Distributed Systems

06. Logical clocks

Paul Krzyzanowski
Rutgers University
Fall 2017
Logical clocks

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

• Assume multiple actors (processes)
  - Each process has a unique ID
  - Each process has its own incrementing counter
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 a “clock” value to each event
- if $a \rightarrow b$ 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 $a \rightarrow b$ nor $b \rightarrow a$ are true
- These events are **concurrent**
- Otherwise, they are **causal**
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:

- $e \rightarrow h$ but $5 \geq 2$
- $f \rightarrow k$ but $6 \geq 2$
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

Applying Lamport’s algorithm

We have good ordering where we used to have bad ordering:

- $e \rightarrow h$ and $5 < 6$
- $f \rightarrow k$ and $6 < 7$
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

\[ a \rightarrow b, b \rightarrow c, \ldots : \text{ local events sequenced} \]
\[ i \rightarrow c, f \rightarrow d, d \rightarrow g, \ldots : \text{ Lamport imposes a} \]
\[ \text{send} \rightarrow \text{receive} \text{ relationship} \]

Concurrent events (e.g., \( b \) & \( g \); \( i \) & \( k \)) \text{ may} have the same timestamp \ldots \text{ 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
  \[T_i < T_j \text{ or } T_i = T_j \text{ and } i < j\]

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

P₁

a
1.1

b
2.1

c
3.1

d
4.1

e
5.1

f
6.1

P₂

g
1.2

h
6.2

i
7.2

j

P₃

1.3

k
7.3
Problem: Detecting causal relations

If \( L(e) < L(e') \)

– We cannot conclude that \( e \rightarrow e' \)

By looking at Lamport timestamps

– We cannot conclude which events are causally related

Solution: use a vector clock

Vector clocks are a way to prove the sequence of events by keeping version history based on each process that made changes to an object
Example

- Group of processes: Alice, Bob, Cindy, David
- They concurrently modify one object: “what should we eat?”
- Each process keeps a local counter

Alice writes the value & sends to group

Alice: 1
Pizza

Bob reads (“Pizza”, <alice:1>), modifies the value & sends to group

Alice: 1, Bob: 1
Chinese

Bob’s version updates Alice’s

Alice reads (“Chinese”, <alice:1, bob:1>), modifies the value & sends to group

Alice: 2, Bob: 1
Moroccan

Alice makes changes over Bob’s

Receiver <alice: 1, bob:1> is causal to & follows <alice: 1>

Receiver <alice: 2, bob:1> is causal to & follows <alice: 1, bob:1>
Cindy modifies & sends to group

Alice: 2, Bob: 1, Cindy: 1

Thai

Bob *concurrently* modifies & sends to group

Alice: 2, Bob: 2

Chinese

Cindy & Bob’s changes are concurrent – members must resolve conflict

Receiver

<alice: 2, bob:1, cindy:1> is *concurrent* with <alice: 1, bob:2>
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
   \[ V_j[i] = \max(V_i[i], V_j[i]) \text{ for } i = 1, \ldots, N \]

For example,
received: \([0, 5, 12, 1]\), have: \([2, 8, 10, 1]\)
new timestamp: \([2, 8, 12, 1]\)
Comparing vector timestamps

Define

\[ V = V' \text{ iff } V[i] = V'[i] \text{ for } i = 1 \ldots N \]
\[ V \leq V' \text{ iff } V[i] \leq V'[i] \text{ for } i = 1 \ldots N \]

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

\[ V(e) \leq V(e') \text{ nor } V(e') \leq V(e) \]
Vector timestamps

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

\[
\begin{array}{cccc}
(0,0,0) & (1,0,0) & (0,0,0) & (0,0,0) \\
P_1 & a & b & c & d & e & f \\
(0,0,0) & & & & & & \\
P_2 & a & b & c & d & e & f \\
(0,0,0) & & & & & & \\
P_3 & a & b & c & d & e & f \\
\end{array}
\]

Event timestamp

\begin{align*}
\text{Event} & \quad \text{timestamp} \\
a & \quad (1,0,0)
\end{align*}
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>
</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>
</tbody>
</table>

Event timestamps:

- P1: (1,0,0)
- P2: (2,0,0)
- P3: (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></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>
Vector timestamps

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

Event       timestamp       concurrent events
a           (1,0,0)           
b           (2,0,0)           
c           (2,1,0)           
d           (2,2,0)           
e           (0,0,1)           
f           (2,2,2)           

P₁ ———— (1,0,0) ———— P₂ ———— (2,2,0) ———— P₃ ———— (2,2,2)
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
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
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
Generalizing Vector Timestamps

• A “vector” can be a list of tuples:
  – For processes \( P_1, P_2, P_3, \ldots \)
  – Each process has a globally unique Process ID, \( P_i \) (e.g., MAC_address:PID)
  – Each process maintains its own timestamp: \( T_{P1}, T_{P2}, \ldots \)
  – Vector: \{ \langle P_1, T_{P1} \rangle, \langle P_2, T_{P2} \rangle, \langle P_3, T_{P3} \rangle, \ldots \} \)

• Any one process may have only partial knowledge of others
  – New timestamp for a received message:
    • Compare all matching sets of process IDs: set to highest of values
    • Any non-matched \( \langle P, T \rangle \) sets get added to the timestamp
  – For a happened-before relation:
    • At least one set of process IDs must be common to both timestamps
    • Match all corresponding \( \langle P, T \rangle \) sets: A:\( \langle P_i, T_a \rangle \), B:\( \langle P_i, T_b \rangle \)
    • If \( T_a \leq T_b \) for all common processes \( P \), then \( A \rightarrow B \)
Vector Clocks Summary

• Vector clocks give us a way of identifying which events are causally related

• We are guaranteed to get the sequencing correct

• But
  – The size of the vector increases with more actors
    … and the entire vector must be stored with the data.
  – Comparison takes more time than comparing two numbers
  – What if messages are concurrent?
    • App will have to decide how to handle conflicts
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