Global Predicate Detection and Event Ordering

Problem

How to compute predicates over the state of a distributed application

Model

Message passing

Asynchronous System
- No upper bound on message delivery time
- No bound on relative process speeds
- No centralized clock

No failures

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 if a set of processors is deadlocked

Real Goal

Design a protocol by which a processor can determine if a given predicate holds in a global state

Any ideas?

Compute Wait-For-Graph!
- Arrow from \( p_i \) to \( p_j \) if \( p_j \) has received a request but has not responded yet

- cycle in WFG
- deadlock
- cycle in WFG

To detect deadlock, use \( p_0 \) to compute WFG of \( p_1, p_2, p_3 \)

The protocol

- \( p_0 \) sends a message to \( p_1, \ldots, p_3 \)
- On receipt of \( p_0 \)'s message, process \( p_i \) replies with state and wait-for info
An execution

Houston, we have a problem...

- Asynchronous system
- How can $p_0$ synchronize the process of collecting the necessary data?

What do we use time for?

- Synchronize actions
- Order events

Can we order events in a Distributed System?

But first...

**Definition:** The local history $h_p$ is the sequence of events executed by a processor $p$.
- $h_p^k$: prefix that contain the first $k$ events
- $h_p^0$: initial, empty sequence

**Definition:** The history $H$ is the set $h_0 \sqcup h_1 \sqcup \ldots \sqcup h_d$.

**Definition:** $e_i^p$ is the $i$-th event of processor $p$. It can be
- a local event
- a send event
- a receive event
Ordering events

- Events in a local history are totally ordered
- For every message \( m \), \( \text{receive}(m) \) is after \( \text{send}(m) \)

Happened before

A binary relation \( \preceq \) defined over events
(Lamport [1978])

1. if \( e_i^k, e_i^l \in h_i \) and \( k < l \), then \( e_i^k \preceq e_i^l \)
2. if \( e_i = \text{send}(m) \) and \( e_j = \text{receive}(m) \), then \( e_i \npreceq e_j \)
3. if \( e_i \npreceq e_j \) and \( e_j \npreceq e_i \) then \( e_i \equiv e_j \)

Space-time diagrams

Given \( H \) and \( \preceq \) we can construct a partially ordered set:
some events cannot be ordered

Runs and consistent runs

Definition: A run is a total ordering of the events in \( H \) that is consistent with the local histories of the processors.

\( h_1, h_2, \ldots, h_n \) is a run

Definition: A run is consistent if the total order imposed in the run is an extension of the partial order induced by \( \preceq \)

Note: A single distributed computation may correspond to several consistent runs
What $p_0$ sees

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

$C = h_1^{c_1} \square h_2^{c_2} \square \ldots \square h_n^{c_n}$

There is a 1-1 correspondence between cuts and global states

Consistent Cuts and Consistent Global States

Definition: A cut is consistent if

$\forall e_i, e_j : e_i \in C \iff e_j \in C$

Observation: A consistent cut defines a unique consistent global state

Our task

- Develop a protocol by which a processor can build a consistent cut
- Informally, we want to be able to take a snapshot of the computation
- We will record
  - processor states
  - channel states

Is this cut consistent? NO!
Our approach

- Develop a simple synchronous protocol
- Refine protocol as we relax assumptions

Snapshot I

Assumptions:
- FIFO channels
- Synchronous system
- Processors timestamp each message with $T(\text{send}(m))$

0: Processor $p_0$ selects $t_{ss}$
1: $p_0$ sends "take a snapshot at $t_{ss}$" to all processes
2: When clock of $p_i$ reads $t_{ss}$ then $p_i$
   - Records its local state
   - Starts recording messages received on each incoming channel
   - Stops recording a channel when receives first message with timestamp greater than or equal to $t_{ss}$
   - Sends an empty message along its outgoing channels

Correctness

Theorem: The protocol produces a consistent cut

Proof: Need to prove $e_j \subseteq C \Rightarrow e_i \subseteq C$

- Property of real time
- Clock condition

Logical Clocks

- A clock that satisfies the Clock Condition is called a logical clock
- Real-time clocks are logical clocks
- Can we implement the Clock Condition in some other way?
Lamport Clocks

- Each process maintains a local variable $LC$
  $LC(e_i) =$ Value of $LC$ for event $e_i$

Increment Rules

$p$  
$\varepsilon_i$  
$\varepsilon_i'$  
$LC(\varepsilon_i') = LC(\varepsilon_i) + 1$

$q$  
$\varepsilon_j$  
$\varepsilon_j'$  
$LC(\varepsilon_j') = \max(LC(\varepsilon_j), LC(\varepsilon_i)) + 1$

Space-Time Diagrams and Logical Clocks

Houston, we still have a problem…

- How do we choose a “logical” $t_{ss}$ so that the message from $p_0$ reaches every other process before $t_{ss}$?

- \textbf{when} $LC = t$ \textbf{do} $S$
  doesn’t make sense for Lamport clocks
  – they are not dense
No \( t_{ss} \)?

- Send \( \text{take checkpoint at } \) where we assume that \( \) is a value that cannot be reached by applying the update rules of logical clocks

SnapShot II

0: processor \( p_0 \) selects \( \)  
1: \( p_0 \) sends "take a snapshot at \( \) to all processes and sets its logical clock to \( \)  
2: when clock of \( p \) reads \( \) then \( p \)
    - records its local state \( \)  
    - 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 \( \)

Hallo-ho? Houston?

- Assumption about \( \) requires to bound relative process speed and message delays…  
  
What about asynchrony?

Here Mission Control… we hear you