Multicast

CS425/ECE428– DISTRIBUTED SYSTEMS – FALL 2021
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 $p$ ->

Application (at process $p$)

send multicast

deliver multicast

MULTICAST PROTOCOL

Incoming messages
class B_multicast:
    def __init__(self, group, my_id, unicast, deliver):
        self.group = group
        self.id = my_id
        self.unicast = unicast
        self.deliver = deliver
        unicast.register_deliver(self.unicast_deliver)

    def send(self, message):
        for member in self.group:
            unicast.send(message, member)

    def unicast_deliver(self, message):
        self.deliver(message)
Basic Multicast (B-multicast)

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

- $\text{B-multicast}(g,m)$: for each process $p$ in $g$, send($p,m$).
- $\text{receive}(m)$: 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
Reliable R-Multicast Algorithm

On initialization

\[
\text{Received} := \{\}\;
\]

For process \( p \) to R-multicast message \( m \) to group \( g \)

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

On B-deliver(\( m \)) at process \( q \) with \( g = \text{group}(m) \)

\[
\text{if (} m \not\in \text{Received)}:
\]

\[
\text{Received} := \text{Received} \cup \{m\}; \\
\text{if (} q \neq p) :
\]

\[
\text{B-multicast}(g,m); \\
\text{R-deliver}(m)
\]
R-multicast

class R_multicast:
    def __init__(self, group, my_id,
                 unicast, deliver):
        self.group = group
        self.id = my_id
        self.b_multicast = B_multicast(group,
                                        my_id, unicast, self.b_multicast_deliver)
        self.deliver = deliver
        self.messages = set()

    def send(self, message):
        B_multicast.send(message)

    def b_multicast_deliver(self, message):
        if message not in self.messages:
            self.messages.add(message)
            if message.sender != self.id:
                B_multicast(message)
        self.deliver(message)
Reliable R-Multicast Algorithm

On initialization

\[ \text{Received} := \{}; \]

For process \( p \) to R-multicast message \( m \) to group \( g \)

\[ \text{B-multicast}(g,m); \]

\((p \in g \text{ is included as destination})\)

On B-deliver\((m)\) at process \( q \) with \( g = \text{group}(m) \)

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

\[ \text{Integrity} \]

\[ \text{Received} := \text{Received} \cup \{m\}; \]

\[ \text{if } (q \neq p): \]

\[ \text{Agreement} \]

\[ \text{B-multicast}(g,m); \]

\[ \text{R-deliver}(m) \]
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 $m$. 
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: 1,3,2,4
- 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: 1,3,2,4
- P3: 3,1,4,2

Also FIFO (always)
Total ordering

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

Not FIFO
- So not causal either
<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
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: **B-multicast**((m, \( p.S \)), \( g \))

**On B-deliver** of message \( m \) from \( q \) with sequence\# \( S \) @ process \( p \)
  - If \( S = p.R[q]+1 \)
    - \( p.R[q] \leftarrow p.R[q]+1 \)
    - **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)**
    - \( p.R[q] \leftarrow p.R[q] + 1 \)
class FIFO_multicast:
    def __init__(self, multicast, deliver):
        self.multicast = multicast
        # B- or R-multicast
        multicast.deliver = self.mcast_deliver
        self.seq_no = 1
        # note: expect_seq[MY_ID] is unused
        self.expected_seq = { m : 1
                              for m in multicast.group.members }
        self.buffer = {}
        self.deliver = deliver
    def send(self, msg):
        multicast.send({ 'message': msg,
                         'seq' : my_seq_no })
        my_seq_no += 1
    def mcast_deliver(self, m):
        if m['seq'] == self.expected_seq[m.sender]:
            deliver(m['message'])
            self.expected_seq[m.sender] += 1
            self.check_buffer(m.sender)
        else:
            buffer[(m.sender, m['seq'])] = m
    def check_buffer(self, sender):
        while (sender, self.expected_seq[sender]) in self.buffer:
            self.deliver(self.buffer.pop((sender,
                                           self.expected_seq[sender]))))
            expected_seq[sender] += 1
Hold-back Queue for Arrived Multicast Messages

Message processing

deliver

Hold-back queue

Incoming messages

Delivery queue

When delivery guarantees are met
Example: FIFO Multicast

Physical Time

P1 000

P2 000

P3 000

Sequence Vector

(Do NOT confuse with vector timestamps)

“Accept” = Deliver

Accept: 1 = 0 + 1

Accept: 2 = 1 + 1

Accept: 1 = 0 + 1

Accept: 1 = 0 + 1

Buffer > 0 + 1

Accept from Buffer 2 = 1 + 1
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($m_{\text{order}} = \langle \text{"order"}, i, S\rangle$) 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($g, \langle \text{"order"}, i, s_g\rangle$);
   $s_g := s_g + 1$;
**Normal Process**

```python
class SequencerClient:
    def __init__(self, multicast, deliver):
        self.multicast = multicast
        self.deliver = deliver
        multicast.deliver = self.mcast_deliver
        self.expected_seq = 1
        self.unordered = {}
        self.ordered = {}
        self.order_msgs = {}

    def send(self, msg):
        # unique identifier
        msg_id = hash(msg, self.multicast.id)
        self.multicast.send({ 'type': 'message',
                             'message': msg, 'id': msg_id })

    def mcast_deliver(self, m):
        if m['type'] == 'message':
            self.unordered[m['msg_id']] = m
        else:  # m['type'] == 'order'
            self.order_msgs[m['msg_id']] = m['seq']

# apply order messages
for msg_id, seq in self.order_msgs.items():
    if msg_id in self.unordered:
        self.ordered[seq] = self.unordered.pop(msg_id)
        self.order_msgs.pop(msg_id)

# deliver messages
while self.expected_seq in self.ordered:
    self.deliver(self.ordered.pop(self.expected_seq))
    self.expected_seq += 1
```

```python
class Sequencer:
    def __init__(self, multicast):
        self.multicast = multicast
        self.deliver = deliver
        multicast.deliver = self.mcast_deliver
        self.seq = 1

    def mcast_deliver(self, m):
        if m['type'] == 'message':
            self.multicast({ 'type': 'order',
                             'msg_id': m['msg_id'],
                             'seq': self.seq })
            self.seq += 1
```
```
ISIS algorithm for total ordering

1 Message
2 Proposed Seq
3 Agreed Seq
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
  - \( \text{finalprior}(m_1) < \text{finalprior}(m_2) \)
- Already in \( p \)'s queue, and not deliverable, so
  - \( \text{finalprior}(m_1) < \text{proposedprior}(m_2) \leq \text{finalprior}(m_2) \)
- 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:

- \( \text{finalprior}(m_2) < \text{finalprior}(m_1) \)
- Contradiction!
Causal Ordering using vector timestamps

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

---

On initialization

$V_i^g[j] := 0$ ($j = 1, 2..., N$);

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

$V_i^g[i] := V_i^g[i] + 1$;

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

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

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

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

$V_i^g[j] := V_i^g[j] + 1$;

---

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

P1 0,0,0 1,0,0 (1,1,0) 1,1,0
P2 0,0,0 1,0,0 (1,1,0) (1,1,0)
P3 0,0,0 1,1,0 (1,1,0) (1,1,0)

Accept: 1,0,0
Accept: 1,1,0
Reject: 1,1,0
Accept: (1,0,0)
Accept: (1,0,0)

Buffer, missing P1(1)
Accept Buffered message

Accept
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.