Multicast
Key Properties

Multiple computers
- Concurrent execution
- Independent failures
- Autonomous administrators
- Heterogeneous capacities, properties
- Large numbers (scalability)

Networked communication
- Asynchronous execution
- Unreliable delivery
- Insecure medium

Common goal
- Consistency – can discuss whole-system properties
- Transparency – can use the system without knowing details
Communication Modes in DS

Unicast
- One-to-one: Message from process $p$ to process $q$.
- Best effort: message *may* be delivered, but will be intact
- Reliable: message *will* be delivered

Broadcast
- One-to-all: Message from process $p$ to *all* processes
- Impractical for large networks

Multicast
- One-to-many: “Local” broadcast within a group $g$ of processes
Objectives

Define multicast properties
  ◦ Reliability
  ◦ Ordering

Examine algorithms for reliable and/or ordered multicast

Readings:
  ◦ 12.4 (4th ed), 15.4 (5th ed)
  ◦ Optional: 4.5 (4th ed), 4.4 (5th ed)
What’re we designing in this class

One process $p$

$send(m, g)$

$deliver(m)$ (metadata)

Incoming messages

Application (at process $p$)

send multicast

deliver multicast

MULTICAST PROTOCOL
Basic Multicast (B-multicast)

A straightforward way to implement B-multicast is to use a reliable one-to-one send (unicast) operation:

- B-multicast($g, m$): for each process $p$ in $g$, send($p, m$).
- receive($m$): B-deliver($m$) at $p$.

Guarantees?

- All processes in $g$ eventually receive every multicast message...
- ... as long as send is reliable
- ... and no process crashes
Reliable Multicast

**Integrity:** A *correct* (i.e., non-faulty) process $p$ delivers a message $m$ at most once.

**Agreement:** If a *correct* process delivers message $m$, then all the other *correct* processes in group($m$) will *eventually* deliver $m$.
  - Property of “all or nothing.”

**Validity:** If a *correct* process multicasts (sends) message $m$, then it will *eventually* deliver $m$ itself.
  - Guarantees liveness to the sender.

Validity and agreement together ensure overall liveness: if some correct process multicasts a message $m$, then, all correct processes deliver $m$ too.

*Assumption:* no process sends exactly the same message twice
R- send : for each \( p \in G \), send \( (m, p) \)

when receive \( (m) \):
  for all \( p \in G \), ask if \( m \) received
  if all say "yes" then deliver \( m \)
  if any say "no" then drop \( m \)

If no response:
  - "Got? "Attack at Dawn"
  - "Yes"
Reliable R-Multicast Algorithm

On initialization

Received := \{\};

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

B-multicast(g,m); \((\text{for } p \in \text{g is included as destination)}\)

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

if (m \notin \text{Received}):

Received := Received \cup \{m\};

if (q \neq p):

B-multicast(g,m);

R-deliver(m)
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 destination)

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

if (m ∉ Received):

Revised := Revised ∪ {m};

if (q ≠ p):

B-multicast(g,m);

R-deliver(m)
Ordered Multicast

**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\((g, m) \rightarrow multicast(g, m')\) then any correct process that delivers \(m'\) will have already delivered \(m\).

- Note that \(\rightarrow\) counts messages **delivered** to the application, rather than all network messages.

**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 \(m\).
Call Bob: "I just posted that our cancelled!"

A

Post

FB

2

Alice said class is cancelled.

B

X

See Alice's post
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.
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.
FIFO Ordering

FIFO: each process delivers
- 1 before 2
- 3 before 4

Order of messages
- P1: 1,2,3,4
- P2: 3,1,2,4
- P3: 3,4,1,2

Not causal:
- multicast(1) -> multicast(4)
Causal ordering

Each process delivers
- 1 before 2
- 3 before 4
- 1 before 4

Order of messages
- P1: 1,2,3,4
- P2: 3,1,2,4
- P3: 3, 1,4,2

Also FIFO (always)
Total ordering

Each process **delivers**
- 2, 3, 1, 4

Not FIFO
- So not causal either
### Display From Bulletin Board Program

<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

A: let’s see a movie
B: no, let’s rob a bank
C: something good to me!
D: hi, I want a movie!!!
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.
- When a message is received, if message # is:
  - as expected (next sequence), accept
  - higher than expected, buffer in a queue
  - lower than expected, reject
Implementing FIFO Ordering

\( p.S \) – number of messages process \( p \) has sent

\( p.R[q] \) – sequence number of latest message \( p \) has delivered from process \( q \).

\( p.Q[q] \) – hold-back queue for messages
- All state initialized to 0 (empty for Q)
- All state specific to a group \( g \)

**FIFO-multicast(m,g) @ process p:**
- Increment sequence number: \( p.S \leftarrow p.S + 1 \)
- Piggy back sequence number with message: \( \text{B-multicast}((m, p.S), g) \)

On **B-deliver** of message \( m \) from \( q \) with sequence# \( S \) @ process \( p \)
- If \( S = p.R[q]+1 \)
  - **FIFO-deliver(m)**
  - **Check-holdback(q)**
- Else if \( S > p.R[q]+1 \)
  - Add \( (m, S) \) to hold-back queue \( p.Q[q] \)

After delivering one message, check for other message that can now be delivered:
- **Check-holdback(q):**
  - While \( p.Q[q].head.S = p.R[q] + 1 \)
  - **FIFO-deliver(m)**
  - \( p.R[q] \leftarrow p.R[q] + 1 \)
Hold-back Queue for Arrived Multicast Messages

Incoming messages → Hold-back queue → Delivery queue → Message processing → deliver

When delivery guarantees are met
Example: FIFO Multicast

Physical Time

P1 100 200 210
   1 1 2
   2 2

P2 100 200 210
   1
   1

P3 000 000 200
   1
   1

Accept: 2 = 1 + 1
Accept: 1 = 0 + 1
Accept: 1 = 0 + 1
Accept from Buffer 2 = 1 + 1
Reject: 1 < 1 + 1
Buffer > 0 + 1

Sequence Vector
(Do NOT confuse with vector timestamps)
“Accept” = Deliver

NIKITA BORISOV - UIUC 2019-01-29
Total Ordering Using a Sequencer

1. Algorithm for group member \( p \)

On initialization: \( r_g := 0 \);

To TO-multicast message \( m \) to group \( g \)

\[ B\text{-multicast}(g \cup \{ \text{sequencer}(g) \}, \langle m, i \rangle); \]

On \( B\text{-deliver}(\langle m, i \rangle) \) with \( g = \text{group}(m) \)

Place \( \langle m, i \rangle \) in hold-back queue;

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

wait until \( \langle m, i \rangle \) 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 \( B\text{-deliver}(\langle m, i \rangle) \) with \( g = \text{group}(m) \)

\[ B\text{-multicast}(g, \langle \text{“order”}, i, s_g \rangle); \]

\[ s_g := s_g + 1; \]
ISIS algorithm for total ordering

1 Message

1 Proposed Seq

3 Agreed Seq

P_1

P_2

P_3

P_4
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 front of priority queue
Example: ISIS algorithm
Collisions

Problem: priority queue requires unique priorities

Solution: add process # to suggested priority
  ◦ i.e., 3.2 == process 2 proposed priority 3

Compare on priority first, use process # to break ties
  ◦ 3.2 > 3.1
  ◦ 2.1 > 1.3
Example: ISIS algorithm
Proof of Total Order

Consider two messages, \( m_1 \) and \( m_2 \), and two processes, \( p \), and \( p' \)

Suppose that \( p \) delivers \( m_1 \) before \( m_2 \)

When \( p \) delivers \( m_1 \), it is at the head of the queue. \( m_2 \) is either:
- Already in \( p \)’s queue, and deliverable, so
  - \( \text{finalpriority}(m_1) < \text{finalpriority}(m_2) \)
- Already in \( p \)’s queue, and not deliverable, so
  - \( \text{finalpriority}(m_1) < \text{proposedpriority}(m_2) \leq \text{finalpriority}(m_2) \)
- Not yet in \( p \)’s queue: same as above, since proposed priority > any delivered message

Suppose \( p' \) delivers \( m_2 \) before \( m_1 \), by the same argument:
- \( \text{finalpriority}(m_2) < \text{finalpriority}(m_1) \)
- 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

Buffer, missing P1(1)

Accept

Reject:

Accept
Buffered message

Accept

A \rightarrow C

(1,1,0)

(0,0,0)

(1,3,0)

(1,2,0)

(1,1,0)

(1,1,0)

P3

P2

P1
Summary

Multicast is operation of sending one message to multiple processes in a given group.

Reliable multicast algorithm built using unicast.

Ordering – FIFO, total, causal.