高级分布式系统

Advanced Distributed Systems

陈鹏飞
数据科学与计算机学院
chenpf7@mail.sysu.edu.cn
办公室：超算5楼529d
主页：http://sdcs.sysu.edu.cn/node/3747

第三讲 — 逻辑时间

本课程主要课件来源于Ajay Kshemkalyani and Mukesh Singhal 为课程Distributed Computing: Principles, Algorithms, and Systems 所设计的PPT，由衷表示感谢！！

Introduction

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time.
- As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

Introduction

- This chapter discusses three ways to implement logical time - scalar time, vector time, and matrix time.
- Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.
- The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems, such as distributed algorithms design, tracking of dependent events, knowledge about the progress of a computation, and concurrency measures.

A Framework for a System of Logical Clocks

Definition

- A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation $<$.
- Relation $<$ is called the *happened before* or *causal precedence*. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as $\mathrm{C}(e)$ and called the timestamp of e, and is defined as follows:

  $$C : H \mapsto T$$

  such that the following property is satisfied:

  $$\text{for two events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \Longrightarrow \mathrm{C}(e_i) < \mathrm{C}(e_j).$$

A Framework for a System of Logical Clocks

- This monotonicity property is called the *clock consistency condition*.
- When T and C satisfy the following condition,

  $$\text{for two events } e_i \text{ and } e_j,\ e_i \longrightarrow e_j \Leftrightarrow \mathrm{C}(e_i) < \mathrm{C}(e_j)$$

  the system of clocks is said to be *strongly consistent*.

Implementing Logical Clocks

- Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process p_i maintains data structures that allow it the following two capabilities:
  - A *local logical clock*, denoted by lc_i, that helps process p_i measure its own progress.

Implementing Logical Clocks

-
  - A *logical global clock*, denoted by gc_i, that is a representation of process p_i's local view of the logical global time. Typically, lc_i is a part of gc_i.

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

- *R1*: This rule governs how the local logical clock is updated by a process when it executes an event.
- *R2*: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.
- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.

Scalar Time

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i.
- Rules *R1* and *R2* to update the clocks are as follows:
- *R1*: Before executing an event (send, receive, or internal), process p_i executes the following:

  $$C_i := C_i + d \qquad\qquad (d > 0)$$

  In general, every time *R1* is executed, d can have a different value; however, typically d is kept at 1.

Scalar Time

- *R2*: Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:
  - $C_i := max(C_i, C_{msg})$
  - Execute *R1*.
  - Deliver the message.
- Figure 3.1 shows evolution of scalar time.

Scalar Time

Evolution of scalar time:

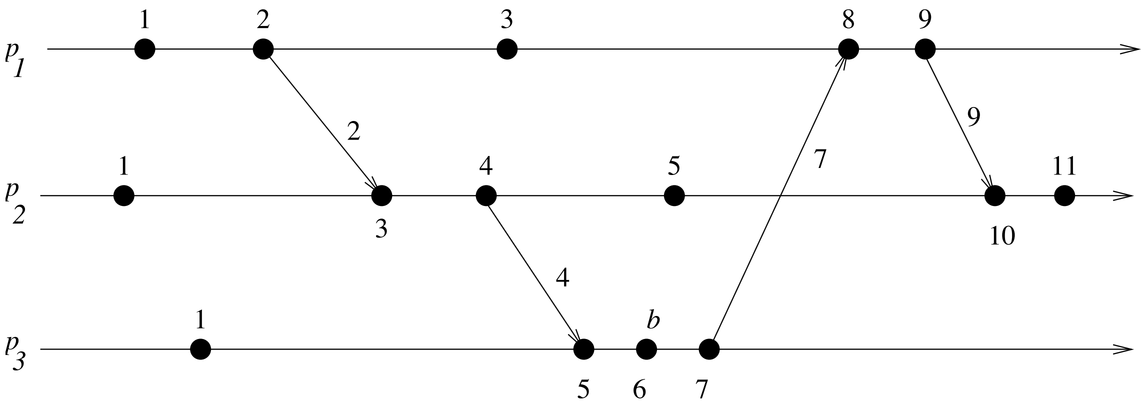

Figure 3.1: The space-time diagram of a distributed execution.

Basic Properties

Consistency Property

- Scalar clocks satisfy the monotonicity and hence the consistency property:

  for two events e_i and e_j, $e_i \rightarrow e_j \implies C(e_i) < C(e_j)$.

Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 3.1, the third event of process P_1 and the second event of process P_2 have identical scalar timestamp.

Total Ordering

A tie-breaking mechanism is needed to order such events. A tie is broken as follows:

- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred.
- The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j), respectively, is defined as follows:

$$x \ \prec \ y \Leftrightarrow \ (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

Properties. . .

Event counting

- If the increment value *d* is always 1, the scalar time has the following interesting property: if event *e* has a timestamp *h*, then *h-1* represents the minimum logical duration, counted in units of events, required before producing the event *e*;
- We call it the height of the event *e*.
- In other words, *h-1* events have been produced sequentially before the event *e* regardless of the processes that produced these events.
- For example, in Figure 3.1, five events precede event *b* on the longest causal path ending at *b*.

Properties. . .

No Strong Consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$.
- For example, in Figure 3.1, the third event of process P_1 has smaller scalar timestamp than the third event of process P_2.However, the former did not happen before the latter.
- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure 3.1, when process P_2 receives the first message from process P_1, it updates its clock to 3, forgetting that the timestamp of the latest event at P_1 on which it depends is 2.

Vector Time

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process p_i maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i.
- $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time.
- If $vt_i[j]{=}x$, then process p_i knows that local time at process p_j has progressed till x.
- The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

Vector Time

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

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

$$vt_i[i] := vt_i[i] + d \qquad\qquad (d > 0)$$

- *R2*: Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message *(m,vt)*, process p_i executes the following sequence of actions:
    - Update its global logical time as follows:

$$1 \le k \le n \ : \ vt_i[k] := max(vt_i[k], vt[k])$$

    - Execute *R1*.
    - Deliver the message m.

Vector Time

- The timestamp of an event is the value of the vector clock of its process when the event is executed.
- Figure 3.2 shows an example of vector clocks progress with the increment value *d*=*1*.
- Initially, a vector clock is $[0, 0, 0, ..., 0]$.

Vector Time

An Example of Vector Clocks

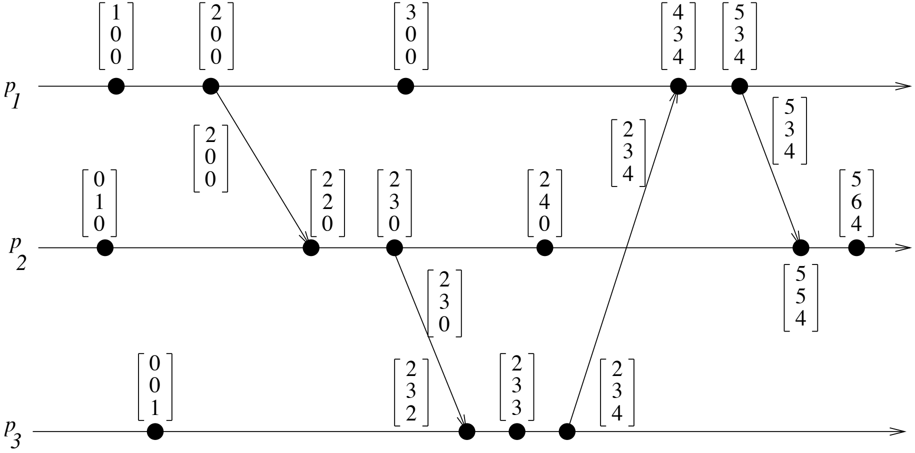

Figure 3.2: Evolution of vector time.

Vector Time

Comparing Vector Timestamps

- The following relations are defined to compare two vector timestamps, *vh* and *vk*:

$$\begin{aligned}
vh = vk &\quad \Leftrightarrow \quad \forall x : vh[x] = vk[x] \\
vh \leq vk &\quad \Leftrightarrow \quad \forall x : vh[x] \leq vk[x] \\
vh < vk &\quad \Leftrightarrow \quad vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\
vh \parallel vk &\quad \Leftrightarrow \quad \neg(vh < vk) \wedge \neg(vk < vh)
\end{aligned}$$

- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows: If events x and y respectively occurred at processes p_i and p_j and are assigned timestamps *vh* and *vk*, respectively, then

$$\begin{aligned}
x \rightarrow y &\quad \Leftrightarrow \quad vh[i] \leq vk[i] \\
x \parallel y &\quad \Leftrightarrow \quad vh[i] > vk[i] \wedge vh[j] < vk[j]
\end{aligned}$$

Vector Time

Properties of Vectot Time

Isomorphism

- If events in a distributed system are timestamped using a system of vector clocks, we have the following property.
  If two events x and y have timestamps vh and vk, respectively, then

$$\begin{aligned} x \rightarrow y \quad &\Leftrightarrow \quad vh < vk \\ x \parallel y \quad &\Leftrightarrow \quad vh \parallel vk. \end{aligned}$$

- Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.

Vector Time

Strong Consistency

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.
- However, Charron-Bost showed that the dimension of vector clocks cannot be less than n, the total number of processes in the distributed computation, for this property to hold.

Event Counting

- If d=1 (in rule *R1*), then the i^{th} component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.
- So, if an event e has timestamp vh, $vh[j]$ denotes the number of events executed by process p_j that causally precede e. Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

Efficient Implementations of Vector Clocks

- If the number of processes in a distributed computation is large, then vector clocks will require piggybacking of huge amount of information in messages.
- The message overhead grows linearly with the number of processors in the system and when there are thousands of processors in the system, the message size becomes huge even if there are only a few events occurring in few processors.
- We discuss an efficient way to maintain vector clocks.
- Charron-Bost showed that if vector clocks have to satisfy the strong consistency property, then in general vector timestamps must be at least of size n, the total number of processes.
- However, optimizations are possible and next, and we discuss a technique to implement vector clocks efficiently.

Singhal-Kshemkalyani's Differential Technique

- *Singhal-Kshemkalyani's differential technique* is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change.
- When a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j.
- If entries $i_1, i_2, \ldots, i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n_1}$, respectively, since the last message sent to p_j, then process p_i piggybacks a compressed timestamp of the form:

  $$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

  to the next message to p_j.

Singhal-Kshemkalyani's Differential Technique

When p_j receives this message, it updates its vector clock as follows:

$$vt_i[i_k] = max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \ldots, n_1.$$

- Thus this technique cuts down the message size, communication bandwidth and buffer (to store messages) requirements.
- In the worst of case, every element of the vector clock has been updated at p_i since the last message to process p_j, and the next message from p_i to p_j will need to carry the entire vector timestamp of size n.
- However, on the average the size of the timestamp on a message will be less than n.

Singhal-Kshemkalyani's Differential Technique

- Implementation of this technique requires each process to remember the vector timestamp in the message last sent to every other process.
- Direct implementation of this will result in $O(n^2)$ storage overhead at each process.
- Singhal and Kshemkalyani developed a clever technique that cuts down this storage overhead at each process to $O(n)$. The technique works in the following manner:
- Process p_i maintains the following two additional vectors:
  - $LS_i[1..n]$ ('Last Sent'):
    $LS_i[j]$ indicates the value of $vt_i[i]$ when process p_i last sent a message to process p_j.
  - $LU_i[1..n]$ ('Last Update'):
    $LU_i[j]$ indicates the value of $vt_i[i]$ when process p_i last updated the entry $vt_i[j]$.
- Clearly, $LU_i[i] = vt_i[i]$ at all times and $LU_i[j]$ needs to be updated only when the receipt of a message causes p_i to update entry $vt_i[j]$. Also, $LS_i[j]$ needs to be updated only when p_i sends a message to p_j.

Singhal-Kshemkalyani's Differential Technique

- Since the last communication from p_i to p_j, only those elements of vector clock $vt_i[k]$ have changed for which $LS_i[j] < LU_i[k]$ holds.
- Hence, only these elements need to be sent in a message from p_i to p_j. When p_i sends a message to p_j, it sends only a set of tuples

  $$\{(x, vt_i[x]) | LS_i[j] < LU_i[x]\}$$

  as the vector timestamp to p_j, instead of sending a vector of n entries in a message.
- Thus the entire vector of size n is not sent along with a message. Instead, only the elements in the vector clock that have changed since the last message send to that process are sent in the format $\{(p_1, latest\_value), (p_2, latest\_value), \ldots\}$, where p_i indicates that the p_ith component of the vector clock has changed.
- This technique requires that the communication channels follow FIFO discipline for message delivery.

Singhal-Kshemkalyani's Differential Technique

- This method is illustrated in Figure 3.3. For instance, the second message from p_3 to p_2 (which contains a timestamp $\{(3, 2)\}$) informs p_2 that the third component of the vector clock has been modified and the new value is 2.
- This is because the process p_3 (indicated by the third component of the vector) has advanced its clock value from 1 to 2 since the last message sent to p_2.
- This technique substantially reduces the cost of maintaining vector clocks in large systems, especially if the process interactions exhibit temporal or spatial localities.

Singhal-Kshemkalyani's Differential Technique

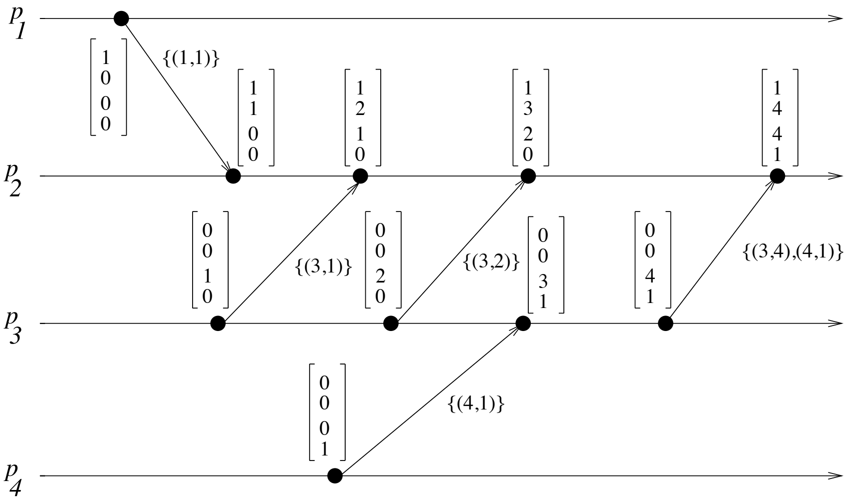

Figure 3.3: Vector clocks progress in Singhal-Kshemkalyani technique.

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:

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

- *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

- Figure 3.4 gives an example to illustrate how matrix clocks progress in a distributed computation. We assume $d{=}1$.
- Let us consider the following events: e which is the x_i-th event at process p_i, e_k^1 and e_k^2 which are the x_k^1-th and x_k^2-th event at process p_k, and e_j^1 and e_j^2 which are the x_j^1-th and x_j^2-th events at p_j.
- Let mt_e denote the matrix timestamp associated with event e. Due to message m_4, e_k^2 is the last event of p_k that causally precedes e, therefore, we have $mt_e[i,k]{=}mt_e[k,k]{=}x_k^2$.
- Likewise, $mt_e[i,j]{=}mt_e[j,j]{=}x_j^2$. The last event of p_k known by p_j, to the knowledge of p_i when it executed event e, is e_k^1; therefore, $mt_e[j,k]{=}x_k^1$. Likewise, we have $mt_e[k,j]{=}x_j^1$.

Matrix Time

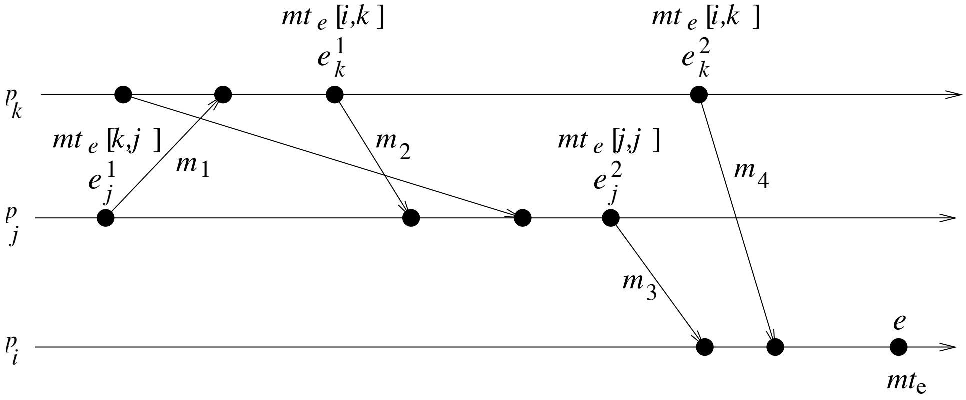

Figure 3.4: Evolution of matrix time.

Matrix Time

Basic Properties

- Vector $mt_i[i, .]$ 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_l'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_l will never send information with a local time $\leq t$.
  - In many applications, this implies that processes will no longer require from p_l 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_l 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 a 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.
- next, we discuss in detail Virtual Time and how Time Warp mechanism is used to implement it.

Virtual Time Definition

“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, occasionaly 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

- Causality of events is an important concept in distributed systems and is also a major constraint in the implementation of virtual time.
- 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.

Virtual Time Definition

Characteristics of Virtual Time

1. Virtual time systems are not all isomorphic; it may be either discrete or continuous.
2. Virtual time may be only partially ordered.
3. Virtual time may be related to real time or may be independent of it.
4. Virtual time systems may be visible to programmers and manipulated explicitly as values, or hidden and manipulated implicitly according to some system-defined discipline
5. 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 optimisticaly 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.

Virtual Time Definition

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.

Virtual Time Definition

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, nad messages may not be delivered in FIFO order.

Virtual Time Definition

Time Warp Mechanism

- Time Warp mechanism consists of two major parts: local control mechanism and global control mechanism.
- The local control mechanism insures that events are executed and messages are processed in the correct order.
- The global control mechanism takes care of global issues such as global progress, termination detection, I/O error handling, flow control, etc.

The Local Control Mechanism

- There is no global virtual clock variable in this implementation; each process has a *local virtual clock* variable.
- The local virtual clock of a process doesn't change during an event at that process but it changes only between events.
- On the processing of next message from the input queue, the process increases its local clock to the timestamp of the message.
- At any instant, the value of virtual time may differ for each process but the value is transparent to other processes in the system.

The Local Control Mechanism

- When a message is sent, the virtual send time is copied from the sender’s virtual clock while the name of the receiver and virtual receive time are assigned based on application specific context.
- All arriving messages at a process are stored in an input queue in the increasing order of timestamps (receive times).
- Processes will receive late messages due to factors such as different computation rates of processes and network delays.
- The semantics of virtual time demands that incoming messages be received by each process strictly in the timestamp order.

The Local Control Mechanism

Antimessages and the Rollback Mechanism

Runtime representation of a process is composed of the following:

- **Process name**: Virtual spaces coordinate which is unique in the system.
- **Local virtual clock: V**irtual time coordinate
- **State**: Data space of the process including execution stack, program counter and its own variables
- **State queue**: Contains saved copies of process's recent states as roll back with Time warp mechanism requires the state of the process being saved.
- **Input queue**: Contains all recently arrived messages in order of virtual receive time. Processed messages from the input queue are not deleted as they are saved in the output queue with a negative sign (antimessage) to facilitate future roll backs.
- **Output queue**: Contains negative copies of messages the process has recently sent in virtual send time order. They are needed in case of a rollback.

For every message, there exists an antimessage that is the same in content but opposite in sign.

Antimessages and the Rollback Mechanism

- Whenever a process sends a message, a copy of the message is transmitted to receiver's input queue and a negative copy (antimessage) is retained in the sender's output queue for use in sender rollback.
- Whenever a message and its antimessage appear in the same queue no matter in which order they arrived, they immediately annihilate each other resulting in shortening of the queue by one message.
- When a message arrives at the input queue of a process with timestamp greater than virtual clock time of its destination process, it is simply enqueued.
- When the destination process' virtual time is greater than the virtual time of message received, the process must do a rollback.

Antimessages and the Rollback Mechanism

Rollback Mechanism

- Search the "State queue" for the last saved state with timestamp that is less than the timestamp of the message received and restore it.
- Make the timestamp of the received message as the value of the local virtual clock and discard from the state queue all states saved after this time. Then the resume execution forward from this point.
- Now all the messages that are sent between the current state and earlier state must be "unsent". This is taken care of by executing a simple rule:

  "*To unsend a message, simply transmit its antimessage.*"
- This results in antimessages following the positive ones to the destination. A negative message causes a rollback at its destination if it's virtual receive time is less than the receiver's virtual time.

Antimessages and the Rollback Mechanism

Depending on the timing, there are several possibilities at the receiver's end:

- First, the original (positive) message has arrived but not yet been processed at the receiver.
- In this case, the negative message causes no rollback, however, it annihilates with the positive message leaving the receiver with no record of that message.
- Second, the original positive message has already been partially or completely processed by the receiver.
- In this case, the negative message causes the receiver to roll back to a virtual time when the positive message was received.
- It will also annihilate the positive message leaving the receiver with no record that the message existed. When the receiver executes again, the execution will assume that these message never existed.
- A rolled back process may send antimessages to other processes.

Antimessages and the Rollback Mechanism

- A negative message can also arrive at the destination before the positive one. In this case, it is enqueued and will be annihilated when positive message arrives.
- If it is negative message's turn to be executed at a processs' input queqe, the receiver may take any action like a no-op.
- Any action taken will eventually be rolled back when the corresponding positive message arrives.
- An optimization would be to skip the antimessage from the input queue and treat it as a no-op, and when the corresponding positive message arrives, it will annihilate the negative message, and inhibit any rollback.

Antimessages and the Rollback Mechanism

The antimessage protocol has several advantages:

- It is extremely robust and works under all possible circumstances.
- It is free from deadlocks as there is no blocking.
- It is also free from domino effects.
- In the worst case, all processes in system roll back to same virtual time as original one did and then proceed forward again.

Physical Clock Synchronization: NTP

Motivation

- 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

Motivation

-
- 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*.

Physical Clock Synchronization: NTP

Definitions and Terminology

Let C_a and C_b be any two clocks.

- **Time:** The time of a clock in a machine p is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
- **Frequency:** Frequency is the rate at which a clock progresses. The frequency at time t of clock C_a is $C_a^{'}(t)$.
- **Offset:** Clock offset is the difference between the time reported by a clock and the *real time*. The offset of the clock C_a is given by $C_a(t) - t$. The offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
- **Skew:** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $(C_a'(t) - C_b'(t))$. If the skew is bounded by ρ, then as per Equation (1), clock values are allowed to diverge at a rate in the range of $1 - \rho$ to $1 + \rho$.
- **Drift (rate):** The drift of clock C_a is the second derivative of the clock value with respect to time, namely, $C_a''(t)$. The drift of clock C_a relative to clock C_b at time t is $C_a''(t) - C_b''(t)$.

Physical Clock Synchronization: NTP

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.)

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho \qquad (1)$$

- Figure 3.5 illustrates the behavior of fast, slow, and perfect clocks with respect to UTC.

Physical Clock Synchronization: NTP

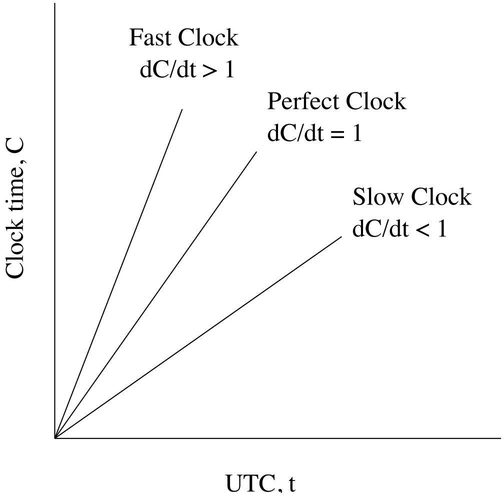

Figure 3.5: The behavior of fast, slow, and perfect clocks with respect to UTC.

Physical Clock Synchronization: NTP

Offset delay estimation method

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

Physical Clock Synchronization: NTP

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 3.6 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.

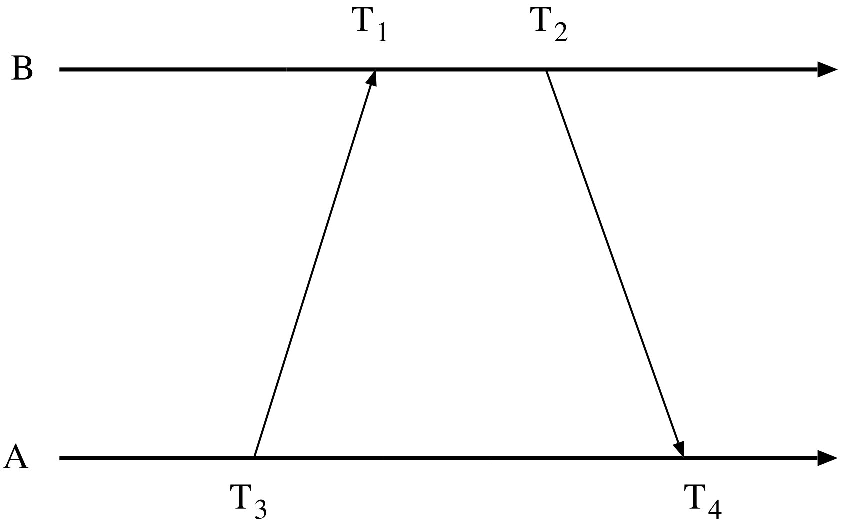

Figure 3.6: Offset and delay estimation.

- 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.

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

- 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 3.7.

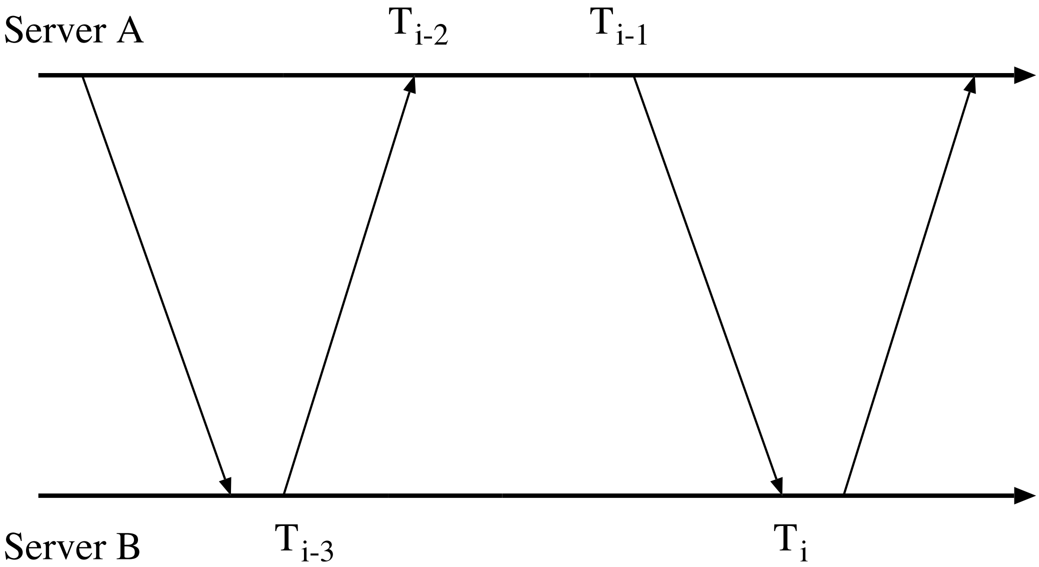

Figure 3.7: Timing diagram for the two servers.

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.

(Continued on the next slide . . . .)

The Network Time Protocol synchronization protocol.

- 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

$$A(t) = B(t) + O \tag{3}$$

Then,

$$T_{i-2} = T_{i-3} + t + O \tag{4}$$
$$T_i = T_{i-1} - O + t' \tag{5}$$

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

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

The round-trip delay is estimated as:

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

- 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.

谢谢