Distributed Coordination

Dr. Ananthanarayana V.S  
Professor and Head  
Dept. of Information Technology  
NITK, Surathkal, INDIA
Objectives of This Module

- This module discusses one of the fundamental issues in DCS – Distributed coordination.
- Need for distributed coordination, and how they can be ensured is also discussed.
- Distributed coordination in some of the distributed applications are also highlighted.
Road Map

- Introduction
- Importance of distributed coordination
- Ordering of events
- Logical clock
- Vector clock
- Protocols for ordering of messages
- Applications
- Summary
Distributed Computing System

- A collection of distinct processes which are spatially separated and which communicate one another by exchanging messages.

- A set of processes that cooperate to achieve a common goal (Distributed coordination).

- Fundamental issues:
  - No shared memory
  - Considerable and unpredictable message delays
  - Lack of global knowledge and global state
  - Lack of global clock
No Global Clock

➢ Two ways to handle this situation in DCS
  
  • Providing a common clock (UTC- Universal Coordinated Time) and is used to set the local clock in the system (External synchronization)
    ▶ Problem is: each local clock may refer the same common clock value at different instant of real time due to communication delay
  
  • Internal synchronization of local clocks, wherein local clocks will communicate among themselves for consensus regarding common clock
    ▶ Problem is: there is a possibility of drift due to technical limitations.[Each clock is made up of quartz and by means of piezoelectricity they will generate resonance frequency. This will vary due to angle of cut, temperature and size of crystal]
Drift situations

Correct clock value

Faster clock value

Slower clock value

\[(1 + \rho)t\]

\[(1 - \rho)t\]

H(t)

Bound on Drift : \(\rho\)

\[(1 - \rho)(t_2 - t_1) \leq H(t_2) - H(t_1) \leq (1 + \rho)(t_2 - t_1)\]

Note that typically \(\rho\) is small \((10^{-6})\)
Importance of Clock

- Clock, sequentially orders the execution of events/operations within one or many applications, which are the part of a single processor system (due to single clock).
- This is not so trivial in the system with multiple processors (hence multiple clocks) which are geographically apart.
- Due to lack of global clock (as per the reason given in earlier slides), ordering of events/operations of applications are not guaranteed automatically. To handle this, notion of logical clock is introduced.
- Logical clocks and its use in handling applications across the multiprocessor system is discussed in this module.
Distributed computing

- **Speed**
  - More speed with many processors

- **Inherent Distribution**
  - Some applications are inherently distributed
    - Example: A bank may have many branches distributed across the globe, where most of the activities are local in nature.

- **Fault tolerance**
  - Availability
    - Prob. of the system being continuously available during a given time interval.
  - Reliability
    - Prob. of the system being up during certain point time.

- **Incremental growth**
  - Resources can be added incrementally without disturbing the user

- **Interoperability**
  - Inter resource sharing across the system
Distributed database system: An example to DCS

MUMBAI

MANGALORE

DELHI

KOLKATA

DBS: Database System
L: Local transaction
R: Remote transaction
G: Global transaction
In previous example,

- **Local transaction (L):** Booking the tickets at Mangalore from Mangalore to Mumbai.
- **Remote transaction (R):** Booking the tickets at Mangalore from Mumbai to Kolkata
- **Global transaction (G):** Booking the tickets at Mangalore from Mumbai to Delhi via Kolkata

**Execution:**

- For **L**, the execution is similar as in centralized database system
- For **R**, the propagation of request from Mangalore to Mumbai is required and later result to Mangalore (without being noticed by the user – access and location transparency)
- For **G**, the request need to be decomposed (Mumbai to Kolkata ticket booking – \( T_1 \) and Kolkata to Delhi ticket booking – \( T_2 \) and submitted to Mumbai and Kolkata respectively. Later result of both should be integrated and available at Mangalore (without being noticed by the user - access, location, parallelism transparency)

**System should give single view of multiple database system to the user.**
Global State Detection

- Need to record the state of the system during the execution of transactions
- This is required to recover the system in case of failure so that consistency of the system is maintained.
- This is a challenging task in distributed system due to lack of global clock and shared memory
Global State Detection – Centralized System

All events of the transaction are recorded with respect to the time.
In distributed system due to lack of global clock, the order of recording of events may not be according to real time. This leads to inconsistency. [As shown above, ordering of events between P1, P2 and P3 is arbitrary.] This is explained in the following slides.
Global State Detection (Contd.)

- Consider a DCS with two sites, Site1 and Site2 with account information, A at Site1 and B at Site2. Let C be the communication channel between Site1 and Site2.
- Let T be the transaction to transfer the fund, $50 from account A to B. This contains following actions:
  - (1) Subtract $50 of A at Site1
  - (2) Transfer $50 across C to Site2
  - (3) Add $50 at B of Site2
- During each sub operation, the state of the system (i.e., state of Site1, Site2 and C) need to be recorded
- Global state is defined as set of state of all sites.
- If states of Site1, Site2 and C are recorded at any of Global state1/Global state2/Global state3 (shown in Figure1), then the system is consistent.
- Due to lack of global clock/memory it may not be guaranteed.
Global State Detection – Figure 1

Site 1
A: $500  C: Empty

Site 2
B: $200

Global State 1

Site 1
A: $450  C: $50

Site 2
B: $200

Global State 2

Site 1
A: $450  C: Empty

Site 2
B: $250

Global State 3
Case 1: Record the state of Site1 and Site2 in Global state 1 and C in Global state 2 ⇒ Excess of $50 (inconsistency state)
Global State Detection – Figure 1

Case 1: Record the state of Site1 and Site2 in Global state 1 and C in Global state 2 ⇒ Excess of $50 (inconsistency state)

Case 2: Record the state of C in Global state1 and state of Site 1 and Site 2 in Global state 2 ⇒ Deficit of $50 (inconsistency state)
Global State Detection – Formalism to capture inconsistencies

- Let $n$ – number of messages sent along $C$ before Site1 state is recorded.
- Let $n^1$ – number of messages sent along $C$ before C’s state is recorded.
  - In Case 1, $n = 0$ and $n^1 = 1$; $n < n^1$
  - In Case 2, $n = 1$ and $n^1 = 0$; $n > n^1$
- Let $m$ – number of messages received along $C$ before Site2 state is recorded.
- Let $m^1$ – number of messages received along $C$ before C’s state is recorded.
  - In Case 1, $m = 0$ and $m^1 = 1$; $m < m^1$
  - In Case 2, $m = 1$ and $m^1 = 0$; $m > m^1$
- If state of Site1, Site2 and C are all recorded in either of Global state1/Global state2/Global state3 then,
  - $n = n^1$ and $m = m^1 \rightarrow (1)$
Since in no communication channel, the number of messages sent is less than the number of messages received.

- So, $n^1 \geq m^1 \rightarrow (2)$
- From (1) & (2), $n \geq m$

In other words, consistent global state must satisfy $n \geq m$. 
Global State Detection – Formalism to capture inconsistencies (Contd…)

Definitions:

- Let $L_{Si}$ – local state of site, $S_i$ at any time
- Let $Send(m_{ij})$ – send event of message $m_{ij}$ from $S_i$ to $S_j$
- Let $Recv(m_{ij})$ – receive event of message $m_{ij}$ at $S_j$
- $Time(x)$ – time at which state $x$ is recorded
- $Time(Send(m_{ij}))$ – time at which event $Send(m_{ij})$ occurred
- $Time(Recv(m_{ij}))$ – time at which event $Recv(m_{ij})$ occurred
Global State Detection – Formalism to capture inconsistencies (Contd…)

➢ Further,
  • Send\(m_{ij}\) ∈ \(LS_i\) iff \(\text{Time}(\text{Send}(m_{ij})) < \text{Time}(LS_i)\)
  • \(\text{Recv}(m_{ij})\) ∈ \(LS_i\) iff \(\text{Time}(\text{Recv}(m_{ij})) < \text{Time}(LS_i)\)

➢ From local states, \(LS_i\) and \(LS_j\) we define two sets of messages
  • Transit: \(\text{transit}(LS_i, LS_j) = \{m_{ij} | \text{Send}(m_{ij}) \in LS_i \land \text{Recv}(m_{ij}) \notin LS_j\}\)
  • Inconsistent: \(\text{inconsistency}(LS_i, LS_j) = \{m_{ij} | \text{Send}(m_{ij}) \notin LS_i \land \text{Recv}(m_{ij}) \in LS_j\}\)

➢ Consistent Global state
  • A Global state, \(GS = \{LS_1, LS_2, \ldots, LS_n\}\) is said to be consistent
    • iff \(\forall i, j: 1 \leq i, j \leq n \mid \text{inconsistency}(LS_i, LS_j) = \emptyset\)
  • A Global state, \(GS = \{LS_1, LS_2, \ldots, LS_n\}\) is said to be transit-less
    • iff \(\forall i, j: 1 \leq i, j \leq n \mid \text{transit}(LS_i, LS_j) = \emptyset\)
  • A Global state, is said to be strongly consistent if it is consistent and transit-less
Global State Detection – Formalism to capture inconsistencies (Contd…)

Ananthanarayana VS, NITK-S
Issue of distributed coordination

- Distributed mutual exclusion (synchronization)
- Distributed scheduling
- Distributed deadlock handling
- ...etc
Ordering of events

Idea:

- Process is considered as set of events
- An event is
  - Sending of a message
  - Receiving of a message
  - Computation
- An example (Space-time diagram)
Happen before relation (→)

Let S be the set of events in DCS. The happen before relation on S satisfies the following conditions.

- If \( a \) and \( b \) are events in the same process and \( a \) occurs before \( b \), then \( a \rightarrow b \)
- If \( a \) is the event of sending the message by a process \( P_i \) and \( b \) is the event of receiving the same message by \( P_j \) (\( i \neq j \)), then \( a \rightarrow b \)
- If \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \) (transitive)
- \( \forall a \in S, a \notightarrow a \) (non-reflexive)
- \( a \parallel b : \text{if} \ a \notightarrow b \text{ and } b \notightarrow a \) (concurrent events)
An example for happen before relation

\[ e_{22} \rightarrow e_{13} \]
\[ e_{13} \rightarrow e_{14} \]
\[ e_{22} \rightarrow e_{14} \]
\[ e_{11} \parallel e_{21} \]
Realization of happen before relation using Lamport’s logical clock

- For each process $P_i$, there is a clock $C_i$

- For an event $a$ of $P_i$, the clock value is $C_i(a)$

- In DCS, it is assumed that there are $N$ processes with clocks, $C_1$, $C_2$, ..., $C_N$
Lamport’s logical clock conditions

- **C1**: For any two events $a$ and $b$ in $P_i$, if $a$ happens before $b$ then $C_i(a) < C_i(b)$

- **C2**: If $a$ is the event of sending the message in $P_i$ and $b$ is the event of receiving the same message at $P_j$ ($i \neq j$) then $C_i(a) < C_j(b)$
Lamport’s logical clock implementation rules

- $IR1$: Let $a$ and $b$ be two successive events in $P_i$ then clock value at $b$ is updated as
  
  $$C_i(b) = C_i(a) + d \quad (\text{where } d > 0)$$

- $IR2$: Let $a$ be the event of sending the message $M$ at $P_i$ (where $M$ carries count value, $t_M = C_i(a)$). On receipt of $M$ by $P_j$, the clock value $C_j$ is updated as
  
  $$C_j = \text{MAX}(C_j, t_M) + d \quad (\text{where } d > 0)$$
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially.
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
Assume that each process’s logical clock is set to 0 initially.
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
An example for working of Lamport’s logical clock

Assume that each process’s logical clock is set to 0 initially
Total ordering of events ($\Rightarrow$)

- If $a$ is any event in $P_i$ and $b$ is any event in $P_j$ ($i \neq j$) then $a \Rightarrow b$ iff

  \[ C_i(a) < C_j(b) \text{ OR } C_i(a) = C_j(b) \text{ AND } (i < j) \]

- Example: Timestamp based concurrency control mechanism

- Timestamp in general is a triplet
  
  $<$Site-id, Process-id, Lamport’s logical clock$>$
Example (total order)

Assume that each process’s logical clock is set to 0
Limitations of Lamport’s logical clock

- **Weak clock condition**
  - \( \forall a, b \in S, \text{ if } a \rightarrow b \text{ then } C(a) < C(b) \)
  
  Reverse condition i.e.,
  
  if \( a \not\rightarrow b \) then \( C(a) \not< C(b) \) does not necessarily hold.

- **Clock values are monotonically increasing**

\[ e_{11} < e_{32} \text{ and } e_{11} \not\rightarrow e_{32} \]
Proof for Lamport’s Weak Condition

- By contradiction
  - Reverse condition: if $a \leftrightarrow b$ then $C(a) \not\preceq C(b)$
  - Consider:

\[ \begin{align*}
P_1 & : a \quad b \\
P_2 & : c
\end{align*} \]

Since $a$ and $c$ are parallel ($a \parallel c$):
\[
\begin{align*}
C(a) & = C(c) \quad \text{(1)}
\end{align*}
\]

Since $b$ and $c$ are parallel ($b \parallel c$):
\[
\begin{align*}
C(b) & = C(c) \quad \text{(2)}
\end{align*}
\]

From (1) and (2), we have contradiction by assuming reverse condition is true. Hence the proof.
Vector clocks

- Let \( N \) be the number of processes in DCS.
- For each process \( P_i \), there is a vector clock \( V_i \) of length \( N \).
- \( V_i[i] \) implies the \( P_i \)’s own clock value.
- \( V_i[j] \) (\( i \neq j \)) implies \( P_i \)’s best guess of logical time of \( P_j \).
**Vector clock – implementation rules**

- **IR1**: Let \( a \) and \( b \) be two successive events in \( P_i \) then \( V_i \) at \( b \) is updated as

\[
V_i[i] = V_i[i] + d \quad \text{(where } d > 0)\]

- **IR2**: Let \( a \) be the event of sending the message \( M \) at \( P_i \) (where \( M \) carries vector \( t_M = V_i \)). On receipt of \( M \) by \( P_j \), the vector \( V_j \) is updated as

\[
\forall k, \ (V_j[k] = \text{MAX}(V_j[k], t_M[k]))
\]
Example for Vector clock updation

$P_1$

$P_2$

$P_3$
Example for Vector clock updation

P₁

1
0
0

P₂

P₃
Example for Vector clock updation
Example for Vector clock updation
Example for Vector clock updation

\[ \begin{array}{c|c|c}
P_1 & 1 & 0 \\
   & 0 & 2 \\
\hline
P_2 & 0 & 0 \\
   & 1 & 0 \\
\hline
P_3 & 0 & 0 \\
   & 1 & 0
\end{array} \]
Example for Vector clock updation

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>P2</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>01</td>
</tr>
</tbody>
</table>
Example for Vector clock updation

- $P_1$: 100 200
- $P_2$: 010 220
- $P_3$: 000 20

Anantharanayana V3, NITK-S
Example for Vector clock updation

Diagram showing vector clock updates for nodes P1, P2, and P3.
Example for Vector clock updation

P1: [1, 0, 0, 0]
P2: [0, 1, 2, 2]
P3: [0, 0, 1, 0]
Example for Vector clock updation

Example of vector clock updation across processes P1, P2, and P3.

- P1 with clock values [1, 2, 3] updates to [2, 2, 2] when receiving an event from P2.
- P2 with clock values [0, 2, 2] updates to [0, 2, 3] when receiving an event from P3.
- P3 with clock values [0, 0, 1] updates to [0, 0, 3] when receiving an event from P2.

The diagram illustrates the vector clock values and the flow of events among the processes.
Example for Vector clock updation

<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Ananthanarayana V S, NITK-5
Example for Vector clock updation

Example for Vector clock updation
Example for Vector clock updation
Example for Vector clock updation

\[
\begin{array}{c|c|c|c}
\text{P}_1 & \text{P}_2 & \text{P}_3 \\
\hline
0 & 0 & 0 \\
1 & 2 & 2 \\
0 & 2 & 3 \\
0 & 3 & 4 \\
0 & 2 & 3 \\
0 & 3 & 4 \\
\end{array}
\]
Example for Vector clock updation

\[
\begin{align*}
P_1 &: 100 \quad 200 \quad 300 \\
P_2 &: 010 \quad 220 \quad 330 \\
P_3 &: 020 \quad 230 \quad 340 \\
\end{align*}
\]
Example for Vector clock updation
Example for Vector clock updation

(P1) (P2) (P3)

Example for Vector clock updation

(P1) (P2) (P3)

Example for Vector clock updation

(P1) (P2) (P3)
Example for Vector clock updation

Example for Vector clock updation
Example for Vector clock updation
Basic Properties – Vector clocks

- Comparing two vector clocks
  - $V_i \leq V_j$ iff $\forall k \ V_i[k] \leq V_j[k]$
  - $V_i < V_j$ iff $\forall k \ V_i[k] \leq V_j[k]$ and $\exists k : V_i[k] < V_j[k]$

- Strong condition
  - For any two events $a$ and $b$, $a \rightarrow b$ iff $V_i^a < V_j^b$
  - Applications: causal ordering of messages, distributed debugging, establishing global breakpoints, consistency of checkpoints in optimistic recovery
Vector Clock – Strong Condition

- For any two events $a$ and $b$, $a \rightarrow b$ iff $V^a_i < V^b_j$
- Proof: By considering all possibilities (exhaustive means)

If part: [ if $a \rightarrow b$ then $V^a_i < V^b_j$ ]

- Case 1: If $a$ and $b$ are two events at P and $a$ occurs before $b$ then, as per IR1, $V^a_i < V^b_j$
- Case 2: If $a$ at $P_i$ is sending event and $b$ at $P_j$ is receiving event (of message sent due to $a$) then, as per IR2, $V^a_i < V^b_j$
- Case 3: If $\exists c$ such that $a \rightarrow c$ and $c \rightarrow b$ then, as per IR1 and/or IR2, $V^a_i < V^b_j$

Ananthanarayana V8, NITK-S
Vector Clock – Strong Condition

Only if part: \([\text{if } a \leftrightarrow b \text{ then } V_i^a \preceq V_j^b]\)

- Case 1: No path from \(a\) to \(b\)
  - \(V_i^a = (2,0), V_i^b = (1,1)\)
  - \(V_i^a = (1,0), V_i^b = (0,1)\)

- Case 2: Path from \(b\) to \(a\)
  - \(V_i^a = (1,1), V_i^b = (0,1)\)

- In all situations highlighted under case 1 and case 2, \(V_i^a \preceq V_j^b\). Hence the proof.

Ananthanarayana VS, NITK-S
Causal ordering of messages

(Send M1)  

P_1  

(Send M2)  

P_2  

(Recv M2)  

P_3  

(Recv M1)
Importance of ordering of messages

Initial balance at account X is Rs. 1500.

Update 1 (M1) : credit Rs. 1000 on account X
Update 2 (M2) : debit Rs. 2000 on account X

Execution should be Update1 → Update 2 at both sites

User want to credit Rs. 1000 first and then debit Rs. 2000. They are referred as M1 and M2. Reach of M1 and M2 at SITE1 and at SITE 2 should be same and is M2 follows M1.[Note: If M1 follows M2, at any site leads to non execution of update 2 due to negative balance.]
Protocols for causal ordering of messages

- Birman-Schiper-Stephenson (BSS) Protocol
- Schiper-Eggli-Sandoz (SES) Protocol

Basic Idea:

- Deliver the message provided its immediately preceding message is delivered.
- Otherwise, the message is buffered till its preceding message is delivered.
- Vector clock is used to infer the preceding messages if any.
Architecture for both protocols

Server

Application process

Network
BSS Protocol

- Before broadcasting a message \(M\), the process \(P_i\) increments the vector time \(VT_i[i]\) and timestamps \(M\).

- A process \(P_j \neq P_i\), upon receiving \(M\) with timestamp \(VT_M\) from \(P_i\), delays its delivery until both of following conditions are satisfied:
  
  - \(VT_j[i] = VT_M[i] - 1\)
  - \(VT_j[k] \geq VT_M[k] \quad \forall k \in \{1, 2, \ldots, N\} - \{i\}\)
    
  (where \(N\) is \# of processes in DCS)

- Delayed messages are queued in the order sorted by vector clock of messages. (Concurrent messages are ordered by the time of their receipt)

- When the messages are delivered at a process \(P_j\), \(VT_j\) is updated according to IR2 of vector clock

- **Assumptions:**
  
  (i) Communication is by broadcasting
  (ii) Messages delivered are lossless
Illustrative example for BSS protocol

Note: The own timestamp will be increased only at the time of sending the message
Ordering of messages

- Broadcast server delivers its messages to the AP in any of the following order:
  - FIFO: If an AP broadcasts message $p$ and then $q$, all APs receive $q$ after $p$.
    - Ex: Banking money transfer w.r.t an account
  - CAUSAL: If $\exists$ SEND($p$) $\rightarrow$ SEND($q$) then $p$ is received before $q$ by all APs.
    - Ex: Bus reservation
  - TOTAL: If an AP receives $p$ before $q$, then all other APs should receive $p$ before $q$.
    - Ex: DME algorithm
SES Protocol

- Data structures:
  - Each processor maintains a vector $V_P$ of size $N-1$, where $N$ is the # of processes in DCS.
  - Each entry of $V_P$ is an ordered pair $(P', t)$, where $P'$ is the id of destination process and $t$ is the timestamp.
  - $t_M$ – timestamp when sending the message, $M$
  - $t_{P_i}$ – current time at $P_i$

- Assumption
  - Processes communicate by point to point means
SES Protocol— Contd.

Sending \( M \) from \( P_1 \) to \( P_2 \):

- Send \( M \) along with \( V_{P_1} \) to \( P_2 \)
- Increment \( t_M \) as per IR1 of Vector Clock
- Insert \((P_2, t_M)\) in \( V_{P_1} \). If \( V_{P_1} \) contains a pair \((P_2, t)\) then overwrite it. Note that \((P_2, t_M)\) was not sent to \( P_2 \)
SES Protocol - Contd.

- After receipt of \( M \) by \( P_2 \)
  - If \( V_M \) (vector accompanying the message \( M \))
    does not contains \((P_2, t)\), then
  - Else,
    - If \( t < t_{P2} \), then
    - Else
SES Protocol - Contd.

After receipt of $M$ by $P_2$

- If $V_M$ (vector accompanying the message $M$) does not contain $(P_2, t)$, then deliver $M$.

- Else,
  - If $(t < t_{P2})$, then
  - Else
SES Protocol - Contd.

- After receipt of \( M \) by \( P_2 \):
  - If \( V_M \) (vector accompanying the message \( M \)) does not contain \( (P_2, t) \), then deliver \( M \).
  - Else,
    - If \( t < t_{P_2} \), then deliver \( M \)
    - Else
SES Protocol - Contd.

After receipt of $M$ by $P_2$

- If $V_M$ (vector accompanying the message $M$) does not contain $(P_2, t)$, then deliver $M$.

- Else,
  - If $(t < t_{P2})$, then deliver $M$
  - Else buffer $M$ for late delivery
SES Protocol – Contd.

- Updation at P₂ after delivery

- Merge V_M accompanying M with VP₂ as follows:
  - If (∃(P,t) ∈ V_M such that P ≠ P₂) and (∀(P',t) ∈ V_P₂, P ≠ P') then insert (P,t) in V_P₂
  - ∀P, P ≠ P₂, if ((P,t) ∈ V_M and (P,t') ∈ V_P₂) then substitute (P,t') ∈ V_P₂ by (P,t.sup) such that ∀i t.sup[i] = MAX(t[i], t'[i])

- Update P₂’s logical clock

- Check for the buffered messages that can now be delivered since the local clock is updated
Applications of Lamport’s logical clock in distributed environment

- Lamport’s DME algorithm
- Distributed deadlock prevention algorithm
- Distributed deadlock detection method
- Conflict resolution in replicated databases
Lamport’s DME algorithm

- **Enter CS**
  - When a site $S_i$ wants to enter its CS, it sends $\text{REQ}(L_i, i)$ to all sites and places it in its Request-queue$_i$
  - When a site $S_j$ receives $\text{REQ}(L_i, i)$ from $S_i$, it returns $\text{REP}$ message to $S_i$ and places it in its Request-queue$_i$

- **$S_i$ executes CS if following two conditions hold:**
  - $S_i$ has received all $\text{REP}$ messages with timestamp greater than $(L_i, i)$ from all other sites
  - $S_i$’s request is at the top of Request-queue$_i$
Lamport’s DME algorithm – Contd.

- **Exit CS**

  - Site $S_i$, upon exiting the CS, removes its request from the top of its request-queue and sends a time-stamped REL message to all other sites.

  - When a site $S_j$ receives a REL message from $S_i$, it removes $S_i$’s request from its request-queue.
Illustrative example for Lamport’s DME

Sites $S_1$ and $S_2$ are making requests for CS
Illustrative example – Contd.

Site $S_2$ enters its CS

$S_2$ enters its CS
Illustrative example – Contd.

Site S2 exits its CS and sends REL messages.
Distributed deadlock prevention

Let a resource R is held by P₁ at some site, and process P₂ requests R. Let e(P₁) and e(P₂) be the timestamps of P₁ and P₂ respectively.

- **Wait-die method:**
  - If e(P₂) < e(P₁) then P₂ waits /* P₂ is older */
  - Else P₂ is killed /* P₂ is younger */

![Diagram](image)
Distributed deadlock prevention

Wound – wait method:

- If \( e(P_2) < e(P_1) \) then \( P_1 \) is killed /* \( P_2 \) is older */
- Else \( P_2 \) is waits /* \( P_2 \) is younger */

In both cases, killed process is restarted with the same timestamp

Wound-wait method preempts the process
Distributed deadlock detection

Centralized approach

- At $S_1$: $M_1 : T_3$ releases $R_2$
- At $S_2$: $M_2 : T_3$ requests $R_3$

$M_1$ occurs before $M_2$
$M_2$ reaches coordinator before $M_1$
Distributed deadlock detection

Centralized approach

At S₁: \( M₁ : T₃ \) releases \( R₂ \)

At S₂: \( M₂ : T₃ \) requests \( R₃ \)

\( M₁ \) occurs before \( M₂ \)

\( M₂ \) reaches coordinator before \( M₁ \)

False Deadlock
Distributed deadlock detection

Centralized approach

At S₁: \( M₁ : T₃ \) releases \( R₂ \)
At S₂: \( M₂ : T₃ \) requests \( R₃ \)
\{ \( M₁ \) occurs before \( M₂ \)
\( M₂ \) reaches coordinator before \( M₁ \) \}

False
Deadlock
Distributed deadlock detection

Centralized approach

At $S_1$: $M_1 : T_3$ releases $R_2$
At $S_2$: $M_2 : T_3$ requests $R_3$

Solution: Timestamp based messages; ordering at Coordinator

$M_1$ occurs before $M_2$
$M_2$ reaches coordinator before $M_1$
False Deadlock
The replication in distributed environment can be effectively handled by means of coordination among the sites.

For this consider Partitioned Primary Copy method.

Data (table) is replicated at many sites.

To improve the availability, each table(T) is partitioned into $P_1$, $P_2$, ..., $P_n$ and partition $P_i$ in site $S_i$ is identified as primary copy.

The modification request for the data in partition $P_i$ is handled only by the site $S_i$. (Similar to centralized approach).

This is explained in the following slide.

However, using coordination concept (in terms of timestamp) this can be achieved more effectively using Dynamic Primary Copy method. This is explained in the subsequent slides.
Partitioned primary copy method

Table T1

S1 S2

S3 S4

Table T1

Primary copy partition

Replica
Dynamic primary copy method

- Idea: The notion of primary copy is ‘dynamic’ in the sense that the write request (r) is served at the site where it originates.
- The site where r originates will broadcast the request and will get the acknowledgements from all others to go ahead with write provided timestamp of r is smallest among all conflicting requests with respect to r.
- This is explained in the next slide, where rₓ and rᵧ are conflicting requests originated at sites Sᵰ and Sᵧ respectively.
- Since timestamp(rₓ) < timestamp(rᵧ), rₓ will get all acknowledgements and updates at site Sᵰ. After execution of request rₓ, the modification due to rₓ and acknowledgement from Sᵰ is piggybacked and sent to Sᵧ. Now, at Sᵧ, after applying the modification, rᵧ will be executed since it got all acknowledgements.
- These activities are explained under REQUEST PHASE, ACKNOWLEDGEMENT PHASE and UPDATE PHASE in the following slide.
Dynamic primary copy method

REQUEST PHASE OF $S_i$ and $S_j$

$S_i \xrightarrow{Request} S_j \xrightarrow{Request} S_k \xrightarrow{Request} \ldots$

ACKNOWLEDGEMENT PHASE OF $S_i$ and $S_j$

$S_i \xrightarrow{Ack} S_j \xrightarrow{Ack} S_k \xrightarrow{Ack} \ldots$

UPDATE PHASE OF $S_i$

$S_i \xrightarrow{Update} S_j \xrightarrow{Update} S_k \xrightarrow{Update} \ldots$

ACK FOR $r_x$

UPDATE DUE TO $r_x +$ ACK FOR $r_y$

UPDATE PHASE OF $S_j$

$S_i \xrightarrow{Update} S_j \xrightarrow{Update} S_k \xrightarrow{Update} \ldots$

TS($r_x$) < TS($r_y$)

Ananthanarayana VS, NITK-S
Applications:

- Real time distributed systems
- Terminating conditions for many s/w protocols are based on time-out mechanism for handling exceptions
- Secure internet communication
- Checking the expiration of privileges for access control
Cristian’s algorithm

Sender

$T_0$  Request  Time Server

$T_1$  Time, $T$

$I$, time to process interrupt
If $d$ is the estimate of delay from the TS to the sender, the sender should set its clock to $T + d$. 

Sender 

$T_0$ 

Request 

Time Server 

$T_1$ 

Time, $T$ 

I, time to process interrupt
If \( d \) is the estimate of delay from the TS to the sender, the sender should set its clock to \( T + d \).

If nothing is known about \( I \), then estimated \( d = (T_1 - T_0)/2 \).
Ananthanarayana VS, NITK-S

Cristian’s algorithm

Sender

\[ T_0 \]

Request

Time Server

\[ T_1 \]

Time, \( T \)

If \( d \) is the estimate of delay from the TS to the sender, the sender should set its clock to \( T + d \).

I, time to process interrupt

If nothing is known about \( I \), then estimated \( d = (T_1 - T_0) / 2 \).

If \( I \) is known, then estimated \( d = (T_1 - T_0 - I) / 2 \).
If \( d \) is the estimate of delay from the TS to the sender, the sender should set its clock to \( T+d \).

If nothing is known about \( I \), then estimated \( d = (T_1 - T_0)/2 \).

If \( I \) is known, then estimated \( d = (T_1 - T_0 - I)/2 \).

Or \( d \) may be estimated using average or minimum of multiple requests.
Berkeley algorithm

- The time server periodically gathers time data from all other machines and determines the average time. Then it communicates the average time (in fact the difference) to all machines.

- Suitable for systems which can not access any UTC (Universal Coordinated Time) server.

- Problem: The Berkeley algorithm is centralized and therefore has a single point of failure.
One important fact

- Note that the clock of the system can not be set backwards. Doing so may result in an inconsistent system state.
One important fact

- Note that the clock of the system cannot be set backwards. Doing so may result in an inconsistent system state.

To solve the problem, the clock is slowed down for a certain period of time. Once the system clock reaches the desired state, the rate of change of the clock may be restored to the old state.
External clock synchronization

Each clock tries to synchronize their clock time w.r.t a standard time like UTC (Universal Coordinated Time)
Internal clock synchronization

Each client will communicate with others in the DCS so that the time difference is always less than some threshold (specified by the application)
Summary

- Importance of ordering of events and messages are discussed
- Realization of both using Lamport’s logical clock and Vector clock are presented
- Applications of Lamport’s logical clock in a distributed environment are focused
References

