Logical Time

CSE 486/586: Distributed Systems

Ethan Blanton
Department of Computer Science and Engineering
University at Buffalo
Time Synchronization

As we have seen, time synchronization is hard.

Often, what we actually care about is causality, not time.

Could some event have caused another event?

If we can establish this, we may not need physical time!
Logical clocks were first introduced by Lamport in 1978 [2]. They address ordering without requiring time synchronization. Not all problems can be solved with logical clocks!
This lecture has another required reading [1].

You are expected to keep up with required readings.

You should have already read all previous required readings!

They may show up on the Midterm/Final, such as:

A centralized failure detector model reduces communication overhead, but violates the end-to-end-principle. Explain why it does not preserve the end-to-end principle, and discuss the trade-offs that it makes in terms of communication complexity and robustness versus end-to-end failure detection.

This is an upper level course, read and think! Ask questions!
Event Ordering

Logical clocks directly encode the has甚么before relationship.

This establishes three possible conditions for events $e_1$ and $e_2$:
- $e_1$ happens before $e_2$
- $e_2$ happens before $e_1$
- Neither event happens before the other, they are concurrent

This is a partial ordering.
Notation

If $e_1$ happens before $e_2$, we say $e_1 \to e_2$.

If $e_1$ does not happen before $e_2$, we say $e_1 \not\to e_2$.

Note that this does not mean that $e_2$ happens before $e_1$!

If $e_1 \leftrightarrow e_2$ and $e_2 \leftrightarrow e_1$, then $e_1$ and $e_2$ are concurrent.
Events in a Process

The events within a single process form a total ordering.

Every event in the process happens before the next, sequentially.

For every event within a process, either $p_1 \rightarrow p_2$ or $p_2 \rightarrow p_1$.

This implies that processes have a single thread of control.

We conventionally number these events in numeric order. (That is, $p_1 \rightarrow p_2$.)
Messages

Sending and receipt of messages are events.

Sending a message happens before the message is received.

Suppose that:
- Message $m$ is sent from process $P$ as event $p_i$
- Process $Q$ receives $m$ as event $q_j$

Therefore $p_i \rightarrow q_j$. 

\[ \begin{array}{c}
\text{P} \\
p_i \\
\text{Q} \\
m \\
\text{Time} \\
q_j
\end{array} \]
Transitivity

Happens before is transitive.

If $e_i \rightarrow e_j$ and $e_j \rightarrow e_k$, then $e_i \rightarrow e_k$.

This allows messages to order events between processes.
Transitivity

Happens before is **transitive**.

If $e_i \rightarrow e_j$ and $e_j \rightarrow e_k$, then $e_i \rightarrow e_k$.

This allows messages to order events between processes.
Concurrent Events

Concurrent events can only occur between processes.
Concurrent Events

Concurrent events can only occur between processes.

$r_1$ and $p_1$ are concurrent.
Concurrent Events

Concurrent events can only occur between processes.

$r_1$ and $p_2$ are concurrent.
Concurrent Events

Concurrent events can only occur between processes.

$r_1 \rightarrow p_3$. 
Lamport Clocks

Lamport clocks number events with a logical timestamp.

The rules are simple:

- Every process starts with a timestamp of 1.
- Every time a process takes an action, it increments its timestamp.
- Sending a message is an action.
- Messages include the timestamp of their action.
- Receiving a message is an action.
- After reception, processes set their timestamp to the maximum of their local timestamp and the message timestamp plus 1.
Timestamp Example

These timestamps follow the Lamport clock rules.

If $e_1 \rightarrow e_2$, the timestamp of $e_1$ is **numerically less than** the timestamp of $e_2$. 

© 2023 Ethan Blanton / CSE 486/586: Distributed Systems
Timestamp Example

Note that *concurrency is ambiguous* in the timestamps.

These points are concurrent.
Timestamp Example

Note that *concurrency is ambiguous* in the timestamps.

So are these!
Timestamp Example

Note that concurrency is ambiguous in the timestamps.

\[ e_1 \text{ and } e_2 \text{ are both concurrent with } e_3 \text{, but } e_1 \rightarrow e_2! \]
Timestamp Example

Note that concurrency is ambiguous in the timestamps.

…and $e_4$ and $e_2$ are concurrent, too!
Causality

Lamport clocks approximate causality:

If the timestamp of $e_1 <$ the timestamp of $e_2$, then $e_1$ could have caused $e_2$.

If $e_1 > e_2$, then $e_1$ could not have caused $e_2$.

The mapping is not perfect, with false positives.

There are no false negatives.
Vector Clocks

Vector clocks associate more than one timestamp with an event [3].

Each process has its own timestamp.

Each event is timestamped with the causality of every process.

This provides a tighter mapping with fewer false positives.

There are still no false negatives.
Vector Clock Rules

Every process $P_i$ keeps a vector of clock values.

There is one vector entry for each process.

$P_i$ can increment only the $i^{th}$ entry.

Each process takes the max of every vector position on message receipt.
Vector Clock Ordering

For vector $v = \langle p_0, \ldots, p_n \rangle$ and another $u$:

$u = v$ iff $\forall_{i=0}^{n} u[i] = v[i]$

$u \leq v$ iff $\forall_{i=0}^{n} u[i] \leq v[i]$

$u < v$ iff $u \leq v$ and $u \neq v$

$u \parallel v$ iff $\neg(u < v)$ and $\neg(v < u)$
Vector Clock Example

\[ \langle 1,0,0,0 \rangle \]
\[ \langle 0,1,0,0 \rangle \]
\[ \langle 0,0,1,1 \rangle \]
\[ \langle 0,1,2,1 \rangle \]
\[ \langle 0,1,3,1 \rangle \]

\[ \langle 2,1,3,1 \rangle \]

\[ \langle 0,0,0,1 \rangle \]

\[ \langle 0,1,0,0 \rangle \parallel \langle 0,1,3,1 \rangle \]

\( e_1 \) is unambiguously concurrent with \( e_2 \) because \( \langle 1,0,0,0 \rangle \parallel \langle 0,1,0,0 \rangle \parallel \langle 0,0,1,1 \rangle \parallel \langle 0,1,2,1 \rangle \parallel \langle 0,1,3,1 \rangle \parallel \langle 2,1,3,1 \rangle \parallel \langle 0,0,0,1 \rangle \).
Vector Clock Example

\[\langle 1, 0, 0, 0 \rangle \parallel \langle 0, 1, 3, 1 \rangle\]
Disadvantages of Vector Clocks

Vector clocks have **better precision** than Lamport clocks.

They identify **concurrent events** more precisely.

However, they **require more state**.

For **large numbers of processes** they may be impractical.
Both Lamport and vector clocks can provide a total ordering. This requires breaking ties between concurrent events.

Some arbitrary mechanism can be used; e.g.:

- process IDs for Lamport clocks
- numerical order for vector clocks
  (For example: \((1, 2, 3, 4)\) comes before \((1, 2, 3, 5)\))
- Supplementary physical timestamps

This total ordering is not physical time ordering!
Summary

- Logical clocks track **causality** of events
- Lamport clocks use a **single integer** to define causality
- Vector clocks provide **greater precision** than Lamport clocks, but require more state
- Logical clock orderings can be **partial** or **total**
Next Time …
References I

Required Readings


Optional Readings

References II

Copyright 2023 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.