Ordering Messages in Distributed Virtual Environments

Richard M. Fujimoto
Professor

Computational Science and Engineering Division
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0765, USA

http://www.cc.gatech.edu/~fujimoto/

Copyright © Richard M. Fujimoto
Outline

• Causal Message Ordering
  – Happens before relationship
  – Causal order message delivery

• Causal Order Implementation
  – Central message dispatcher
  – Barriers
  – Vector time stamps

• Limitations
• “Things” happen in the real world in a certain order (e.g., cause & effect).
• It should appear that events in the simulated world happen in the same order as the real world actions that they represent.

Correct ordering of events can be achieved by assigning a time stamp (logical time) to each event, and ensuring events are delivered in time stamp order (e.g., using conservative or optimistic synchronization)

This may entail significant computation/communication overheads
Causal Order

• Defined for distributed computing in general (not just simulation)

• Assumptions
  – Set of processes; message-based communication
  – Each process’s execution a sequence of actions
    • Local computation (event)
    • Send message
    • Deliver message to process

• Causal order defines an ordering among actions/messages
The “Happens Before” Relationship

Definition (Lamport `78) “happens before” relationship ( -> ):
Consider two actions (event, send, or deliver), A₁ and A₂
• if A₁ & A₂ occur in the same process and A₁ precedes A₂, then A₁ -> A₂
• if A₁ is a send, and A₂ is a delivery of the same message, then A₁ -> A₂
• if A₁ -> A₂ and A₂ -> A₃, then A₁ -> A₃ (transitivity)
Actions that are not causally related are said to be concurrent
Basic idea: If there is a left-to-right path from A₁ to A₂, then A₁ -> A₂
A message delivery service is said to be **causally ordered** if for any two messages $M_1$ and $M_2$ sent to the same federate where $M_1$ and $M_2$ contain notices for events $E_1$ and $E_2$, respectively, and $E_1 \rightarrow E_2$, then $M_1$ is delivered to the federate before $M_2$.

Messages for concurrent events may be delivered in any order.

**Observation:** Causal order message delivery avoids certain anomalies that might occur in receive order delivery. Causally order does not consider application semantics - may impose an ordering on independent events.
Outline

• Causal Message Ordering
  – Happens before relationship
  – Causal order message delivery

• Causal Order Implementation
  – Central message dispatcher
  – Barriers
  – Vector time stamps

• Limitations
Central Dispatcher Implementation

- Assume reliable and ordered communication (e.g., TCP)
- Route all communications through a central message dispatcher (within the RTI) that forwards message to its destination

Any event that depends on Fire event must be sent by the dispatcher after it sends all messages for the Fire event, ensuring causal order delivery
Implementation Using Barriers

Assume reliable, ordered message delivery

Main loop in each simulator:

While (not done with simulation)

Deliver all events from RTI

Barrier

Process events, send messages

Flush barrier

Barrier serializes causally related events in different processors
Vector Clocks

• Applicable to closed multicast group (sender is also a member of the group)

• Process $i$ maintains vector clock $C_i$:
  - $C_i[i] = \text{number of messages process } i \text{ has sent to group}$
  - $C_i[j] (j \neq i) = \text{number of messages sent to group by process } j \text{ that have been delivered to process } i$ (number of messages from $j$ causally preceding process $i$’s current event)
  - Example: $C_1 = (1, 2, 3, 4)$
    - Process 1 has sent 1 message
    - Process 1 has delivered 2 messages from process 2, 3 messages from process 3, and 4 messages from process 4

• Attach a vector time stamp to each message; when process $i$ sends a message:
  - Increment $C_i[i]$
  - Use process $i$’s vector clock as message time stamp
  - Indicates which messages causally precede this message
Message Delivery

Rules for message delivery

• A message M with time stamp T[S] is sent by process S to process R

• This message can be delivered to R when
  – T[S] = C[R][S] + 1, and /* check this is the next message from S */
  – T[j] ≤ C[R][j], all j ≠ S, /* messages preceding M already delivered */
  – /* If T[j] < C[R][j], R has received other messages from process j */
Outline

• Causal Message Ordering
  – Happens before relationship
  – Causal order message delivery

• Causal Order Implementation
  – Central message dispatcher
  – Barriers
  – Vector time stamps

• Limitations
Observation: causal order allows different federates to receive messages for concurrent events in different orders.

- pilot 1 has orders to attack the first enemy aircraft to take off; pilot 2 has orders to attack the second.
- both pilots attack enemy aircraft #2.

Causal and totally ordered: all federates receiving messages for the same events receive them in the same order.
Even with total ordering, no ordering guarantees for concurrent events

- Federate A has orders to fire upon first target to come with range
- Federate B comes into range first, then Federate C comes into range
- “Come into range” events are concurrent; causal order does not guarantee any order of delivery
- Federate C’s message is delivered to Federate A first; C is incorrectly fired upon.
- Timestamp order ensures proper order of delivery
Causal/Total Order: Limitations

Hidden dependencies: dependencies between events that are not conveyed explicitly via messages may not be preserved.

- Federate A issues orders for operation (diversion, then main attack)
- Federate B begins diversion attack
- Federate C begins main attack
- Messages from B and C are not causally related; enemy federate may observe the main attack before the diversion!
- Timestamp order guarantees proper order of delivery
Summary

• Receive order is commonly used in virtual environments, but can lead to anomalies
• Time stamp order solves problem, but at cost of relatively high overheads
• Causal order provides an alternate solution
  – Based on Lamport’s happens-before relationship
  – Relatively simple implementations (central dispatcher, vector time stamps) available
  – Does not eliminate all temporal anomalies
• Total ordering eliminates more, but not all anomalies