分布式系统

05-分布式系统的同步机制 Synchronization in Distributed Systems

Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

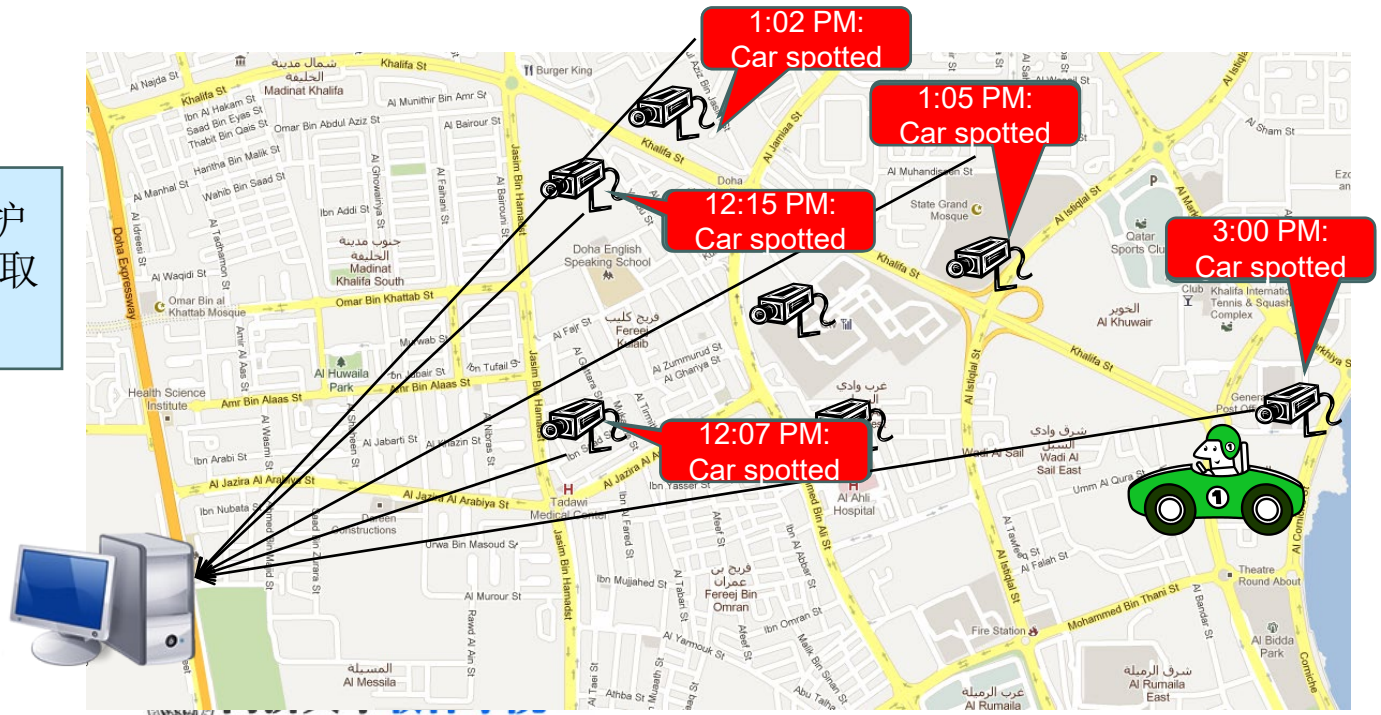
同步Synchronization

- Until now, we have looked at:
  - how entities communicate with each other
- In addition to the above requirements, entities in DS often have to *cooperate* and *synchronize* to solve the problem correctly
  - e.g., In a distributed file system, processes have to synchronize and cooperate, such that two processes are not allowed to write to the same part of a file

同步的应用场景之一

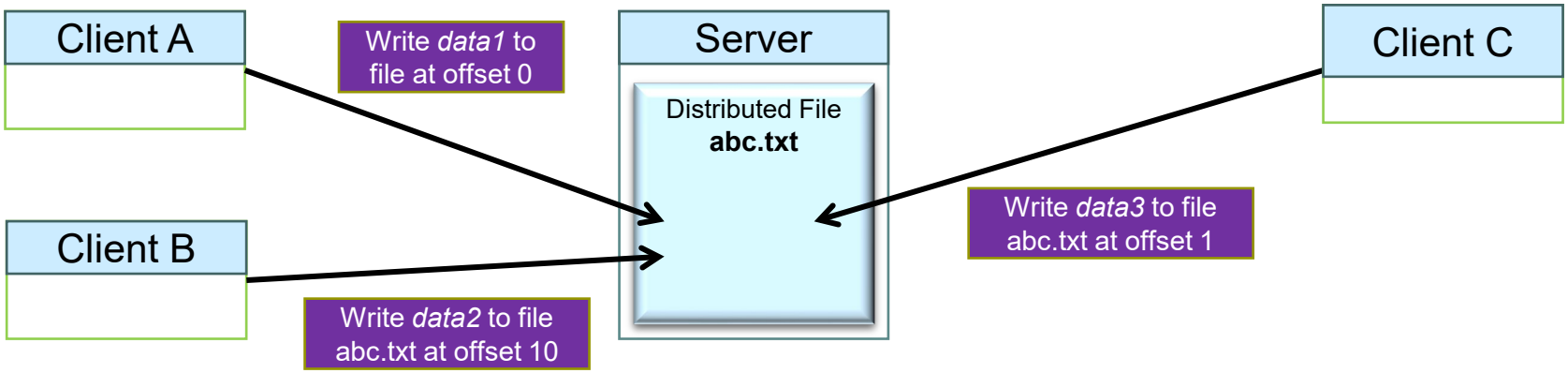
- 城市应急系统的车辆追踪: 使用分布式相机传感器网络
  - 目标: 追踪可疑车辆
  - 相机传感器网络部署到整个城市范围
  - 每个相机传感器检测**车辆信息**, 并向中心服务器报告检测到车辆的**时间信息**
  - 服务器追踪可疑策略的运动踪迹

如果传感器节点没有维护统一的时间信息, 很难获取准确的车辆运动踪迹



同步的应用场景之二

- 分布式文件系统的写操作



如果分布式的客户端在写文件的过程中不考虑写操作的同步机制，该文件的数据内容损坏。

同步应用的分类

使用同步/协作的缘由	实体需要对事件发生的时间顺序性达成一致	实体需要对分享公共资源
举例	相机传感网络的车辆追踪，分布式电子商务系统的商业交易操作	分布式文件系统的读写操作
实体的设计需求	Entities should have a common understanding of time across different computers	Entities should coordinate and agree on when and how to access resources
主题	Time Synchronization	Mutual Exclusion

同步机制的课程结构

Today's lecture

- Time Synchronization
  - Physical Clock Synchronization (or, simply, Clock Synchronization)
    - Here, actual time on computers are synchronized
  - Logical Clock Synchronization
    - Computers are synchronized based on 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 two lectures

同步机制的课程结构

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
- Election Algorithms

时钟同步Clock Synchronization

- Clock Synchronization is a mechanism to synchronize the time of all the computers in a DS

- We will study
  - Coordinated Universal Time
  - Tracking Time on a Computer
  - Clock Synchronization Algorithms
    - Cristian's Algorithm
    - Berkeley Algorithm
    - Network Time Protocol

时钟同步Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Network Time Protocol
  - Berkeley Algorithm

Coordinated Universal Time (UTC)

- All the computers are generally synchronized to a standard time called Coordinated Universal Time (UTC)
  - UTC is the primary time standard by which the world regulates clocks and time
- UTC is broadcasted via the satellites
  - UTC broadcasting service provides an accuracy of 0.5 msec
- Computer servers and online services with **UTC receivers** can be synchronized by satellite broadcasts (e.g., GPS).
  - Many popular synchronization protocols in distributed systems use UTC as a reference time to synchronize clocks of computers

The most accurate physical clocks use atomic oscillators whose drift rate is about one part in 10^{13}.

Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

Tracking Time on a Computer

- How does a computer keep track of its time?
  - Each computer has a hardware *timer*
    - The timer causes an interrupt 'H' times a second
  - The interrupt handler adds 1 to its Software Clock (C)
- Issues with clocks on a computer
  - In practice, the hardware timer is imprecise
    - It does not interrupt 'H' times a second due to material imperfections of the hardware and temperature variations
    - The computer counts the time slower or faster than actual time
  - Loosely speaking, Clock Skew is the skew between:
    - the computer clock and the actual time (e.g., UTC)

The operating system reads the node's hardware clock value, $Hi(t)$, scales it and adds an offset so as to produce a software clock $Ci(t) = \alpha Hi(t) + \beta$ that approximately measures real, physical time t for process pi.

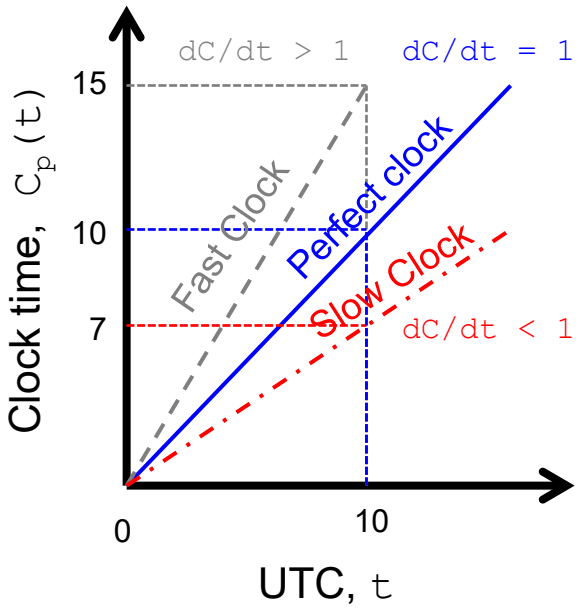
Clock Skew

- When the UTC time is `t`, let the clock on the computer have a time `C(t)`
- Three types of clocks are possible
  - Perfect clock:
    - The timer ticks 'H' interrupts a second

      `dC/dt = 1`
  - Fast clock:
    - The timer ticks more than 'H' interrupts a second

      `dC/dt > 1`
  - Slow clock:
    - The timer ticks less than 'H' interrupts a second

      `dC/dt < 1`



Clock Skew (cont'd)

- Frequency of the clock is defined as the ratio of the number of seconds counted by the software clock for every UTC second

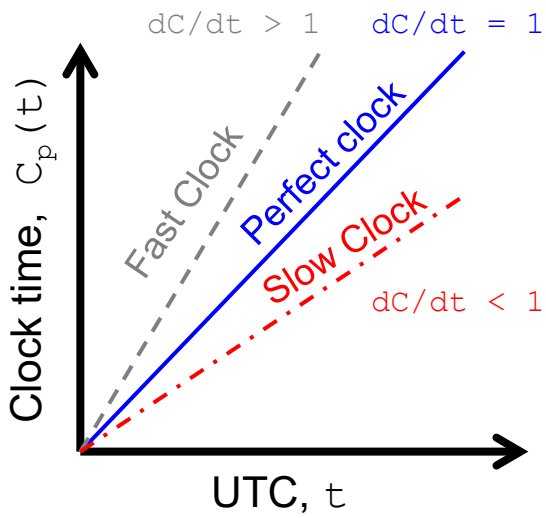
$$\text{Frequency} = dC/dt$$

- Skew of the clock is defined as the extent to which the frequency differs from that of a perfect clock

$$\text{Skew} = dC/dt - 1$$

- Hence,

$$Skew \begin{cases} > 0 & \text{for a fast clock} \\ = 0 & \text{for a perfect clock} \\ < 1 & \text{for a slow clock} \end{cases}$$



Maximum Drift Rate of a Clock

- The manufacturer of the timer specifies the upper and the lower bound that the clock skew may fluctuate. This value is known as *maximum drift rate* (ρ)

$$1 - \rho <= dC/dt <= 1 + \rho$$

- How far can two clocks drift apart?
  - If two clocks were synchronized Δt seconds before to UTC, then the two clocks can be as much as $2\rho\Delta t$ seconds apart

- Guaranteeing maximum drift between computers in a DS
  - If maximum drift permissible in a DS is δ seconds, then clocks of every computer has to resynchronize at least $\delta/2\rho$ seconds

$$2\rho\Delta t = \delta \rightarrow \Delta t = \delta/2\rho$$

Clock Synchronization

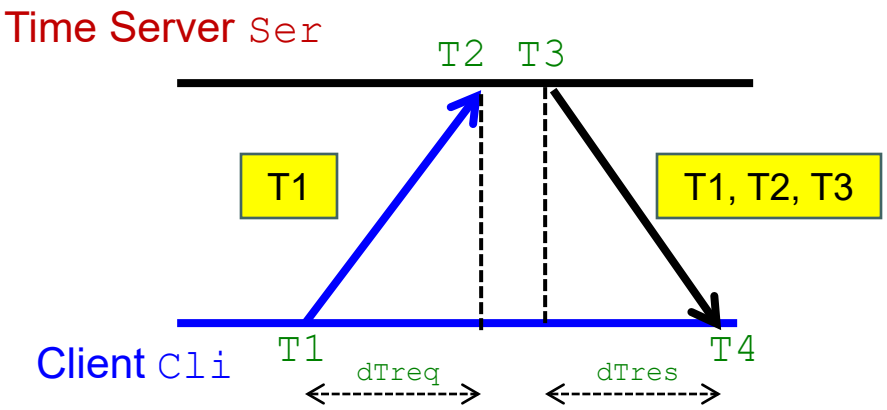
- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

Cristian's Algorithm

- Flaviu Cristian (in 1989) provided an algorithm to synchronize networked computers with a time server
- The basic idea:
  - Identify a network time server that has an accurate source for time (e.g., the time server has a UTC receiver)
  - All the clients contact the network time server for synchronization
- However, the network delays (incurred while the client contacts the time server) will outdate the reported time
  - The algorithm estimates the network delays and compensates for it

Cristian' Algorithm – Approach

- Client Cli sends a request to Time Server Ser, time stamped its local clock time T1
- Ser will record the time of receipt T2 according to its local clock
  - dTreq is network delay for request transmission



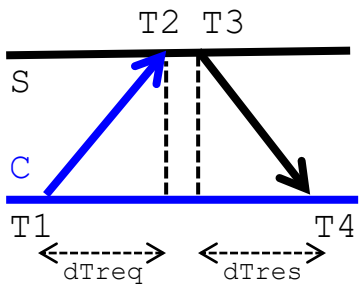
- Ser replies to Cli at its local time T3, piggybacking T1 and T2
- Cli receives the reply at its local time T4
  - dTres is the network delay for response transmission
- Now Cli has the information T1, T2, T3 and T4
- Assuming that the transmission delay from Cli→Ser and Ser→Cli are the same

T2-T1 ≈ T4-T3

Christian' Algorithm – Synchronizing Client Time

- Client C estimates its offset θ relative to Time Server S

$$\theta = T3 + dTres - T4$$
$$= T3 + ((T2-T1)+(T4-T3))/2 - T4$$
$$= ((T2-T1)+(T3-T4))/2$$



- If $\theta > 0$ or $\theta < 0$, then the client time should be incremented or decremented by θ seconds: t + θ, where t is the time value in S.

Gradual Time Synchronization at the client

- Instead of changing the time drastically by θ seconds, typically the time is gradually synchronized
  - The software clock is updated at a lesser/greater rate whenever timer interrupts

Note: Setting clock backward (say, if $\theta < 0$) is not allowed in a DS since decrementing a clock at any computer has adverse effects on several applications (e.g., *make* program)

Cristian's Algorithm – Discussion

1. Assumption about packet transmission delays

- Cristian's algorithm assumes that the round-trip times for messages exchanged over the network is reasonably short
- The algorithm assumes that the delay for the request and response are equal

- Will the trend of increasing Internet traffic decrease the accuracy of the algorithm?
- Can the algorithm handle delay asymmetry that is prevalent in the Internet?
- Can the clients be mobile entities with intermittent connectivity?

Cristian's algorithm is intended for synchronizing computers within intranets

2. A probabilistic approach for calculating delays

- There is no tight bound on the maximum drift between clocks of computers

3. Time server failure or faulty server clock

- Faulty clock on the time server leads to inaccurate clocks in the entire DS
- Failure of the time server will render synchronization impossible

Summary

- 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

Next ...

- Remaining topics in Time Synchronization
  - Clock Synchronization
    - Berkeley algorithm, Network Time Protocol
  - Logical Clock Synchronization
    - Computers are synchronized based on relative ordering of events
- Mutual Exclusion
  - How to coordinate between processes that access the same resource?

Where do We Stand in Synchronization Chapter?

Previous lecture

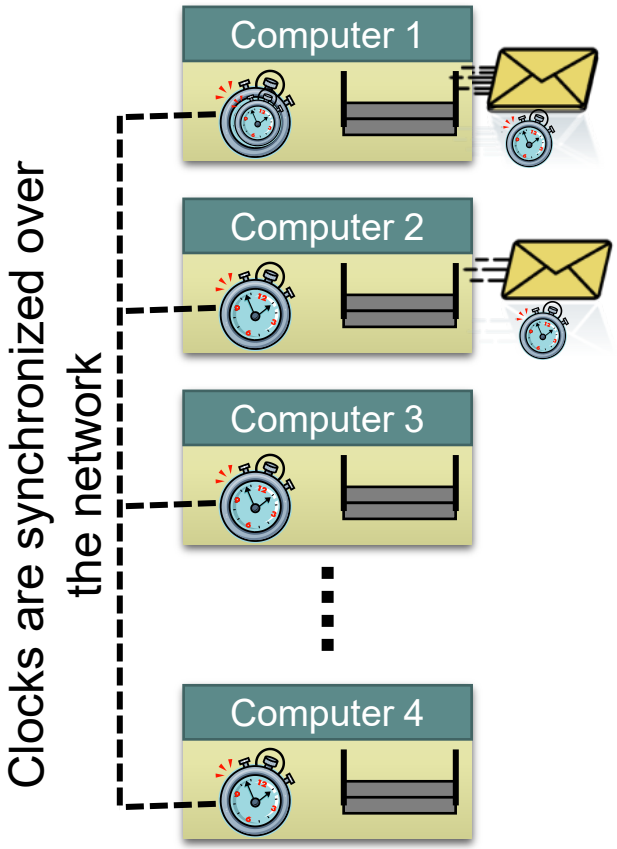
- Time Synchronization
  - Physical Clock Synchronization (or, simply, Clock Synchronization)
    - Here, actual time on the computers are synchronized
  - Logical Clock Synchronization
    - Computers are synchronized based on the relative ordering of events

Today's lecture

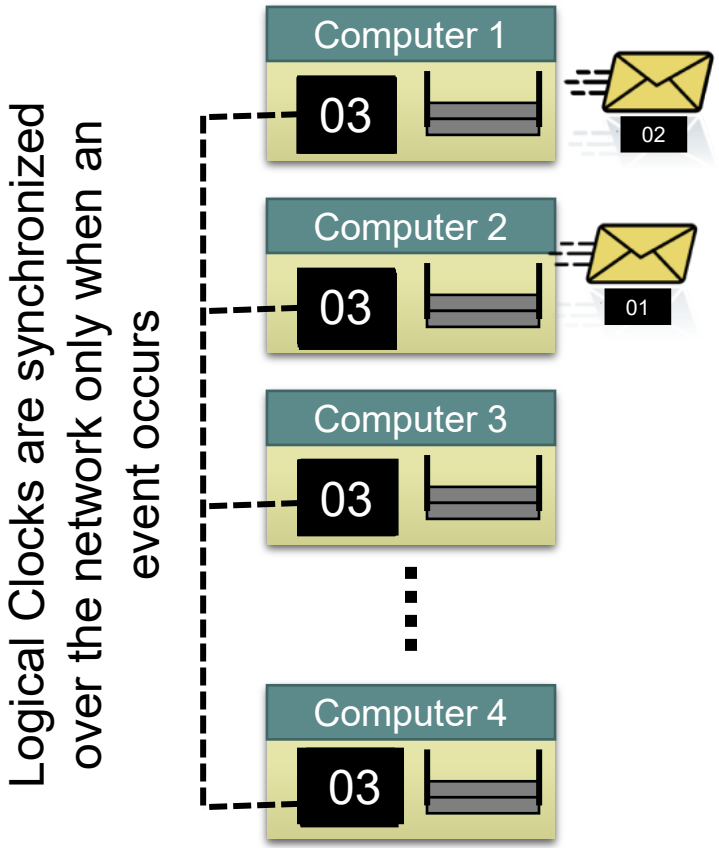
- Mutual Exclusion
  - How to coordinate between processes that access the same resource?
- Election Algorithms
  - Here, a group of entities elect one entity as the coordinator for solving a problem

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

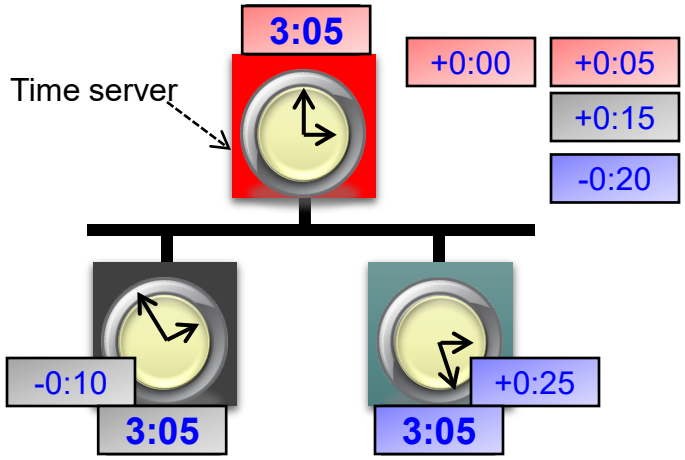
Clock Synchronization

- 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. Decreases the effect of faulty clocks

- Fault-tolerant averaging, where outlier clocks are ignored, can be easily performed in Berkeley Algorithm

4. Time server failures can be masked

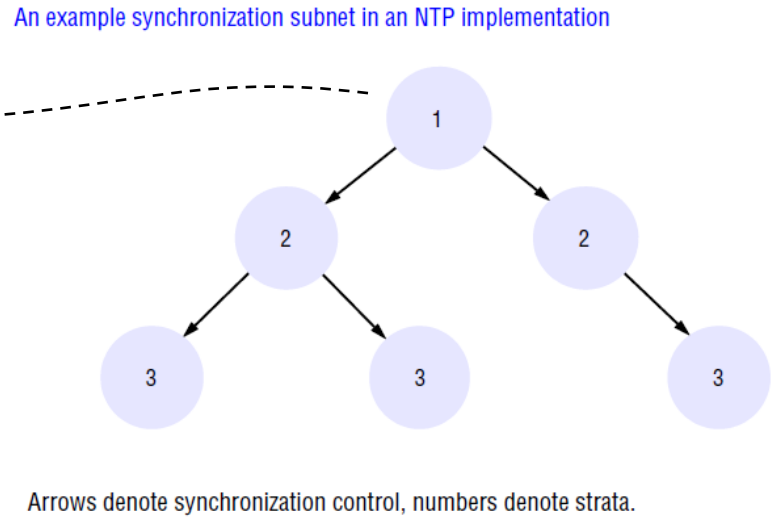
- If a time server fails, another computer can be elected as a time server

Clock Synchronization

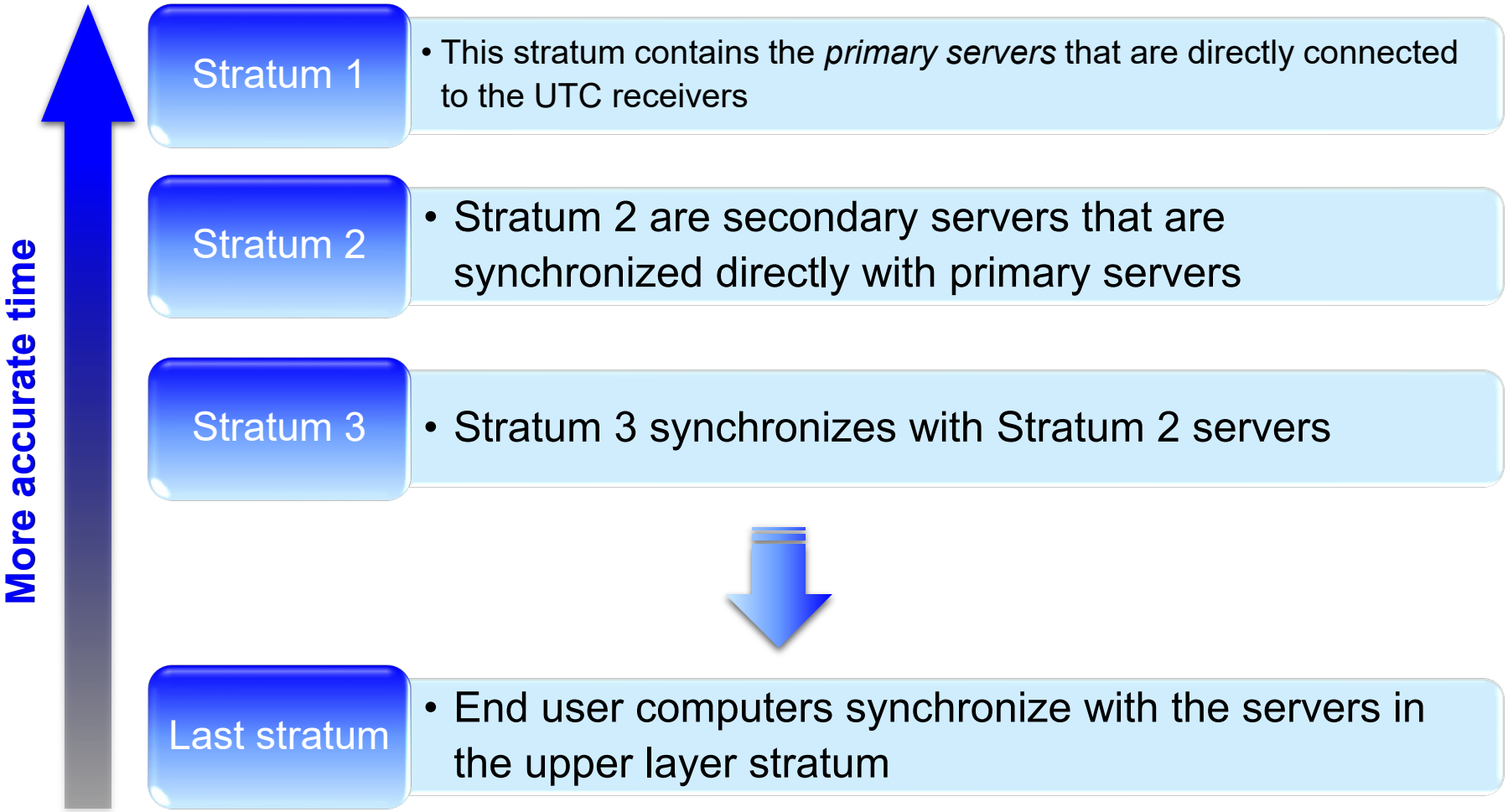
- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

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. However, larger statistical samples are taken before updating the clock
    - 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

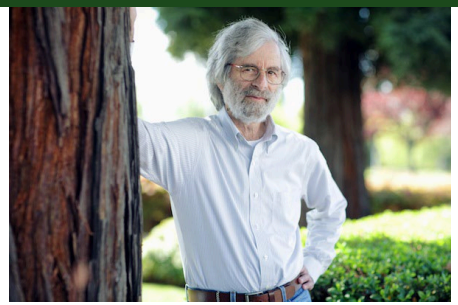
- Physical clocks on computers are not accurate
- Clock synchronization algorithms provide mechanisms to synchronize clocks on networked computers in a DS
  - Computers on a local network use various algorithms for synchronization
    - Some algorithms (e.g, Cristian's algorithm) synchronize time with 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? 逻辑时钟

- 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



原文名	Leslie Lamport
出生	1941年2月7日, 79岁 美国纽约市
母校	麻省理工学院（学士） 布兰戴斯大学（博士）
知名于	LaTeX Sequential consistency Atomic Register Hierarchy Lamport面包店算法 拜占庭将军问题 Paxos算法
奖项	Dijkstra Prize ('00、 ' 05) 约翰·冯诺依曼奖 ' 08 图灵奖 '13
网站	www.lamport.org
	科学生涯
研究领域	计算机科学
机构	微软研究院 康柏电脑 DEC SRI International

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 向量时钟

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 (可传递性)
  - If **a→b** and **b→c**, then **a→c**

Time values in Logical Clocks

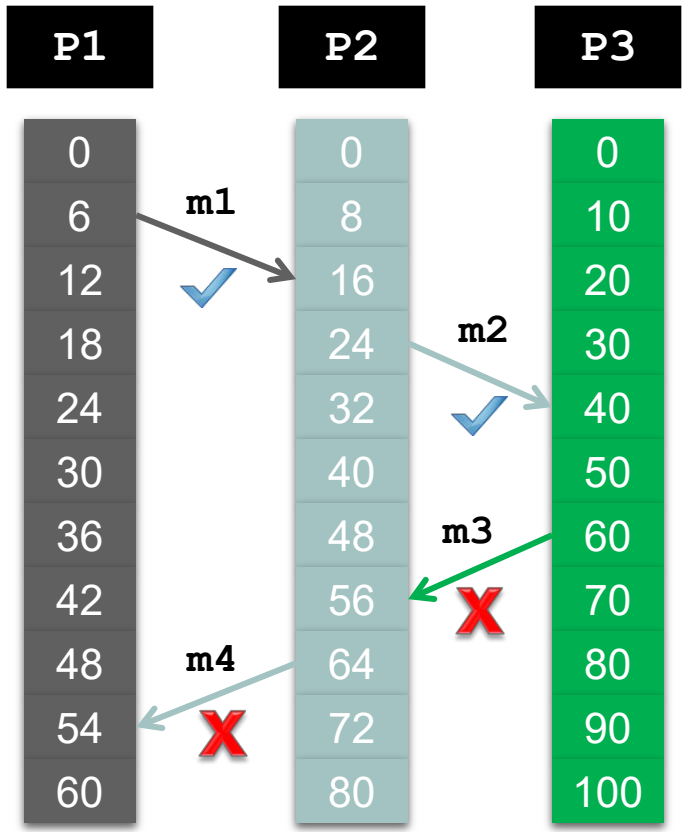
- For every event **a**, assign a logical *time value* **C(a)** on which all processes agree
- Time value for events have the property that
  - If **a→b**, then **C(a)< C(b)**

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

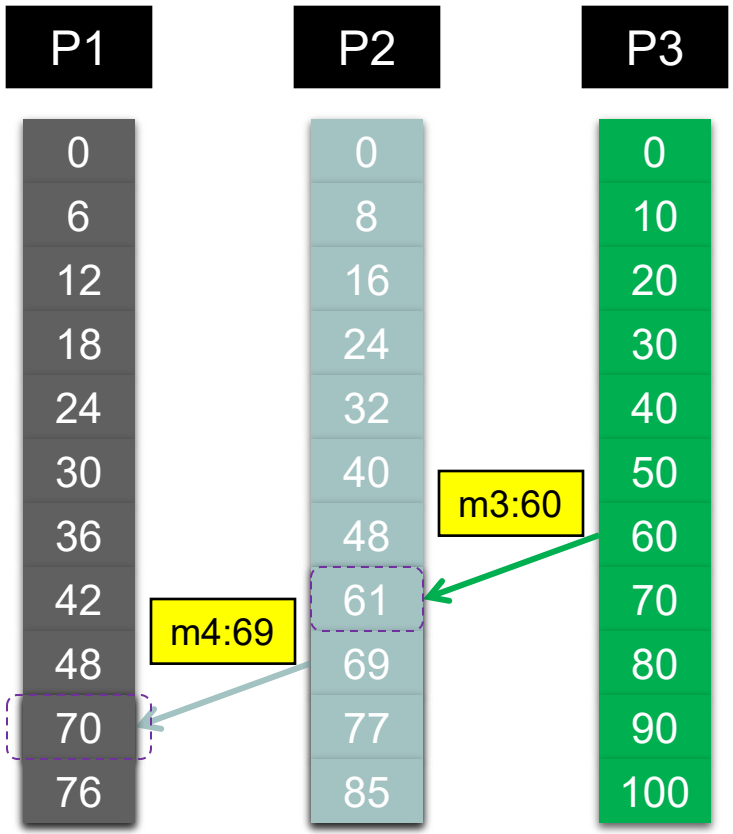
When a message is being sent:

- 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
```

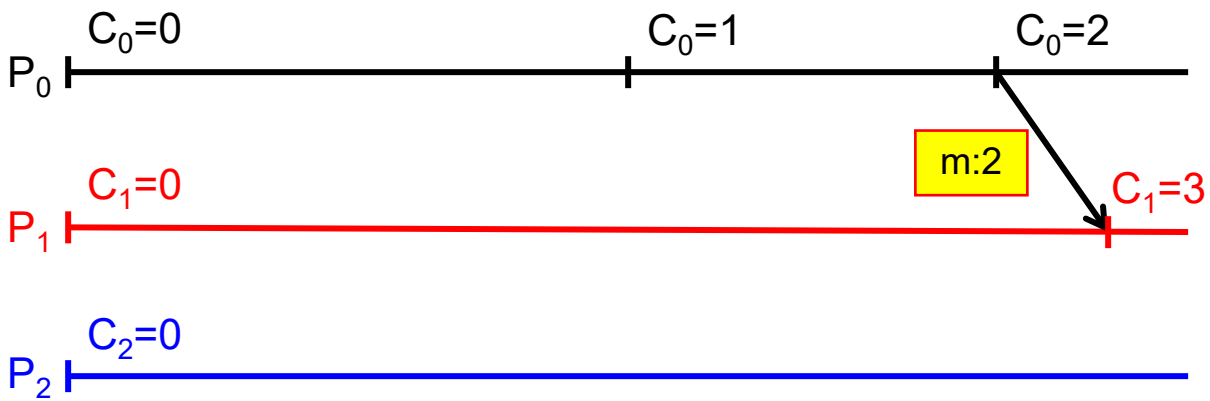


Logical Clock Without a Physical Clock

- 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?

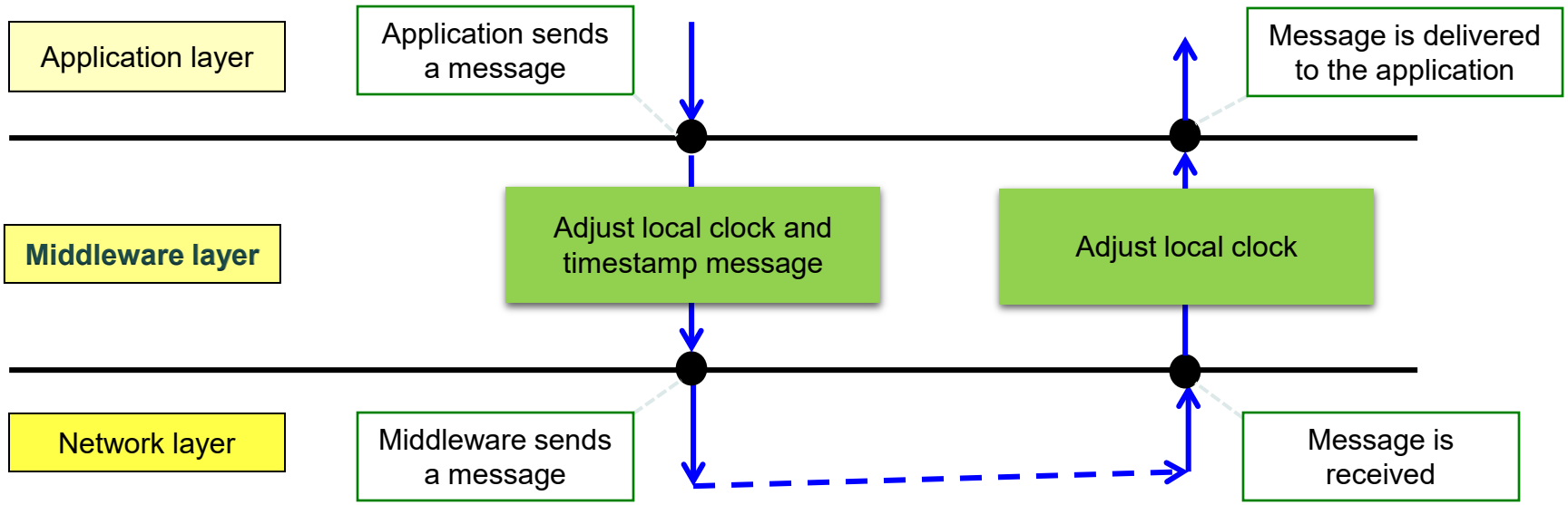
Implementation of Lamport's Clock

- 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 **$\max(C_j, ts(m)) + 1$**



Placement of Logical Clock

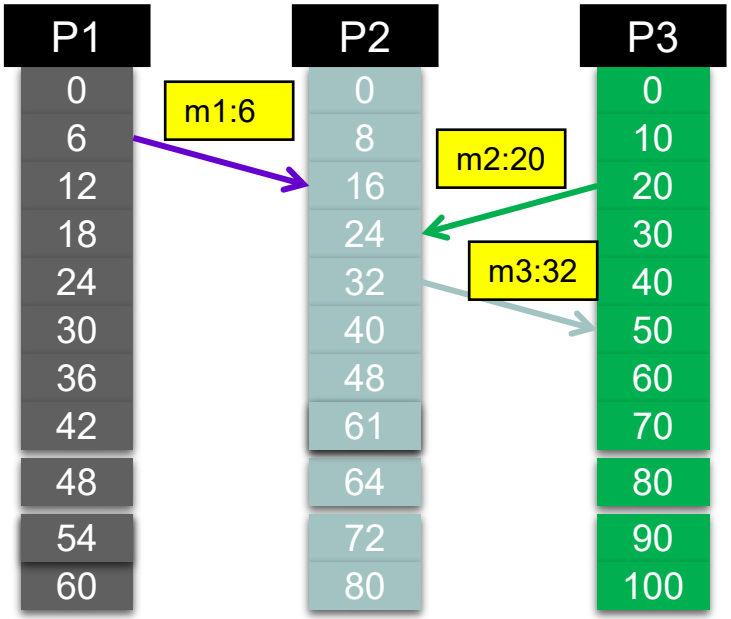
- In a computer, several processes use Logical Clocks
  - Similar to how several processes on a computer use one physical clock
- Instead of each process maintaining its own Logical Clock, Logical Clocks can be implemented as a **middleware** for time service



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
  - Lamport's Clock
  - Vector Clocks

Vector Clocks

- Vector Clocks was proposed to overcome the limitation of Lamport's clock: the fact that $C(a)<C(b)$ does not mean that $a \rightarrow b$
  - The property of inferring that **a** occurred before **b** is called as causality property
- A Vector clock for a system of N processes is an array of N integers
- Every process P_i stores its own vector clock VC_i
  - Lamport's time value for events are stored in VC_i
  - $VC_i(a)$ is assigned to an event a
- If $VC_i(a) < VC_i(b)$, then we can infer that $a \rightarrow b$

Updating Vector Clocks

- Vector clocks are constructed by the following two properties:
  1. $VC_i[i]$ is the number of events that have occurred at process P_i so far
     - $VC_i[i]$ is the local logical clock at process P_i

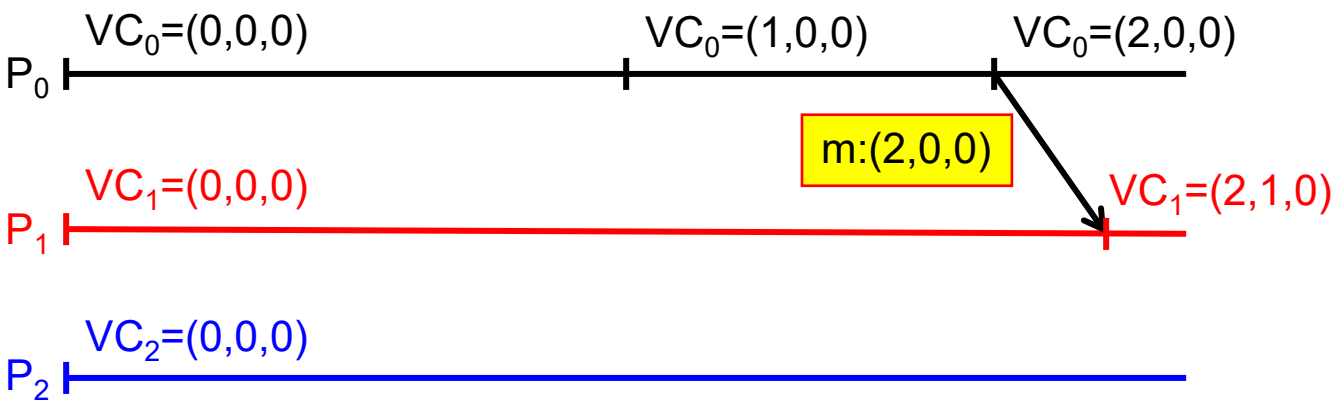
→ Increment VC_i whenever a new event occurs

  2. If $VC_i[j]=k$, then P_i knows that k events have occurred at P_j
     - $VC_i[j]$ is P_i's knowledge of local time at P_j

→ Pass VC_j along with the message

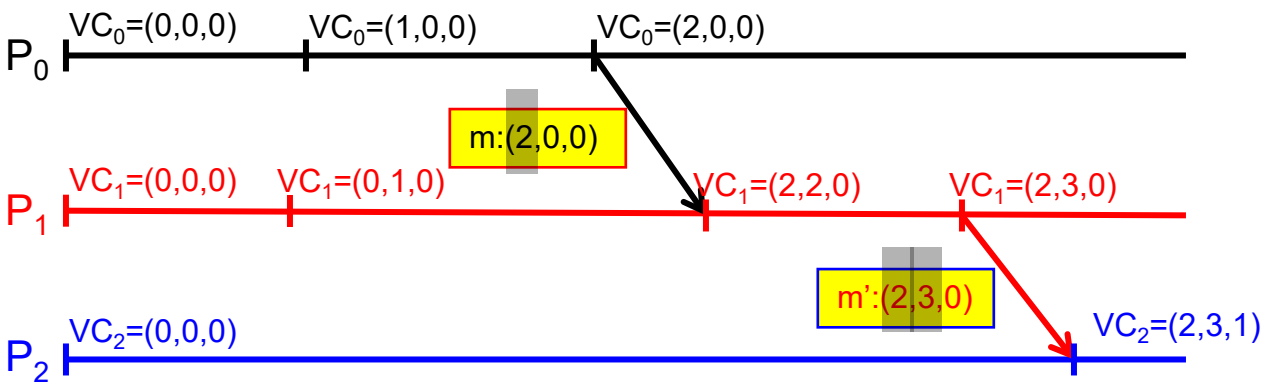
Vector Clock Update Algorithm

- 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