CSE535 Asynchronous Systems

Snapshots for Non-FIFO Channels

YoungMin Kwon

Difficulties in Non-FIFO Channels

- In FIFO channels
  - Messages sent after a marker are received after the marker
  - It ensures that messages received or in transit in the past are not from the future
- In Non-FIFO channels
  - Markers cannot delineate messages to be recorded or not to be recorded
- Two Approaches
  - Inhibit message transmission
  - Piggyback control information on messages

Inhibition Method

- Channel state
  - Only the number of messages in transit along the channel
  - # of msgs in transit = # of msgs sent - # of msgs received.

- When P_j receives a marker, P_j returns an ack immediately.

- After sending a marker to P_j, P_i inhibits sending any messages to P_j until it knows that P_j has recorded its local state.
  - Receiving an ack or a marker message from P_j ensures that P_j recorded its local state.

Inhibition Method

- C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$
  - m_{ij} is counted as sent
  - m_{ij} is either counted as received or in transit *(# of msgs sent - # of msgs received*)

- C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$
  - Sending a future message is inhibited until P_j saves its state
  - m_{ij} is not counted as sent or received

Lai-Yang Algorithm

- Two roles of a marker in FIFO systems
  - Ensures that C2 is satisfied for process states LS_i
    C2: future messages are not received nor in transit
  - Inform P_j the set $\{send(m_{ij}) | send(m_{ij}) \in LS_i\}$ so that the channel state can be computed as $transit(LS_i, LS_j)$

Lai-Yang Algorithm

- Coloring scheme on messages (piggybacking color)
  - Every process is initially white and turns red after taking a snapshot
  - White (red) processes send white (red) messages
  - Before receiving a red message, each white process records its state at a convenient time.

- First role of a marker:
  - Ensure that future messages are not in the recorded process state
- Second role of a marker:
  - Enable computing transit(LS_i,LS_j)

Lai-Yang Algorithm

- Algorithm
  - White processes record a history of all white messages sent or received along each channel
  - When a process turns red, it sends this history and its snapshot to the initiator process
  - The initiator computes transit(LS_i,LS_j)
    - SC_{ij} = {white messages sent by P_i on C_{ij}} – {white messages received by P_j from C_{ij}}
      = $\{m_{ij} \mid send(m_{ij}) \in LS_i\} - \{m_{ij} \mid rec(m_{ij}) \in LS_j\}$

Lai-Yang Algorithm

- The algorithm requires each process to record the entire message history
- In some applications, such as termination detection, the number of messages in transit is enough
  - For these applications, record only the number of messages sent and received

Li *et al.*'s Algorithm

- Accommodates
  - Repeated invocations and
  - Multiple initiators

- The state of a channel is the number of messages in transit
  - State of C_{ij} is

    # of msg in C_{ij} in the previous snapshot
    + # of msg sent on C_{ij} since the last snapshot
    - # of msg received from C_{ij} since the last snapshot

Li *et al.*'s Algorithm

- All messages sent after a local snapshot are tagged with ⟨init_id, seq⟩
  - Generalization of white/red coloring scheme
  - init_id: initiator's id
  - seq: the sequence number

- Process state recording rules are analogous to those of Lai-Yang Algorithm

Li *et al.*'s Algorithm

- Channel State Recording Rules
  - Before receiving a new sequence number
    - Count the # of msgs sent and msgs received along each channel
    - Send the counts and its snapshot to the initiator
  - The initiator computes the channel state C_{ij} as
    - SC_{ij} = transit(LS_i,LS_j)
      = $TRANSIT_{ij}$
      + # of msg sent on C_{ij} since the last snapshot
      - # of msg received from C_{ij} since the last snapshot

Li *et al.*'s Algorithm

- If the initiator starts a new snapshot before finishing the previous one
  - A process may receive a lower sequence number while participating in a snapshot initiated later
  - Use the snapshot for the higher sequence number

Snapshot in a Causal Delivery System

- Process State Recording
  - An initiator broadcasts a token to all processes including itself
  - On receiving the token P_i records its state and sends it to the initiator
  - No need to send markers on all channels

- P1: $send(m_{ij}) \notin LS_i \Rightarrow rec(m_{ij}) \notin LS_j$
  - *Suppose that* $rec(token_{ii}) \rightarrow send(m_{ij})$
  - *Then,* $send(token_{ij}) \rightarrow send(m_{ij})$
  - *Causal ordering ensures that* $rec(token_{ij}) \rightarrow rec(m_{ij})$
  - *Because* LS_j *is recorded at* $rec(token_{ij})$, $rec(m_{ij}) \notin LS_j$

Snapshot in a Causal Delivery System

Channel State Recording (Acharya-Badrinath Algorithm):

- Each process P_i maintains arrays $sent_i[1..n]$ and $recv_i[1..n]$.
  - $sent_i[j]$: # of msgs sent from P_i to P_j
  - $recv_i[j]$: # of msgs received by P_i sent from P_j
- When P_i records its local state, include $sent_i$ and $recv_i$ in the snapshot
- From the initiator
  - SC_{ij} is empty when P_i is the initiator
  - SC_{ij} is the set of messages whose sequence number is $\{recv_j[i]+1,\ldots,sent_i[j]\}$

Snapshot in a Causal Delivery System

- Channel State Recording (Alagar-Venkatesan Algorithm)
  - Use the vector timestamp in the message
  - A message m is old if send(token) $\not\rightarrow$ send(m)
    - Textbook: send(m) $\rightarrow$ send(token), I think it is incorrect
    - send(token) $\not\rightarrow$ send(m) includes concurrent case also
  - Otherwise m is new

Snapshot in a Causal Delivery System

- When a process P_i receives a token
  - Save LS_i
  - Initialize $SC_{ij} = \varnothing$ for j = 1..n
  - Send Done message to the initiator
  - Add received messages to SC_{ij} if they are old
- The initiator broadcasts Terminate message after receiving Done messages from all processes
- The process stops the snapshot algorithm after receiving the Terminate message

Snapshot in a Causal Delivery System

- C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$
  - Causal ordering: no new message is delivered before the token
  - This condition and P1 ensures implies C2

- C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$
  - Each old message m_{ij} is delivered before the token is delivered or before the Terminate is delivered.
  - Hence, m_{ij} is either in LS_i or SC_{ij}

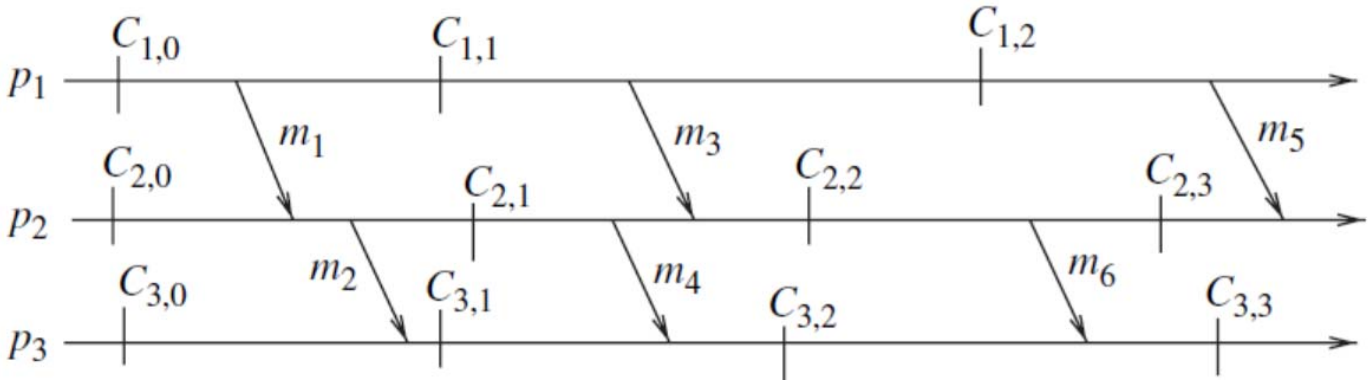
Necessary and Sufficient Conditions for Consistent Global Snapshots

- Many applications record their local process states periodically
  - For failure recovery, debugging, monitoring, verifications…
- Can we find a set of saved local process states, one from each process, that might have been taken concurrently or simultaneously?

Necessary and Sufficient Conditions for Consistent Global Snapshots

- Local checkpoint: a saved intermediate state of a process during its execution
- A global snapshot is a set of local checkpoints one from each process
- A consistent global snapshot does not have a causal path between any two checkpoints
  - A consistent snapshot comprises concurrent process states

Consistent Global Snapshots



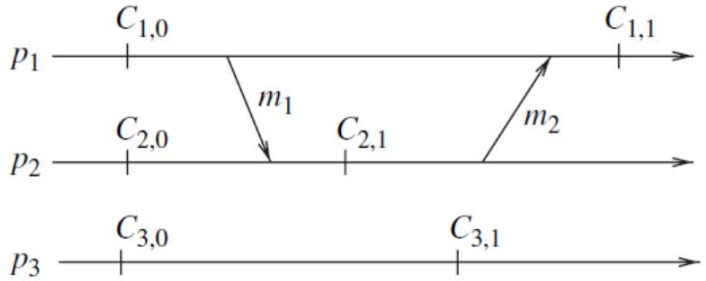
- Even if two local checkpoints do not have a causal path, they may not belong to the same consistent global snapshot
  - $C_{1,1}$ and $C_{3,2}$ cannot be in a consistent global snapshot together
  - Because of m_4, $\{C_{1,1}, C_{2,1}, C_{3,2}\}$ cannot be consistent
  - Because of m_3, $\{C_{1,1}, C_{2,2}, C_{3,2}\}$ cannot be consistent

Zigzag Paths

- A zigzag path exists from a checkpoint $C_{x,i}$ to a checkpoint $C_{y,j}$ iff there exists messages $m_1, m_2,\ldots,m_n$ such that
  - m_1 is sent by P_x after $C_{x,i}$
  - If m_k $(1 \leq k \leq n\text{-}1)$ is received by P_z, then m_{k+1} is sent by P_z in the same or a later checkpoint interval (m_{k+1} may be sent **before** or after m_k is received)
  - m_n is received by p_y before $C_{y,j}$
- A zigzag path exists from $C_{1,1}$ to $C_{3,2}$ due to m_3 and m_4
- A checkpoint C is in a zigzag cycle iff there is a zigzag path to itself

Zigzag Path

- A causal path is a Zigzag path
- A zigzag path may not be a causal path
  - A path from $C_{1,0}$ to $C_{3,1}$ by m_1 and m_2 is a causal path and is a zigzag path
  - A path from $C_{1,1}$ to $C_{3,2}$ by m_3 and m_4 is a zigzag path, but is not a causal path
- A zigzag path can form a cycle, but a causal path cannot

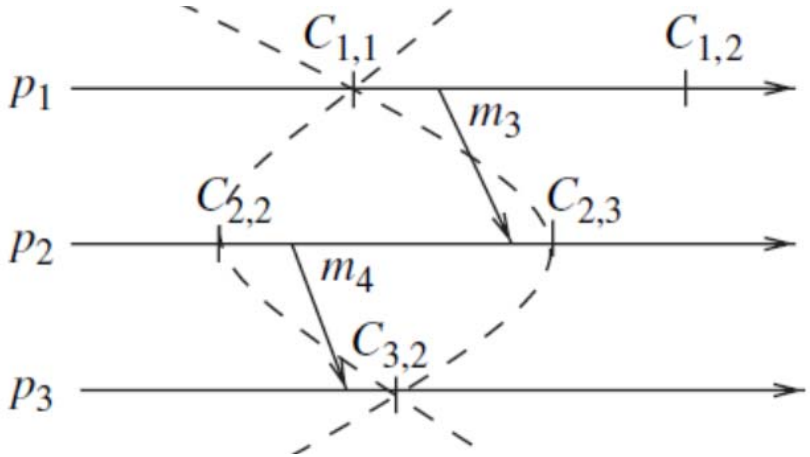


Consistent Global Snapshots

- Netzer and Xu proved that
  - If no zigzag path exists between any two checkpoints from a set S of checkpoints, then a consistent snapshot can be formed that includes the set S and vice versa

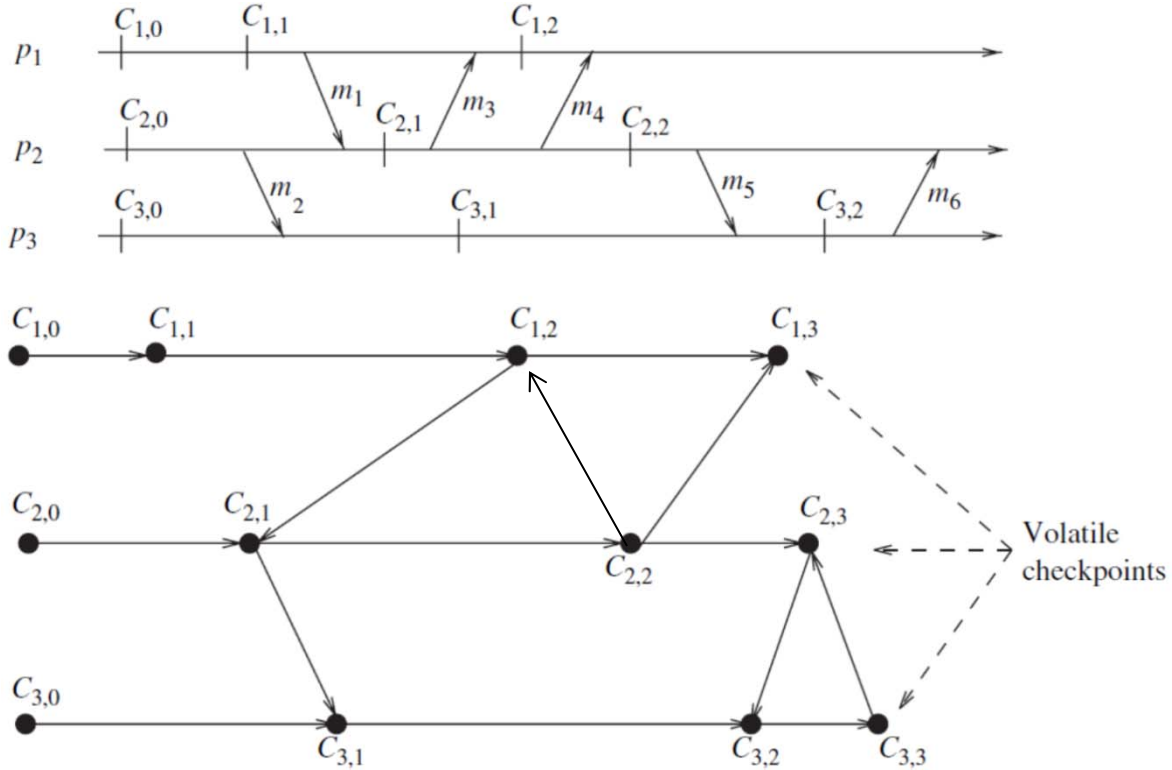
Consistent Global Snapshots

- Intuitive explanation
  - If a zigzag path is a causal path, no consistent global state.
  - If a zigzag path is not a causal path, the zigzag nature of the path make two paths inconsistent



Rollback-Dependency Graph

- R-graph: G=(V,E)
- V is the set of checkpoints, $(C_{p,i},C_{q,j}) \in E$ if
  - p = q and j = i + 1 or
  - $p \neq q$ and a message sent from i^{th} checkpoint interval of P_p is received by P_q in its j^{th} checkpoint interval



Finding Z-path

- Construction of R-Graph
  - When P_p sends a message m in its i^{th} interval, it piggybacks the pair (p,i) with m
  - When the receiver P_q receives m in its j^{th} interval, it records the existence of the edge $C_{p,i}$ to $C_{q,j}$
  - A process can broadcast a message to collect the edges and construct the R-graph
  - Volatile checkpoints are appended at the end of each process

Finding Z-Path

- Let G=(V,E) be the R-graph, then there is a Z-path from $C_{p,i}$ to $C_{q,j}$ iff
  - p = q and i < j or
  - There is a path from $C_{p,i+1}$ to $C_{q,j}$ in G.