Runs and Consistent Runs

A run is a total ordering of the events in $H$ that is consistent with the local histories of the processors.

- Ex: $h_1, h_2, \ldots, h_n$ is a run.

A run is consistent if the total order imposed in the run is an extension of the partial order induced by $\rightarrow$.

A single distributed computation may correspond to several consistent runs!

Cuts

A cut $C$ is a subset of the global history of $H$:

$$C = h^{c_1}_1 \cup h^{c_2}_2 \cup \ldots \cup h^{c_n}_n$$

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

What \( p_0 \) sees

Not a consistent global state: the cut contains the event corresponding to the receipt of the last message by \( p_3 \) but not the corresponding send event

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 message timestamped with $T(send(m))$

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 $p$
   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. $e_j \in C$  \hspace{1cm} \[ < Assumption > \]
2. $e_i \rightarrow e_j$  \hspace{1cm} \[ < 2 and 4 > \]
3. $T(e_j) < t_{ss}$  \hspace{1cm} \[ < 1 > \]
4. $e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$  \hspace{1cm} \[ < Property of real time > \]
5. $T(e_i) < T(e_j)$  \hspace{1cm} \[ < 5 and 3 > \]
6. $T(e_i) < t_{ss}$  \hspace{1cm} \[ < Definition > \]
Clock Condition

Can the Clock Condition be implemented some other way?

Lamport Clocks

Each process maintains a local variable \( LC \)

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

\[ LC(e_p^{i+1}) < LC(e_q) \]

Increment Rules

\[ LC(e_p^{i+1}) = LC(e_p^i) + 1 \]

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

Timestamp \( m \) with \( TS(m) = LC(\text{send}(m)) \)
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 $e = \text{receive}(m) \land (TS(m) \geq t) \land (LC \leq t - 1)$
  - 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

An obvious problem

No $t_{ss}$!

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

mmmhhhh...

An obvious problem

No $t_{ss}$!

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

mmmhhhh...

Doing so assumes

- upper bound on message delivery time
- upper bound relative process speeds

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

Use empty message to announce snapshot!

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
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.

An Execution and its Lattice

\[
\begin{array}{cccccc}
  p_1 & e_1^1 & e_2^1 & e_3^1 & e_4^1 & e_5^1 & e_6^1 \\
  p_2 & e_1^2 & e_2^2 & e_3^2 & e_4^2 & e_5^2 & e_6^2 \\
\end{array}
\]
An Execution and its Lattice
An Execution and its Lattice

\[
\begin{align*}
\Sigma_{00} & \quad \Sigma_{01} & \quad \Sigma_{02} \\
\Sigma_{10} & \quad \Sigma_{11} & \quad \Sigma_{12}
\end{align*}
\]
Reachability

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

$\Sigma^{ij} \rightarrow \Sigma^{kl}$
So, why do we care about $\Sigma^s$ again?

- Deadlock is a stable property
  - $\text{Deadlock} \Rightarrow \square \text{Deadlock}$
  - If a run $R$ of the snapshot protocol starts in $\Sigma^i$ and terminates in $\Sigma^f$, then $\Sigma^i \rightsquigarrow_R \Sigma^f$

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

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

- Deadlock is a stable property
  - $\text{Deadlock} \Rightarrow \square \text{Deadlock}$
  - If a run $R$ of the snapshot protocol starts in $\Sigma^i$ and terminates in $\Sigma^f$, then $\Sigma^i \rightsquigarrow_R \Sigma^f$

- Deadlock in $\Sigma^s$ implies 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])

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.

To obtain a run, messages must be delivered to the monitor in FIFO order.
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:
send_i(m) → send_i(m') ⇒ deliver_j(m) → deliver_j(m')

Causal delivery generalizes FIFO:
send_i(m) → send_k(m') ⇒ deliver_j(m) → deliver_j(m')

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_k(m') ⇒ deliver_j(m) → deliver_j(m')

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_k(m') ⇒ deliver_j(m) → deliver_j(m')