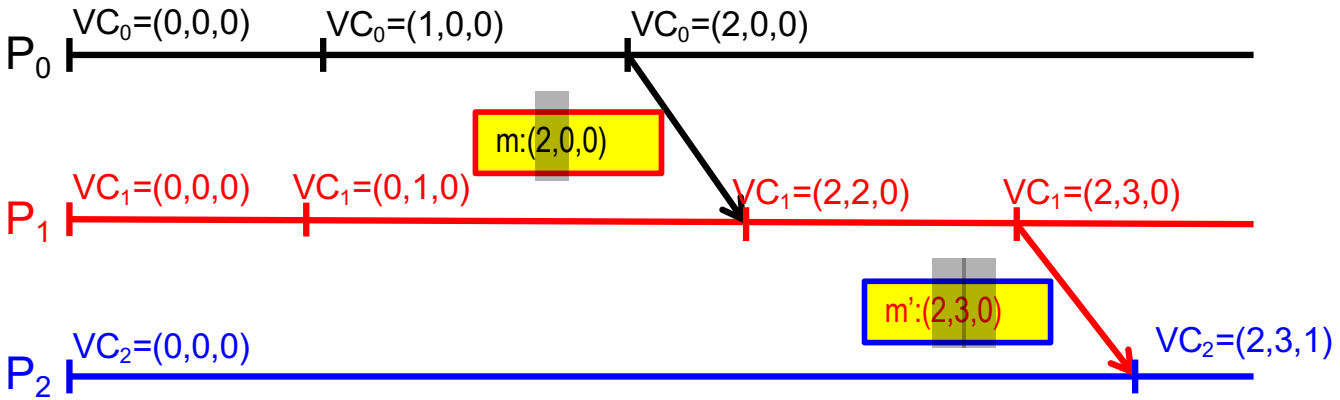
- $VC_j[k] = \max(VC_j[k], ts(m)[k])$; (for all **k**)
- Increment $VC_j[j]$



Inferring Events with Vector Clocks

Let a process P_i send a message **m** to P_j with timestamp **ts(m)**, then:

- P_j knows the number of events at the sender P_i that causally precede **m**
  ts(m)[i] – 1 denotes the number of events at P_i
- P_j also knows the minimum number of events at other processes P_k that causally precede **m**
  ts(m)[k] – 1 denotes the minimum number of events at P_k



Summary – Logical Clocks

Logical Clocks are employed when processes have to agree on relative ordering of events, but not necessarily actual time of events

Two types of Logical Clocks

- Lamport's Logical Clocks
  Supports relative ordering of events across different processes by using `happen-before` relationship

- Vector Clocks
  Supports causal ordering of events

Next Class

Mutual Exclusion

- How to coordinate between processes that access the same resource?

Election Algorithms

- Here, a group of entities elect one entity as the coordinator for solving a problem

References

http://en.wikipedia.org/wiki/Causality