

Lecture: 05

Size of vector clock, Matrix clocks, Virtual time and Physical clock synchronization



Rajiv Misra

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture we have discussed the models of distributed computation, and presented the idea of causality & logical time that was proposed by Lamport in 1978 in an attempt to order events in distributed systems.
- We have discussed two systems of logical clocks, namely, scalar and vector clocks to capture causality between events of a distributed computation.

Preface

Content of this Lecture:

In this lecture we will discuss about:

- Size of vector clocks
- Matrix clocks
- Virtual time and
- Physical clock synchronization

Size of vector clocks

- An important question to ask is whether vector clocks of size n are necessary in a computation consisting of n processes. To answer this, we examine the usage of vector clocks.
- A vector clock provides the latest known local time at each other process. If this information in the clock is to be used to explicitly track the progress at every other process, then a vector clock of size n is necessary.
- A popular use of vector clocks is to determine the causality between a pair of events. Given any events e and f , the test for $e \prec f$ if and only if $T(e) < T(f)$, which requires a comparison of the vector clocks of e and f .
- Although it appears that the clock of size n is necessary, that is not quite accurate. It can be shown that a size equal to the dimension of the partial order $(E, \prec)$ is necessary, where the upper bound on this dimension is n .

Size is equal to dimension of Partial Order

Definitions:

- To understand this result on the size of clocks for determining causality between a pair of events, we first introduce some definitions:
- A **linear extension** of a partial order $(E, <)$ is a linear ordering of E that is consistent with the partial order, i.e. if two events are ordered in the partial order, they are also ordered in the linear order.
- A linear extension can be viewed as projecting all the events from the different processes on a single time axis.
- The **dimension** of a partial order is the minimum number of linear extensions, whose intersection gives exactly the partial order.

Contd...

- However, the linear order will necessarily introduce ordering between each pair of events, and some of these orderings are not in the partial order. Also observe that different linear extensions are possible in general.
- Let $\mathcal{P}$ denote the **set of tuples** in the partial order defined by the causality relation; so there is a **tuple (e, f)** in $\mathcal{P}$ for each pair of **events e and f** such that **e < f**. Let $\mathcal{L}_1, \mathcal{L}_2, \dots$ denote the **sets of tuples in different linear extensions** of this partial order.
- The set $\mathcal{P}$ is contained in the set obtained by taking the intersection of any such collection of linear extensions $\mathcal{L}_1, \mathcal{L}_2, \dots$. This is because each $\mathcal{L}_i$ must contain all the tuples, i.e., causality dependencies, that are in $\mathcal{P}$.

1. Client-Server Interaction

- Consider a client–server interaction between a pair of processes. Queries to the server and responses to the client occur in strict alternating sequences.
- Although $n=2$, all the events are strictly ordered, and there is only one linear order of all the events that is consistent with the “partial” order.
- Hence the dimension of this “partial order” is 1. A scalar clock such as one implemented by Lamport’s scalar clock rules is adequate to determine $e < f$ for any events e and f in this execution.

2. Concurrent Send-Receive

- Now consider an execution on processes P_1 and P_2 such that each sends a message to the other before receiving the other's message.
- The two send events are concurrent, as are the two receive events. To determine the causality between the send events or between the receive events, it is not sufficient to use a single integer; a vector clock of size $n = 2$ is necessary.
- This execution exhibits the graphical property called a **crown**, wherein there are some messages $m_0, \dots, m_{n-1}$ such that **$\text{Send}(m_i) < \text{Receive}(m_{i+1 \bmod (n-1)})$** for all i from 0 to $n-1$. A crown of n messages has dimension n .

3. Complex Execution

- For a complex execution, it is not straightforward to determine the dimension of the partial order. Figure 5.1 shows an execution involving four processes. However, the dimension of this partial order is two.
- To see this informally, consider the **longest chain** $\langle a, b, d, g, h, i, j \rangle$. There are events outside this chain that can yield multiple linear extensions. Hence, the dimension is more than 1.
- The right side of Figure 5.1 shows the earliest possible and the latest possible occurrences of the events not in this chain, with respect to the events in this chain.

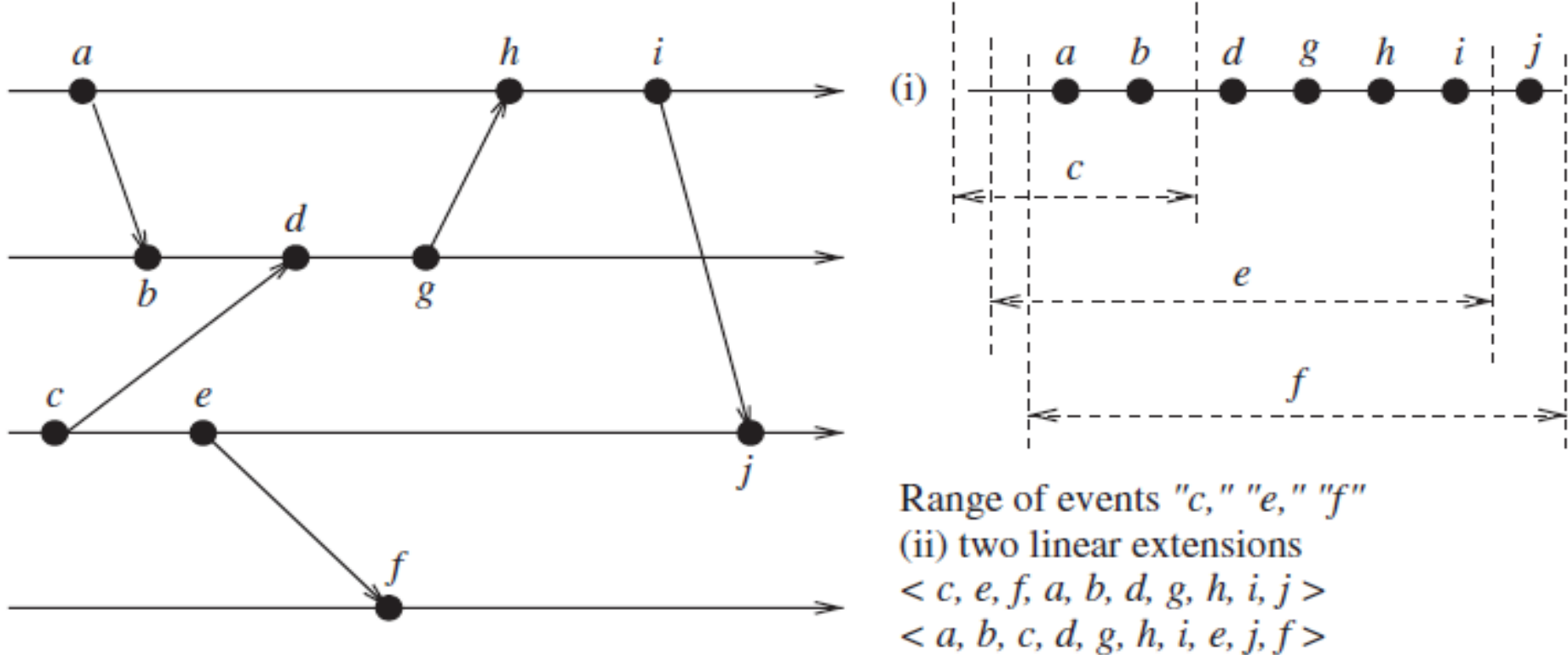


Figure 5.1 Example illustrating dimension of an execution $(E, <)$.

For $n = 4$ processes, the dimension is 2.

3. Complex Execution

Let $\mathcal{L}_1$ be $\langle c, e, f, a, b, d, g, h, i, j \rangle$ which contains the following tuples that are not in $\mathcal{P}$:

$(c, a), (c, b), (c, d), (c, g), (c, h), (c, i), (c, j)$
 $(e, a), (e, b), (e, d), (e, g), (e, h), (e, i), (e, j)$
 $(f, a), (f, b), (f, d), (f, g), (f, h), (f, i), (f, j)$

Let $\mathcal{L}_2$ be $\langle a, b, c, d, g, h, i, e, j, f \rangle$, which contains the following tuples not in $\mathcal{P}$:

$(a, c), (b, c), (c, d), (c, g), (c, h), (c, i), (c, j)$
 $(a, e), (b, e), (d, e), (g, e), (h, e), (i, e), (e, j)$
 $(a, f), (b, f), (d, f), (g, f), (h, f), (i, f), (j, f)$

- Further, observe that $(\mathcal{L}_1 \setminus \mathcal{P}) \cap \mathcal{L}_2 = \emptyset$ and $(\mathcal{L}_2 \setminus \mathcal{P}) \cap \mathcal{L}_1 = \emptyset$.
Hence, $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathcal{P}$ and the dimension of the execution is 2 as these two linear extensions are enough to generate $\mathcal{P}$.

Matrix Time

- In a system of matrix clocks, the time is represented by a set of $n \times n$ matrices of non-negative integers.
- A process p_i maintains a matrix $mt_i[1..n, 1..n]$ where,
 - $mt_i[i, i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i .
 - $mt_i[i, j]$ denotes the latest knowledge that process p_i has about the local logical clock, $mt_j[j, j]$, of process p_j .
 - $mt_i[j, k]$ represents the knowledge that process p_i has about the latest knowledge that p_j has about the local logical clock, $mt_k[k, k]$, of p_k .
- The entire matrix mt_i denotes p_i 's local view of the global logical time.

Matrix Time

Process p_i uses the following rules $R1$ and $R2$ to update its clock:

$R1$: Before executing an event, process p_i updates its local logical time as follows:

$$mt_i[i, i] := mt_i[i, i] + d \quad (d > 0)$$

$R2$: Each message m is piggybacked with matrix time mt . When p_i receives such a message (m, mt) from a process p_j , p_i executes the following sequence of actions:

- Update its global logical time as follows:

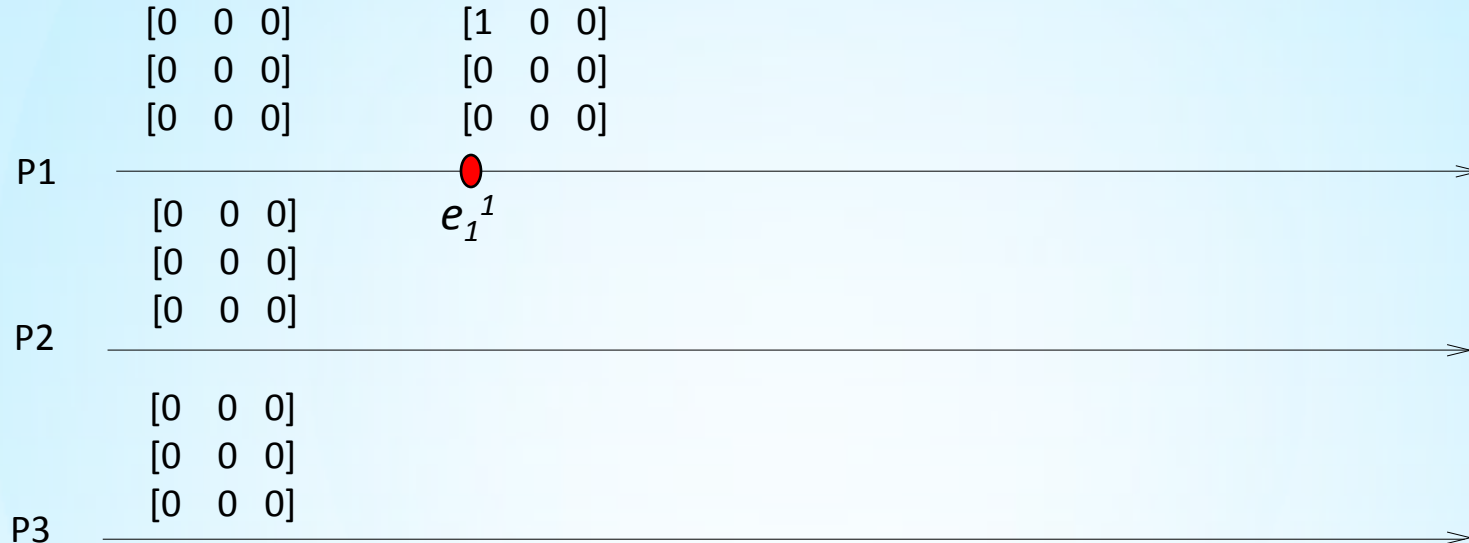
$$(a) \ 1 \leq k \leq n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$$

(That is, update its row $mt_i[i, *]$ with the p_j 's row in the received timestamp, mt)

$$(b) \ 1 \leq k, l \leq n : mt_i[k, l] := \max(mt_i[k, l], mt[k, l])$$

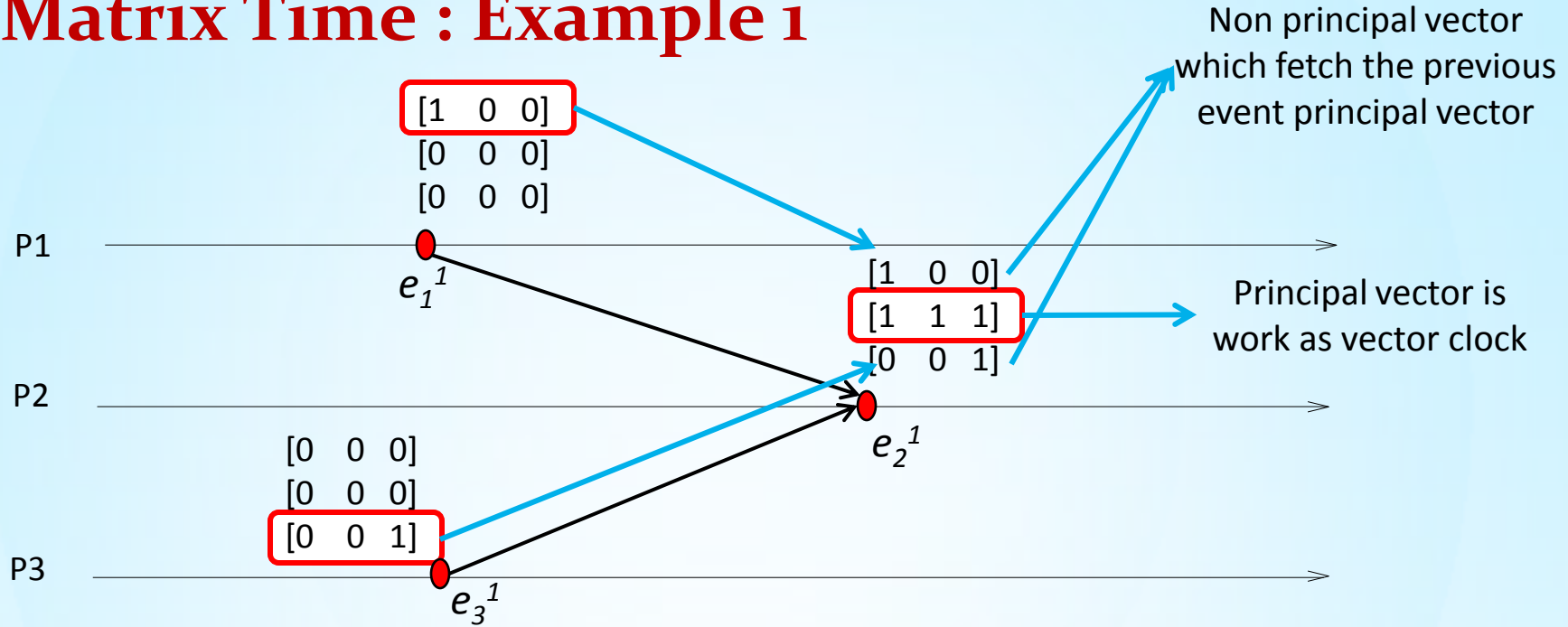
- Execute $R1$
- Deliver message m

Matrix Time : Example 1



Initial Form of Matrix Clock

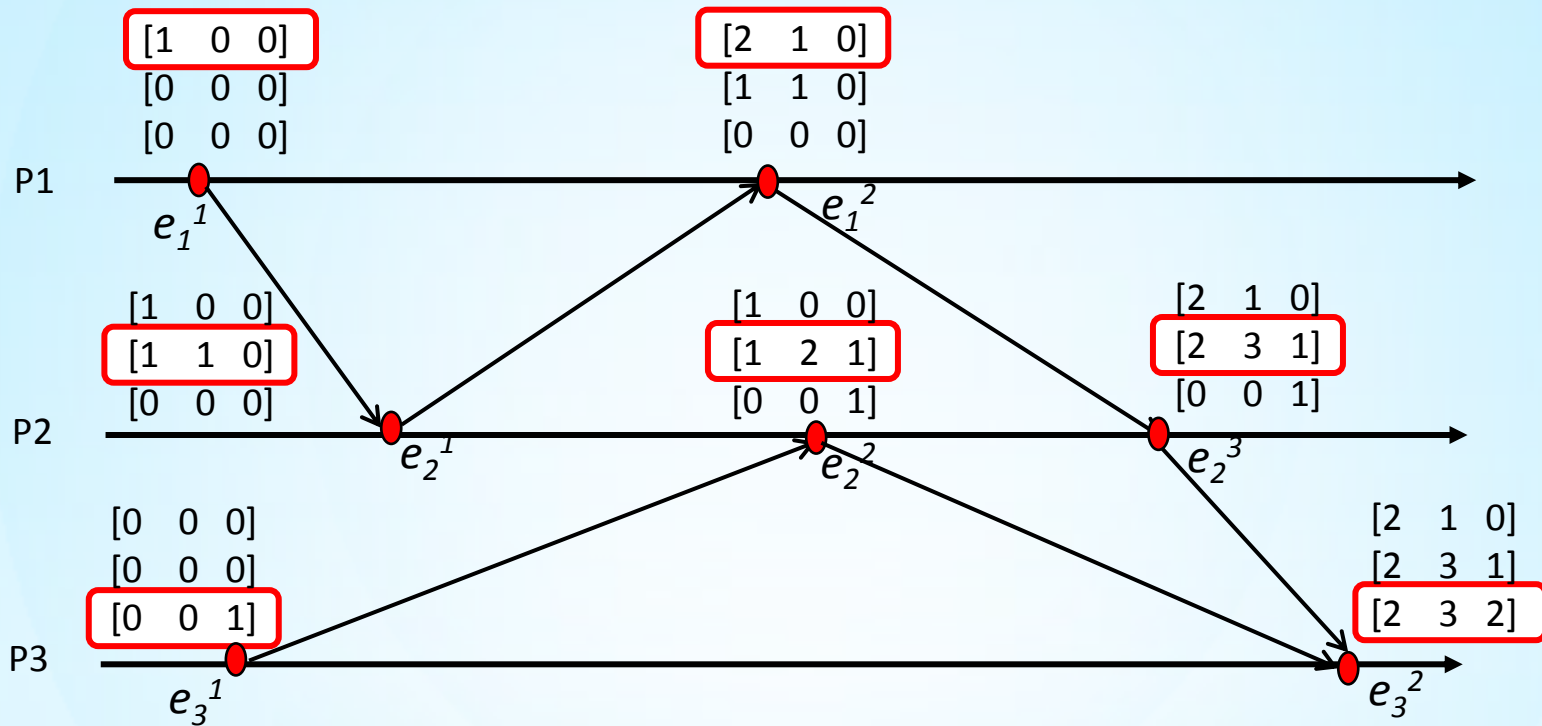
Matrix Time : Example 1



Working of principal and non-principal vector in matrix clock

Matrix Time : Example 2

 → Principal vector



Working of principal and non-principal vector in matrix clock

Basic Properties

- Vector $mt_i[j, .]$ contains all the properties of vector clocks. In addition, matrix clocks have the following property:
- **$\min_k (mt_i[k, l]) \geq t \Rightarrow$ process p_i** knows that every other process p_k knows that p_i 's local time has progressed till t .
- If this is true, it is clear that process p_i knows that all other processes know that p_i will never send information with a local time $\leq t$.
- In many applications, this implies that processes will no longer require from p_i certain information and can use this fact to discard obsolete information.
- If d is always 1 in the rule $R1$, then $mt_i[k, l]$ denotes the number of events occurred at p_i and known by p_k as far as p_i 's knowledge is concerned.

Virtual Time

- Virtual time system is a paradigm for organizing and synchronizing distributed systems.
- This section provides description of virtual time and its implementation using the **Time Warp mechanism**.
- The implementation of virtual time using Time Warp mechanism works on the basis of an optimistic assumption.
- Time Warp relies on the **general lookahead-rollback mechanism** where each process executes without regard to other processes having synchronization conflicts.

Virtual Time

- If a conflict is discovered, the offending processes are rolled back to the time just before the conflict and executed forward along the revised path.
- Detection of conflicts and rollbacks are transparent to users.
- The implementation of Virtual Time using Time Warp mechanism makes the following optimistic assumption: synchronization conflicts and thus rollbacks generally occurs rarely.

Virtual Time

- “Virtual time is a global, one dimensional, temporal coordinate system on a distributed computation to measure the computational progress and to define synchronization.”
- A virtual time system is a distributed system executing in coordination with an imaginary virtual clock that uses virtual time.
- Virtual times are real values that are totally ordered by the less than relation, “ $<$ ”.
- Virtual time is implemented a collection of several loosely synchronized local virtual clocks.
- These local virtual clocks move forward to higher virtual times; however, occasionally they move backwards.

Virtual Time: *Definition*

- Processes run concurrently and communicate with each other by exchanging messages.
- Every message is characterized by four values:

a) Name of the sender

b) Virtual send time

c) Name of the receiver

d) Virtual receive time

- Virtual send time is the virtual time at the sender when the message is sent, whereas virtual receive time specifies the virtual time when the message must be received (and processed) by the receiver.

Virtual Time: *Definition*

- A problem arises when a message arrives at process late, that is, the virtual receive time of the message is less than the local virtual time at the receiver process when the message arrives.
- Virtual time systems are subject to **two semantic rules** similar to Lamport's clock conditions:
 - Rule 1:** Virtual send time of each message $<$ Virtual receive time of that message.
 - Rule 2:** Virtual time of each event in a process $<$ Virtual time of next event in that process.
- The above two rules imply that a process sends all messages in increasing order of virtual send time and a process receives (and processes) all messages in the increasing order of virtual receive time.

Virtual Time: *Definition*

- It is important an event that causes another should be completely executed before the caused event can be processed.
- The constraint in the implementation of virtual time can be stated as follows:

“If an event A causes event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts”.

Virtual Time: *Definition*

- If event A has an earlier virtual time than event B, we need execute A before B provided there is no causal chain from A to B.
- Better performance can be achieved by scheduling A concurrently with B or scheduling A after B.
- If A and B have exactly the same virtual time coordinate, then there is no restriction on the order of their scheduling.
- If A and B are distinct events, they will have different virtual space coordinates (since they occur at different processes) and neither will be a cause for the other.
- To sum it up, events with virtual time $< 't'$ complete before the starting of events at time $'t'$ and events with virtual time $> 't'$ will start only after events at time $'t'$ are complete.

Characteristics of Virtual Time

- Virtual time systems are not all isomorphic; it may be either discrete or continuous.
- Virtual time may be only partially ordered.
- Virtual time may be related to real time or may be independent of it.
- Virtual time systems may be visible to programmers and manipulated explicitly as values, or hidden and manipulated implicitly according to some system-defined discipline
- Virtual times associated with events may be explicitly calculated by user programs or they may be assigned by fixed rules.

Comparison with Lamport's Logical Clocks

- In Lamport's logical clock, an artificial clock is created one for each process with unique labels from a totally ordered set in a manner consistent with partial order.
- In virtual time, the reverse of the above is done by assuming that every event is labeled with a clock value from a totally ordered virtual time scale satisfying Lamport's clock conditions.
- Thus the *Time Warp mechanism* is an inverse of Lamport's scheme.
- In Lamport's scheme, all clocks are conservatively maintained so that they never violate causality.
- A process advances its clock as soon as it learns of new causal dependency. In the virtual time, clocks are optimistically advanced and corrective actions are taken whenever a violation is detected.
- Lamport's initial idea brought about the concept of virtual time but the model failed to preserve causal independence.

Time Warp Mechanism

- In the implementation of virtual time using Time Warp mechanism, virtual receive time of message is considered as its timestamp.
- The necessary and sufficient conditions for the correct implementation of virtual time are that each process must handle incoming messages in *timestamp* order.
- This is highly undesirable and restrictive because process speeds and message delays are likely to highly variable.
- It natural for some processes to get ahead in virtual time of other processes.

Time Warp Mechanism

- It is impossible for a process on the basis of local information alone to block and wait for the message with the next timestamp.
- It is always possible that a message with earlier timestamp arrives later.
- So, when a process executes a message, it is very difficult for it determine whether a message with an earlier timestamp will arrive later.
- This is the central problem in virtual time that is solved by the Time Warp mechanism.
- The Time warp mechanism assumes that message communication is reliable, and messages may not be delivered in FIFO order.

Time Warp Mechanism

Time Warp mechanism consists of two major parts:

(i) local control mechanism and (ii) global control mechanism

(i) The **local control mechanism** ensures that events are executed and messages are processed in the correct order.

(ii) The **global control mechanism** takes care of global issues such as global progress, termination detection, I/O error handling, flow control, etc.

Physical Clock Synchronization: NTP

- In centralized systems, there is only single clock. A process gets the time by simply issuing a system call to the kernel.
- In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time.
- These clocks can easily drift seconds per day, accumulating significant errors over time.
- Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start.
- This clearly poses serious problems to applications that depend on a synchronized notion of time.

Physical Clock Synchronization: NTP

For most applications and algorithms that run in a distributed system, we need to know time in one or more of the following contexts:

- **The time of the day** at which an event happened on a specific machine in the network.
- **The time interval** between two events that happened on different machines in the network.
- **The relative ordering of events** that happened on different machines in the network.
- Unless the clocks in each machine have a common notion of time, time-based queries cannot be answered.
- **Clock synchronization** has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.

Physical Clock Synchronization: NTP

- Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.
- Due to different clocks rates, the clocks at various sites may diverge with time and periodically a clock synchronization must be performed to correct this clock skew in distributed systems.
- Clocks are synchronized to an accurate real-time standard like **UTC (Universal Coordinated Time)**.
- Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed ***physical clocks***.

Clock Inaccuracies

- Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- However, due to the clock inaccuracy discussed above, **a timer (clock)** is said to be working within its specification if (where **constant ρ** is the **maximum skew rate** specified by the manufacturer.)

$$1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho \quad (1)$$

- Figure 5.3 illustrates the behavior of **fast, slow, and perfect clocks** with respect to UTC.

Clock Inaccuracies

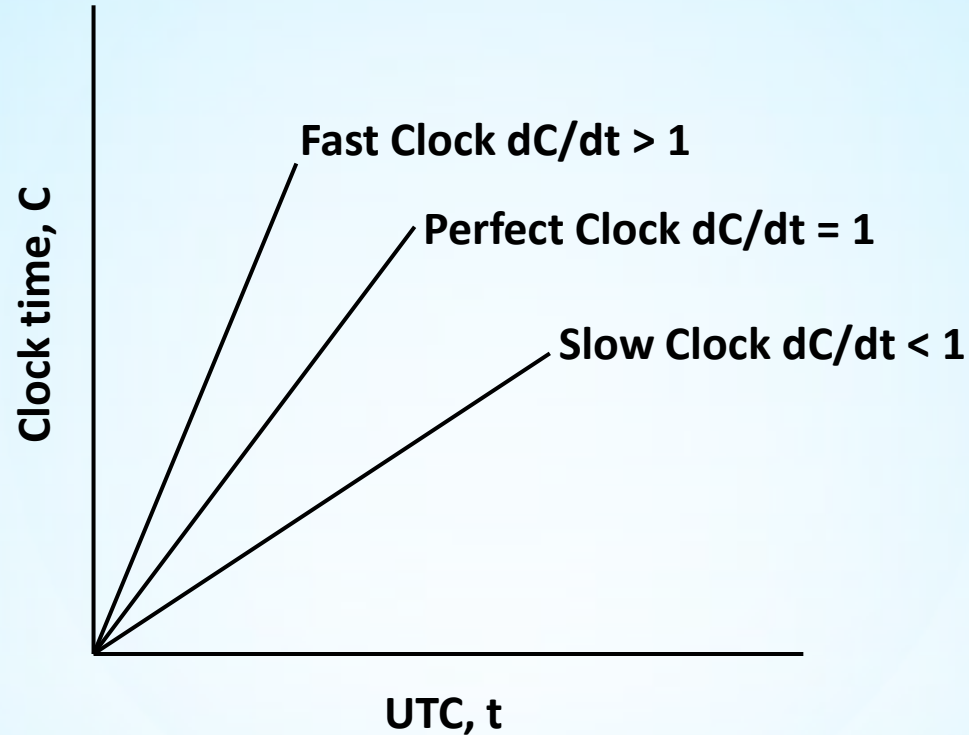


Figure 5.3: The behavior of fast, slow, and perfect clocks with respect to UTC.

Offset delay estimation method

- The **Network Time Protocol (NTP)** which is widely used for clock synchronization on the Internet uses the **Offset Delay Estimation** method.
- The design of NTP involves a **hierarchical tree** of time servers:
 - (i) The primary server at the root synchronizes with the UTC.
 - (ii) The next level contains secondary servers, which act as a backup to the primary server.
 - (iii) At the lowest level is the synchronization subnet which has the clients.

Clock offset and delay estimation:

- In practice, a source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes.
- This protocol employs a common practice of performing several trials and chooses the trial with the minimum delay.
- Figure 5.4 shows how **NTP timestamps** are numbered and exchanged between *peers A and B*.
- Let T_1, T_2, T_3, T_4 be the values of the *four most recent timestamps* as shown. Assume **clocks A and B are stable** and **running at the same speed**.

Clock offset and delay estimation:

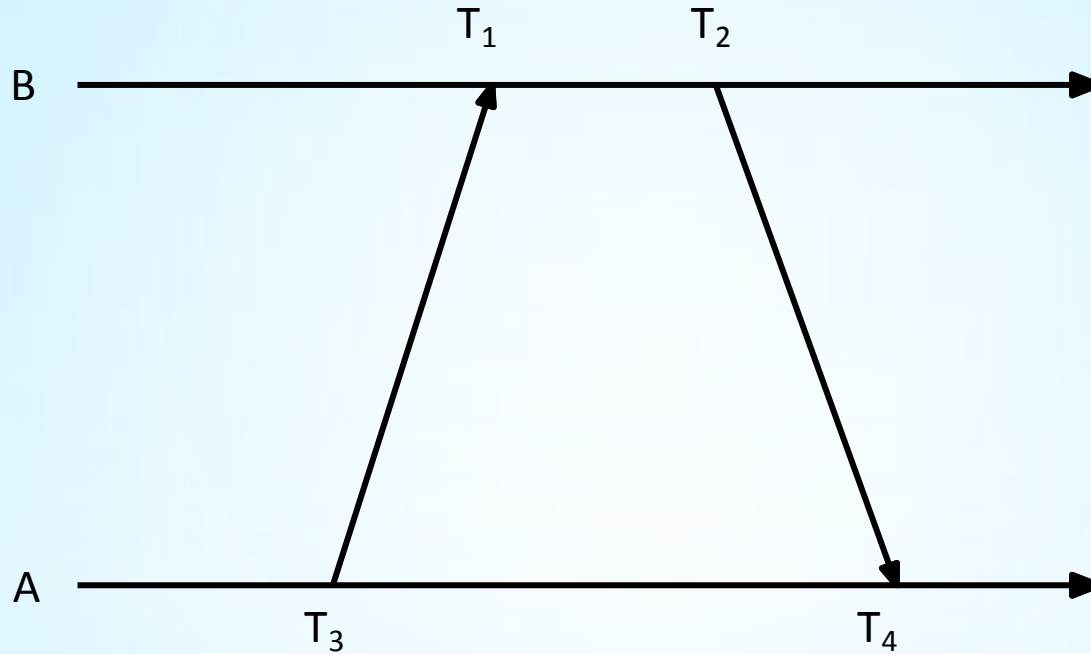


Figure 5.4: Offset and delay estimation

Clock offset and delay estimation:

- Let $a = T_1 - T_3$ and $b = T_2 - T_4$.
- If the network delay difference from A to B and from B to A , called **differential delay**, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T_4 are approximately given by the following:

$$\theta = \frac{a + b}{2}, \quad \delta = a - b \quad (2)$$

- Each NTP message includes the latest three timestamps T_1 , T_2 and T_3 , while T_4 is determined upon arrival.
- Thus, both peers A and B can independently calculate delay and offset using a single bidirectional message stream as shown in Figure 5.5.

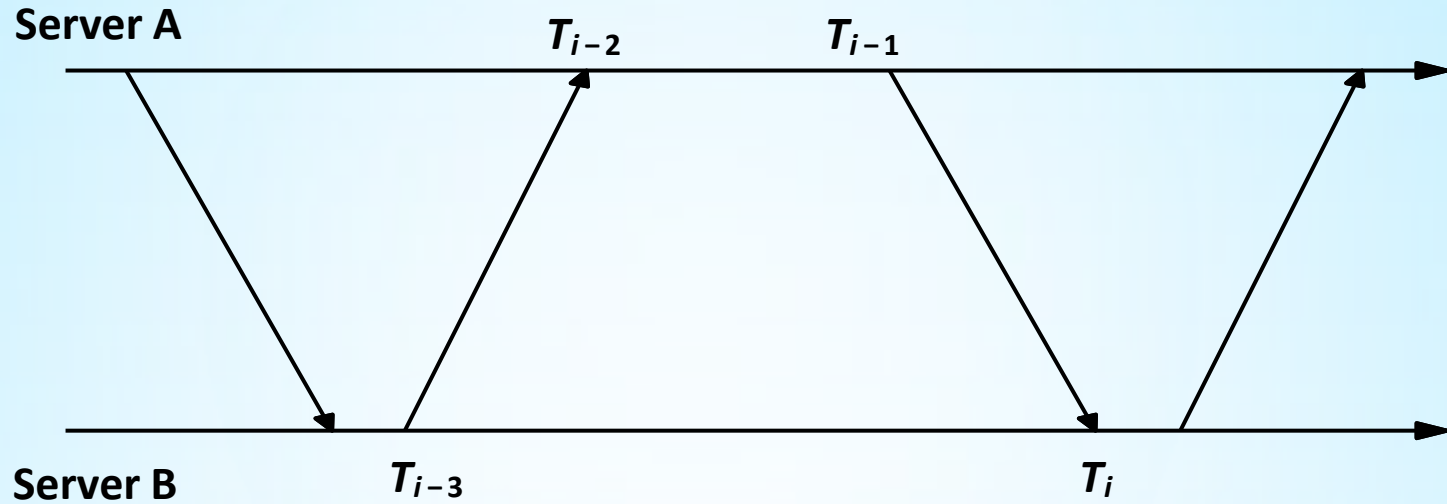


Figure 5.5: Timing diagram for the two servers

The Network Time Protocol: Synchronization protocol

- A pair of servers in symmetric mode exchange pairs of timing messages.
- A store of data is then built up about the relationship between the two servers (pairs of offset and delay).
- Specifically, assume that each peer maintains **pairs (O_i, D_i)** , where
- **O_i** - measure of offset (**θ**)
- **D_i** - transmission delay of two messages (**δ**).
- The offset corresponding to the minimum delay is chosen.
- Specifically, the delay and offset are calculated as follows. Assume that **message m** takes **time t** to transfer and **m'** takes **t'** to transfer.

Contd...

The offset between A's clock and B's clock is O .

If **A's local clock time is $A(t)$** and **B's local clock time is $B(t)$** , we have

$$A(t) = B(t) + O \quad (3)$$

Then,

$$T_{i-2} = T_{i-3} + t + O \quad (4)$$

$$T_i = T_{i-1} - O + t' \quad (5)$$

Assuming **$t = t'$** , the **offset O_i** can be estimated as:

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 \quad (6)$$

The **round-trip delay D_i** is estimated as:

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) \quad (7)$$

The eight most recent pairs of (O_i, D_i) are retained.

The value of O_i that corresponds to minimum D_i is chosen to estimate O

Q: Whether Physical Clock or Logical Clock is used to capture the 'Causality' (In Distributed System)?

- In day-to-day life, the global time(physical time) to deduce **causality relation** is obtained from **loosely synchronized clocks** (i.e., wrist watches, wall clocks).
- However, in distributed computing systems, the rate of occurrence of events is several magnitudes higher and the event execution time is several magnitudes smaller. Consequently, if the **physical clocks** are not precisely synchronized, the causality relation between events may not be accurately captured.
- Therefore, it turns out that in a distributed computation, the causality relation between events produced by a program execution and its fundamental monotonicity property can be accurately captured by **logical clocks**.

Conclusion

- In this lecture we have discussed about the **size of vector clock, matrix clocks and virtual time** to capture causality between events of a distributed computation and
- How the **physical clock synchronization** can be used as a paradigm for organizing and synchronizing distributed systems.
- In upcoming lecture, we will discuss about Global state and snapshot recording algorithms.