**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 \( t_{ss} \)

a. records its local state \( \sigma_i \)

b. starts recording messages received on each of incoming channels

c. stops recording a channel when it receives first message with timestamp greater than or equal to \( t_{ss} \)

**Correctness**

**Theorem**  
Snapshot I produces a consistent cut

**Proof**  
Need to prove \( e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C \)

1. \( e_j \in C \)

2. \( e_i \rightarrow e_j \)

\(< \text{Assumption}>\)

3. \( T(e_j) < t_{ss} \)

\(<\text{Property of real time}>\)

4. \( e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j) \)

\(<\text{Definition}>\)

5. \( T(e_i) < T(e_j) \)

\(<\text{Property of real time}>\)

6. \( T(e_i) < t_{ss} \)

\(<\text{Assumption}>\)

7. \( e_{i} \in C \)

\(<\text{Definition}>\)

**Clock Condition**

Can the Clock Condition be implemented some other way?
Lamport Clocks

Each process maintains a local variable $LC$

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

Increment Rules

$p \quad e_p^i \quad e_p^{i+1} \quad p$

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

$p \quad e_p^i \quad q \quad e_q^j$

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

$p \quad e_p^i \quad e_p^{i+1} \quad q$

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

$q \quad e_q^j$

$LC(e_q^j) = \max(LC(e_q^{j-1}), LC(e_q^{j})) + 1$

Timestamp $m$ with $TS(m) = LC(send(m))$

Space-Time Diagrams and Logical Clocks

A subtle problem

when $LC = t$ do $S$

doesn't make sense for Lamport clocks!

- there is no guarantee that $LC$ will ever be $t$
- $S$ is anyway executed after $LC = t$

Fixes:

- if $e$ is internal/send and $LC = t-2$
  - execute $e$ and then $S$
- if $e = \text{receive}(m) \wedge (TS(m) \geq t) \wedge (LC \leq t-1)$
  - put message back in channel
  - re-enable $e$; set $LC = t-1$; execute $S$
An obvious problem

No $t_{ss}$!

Choose $\Omega$ large enough that it cannot be reached by applying the update rules of logical clocks

mmmhhhhhh...

An obvious problem

No $t_{ss}$!

Choose $\Omega$ large enough that it cannot be reached by applying the update rules of logical clocks

mmmhhhhhh...

An obvious problem

No $t_{ss}$!

Choose $\Omega$ large enough that it cannot be reached by applying the update rules of logical clocks

mmmhhhhhh...

An obvious problem

No $t_{ss}$!

Choose $\Omega$ large enough that it cannot be reached by applying the update rules of logical clocks

mmmhhhhhh...

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 $\Omega$

Doing so assumes

□ upper bound on message delivery time

□ upper bound relative process speeds

We better relax it...
Relaxing synchrony

Process does nothing for the protocol during this time!

Use empty message to announce snapshot!

Snapshots: a perspective

The global state $\Sigma^*$ saved by the snapshot protocol is a consistent global state

But did it ever occur during the computation?

- A distributed computation provides only a partial order of events
- Many total orders (runs) are compatible with that partial order
- All we know is that $\Sigma^*$ could have occurred

Snapshots 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.
Snapshots: a perspective

- The global state $\Sigma^*$ saved by the snapshot protocol is a consistent global state.
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events
  - many total orders (runs) are compatible with that partial order
  - all we know is that $\Sigma^*$ could have occurred
- We are evaluating predicates on states that may have never occurred!

An Execution and its Lattice

An Execution and its Lattice

An Execution and its Lattice
An Execution and its Lattice

\[ \Sigma^{00} \rightarrow \Sigma^{01} \]

\[ e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6 \]

\[ p_1, p_2 \]

\[ \Sigma^{10} \rightarrow \Sigma^{01} \]

\[ e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_2^6 \]

\[ \Sigma^{00} \rightarrow \Sigma^{11} \]

\[ e_3^1, e_3^2, e_3^3, e_3^4, e_3^5, e_3^6 \]

\[ \Sigma^{10} \rightarrow \Sigma^{11} \]

\[ e_4^1, e_4^2, e_4^3, e_4^4, e_4^5, e_4^6 \]

\[ \Sigma^{02} \]

\[ e_5^1, e_5^2, e_5^3, e_5^4, e_5^5, e_5^6 \]

\[ \Sigma^{12} \]

\[ e_6^1, e_6^2, e_6^3, e_6^4, e_6^5, e_6^6 \]

\[ \Sigma^{10} \rightarrow \Sigma^{12} \]

\[ \Sigma^{01} \rightarrow \Sigma^{12} \]
An Execution and its Lattice
An Execution and its Lattice

Reachability

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

Reachability is reachable from Σ^i_j if there is a path from Σ^i_j to Σ^k_l in the lattice.

So, why do we care about Σ^s again?

- Deadlock is a stable property
  - Deadlock ⇒ □ Deadlock
- If a run R of the snapshot protocol starts in Σ^i and terminates in Σ^f, then Σ^i ~ R Σ^f
So, why do we care about $\Sigma^s$ again?

- Deadlock is a stable property
  
  Deadlock $\Rightarrow$ $\Box$ 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 $\Sigma^i$

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.

Observations: a few observations

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

  \[ p_0 \quad p_1 \quad e_1 \]

  An observation $p_0$ precedes $p_1$ and $e_1$ precedes $p_1$.
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 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
Causal Delivery in Synchronous Systems

We use the upper bound $\Delta$ on message delivery time

**DR1:** 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.

Wouldn't one see LC 2 and 3? If so, I should know when I can deliver...

Or is it that there my be multiple 4s?
Causal Delivery with Lamport Clocks

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

![Diagram showing events p0, p1, and p4 and question mark indicating whether p0 should 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.

$LC(e) < LC(e'') < LC(e')$

**Implementing Stability**

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

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 $e_i^k$ 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 $\theta(e_i^k) := \{e_i^k\} \cup \theta(e_i^{k-1}) \cup \theta(send(m))$

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 $h'$: $\theta_i(e) = h_i^k$ - 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 $VC(e)[i] = k \iff \theta_i(e) = h_i^k$

Update rules

- $VC(e_i)[i] := VC[i] + 1$
- Message $m$ is timestamped with $TS(m) = VC(send(m))$
- $VC(e_i) := \max(VC, TS(m))$
- $VC(e_i)[i] := VC[i] + 1$
Example

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:
\[ e \to e' \equiv VC(e) < VC(e') \]

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

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: weak gap detection

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
\[ \neg(e_k \to e_i) \land (e_k \to e_j) \]

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: strong gap detection

- **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.
  \[-(e_k \rightarrow e_i) \land (e_k \rightarrow e_j)\]

- **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.
  \[(e_i \rightarrow e'_i) \land (e'_i \rightarrow e_j)\]

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 \( m_j \)?

- **Stability**
  Suppose \( p_0 \) has received \( m_j \) from \( p_j \). When is it safe for \( p_0 \) to deliver \( m_j \)?
  \[ \forall m \in M : \neg (m \rightarrow m_j) \]
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 $p_j$
  \[ TS(m_j)[j] = 1 + \text{no. of } p_j \text{ messages delivered by } p_0 \]

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

Property: a predicate that is evaluated over a run of the program (a trace)

“every message that is received was previously sent”

Not everything you may want to say about a program is a property:

“the program sends an average of 50 messages in a run”

Safety properties

“nothing bad happens”

- no more than k processes are simultaneously in the critical section
- messages that are delivered are delivered in causal order
- Windows never crashes

A safety property is “prefix closed”:

- if it holds in a run, it holds in every prefix

Liveness properties

“something good eventually happens”

- a process that wishes to enter the critical section eventually does so
- some message is eventually delivered
- Windows eventually boots

Every run can be extended to satisfy a liveness property

- if it does not hold in a prefix of a run, it does not mean it may not hold eventually

A really cool theorem

Every property is a combination of a safety property and a liveness property

(Alpern & Schneider)
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!

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 \( \Sigma^s \).

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 \( \Sigma^s \).
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:

Assume that initially:

$x = 3$
$x = 4$
$x = 5$
$y = 6$
$y = 4$
$y = 2$
$e_1$
$e_2$
$e_3$
$e_4$
$e_5$

Possibly

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

Possibly

If $\Sigma^s$ 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 $\mathcal{O}$ such that $\Phi$ holds in a global state of $\mathcal{O}$.
Definitely

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

Definitely($\Phi$)

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($\Phi$)
Computing Possibly

Scan lattice, level after level

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

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

Computing Possibly

Scan lattice, level after level

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

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

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

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

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

1. Scan lattice, level after level
2. Given a level, only expand nodes that correspond to states for which $\neg \Phi$
3. If no such state, then Definitely$(\Phi)$
4. If reached last state $\Sigma_i'$ and $\neg \Phi(\Sigma_i')$, then $\neg$Definitely$(\Phi)$

Building the lattice: collecting local states

1. To build the global states in the lattice, $p_0$ collects local states from each process.
2. $p_0$ 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 $\sigma^k_i$, we need to determine:

- $\Sigma_{\text{min}}(\sigma^k_i)$, the earliest consistent state that $\sigma^k_i$ can belong to
- $\Sigma_{\text{max}}(\sigma^k_i)$, the latest consistent state that $\sigma^k_i$ can belong to

Defining “earliest” and “latest” Consistent Global State

Defining “earliest” and “latest” Consistent Cut

Defining “earliest” and “latest” Consistent Global State

Defining “earliest” and “latest” Frontier
Defining “earliest” and “latest” Consistent Global State

Consistent Cut

Frontier

Vector Clock

Consistent Global State

Consistent Cut

Frontier

Vector Clock

Label \( \sigma_i^k \) with \( VC(e_i^k) \)

\[ \Sigma_{\min}(\sigma_i^k) = (\sigma_1^{k_1}, \sigma_2^{k_2}, \ldots, \sigma_n^{k_n}) : \forall j : c_j = VC(\sigma_i^k)[j] \]

\( \Sigma_{\min}(\sigma_i^k) \) and \( \sigma_i^k \) have the same vector clock!

\[ \Sigma_{\max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n}) : \\
\land \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i] \land (\sigma_j^{c_j} = \sigma_j^{c_j+1}) \lor VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]) \]
Computing $\Sigma_{max}$

$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n})$

$\land \forall j : VC(\sigma_j^{c_j})(i) \leq VC(\sigma_i^k)(i)$

$\land ((\sigma_j^{c_j} = \sigma_j^{c_i}) \lor VC(\sigma_j^{c_j+1})(i) > VC(\sigma_i^k)(i))$

set of local states
one for each process, s.t.
all local states are pair-wise consistent with $\sigma_i^k$

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 $\Sigma$ on level $l$, build
  $\sum_{i_1=1}^n \sum_{i_2=1}^{i_1} \ldots \sum_{i_n=1}^{i_{n-1}} \ldots \sum_{i_{n+1}=1}^{i_n}$

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