Distributed Algorithms – 2g1513

Lecture 1b – by Ali Ghodsi
Models of distributed systems continued and logical time in distributed systems
Example algorithm:

\[
\begin{align*}
X & := 0; \\
\textbf{while} \ (X < 3) \ \textbf{do} \\
\quad & X = X + 1; \\
\textbf{endwhile} \\
X & := 1
\end{align*}
\]

Formally:

- States \( \{X_0, X_1, X_2\} \)
- Possible transitions \( \{X_0 \rightarrow X_1, X_1 \rightarrow X_2, X_2 \rightarrow X_1\} \)
- Start states \( \{X_0\} \)
A STS is formally described as the triple:

$$(C, \rightarrow, I)$$

Such that:

1. $C$ is a set of states
2. $\rightarrow$ is a subset of $C \times C$, describing the possible transitions ($\rightarrow \subseteq C \times C$)
3. $I$ is a subset of $C$ describing the initial states ($I \subseteq C$)

Note that the system may have several transitions from one state

E.g. $\rightarrow = \{X2 \rightarrow X1, X2 \rightarrow X0\}$
Local Algorithms

- The local algorithm will be modeled as a STS, in which the following three events happen:
  - A processor changes state from one state to a another state (internal event)
  - A processor changes state from one state to a another state, and sends a message to the network destined to another processor (send event)
  - A processor receives a message destined to it and changes state from one state to another (receive event)
Model of the distributed system

Based on the STS of a local algorithm, we can now define for the whole distributed system:

- Its configurations
  - We want it to be the state of all processes and the network
- Its initial configurations
  - We want it to be all possible configurations where every local algorithm is in its start state and an empty network
- Its transitions, and
  - We want each local algorithm state event (send, receive, internal) be a configuration transition in the distributed system
Let's do a simple distributed application, where a client sends a **ping**, and receives a **pong** from a server.

This is repeated indefinitely.
\[ M = \{ \text{ping}_{p2}, \text{pong}_{p1} \} \]

\[ (Z_{p1}, I_{p1}, \bar{I}_{p1}, \bar{S}_{p1}, \bar{R}_{p1}) \]
\[ Z_{p1} = \{ c_{\text{init}}, c_{\text{sent}} \} \]
\[ I_{p1} = \{ c_{\text{init}} \} \]
\[ \bar{I}_{p1} = \emptyset \]
\[ \bar{S}_{p1} = \{ (c_{\text{init}}, \text{ping}_{p2}, c_{\text{sent}}) \} \]
\[ \bar{R}_{p1} = \{ (c_{\text{sent}}, \text{pong}_{p1}, c_{\text{init}}) \} \]

\[ (Z_{p2}, I_{p2}, \bar{I}_{p2}, \bar{S}_{p2}, \bar{R}_{p2}) \]
\[ Z_{p2} = \{ s_{\text{init}}, s_{\text{recv}} \} \]
\[ I_{p2} = \{ s_{\text{init}} \} \]
\[ \bar{I}_{p2} = \emptyset \]
\[ \bar{S}_{p2} = \{ (s_{\text{recv}}, \text{pong}_{p1}, s_{\text{init}}) \} \]
\[ \bar{R}_{p2} = \{ (s_{\text{init}}, \text{ping}_{p2}, s_{\text{recv}}) \} \]
### Execution Example 3/9

**p1 (client)**

\[
(Z_{p1}, I_{p1}, \langle p_1 \rangle, \langle s_{p1} \rangle, \langle R_{p1} \rangle)
\]
\[
Z_{p1} = \{ c_{init}, c_{sent} \}
\]
\[
I_{p1} = \{ c_{init} \}
\]
\[
\langle s_{p1} \rangle = \{ (c_{init}, p_{init}) \}
\]
\[
\langle R_{p1} \rangle = \{ (c_{sent}, p_{init}) \}
\]

**p2 (server)**

\[
(Z_{p2}, I_{p2}, \langle p_2 \rangle, \langle s_{p2} \rangle, \langle R_{p2} \rangle)
\]
\[
Z_{p2} = \{ s_{init}, s_{rec} \}
\]
\[
I_{p2} = \{ s_{init} \}
\]
\[
\langle s_{p2} \rangle = \{ (s_{rec}, p_{init}) \}
\]
\[
\langle R_{p2} \rangle = \{ (s_{init}, p_{init}) \}
\]

\[
C = \{ (c_{init}, s_{init}, \emptyset), (c_{init}, s_{rec}, \emptyset), (c_{sent}, s_{init}, \emptyset), (c_{sent}, s_{rec}, \emptyset),
\]
\[
(c_{init}, s_{init}, \{ ping_{p2} \}), (c_{init}, s_{rec}, \{ ping_{p2} \}), (c_{sent}, s_{init}, \{ ping_{p2} \}), (c_{sent}, s_{rec}, \{ ping_{p2} \}),
\]
\[
(c_{init}, s_{init}, \{ pong_{p1} \}), (c_{init}, s_{rec}, \{ pong_{p1} \}), (c_{sent}, s_{init}, \{ pong_{p1} \}), (c_{sent}, s_{rec}, \{ pong_{p1} \})
\]

\[
I = \{ (c_{init}, s_{init}, \emptyset) \}
\]

\[
\rightarrow = \{ (c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ ping_{p2} \}),
\]
\[
(c_{sent}, s_{init}, \{ ping_{p2} \}) \rightarrow (c_{sent}, s_{rec}, \emptyset),
\]
\[
(c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ pong_{p1} \}),
\]
\[
(c_{sent}, s_{init}, \{ pong_{p1} \}) \rightarrow (c_{init}, s_{init}, \emptyset) \}
\]
### Execution Example 4/9

#### $p1$ (client)

\[
\begin{align*}
Z_{p1} &= \{c_{init}, c_{sent}\} \\
I_{p1} &= \{c_{init}\} \\
S_{p1} &= \{\langle c_{init}, \text{ping}_{p2}, c_{sent} \rangle \} \\
R_{p1} &= \{\langle c_{sent}, \text{pong}_{p1}, c_{init} \rangle \}
\end{align*}
\]

#### $p2$ (server)

\[
\begin{align*}
Z_{p2} &= \{s_{init}, s_{rec}\} \\
I_{p2} &= \{s_{init}\} \\
S_{p2} &= \{\langle s_{rec}, \text{pong}_{p1}, s_{init} \rangle \} \\
R_{p2} &= \{\langle s_{init}, \text{ping}_{p2}, s_{rec} \rangle \}
\end{align*}
\]

\[
I = \{ (c_{init}, s_{init}, \emptyset) \} \\
\rightarrow = \{(c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{\text{ping}_{p2}\}), \}
\]

\[
(c_{sent}, s_{init}, \{\text{ping}_{p2}\}) \rightarrow (c_{sent}, s_{rec}, \emptyset),
\]

\[
(c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{\text{pong}_{p1}\}),
\]

\[
(c_{sent}, s_{init}, \{\text{pong}_{p1}\}) \rightarrow (c_{init}, s_{init}, \emptyset)
\]

\[
E = ((c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{\text{ping}_{p2}\}),
\]

\[
(c_{sent}, s_{rec}, \emptyset), (c_{sent}, s_{init}, \{\text{pong}_{p1}\}),
\]

\[
(c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{\text{ping}_{p2}\}), \ldots)
\]

---

**$p1$ state:** $c_{init}$

**$p2$ state:** $s_{init}$
**Execution Example 5/9**

### p1 (client)

\[ Z_{p1}, I_{p1}, |I'_{p1}|, |S'_{p1}|, |R'_{p1}| \]

\[ Z_{p1} = \{ c_{init}, c_{sent} \} \]

\[ I_{p1} = \{ c_{init} \} \]

\[ I'_{p1} = \emptyset \]

\[ S'_{p1} = \{(c_{init}, ping_{p2}, c_{sent})\} \]

\[ R'_{p1} = \{(c_{sent}, pong_{p1}, c_{init})\} \]

### p2 (server)

\[ Z_{p2}, I_{p2}, |I'_{p2}|, |S'_{p2}|, |R'_{p2}| \]

\[ Z_{p2} = \{ s_{init}, s_{rec} \} \]

\[ I_{p2} = \{ s_{init} \} \]

\[ I'_{p2} = \emptyset \]

\[ S'_{p2} = \{(s_{rec}, pong_{p1}, s_{init})\} \]

\[ R'_{p2} = \{(s_{init}, ping_{p2}, s_{rec})\} \]

\[ I = \{ (c_{init}, s_{init}, \emptyset) \} \]

\[ \rightarrow = \{(c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ping_{p2}\})\}, \]

\[ (c_{sent}, s_{init}, \{ping_{p2}\}) \rightarrow (c_{sent}, s_{rec}, \emptyset), \]

\[ (c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{pong_{p1}\})\],

\[ (c_{sent}, s_{init}, \{pong_{p1}\}) \rightarrow (c_{init}, s_{init}, \emptyset)\} \]

\[ E = \{(c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{ping_{p2}\})\}, \]

\[ (c_{sent}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{pong_{p1}\})\],

\[ (c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{ping_{p2}\}), \ldots \} \]

**p1 state:** c_{init}

**p2 state:** s_{init}
**Execution Example 6/9**

<table>
<thead>
<tr>
<th><strong>p1 (client)</strong></th>
<th><strong>p2 (server)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>(Z_{p1}, I_{p1},</td>
<td>I'_{p1},</td>
</tr>
<tr>
<td>(Z_{p1} = {c_{init}, c_{sent}})</td>
<td>(Z_{p2} = {s_{init}, s_{rec}})</td>
</tr>
<tr>
<td>(I_{p1} = {c_{init}})</td>
<td>(I_{p2} = {s_{init}})</td>
</tr>
<tr>
<td>(</td>
<td>I'_{p1} = \emptyset])</td>
</tr>
<tr>
<td>(</td>
<td>S_{p1} = {(c_{init}, ping_{p2}, c_{sent})})</td>
</tr>
<tr>
<td>(</td>
<td>R_{p1} = {(c_{sent}, pong_{p1}, c_{init})})</td>
</tr>
</tbody>
</table>

\(I = \{ (c_{init}, s_{init}, \emptyset) \}\)

\(\rightarrow = \{ (c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ping_{p2}\})\),
\( (c_{sent}, s_{init}, \{ping_{p2}\}) \rightarrow (c_{sent}, s_{rec}, \emptyset), \)
\( (c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{pong_{p1}\})\),
\( (c_{sent}, s_{init}, \{pong_{p1}\}) \rightarrow (c_{init}, s_{init}, \emptyset)\}\)

\(E = \{ (c_{init}, s_{init}, \emptyset)\),
\( (c_{sent}, s_{init}, \{ping_{p2}\})\),
\( (c_{sent}, s_{rec}, \emptyset), (c_{sent}, s_{init}, \{pong_{p1}\})\),
\( (c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{ping_{p2}\})\), \ldots \)
**Execution Example 7/9**

### p1 (client)

\[ Z_{p1}, I_{p1}, I_{p1}' = S_{p1}, R_{p1} \]
\[ Z_{p1} = \{ c_{init}, c_{sent} \} \]
\[ I_{p1} = \{ c_{init} \} \]
\[ I_{p1}' = \emptyset \]
\[ S_{p1} = \{ (c_{init}, ping_{p2}, c_{sent}) \} \]
\[ R_{p1} = \{ (c_{sent}, pong_{p1}, c_{init}) \} \]

\[ I = \{ (c_{init}, s_{init}, \emptyset) \} \]
\[ \rightarrow = \{ (c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ ping_{p2} \}), \]
\[ (c_{sent}, s_{init}, \{ ping_{p2} \}) \rightarrow (c_{sent}, s_{rec}, \emptyset), \]
\[ (c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ pong_{p1} \}), \]
\[ (c_{sent}, s_{init}, \{ pong_{p1} \}) \rightarrow (c_{init}, s_{init}, \emptyset) \} \]

\[ E = ( (c_{init}, s_{init}, \emptyset) , (c_{sent}, s_{init}, \{ ping_{p2} \}) , \]
\[ (c_{sent}, s_{rec}, \emptyset) , (c_{sent}, s_{init}, \{ pong_{p1} \}) , \]
\[ (c_{init}, s_{init}, \emptyset) , (c_{sent}, s_{init}, \{ ping_{p2} \}) , \ldots ) \]
### Execution Example 8/9

**p1 (client)**

\[
\begin{align*}
Z_{p1} &= \{c_{init}, c_{sent}\} \\
I_{p1} &= \{c_{init}\} \\
S_{p1} &= \{c_{init} \text{ ping}_{p2}, c_{sent}\} \\
R_{p1} &= \{c_{sent} \text{ pong}_{p1}, c_{init}\}
\end{align*}
\]

\[
I = \{ (c_{init}, s_{init}, \emptyset) \} \\
\rightarrow = \{(c_{init}, s_{init}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{ping_{p2}\}), (c_{sent}, s_{init}, \{ping_{p2}\}) \rightarrow (c_{sent}, s_{rec}, \emptyset), (c_{sent}, s_{rec}, \emptyset) \rightarrow (c_{sent}, s_{init}, \{pong_{p1}\}), (c_{sent}, s_{init}, \{pong_{p1}\}) \rightarrow (c_{init}, s_{init}, \emptyset)\}
\]

\[
E = \{(c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{ping_{p2}\}), (c_{sent}, s_{rec}, \emptyset), (c_{sent}, s_{init}, \{pong_{p1}\}), (c_{init}, s_{init}, \emptyset), (c_{sent}, s_{init}, \{ping_{p2}\}), \ldots \}
\]

**p2 (server)**

\[
\begin{align*}
Z_{p2} &= \{s_{init}, s_{rec}\} \\
I_{p2} &= \{s_{init}\} \\
S_{p2} &= \{s_{rec} \text{ pong}_{p1}, s_{init}\} \\
R_{p2} &= \{(s_{init} \text{ ping}_{p2}, s_{rec})\}
\end{align*}
\]

---

- **p1 state:** c\text{sent}  
  *send<pong, p1>*

- **p2 state:** s\text{init}
### Execution Example 9/9

**p1 (client)**

- $(Z_{p1}, I_{p1}, I_{p1}^S, I_{p1}^R)$
- $Z_{p1} = \{ c_{\text{init}}, c_{\text{sent}} \}$
- $I_{p1} = \{ c_{\text{init}} \}$
- $I_{p1}^S = \emptyset$
- $I_{p1}^R = \{ (c_{\text{sent}} \text{ ping}_{p2}, c_{\text{sent}}) \}$

**p2 (server)**

- $(Z_{p2}, I_{p2}, I_{p2}^S, I_{p2}^R)$
- $Z_{p2} = \{ s_{\text{init}}, s_{\text{rec}} \}$
- $I_{p2} = \{ s_{\text{init}} \}$
- $I_{p2}^S = \emptyset$
- $I_{p2}^R = \{ (s_{\text{rec}} \text{ pong}_{p1}, s_{\text{init}}) \}$

$I = \{ (c_{\text{init}}, s_{\text{init}}, \emptyset) \}$

$\rightarrow = \{(c_{\text{init}}, s_{\text{init}}, \emptyset) \rightarrow (c_{\text{sent}}, s_{\text{init}}, \{ \text{ping}_{p2} \}),$

$(c_{\text{sent}}, s_{\text{init}}, \{ \text{ping}_{p2} \}) \rightarrow (c_{\text{sent}}, s_{\text{rec}}, \emptyset),$

$(c_{\text{sent}}, s_{\text{rec}}, \emptyset) \rightarrow (c_{\text{sent}}, s_{\text{init}}, \{ \text{pong}_{p1} \}),$

$(c_{\text{sent}}, s_{\text{init}}, \{ \text{pong}_{p1} \}) \rightarrow (c_{\text{init}}, s_{\text{init}}, \emptyset)\}$

$E = \{(c_{\text{init}}, s_{\text{init}}, \emptyset), (c_{\text{sent}}, s_{\text{init}}, \{ \text{ping}_{p2} \}),$

$(c_{\text{sent}}, s_{\text{rec}}, \emptyset), (c_{\text{sent}}, s_{\text{init}}, \{ \text{pong}_{p1} \}),$

$(c_{\text{init}}, s_{\text{init}}, \emptyset), (c_{\text{sent}}, s_{\text{init}}, \{ \text{ping}_{p2} \}), ... \}$

---

**p1 state:** $c_{\text{init}}$

**rec <pong, p1>**

**p2 state:** $s_{\text{init}}$
Applicable events

- Any internal event \(e = (c, d) \in I'_{pi}\) is said to be applicable in a configuration \(C = (c_{p1}, \ldots, c_{pi}, \ldots, c_{pn}, M)\) if \(c_{pi} = c\)
  - If event \(e\) is applied, we get \(e(C) = (c_{p1}, \ldots, d, \ldots, c_{pn}, M)\)

- Any send event \(e = (c, m, d) \in S_{pi}\) is said to be applicable in a configuration \(C = (c_{p1}, \ldots, c_{pi}, \ldots, c_{pn}, M)\) if \(c_{pi} = c\)
  - If event \(e\) is applied, we get \(e(C) = (c_{p1}, \ldots, d, \ldots, c_{pn}, M \cup \{m\})\)

- Any receive event \(e = (c, m, d) \in R_{pi}\) is said to be applicable in a configuration \(C = (c_{p1}, \ldots, c_{pi}, \ldots, c_{pn}, M)\) if \(c_{pi} = c\) and \(m \in M\)
  - If event \(e\) is applied, we get \(e(C) = (c_{p1}, \ldots, d, \ldots, c_{pn}, M \setminus \{m\})\)
Order of events

- The following theorem shows an important result:
  - The order in which two applicable events are executed is not important!

**Theorem:**

- Let \( e_p \) and \( e_q \) be two events on two different processors \( p \) and \( q \) which are both applicable in configuration \( \gamma \). Then \( e_p \) can be applied to \( e_q(\gamma) \), and \( e_q \) can be applied to \( e_p(\gamma) \).
- Moreover, \( e_p(e_q(\gamma)) = e_q(e_p(\gamma)) \).
To avoid a proof by cases ($3 \times 3 = 9$ cases) we represent all three event types in one abstraction.

We let the quadtuple $(c, X, Y, d)$ represent any event:
- $c$ is the initial state of the processor
- $X$ is a set of messages that will be received by the event
- $Y$ is a set of messages that will be sent by the event
- $d$ is the state of the processor after the event

Examples:
- $(c_{init}, \emptyset, \{ping\}, c_{sent})$ represents a send event
- $(s_{init}, \{pong\}, \emptyset, s_{rec})$ represents a receive event
- $(c1r, \emptyset, \emptyset, c2)$ represents an internal event

Any such event $e_p = (c, X, Y, d)$, at $p$, is applicable in a state $\gamma = \{\ldots, c_p, \ldots, M\}$ if and only if $c_p = c$ and $X \subseteq M$. 
Order of events

**Proof:**

Let $e_p = \{c, X, Y, d\}$ and $e_q = \{e, Z, W, f\}$ and $\gamma = \{\ldots, c_p, \ldots, c_q, \ldots, M\}$

As both $e_p$ and $e_q$ are applicable in $\gamma$ we know that $c_p = c$, $c_q = e$, $X \subseteq M$, and $Z \subseteq M$.

$e_q(\gamma) = \{\ldots, c_p, \ldots, f, \ldots, (M-Z) \cup W\}$,

$c_p$ is untouched and $c_p = c$, and $X \subseteq (M-Z) \cup W$ as $X \cap Z = \emptyset$, hence $e_p$ is applicable in $e_q(\gamma)$

Similar argument to show that $e_q$ is applicable in $e_p(\gamma)$
Order of events

Proof:

Let's prove \( e_p(e_q(\gamma)) = e_q(e_p(\gamma)) \)

\[ e_q(\gamma) = \{ \ldots, c_p, \ldots, f, \ldots, (M-Z) \cup W \} \]
\[ e_q(e_p(\gamma)) = \{ \ldots, d, \ldots, f, \ldots, ((M-Z) \cup W-X) \cup Y \} \]

\[ e_p(\gamma) = \{ \ldots, d, \ldots, c_q, \ldots, (M-X) \cup Y \} \]
\[ e_q(e_p(\gamma)) = \{ \ldots, d, \ldots, f, \ldots, ((M-X) \cup Y-Z) \cup W \} \]

\[ ((M-Z) \cup W-X) \cup Y = ((M-X) \cup Y-Z) \cup W \]

Because \( X \cap Z = \emptyset, W \cap X = \emptyset, Y \cap Z = \emptyset \)

Both LHS and RHS can be transformed to \( (M \cup W \cup Y-Z) - X \)
Exact order does not always matter

In two cases the theorem does not apply:

- If $p=q$, i.e. when the events occur on different processes
  - They would not both be applicable if they are executed out of order

- If one is a sent event, and the other is the corresponding receive event
  - They cannot be both applicable

In such cases, we say that the two events are causally related!
The relation $\leq_H$ on the events of an execution, called *causal order*, is defined as the smallest relation such that:

- If $e$ occurs before $f$ on the same process, then $e \leq_H f$
- If $s$ is a send event and $r$ its *corresponding* receive event, then $s \leq_H r$
- $\leq_H$ is transitive. I.e. If $a \leq_H b$ and $b \leq_H c$ then $a \leq_H c$
- $\leq_H$ is reflexive. I.e. If $a \leq_H a$ for any event $a$ in the execution
- Two events, $a$ and $b$, are *concurrent* iff $a \not\leq_H b$ and $b \not\leq_H a$ holds
Example of Causally Related events

Concurrent Events

Time-space diagram

Causally Related Events

Causally Related Events

Concurrent Events

p1

p2

p3

time
Equivalence of Executions:

**Computations**

**Computation Theorem:**

- Let $E$ be an execution $E=(\gamma_1, \gamma_2, \gamma_3\ldots)$, and $V$ be the sequence of events $V=(e_1, e_2, e_3\ldots)$ associated with it.
  - i.e. applying $e_k(\gamma_k)=\gamma_{k+1}$ for all $k \geq 1$

- A permutation $P$ of $V$ that preserves causal order, and starts in $\gamma_1$, defines a unique execution with the same number of events, and if finite, $P$ and $V$’s final configurations are the same.
  - $P=(f_1, f_2, f_3\ldots)$ preserves the causal order of $V$ when for every pair of events $f_i \leq_H f_j$ implies $i<j$. 

Equivalence of executions

If two executions $F$ and $E$ have the same collection of events, and their causal order is preserved, $F$ and $E$ are said to be equivalent executions, written $F \sim E$

$F$ and $E$ could have different permutation of events as long as causality is preserved!
Equivalent executions form equivalence classes where every execution in class is equivalent to the other executions in the class.

I.e. the following always holds for executions:

- ~ is reflexive
  - I.e. If $a~a$ for any execution

- ~ is symmetric
  - I.e. If $a~b$ then $b~a$ for any executions $a$ and $b$

- ~ is transitive
  - If $a~b$ and $b~c$, then $a~c$, for any executions $a$, $b$, $c$

Equivalence classes are called *computations* of executions
Example of equivalent executions

All three executions are part of the same computation, as causality is preserved.

Same color ~ Causally related
The computation theorem gives two important results

**Result 1:**
There is no distributed algorithm which can observe the order of the sequence of events (that can “see” the time-space diagram)

**Proof:**
1. Assume such an algorithm exists. Assume process $p$ knows the order in the final configuration
2. Run two different executions of the algorithm that preserve the causality.
3. According to the computation theorem their final configurations should be the same, but in that case, the algorithm cannot have observed the actual order of events as they differ
Two important results (2)

 Result 2:

The computation theorem does not hold if the model is extended such that each process can read a local hardware clock.

Proof:
Similarly, assume a distributed algorithm in which each process reads the local clock each time a local event occurs.

The final configuration of different causality preserving executions will have different clock values, which contradicts the computation theorem.
Lamport Logical Clock (informal)

Each process has a local *logical* clock, which is kept in a local variable $\theta_p$ for process $p$, which is initially 0.

The logical clock is updated for each local event on process $p$ such that:

- If an *internal* or *send* event occurs:
  
  $\theta_p = \theta_p + 1$

- If a receive event happens on a message from process $q$:
  
  $\theta_p = \max(\theta_p, \theta_q) + 1$

Lamport logical clocks guarantee that:

- If $a \leq_H b$, then $\theta_p \leq \theta_q$,
  
  where $a$ and $b$ happen on $p$ and $q$.
Example of equivalent executions
Vector Timestamps: useful implication

- Each process $p$ keeps a vector $v_p[n]$ of $n$ positions for system with $n$ processes, initially $v_p[p]=1$ and $v_p[i]=0$ for all other $i$.
- The logical clock is updated for each local event on process $p$ such that:
  - If any event: $v_p[p]=v_p[p]+1$
  - If a receive event happens on a message from process $q$: $v_p[x]=\max(v_p[x], v_q[x])$, for $1 \leq x \leq n$

- We say $v_p \leq v_q$ if $v_p[x] \leq v_q[x]$ for $1 \leq x \leq n$.
- If not $v_p \leq v_q$ and not $v_q \leq v_p$ then $v_p$ is concurrent with $v_q$.

- Lamport logical clocks guarantee that:
  - If $v_p \leq v_q$ then $a \leq_H b$, and $a \leq_H b$, then $v_p \leq v_q$, where $a$ and $b$ happen on $p$ and $q$. 

-
Example of Vector Timestamps

This is great! But cannot be done with smaller vectors than size n, for n processes
Useful Scenario: what is most recent?

\( p_2 \) examines the two messages it receives, one from \( p_1 \) \([4,0,0]\) and one from \( p_2 \) \([5,0,3]\) and deduces that the information from \( p_1 \) is the oldest \((4,0,0) \leq (5,0,3)\).
The total order of executions of events is not always important
- Two different executions could yield the same “result”

Causal order matters:
- Order of two events on the same process
- Order of two events, where one is a send and the other one a corresponding receive
- Order of two events, that are transitively related according to above

Executions which contain permutations of each others event such that causality is preserved are called equivalent executions

Equivalent executions form equivalence classes called *computations*.
- Every execution in a computation is equivalent to every other execution in its computation

Vector timestamps can be used to determine causality
- Cannot be done with smaller vectors than size n, for n processes