CS 138: Ordering and Global State
Administrivia

• HW2 is out today, due on the 15th (1 week)

• Review session will be on Monday, March 21st, 5:30pm

• Midterm will be on Tuesday, March 22nd, with material up to Raft (next two classes)
Using Logical Timestamps

- We can use Lamport Clocks to create a total order of events agreed to by all processes
Distributed Banking

add interest based on current balance

deposit $1000
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
Totally Ordered Multicast

• To send multicast:
  – tag message with sender’s timestamp (<time, sender ID>)
    - sender receives own multicast
• On receipt of message
  – queue message in timestamp order
  – multicast an acknowledgement
• On receipt of acknowledgement
  – link to acknowledged message
• Deliver message to application when
  – message is at front of receive queue
  – has been acknowledged by all
Totally Ordered Multicast

SFO(1)

In
1: compute interest (1,1)
2: compute interest (1,1)
4: ack interest-1
5: deposit $1000 (1,2)
7: ack deposit-1
8: ack deposit-2
9: ack interest-2

Out
3: ack interest-1
6: ack deposit-1

PVD(2)

In
3 : ack deposit-2
4: ack deposit-2
5: compute interest (1,1)
7: ack interest-2
8: ack interest-1
9: ack deposit-1

Out
1: deposit $1000 (1,2)
2 : deposit $1000 (1,2)
6: ack interest-2

PVD must reorder queue once all acks are in
Mutual Exclusion

- Central server
- Logical clocks
Central-Server Mutual Exclusion

May I?

Smart Object

May I?

a

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)
Mutex Exclusion (3)

```
<table>
<thead>
<tr>
<th>Got It</th>
<th>Waiting:2</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2: May I?</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2: May I?</td>
<td></td>
</tr>
</tbody>
</table>

Got It

b:2

b

Copyright © 2015 Thomas W. Doeppner, Rodrigo Fonseca. All rights reserved.
Mutex Exclusion (5)

Got It
a

Waiting:2
b

Waiting:3
C
c:3

b:2

3: May I?

c:3

3: May I?
Mutex Exclusion (6)

Waiting: 2

b

Waiting: 3

c

OK

a

OK

Waiting: 3

C

a
Mutex Exclusion (7)
Mutex Exclusion (8)

Diagram:
- Circle labeled "b" with an arrow labeled "OK" pointing to a circle labeled "Waiting: 3 C" with triangles labeled "a" and "b".
- Circle labeled "a".
- Circle labeled "c: 3".
Mutex Exclusion (9)

Diagram:
- Circle labeled 'b'
- Circle labeled 'a'
- Circle labeled 'C'

Text:

Got It
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
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)
Causally Ordered Multicast (2)

\[ \begin{align*}
P_0 & \quad (1,0,0,0) \\
\quad m_1 & \quad (1,0,1,0) \\
\quad & \quad (1,0,1,0) \\
P_1 & \quad (1,0,0,0) \\
\quad & \quad (1,0,1,0) \\
\quad m_2 & \quad (0,0,1,0) \\
\quad & \quad (1,0,1,0) \\
P_2 & \quad (0,0,1,0) \\
\quad & \quad (1,0,1,0) \\
\quad & \quad (0,0,1,0) \\
\quad & \quad (1,0,1,0) \\
\end{align*} \]
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)
A cut is a **consistent cut** if, for each event $e$ it contains, it also contains all events that happened before $e$. 

---

**Distributed Snapshots (2)**

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

P1

Initial state

P2

Checkpoint

Time

Failure
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

\[
\begin{align*}
P_1 & \rightarrow C_{1,0} \rightarrow C_{1,1} \rightarrow C_{1,2} \\
P_2 & \rightarrow C_{2,0} \rightarrow C_{2,1} \rightarrow C_{2,2} \\
P_3 & \rightarrow C_{3,0} \rightarrow C_{3,1} \rightarrow C_{3,2}
\end{align*}
\]

\(\text{checkpoint interval}\)
Causal Paths

$P_1$ $C_{1,0}$ $C_{1,1}$ $C_{1,2}$

$P_2$ $C_{2,0}$ $C_{2,1}$ $C_{2,2}$

$P_3$ $C_{3,0}$ $C_{3,1}$ $C_{3,2}$

$m_1$ $m_2$ $m_3$ $m_4$

checkpoint interval
Non-Causal Paths

C1,0  C1,1  C1,2
P1  checkpoint interval

C2,0  C2,1  C2,2
P2  m1  m3

C3,0  C3,1  C3,2
P3  m2  m4

m1, m2, m3, m4, checkpoint interval
Zigzag Paths

P_1 \quad C_{1,0} \quad C_{1,1} \quad \text{checkpoint interval} \quad C_{1,2}

P_2

P_3

C_{2,0} \quad C_{2,1} \quad C_{2,2}

C_{3,0} \quad C_{3,1} \quad C_{3,2}

m1 \quad m2 \quad m3 \quad m4
Zigzag Path Definition

• A zigzag path exists from $C_{p,i}$ to $C_{q,k}$ iff there are messages $m_1$, $m_2$, ... $m_n$ 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 $C_{q,k}$
Theorem

- A set of checkpoints $S$, each from a different process, can belong to the same consistent global snapshot iff no checkpoint in $S$ has a zigzag path to any other checkpoint (including itself) in $S$
Zigzag Cycles

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

m1, m2, m3, m4
Corollary

• A checkpoint is *useful* if it potentially belongs to some consistent global checkpoint

• Corollary: A checkpoint is useful iff it is part of no zigzag cycle
Adaptive Checkpointing

• On receipt of a message, receiver checks if message completes a zigzag cycle
  – if so, a new checkpoint is taken before the message is processed
  – thus, no cycle
However ...
Coping …

• On receipt of message, check for a causal path to a checkpoint preceding the send
  – the path plus the just-received message form a zigzag cycle
• Doesn’t catch all zigzag cycles
  – testing shows it catches most of them
Finding Causal Paths

• Use vector clocks
  – components are counts of checkpoints in each process
  – details may be an exercise …
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

\[
\begin{align*}
  & p_1 \\
  & \downarrow \text{state} \\
  \quad & \quad \quad \quad \quad m3 \rightarrow M \rightarrow m2 \rightarrow m1 \\
  \quad & \quad \quad \quad \quad \downarrow \\
  & p_2
\end{align*}
\]
Example: Receiving

![Diagram showing states and transitions between processes p1 and p2. The states are labeled as state and state, with processes marked as m1, m2, m3, and M. The transitions are numbered 1, 2, 3, and 4.]
Another Example: part 1

\[ \text{state} \]

\[ p_1 \rightarrow m1 \rightarrow p_2 \]

\[ p_3 \rightarrow M \rightarrow p_2 \]

\[ p_1 \rightarrow 3 \rightarrow p_2 \]

\[ p_1 \rightarrow 2 \rightarrow \text{state} \]

\[ p_1 \rightarrow 1 \rightarrow p_3 \]
Another Example: part 2

```
state

M

m1

M

m2

1

2

3

4

p1

p2

p3
```
Another Example: part 3

Diagram showing a state transition graph with nodes labeled as follows:

- $p_1$ connected to $p_3$ and $p_2$
- $p_2$ connected to $p_3$
- $p_3$ connected to $p_1$ and $p_2$

Transition labels include:

- $m_2$
- $m_3$
- $r(m_2)$

States are indicated by squares, and actions by circles with labels.
Another Example: part 4

Diagram:

- **p₁**
  - state

- **p₂**
  - state
  - r(m2)

- **p₃**
  - m3
  - 1
  - 2
  - M

- **state**
Snapshot Rules

• *Marker receiving rule for process* $p_i$

  On $p_i$'s receipt of a *marker* message over channel $c$:
  
  *if* ($p_i$ 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

  *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 $c$:

  $p_i$ sends one marker message over $c$ (*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
Example (part 1)

\[ p_1 \rightarrow c_2 \rightarrow p_2 \]

\[ \text{account: } \$1000 \]
\[ \text{widgets: } (\text{none}) \]

\[ \text{account: } \$50 \]
\[ \text{widgets: } 2000 \]
Example (part 2)

1. Global state $S_0$

2. Global state $S_1$

3. Global state $S_2$

4. Global state $S_3$

(M = marker message)
Reachability

actual execution: \( e_0, e_1, \ldots, e_N \)

pre-snap: \( e'_0, e'_1, \ldots, e'_{R-1} \)

post-snap: \( e'_R, e'_{R+1}, \ldots, e'_N \)
Global Properties

• Safety
  – bad things will not happen
  – e.g., mutual exclusion is a safety property

• Liveness
  – good things will happen
  – e.g., termination is a liveness property

• Stable properties
  – once true — always true

• Transient properties
  – once true — who knows?
Stable Global Properties

(a) Garbage collection

(b) Deadlock

(c) Termination
Transient Properties

• Distributed debugging
  – assert(∀a≠b (|x_a − y_b| < 10))
    - x_a and y_a reside in process a
How To …

• State collection
  – each process sends snapshots to central server
  – contain vector timestamps

• Central server checks for transient property $\phi$
  – looks at global states that could have resulted from initial state, given vector timestamps

• possibly $\phi$
  – if $\phi$ holds in at least one of them

• definitely $\phi$
  – for all possible (causally consistent) orderings, $\phi$ holds at some point