Reliable multicast

- Multicast – group communication
- States or replicas should be a deterministic function of their initial states and the sequence of operations applied to them.
- Message should be delivered to all members in a process group.
- Latency and scalability?

Basic Reliable-Multicasting Schemes

- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail
  - a) Message transmission
  - b) Reporting feedback

Nonhierarchical Feedback Control

- Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.
Hierarchical Feedback Control

- The essence of hierarchical reliable multicasting.
  a) Each local coordinator forwards the message to its children.
  b) A local coordinator handles retransmission requests.

Atomic Multicast (Layered Architecture)

- Figure 8-12. The logical organization of a distributed system to distinguish between message receipt and message delivery.

Virtual Synchrony

- Figure 8-13. The principle of virtual synchronous multicast.

Ordered multicast

- Unordered multicasts
- FIFO ordering
  o Sender ordered
- Causal ordering
  o Happens-before
- Total ordering
- Assume no overlapping groups
Message Ordering (1)

Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sends m1</td>
<td>receives m1</td>
<td>receives m2</td>
</tr>
<tr>
<td>sends m2</td>
<td>receives m2</td>
<td>receives m1</td>
</tr>
</tbody>
</table>

Message Ordering (2)

Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
<th>Process P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sends m1</td>
<td>receives m1</td>
<td>receives m3</td>
<td>sends m3</td>
</tr>
<tr>
<td>sends m2</td>
<td>receives m3</td>
<td>receives m1</td>
<td>sends m4</td>
</tr>
<tr>
<td>receives m2</td>
<td>receives m2</td>
<td>receives m4</td>
<td></td>
</tr>
<tr>
<td>receives m4</td>
<td>receives m4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIFO-ordered Multicast

- *Fifo or sender ordered multicast:* Messages are delivered in the order they were sent (by any single sender)

FIFO-ordered multicast

- *Fifo or sender ordered multicast* Messages are delivered in the order they were sent (by any single sender)

delivery of c to p is delayed until after b is delivered
Implementing FIFO order

- Basic reliable multicast algorithm has this property
  - Without failures all we need is to run it on FIFO channels (like TCP)
  - With failures need to be careful about the order in which things are done but problem is simple

Causally-ordered multicast

- Causal or happens-before ordering:
  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations
  - delivery of c to p is delayed until after b is delivered
  - e is sent (causally) after b
Causally-ordered multicast

- **Causal or happens-before ordering:**
  
  \[ \text{If } \text{send}(a) \rightarrow \text{send}(b) \text{ then } \text{deliver}(a) \text{ occurs before } \text{deliver}(b) \text{ at common destinations} \]

Implementing causal order

- Start with a FIFO multicast
- Frank Schmuck showed that we can always strengthen this into a causal multicast by adding vector time (no additional messages needed)
  - If group membership were static this is easily done, small overhead
  - With dynamic membership, at least abstractly, we need to identify each VT index with the corresponding process.

Observations

- These two orderings are for **asynchronous**:
  - Sender doesn’t get blocked and can deliver a copy to itself without “stopping” to learn a safe delivery order
  - If used this way, the multicast can seem to sit in the output buffers a long time, leading to surprising behavior
  - But this also gives the system a chance to concatenate multiple small messages into one larger one.
- Sometimes, we want a replicated object or service that advances through a series of transitions in the same order
  - Clearly will need all copies to make the same transitions
  - Leads to a need for totally ordered multicast

Totally-ordered multicast

- **Total or locally total multicast:**
  
  Messages are delivered in same order to all recipients (including the sender)
Totally-ordered multicast

- Can visualize as “closely synchronous”
- Real delivery is less synchronous, as on the previous slide

Implementing Total Order

- Many ways have been proposed
  - Centralized sequencer
  - Just have a token that moves around
    - Token has a sequence number
    - When you hold the token you can send the next burst of multicasts

What about membership changes?

- Virtual synchrony model synchronizes membership change with multicasts
- Idea is that:
  - Between any pair of successive group membership views...
  - ... same set of multicasts are delivered to all members
- If you implement distributed code, this makes algorithms much simpler for you!

Process groups with joins, failures

G₀={p,q}          G₁={p,q,r,s}           G₂={q,r,s}           G₃={q,r,s,t}

r, s request to join
r, s added; state xfer

q

p fails

r requests to join
p

r added, state xfer
Asynchrony

- Notice that FIFO-order and causally-order can be used asynchronously, while total-order always "stutters"
  - Insight is that the first two can always be delivered to the sender at the time the multicast is sent
  - Total-order delivery ordering usually isn’t known until a round of message exchange has been completed
- Results in a tremendous performance difference
  - With asynchrony, we gain concurrency at the sender side, but this helps mostly if remainder of group is idle or doing a non-conflicting task
  - Too much asynchrony
    - Means things pile up in output buffers
    - If a failure occurs, much is lost
    - And we could consume a lot of sender-side buffering space