Clocks, Event Ordering, and Global Predicate Computation
Events and Histories

- Processes execute sequences of events.
- Events can be of 3 types: local, send, and receive.
- $e^i_p$ is the $i$-th event of process $p$.
- The local history $h_p$ of process $p$ is the sequence of events executed by process $p$. 
Observation 1:
Events in a local history are **totally ordered**

Observation 2:
For every message \( m \), \( \text{send}(m) \) precedes \( \text{receive}(m) \)
Happened-before (Lamport[1978])

A binary relation $\rightarrow$ defined over events

1. if $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$

2. if $e_i = \text{send}(m)$ and $e_j = \text{receive}(m)$, then $e_i \rightarrow e_j$

3. if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$
Lamport Clocks

Each process maintains a local variable $LC$

$$LC(e) \equiv \text{value of } LC \text{ for event } e$$

$$LC(e_p^i) < LC(e_p^{i+1})$$

$$LC(e_p^i) < LC(e_q^j)$$
Increment Rules

\[
LC(e_p^{i+1}) = LC(e_p^i) + 1
\]

\[
LC(e_q^j) = \max(LC(e_q^{j-1}), LC(e_p^i)) + 1
\]

Timestamp \( m \) with \( TS(m) = LC(send(m)) \)
Discussion

What are the strengths of Lamport clocks?

What are the limitations of Lamport clocks?

What model assumptions are too constraining in Lamport’s clock paper?
Example of Global Predicate

Setting: Locks in distributed system

Objects locked by nodes and moved to the node that is currently modifying it

Nodes requesting the object/lock, send a message to the current node locking it and blocks for a response

How do we detect deadlocks in this scenario?
Global States & Clocks

Need to reason about global states of a distributed system

Global state: processor state + communication channel state

Consistent global state: causal dependencies are captured

Use virtual clocks to reason about the timing relationships between events on different nodes
Space-Time diagrams

A graphic representation of a distributed execution

H and → impose a partial order
Cuts

A cut $C$ is a subset of the global history of $H$

The frontier of $C$ is the set of events $e_{1}^{c_1}, e_{2}^{c_2}, \ldots, e_{n}^{c_n}$
Consistent cuts and consistent global states

A cut is consistent if

$$\forall e_i, e_j : e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C$$

A **consistent global state** is one corresponding to a consistent cut
What $p_0$ sees

Not a consistent global state: the cut contains the event corresponding to the receipt of the last message by $p_3$ but not the corresponding send event.
Can we use Lamport Clocks as part of a mechanism to get globally consistent states?
Global Snapshot

- Develop a simple global snapshot protocol
- Refine protocol as we relax assumptions
- Record:
  - processor states
  - channel states
- Assumptions:
  - FIFO channels
  - Each \( m \) timestamped with \( T(send(m)) \)
Snapshot I

i. $p_0$ selects $t_{ss}$

ii. $p_0$ sends “take a snapshot at $t_{ss}$” to all processes

iii. when clock of $p_i$ reads $t_{ss}$ then $p$
   a. records its local state $\sigma_i$
   b. sends an empty message along its outgoing channels
   c. starts recording messages received on each of incoming channels
   d. stops recording a channel when it receives first message with timestamp greater than or equal to $t_{ss}$
processor $p_0$ selects $\Omega$

$p_0$ sends “take a snapshot at $\Omega$” to all processes; it waits for all of them to reply and then sets its logical clock to $\Omega$

when clock of $p_i$ reads $\Omega$ then $p_i$

- records its local state $\sigma_i$
- sends an empty message along its outgoing channels
- starts recording messages received on each incoming channel
- stops recording a channel when receives first message with timestamp greater than or equal to $\Omega$
Relaxing synchrony

Process does nothing for the protocol during this time!

\[ TS(m) \geq \Omega \]

\[ \sigma_i \]

\[ TS(m) \geq \Omega \]
Snapshot III

- Processor $p_0$ sends itself "take a snapshot"

- When $p_i$ receives "take a snapshot" for the first time from $p_j$:
  - Records its local state $\sigma_i$
  - Sends "take a snapshot" along its outgoing channels
  - Sets channel from $p_j$ to empty
  - Starts recording messages received over each of its other incoming channels

- When $p_i$ receives "take a snapshot" beyond the first time from $p_k$:
  - Stops recording channel from $p_k$

- When $p_i$ has received "take a snapshot" on all channels, it sends collected state to $p_0$ and stops.
Same problem, different approach

Monitor process does not query explicitly

Instead, it passively collects information and uses it to build an observation.
(reactive architectures, Harel and Pnueli [1985])

An observation is an ordering of events of the distributed computation based on the order in which the receiver is notified of the events.
Update rules

Message $m$ is timestamped with $TS(m) = VC(send(m))$

$VC(e_i) := \max(VC, TS(m))$

$VC(e_i)[i] := VC[i] + 1$

$VC(e_i)[i] := VC[i] + 1$
Example

\[ p_1 \]
\[ p_2 \]
\[ p_3 \]
Operational interpretation

\[ VC(e_i)[i] = \text{no. of events executed by } p_i \text{ up to and including } e_i \]

\[ VC(e_i)[j] = \text{no. of events executed by } p_j \text{ that happen before } e_i \text{ of } p_i \]
VC properties: event ordering

Given two vectors $V$ and $V'$, less than is defined as:

$$V < V' \equiv (V \neq V') \land (\forall k: 1 \leq k \leq n : V[k] \leq V'[k])$$

- **Strong Clock Condition:** $e \rightarrow e' \equiv VC(e) < VC(e')$

- **Simple Strong Clock Condition:**
  
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$, where $i \neq j$
  
  $e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$

- **Concurrency**
  
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$, where $i \neq j$
  
  $e_i \| e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \land (VC(e_j)[j] > VC(e_i)[j])$
The protocol

$p_0$ maintains an array $D[1, \ldots, n]$ of counters

$D[i] = TS(m_i)[i]$ where $m_i$ is the last message delivered from $p_i$

Rule: Deliver $m$ from $p_j$ as soon as both of the following conditions are satisfied:

$$D[j] = TS(m)[j] - 1$$

$$D[k] \geq TS(m)[k], \forall k \neq j$$
Summary

Lamport clocks and vector clocks provide us with good tools to reason about timing of events in a distributed system.

Global snapshot algorithm provides us with an efficient mechanism for obtaining consistent global states.