CSCI 5673
Distributed Systems

Lecture Set Three

Message Ordering

Lecture Notes by
Shivakant Mishra
Computer Science, CU Boulder
Last update: January 31, 2017

Message Ordering

- Maintaining an order in the *delivery* of all messages exchanged in a distributed system is important
- Message receive vs message delivery
  - Order in which messages are received depends on the underlying communication protocol
    - UDP, TCP, …
  - Order of message delivery to a higher-level application is determined by a middleware layer: *ordered message delivery layer*

<Figure>

Middleware Layer

- Message delivery ordering
- Communication paradigms: RPC, …
- Support for fault tolerance: group communication
- Support for security: admission control, …
- *etc.*

- Why ordered message delivery layer?
  - Application requirements
    - Database update
    - Replicated objects
- Types of message delivery ordering
  - FIFO
  - Causal
  - Total
  - Total order based on causal order

FIFO ordering

- Messages m_1 and m_2 are sent from the same process, such that

  $\text{send}(m_1) \rightarrow \text{send}(m_2)$

- FIFO delivery order: Every process that delivers both m_1 and m_2 must deliver m_1 before m_2.

FIFO Ordering: Implementation

- Ideas???

FIFO Ordering: Implementation

- Each process *i* maintains a sequence number C_i
  - C_i is initialized to 0
  - On sending a message *m*, *i* does the following
    - Attach $<i, C_i>$ to *m*
    - Increment C_i
- Each process *i* maintains a vector *V* of size *n* (*n*: # of processes)
  - All entries of *V* are initialized to *–1*
  - On receiving a message *m* (m:$<i,C_i>$), a process *j* does the following
    - If $V[\ i\ ] < C_i$,
      - deliver m; $V[\ i\ ] = C_i$
    - Otherwise discard *m*

FIFO Ordering: Observation

- Notice that our FIFO implementation doesn’t provide any other properties, e.g. reliability, security, etc.
  - A message sent by a process may not be delivered
  - A message delivered by a process may have been altered
- If additional properties are needed, they will be implemented in the middleware layer in conjunction with FIFO
  - Reliable FIFO ordering
  - Secure FIFO ordering
  - …

Reliable FIFO Ordering: Implementation

- Each process *i* maintains a sequence number C_i
  - C_i is initialized to 0
  - On sending a message *m*, *i* does the following
    - Attach $<i, C_i>$ to *m*
    - Increment C_i
- Each process *i* maintains a vector *V* of size *n* (*n*: # of processes)
  - All entries of *V* are initialized to *–1*
  - On receiving a message *m* (*m*:$<i,C_i>$), a process *j* does the following
    - If $C_i == V[i] + 1$
    - If $C_i > (V[i] + 1)$
    - If $C_i <= V[i]$

Reliable FIFO Ordering: Implementation

- If $C_i == V[i] + 1$
  - Deliver m; $V[i] = V[i] + 1$
- If $C_i <= V[i]$
  - Discard m
- If $C_i > (V[i] + 1)$
  - Cannot deliver m at this time
  - Either discard m or buffer m to be delivered at a later time

Reliable FIFO Ordering: buffer messages

- If $C_i > (V[i] + 1)$
  - Buffer m to be delivered at a later time
  - When (?)
- If $C_i == V[i] + 1$
  1. Deliver m; $V[i] = V[i] + 1$
  2. Check the buffer: if there is a message m: $<i,C_i>$ in the buffer such that $C_i == V[i] + 1$, go to 1

- Question: What if a message that was sent is lost?

Recovering Lost Messages

- If a message is lost in the transit, the sender needs to resend that message
- How does the sender know that a message it had sent has been lost?
- Two approaches:
  - Positive Acknowledgement
  - Negative Acknowledgement

Positive Acknowledgement

- After sending a message, the sender starts a timer and buffers that message
- On receiving a message, the receiver sends an acknowledgement to the sender. The ack contains the sequence number of the message received
- If the timer expires before receiving an ack, the sender resends the message and starts a new timer
  - Note: The sender keeps the message in the buffer
- On receiving an ack for a message, the sender purges that message from its buffer and cancels the corresponding timer

Reliable FIFO: Positive Acknowledgement

- Each process *i* maintains a send buffer, a receive buffer, a sequence number C_i and a vector V of size n (n: # of processes)
  - C_i is initialized to 0
  - All entries of V are initialized to -1
  - On sending a message *m*, *i* does the following
    - Attach $<i, C_i>$ to *m*
    - Increment C_i
    - Store $m<i, C_i>$ in the send buffer
    - Start a timer $T_i<m<i, C_i>>$

Reliable FIFO: Positive Acknowledgement

- On receiving a message *m* (m:<i,C_i>), a process *j* does the following
  - Send an ack <m:<i,C_i>> to process *i*
  - If C_i == V[i] + 1
    1. Deliver m; V[i] = V[i] + 1
    2. Check the receiver buffer: if there is a message m: <i,C_i> in the buffer such that C_i == V[i] + 1, go to 1
  - If C_i > (V[i] + 1)
    - Store m in the receive buffer to be delivered at a later time
  - If C_i <= V[i]
    - Discard *m*

Reliable FIFO: Positive Acknowledgement

- On receiving an ack (m:<i,C_i>), a process i does the following
  - Cancel timer T_i<m<i, C_i>>
  - Purge m from the send buffer
- On timer expiry of T_i<m<i, C_i>>
  - Resend m from the send buffer; keep m in the send buffer
  - Start a timer T_i<m<i, C_i>>

Reliable FIFO: Positive Acknowledgement

- *Middleware figure and example*

Reliable FIFO: Positive Acknowledgement

- Some observations
  - A reliable FIFO middleware layer *guarantees* that a message sent is *eventually* received
  - No guarantees on the time it takes to deliver a message
  - Overhead: For every message sent
    - At least one ack message for every message sent
    - Timer
    - Send and receive buffers
  - What if there is a network partition between the sender and the receiver?
  - What if the sender or the receiver process fail?

Reliable FIFO: Positive Acknowledgement

- Other system issues
  - How large should the send and the receive buffers be?
  - What is an appropriate length of a timer?
  - What if the send or the receive buffers fill up?

Negative Acknowledgement

- One problem with positive acknowledgement is high overhead – at least one extra message for every every message sent even when there are no message losses
- Negative acknowledgement addresses this overhead
- After sending a message, the sender buffers that message
- On receiving a message, the receiver sends a *retransmit* message to the sender if it discovers that it has missed an earlier message. The retransmit message contains the sequence number of the missed message received
- On receiving a retransmit message, the sender resends the requested message from its buffer

Reliable FIFO: Negative Acknowledgement

- Each process *i* maintains a send buffer, a receive buffer, a sequence number C_i and a vector V of size n (n: # of processes)
  - C_i is initialized to 0
  - All entries of *V* are initialized to *–1*
  - On sending a message *m*, *i* does the following
    - Attach <*i*, C_i> to *m*
    - Increment C_i
    - Store *m*<*i*, C_i> in the send buffer

Reliable FIFO: Negative Acknowledgement

- On receiving a message *m* (m:<i,C_i>), a process *j* does the following
  - If $C_i == V[i] + 1$
    1. Deliver m; $V[i] = V[i] + 1$
    2. Check the receiver buffer: if there is a message m: <i,C_i> in the buffer such that $C_i == V[i] + 1$, go to 1
  - If $C_i > (V[i] + 1)$
    - Store m in the receive buffer to be delivered at a later time
    - Send a retransmit message to *i* for each *k* such that
      - $C_i > k > V[i]$ and m<i, k> is not in the receive buffer
  - If $C_i <= V[i]$
    - Discard *m*

Reliable FIFO: Negative Acknowledgement

- On receiving a retransmit request for a message *m<i,k>,* a process *i* does the following
  - Resend m from the send buffer; keep m in the send buffer

Reliable FIFO: Negative Acknowledgement

- *Example*

Reliable FIFO: Negative Acknowledgement

- Some observations
  - A reliable FIFO middleware layer *guarantees* that a message sent is *eventually* received (caveat?)
  - No guarantees on the time it takes to deliver a message
  - Overhead: For every message sent
    - For every lost message, at least one retransmit message
    - Send and receive buffers
  - What if there is a network partition?
  - What if the sender or the receiver process fail?
  - What if the sender doesn't send any message after the lost message?
  - When can a sender purge a message from its buffer?

Causal ordering

- Chatroom example
- Deliver messages in an order that is consistent with the happened before relation of the corresponding message send events
- If send(m_1) → send(m_2) then every process that delivers both m_1 and m_2 must deliver m_1 before m_2

<Figure>

Note: Messages may not be received in causal order, even if the underlying protocol delivers messages in a FIFO order

Causal Ordering: Implementation

- ISIS CBCAST protocol
  - Birman, K. and Joseph, T. “*Reliable Communications in the Presence of Failures*”. ACM TOCS, 5(1), 1987.
  - Birman ,K., Schiper A. and Stephenson, P. “Lightweight Causal and Atomic Group Multicast”. ACM TOCS , 9(3), 1991.
- CBCAST provides causal delivery of messages multicast with in a *group* of processes

Process Group

- A set of *n* members (processes)
- A single logical entity
  - Group id, send to group, receive from group, …
  - Example: replicated object
- Each group member can communicate with every other group member
- Together, group members perform a given functionality, e.g. support replication of objects

Group Communication Protocols

- A set of protocols designed to manage replication of objects
- Atomic broadcast protocol
  - A message sent by a group member is either delivered by every group member or none of the group members
  - Follow some delivery order
- Group membership protocol
  - A consistent view of which members are alive and which have failed
  - *Reliable* atomic broadcast protocol: A message sent by a group member is delivered by every group member
- Member recovery protocol

Group Middleware

- *<Figure>*

Causal Ordering: Implementation

- If send(m_1) $\rightarrow$ send(m_2) then every process that delivers both m_1 and m_2 must deliver m_1 before m_2
- When a message m is sent, we need to attach the ids of all messages m’ such that send(m’) $\rightarrow$ send(m)
  - The receiver will then deliver m only after it has delivered all m’ *s*

Recap …

- Group middleware
  - Support for replication
  - Atomic broadcast, group membership, recovery
- FIFO, Causal, Total ordering of messages
  - Send, receive and deliver events
  - Impose ordering on deliver events based on send events

Announcement

- No class on Thursday (02/09)
- Please read the following paper. We will discuss the ideas from this paper on Tuesday, 02/14

Y. Amir, L. Moser and P. Melliar-Smith. The totem single-ring ordering and membership protocol. ACM TOCS, 13(4), 1995.

ISIS: CBCAST

- Each group member *i* maintains a vector time VT_i
- All entries of VT_i are initialized to -1
- Before multicasting a message *m*, a member *i* increments $VT_i[i]$ and attaches a timestamp t_m to *m*

  $t_m = VT_i$

- On receiving *m* from *i*, a member *j* ($i \neq j$) delays the delivery of *m* until

  1) $VT_j[i] = t_m[i] - 1$ <u>and</u>

  2) $VT_j[k] \geq t_m[k]$, for all *k* in $\{0, 1, \ldots, n\text{-}1\} - \{i\}$

  – Delayed messages are buffered

– When *j* delivers *m*, it updates VT_j as follows

for all k, $VT_j[k] = \max(VT_j[k], t_m[k])$

- Example

CBCAST Observations

- Positive ack or negative ack techniques can be used in conjunction to recover from message losses
- Overhead
  - Each process has to maintain a vector
  - Send buffer and receiver buffer
  - Timer if positive ack is used
  - For every lost message, at least one retransmit message, if negative ack is used

Psync

- Reference: Peterson, L., Buchholz, N., Schlichting, R. "*Preserving and Using Context Information in Interprocess Communication*". ACM TOCS, 7, 1989.
- Psync provides causal delivery of messages multicast with in a group of processes
- Each group member maintains a context graph with nodes representing messages and edges representing causal ordering between messages
- All messages are assigned unique ids by the message senders

<sender id, seq#>

- Before multicasting a message *m*, a group member appends to *m* the messages ids of all leaves in its context graph
- The sender puts *m* in its context graph with an edge from every leaf to *m*
- On receiving *m*, a group member delays putting *m* in its context graph until all messages whose ids are attached to *m* are in its context graph
- A message is delivered as soon as it is put in the context graph

- *Example*

Psync: Observations

- Positive ack or negative ack technique can be used to recover lost messages
- Overhead
  - Context graph maintenance
  - Send buffer and receiver buffer
  - Timer if positive ack is used
  - For every lost message, at least one retransmit message, if negative ack is used

Causal ordering: Issues

- What if there is a network partition?
- What if the sender or the receiver process fail?
- What if the sender doesn't send any message after the lost message (negative ack)?
- When can a sender purge a message from its buffer?
- When can a message be purged from context graph (Psync)?

Total Order

- If one process delivers m_1 before m_2, then all processes that deliver both m_1 and m_2 must deliver m_1 before m_2.
- Messages multicast in a group are delivered in the same order at every group member
- Three types of algorithm
  - Sequencer based algorithms
  - Token based algorithms
  - History based algorithms
- Reference: F. Cristian, R. deBeijer, S. Mishra. "*A Performance Comparison of Asynchronous Atomic Broadcast Protocols*". IEE Distributed Systems Engineering, 1, 1994.

Total Ordering: Sequencer Based

- A single process is designated as a central coordinator
- ISIS ABCAST protocol

<figure>

Total Ordering: Token Based

- A token circulates among all all group members
  - Token carries a global sequence number

Reference: Y. Amir, L. Moser and P. Melliar-Smith. The totem single-ring ordering and membership protocol. ACM TOCS, 13(4), 1995.

Total Ordering: History Based

- Total order is constructed from causal order
  - Reference: Peterson, L., Buchholz, N., Schlichting, R. "*Preserving and Using Context Information in Interprocess Communication*". ACM TOCS, 7, 1989.

Network Partitions and Process Failures

- What should we do if a message is not received despite repeated retransmissions
  - Message sender failure, communication channel failure
  - Group membership protocol

- *Please read the following paper*

Y. Amir, L. Moser and P. Melliar-Smith. The totem single-ring ordering and membership protocol. ACM TOCS, 13(4), 1995.