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 event of the distributed computation based on the order in which the receiver is notified of the events.

An observation puts no constraint on the order in which the monitor receives notifications.
Observations: a few observations

An observation puts no constraint on the order in which the monitor receives notifications

To obtain a run, messages must be delivered to the monitor in FIFO order

Causal delivery

FIFO delivery guarantees:

send_i(m) → send_i(m′) ⇒ deliver_j(m) → deliver_j(m′)

Causal delivery generalizes FIFO:

send_i(m) → send_i(m′) ⇒ deliver_j(m) → deliver_j(m′)
Causal delivery

FIFO delivery guarantees:
\[ \text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m') \]

Causal delivery generalizes FIFO:
\[ \text{send}_i(m) \rightarrow \text{send}_k(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m') \]
Causal delivery

FIFO delivery guarantees:
\[
send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')
\]

Causal delivery generalizes FIFO:
\[
send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')
\]

Causal Delivery in Synchronous Systems

We use the upper bound \( \Delta \) on message delivery time

\[ DR1: \text{ At time } t, p_0 \text{ delivers all messages it received with timestamp up to } t - \Delta \text{ in increasing timestamp order}. \]
Causal Delivery with Lamport Clocks

**DR1.1:** Deliver all received messages in increasing (logical clock) timestamp order.

```

| p0 | 1 | 4 |

Should p0 deliver?
```

Problem: Lamport Clocks don’t provide gap detection

Given two events \( e \) and \( e' \) and their clock values \( LC(e) \) and \( LC(e') \) — where \( LC(e) < LC(e') \)

determine whether some event \( e'' \) exists s.t.

\[ LC(e) < LC(e'') < LC(e') \]

Causal Delivery with Lamport Clocks

**DR1.1:** Deliver all received messages in increasing (logical clock) timestamp order.

```

| p0 | 1 | 4 |

Should p0 deliver?
```

Stability

**DR2:** Deliver all received stable messages in increasing (logical clock) timestamp order.

A message \( m \) received by \( p \) is stable at \( p \) if \( p \)
will never receive a future message \( m' \) s.t.

\[ TS(m') < TS(m) \]
Implementing Stability

Real-time clocks
- wait for $\Delta$ time units

Lamport clocks
- wait on each channel for $m$ s.t. $TS(m) > LC(e)$
- Design better clocks!

Clocks and STRONG Clocks

Lamport clocks implement the clock condition:
$$e \rightarrow e' \Rightarrow LC(e) < LC(e')$$

We want new clocks that implement the strong clock condition:
$$e \rightarrow e' \equiv SC(e) < SC(e')$$

Causal Histories

The causal history of an event $e$ in $(H, \rightarrow)$ is the set
$$\theta(e) = \{ e' \in H | e' \rightarrow e \} \cup \{ e \}$$
Causal Histories

The causal history of an event \( e \) in \( (H, \rightarrow) \) is the set
\[
\theta(e) = \{ e' \in H \mid e' \rightarrow e \} \cup \{ e \}
\]

Causal Histories

The causal history of an event \( e \) in \( (H, \rightarrow) \) is the set
\[
\theta(e) = \{ e' \in H \mid e' \rightarrow e \} \cup \{ e \}
\]

Pruning causal histories

Prune segments of history that are known to all processes (Peterson, Bucholz and Schlichting)

Use a more clever way to encode \( \theta(e) \)

How to build \( \theta(e) \)

Each process \( p_i \):
- initializes \( \theta : \theta := \emptyset \)
- if \( e_i^k \) is an internal or send event, then
  \[
  \theta(e_i^k) := \{ e_i^k \} \cup \theta(e_i^{k-1})
  \]
- if \( e_i^k \) is a receive event for message \( m \), then
  \[
  \theta(e_i^k) := \{ e_i^k \} \cup \theta(e_i^{k-1}) \cup \theta(\text{send}(m))
  \]
Vector Clocks

Consider $\theta_i(e)$, the projection of $\theta(e)$ on $p_i$

$\theta_i(e)$ is a prefix of $h^i$: $\theta_i(e) = h^i_{k_i}$ – it can be encoded using $k_i$

$\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \ldots \cup \theta_n(e)$ can be encoded using $k_1, k_2, \ldots, k_n$

Represent $\theta$ using an $n$-vector $VC$ such that $VC(e)[i] = k \iff \theta_i(e) = h^i_{k_i}$

Update rules

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

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

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

Example

Operational interpretation
Operational interpretation

Operational interpretation = no. of events executed by $p_i$ up to and including $e_i$

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

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

VC properties: event ordering

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

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

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

VC properties: consistency

Pairwise inconsistency

Events $e_i$ of $p_i$ and $e_j$ of $p_j$ ($i \neq j$) are pairwise inconsistent (i.e. can't be on the frontier of the same consistent cut) if and only if

$$(VC(e_i)[i] < VC(e_j)[i]) \lor (VC(e_j)[j] < VC(e_i)[j])$$

Consistent Cut

A cut defined by $(c_1, \ldots, c_n)$ is consistent if and only if

$$\forall i,j : 1 \leq i \leq n, 1 \leq j \leq n : (VC(c_i^e)[i] \geq VC(c_j^e)[i])$$
**VC properties:**

*Weak gap detection*

Given $e_i$ of $p_i$ and $e_j$ of $p_j$, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists $e_k$ s.t.

$$\neg(e_k \rightarrow e_i) \land (e_k \rightarrow e_j)$$

**VCs for Causal Delivery**

Each process increments the local component of its $VC$ only for events that are notified to the monitor.

Each message notifying event $e$ is timestamped with $VC(e)$.

The monitor keeps all notification messages in a set $M$. 

---

**VC properties:**

*Weak gap detection*

Given $e_i$ of $p_i$ and $e_j$ of $p_j$, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists $e_k$ s.t.

$$\neg(e_k \rightarrow e_i) \land (e_k \rightarrow e_j)$$

**Strong gap detection**

Given $e_i$ of $p_i$ and $e_j$ of $p_j$, if $VC(e_i)[i] < VC(e_j)[i]$ then there exists $e'_i$ s.t.

$$(e_i \rightarrow e'_i) \land (e'_i \rightarrow e_j)$$
Stability

Suppose $p_0$ has received $m_j$ from $p_j$. When is it safe for $p_0$ to deliver $m_j$?

- There is no earlier message in $M$
  $\forall m \in M : \neg (m \rightarrow m_j)$
- There is no earlier message from $p_j$
  $TS(m_j)[j] = 1 + \text{no. of } p_j \text{ messages delivered by } p_0$
- There is no earlier message from $p_i$
  $TS(m_j)[j] = 1 + \text{no. of } p_j \text{ messages delivered by } p_0$
- There is no earlier message $m''_{k}$ from $p_k, k \neq j$
  see next slide...
Checking for $m_k''$

- Let $m_k'$ be the last message $p_0$ delivered from $p_k$
- By strong gap detection, $m_k''$ exists only if $TS(m_k')[k] < TS(m_j)[k]$
- Hence, deliver $m_j$ as soon as
  $\forall k : TS(m_k')[k] \geq TS(m_j)[k]$

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 $p_i$
- DR3: Deliver $m$ from $p_j$ as soon as both of the following conditions are satisfied:
  1. $D[j] = TS(m)[j] - 1$
  2. $D[k] \geq TS(m)[k], \forall k \neq j$

The challenges of non-stable predicates

- Consider a non-stable predicate $\Phi$ encoding, say, a safety property. We want to determine whether $\Phi$ holds for our program.

The challenges of non-stable predicates

- Consider a non-stable predicate $\Phi$ encoding, say, a safety property. We want to determine whether $\Phi$ holds for our program.
- Suppose we apply $\Phi$ to $\Sigma^\omega$
The challenges of non-stable predicates

Consider a non-stable predicate $\Phi$ encoding, say, a safety property. We want to determine whether $\Phi$ holds for our program.

Suppose we apply $\Phi$ to $\Sigma^s$

$\Phi$ holding in $\Sigma^s$ does not preclude the possibility that our program violates safety!

The challenges of non-stable predicates

Consider now a different non-stable predicate $\Phi$. We want to determine whether $\Phi$ ever holds during a particular computation.

Suppose we apply $\Phi$ to $\Sigma^s$

$\Phi$ holding in $\Sigma^s$ does not imply that $\Phi$ ever held during the actual computation!

Example

Detect whether the following predicates hold:

- $x = y$
- $x = y - 2$

Assume that initially:

- $x = 0; y = 10$
Possibly

If $\Sigma^a$ is $\Sigma^{31}$ or $\Sigma^{41}$, $x = y - 2$ is detected, but it may never have occurred

Definitely

We know that $x = y$ has occurred, but it may not be detected if tested before $\Sigma^{32}$ or after $\Sigma^{54}$

Possibly

If $\Sigma^a$ is $\Sigma^{31}$ or $\Sigma^{41}$, $x = y - 2$ is detected, but it may never have occurred

Possibly(Φ)
There exists a consistent observation of the computation $O$ such that $\Phi$ holds in a global state of $O$

Definitely

We know that $x = y$ has occurred, but it may not be detected if tested before $\Sigma^{32}$ or after $\Sigma^{54}$

Definitely(Φ)
For every consistent observation $O$ of the computation, there exists a global state of $O$ in which $\Phi$ holds
Computing Possibly Scan lattice, level after level

If $\Phi$ holds in one global state, then Possibly($\psi$)
Computing Possibly

Scan lattice, level after level

If $\Phi$ holds in one global state, then $\text{Possibly}(\psi)$

Possibly($x = y - 2$)

Computing Definitely

Scan lattice, level after level

Given a level, only expand nodes that correspond to states for which $\neg \Phi$

Computing Definitely

Scan lattice, level after level

Given a level, only expand nodes that correspond to states for which $\neg \Phi$

If no such state, then $\text{Definitely}(\psi)$

If reached last state $\Sigma'$ and $\Phi(\Sigma')$, then $\neg \text{Definitely}(\psi)$
Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which \( \neg \Phi \)
- If no such state, then \( \text{Definitely}(\Phi) \)
- If reached last state \( \Sigma_i \), and \( \Phi(\Sigma_i) \), then \( \neg \text{Definitely}(\Phi) \)

Building the lattice: collecting local states

- To build the global states in the lattice, \( p_0 \) collects local states from each process.
- \( p_0 \) keeps the set of local states received from \( p_i \) in a FIFO queue \( Q_i \)

Key questions:
1. when is it safe for \( p_0 \) to discard a local state \( \sigma_i^k \) of \( p_i \)?
2. Given level \( i \) of the lattice, how does one build level \( i + 1 \)?

Garbage-collecting local states

- For each local state \( \sigma_i^k \), we need to determine:
  - \( \Sigma_{\min}(\sigma_i^k) \), the \textit{earliest} consistent state that \( \sigma_i^k \) can belong to
  - \( \Sigma_{\max}(\sigma_i^k) \), the \textit{latest} consistent state that \( \sigma_i^k \) can belong to

Defining “earliest” and “latest”

Consistent Global State
Defining “earliest” and “latest”

Consistent Global State

Consistent Cut

Defining “earliest” and “latest”

Consistent Global State

Consistent Cut

Frontier

Defining “earliest” and “latest”

Consistent Global State

Consistent Cut

Frontier

Vector Clock

Defining “earliest” and “latest”

Consistent Global State

Consistent Cut

Frontier

Vector Clock

Associate a vector clock with each consistent global state

\( \Sigma_{\text{min}}(\sigma_k) \) is the consistent global state with the lowest vector clock that has \( \sigma_k \) on its frontier

\( \Sigma_{\text{max}}(\sigma_k) \) is the one with the highest
Computing $\Sigma_{\min}$

- Label $\sigma^k_i$ with $VC(\epsilon^k_i)$
- $\Sigma_{\min}(\sigma^k_i) = (\sigma^1_i, \sigma^2_i, \ldots, \sigma^n_i) : \forall j : \epsilon_j = VC(\sigma^k_i)[j]

- $\Sigma_{\min}(\sigma^k_i)$ and $\sigma^k_i$ have the same vector clock!

Computing $\Sigma_{\max}$

- $\Sigma_{\max}(\sigma^k_i) = (\sigma^1_i, \sigma^2_i, \ldots, \sigma^n_i) :$
  - $\forall j : VC(\sigma^j_i)[i] \leq VC(\sigma^k_i)[i]$
  - $((\sigma^j_i = \sigma^j_i) \lor VC(\sigma^{j+1}_i)[i] > VC(\sigma^k_i)[i])$

Set of local states one for each process, s.t. all local states are pairwise consistent with $\sigma^k_i$
Computing $\Sigma_{max}$

$\Sigma_{max}(\sigma_k^i) = (\sigma_1^1, \sigma_2^2, ..., \sigma_n^n)$:

$\forall j : VC(\sigma_j^j)[i] \leq VC(\sigma_k^i)[i]$ and they are the last such state.

Assembling the levels

- To build level $l$:
  - wait until each $Q_i$ contains a local state for whose vector clock:
    $$\sum_{i=1}^{n} VC[i] \geq l$$

- To build level $l + 1$:
  - For each global state $\sum$ on level $l$, build
    $$\sum i_1, i_2, ..., i_n$$

- Using VC's, check whether these global states are consistent