CS 138: Ordering and Global State

L11
Last Class

• Time synchronization: NTP & PTP
• Virtual time
  – Logical clocks
  – Vector clocks
  – Total ordering with mutual exclusion
Central-Server Mutual Exclusion

Smart Object

May I?

a

May I?

b

May I?

c
Mutual Exclusion with Logical Clocks

• Requester
  – multicast request with timestamp
  – proceed when all other parties respond OK

• Receiver of request
  – if neither using nor waiting for resource, respond OK
  – if waiting for resource, respond OK if request’s timestamp is lower than own, otherwise queue request
  – if using resource, queue request

• When finished
  – respond OK to queued requests
Mutex Exclusion (1)

1: May I?
Mutex Exclusion (2)

Got It

a

b

OK

OK

c
Mutex Exclusion (3)

Got It
a

Waiting: 2
b

2: May I?

2: May I?

C
Mutex Exclusion (4)
Mutex Exclusion (5)
Mutex Exclusion (6)
Mutex Exclusion (7)

Got It

b

c:3

Waiting:3

C

a

a
Mutex Exclusion (8)

Waiting: 3

OK

b

c: 3

a

C

a

b
Mutex Exclusion (9)
Why Total Order is Important

“if waiting for resource, respond OK if request’s timestamp is lower than own, otherwise queue request”

b:2 < c:2
Total Order

- Tie-breaking rule
  - what if $T_i(a) = T_h(b)$?
  - $a$ comes before $b$ iff $i<h$

- Total order for all events in a distributed system

  "if waiting for resource, respond OK if request’s timestamp is lower than own, otherwise queue request"

  $b:2 < c:2$
Causal Ordering
Causally Ordered Multicast

- Application of vector clocks
  - the only events are sending messages
  - all messages are multicast to all

- Strategy
  - \( P_h \) receives multicast message \( m \) from \( P_i \)
  - deliver \( m \) to application when:
    - \( \text{timestamp}(m)[i] = \text{VC}_h[i] + 1 \)
      - next expected message from \( P_i \)
    - \( \text{timestamp}(m)[k] \leq \text{VC}_h[k] \), for all \( k \neq i \)
      - \( P_h \) has seen all events \( P_i \) had seen when it sent the message
Causally Ordered Multicast (1)

```
P_0  (1,0,0)  (1,1,0)
m_1
P_1  (1,1,0)  (1,0,0)  (1,1,0)
m_2
P_2  (0,0,0)  (1,0,0)  (1,1,0)
middleware
application
```
Causally Ordered Multicast (2)

Causal order != total order
- causal ordering = unrelated msgs delivered in any order
- total ordering = all messages delivered in precise order based on logical clock

\[ \begin{align*}
P_0 & \rightarrow (1,0,0,0) \rightarrow (1,0,1,0) \\
P_1 & \rightarrow (1,0,0,0) \rightarrow (1,0,1,0) \\
P_2 & \rightarrow (0,0,1,0) \rightarrow (1,0,1,0) \\
P_3 & \rightarrow (0,0,1,0) \rightarrow (1,0,1,0)
\end{align*} \]
Global State
Failure Happens

• What to do about it?
  – you of course have everything backed up
  – so, restore the backups
Global State

• Your system consists of 100 nodes
  – each produces a snapshot of itself periodically
  – does some collection of these snapshots constitute a meaningful notion of “global state”? 
Distributed Snapshots (1)

Is this snapshot consistent?
A cut is a **consistent cut** if, for each event $e$ it contains, it also contains all events that happened before $e$.
Checkpointing

• Produce a distributed snapshot
  – how?

• Independent checkpointing
  – each process checkpoints itself periodically when convenient
  – to produce distributed snapshot
    - start with most recent checkpoints
    - roll back until consistent global checkpoint is achieved
Independent Checkpointing

Roll back
Domino Effect

- Initial state
- Checkpoint
- Failure

Time
Coping

- Take independent, periodic checkpoints, plus a few more
- or
- Produce a global snapshot on demand
Independent Checkpoints

• Goal
  – all checkpoints are “useful”
    - no need to roll back

• What are the conditions for checkpoints to for a consistent cut?
Causal Paths

Causal Paths

\[ P_1 \rightarrow C_{1,0} \rightarrow C_{1,1} \rightarrow C_{1,2} \]
\[ P_2 \leftrightarrow C_{2,0} \rightarrow C_{2,1} \rightarrow C_{2,2} \]
\[ P_3 \rightarrow C_{3,0} \rightarrow C_{3,1} \rightarrow C_{3,2} \]

checkpoint interval

\[ m1 \rightarrow m2 \rightarrow m3 \rightarrow m4 \]
Causal Paths

\[ \begin{align*}
\text{P}_1 & \quad \text{C}_{1,0} \quad \text{C}_{1,1} \quad \text{checkpoint interval} \quad \text{C}_{1,2} \\
\text{P}_2 & \quad \text{C}_{2,0} \quad \text{C}_{2,1} \quad \text{C}_{2,2} \\
\text{P}_3 & \quad \text{C}_{3,0} \quad \text{C}_{3,1} \quad \text{C}_{3,2} 
\end{align*} \]
Non-Causal Paths

C₁,₀ — C₁,₁ — C₁,₂

P₁ — m₁ — P₂

C₂,₀ — C₂,₁ — C₂,₂

P₂ — m₂ — P₃

C₃,₀ — C₃,₁ — C₃,₂

C₁,₀, C₁,₁, C₁,₂, C₂,₀, C₂,₁, C₂,₂, C₃,₀, C₃,₁, C₃,₂

checkpoint interval

m₁, m₂, m₃, m₄
Zigzag Paths

P_1

C_{1,0} \quad C_{1,1} \quad \text{checkpoint interval} \quad C_{1,2}

\begin{align*}
m_1 & \quad \text{m2} & \quad \text{m3} & \quad \text{m4} \\
C_{2,0} & \quad C_{2,1} & \quad C_{2,2} \\
C_{3,0} & \quad C_{3,1} & \quad C_{3,2}
\end{align*}
Zigzag Path Definition

• A zigzag path exists from $C_{p,i}$ to $C_{q,k}$ iff there are messages $m_1, m_2, \ldots, m_n$ such that
  – $m_1$ is sent by process $p$ after $C_{p,i}$
  – if $m_h$ ($1 \leq h \leq n$) is received by process $r$, then $m_{h+1}$ is sent by $r$ in the same or a later checkpoint interval (although $m_{h+1}$ may be sent before or after $m_h$ is received), and
  – $m_n$ is received by process $q$ before $C_{q,k}$
Finding Causal Paths

- Use vector clocks
  - components are counts of checkpoints in each process
  - details may be an exercise ...
Domino Effect

- Initial state
- Checkpoint
- Failure

Time
Producing a Consistent Global Snapshot on Demand

• Process A wants all other processes to send it snapshots that together form a consistent cut (and thus a global snapshot)

• Can this be done?
Distributed Snapshot Algorithm

• Chandy & Lamport, 1985
  – algorithm to select a consistent cut
  – any process may initiate a snapshot at any time
  – processes can continue normal execution
    - send and receive messages
  – assumes:
    - no failures of processes & channels
    - strong connectivity
      • at least one path between each process pair
    - unidirectional, FIFO channels
    - reliable delivery of messages
Approach

• Snapshot consists of saved states of all nodes along with messages in transit
• For each pair of directly connected nodes A and B
  – must record messages sent before A saved its state but received after B saved its state
  – nodes send out special *marker* messages immediately after saving their states
Example: Sending

p₁ → m₃ → M → m₂ → m₁ → p₂

state
Example: Receiving

\[ p_1 \xrightarrow{m_3} \xrightarrow{M} \xrightarrow{m_2} m_1 \xrightarrow{p_2} \]

\[ \text{state} \]

\[ \text{state} \]
Another Example: part 1
Another Example: part 2
Another Example: part 3

$p_1$ → state → $p_3$ → state → $p_2$

$m_2$, $M$, $m_3$, $M$, $M$, $m_1$
Another Example: part 4
Snapshot Rules

• **Marker receiving rule for process** $p_i$

  On $p_i$’s receipt of a *marker* message over channel $c$:
  \[
  \text{if (} p_i \text{ has not yet recorded its state) }
  \]
  it records its state
  it records the state of $c$ as the empty sequence
  it turns on recording of messages arriving over other channels
  
  \[\text{else}\]
  $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state and before it received the marker over $c$

• **Marker sending rule for process** $p_i$

  After $p_i$ has recorded its state, for each outgoing channel $c$:
  \[
  p_i \text{ sends one marker message over } c \text{ (before it sends any other messages over } c)\]
Termination

• Process P has completed its part of the algorithm when it has processed markers on all input channels

• It sends its saved local state and channel histories to the initiator
  – the intent is that collection of local states form consistent cut
    - channel histories are the messages in transit at time of cut
Analysis

• Does it find a consistent cut?
  – if so, then for any $P_a$ and $P_b$, if $m$ is a message sent from $P_a$ to $P_b$, then if $\text{recv}(m)$ is in the cut, so is $\text{send}(m)$
    - i.e., if $\text{recv}(m)$ occurred before $P_b$ recorded its state, then $\text{send}(m)$ occurred before $P_a$ recorded its state
  – stronger statement: if for any $P_a$ and $P_b$, if $e_a$ and $e_b$ are events in $P_a$ and $P_b$, such that $e_a$ happens before $e_b$ ($e_a \rightarrow e_b$), then if $e_b$ is in the cut, so is $e_a$
    - i.e., if $e_b$ occurred before $P_b$ recorded its state, then $e_a$ occurred before $P_a$ recorded its state
Proof

• Assume no: $P_a$ recorded its state before $e_a$ occurred ($e_b$ is in the cut, but $e_a$ is not)
  – since $e_a \rightarrow e_b$, there was some sequence of messages $m_1, m_2, \ldots, m_h$ that brought on $e_a \rightarrow e_b$
  – since $P_a$ recorded its state before $e_a$ occurred, it sent marker messages out on all its outgoing channels before transmitting $m_1$
  – since the channels are FIFO, a marker reached $P_b$ before $m_h$
  – but then $P_b$ would have recorded its state before $e_a$
  – but then $e_b$ would not have been in the cut
    - contradiction
More Analysis

• Snapshot taken isn’t necessarily a state that actually happened!
  – but it could have happened …
• If distributed system deadlocks, no distributed snapshot