Logical Time in Distributed Systems
Logical clock

• Physical clock synchronization algorithms try to coordinate distributed clocks to reach a common value
  – Based on the estimation of transmission times
    • It can be hard to find a good estimation.
  – In several applications it is not important when things happened but in which order they happened

• Reliable way of ordering events is required!
Notes:

1. Two events occurred at some process $p_i$ happened in the same order as $p_i$ observes them.
2. When $p_i$ sends a message to $p_j$ the send event happens before the receive event.

- Lamport introduced the relation that captures the causal dependencies between events (causal order relation):
  - We denote with $\rightarrow_i$ the ordering relation between events in a process $p_i$.
  - We denote with $\rightarrow$ the happened-before relation between any pair of events.
Happened-Before Relation: Definition

- Two events \( e \) and \( e' \) are related by happened-before relation \( (e \rightarrow e') \) if:
  - \( \exists p_i \mid e \rightarrow_i e' \)
Happened-Before Relation: Definition

- Two events $e$ and $e'$ are related by happened-before relation ($e \rightarrow e'$) if:
  - $\exists p_i \mid e \rightarrow_i e'$
  - $\forall$ message $m$ $\text{send}(m) \rightarrow \text{receive}(m)$
    - $\text{send}(m)$ is the event of sending a message $m$
    - $\text{receive}(m)$ is the event of receipt of the same message $m$
Happened-Before Relation: Definition

- Two events \( e \) and \( e' \) are related by happened-before relation \( (e \rightarrow e') \) if:
  - \( \exists p_i \mid e \rightarrow_i e' \)
  - \( \forall \) message \( m \) \( \text{send}(m) \rightarrow \text{receive}(m) \)
    - \( \text{send}(m) \) is the event of sending a message \( m \)
    - \( \text{receive}(m) \) is the event of receipt of the same message \( m \)
  - \( \exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e') \)
    (happened-before relation is transitive)
Happened-Before Relation

- Using these three rules it is possible to define a causal-ordered sequence of events $e_1, e_2, \ldots, e_n$

- **Notes:**
  - The sequence $e_1, e_2, \ldots, e_n$ may not be unique
  - It may exists a pair of events $<e_1,e_2>$ such that $e_1$ and $e_2$ are not in happened-before relation
  - If $e_1$ and $e_2$ are not in happened-before relation then they are *concurrent* ($e_1||e_2$)
  - For any two events $e_1$ and $e_2$ in a distributed system, either
    - $e_1 \rightarrow e_2$
    - $e_2 \rightarrow e_1$
    - $e_1||e_2$
happened-before: example

\[ S_1 = \langle e_{1}^1, e_{1}^2, e_{2}^2, e_{3}^2, e_{3}^3, e_{3}^1, e_{1}^4, e_{1}^5, e_{2}^4 \rangle \]

\[ S_2 = \langle e_{3}^1, e_{2}^1, e_{3}^3, e_{4}^1, e_{5}^3 \rangle \]

Note: \( e_{1}^3 \) and \( e_{1}^2 \) are concurrent
Logical Clock

• The Logical Clock, introduced by Lamport, is a software counting register *monotonically* increasing its value
  – Logical clock is not related to physical clock
• Each process $p_i$ employs its logical clock $L_i$ to apply a *timestamp* to events
• $L_i(e)$ is the “logical” timestamp assigned, using the logical clock, by a process $p_i$ to event $e$
• **Property:**
  – If $e \rightarrow e'$ then $L(e) < L(e')$
• **Observation:**
  – The ordering relation obtained through logical timestamps is only a partial order. Consequently, timestamps could not be sufficient to relate two events
Scalar Logical Clock: an implementation

- Each process $p_i$ initializes its logical clock $L_i = 0 \ (\forall \ i = 1 \ldots N)$
- $p_i$ increases $L_i$ of 1 when it generates an event (either $send$ or $receive$)
  - $L_i = L_i + 1$
- When $p_i$ sends a message $m$
  - creates an event $send(m)$
  - increases $L_i$
  - timestamps $m$ with $t = L_i$
- When $p_i$ receives a message $m$ with timestamp $t$
  - Updates its logical clock $L_i = \max(t, L_i)$
  - Produces an event $receive(m)$
  - Increases $L_i$
Scalar Logical Clock: example

- $e^i_j$ is $j$-th event of process $p_i$
- $L_i$ is the logical clock of $p_i$

**Note:**
- $e^1_1 \rightarrow e^2_1$ and timestamps reflect this property
- $e^1_1 \parallel e^1_3$ and respective timestamps have the same value
- $e^1_2 \parallel e^1_3$ but respective timestamps have different values
Limits of Scalar Logical Clock

- Scalar logical clock can guarantee the following property
  - IF e → e’ then L(e) < L(e’)
- But it is not possible to guarantee
  - IF L(e) < L(e’) then e → e’

- **Consequently:**
  - It is not possible to determine, by analysing only scalar clocks, if two events are concurrent or correlated by the happened-before relation
- Mattern [1989] and Fridge [1991] proposed an improved version of logical clock where events are time-stamped with local logical clock and node identifier
  - **Vector Clock**
Logical Time and Ricart-Agrawala Mutual Exclusion Algorithm
Logical clock in distributed algorithms

Scalar Clock can be used to solve Lamport’s Mutual Exclusion problem in a distributed setting
Ricart-Agrawala’s algorithm: implementation (see also lecture notes)

- Local variables
  - #replies (initially 0)
  - State ∈ {Requesting, CS, NCS} (initially NCS)
  - Q pending requests queue (initially empty)
  - Last_Req (initially MAX_INT)
  - Num (initially 0)

- Algorithm:

\begin{align*}
\text{begin} \\
1. & \text{State = Requesting} \\
2. & \text{Num} = \text{Num} + 1; \text{Last}_\text{Req} = \text{num} \\
3. & \forall i = 1 \ldots N, \text{send REQUEST to } p_i \\
4. & \text{Wait until } \#\text{replies} == N - 1 \\
5. & \text{State} = \text{CS} \\
6. & \text{CS} \\
7. & \forall r \in Q, \text{send REPLY to } r \\
8. & Q = \emptyset; \text{State} = \text{NCS}; \#\text{replies} = 0; \\
& \text{Last}_\text{Req} = \text{MAX_INT} \\
\end{align*}

\begin{align*}
\text{Upon receipt REQUEST}(t) \text{ from } p_j \\
1. & \text{Num} = \max(t, \text{Num}) \\
2. & \text{If State} == \text{CS} \text{ or } (\text{State} == \text{Requesting} \text{ and } \{\text{Last}_\text{Req}, i\} < \{t, j\}) \\
3. & \text{Then insert in } Q\{t, j\} \\
4. & \text{Else send REPLY to } p_j \\
\end{align*}

\begin{align*}
\text{Upon receipt of REPLY from } p_j \\
1. & \#\text{replies} = \#\text{replies} + 1
\end{align*}
Ricart-Agrawala’s algorithm: example

P2 requires the CS
Ricart-Agrawala’s algorithm: example

P3 receives the request of P2

Num = 1
Last_Req = Num = 1

P1

P2

P3

Num = max(t, Num) =
= max(1, 0) = 1

Reply
Ricart-Agrawala’s algorithm: example

Also P1 tries to access the CS

Num = 1
Last_Req = Num = 1

Reply

Num = 1
Num = 1
Last_Req = Num = 1
Ricart-Agrawala’s algorithm: example

P1 receives the request of P2

Num = 1
Last_Req = Num = 1

Reply
Ricart-Agrawala’s algorithm: example

Num = 1
Last_Req = Num = 1

{Last_Req, i} < {t, j}?
{1, 1} < {1, 2}? YES

Reply

P1

P2

P3
Ricart-Agrawala’s algorithm: example

Num = 1
Last_Req = Num = 1

Num = max(t, Num) =
= max(1, 1) = 1

P1
P2
P3

NCS

R

Reply
Ricart-Agrawala’s algorithm: example

P1

Num = 1
Last_Req = Num = 1

R

1

Q

1,2

P2

Num = 1
Last_Req = Num = 1

R

1

Reply

1

Reply

P3

Num = 1

P3 receives the request of P1

Num = max(t, Num) = max(1, 1) = 1
Ricart-Agrawala’s algorithm: example

P2 receives the request of P1

P2 requests NCS

Num = 1
Last_Req = 1

Q_num = 1

{Last_Req, i} < {t, j}?

{1, 2} < {1, 1}? NO

P2 receives the request of P1
Ricart-Agrawala’s algorithm: example

P2 receives the request of P1

NCS

P1

Num = 1
Last_Req = 1

R

1

Q

1,2

NCS

P2

Num = 1
Last_Req = 1

1

1

R

NCS

P3

Num = 1

1

1

1

Num = 1

Reply

Reply

Reply

P2 receives the request of P1
Ricart-Agrawala’s algorithm: example

P2 receives the Reply sent by P3

P1

P2

P3

Num = 1
Last_Req = 1

Num = 1

#replies = 1

P2 receives the Reply sent by P3
Ricart-Agrawala’s algorithm: example

P1 receives the Reply sent by P2
Ricart-Agrawala’s algorithm: example

P1 receives the Reply sent from P3

#replies = 2

P1

P2

P3

P1 receives the Reply sent from P3
P2 receives the second Reply and accesses CS.

Ricart-Agrawala’s algorithm: example

P2 receives the second Reply and accesses CS.