The Idea

- Use empty message as the announcement for taking a snapshot!

Properties of Snapshots

- The global state 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 one could have occurred.
- We are evaluating predicates on states that may have never occurred!

Snapshot III

0: processor $p_0$ sends itself "take a snapshot"

1: when $p_i$ receives "take a snapshot" for the first time from $p_j$:
   - records its local state
   - sends "take a snapshot" along its outgoing channels
   - sets channel from $p_i$ to empty
   - starts recording messages received over each of its other incoming channels

2: when $p_i$ receives "take a snapshot" beyond the first time from $p_k$:
   - stops recording channel from $p_k$

3: when $p_i$ has received "take a snapshot" on all channels, it sends collected state to $p_0$ and stops.

Guess who? (with M. Chandy)

Properties

- The global state 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 one could have occurred.
- We are evaluating predicates on states that may have never occurred!

An Execution and its Lattice
Reachability

We say that $S^{ij}$ is reachable from $S^{kl}$ if there is a path from $S^{ij}$ to $S^{kl}$ in the lattice.

So, why do we care about $S^s$ again?

- Deadlock is a stable property

- If $[T]$ initial state and $[T']$ termination state for snapshot:

  $[T] \sim [T'] \sim [T'']$

  for a run $R$

Consequences

- Deadlock in $[T]$ implies Deadlock in $[T']$

- No Deadlock in $[T]$ implies no Deadlock in $[T']$

Same problem, a different approach

- Monitor process does not query explicitly

- It just passively collects information

- and uses it to build an observation.

  (reactive architectures, Harel and Pnueli [1985])

Definition An observation is an ordering of event of the distributed computation based on the order in which the receiver is notified of the events.
Building runs

- Messages must be delivered to $p_0$ in FIFO order

$$\text{send}(m) \quad \text{send}(m) \quad \text{deliver}(m) \quad \text{deliver}(m)$$

- Is FIFO delivery sufficient to build consistent runs?

  NO. We need a stronger delivery rule…

Causal Delivery

Implementing Causal Delivery in Synchronous Systems

If upper bound $d$ on message delivery time:
- Notification message for event $e$ carries $TS(e)$

$$\text{send}(m) \quad \text{send}(m) \quad \text{deliver}(m) \quad \text{deliver}(m)$$

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

Implementing Causal Delivery with Lamport Clocks

**DR1:** At time $t$, $p_0$ delivers all received messages with timestamps up to $t$ in increasing timestamp order.
(Il)logical clocks?

- If no bound on message delay
- And no real time clock to measure it
- No way to decide when to deliver a message?

Gap-Detection

Given two events $e$ and $\hat{e}$ and their clock values $LC(e)$ and $LC(\hat{e})$ where $LC(e) < LC(\hat{e})$, determine whether some other event $e'$ exists such that $LC(e) < LC(e') < LC(\hat{e})$.

Stability

Definition A message $m$ received by $p$ is stable at $p$ if $p$ will never receive a future message $m$ such that $TS(m') < TS(m)$.

implementing Stability

Real-time Clocks
- Wait for $d$ time units

$RC(e) < RC(\hat{e}) \land e' \in \hat{e}$

Lamport Clocks
- Wait on every channel for message $m$ with $TS(m) < LC(e)$

$LC(e) < LC(e') \land e' \in \hat{e}$

Our approach
- Design better clocks!
Clocks and **Strong** Clocks

*RC and LC implement the Clock Condition*

\[ e [\cdot] e \quad RC(e) < RC(e) \]
\[ e [\cdot] e \quad LC(e) < LC(e) \]

We want new clocks *TC* that implement the following

*Strong Clock Condition*

\[ e [\cdot] e \quad TC(e) < TC(e) \]

---

**Causal Histories**

**Definition:** The *causal history* of an event *e* in \((H, \mathcal{R})\) is the set \(C(e) = \{e\} \cup \mathcal{R}(e)\)

---

**How to build \(C(e)\)**

Each process \(p_i\):
- Initializes a local variable \(I\) to the empty set
- If *e* is an *internal* or a *send* event, then
  \[
  I(e) \cup \{e\} \cup \{\text{previous event in local history}\}
  \]
- If *e* is a *receive* event, then
  \[
  I(e) \cup \{e\} \cup \{\text{previous event in local history}\}
  \]
  \[
  \sqcup I(\text{send event})
  \]

---

**Pruning Causal Histories**

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

- Projection $\mathcal{P}(e)$ of $\mathcal{D}(e)$ for process $p_i$ is a prefix of $p_i$'s local history: $\mathcal{D}(e) = h^e$ and can be encoded using $k_i$
- $\mathcal{D}(e) = \mathcal{D}(e) \bigg\uparrow \mathcal{D}(e) \bigg\uparrow \ldots \bigg\uparrow \mathcal{D}(e)$ can be encoded using $k_i, k_2, \ldots, k_n$

Represent $\mathcal{D}$ using an $n$-vector $VC$ such that $VC(e)[i] = k_i$ $\mathcal{D}(e) = h^e$.

Update Rules

- $p_i$ $e_i$
- $VC[e_i][i] = VC[i] + 1$
- $p_i$ $e_i$
- $VC[e_i] = \max(VC, TS(m))$
- $VC[e_i][i] = VC[i] + 1$

Example

- $p_1$
- $[1,0,0] [2,1,0] [3,1,2] [4,1,2] [5,1,2]$
- $p_2$
- $[0,1,0] [1,2,3] [4,3,3]$
- $p_3$
- $[0,1,0] [1,0,2] [1,0,3] [3,1,2] [2,2,3] [4,1,2] [5,1,2] [4,3,3] [5,1,4]$

Operational Interpretation

- $VC[e_i][j]$ = number of events $p_i$ has executed up to and including $e_i$
- $VC[e_i][j]$ = number of events of $p_j$ that causally precede event $e_i$ of $p_i$
Properties of Vector Clocks: Event Ordering

Definition: Given two vectors \( V \) and \( V' \) the relation \( V < V' \) is defined as:
\[
V < V' \iff \exists k \in \mathbb{N}, V_k < V'_k, V_k = V'_k \land \sum_{i=1}^{k-1} (V_i - V'_i) = 0
\]

Property 1 (Strong Clock Condition)
Given event \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \), where \( i \neq j \)
\[
e_i = PC(e_i[i]) > PC(e_j[j])
\]

Property 2 (Simple Strong Clock Condition)
Given event \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \), where \( i \neq j \)
\[
e_i = PC(e_i[i]) > PC(e_j[j])
\]

Property 3 (Concurrent)
Given event \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \), where \( i \neq j \)
\[
e_i = PC(e_i[i]) > PC(e_j[j]) > PC(e_j[j]) > PC(e_j[j])
\]

Properties of Vector Clocks: Consistency

Vector clocks can be used to check if a set of \( n \) events constitute the frontier of a consistent cut.

Definition: Two events are pairwise inconsistent if they can’t be on the frontier of the same consistent cut.

Property 4 (Pairwise Inconsistent)
Events \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \), where \( i \neq j \) are pairwise inconsistent if and only if
\[
PC(e_i[i]) < PC(e_j[j]) \land PC(e_j[j]) < PC(e_i[i])
\]

Property 5 (Consistent Cut)
A cut defined by \( \{c_1, \ldots, c_n\} \) is consistent if and only if
\[
\forall i,j: 1 \leq i < j \leq n: PC(c_i[i]) \leq PC(c_j[j])
\]

Properties of Vector Clocks: Gap Detection

Property 6 (Weak Gap Detection)
Given event \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \) if \( PC(e_i[i]) < PC(e_j[j]) \) for some \( k \), then there exists an event \( e_\ell \) such that:
\[
\ell \in [1, \ell] \cap [i, j]
\]

Strong Gap Detection

Recall WGP:
Given event \( e_i \) of process \( p_i \) and \( e_j \) of process \( p_j \) if \( PC(e_i[i]) < PC(e_j[j]) \) for some \( k \), then there exists an event \( e_\ell \) such that:
\[
\ell \in [1, \ell] \cap [i, j]
\]

If \( \ell=k \), then we have strong gap detection: if \( PC(e_i[i]) < PC(e_j[j]) \) then there exists \( e_\ell \) such that:
\[
\ell \in [1, \ell] \cap [i, j]
\]

If \( \ell=k \), then we have strong gap detection: if \( PC(e_i[i]) < PC(e_j[j]) \) then there exists \( e_\ell \) such that:
\[
\ell \in [1, \ell] \cap [i, j]
\]
VCs for Causal Delivery

- Each process increments the local component of its VC only for events that are notified to monitor
- Each notification message is timestamped with VC (event notified through m)
- Monitor keeps all notification messages in a set M

Checking for messages from $p_k$

- Let $m_j$ be the last message that $p_0$ delivered from $p_k$
- Consider $m[j][k]$ and $m_j$
- By Strong Gap Detection, $m[j][k]$ exists only if $TS(m[j][k]) < TS(m_j)[k]$
- Hence deliver $m_j$ as soon as $j : TS(m_j)[k] = TS(m_j)[k]$

Stability

Suppose monitor $p_0$ has received $m_j$ from $p_j$. When is it safe for $p_0$ to deliver $m_j$?
- There is no earlier message in $M$
- There is no earlier message from $p_j$ $TS(m_j)[j] = 1 \cdot \text{no. of messages from } p_j \text{ delivered by } p_0$
- There is no earlier message $m[k]$ from $p_k$, where $k \neq j$

The Protocol

- $p_0$ maintains array $D[1...n]$ of counters
- $D[l] = TS(m_l)[i]$ where $m_l$ is the last message delivered from $p_l$

**DR3:** Deliver $m$ from $p_j$ as soon as both of the following conditions are satisfied:
- $D[l] = TS(m_l)[j] \cdot 1$
- $D[k] \neq TS(m_k)[k], k \neq j$
More on GPD

What if we want to detect non-stable predicates?

Say we want to evaluate \( \Box (\varphi) \)

- by the time the predicate is evaluated, the value of \( \Box (\varphi) \) may have changed
- the global state \( \varphi \) that we use may not have even occurred!

Example

```
\begin{align*}
\text{Detect if the following predicates hold:} & \\
& x = y; x = y - 2
\end{align*}
```

Assume that initially:

\( y = 10 \) and \( x = 0 \)

The Lattice

```
\begin{align*}
\text{Possibly} & : \text{There exists a consistent observation } O \text{ of the computation such that } \Box \varphi \text{ holds in a global state of } O \\
\text{Definitely} & : \text{For every consistent observation } O \text{ of the computation, there exists a global state of } O \text{ in which } \Box \varphi \text{ holds}
\end{align*}
```

Possibly and Definitely

```
\begin{align*}
\text{Possibly} & : \text{There exists a consistent observation } O \text{ of the computation such that } \Box \varphi \text{ holds in a global state of } O \\
\text{Definitely} & : \text{For every consistent observation } O \text{ of the computation, there exists a global state of } O \text{ in which } \Box \varphi \text{ holds}
\end{align*}
```
Computing Possibly and Definitely

- Scan lattice level after level
- To compute Possibly(\[\emptyset\])
  - If \[\emptyset\] holds in one global state, then Possibly(\[\emptyset\])
- To compute Definitely(\[\emptyset\])
  - Given a level, only expand those nodes that correspond to states in which \[\emptyset\]
  - If no such state, announce Definitely(\[\emptyset\])

Definitely (\(x = y\))

Building the Lattice - 1: Collecting Local States

- \(p_0\) collects local states from each process
- For each \(p_i\), keeps the sequence \(Q_i\) of local states in FIFO order

- When is it safe to “drop” a local state \(Q_i\) of \(p_i\)?
- How to build level \(i+1\) of the lattice, given level \(i\)?

Building the Lattice - 2: Garbage Collecting States

To garbage collect \(Q_i\) determine:
- earliest consistent global state \(S_{\min}(Q_i)\)
- latest consistent global state \(S_{\max}(Q_i)\)

that \(Q_i\) can belong to

But what do “earliest” and “latest” mean?

Building the Lattice - 3: Defining Earliest and Latest

Associate a Vector Clock with each consistent global state
- \(S_{\min}(Q_i)\) is the consistent global state with the lowest vector clock to which \(Q_i\) belongs
- \(S_{\max}(Q_i)\) is the one with the highest vector clock

Consistent Global State
Consistent Cut
Frontier of the Cut
Vector Clock
Example

Building the Lattice - 4: Computing $\square_{\min}$

- Label $D_i^l$ with the VC of $c_i$

$S_{\min}(s_{i_k}) = (s_{1c_1}, s_{2c_2}, ..., s_{nc_n})$

$\square_{\text{set}}$ and $D_i^l$ have the same vector clock!

Building the Lattice - 5: Computing $\square_{\max}$

- To build level $l$:
  - wait until states at end of each $Q_i$ have VC such that

$\bigwedge_i \text{VC}(Q_i) \geq n_i$

- To build level $l+1$:
  - For each state $\square_{\text{set}}$ on level $l$, build

$\square_{\text{set}} \sqcap c_i$

- Then, using vector clocks, check whether these global states are consistent

Building the Lattice: Constructing the Levels

- To build level $l$:
  - wait until states at end of each $Q_i$ have VC such that

$\bigwedge_i \text{VC}(Q_i) \geq n_i$

- To build level $l+1$:
  - For each state $\square_{\text{set}} \sqcap c_i$ on level $l$, build

$\square_{\text{set}} \sqcap c_i$

- Then, using vector clocks, check whether these global states are consistent
Multiple Monitors

- Create a group of monitor processes
  - increased performance
  - increased reliability
- Notify through a *causal multicast* to the group
- Each replica will construct a (possibly different) observation
  - if property stable, if one monitor detects, eventually all monitors do
  - otherwise either use Possibly and Definitely
  - or use *causal atomic multicast*
- What about failures?

Sequences of Non-Stable Predicates

Simple predicates (even if not stable) cannot capture dynamic properties of distributed systems

- every acquisition of the lock must be preceded by a release
- variable $x$ should be set to 0 only after variable $y$ has been negative at least once
- message $m$ can be discarded once it becomes stable at all destinations

**Definition**

*Observation $O$ satisfies* $\square = \square_1; \square_2; \ldots; \square_n$ *iff there exists* $\square_1, \square_2, \ldots, \square_n$ *$O$ such that*

$$1 \vdash \square_1[\square_2[\ldots[\square_n[O]]\ldots]]$$