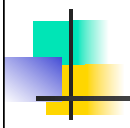
DISTRIBUTED COMPUTER SYSTEMS



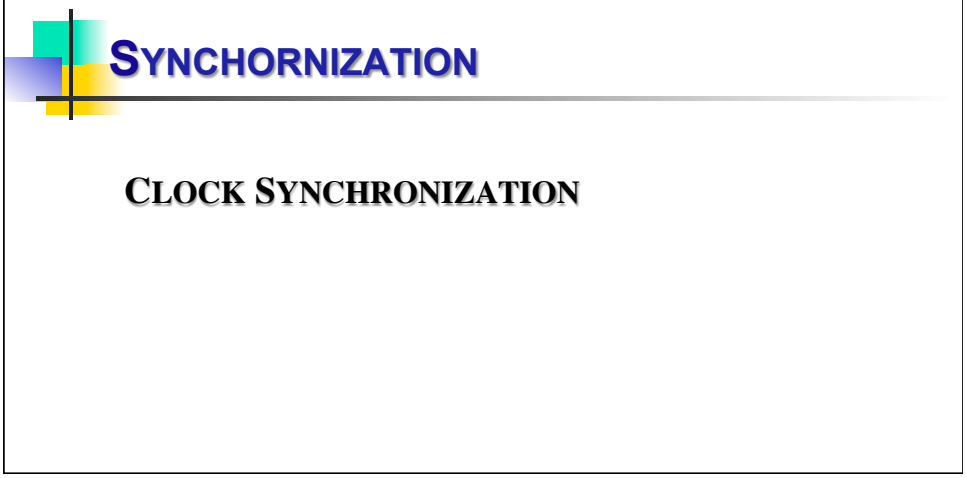
SYNCHRONIZATION

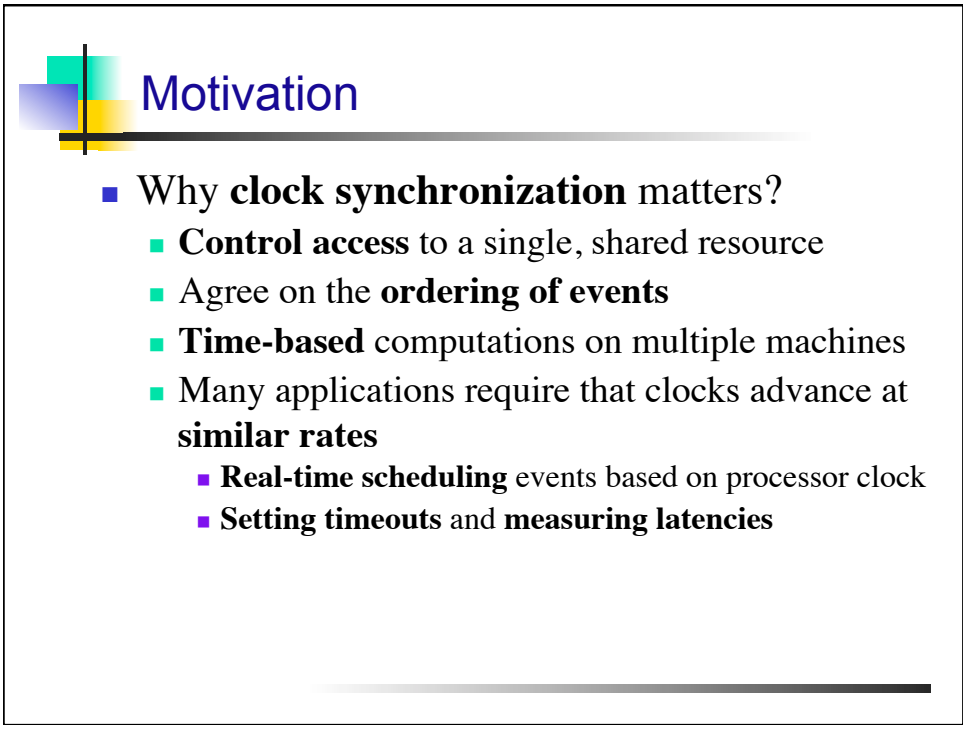
Dr. Jack Lange
Computer Science Department
University of Pittsburgh

Fall 2015

Topics – Clock Synchronization

- Physical Clocks
- Clock Synchronization Algorithms
  - Network Time Protocol
  - Berkeley Algorithm
  - Clock Synchronization in Wireless Networks
- Logical Clocks
  - Lamport's Logical Clocks
  - Vector Clocks


SYNCHORNIZATION
CLOCK SYNCHRONIZATION


Motivation
Why clock synchronization matters?
Control access to a single, shared resource
Agree on the ordering of events
Time-based computations on multiple machines
Many applications require that clocks advance at similar rates
Real-time scheduling events based on processor clock
Setting timeouts and measuring latencies

Synchronization

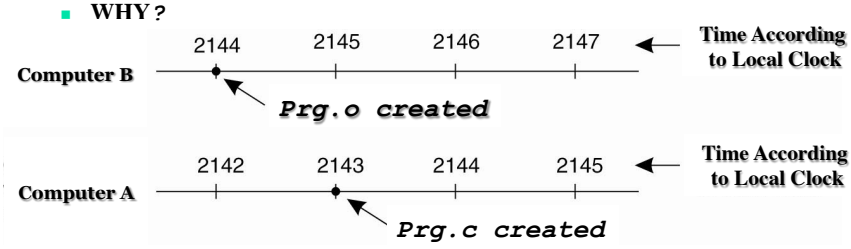
- Synchronization within one system is hard enough
  - Messages
  - Semaphores,
  - Monitors,
  - Barriers,
  - …
- Synchronization among processes in a distributed system is much harder
- Two approaches to achieve synchronization
  - Synchronization based on physical time
  - Synchronization using **relative ordering** rather than **absolute time**

Clock Synchronization

- In a centralized system, time is **unambiguous**
  - Processes obtain the time, by issuing a system call to the kernel
    - Process 1 issues R1 to obtain time – Kernel return T1
    - Process 2 issues R2 to obtain time – Kernel returns T2
    - It is guaranteed that T2 ≥ T1
- In a distributed system, achieving agreement on time is **not** trivial
  - It is impossible to guarantee that physical clocks run at the same frequency

Clock Synchronization

- Lack of global time may cause undesirable behavior for time dependent application
- Example – ***make*** recompiles if ***prg.c*** is newer than ***prg.o***
  - ***make*** on machine A to build ***prg.o***
  - Test on machine B; find and fix a bug in ***prg.c***
  - Re-run ***make*** on machine B, ***prg.o*** remains the same.
  - **WHY?**



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time

SYNCHORNIZATION

PHYSICAL CLOCKS

Physical clocks

- Computer Clocks are acutually Timers, a precisely machined quartz crystal
  - When kept under tension, crystals oscillate at a well-defined frequency
- Associated with the crystal are two registers, a counter and holding register
  - At each oscillation, the counter decrements by 1
  - When counter reaches zero, an interrupt is generated
    - Possible to program a timer to generate an interrupt 60 times a second – Each interrupt corresponds to a clock tick
- Within a singly computer, a clock being slightly off is tolerable, with minimal impact
- In multi-computers systems, clock skew is problematic

Clock Synchronization – Actual Time

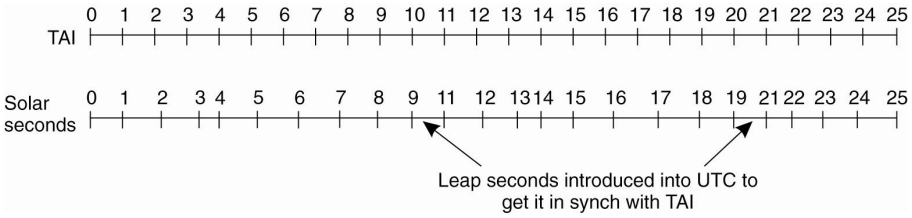
- For time dependent applications, e.g., real-time systems, the **actual time** is important
  - External physical clocks become critical
  - For efficiency and redundancy, multiple physical clocks are needed
- Two issues arise:
  1. How to synchronize the physical clocks with real-world clocks?
  2. How to synchronize these clocks with each other?

Time Measurement

- Atomic clocks make it possible to measure time accurately
  - Several laboratories, equipped with Cesium 133 clocks, periodically report the number of times their clock has ticket to the Bureau International de l'Heure (BIH)
  - BIH computes the average to produce the International Atomic Time (TAI), the number of seconds since 01/01/1958 midnight divided by 9,192,631,370
  - TAI seconds are of constant length, unlike solar seconds.
    - As the mean solar day is getting longer, TAI get out of synch
      - Now, 86,600 TAI seconds are about 3 msec less than a mean solar day

Universal Coordinated Time

- BIH solves the TAI problem by introducing leap seconds whenever the discrepancy grows to 800 msec



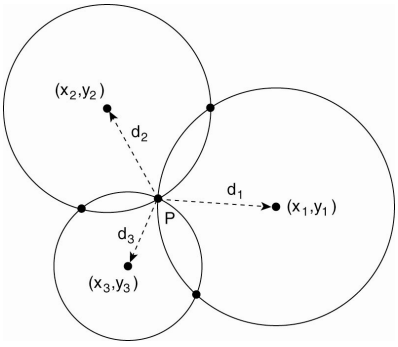
- The resulting time system is referred to as the **Universal Coordinated Time** (UTC)
  - **UTC** is based on constant TAI seconds, but remains in phase with the motion of the sun

UTC Service

- National Institute of Standard Time (NIST) provide UTC by operating a shortwave radio station with call letters WWV from Fort Collins, Colorado
  - WWV broadcasts a short pulse at the start of each UTC second
  - WWV accuracy is ± 1 msec, but due atmospheric fluctuations the accuracy is ± 10 msec
- Satellites offer UTC service
  - GEOS provides UTC with accuracy of 0.5 msec
- Radio receivers for WWV, GEOS and other UTC sources are commercially available
  - Accurate knowledge of the sender and receiver positions is required to compensate for signal propagation delay

GPS – Computing Position in 2-D Space

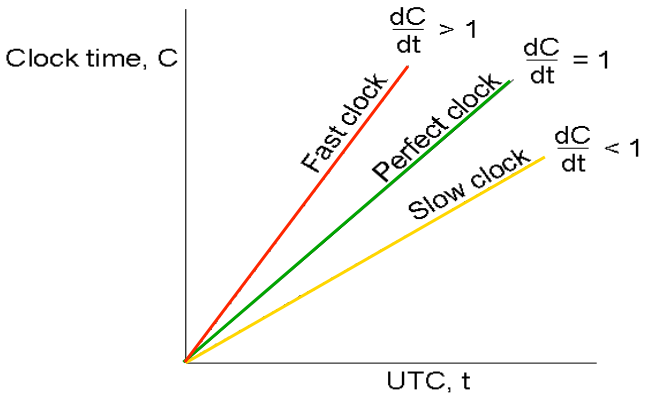
- Measured delay
  - $\Delta_i = (T_{now} - T_i) + \Delta_r$
  - T_i = Timestamp from satellite i
  - T_{now} = Time when message is received
  - Δ_r = Deviation of the receiver's clock from the actual time
- Measured distance
  - $d_i = c\ \Delta_r$, where c is speed of light
  - $d_i = \sqrt{(xi-xr)2+(yi-yr)2}$



Global Positioning System

- Real world facts that complicate GPS
- It takes a while before data on a satellite's position reaches the receiver.
- The receiver's clock is generally not in synch with that of a satellite.

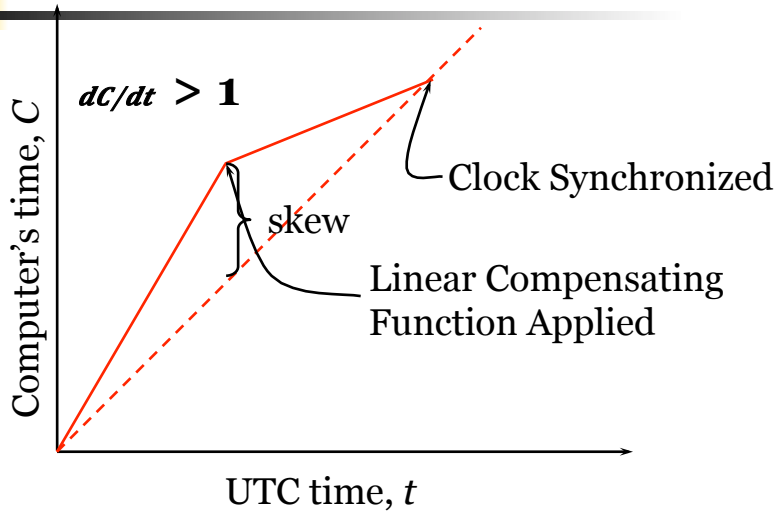
Relation between Clock Time and UTC



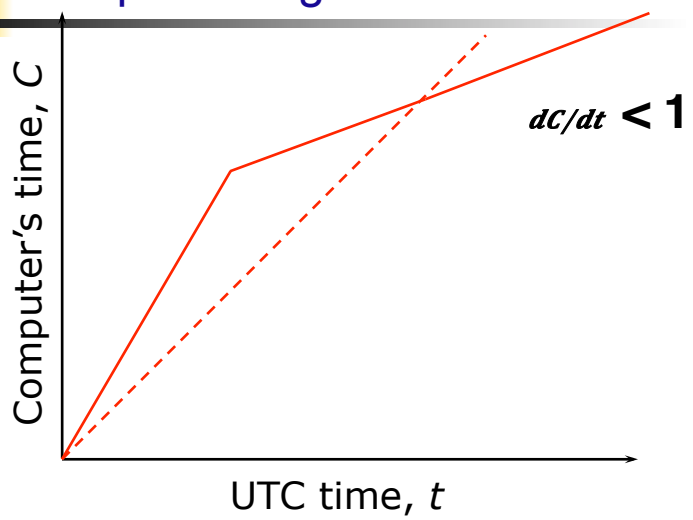
Dealing with drift

- Set computer to true time
  - Not good idea to set clock back – Illusion of time moving backwards can confuse message ordering and software development environments
- ***Gradual*** clock correction
  - **If** fast **then** make clock run slower until it synchronizes
  - **If** slow **then** make clock run faster until it synchronizes

Compensating for a fast clock



Compensating for a fast clock



OS – Dealing With Drift

- To deal with drift OS can take different actions
  - Change rate at which it requests interrupts
    - If system requests interrupts every 10 msec and clock is too slow, request interrupts at 9 msec, for example
  - Perform software correction – Redefine the interval
- Adjustment changes slope of system time
  - Linear compensating function
- After synchronization period is reached
  - Resynchronize periodically
  - Successive application of a second linear compensating function can bring the system time slope closer to the actual slope

SYNCHORNIZATION

Network Time Protocol

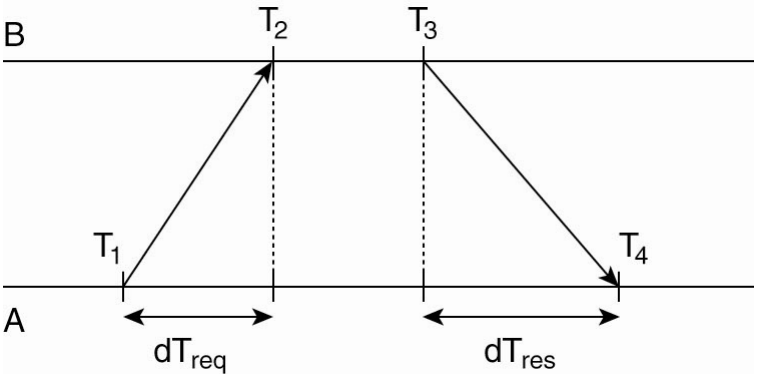
Network Time Protocols

- Clients contact a time server for synchronization,
  - Time server is typically equipped with a WWV receiver or an accurate clock

- Does not account for processing and communication delay
  - Delay Approximation

Network Time Protocol

- Getting the current time from a time server.

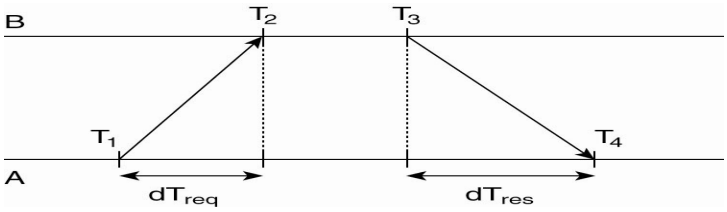


NTP Skew

- Assume delay is nearly symmetric
- Skew Estimation
  - $\theta = [(T2 - T1) + (T3 - T4)]/2$.
- If $\theta < 0$ then clock must be set backward
  - Not advisable
  - Change must be introduced gradually, until the correction is made

NTP Basic Operation

- NTP is set up pairwise
  - A probes B for its current time and B also probes A
- Compute the offset, θ, and the delay estimation, δ
  - $\delta = [(T_2 - T_1)+(T_4 - T_3)]/2$
- Buffer 8 pairs of (θ, δ), and select the minimal value for δ as the best delay estimation between the two servers
  - The associated θ with the minimal delay δ is the most reliable estimation of the offset



NTP Clock Adjustment

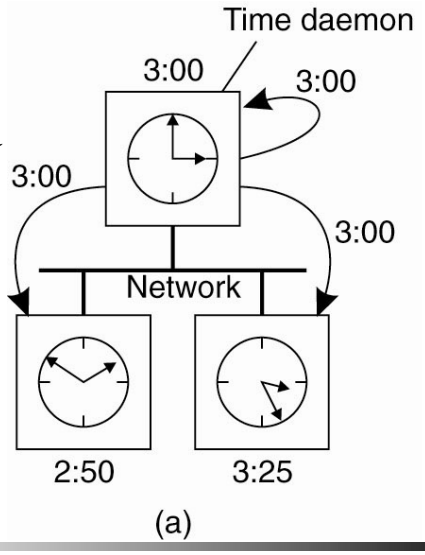
- NTP divides servers into strata
- Strata-1 Servers – Servers with a reference clock such as WWV receiver
  - Clock itself is said to operate at stratum 0
- **A** contacting **B** adjusts its time **only** if its own stratum level is higher than **B**'s
  - After synchronization , **A** set its stratum to **B**'s stratum + 1
  - If **A**'s stratum is lower than **B**, then **B** adjusts itself to **A**

SYNCHORNIZATION

Berkeley Protocol

The Berkeley Algorithm – Request

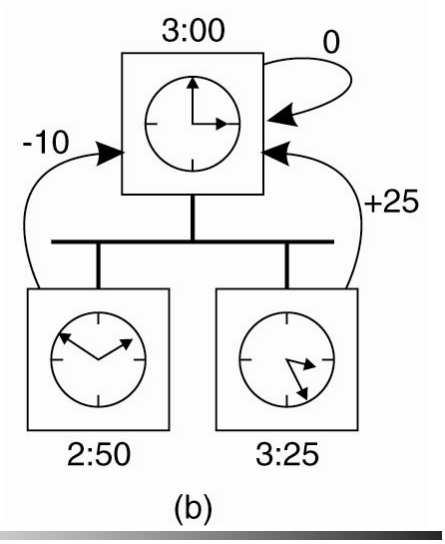
- The algorithm is suitable for a system in which no machine has a WWV receiver
- The time daemon asks all the other machines for their clock values.



(a)

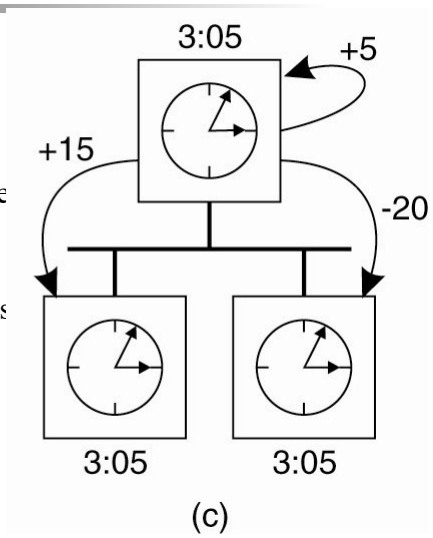
The Berkeley Algorithm – Reply

- The machines answer, with their current time
  - Daemon computes the average of the received answers



The Berkeley Algorithm (3)

- The time daemon tells everyone how to adjust their clock.
  - Advance the clock to the new time or slow the clock down until some specification reduction is achieved



SYNCHORNIZATION

Logical Clocks

Logical Clocks

- For many DS algorithms, associating an event to an absolute "real time" is not essential
  - What's important is that the processes in the Distributed System agree on the ordering in which certain events occur
  - "relative time" may not relate to the "real time".
- Such "clocks" are referred to as Logical Clocks
  - Lamport's timestamps
  - Vector timestamps

Lamport's Logical Clocks

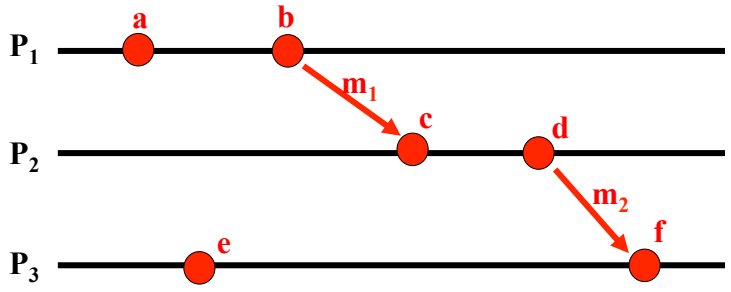
- To synchronize logical clocks, Lamport defines a relation called "happens-before"
- "happens before" relation is denoted as →
- The relation can be observed directly in two situations:
  - If a and b are events in the same process, and a occurs before b, then a → b is true.
  - If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b

"Happens Before" Relation

- The "happens before" relation is transitive
  - a → b and b → c then a → c
- If two events e1 and e2 happen in different processes that do not exchange messages, then
  - e1 → e2 is not true
- The events e1 and e2 are said to be concurrent
  - Nothing can be said about the order in which these events happened

Concurrent Events

- If two events happen without any message, then we can't say anything about their relative occurrence in time.
- We can say that a → b → c → d → f, but we can say little about e other than e → f.

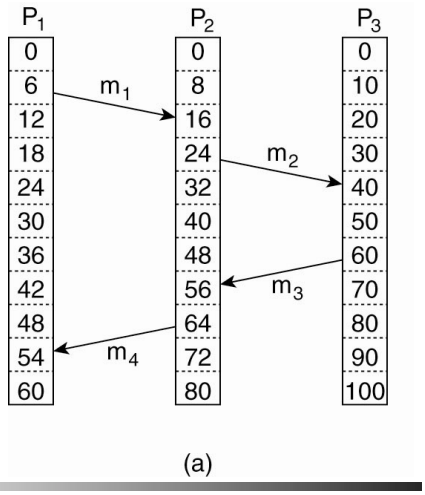


Lamport Logical Clocks

- A way is needed to measure the notion of time, such that each event is assigned a time value
  - $e_i \rightarrow e_j \Rightarrow LC(e_i) < LC(e_j)$
  - LC_i is a local clock and contains increasing values
    - Each process i has own LC_i
  - Increment LC_i on each event occurrence
  - Within same process i, if e_j occurs before e_k
    - $LC_i(e_j) < LC_i(e_k)$
  - If e_s is a send event and e_r receives that send, then
    - $LC_i(e_s) < LC_j(e_r)$

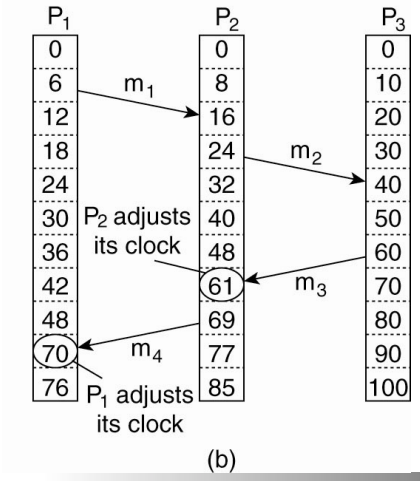
Lamport's Logical Clocks

- Three processes, each with its own clock.
  - The clocks run at different rates.
- Message m_3 arrives before it was sent!



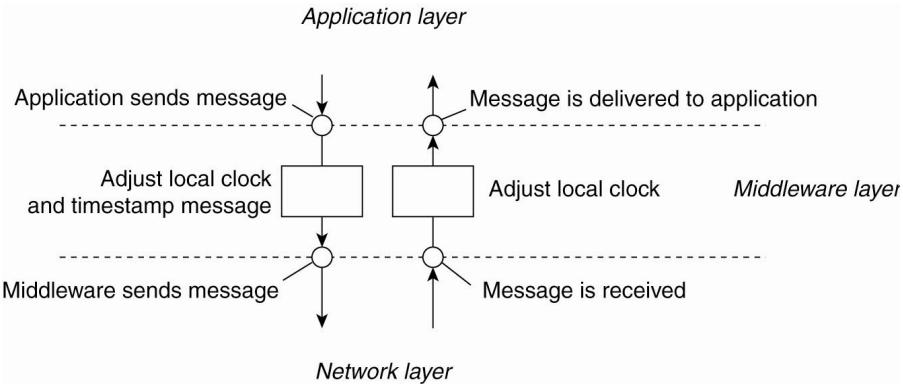
Logical Clocks – Readjusting

- Lamport's algorithm corrects the clocks.



Lamport's Logical Clocks

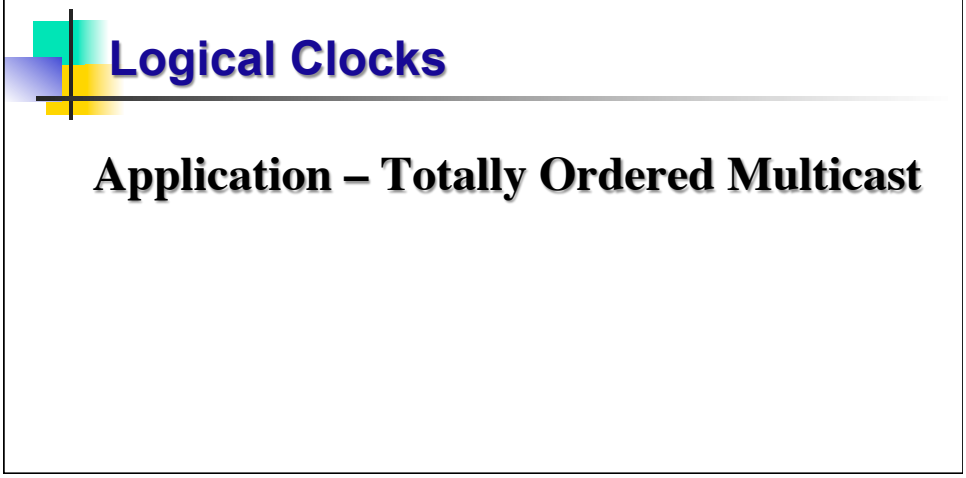
- The positioning of Lamport's logical clocks in distributed systems.

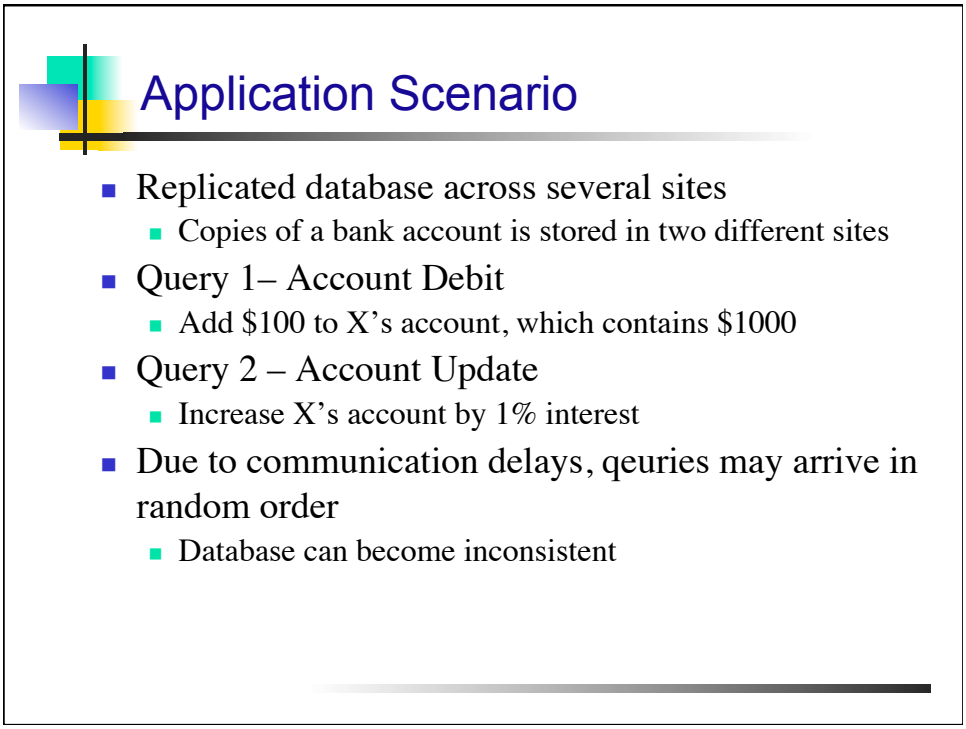


Lamport's Logical Clocks

- To implement Lamport's algorithm, each process, $\mathbf{P}_i$, maintains a local counter, $\mathbf{C}_i$
- Updating counter $\mathbf{C}_i$ for process $\mathbf{P}_i$

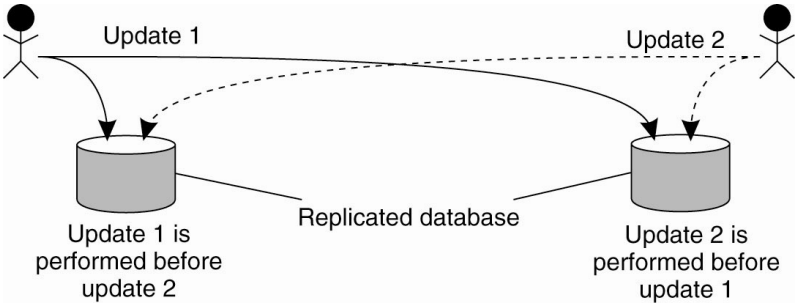
1. Before executing an event $\mathbf{P}_i$ executes $\mathbf{C}_i \leftarrow \mathbf{C}_i + \mathbf{1}$.
2. When process $\mathbf{P}_i$ sends a message m to P_j, it sets m's timestamp $\boldsymbol{ts\ (m)}$ equal to $\mathbf{C}_i$ after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own local counter as $\mathbf{C}_j \leftarrow \mathbf{max\{C}_j\mathbf{,}\ \boldsymbol{ts\ (m)}\mathbf{\}}$, after which it then executes the first step and delivers the message to the application.


Logical Clocks
Application – Totally Ordered Multicast


Application Scenario
Replicated database across several sites
Copies of a bank account is stored in two different sites
Query 1– Account Debit
Add $100 to X's account, which contains $1000
Query 2 – Account Update
Increase X's account by 1% interest
Due to communication delays, qeuries may arrive in random order
Database can become inconsistent

DB Inconsistent State

- Updating a replicated database and leaving it in an inconsistent state.

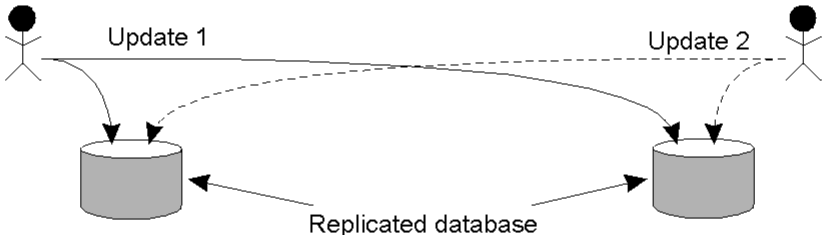


- Totally-Ordered Multicasting

Totally-Ordered Multicasting

- Message is timestamped with sender's logical time
- Message is multicast (including sender itself)
- When message is received
  - It is put into local queue
  - Ordered according to timestamp
  - Multicast acknowledgement
- Message is delivered to applications only when
  - It is at head of queue
  - It has been acknowledged by all involved processes

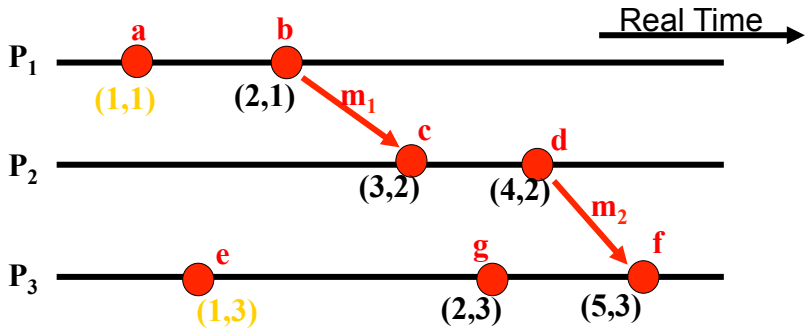
Totally-Ordered Multicasting



- Update 1 is time-stamped and multicast. Added to local queues.
- Update 2 is time-stamped and multicast. Added to local queues.
- Acknowledgements for Update 2 sent/received. Update 2 can now be processed.
- Acknowledgements for Update 1 sent/received. Update 1 can now be processed.
- Note – All queues are the same, as the timestamps have been used to ensure the "happens-before" relation holds.

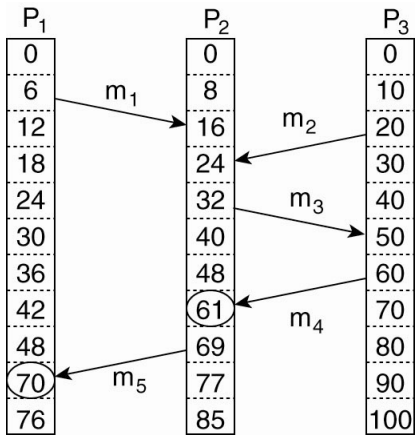
Limitation of Lamport's Algorithm

- $e_i \rightarrow e_j \Rightarrow LC(e_i) < LC(e_j)$
- However, $LC(e_i) < LC(e_j)$ does not imply $e_i \rightarrow e_j$
  - For example, (1,1) < (1,3), but events a and e are concurrent



Vector Clocks (1)

- Lamport clocks do not capture causality
  - Sending m3 may have depended on m1
- Causality can be captured by **Vector Clocks**



Vector Clocks (2)

Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

1. $VC_i [i]$ is the number of events that have occurred so far at P_i. In other words, $VC_i [i]$ is the local logical clock at process P_i .
2. If $VC_i [j] = k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j .

Vector Clocks – Basic Steps

Steps carried out to accomplish property 2:

1. Before executing an event P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets *m*'s (vector) timestamp *ts (m)* equal to VC_i after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own vector by setting $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each *k*, after which it executes the first step and delivers the message to the application.

Vector Clocks –Local Events

- **Initialization**
  - The vector timestamp for each process is initialized to (0,0,…,0)
- **Local Event**
  - When an event occurs on process P_i:
    - $VT_i[i] \leftarrow VT_i[i] + 1$
    - For example on processor 3, (1,2,1,3) → (1,2,2,3)

Vector Clocks – Messages

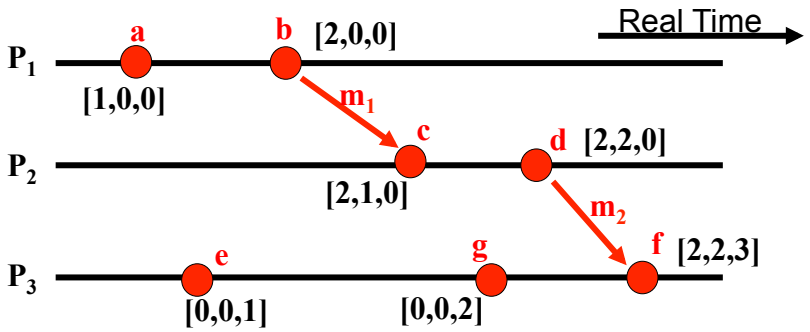
- **Message Passing**
  - When P_i sends a message to P_j, the message has timestamp t[]=VT_i[]
  - When P_j receives the message, it sets VT_j[k] to max (VT_j[k],t[k]), for k = 1, 2, …, N
    - For example, P_2 receives a message with timestamp (3,2,4) and P_2's timestamp is (3,4,3), then P_2 adjusts its timestamp to (3,4,4)

Comparing Clock Vectors

- $VT_1 = VT_2$ **iff** $VT_1[i] = VT_2[i]$ for all i
- $VT_1 \leq VT_2$ **iff** $VT_1[i] \leq VT_2[i]$ for all i
- $VT_1 < VT_2$ **iff** $VT_1 \leq VT_2$ & $VT_1 \neq VT_2$
  - For example, (1, 2, 2) < (1, 3, 2)

Vector Clocks Analysis

- Claim – e → e' **iff** e.VT < e'.VT



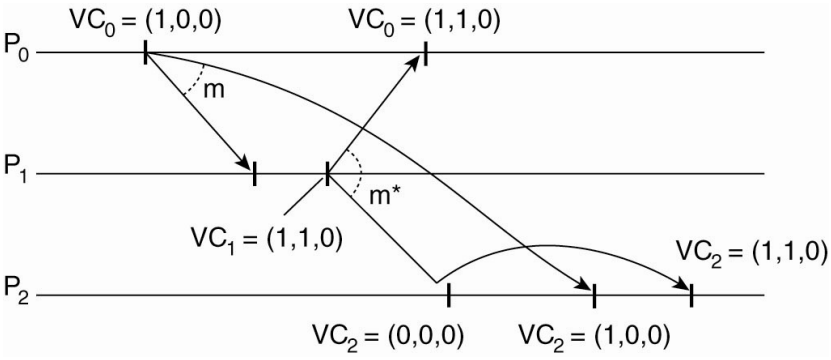
Causally-Ordered Multicasting

- For ordered delivery of related messages
  - Upon sending a message, P_i will only increment $V_i[i]$ by 1
  - When it delivers a message m with timestamp ts(m), it only adjusts $VC_i[k]$ to **max**$\{VC_i[k], ts(m)]\}$, for each k
- When P_k gets a message, m, from P_j, with ts(m), the message is buffered until 1 and 2 are met:
  - **Condition 1: $ts(m)[j] = VC_k[j] + 1$**
    - Timestamp indicates this is the next message that $\mathbf{P_k}$ is expecting from j
  - **Condition 2: $ts(m)[i] \leq VC_k[i]$ for all $i \neq j$**
    - P_k has seen all messages that were seen by P_j, when it sent the message, m

Causally-Ordered Multicasting

- For ordered delivery of related messages
  - Upon sending a message, Pi will only increment Vi[i] by 1
  - When it delivers a message m with timestamp ts(m), it only adjusts VCi [k] to max{VC_i[k], ts(m)]}, for each k
- When P_k gets a message, m, from P_j, with ts(m), the message is buffered until 1 and 2 are met:
  - Condition 1: ts(m)[j] = VC_k[j] + 1
    - Timestamp indicates this is the next message that P_k is expecting from j
  - Condition 2: ts(m)[i] ≤ VC_k[i] for all i ≠ j
    - P_k has seen all messages that were seen by P_j, when it sent the message, m
- Causally-ordered multicasting is weaker then totally-ordered multicasting
  - Two messages that are not related to each may be delivered in any order
    - May be delivered in different order in different locations

Enforcing Causal Communication



Conclusion

- Physical Clocks
- Clock Synchronization Algorithms
  - Network Time Protocol
  - Berkeley Algorithm
  - Clock Synchronization in Wireless Networks
- Logical Clocks
  - Lamport's Logical Clocks
  - Vector Clocks
  - Enforcing Causal Communication