Lecture 7

Multicast

Reading: Sections 15.4
Communication Modes in Distributed System

- **Unicast** *(best effort or reliable)*
  - Messages are sent from exactly **one** process to **one** process.
  - *Best effort*: if a message is delivered it would be intact; no reliability guarantees.
  - *Reliable*: guarantees delivery of messages.

- **Broadcast**
  - Messages are sent from exactly **one** process to **all** processes on the network.
  - Broadcast protocols are not practical.

- **Multicast**
  - Messages broadcast within a **group** of processes.
  - A multicast message is sent from any **one** process to the group of processes on the network.
  - Reliable multicast can be implemented “above” (i.e., “using”) a reliable unicast.
  - This lecture!
Other Examples of Multicast Use

- Akamai’s Configuration Management System (called ACMS) uses a core group of 3-5 servers. These servers continuously multicast to each other the latest updates. They use **reliable multicast**. After an update is reliably multicast within this group, it is then sent out to all the (1000s of) servers Akamai has all over the world.

- Air Traffic Control System: orders by one ATC need to be **ordered** (and reliable) multicast out to other ATC’s.

- Newsgroup servers multicast to each other in a reliable and ordered manner.

- Facebook servers multicast your updates to each other
What’re we designing in this class

One process $p$

Incoming messages

Application (at process $p$)

MULTICAST PROTOCOL

send multicast

deliver multicast (upcall)
Basic Multicast (B-multicast)

- Let’s assume the all processes know the group membership
- A straightforward way to implement B-multicast is to use a reliable one-to-one send (unicast) operation:
  - B-multicast(group g, message m):
    for each process p in g, send (p,m).
  - receive(m): B-deliver(m) at p.
- A “correct” process = a “non-faulty” process
- A basic multicast primitive guarantees a correct process will eventually deliver the message, as long as the sender (multicasting process) does not crash.
  - Can we provide reliability even when the sender crashes (after it has sent the multicast)?
Reliable Multicast

- **Integrity**: A correct (i.e., non-faulty) process \( p \) delivers a message \( m \) at most once.
- **Validity**: If a correct process multicasts (sends) message \( m \), then it will eventually deliver \( m \) itself.
  - Guarantees *liveness* to the sender.
- **Agreement**: If some one correct process delivers message \( m \), then all other correct processes in \( \text{group}(m) \) will eventually deliver \( m \).
  - Property of “all or nothing.”
  - Validity and agreement together ensure overall liveness: if some correct process multicasts a message \( m \), then, all correct processes deliver \( m \) too.
**Reliable R-Multicast Algorithm**

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m);        // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)

if (m ∉ Received)

then

Received := Received ∪ {m};

if (q ≠ p) then B-multicast(g, m); end if

R-deliver m;

end if
Reliable Multicast Algorithm (R-multicast)

On initialization

Received := {};

For process p to R-multicast message m to group g
B-multicast(g, m); // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
if (m ∉ Received) Integrity
then

Received := Received ∪ {m};
if (q ≠ p) then B-multicast(g, m); end if Agreement
R-deliver m; Integrity, Validity

end if

if some correct process B-multicasts a message m, then, all correct processes R-deliver m too. If no correct process B-multicasts m, then no correct processes R-deliver m.
What about Multicast Ordering?

- **FIFO ordering**: If a correct process issues `multicast(g,m)` and then `multicast(g,m')`, then every correct process that delivers `m'` will have already delivered `m`.
- **Causal ordering**: If `multicast(g,m) → multicast(g,m')` then any correct process that delivers `m'` will have already delivered `m`.
- **Total ordering**: If a correct process delivers message `m` before `m'` (independent of the senders), then any other correct process that delivers `m'` will have already delivered `m`. 
Total, FIFO and Causal Ordering

- Totally ordered messages $T_1$ and $T_2$.
- FIFO-related messages $F_1$ and $F_2$.
- Causally related messages $C_1$ and $C_3$.

- Causal ordering implies FIFO ordering (why?)
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.
### Display From Newsgroup

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A.Hanlon</td>
<td>Mach</td>
</tr>
<tr>
<td>24</td>
<td>G.Joseph</td>
<td>Microkernels</td>
</tr>
<tr>
<td>25</td>
<td>A.Hanlon</td>
<td>Re: Microkernels</td>
</tr>
<tr>
<td>26</td>
<td>T.L’Heureux</td>
<td>RPC performance</td>
</tr>
<tr>
<td>27</td>
<td>M.Walker</td>
<td>Re: Mach</td>
</tr>
</tbody>
</table>

What is the most appropriate ordering for this application?  
(a) FIFO (b) causal (c) total

What is the most appropriate ordering for Facebook posts?
Providing Ordering Guarantees (FIFO)

- Look at messages from each process in the order they were sent:
  - Each process keeps a sequence number for each other process (vector)
  - When a message is received,
    - as expected (next sequence), accept
    - higher than expected, buffer in a queue
    - lower than expected, reject
Implementing FIFO Ordering

- $S^p_g$: the number of messages $p$ has sent to $g$.
- $R^q_g$: the sequence number of the latest group-$g$ message that $p$ has delivered from $q$ (maintained for all $q$ at $p$).
- For $p$ to FO-multicast $m$ to $g$:
  - $p$ increments $S^p_g$ by 1.
  - $p$ “piggy-backs” the value $S^p_g$ onto the message.
  - $p$ B-multicasts $m$ to $g$.
- At process $p$, Upon receipt of $m$ from $q$ with sequence number $S$:
  - $p$ checks whether $S = R^q_g + 1$. If so, $p$ FO-delivers $m$ and increments $R^q_g$.
  - If $S > R^q_g + 1$, $p$ places the message in the hold-back queue until the intervening messages have been delivered and $S = R^q_g + 1$.
  - If $S < R^q_g + 1$, reject $m$. 
Hold-back Queue for Arrived Multicast Messages
Example: FIFO Multicast

(do NOT confuse with vector timestamps)
"Accept" = Deliver

Physical Time

P1 0 0 0

1 0 0

2 0 0

Accept: 2 = 1 + 1

1

2

P2 0 0 0

1 0 0

2 0 0

Accept: 1 = 0 + 1

P3 0 0 0

0 0 0

1 0 0

2 1 0

Buffer 2 > 0 + 1

Accept 1 = 0 + 1

Accept Buffer 2 = 1 + 1

Reject: 1 < 1 + 1

Sequence Vector

0 0 0
Total Ordering Using a Sequencer

1. Algorithm for group member $p$

On initialization: $r_g := 0$;

To $TO$-multicast message $m$ to group $g$

$B$-multicast($g \cup \{ \text{sequencer}(g) \}$, $<m, i>$);

On $B$-deliver($<m, i>$) with $g = \text{group}(m)$

Place $<m, i>$ in hold-back queue;

On $B$-deliver($m_{\text{order}} = <\text{“order”}, i, S>$) with $g = \text{group}(m_{\text{order}})$

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 $g = \text{group}(m)$

$B$-multicast($g$, $<\text{“order”}, i, s_g>$);

$s_g := s_g + 1$;
**ISIS: Total ordering without sequencer**

A diagram illustrating the protocol's operation:

1. **1 Message**
2. **2 Proposed Seq**
3. **3 Agreed Seq**

Nodes labeled as P1, P2, P3, P4.
ISIS algorithm for total ordering

1. The multicast sender multicasts the message to everyone.

2. Recipients add the received message to a special queue called the *priority queue*, tag the message *undeliverable*, and reply to the sender with a *proposed priority* (i.e., proposed sequence number). Further, this proposed priority is 1 more than the latest sequence number heard so far at the recipient, suffixed with the recipient's process ID. The *priority queue is always sorted by priority*.

3. The sender collects all responses from the recipients, calculates their *maximum*, and re-multicasts original message with this as the *final priority* for the message.

4. On receipt of this information, recipients mark the message as *deliverable*, reorder the priority queue, and deliver the set of lowest priority messages that are marked as *deliverable*. 

Lecture 7-19
Proof of Total Order

• For a message m1, consider the first process p that delivers m1
• At p, when message m1 is at head of priority queue
• Suppose m2 is another message that has not yet been delivered (i.e., is on the same queue or has not been seen yet by p)
  \[ \text{finalpriority}(m2) \geq \text{proposedpriority}(m2) > \text{finalpriority}(m1) \]
  Due to “max” operation at sender
  and since proposed priorities by process p only increase
  Since queue ordered by increasing priority

• Suppose there is some other process p′ that delivers m2 before it delivers m1. Then at p′,
  \[ \text{finalpriority}(m1) \geq \text{proposedpriority}(m1) > \text{finalpriority}(m2) \]
  Due to “max” operation at sender
  Since queue ordered by increasing priority

a contradiction!
Causal Ordering using vector timestamps

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

On initialization

$$V^g_i[j] := 0 \ (j = 1, 2..., N);$$

To CO-multicast message $m$ to group $g$

$$V^g_i[i] := V^g_i[i] + 1;$$

$B$-multicast($g$, $<V^g_i, m>$);

On $B$-deliver($<V^g_j, m>$) from $p_j$, with $g = $ group($m$)

place $<V^g_j, m>$ in hold-back queue;

wait until $V^g_j[j] = V^g_i[j] + 1$ and $V^g_j[k] \leq V^g_i[k]$ ($k \neq j$);

$CO$-deliver $m$; // after removing it from the hold-back queue

$$V^g_i[j] := V^g_i[j] + 1;$$

The number of group-$g$ messages from process $j$ that have been seen at process $i$ so far
Example: Causal Ordering Multicast

Physical Time

0,0,0
(1,0,0)
(1,1,0)
(1,1,0)
(1,0,0)
1,1,0
1,1,0

P1
P2
P3

Accept
Reject:
Accept
Accept
Buffer, missing P1(1)
Accept Buffered message
Multicast is operation of sending one message to multiple processes in a given group

- Reliable multicast algorithm built using unicast
- Ordering – FIFO, total, causal

Thursday

- RPCs: Section 4.3, parts of Chapter 5
  - Important for MP2
- **Homework 1 due this Thursday**
  - Hand in to me at start of lecture (not during or after lecture)