Global Predicate Detection and Event Ordering
Our Problem

To compute predicates over the state of a distributed application
Model

- Message passing
- No failures
- Two possible timing assumptions:
  1. Synchronous System
  2. Asynchronous System
     - No upper bound on message delivery time
     - No bound on relative process speeds
     - No centralized clock
Asynchronous systems

- Weakest possible assumptions
- cfr. “finite progress axiom”
- Weak assumptions $\equiv$ less vulnerabilities
- Asynchronous $\neq$ slow
- “Interesting” model w.r.t. failures (ah ah ah ah!)
Client-Server

Processes exchange messages using Remote Procedure Call (RPC)

A client requests a service by sending the server a message. The client blocks while waiting for a response.
Client-Server

Processes exchange messages using Remote Procedure Call (RPC)

A client requests a service by sending the server a message. The client blocks while waiting for a response.

The server computes the response (possibly asking other servers) and returns it to the client.
Deadlock!
Goal

Design a protocol by which a processor can determine whether a global predicate (say, deadlock) holds
Wait-For Graphs

Draw arrow from $p_i$ to $p_j$ if $p_j$ has received a request but has not responded yet.
Wait-For Graphs

- Draw arrow from $p_i$ to $p_j$ if $p_j$ has received a request but has not responded yet.

- Cycle in WFG $\Rightarrow$ deadlock

- Deadlock $\Rightarrow$ cycle in WFG
The protocol

$p_0$ sends a message to $p_1 \ldots p_3$

On receipt of $p_0$'s message, $p_i$ replies with its state and wait-for info
An execution
An execution
An execution

Ghost Deadlock!
Houston, we have a problem...

- Asynchronous system
  - no centralized clock, etc. etc.
- Synchrony useful to
  - coordinate actions
  - order events
- Mmmmhhhh...
Events and Histories

- Processes execute sequences of events.
- Events can be of 3 types: local, send, and receive.
- $e^i_p$ is the $i$-th event of process $p$.
- The local history $h_p$ of process $p$ is the sequence of events executed by process $p$.
- $h^k_p$: prefix that contains first $k$ events.
- $h^0_p$: initial, empty sequence.
- The history $H$ is the set $h^0_p \cup h^1_p \cup \ldots h^{p_{n-1}}$.

**NOTE:** In $H$, local histories are interpreted as sets, rather than sequences, of events.
Observation 1:

Events in a local history are **totally ordered**.
Observation 1:
Events in a local history are **totally ordered**

Observation 2:
For every message \( m \), \( \text{send}(m) \) precedes \( \text{receive}(m) \)
Happened-before
(Lamport[1978])

A binary relation $\rightarrow$ defined over events

1. if $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$

2. if $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

3. if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$
Space-Time diagrams

A graphic representation of a distributed execution
Space-Time diagrams

A graphic representation of a distributed execution
Space-Time diagrams

A graphic representation of a distributed execution
Space-Time diagrams

A graphic representation of a distributed execution
Space-Time diagrams

A graphic representation of a distributed execution

$H$ and $\rightarrow$ impose a partial order
Space-Time diagrams

A graphic representation of a distributed execution

H and → impose a partial order
Space-Time diagrams

A graphic representation of a distributed execution

$H$ and $\rightarrow$ impose a partial order
Space-Time diagrams

A graphic representation of a distributed execution

H and \( \rightarrow \) impose a partial order
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!
A cut $C$ is a subset of the global history of $H$

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots h_n^{c_n}$$
A cut $C$ is a subset of the global history of $H$

$$C = h_{1}^{c1} \cup h_{2}^{c2} \cup \ldots h_{n}^{cn}$$

The frontier of $C$ is the set of events

$$e_{1}^{c1}, e_{2}^{c2}, \ldots e_{n}^{cn}$$
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 \((\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})\)
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
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 $m$ 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 \( t_{ss} \)
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. sends an empty message along its outgoing channels
   c. starts recording messages received on each of incoming channels
   d. 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. \( e_j \in C \equiv T(e_j) < t_{ss} \)
  1. \( e_j \in C \)
  2. \( e_i \rightarrow e_j \)
  3. \( T(e_j) < t_{ss} \)
  4. \( e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j) \)
  5. \( T(e_i) < T(e_j) \)
  6. \( T(e_i) < t_{ss} \)
  7. \( e_i \in C \)

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

- **< Property of real time>**
  5. \( T(e_i) < T(e_j) \)

- **< 0 and 1>**

- **< 5 and 3>**

- **< Definition >**

- **< 2 and 4>**
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?
Each process maintains a local variable $LC$

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

$L_C(e_i^p) < L_C(e_{i+1}^p)$

$L_C(e_i^p) < L_C(e_j^q)$
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
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...

- 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

Process does nothing for the protocol during this time!

Take a snapshot at $\Omega$

$TS(m) \geq \Omega$

Records local state $\sigma_i$

Sends empty message:

$TS(m) \geq \Omega$

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 $\Sigma^s$ saved by the snapshot protocol is a consistent global state
Snapshots: a perspective

- The global state $\Sigma^s$ 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 $\Sigma^s$ could have occurred.
Snapshots: a perspective

- The global state $\Sigma^s$ 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 $\Sigma^s$ could have occurred.
- We are evaluating predicates on states that may have never occurred!
An Execution and its Lattice
An Execution and its Lattice

\[ \Sigma^{00} \]
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice

\[ \Sigma \]

\[ \Sigma^{10} \rightarrow \Sigma^{01} \rightarrow \Sigma^{02} \]

\[ \Sigma^{21} \rightarrow \Sigma^{12} \rightarrow \Sigma^{22} \rightarrow \Sigma^{32} \]

\[ p_1 \]

\[ p_2 \]

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

\[ e_2^1 \rightarrow e_2^2 \rightarrow e_2^3 \rightarrow e_2^4 \rightarrow e_2^5 \rightarrow e_2^6 \]
An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice
Reachability

\( \Sigma^{kl} \) is reachable from \( \Sigma^{ij} \) if there is a path from \( \Sigma^{kl} \) to \( \Sigma^{ij} \) in the lattice.
Reachability

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

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

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

\( \Sigma^{ij} \leadsto \Sigma^{kl} \)
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 $\Sigma^i \leadsto^R \Sigma^f$
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 $\Sigma^i \rightsquigarrow_R \Sigma^f$

- Deadlock in $\Sigma^s$ implies deadlock in $\Sigma^f$
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 event of the distributed computation based on the order in which the receiver is notified of the events.