CSE 486/586 Distributed Systems
Reliable Multicast --- 2

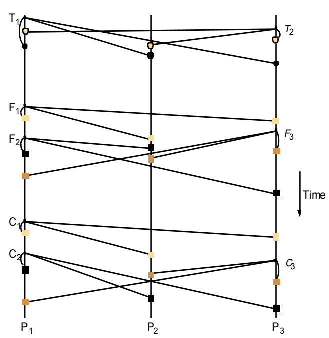
Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last Time

- How do a group of processes communicate?
- Multicast
  - One-to-many: “Local” broadcast within a group *g* of processes
- What are the issues?
  - Processes crash (we assume crash-stop)
  - Messages get delayed
- B-multicast
- R-Multicast
  - Properties: integrity, agreement, validity
- Ordering
  - Why do we care about ordering?

Recap: Ordering

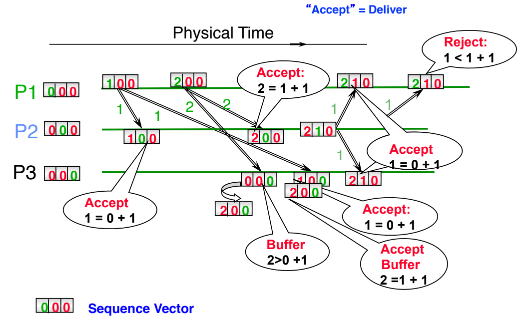
- Totally ordered messages T_1 and T_2.
- FIFO-related messages F_1 and F_2.
- Causally related messages C_1 and C_3
- Total ordering does not imply causal ordering.
- Causal ordering implies FIFO ordering
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.



Example: FIFO Multicast

*(do **NOT** be confused with vector timestamps)*

“Accept” = Deliver



Totally Ordered Multicast

- Using a sequencer
  - One dedicated “sequencer” that orders all messages
  - Everyone else follows.
- ISIS system
  - Similar to having a sequencer, but the responsibility is distributed to each sender.

Total Ordering Using a Sequencer

Sequencer = Leader process

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);   *i*: unique message id

On B-deliver(<*m, i*>) *with g = group*(*m*)
  Place <*m, i*> in hold-back queue;

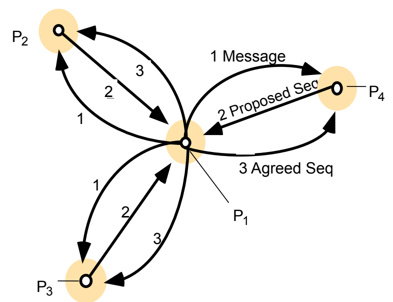
On B-deliver(m_{order} = <“order”, *i, S*>) *with g = group*(m_{order})
  wait until <*m, i*> in hold-back queue and $S = r_g$;
  TO-deliver m;   // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of *g*

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with g = group*(*m*)
  B-multicast(*g*, <“order”, *i*, s_g>);
  $s_g := s_g + 1$;

ISIS algorithm for total ordering



ISIS algorithm for total ordering

- Sender multicasts message to everyone
- Reply with proposed priority (sequence no.)
  - Larger than all observed *agreed* priorities
  - Larger than any previously proposed (by self) priority
- Store message in priority queue
  - Ordered by priority (proposed or agreed)
  - Mark message as undeliverable
- Sender chooses agreed priority, re-multicasts message with agreed priority
  - Maximum of all proposed priorities
- Upon receiving agreed (final) priority
  - Mark message as deliverable
  - Deliver any deliverable messages at the front of priority queue
- Notice any (small) issue?

CSE 486/586 Administrivia

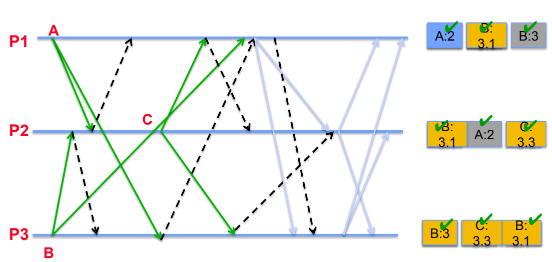
- Please start PA2 if you haven't.

Problematic Scenario

- Two processes P1 & P2 at their initial state.
- P1 sends M1 & P2 sends M2.
- P1 receives M1 (its own) and proposes 1. P2 does the same for M2.
- P2 receives M1 (P1's message) and proposes 2. P1 does the same for M2.
- P1 picks 2 for M1 & P2 also picks 2 for M2.
- Same sequence number for two different msgs.
- How do you want to solve this?

Example: ISIS algorithm

Showing the process id only when necessary



Proof of Total Order

- For a message m_1, consider the first process *p* that delivers m_1
- At *p*, when message m_1 is at head of priority queue and has been marked deliverable, let m_2 be another message that has not yet been delivered (i.e., is on the same queue or has not been seen yet by *p*)

  finalpriority(m_2) >= Due to "max" operation at sender

  proposedpriority(m_2) > Since queue ordered by increasing priority

  finalpriority(m_1)
- Suppose there is some other process *p'* that delivers m_2 before it delivers m_1. Then at *p'*,

  finalpriority(m_1) >= Due to "max" operation at sender

  proposedpriority(m_1) > Since queue ordered by increasing priority

  finalpriority(m_2)
- **a contradiction**!

Causally Ordered Multicast

- Each process keeps a vector clock.
  - Each counter represents the number of messages received from each of the other processes.
- When multicasting a message, the sender process increments its own counter and attaches its vector clock.
- Upon receiving a multicast message, the receiver process waits until it can preserve causal ordering:
  - It has delivered all the messages from the sender.
  - It has delivered all the messages that the sender had delivered before the multicast message.

Causal Ordering

Algorithm for group member p_i $(i = 1, 2 \ldots, N)$

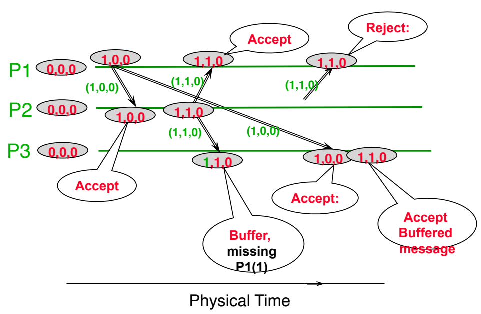
On initialization
$V_i^g[j] := 0 \; (j = 1, 2 \ldots, N);$

The number of group-g messages from process j that have been seen at process i so far

To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1;$
B-multicast$(g, \langle V_i^g, m \rangle);$

On B-deliver$(\langle V_j^g, m \rangle)$ *from* p_j, *with* $g = group(m)$
place $\langle V_j^g, m \rangle$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$;
CO-deliver m; // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1;$

Example: Causal Ordering Multicast



Summary

- Two multicast algorithms for total ordering
  - Sequencer
  - ISIS
- Multicast for causal ordering
  - Uses vector timestamps

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).