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
- The local history of a process is the sequence of events executed by process
Ordering events

- **Observation 1:**
  - Events in a local history are **totally ordered**

- **Observation 2:**
  - For every message, send precedes receive
Lamport Clock: Increment Rules

\[ LC(e^i_p + 1) = LC(e^i_p) + 1 \]

\[ LC(e^j_q) = \max(LC(e^j_q^{-1}), LC(e^i_p)) + 1 \]

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

- What are the strengths of Lamport clocks?
- What are the limitations of Lamport clocks?
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 $\rightarrow$ impose a partial order
A cut $C$ is a subset of the global history of $H$

The frontier of $C$ is the set of events

$e_1^c, e_2^c, \ldots e_n^c$
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
Global Consistent States

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 $p$
    - records its local state $\sigma_i$
    - sends an empty message along its outgoing channels
    - starts recording messages received on each of incoming channels
    - stops recording a channel when it receives first message with timestamp greater than or equal to $t_{ss}$
Snapshot II

- 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!

$p_i$

Take a snapshot at $\Omega$

empty message:

$TS(m) \geq \Omega$

records local state $\sigma_i$

monitors channels

sends empty message:

$TS(m) \geq \Omega$
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

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

Message \( m \) is timestamped with
\[ TS(m) = VC(send(m)) \]

\[ VC(e_i) := \max(VC, TS(m)) \]

\[ VC(e_i)[i] := VC[i] + 1 \]
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 $i \neq j$
  $$e_i \parallel e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \land (VC(e_j)[j] > VC(e_i)[j])$$
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