Distributed Systems

Logical Clocks

Paul Krzyzanowski
pxk@cs.rutgers.edu

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.
Logical clocks

Assign sequence numbers to messages
- All cooperating processes can agree on order of events
- vs. physical clocks: time of day

Assume no central time source
- Each system maintains its own local clock
- No total ordering of events
  - No concept of happened-when
Happened-before

Lamport’s “happened-before” notation

\[ a \rightarrow b \]  event \( a \) happened before event \( b \)

e.g.: \( a \): message being sent, \( b \): message receipt

Transitive:

if \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \)
Logical clocks & concurrency

Assign “clock” value to each event
- if $a \to b$ then $\text{clock}(a) < \text{clock}(b)$
- since time cannot run backwards

If $a$ and $b$ occur on different processes that do not exchange messages, then neither $a \to b$ nor $b \to a$ are true
- These events are concurrent
Event counting example

- Three systems: $P_0, P_1, P_2$
- Events $a, b, c, ...$
- Local event counter on each system
- Systems occasionally communicate
Event counting example
Event counting example

Bad ordering:

\[ e \rightarrow h \]
\[ f \rightarrow k \]
Lamport’s algorithm

- Each message carries a timestamp of the sender’s clock

- When a message arrives:
  - if receiver’s clock < message timestamp
    set system clock to (message timestamp + 1)
  - else do nothing

- Clock must be advanced between any two events in the same process
Lamport’s algorithm

Algorithm allows us to maintain time ordering among related events

- Partial ordering
Event counting example
Summary

• Algorithm needs monotonically increasing software counter

• Incremented at least when events that need to be timestamped occur

• Each event has a Lamport timestamp attached to it

• For any two events, where $a \rightarrow b$:
  $$L(a) < L(b)$$
Problem: Identical timestamps

Concurrent events (e.g., a & i) \textit{may} have the same timestamp ... or not
Unique timestamps (total ordering)

We can force each timestamp to be unique

- Define *global logical timestamp* \((T_i, i)\)
  - \(T_i\) represents local Lamport timestamp
  - \(i\) represents process number (globally unique)
    - E.g. (host address, process ID)

- Compare timestamps:
  \((T_i, i) < (T_j, j)\)
  if and only if
  \(T_i < T_j\) or
  \(T_i = T_j\) and \(i < j\)

Does not relate to event ordering
Unique (totally ordered) timestamps

P₁

1.1
2.1
3.1
4.1
5.1
6.1

P₂

1.2
2.1
3.1
4.1
5.1
6.1
7.1

P₃

1.3
2.1
3.1
4.1
5.1
6.1
7.1

j

k

P₁ → P₂ → P₃
Problem: Detecting causal relations

If \( L(e) < L(e') \)
- Cannot conclude that \( e \rightarrow e' \)

Looking at Lamport timestamps
- Cannot conclude which events are causally related

Solution: use a \textit{vector clock}
Vector clocks

Rules:

1. Vector initialized to 0 at each process
   \[ V_i[j] = 0 \text{ for } i, j = 1, \ldots, N \]

2. Process increments its element of the vector in local vector before timestamping event:
   \[ V_i[i] = V_i[i] + 1 \]

3. Message is sent from process \( P_i \) with \( V_i \) attached to it

4. When \( P_j \) receives message, compares vectors element by element and sets local vector to higher of two values
   \[ V_j[i] = \max(V_i[i], V_j[i]) \text{ for } i = 1, \ldots, N \]
Comparing vector timestamps

**Define**

\[
\begin{align*}
V &= V' \text{ iff } V[i] = V'[i] \text{ for } i = 1 \ldots N \\
V &\leq V' \text{ iff } V[i] \leq V'[i] \text{ for } i = 1 \ldots N
\end{align*}
\]

**For any two events** \( e, e' \)

if \( e \rightarrow e' \) then \( V(e) < V(e') \)

- Just like Lamport's algorithm

if \( V(e) < V(e') \) then \( e \rightarrow e' \)

**Two events are concurrent if neither**

\( V(e) \leq V(e') \) nor \( V(e') \leq V(e) \)
Vector timestamps

(0,0,0)  
P_1 - a - b - c - d - (0,0,0)  
P_2 - c - d - f - (0,0,0)  
P_3 - e - f - (0,0,0)
Vector timestamps

(a, 0, 0) (1, 0, 0)

Event timestamp

a (1, 0, 0)
Vector timestamps

Event timestamp

<table>
<thead>
<tr>
<th>Event</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1,0,0)</td>
</tr>
<tr>
<td>b</td>
<td>(2,0,0)</td>
</tr>
</tbody>
</table>
Vector timestamps

Event          timestamp
  a             (1,0,0)
  b             (2,0,0)
  c             (2,1,0)
Vector timestamps

<table>
<thead>
<tr>
<th>Event</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1,0,0)</td>
</tr>
<tr>
<td>b</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>c</td>
<td>(2,1,0)</td>
</tr>
<tr>
<td>d</td>
<td>(2,2,0)</td>
</tr>
</tbody>
</table>

The diagram shows three processes, P₁, P₂, and P₃, each marked with a timestamp. The events a, b, c, and d are annotated with their respective timestamps.
Vector timestamps

Event | timestamp
--- | ---
a | (0,0,1)
b | (0,0,0)
c | (2,0,0)
d | (1,0,0)
e | (0,0,0)
f | (0,0,0)
Vector timestamps

<table>
<thead>
<tr>
<th>Event</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1,0,0)</td>
</tr>
<tr>
<td>b</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>c</td>
<td>(2,1,0)</td>
</tr>
<tr>
<td>d</td>
<td>(2,2,0)</td>
</tr>
<tr>
<td>e</td>
<td>(0,0,1)</td>
</tr>
<tr>
<td>f</td>
<td>(2,2,2)</td>
</tr>
</tbody>
</table>
Vector timestamps

<table>
<thead>
<tr>
<th>Event</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1,0,0)</td>
</tr>
<tr>
<td>b</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>c</td>
<td>(2,1,0)</td>
</tr>
<tr>
<td>d</td>
<td>(2,2,0)</td>
</tr>
<tr>
<td>e</td>
<td>(0,0,1)</td>
</tr>
<tr>
<td>f</td>
<td>(2,2,2)</td>
</tr>
</tbody>
</table>

Concurrent events:
- b and c
- d and f
- e
Vector timestamps

Event | timestamp
--- | ---

a | (1,0,0)
b | (2,0,0)
c | (2,1,0)
d | (2,2,0)
e | (0,0,1)
f | (2,2,2)

concurrent events
Vector timestamps

Event | timestamp
---|---
a | (1,0,0)
b | (2,0,0)
c | (2,1,0)
d | (2,2,0)
e | (0,0,1)
f | (2,2,2)

concurrent events
Vector timestamps

Event | timestamp
--- | ---
a | (1,0,0)
b | (2,0,0)
c | (2,1,0)
d | (2,2,0)
e | (0,0,1)
f | (2,2,2)

concurrent events
Summary: Logical Clocks & Partial Ordering

• Causality
  - If \( a \rightarrow b \) then event \( a \) can affect event \( b \)

• Concurrency
  - If neither \( a \rightarrow b \) nor \( b \rightarrow a \) then one event cannot affect the other

• Partial Ordering
  - Causal events are sequenced

• Total Ordering
  - All events are sequenced
The end.