Unit 3:
Synchronization, Coordination and Agreement

References:

1. G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**
2. Andrew S. Tanenbaum and Maarten van Steen**; Distributed Systems: Principles and Paradigms, 2nd Edition.**

Synchronization, Coordination and Agreement

3.1 Time Synchronization

3.1.1 Time and time synchronization

3.1.2 Physical Clock Synchronization

3.1.2.1 Berkeley Algorithm

3.1.2.2 Cristian's Algorithm

3.1.2.3 Network Time Protocol (NTP)

3.1.3 Logical Clocks Synchronization

3.1.3.1 Events and ordering

3.1.3.2 Partical, causal and total ordering

3.1.3.3 Global State and State Recording

3.1.3.4 Lamport logical clock

3.1.3.5 Vector clock

3.2 Distributed Co-ordination

3.2.1 Distributed mutual exclusion

3.2.2 Mutual Exclusion Algorithms

3.2.2.1 Central Coordinator Algorithm

3.2.2.2 Token Ring Algorithm

3.2.2.3 Lamport's Algorithm

3.2.2.4 Ricart-Agrawala
(Non-Token based & token based)

3.2.3 Distrbuted Leader Election

3.2.3.1 Bully algorithm

3.2.3.2 Ring Algorithms

Background of Synchronization

1. ***Successful handover of computed data/result from one process to another is simply called Inter Process Communication.*** Different methods/techniques are used in IPCs as ***Pipe, Monitor, Semaphore, message passing, and group communication***. IPCs is an entire story and closely related to the process of cooperation and synchronization. From view of operating systems how are the critical regions are implemented and resources are allocated are major focus.

2. In single CPU systems, critical regions, mutual exclusion and others synchronization problems are solved mostly by using semaphore, Monitors etc. But, these methods are only not suited for the distributed environment because they used the shared memory of same Kernel. In case of distributed operating system other techniques are also needed.

Clock Synchronization DOS

Communication between processes in a distributed system can have unpredictable delays, processes can fail, messages may be lost

Synchronization in distributed systems is harder than in centralized systems because the need for distributed algorithms.

Synchronization in distributed system is more complicated than in the centralized system because of 4 major reasons.

1. *Information is scatter among multiple machine.*
2. *Processes make decision based on only on local information.*
3. *A Single point of failure in system should be avoided*
4. ***No common clock or other precise global time source exists***

=> Points number 1 to 3 explain that it is unpredictable to collect all the information in a single place for processing because for resource allocation all requests need to send into a single manager, which examines and grant or denies request based on resource allocation table. For heavy system it is great burden on particular machine or process only. Moreover, if a system is failed then entire system become non predictable or unreliable

=> **Clock is crucial issue because it is ambiguous. If a process of a system wants to talk to its kernel. It means the time of asking and reply should hold the principles HAPPEN BEFORE.**

Clock

- How a computer timer works?

1. A counter register and a holding register.
2. The counter is decremented by a quartz crystals oscillator.
   When it reaches zero, an interrupt is generated and the counter is reloaded from the holding register.
   - e.g, interrupt 60 times per second.
   - Each interrupt is called **clock tick**.

Clock Synchronization

- In a distributed system the internal clocks of several computers may differ with each other .

- Even when initially set accurately, real clocks will differ after some time due to
  - clock drift, caused by clocks counting time at slightly different rates.
- In serial communication, some people use the term **"clock synchronization"** merely to discuss frequency synchronization and phase synchronization. Such "clock synchronization" is used in synchronization in telecommunications and automatic baud rate detection.

Physical time

Solar time

- 1 sec = 1 day / 86400
- Problem: days are of different lengths (due to tidal friction, etc.)
- mean solar second: averaged over many days

Greenwich Mean Time (GMT)

- The mean solar time at Royal Observatory in Greenwich, London
- Greenwich located at longitude 0, the line that divides east and west

1 Sec

- Hong long is 1 second?
  - determined by solar day
  - clock of 1 second is adjusted to match the length of 1 solar-second.
- Solar-day: noon to noon (the sun at the highest point in the sky from a certain point on earth)

- GMT time measured at Greenwich, which is located at longitude 0 line (similar to the equator line dividing north and south)

Coordinated Universal Time (UTC)

International atomic time (TAI)

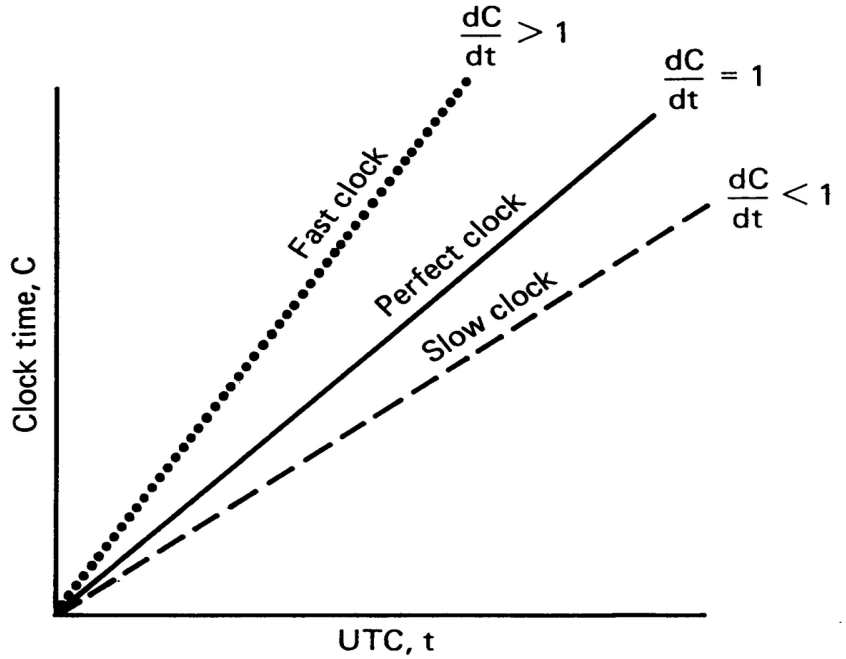
- 1 sec ≡ time for Cesium-133 atom to make 9,192,631,770 state transitions.
- TAI time is simply the number of Cesium-133 transitions since midnight on Jan 1, 1958.
- Accuracy: better than 1 second in six million years
- Problem: Atomic clocks do not keep in step with solar time

Coordinated Universal Time (UTC)

- Based on the atomic time (TAI) and introduced from 1 Jan 1972
- A leap second is occasionally inserted or deleted to keep in step with solar time when the difference btw a solar-day and a TAI-day is over 800ms

Drift Rate

- Is the rate at which the clock ticks.
- Different clocks may have different drift rates and hence needs to be synchronized.
- One can determine how often they should be synchror



Not all clock's tick precisely at the current rate.

Clock Synchronization

It is the process of setting all the cooperating systems of distributed network to the same logical or physical clock.

Logical and Physical Clocks

- Clock synchronization is dramatic and it is fitting in process execution. But, Is it possible to synchronize all the clocks in the distributed environment into single clock.

- There are different concepts and implementations regarding the clock synchronization by using logical and physical clocks.
- logical clocks - to provide consistent event ordering
- physical clocks - clocks whose values must not deviate from the real time by more than a certain amount.

Logical and Physical Clocks

- **Logical Clocks.**
  - For many applications:
    - it is sufficient that all machines agree on the same time.
    - it is not essential that this time also agree with the real time
  - E.g. make example - it is adequate that all machines agree that it is 10:00 even if it is really 10:02.
  - Meaning: it is the *internal consistency of the clocks that matters, not* whether they are particularly close to the real time.
  - For these algorithms it is conventional to speak of the clocks as ***logical clocks.***

- **Physical Clocks**
  - when the additional constraint is present that the clocks
    - must not only be the same,
    - but also must not deviate from the real time by more than a certain amount,
  - Then the clocks are called ***physical clocks.***

Problems with Different Clocks

- There are several problems that occur as a repercussion of clock rate differences.
- Besides the incorrectness of the time itself, there are problems associated with clock skew that take on more complexity in a distributed system in which several computers will need to realize the same global time.
- **Example:** in Unix systems the make command is used to compile new or modified code without the need to recompile unchanged code. The make command uses the clock of the machine it runs on to determine which source files need to be recompiled. If the sources reside on a separate file server and the two machines have unsynchronized clocks, the make program might not produce the correct results
- Hence there is a need of **Clock Synchronization**

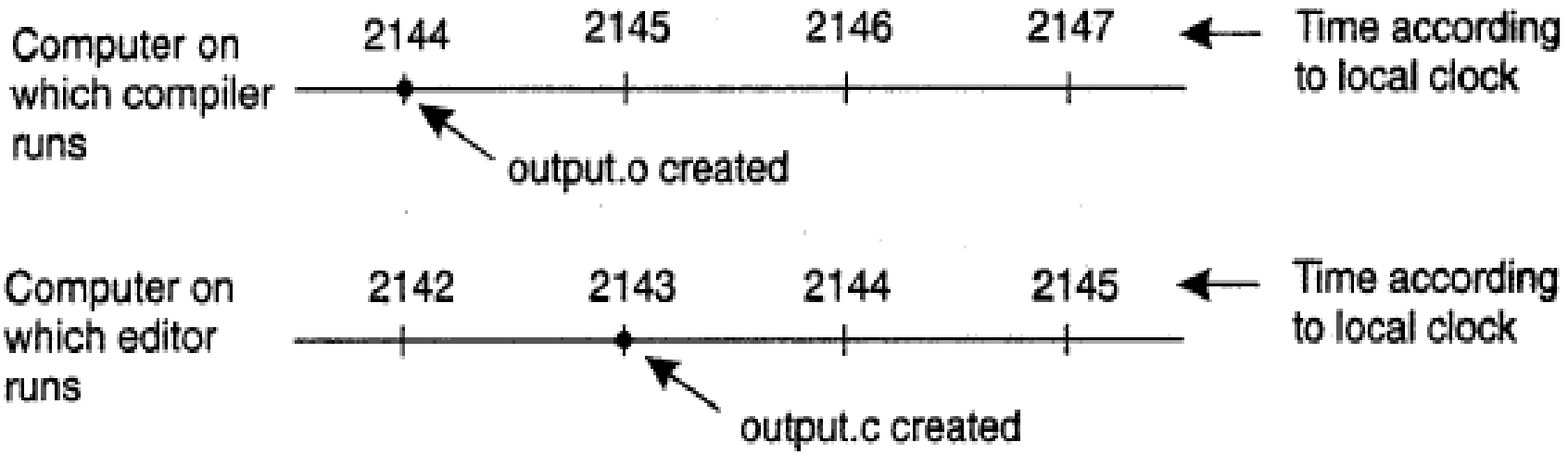


Fig: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time

Clock Synchronization Algorithms

- Physical Clock Synchronization
  1. **Cristian's algorithm – Centralized System**
  2. **Berkeley algorithm – Centralized System**
  3. **Network Time Protocol – Distributed System**

- Logical Clock Synchronization
  1. **Lamport timestamps – Distributed System**
  2. **Vector clocks – Distributed System**

Physical Clock Synchronization

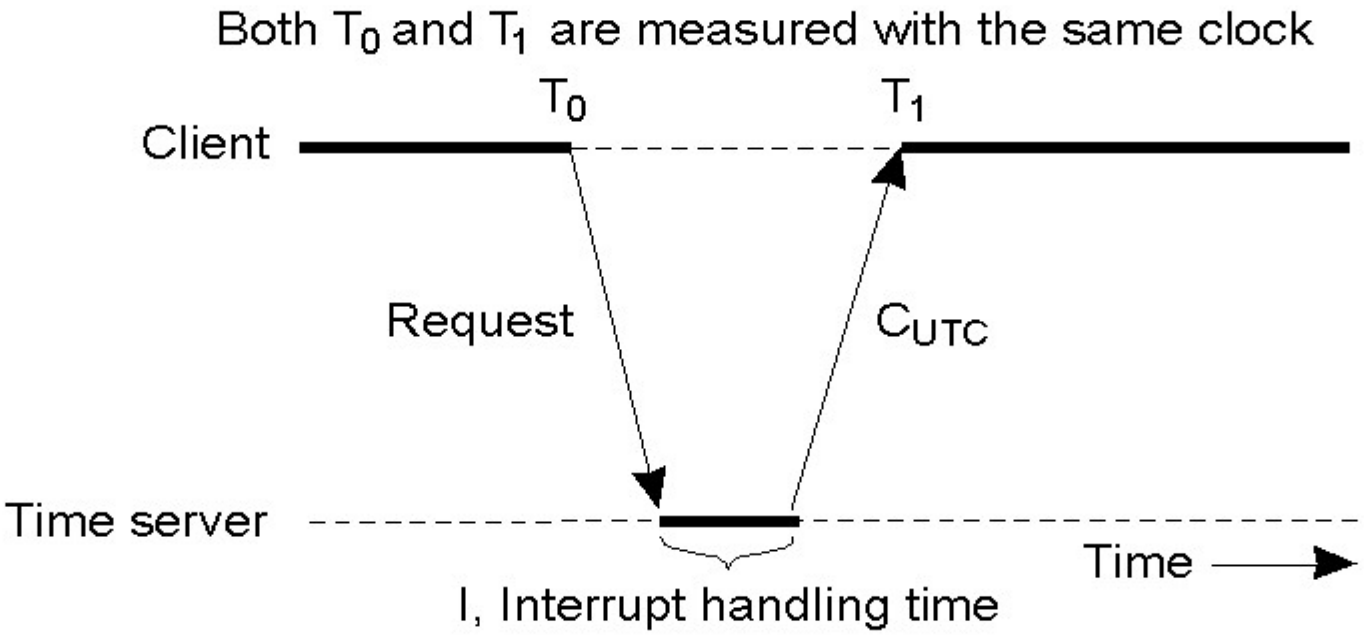
1. **Cristian's algorithm**
2. **Berkeley algorithm**
3. **Network Time Protocol (NTP)**

Cristian's algorithm

- Cristian's algorithm relies on the existence of a time server as a physical clock synchronization algorithm.
- The time server maintains its clock by using a radio clock or other accurate time source, then all other computers in the system stay synchronized with it.
- A time client will maintain its clock by making a procedure call to the time server.
- Variations of this algorithm make more precise time calculations by factoring in network radio propagation time.

Cristian's algorithm

- The time server needs to change its time gradually
- The time server, while responding to a client, needs to consider msg delays, subtract
  - (T1 - T0 – I) which is called Round Trip Time (RTT)



- Getting the current time from a time server

Cristian's algorithm

Cristian's Algorithm works between a process P, and a time server S — connected to a source of UTC (Coordinated Universal Time). Then the algorithm goes like this.

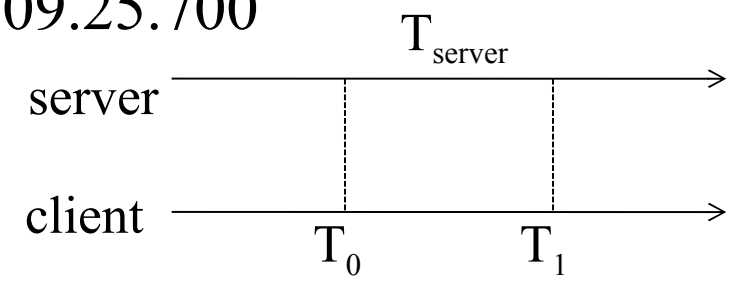
1. P requests the time from S
2. After receiving the request from P, S prepares a response and appends the time T from its own clock.
3. P then sets its time to be T + RTT/2

Cristian's algorithm - Accuracy

- P needs to record the Round Trip Time (RTT) of the request it made to S so that it can set its clock to T + RTT/2. This method assumes that the RTT is split equally between both request and response, which may not always be the case but is a reasonable assumption on a LAN connection.
- Further accuracy can be gained by making multiple requests to S and using the response with the shortest RTT. We can estimate the accuracy of the system as follows.
  - Let min be the minimum time to transmit a message one-way. The earliest point at which S could have placed the time T, was min after P sent its request. Therefore, the time at S, when the message is received by P, is in the range (T + min) to (T + RTT - min). The width of this range is (RTT - 2*min). This gives an accuracy of (RTT/2 - min).

Cristian's algorithm: example

- Send request at 5:08:15.100 (T_0)
- Receive response at 5:08:15.900 (T_1)
  - Response contains 5:09:25.300 (T_{server})
- Round-trip time is $T_1 - T_0$
  - 5:08:15.900 - 5:08:15.100 = 800 ms
- Best guess: timestamp was generated 400 ms ago
- Set the local time to T_{server} + *round-trip-time/2*
  - 5:09:25.300 + 400 = 5:09.25.700
- Accuracy: ± *round-trip-time/2*



Physical Clock Synchronization

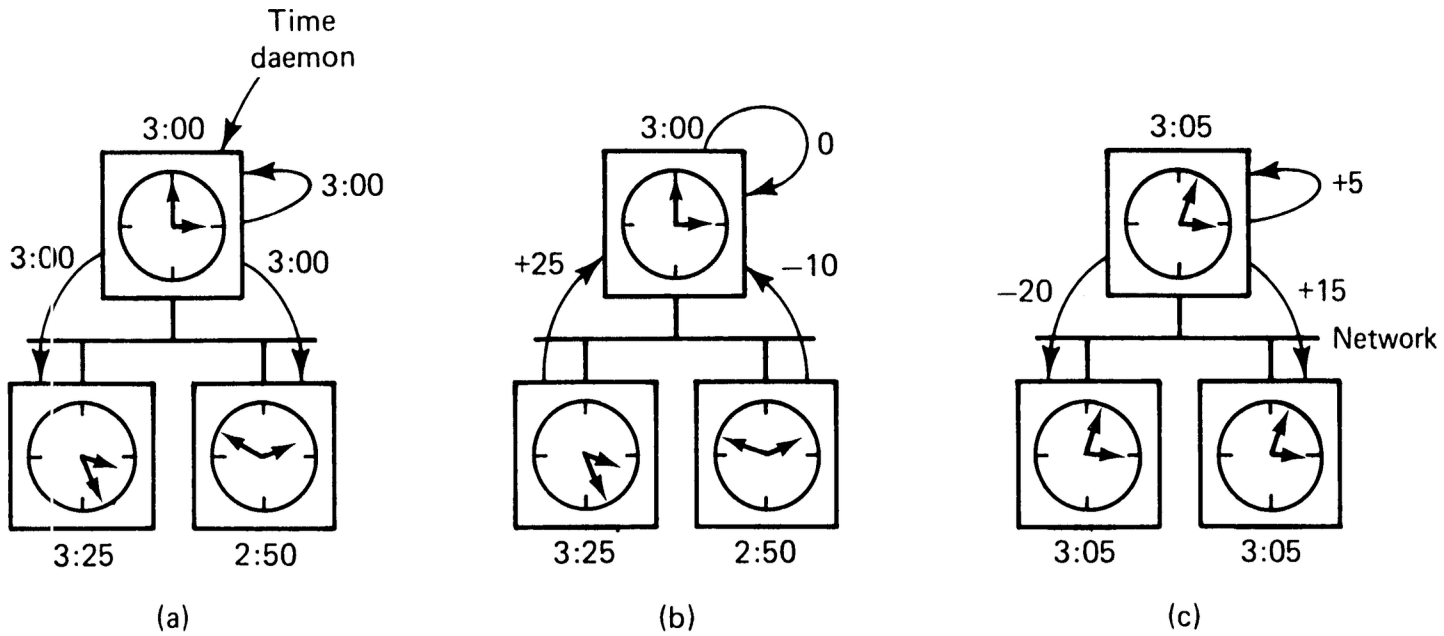
1. **Cristian's algorithm**
2. **Berkeley algorithm**
3. **Network Time Protocol (NTP)**

The Berkeley algorithm

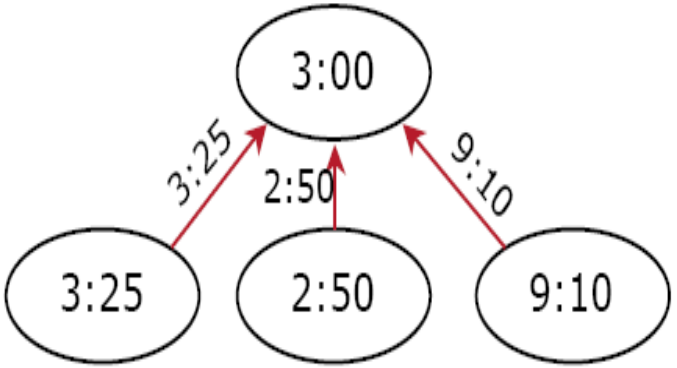
- Berkeley algorithm is also a physical clock synchronization algorithm based on centralized system.
- This algorithm is more suitable for systems where a radio clock is not present.
- This system has no way of making sure of the actual time other than by maintaining a global average time as the global time.
- A time server will periodically fetch the time from all the time clients, average the results, and then report back to the clients the adjustment that needs be made to their local clocks to achieve the average.
- This algorithm highlights the fact that internal clocks may vary not only in the time they contain but also in the clock rate.
- Often, any client whose clock differs by a value outside of a given tolerance is disregarded when averaging the results. This prevents the overall system time from being drastically skewed due to one erroneous clock.

The Berkeley algorithm

- Is an averaging algorithm
  - The time daemon asks all the other machines for their clock values.
  - The machines answer.
  - The Time daemon tells everyone how to adjust their clock.

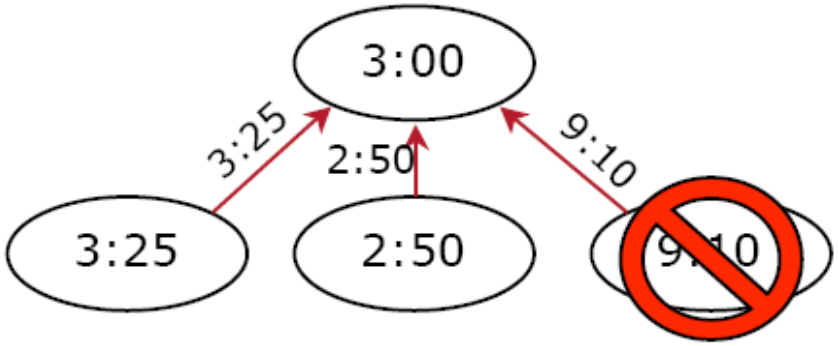


(a)



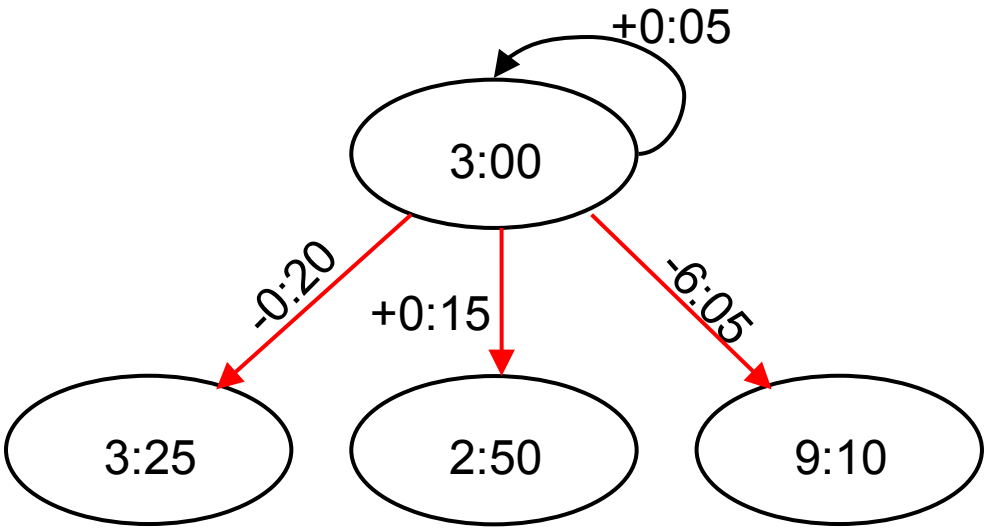
1. Request timestamps from all slaves

(b)



2. Compute fault-tolerant average:

$$\frac{3:25+2:50+3:00}{3}=3:05$$



(c)

3. Send offset to each client

The Berkeley algorithm

- The server process in the Berkeley algorithm, called the master, periodically polls other slave processes to synchronize the physical clock. Generally speaking, the algorithm is:
  1. A master is chosen via an election process such as Chang and Roberts algorithm.
  2. The master polls the slaves who reply with their time in a similar way to Cristian's algorithm.
  3. The master observes the round-trip time (RTT) of the messages and estimates the time of each slave and its own.
  4. The master then averages the clock times, ignoring any values it receives far outside the values of the others.
  5. Instead of sending the updated current time back to the other process, the master then sends out the amount (positive or negative) that each slave must adjust its clock. This avoids further uncertainty due to RTT at the slave processes.

Problem with Berkeley Algorithm

- Time *not* a reliable method of synchronization
- Users mess up clocks
  - (and forget to set their time zones!)
- Unpredictable delays in Internet
- Relativistic issues
  - If *A* and *B* are far apart physically, and two events T_A and T_B are very close in time, then which comes first? how do you know?

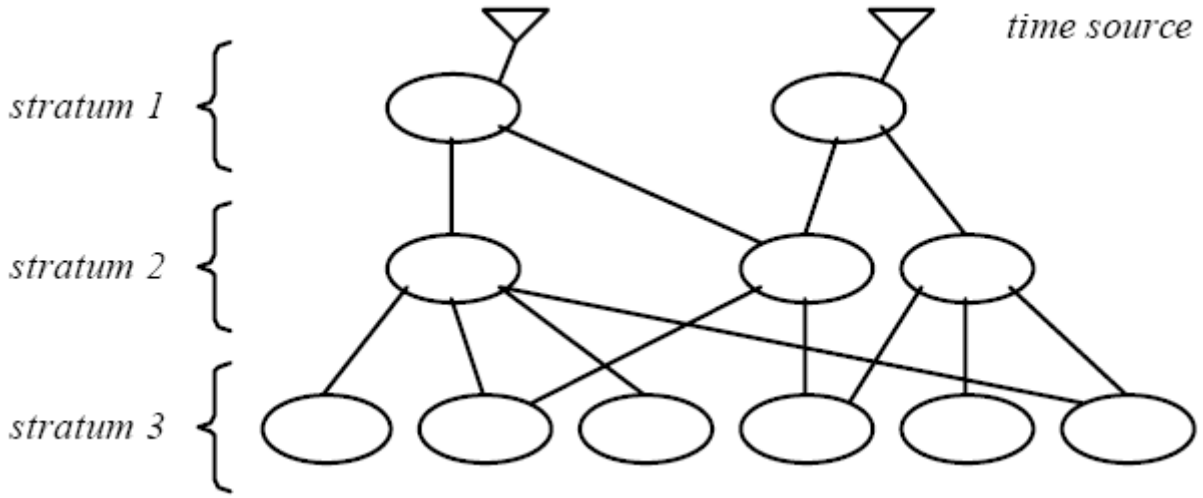
Physical Clock Synchronization

1. **Cristian's algorithm**
2. **Berkeley algorithm**
3. **Network Time Protocol (NTP)**

Network Time Protocol (NTP)

- NTP is the most commonly used Internet time protocol and the one provides best accuracy (RFC 1305, http://tf.nist.gov/service/its.htm).
- Computers often include NTP software in OS. The client software periodically gets updates from one or more servers (average them).
- Time servers listen to NTP requests on port 123, and reply a UDP/IP data packet in NTP format, which is a 64-bit timestamp in UTC seconds since Jan 1, 1900 with a resolution of 200 pico-s.
- Many NTP client software for PC only gets time from a single server (no averaging). The client is called SNTP (Simple Network Time Protocol, RFC 2030), a simple version of NTP.

NTP synchronization subnet



1st stratum: machines connected directly to accurate time source

2nd stratum: machines synchronized from 1st stratum machines

…

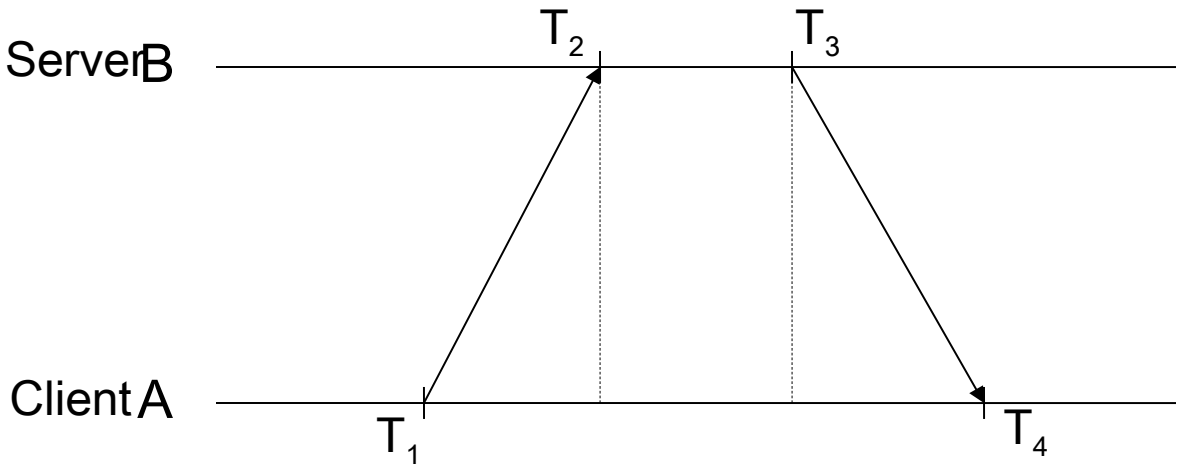
NTP goals

- Enable clients across Internet to be accurately synchronized to UTC despite message delays
  - Use statistical techniques to filter data and improve quality of results
- Provide reliable service
  - Survive lengthy losses of connectivity
  - Redundant paths
  - Redundant servers
- Enable clients to synchronize frequently
  - Adjustment of clocks by using offset (for symmetric mode)
- Provide protection against interference
  - Authenticate source of data

NTP Synchronization Modes

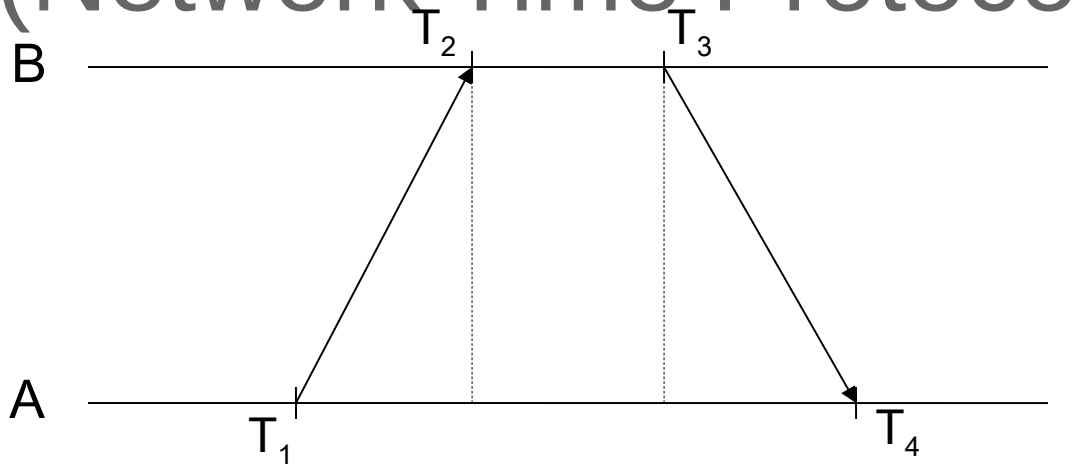
- Multicast (for quick LANs, low accuracy)
  - server periodically multicasts its time to its clients in the subnet
- Remote Procedure Call (medium accuracy)
  - server responds to client requests with its actual timestamp
  - like Cristian's algorithm
- Symmetric mode (high accuracy)
  - used to synchronize between the time servers (peer-peer)
- *All messages delivered unreliably with UDP*

NTP (Network Time Protocol)



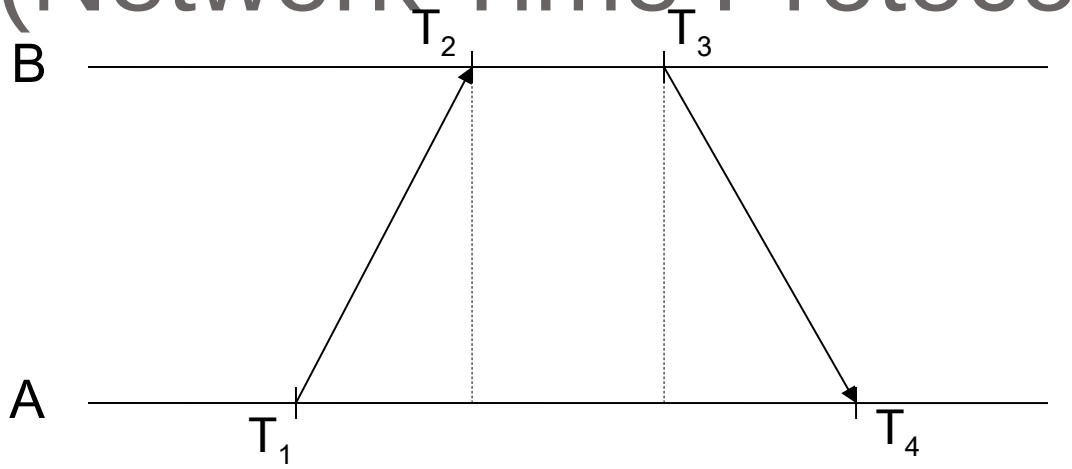
- A requests time of B at its own T_1
- B receives request at its T_2, records
- B responds at its T_3, sending values of T_2 and T_3
- A receives response at its T_4.

NTP (Network Time Protocol)



- Question: what is θ= $T_B - T_A$?
- Assume transit time is approximately the same both ways
- Assume that *B* is the time server that *A* wants to synchronize to

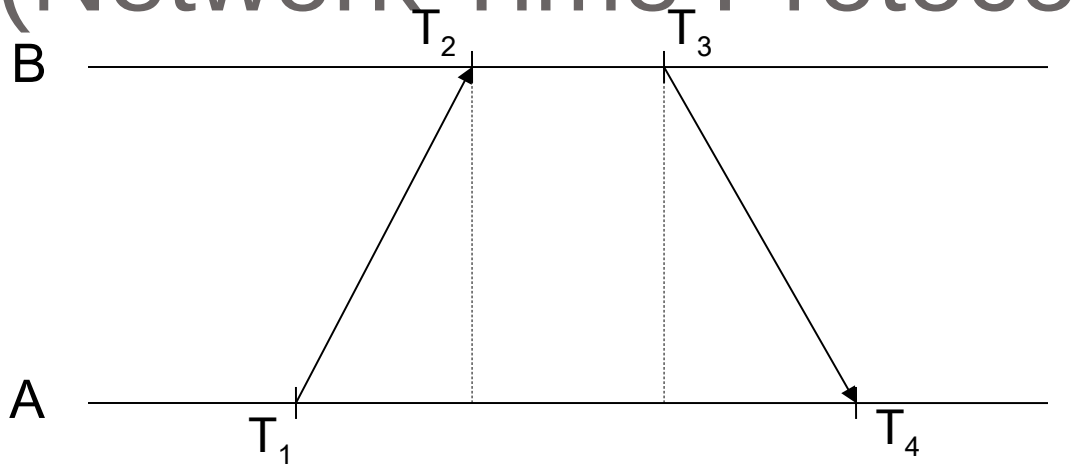
NTP (Network Time Protocol)



- *A* knows ($T_4 - T_1$) from its own clock
- *B* reports T_3 and T_2 in response to NTP request
- *A* computes total transit time of

$$(T_4 - T_1) - (T_3 - T_2)$$

NTP (Network Time Protocol)

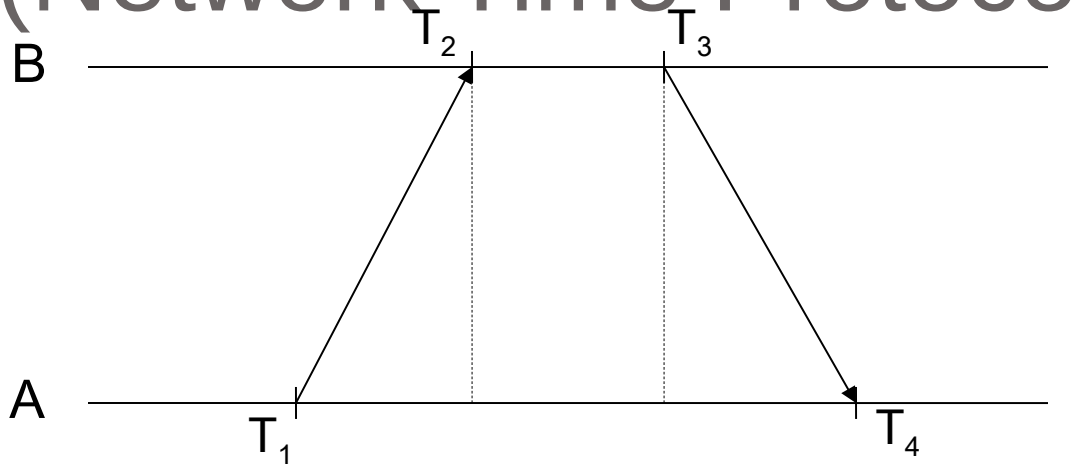


- One-way transit time is approximately ½ total, i.e., $\frac{(T_4 - T_1) - (T_3 - T_2)}{2}$

- *B*'s clock at T_4 reads approximately

$$T_3 + \frac{(T_4 - T_1) - (T_3 - T_2)}{2} = \frac{(T_4 - T_1) + (T_2 + T_3)}{2}$$

NTP (Network Time Protocol)



- *B*'s clock at T_4 reads approximately

$$\frac{(T_4 - T_1) + (T_2 + T_3)}{2}$$

- Thus, difference between *B* and *A* clocks at T_4 is

$$\frac{(T_4 - T_1) + (T_2 + T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Synchronization, Coordination and Agreement

3.1 Time Synchronization

3.1.1 Time and time synchronization

3.1.2 Physical Clock Synchronization

3.1.2.1 Berkeley Algorithm

3.1.2.2 Cristian's Algorithm

3.1.2.3 Network Time Protocol (NTP)

3.1.3 Logical Clocks Synchronization

3.1.3.1 Events and ordering

3.1.3.2 Partical, causal and total ordering

3.1.3.3 Global State and State Recording

3.1.3.4 Lamport logical clock

3.1.3.5 Vector clock

3.2 Distributed Co-ordination

3.2.1 Distributed mutual exclusion

3.2.2 Mutual Exclusion Algorithms

3.2.2.1 Central Coordinator Algorithm

3.2.2.2 Token Ring Algorithm

3.2.2.3 Lamport's Algorithm

3.2.2.4 Ricart-Agrawala
(Non-Token based & token based)

3.2.3 Distrbuted Leader Election

3.2.3.1 Bully algorithm

3.2.3.2 Ring Algorithms

Logical Clock Synchronization

1. Lamport's timestamps (Algorithm)
2. Vector clocks (Algorithm)

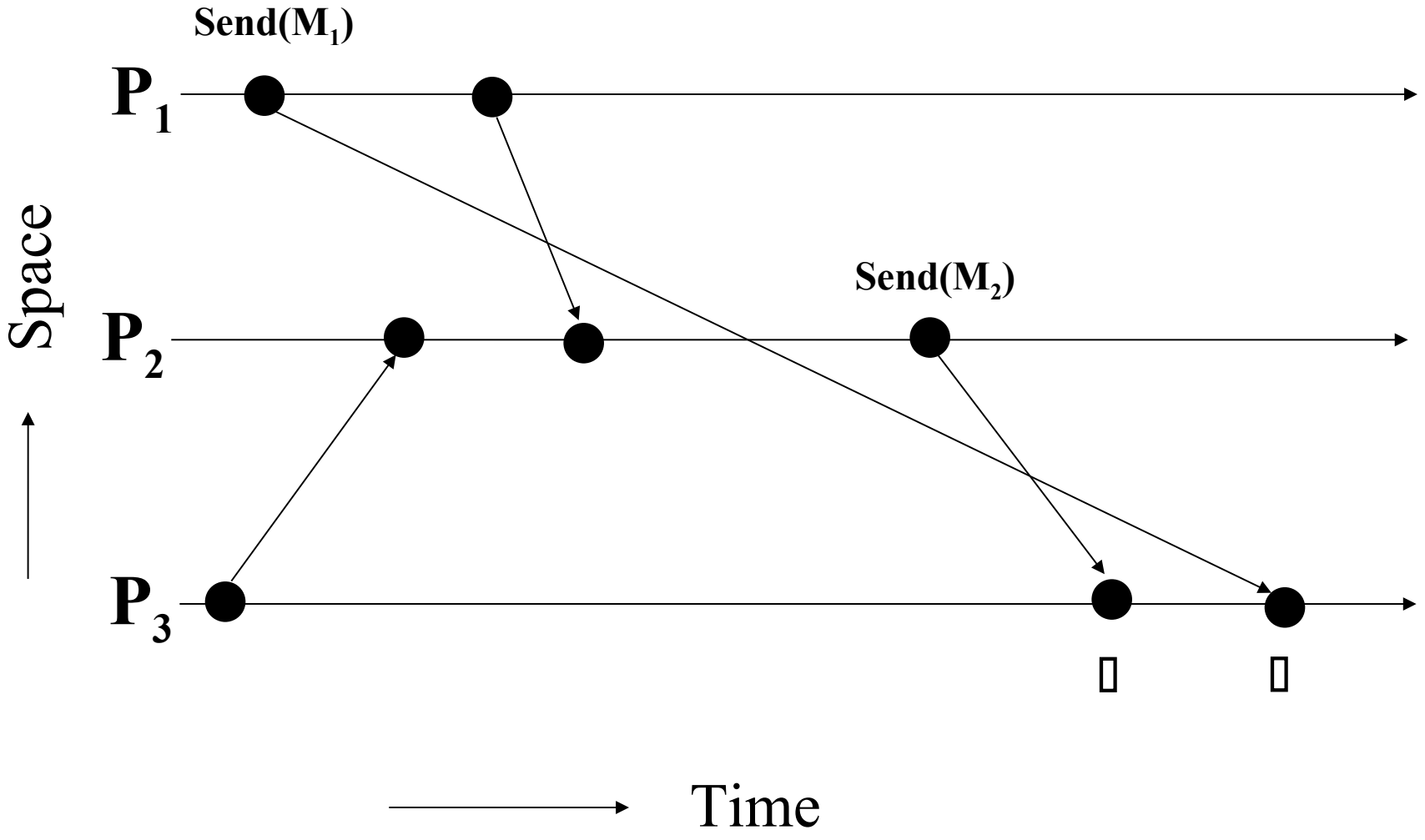
Event Ordering

- Since there is no common memory or clock, it is sometimes impossible to say which of two events occurred first.
- The *happened-before* relation is a partial ordering of events in distributed systems such that
  1. If *A* and *B* are events in the same process, and *A* was executed before *B*, then *A* => *B*.
  2. If *A* is the event of sending a message by one process and *B* is the event of receiving that by another process, then *A* => *B*.
  3. If *A* => *B* and *B* => *C*, then *A* => *C*.
- If two events *A* and *B* are not related by the => relation, then they are executed concurrently (no causal relationship)
- To obtain a global ordering of all the events, each event can be *time stamped* satisfying the requirement: for every pair of events A and B, if *A* => *B* then the time stamp of A is less than the time stamp of B. (Note that the converse need not be true.)

Causal Ordering of Messages

- If M1 is sent before M2, then every recepient of both messages must get M1 before M2
  - underlying network will not necessarily give this guarantee.
- Consider a replicated database system. Updates to the entries should be received in order!
- Basic idea -- buffer a later message

Causal Ordering of Messages



Global ordering

- How do we enforce the global ordering requirement in a distributed environment (without a common clock)?
  - For each process P_i, a logical clock LC_i assign a unique value to every event in that process.
  - If process P_i receives a message (event B) with time stamp t and $LC_i(B) < t$, then advance its clock so that $LC_i(B) = t+1$.
  - Use processor ids to break ties to create a total ordering.

Global State

- Due to absence of global clock, states are recorded at different times
- For global consistency, state of the communication channel should be the sequence of messages sent before the sender's state was recorded minus the messages received before the receivers state was recorded.
- Local states are defined in context of an application
  - a send is a part of the local state if it happened before the state was recorded. Ditto for a recv.

Global State

- A message causes an inconsistency if it was received, but not sent
- A collection of local states forms a global state
- This global state is consistent iff there are no pairwise inconsistency between local states.
- A message is in transit when it has been sent, but not received.
- The global state is transitless iff there are no local state pairs with messages in transit.
- Transitless + Consistent ➔ Strongly Consistent State

Clocks Synchronization and Reliability

- Always remember: the goal of clock synchronization is to minimize the difference between the accepted actual time and the time on a given client machine.
- When this is achieved, it can be said that in a given set of computers that synchronize that their clocks are more closely in sync.
- Even in this case of more accurate and more often corrected for clocks, developers of distributed systems should still be wary of relying on local clock time.

Clocks Synchronization and Reliability

- Because there are corrections going on, the time recorded for an event might have actually happened at a different time on another computer because of differing drift rates of those computer timers.
- Because we have correction of time does not mean that all machines agree on time, it just means they are much closer to each other on average.
- For some distributed systems, this may be sufficient, for others it may not be.

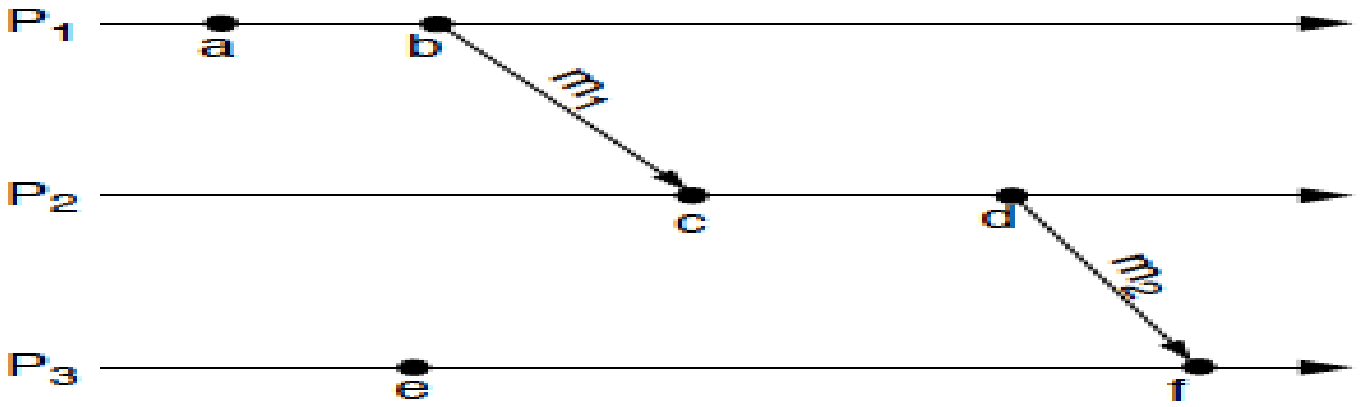
Event Ordering

- Since there is no common memory or clock, it is sometimes impossible to say which of two events occurred first.
- The *happened-before* relation is a partial ordering of events in distributed systems such that
  - If *A* and *B* are events in the same process, and *A* was executed before *B*, then *A* => *B*.
  - If *A* is the event of sending a message by one process and *B* is the event of receiving that by another process, then *A* => *B*.
  - If *A* => *B* and *B* => *C*, then *A* => *C*.
- If two events *A* and *B* are not related by the => relation, then they are executed concurrently (no causal relationship)
- To obtain a global ordering of all the events, each event can be *time stamped* satisfying the requirement: for every pair of events A and B, if *A* => *B* then the time stamp of A is less than the time stamp of B. (Note that the converse need not be true.)

Causality Example: Event Ordering

- If $a \rightarrow b$, we say that event *a* causally affects event *b*. The two events are causally related.

- There are events which are not related by the *happened-before* relation. If *both* a $\rightarrow$ e *and* e $\rightarrow$ a *are false*, then *a* and *e* are concurrent events; we write *a* || *e*.



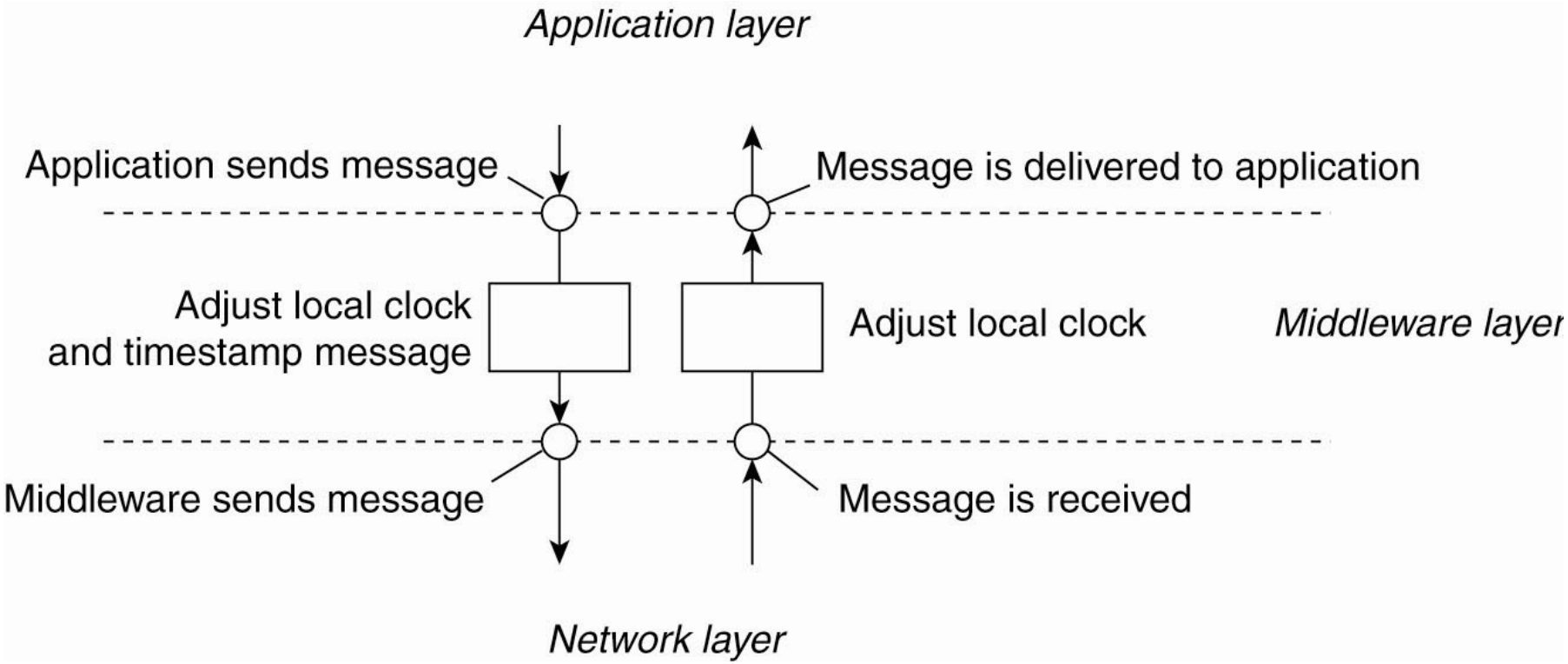
P1, *P2*, *P3*: processes;
a, *b*, *c*, *d*, *e*, *f*: events;

$a \rightarrow b$, $c \rightarrow d$, $e \rightarrow f$, $b \rightarrow c$, $d \rightarrow f$
$a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$, $b \rightarrow d$, $b \rightarrow f$, ...
$a \parallel e$, $c \parallel e$, ...

Causality and Logical Time

- **Constraint:** The update ordering must respect potential causality.
  - Communication patterns establish a happened-before order on events, which tells us when ordering might matter.
  - Event e_1 happened-before e_2
    - Iff (if and only of) e_1 could possibly have affected the generation of e_2: we say that $e_1 < e_2$.
    - $e_1 < e_2$ iff e_1 was "known" when e_2 occurred.
    - Events e_1 and e_2 are potentially causally related.

Lamport's Logical Clocks (3)

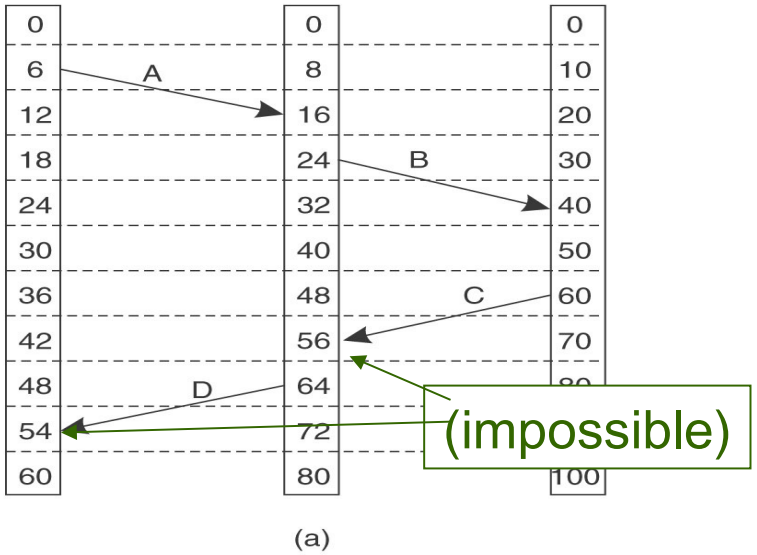


Lamport's Algorithm

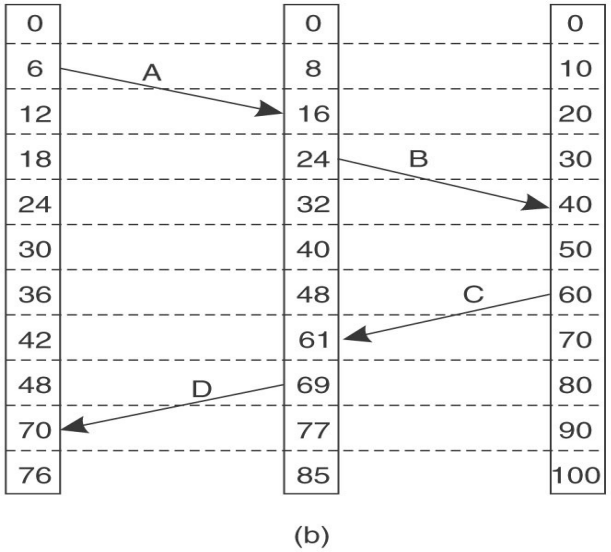
- Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically. A Lamport logical clock is a incrementing software counter maintained in each process.
- Algorithm follows:
  1. A process increments its counter before each event in that process;
  2. When a process sends a message, it includes its counter value with the message;
  3. On receiving a message, the receiver process sets its counter to be greater than the maximum of its own value and the received value before it considers the message received.
- 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.

Example: Lamport's Algorithm

- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's Algorithm corrects the clock.



(a)



(b)

- Note: ts(A) < ts(B) does not imply A happened before B.

Lamport's Considerations

- For every two events a and b occurring in the same process, and C(x) being the **timestamp** for a certain event x, it is necessary that C(a) never equals C(b).
- Therefore it is necessary that:
  1. The logical clock be set so that there is minimum of one clock "tick" (increment of the counter) between events and ;
  2. In a multiprocess or multithreaded environment, it might be necessary to attach the process ID (PID) or any other unique ID to the timestamp so that it is possible to differentiate between events and which may occur simultaneously in different processes.

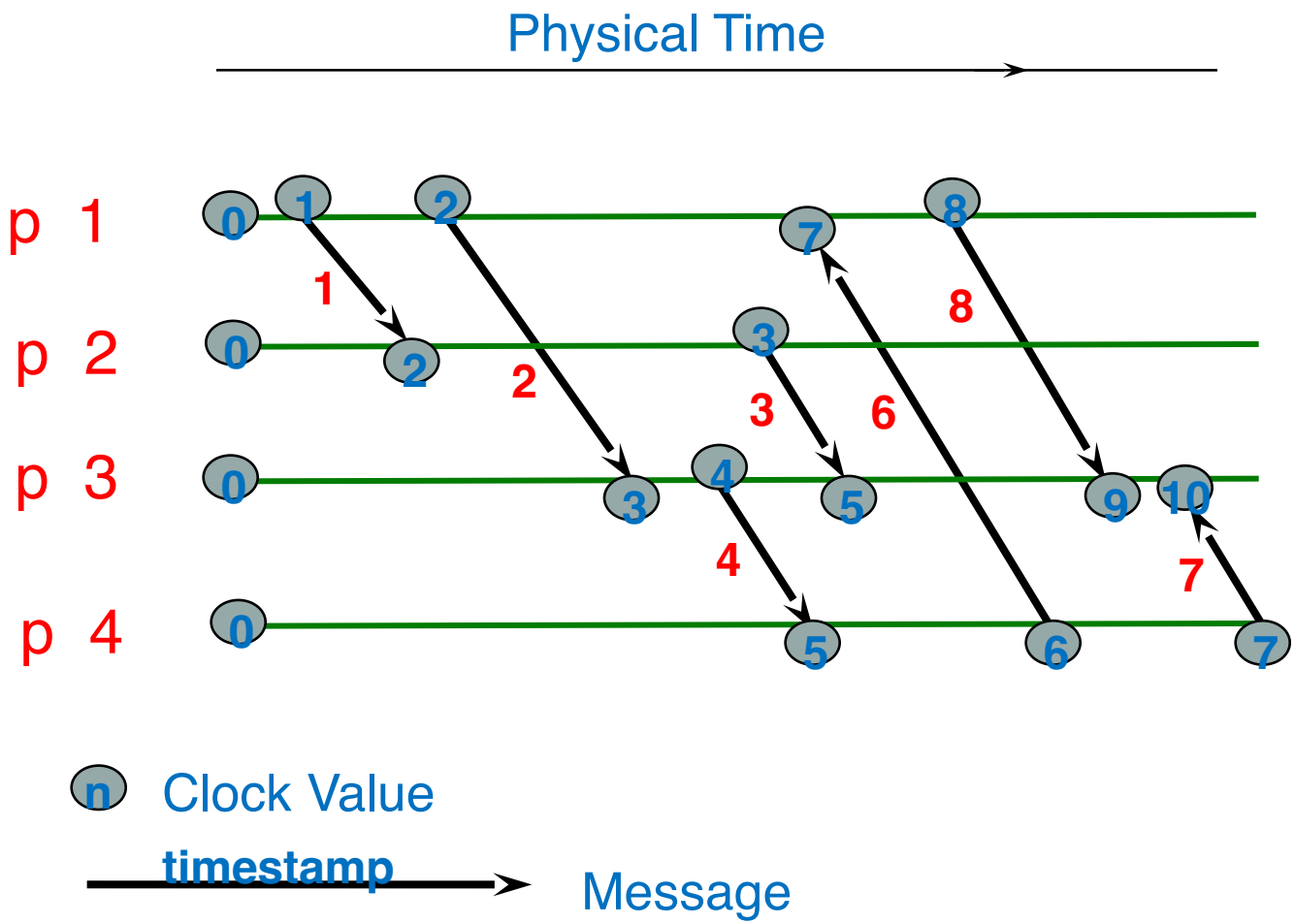
Lamport's logical clock in distributed systems

- In a distributed system, it is not possible in practice to synchronize time across entities (typically thought of as processes) within the system; hence, the entities can use the concept of a logical clock based on the events through which they communicate.
- If two entities do not exchange any messages, then they probably do not need to share a common clock; events occurring on those entities are termed as **concurrent events**.
- Among the processes on the same local machine we can order the events based on the local clock of the system.
- When two entities communicate by message passing, then the send event is said to 'happen before' the receive event, and the logical order can be established among the events.
- A distributed system is said to have partial order if we can have a partial order relationship among the events in the system. If 'totality', i.e., causal relationship among all events in the system can be established, then the system is said to have total order.

Update Ordering

- **Problem**: how to ensure that all sites recognize a fixed order on updates, even if updates are delivered out of order?
- **Solution:** Assign timestamps to updates at their accepting site, and order them by source timestamp at the receiver.
  - Assign nodes unique IDs: break ties with the origin node ID.
  - Problem: What (if different) ordering exists between updates accepted by different sites?
    - Comparing physical timestamps is arbitrary: physical clocks drift.
    - Even a protocol to maintain loosely synchronized physical clocks cannot assign a meaningful ordering to events that occurred at "almost exactly the same time".

Lamport Logical Time Example



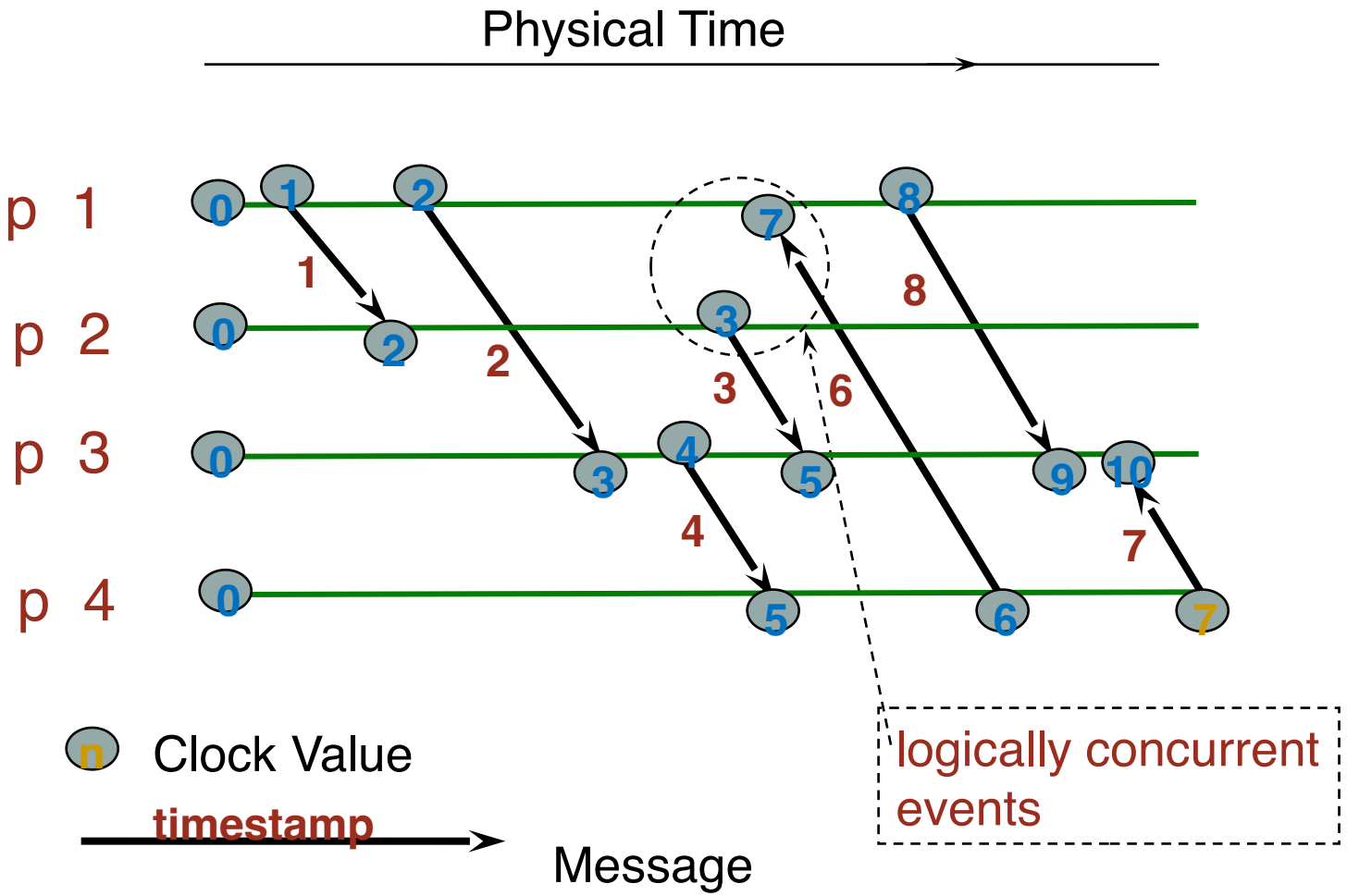
Problem with Lamport Logical Clock

- Let *timestamp(a)* be the Lamport logical clock timestamp

- *a => b => timestamp(a) < timestamp(b)*
- (if *a* happens before *b*, then *Lamport_timestamp(a) < Lamport_timestamp(b)*)

- *timestamp(a) < timestamp(b)* => *a => b*
- *(*If *Lamport_timestamp(a) < Lamport_timestamp(b),* it does NOT imply that *a* happens before *b*

Example



Note: Lamport Timestamps: 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in 'happen-before' relation.(we can not say 3 => 7)
Cannot determine whether two events are causally related from timestamps

Limitations of Lamport's Logical Clocks

- Lamport's logical clocks lead to a situation where all events in a distributed system are totally ordered. That is, if A =>B, then we can say C(A) < C(B).
- Unfortunately, with Lamport's clocks, nothing can be said about the actual time of A and B. If the logical clock says A =>B, that does not mean in reality that A actually happened before B in terms of real time.
- The problem with Lamport clocks is that they do not capture causality.
- If we know that A => C and B => C we cannot say which action initiated C.
- This kind of information can be important when trying to replay events in a distributed system (such as when trying to recover after a crash).
- The theory goes that if one node goes down, if we know the causal relationships between messages, then we can replay those messages and respect the causal relationship to get that node back up to the state it needs to be in.

Logical Clock Synchronization

1. Lamport's timestamps (Algorithm)
2. **Vector clocks (Algorithm)**

Vector Clocks

- Vector clocks is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.

- Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock.

- A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process

Motivation for Vector Clocks

- Logical clocks induce an order consistent with causality, but
  - the converse of the clock condition does not hold: it may be that $LC(e_1) < LC(e_2)$ even if e_1 and e_2 are concurrent.
    - If A could know anything B knows, then it must be $LC_A > LC_B$.
    - But if $LC_A > LC_B$ then this doesn't make it so; i.e., "false positives".
    - Concurrent updates may be ordered unnecessarily.
- We need a clock mechanism that is necessary and sufficient in capturing causality.

Vector Clocks

- *Vector clocks* (AKA *vector timestamps* or *version vectors*) are a more detailed representation of what a site might know.

1. In a system with ***N*** nodes, each site keeps a vector timestamp *TS[N]* as well as a logical clock *LC*.

   $TS_i[i]$ at site *i* is the most recent value of site *j*'s logical clock that site *i* "heard about".

   $TS_i[i] = LC_i$: each site *i* keeps its own *LC* in *TS[i]*.

2. When site *i* generates a new event, it increments its logical clock.

   $TS_i[i] = TS_i[i] + 1$

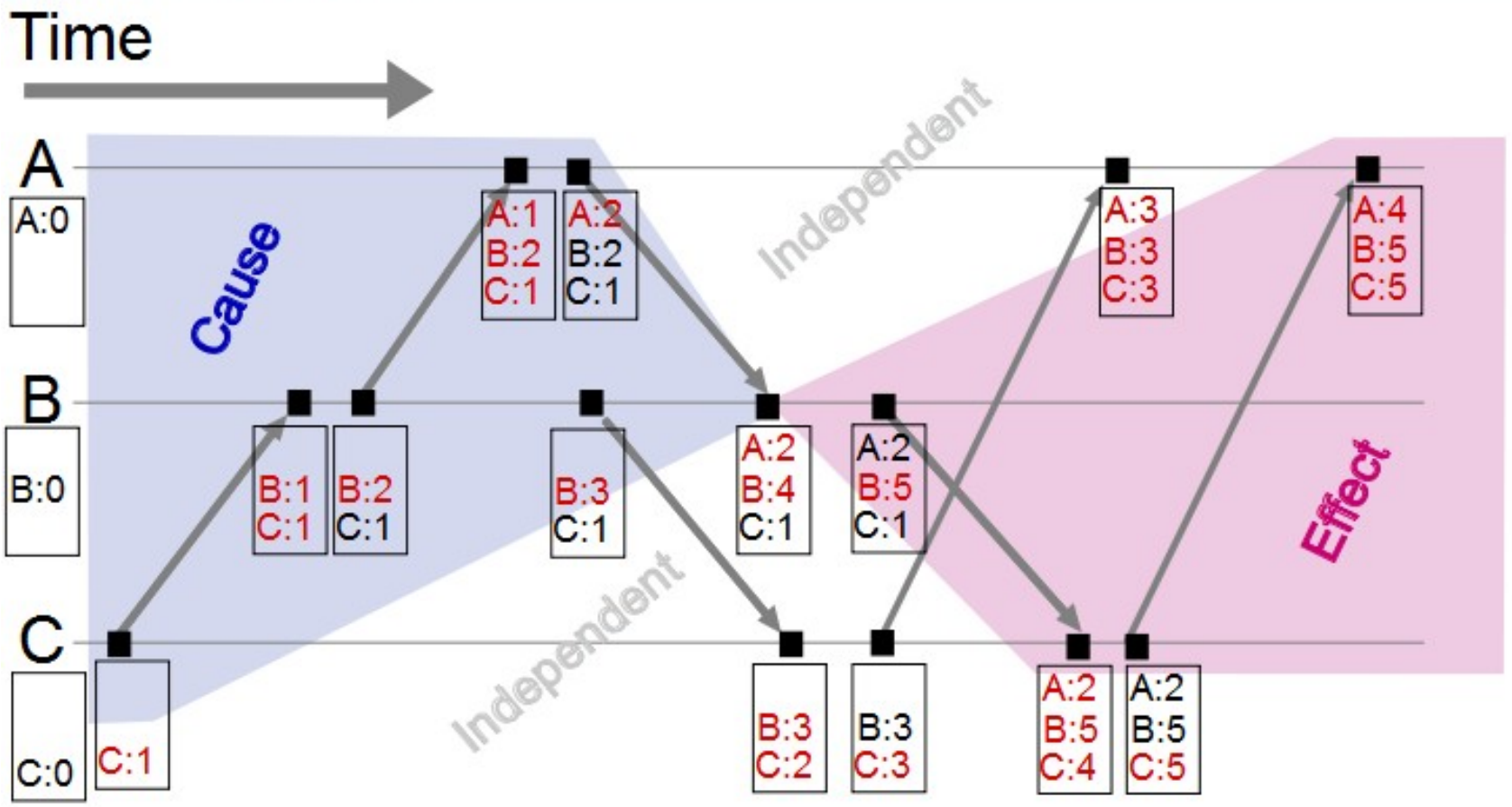
3. A site *r* observing an event (e.g., receiving a message) from site *s* sets its TS_r to the pairwise maximum of TS_s and TS_r.

   For each site *i*, $TS_r[i] = \mathbf{max}\ (TS_r[i], TS_s[i])$

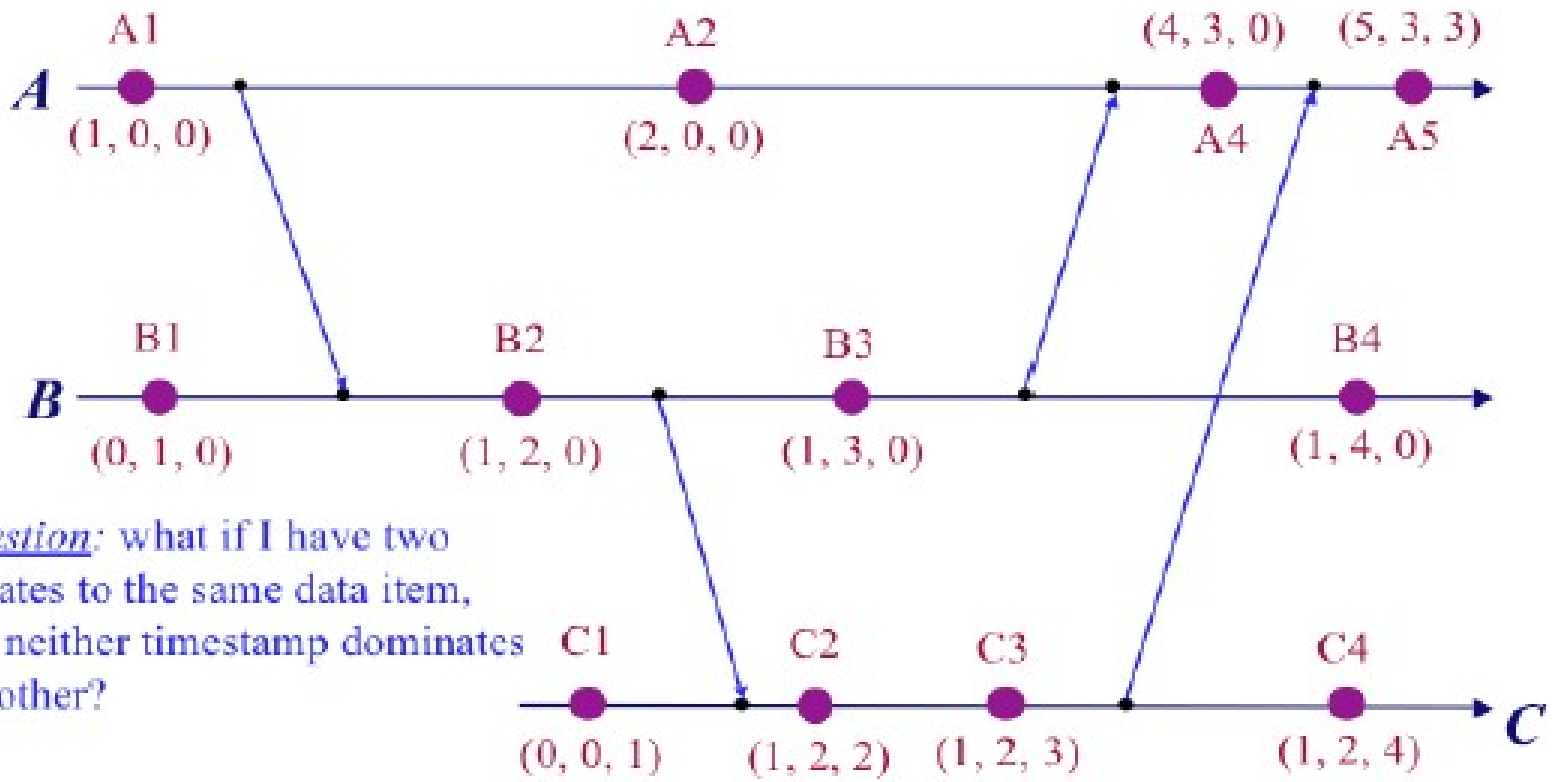
Vector Clocks: Algorithm(Simple Version)

1. Initially all clocks are zero.
2. Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
3. Each time a process prepares to send a message, it increments its own logical clock in the vector by one and then sends its entire vector along with the message.
4. Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Vector Clock: Example



Vector Clocks: Example

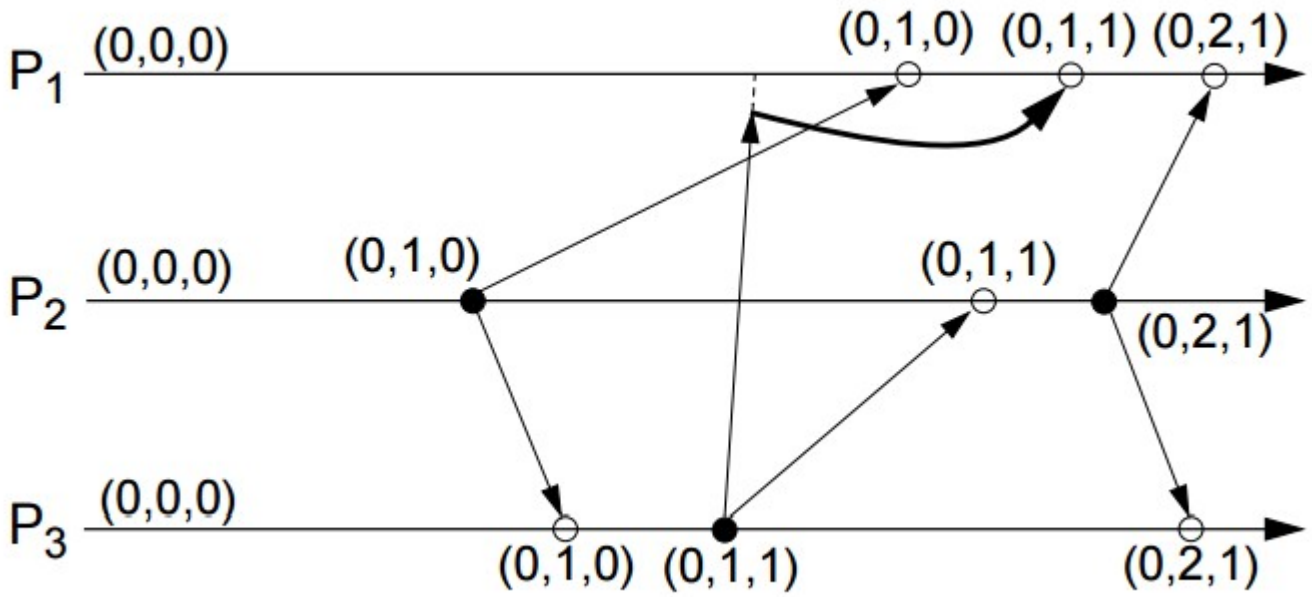


Question: what if I have two updates to the same data item, and neither timestamp dominates the other?

Causal Ordering of Message using Vector Clock

- Causal ordering of messages
  - maintaining the same causal order of message receive events as message sent
  - that is: if Send (M1) → Send(M2) and Receive(M1) and Receive (M2) are on the same process than Receive(M1) → Receive(M2)
- causal ordering is useful, for example for replicated databases
- two algorithms using VC
  - ***Birman-Schiper-Stephenson (BSS) causal ordering of broadcasts***
  - ***Schiper-Eggli-Sandoz (SES) causal ordering of regular messages***
- basic idea – use VC to delay out of order message delivery

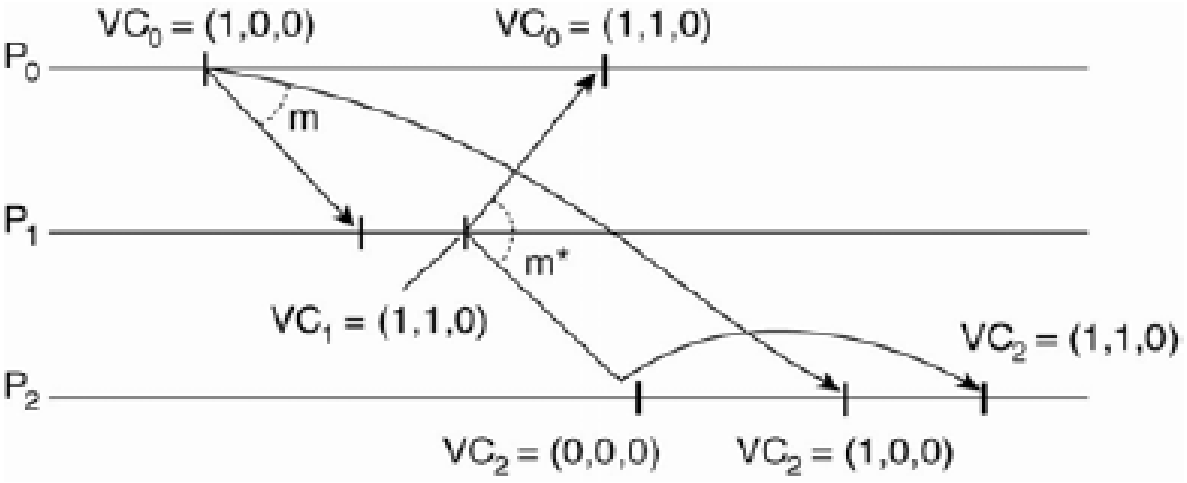
Causal Ordering of Message using Vector Clock



Vector Clock and Causality

- Vector clocks induce an order that exactly reflects causality.
  - Tag each event e with current TS vector at originating site.
    - vector timestamp TS(e)
  - e_1 **happened-before** e_2 if and only if TS(e_2) **dominates** TS(e_1)
    - $e_1 < e_2$ iff TS(e_1)[i] <= TS(e_2)[i] for each site i
    - "Every event or update visible when e_1 occurred was also visible when e_2 occurred."
  - Vector timestamps allow us to ask if two events are concurrent, or if one happened-before the other.
    - If $e_1 < e_2$ then LC(e_1) < LC(e_2) and TS(e_2) dominates TS(e_1).
    - "If TS(e_2) does not dominate TS(e_1) then it is not true that $e_1 < e_2$."

Causally-Ordered Multicasting



- **Causally-ordered multicasting**: a message is delivered only if all messages that causally precede it have also been delivered
- Implementing causally-ordered multicasting:

  If P_j receives a message from P_i, delay delivery of the message until

  1. ts(m) [i] == VC_j[i] + 1 (m is the next expected message from P_i)
  2. ts(m) [k] <= VC_j[k] for all k ≠ i (P_j has seen all messages seen by P_i when it sent m)

Summary

- We can say if an event *a* has a timestamp *ts(a)*, then *ts(a)[i]−1* denotes the number of events processed at P_i that causally precede *a*
- This means that when P_j receives a message from P_i with timestamp *ts(m)*, it knows about the number of events that occured at P_i that causally preceded the sending of *m*
- Even more importantly, P_j has been told how many events in **other** processes have taken place before P_i sent message *m*.
- So, this means we could achieve a very important capability in a distributed system: we can ensure that a message is delivered only if all messages that causally precede it have also been received as well.
- We can use this capability to build a truly distributed dataflow graph with dependencies without having a centralized coordinating process.

Synchronization, Coordination and Agreement

3.1 Time Synchronization

3.1.1 Time and time synchronization

3.1.2 Physical Clock Synchronization

3.1.2.1 Berkeley Algorithm

3.1.2.2 Cristian's Algorithm

3.1.2.3 Network Time Protocol (NTP)

3.1.3 Logical Clocks Synchronization

3.1.3.1 Events and ordering

3.1.3.2 Partial, causal and total ordering

3.1.3.3 Global State and State Recording

3.1.3.4 Lamport logical clock

3.1.3.5 Vector clock

3.2 Distributed Co-ordination

3.2.1 Distributed mutual exclusion

3.2.2 Mutual Exclusion Algorithms

3.2.2.1 Central Coordinator Algorithm

3.2.2.2 Token Ring Algorithm

3.2.2.3 Lamport's Algorithm

3.2.2.4 Ricart-Agrawala
(Non-Token based & token based)

3.2.3 Distrbuted Leader Election

3.2.3.1 Bully algorithm

3.2.3.2 Ring Algorithms