Cuts

A cut \( C \) is a subset of the global history of \( H \)
\[ C = h^{c_1}_1 \cup h^{c_2}_2 \cup \ldots h^{c_n}_n \]

The frontier of \( C \) is the set of events
\[ e^{c_1}_1, e^{c_2}_2, \ldots e^{c_n}_n \]

Global states and cuts

- The global state of a distributed computation is an \( n \)-tuple of local states
  \[ \Sigma = (\sigma_1, \ldots, \sigma_n) \]
- To each cut \((c_1 \ldots c_n)\) corresponds a global state \((\sigma^{c_1}_1, \ldots, \sigma^{c_n}_n)\)

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
Our task

- Develop a protocol by which a processor can build a consistent global state
- Informally, we want to be able to take a snapshot of the computation
- Not obvious in an asynchronous system...

Our approach

- Develop a simple synchronous protocol
- Refine protocol as we relax assumptions
- Record:
  - processor states
  - channel states
- Assumptions:
  - FIFO channels
  - Each $m$ timestamped with $T(\text{send}(m))$
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 \(p_i\)
   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 \(e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C\)

< Definition >
0. \(e_j \in C \equiv T(e_j) < t_{ss}\)

< 0 and 1>
3. \(T(e_j) < t_{ss}\)

< 5 and 3>
6. \(T(e_i) < t_{ss}\)

< Assumption >
1. \(e_j \in C\)

< Property of real time>
4. \(e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)\)

< Definition >
7. \(e_i \in C\)

< Assumption >
2. \(e_i \rightarrow e_j\)

< 2 and 4>
5. \(T(e_i) < T(e_j)\)

Clock Condition

< Property of real time>
4. \(e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)\)

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

Each process maintains a local variable $LC$

$LC(e) \equiv$ value of $LC$ for event $e$

Increment Rules

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

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

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 $e = receive(m) \land (TS(m) \geq t) \land (LC \leq t - 1)$
  \- 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

$mmmmmhhhh...$

An obvious problem

No $t_{ss}!$

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

$mmmmmhhhh...$

An obvious problem

No $t_{ss}!$

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

$mmmmmhhhh...$

An obvious problem

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

$mmmmmhhhh...$

We better relax it...

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

- Take a snapshot at $\Omega$
- 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.

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.

Snapshot 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

Reachability

\[ \Sigma^{kl} \text{ is reachable from } \Sigma^{ij} \text{ if there is a path from } \Sigma^{ij} \text{ to } \Sigma^{kl} \text{ in the lattice} \]
Reachability

$\Sigma_{ij}^k$ is reachable from $\Sigma_{ij}$ if there is a path from $\Sigma_{ij}$ to $\Sigma_{ij}^k$ in the lattice

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

- Deadlock is a stable property
  - Deadlock $\Rightarrow \Box \text{Deadlock}$

- If a run $R$ of the snapshot protocol starts in $\Sigma^i$ and terminates in $\Sigma^f$, then $\Sigma^i \sim_R \Sigma^f$
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 $\Sigma^i \leadsto_R \Sigma^f$
- 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 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  
\downarrow
p_1  
```

```
v_1  
\downarrow
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:
\[ \text{send}_i(m) \rightarrow \text{send}_k(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m') \]
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

**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.

Causal Delivery with Lamport Clocks

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

Should $p_0$ deliver?
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 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
  \[ \theta(e) = \{ e' \in H | e' \rightarrow e \} \cup \{ e \} \]
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
  $\theta(e_i^k) := \{e_i^k\} \cup \theta(e_i^{k-1}) \cup \theta(\text{send}(m))$

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 $h_i$: $\theta_i(e) = h_i^k$—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

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

Message $m$ is timestamped with $TS(m) = VC(\text{send}(m))$
Example

Operational interpretation

Operational interpretation

\[ VC(e_i)[i] = \text{no. of events executed by } p_i \text{ up to and including } e_i \]
\[ VC(e_i)[j] = \text{no. of events executed by } p_j \text{ that happen before } e_i \text{ of } p_i \]