An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice

\[ \Sigma_0^{00} \quad \Sigma_1^{01} \quad \Sigma_0^{12} \quad \Sigma_1^{11} \quad \Sigma_0^{22} \quad \Sigma_1^{21} \]

\[ e_1^1 \quad e_2^1 \quad e_3^1 \quad e_4^1 \quad e_5^1 \quad e_6^1 \]

\[ \Sigma_0^{00} \quad \Sigma_1^{01} \quad \Sigma_0^{12} \quad \Sigma_1^{11} \quad \Sigma_0^{22} \quad \Sigma_1^{21} \]

\[ e_1^1 \quad e_2^1 \quad e_3^1 \quad e_4^1 \quad e_5^1 \quad e_6^1 \]

\[ p_1 \quad p_2 \]

An Execution and its Lattice
An Execution and its Lattice

Reachability

$\Sigma^k_l$ is reachable from $\Sigma^{ij}$ if there is a path from $\Sigma^k_l$ to $\Sigma^{ij}$ in the lattice
Reachability

\[ \Sigma^k_l \text{ is reachable from } \Sigma^{ij} \text{ if there is a path from } \Sigma^k_l \text{ to } \Sigma^{ij} \]

in the lattice.

Σ^k_l \Rightarrow □
Σ^i Σ^f R Σ^i \Rightarrow R Σ^f
Σ^f Σ^s

Σ^k_l \text{ is reachable from } \Sigma^{ij} \text{ if there is a path from } \Sigma^{ij} \text{ to } \Sigma^k_l

in the lattice.

Σ^{ij} \sim Σ^k_l

So, why do we care about Σ^s again?

Deadlock is a stable property

Deadlock \Rightarrow □ Deadlock

If a run \( R \) of the snapshot protocol starts in Σ^i and terminates in Σ^f, then Σ^i \sim_R Σ^f

Deadlock in Σ^s implies deadlock in Σ^f

Deadlock is a stable property

Deadlock \Rightarrow □ Deadlock

If a run \( R \) of the snapshot protocol starts in Σ^i and terminates in Σ^f, then Σ^i \sim_R Σ^f

Deadlock in Σ^s implies deadlock in Σ^f
So, why do we care about $\Sigma^s$ again?

- Deadlock is a **stable property**
  
  Deadlock $\Rightarrow \square$ Deadlock

- If a run $R$ of the snapshot protocol starts in $\Sigma^i$ and terminates in $\Sigma^f$, then $\Sigma^i \sim_R \Sigma^f$

- Deadlock in $\Sigma^s$ implies deadlock in $\Sigma^f$

- No deadlock in $\Sigma^s$ implies no deadlock in $\Sigma^f$

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])

A **observation** is an ordering of event of the distributed computation based on the order in which the receiver is notified of the events.

Observations: a few observations

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

- $p_0 \rightarrow c_1 \rightarrow p_1$

Observations: a few observations

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

- $p_0 \rightarrow c_1 \rightarrow c_2 \rightarrow p_1$
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.

What about consistent runs?

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

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

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 in Synchronous Systems

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

\[ \Delta \text{t}_{p0} \]

DRI: At time \( t, p_0 \) delivers all messages it received with timestamp up to \( t - \Delta \) in increasing timestamp order
Causal Delivery with Lamport Clocks

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

---

### 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 such that:

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

\[ \theta(e) = \{ e' \in H \mid e' \rightarrow e \} \cup \{ e \} \]

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

Each process \( p_i \):

- Initializes \( \theta : \theta := \emptyset \)
- If \( e^k_i \) is an internal or send event, then
  \[ \theta(e^k_i) := \{ e^k_i \} \cup \theta(e^{k-1}_i) \]
- If \( e^k_i \) is a receive event for message \( m \), then
  \[ \theta(e^k_i) := \{ e^k_i \} \cup \theta(e^{k-1}_i) \cup \theta(\text{send}(m)) \]
Pruning causal histories

- Prune segments of history that are known to all processes (Peterson, Bucholz and Schlichting)
- Use a more clever way to encode $\theta(e)$

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

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

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

Example

\[ \begin{align*}
  p_1 & \quad [1,0,0] \quad [2,1,0] \quad [3,1,2] \quad [4,1,2] \quad [5,1,2] \\
  p_2 & \quad [0,1,0] \quad [1,2,3] \quad [4,3,3] \\
  p_3 & \quad [1,0,1] \quad [1,0,2] \quad [1,0,3] \quad [5,1,4]
\end{align*} \]
Operational interpretation

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

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

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**: $e \rightarrow e' \equiv VC(e) \leq VC(e')$

- **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
  \[(\text{VC}(e_i)[i] < \text{VC}(e_j)[i]) \vee (\text{VC}(e_j)[j] < \text{VC}(e_i)[j])\]

- Consistent Cut
  A cut defined by $(c_1, \ldots, c_n)$ is consistent if and only if
  \[\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : (\text{VC}(c_i^e)[i] \geq \text{VC}(c_j^e)[i])\]

VC properties: weak gap detection

- Weak gap detection
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$ if $\text{VC}(e_i)[k] < \text{VC}(e_j)[k]$ for some $k \neq j$, then there exists $e_k$ s.t
  \[\neg (e_k \rightarrow e_i) \land (e_k \rightarrow e_j)\]

VC properties: strong gap detection

- Weak gap detection
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$, if $\text{VC}(e_i)[k] < \text{VC}(e_j)[k]$ 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 $\text{VC}(e_i)[i] < \text{VC}(e_j)[i]$ then there exists $e'_i$ s.t.
  \[(e_i \rightarrow e'_i) \land (e'_i \rightarrow e_j)\]
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 $M$.

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 $M$:

$\forall m \in M : \neg(m \rightarrow m_j)$

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 $p_j$:

$TS(m_j)[j] = 1 + \text{no. of } p_j \text{ messages delivered by } p_0$
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 \( m''_k \) 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 $\Sigma^*$

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

$\Phi$ holding in $\Sigma^*$ 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 $\Sigma^*$

$\Phi$ holding in $\Sigma^*$ does not imply that $\Phi$ ever held during the actual computation!
Detect whether the following predicates hold:

Assume that initially:

\[ x = y \]
\[ x = y - 2 \]
\[ x = 0; y = 10 \]

\( \Sigma \) is \( \Sigma^{31} \) or \( \Sigma^{41} \),
\( x = y - 2 \) is detected,
but it may never have occurred.

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

We know that \( x = y \) has occurred, but it may not be detected
if tested before \( \Sigma^{32} \) or after \( \Sigma^{34} \).
Definitely

We know that $x = y$ has occurred, but it may not be detected if tested before $\Sigma^{52}$ or after $\Sigma^{54}$.

Definitely($\Phi$)

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($\Phi$).
Computing Possibly

- Scan lattice, level after level
- If $\Phi$ holds in one global state, then Possibly(\(\Phi\))

Possibly(\(x = y - 2\))

Computing Definitely

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

possible, level after level
If $\Phi$ holds in one global state, then Possibly(\(\Phi\))
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 Definitely($\Phi$)
- If reached last state $\Sigma_i$ and $\Phi(\Sigma_i)$, then $\neg$ 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 $i + 1$?

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 Definitely($\Phi$)
- If reached last state $\Sigma_i$ and $\Phi(\Sigma_i)$, then $\neg$ Definitely($\Phi$)

Definitely ($x = y$)

Garbage-collecting local states

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

Associate a vector clock with each consistent global state

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

\( \Sigma_{\text{max}}(\sigma_k^i) \) is the one with the highest

Computing \( \Sigma_{\text{min}} \)

\( \Sigma_{\text{min}}(\sigma_k^i) = (\sigma_1^{\sigma_k^i}, \sigma_2^{\sigma_k^i}, \ldots, \sigma_n^{\sigma_k^i}) : \forall j : VC(\sigma_j)[i] \leq VC(\sigma_k^i)[i] \land ((\sigma_j^i = \sigma_k^i) \lor VC(\sigma_j^{i+1})[i] > VC(\sigma_k^i)[i]) \)

Computing \( \Sigma_{\text{max}} \)

\( \Sigma_{\text{max}}(\sigma_k^i) = (\sigma_1^{\sigma_k^i}, \sigma_2^{\sigma_k^i}, \ldots, \sigma_n^{\sigma_k^i}) : \forall j : VC(\sigma_j^i)[i] \leq VC(\sigma_k^i)[i] \land ((\sigma_j^i = \sigma_k^i) \lor VC(\sigma_j^{i+1})[i] > VC(\sigma_k^i)[i]) \)

set of local states one for each process, s.t.
Computing $\sum_{max}$

$\Sigma_{max}(\sigma^i) = (\sigma^{i_1}, \sigma^{i_2}, \ldots, \sigma^{i_n})$:

$\forall j : VC(\sigma^{i_j}[i] \leq VC(\sigma^i[i])$

$\wedge ((\sigma^{i_j} = \sigma^{i'_j}) \vee VC(\sigma^{i_j+1}[i] > VC(\sigma^i[i])))$

\text{set of local states one for each process, s.t. all local states are pair-wise consistent with $\sigma^{i}$}

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 $\sum_{i_1,i_2,...,i_m}$ on level $l$, build
    $\sum_{i_1+i_2+...+i_m = 0}^{\infty}, \sum_{i_1=0}^{\infty} \sum_{i_2=0}^{\infty} \ldots \sum_{i_m=0}^{\infty}$

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