Reliable Multicasting in the Presence of Faulty Processes

• Reliable multicasting in the presence of process failures can be accurately defined in terms of process groups and changes to group membership

• Each multicast message \( m \) is associated with a list of processes to which it should be delivered; this list is called a **group view**
  – A group view is the view of the process group when \( m \) was sent
  – Each process on the list should have the same view

• A **view change** takes place by multicasting a message \( v_c \) announcing the joining or leaving of a process
  – A process leaves a group when it crashes
  – A process joins a group when it recovers

---

Distinction between message receipt and message delivery.
Virtual Synchrony (1)

• Suppose message $m$ is multicast when its sender has group view $G$ and while the multicast is taking place, another process joins or leaves the group $\rightarrow$ Two multicast messages are simultaneously in transit: $m$ and $vc$

• We need to guarantee that
  
  – Message $m$ is either delivered to all processes in $G$ before $vc$ is delivered or $m$ is not delivered at all
  
  – There is only one case in which delivery of $m$ is allowed to fail: Group membership change is due to the sender of $m$ crashing

• Reliable multicasts with the above property are said to be virtually synchronous
Virtual Synchrony (2)

• In virtually synchronous reliable multicast, all multicasts take place between view changes → all multicasts that are in transit while a view change takes place are completed before the view change comes into effect.

![Diagram of multicast communication between processes P1, P2, P3, and P4 illustrating reliable multicast by multiple point-to-point messages and partial and total group membership.

Ordering of Multicasts

• Different orderings of multicast
  – **Unordered multicast**: no guarantees are given concerning the order in which received messages are delivered by different processes.
  – **FIFO-ordered multicast**: messages from the same sender are delivered in the same order as they were sent.
  – **Causally-ordered multicast**: all receivers must have causally related messages delivered in the happened-before order.
    - Can be implemented using vector clocks.
  – **Totally-ordered multicast**: all messages are delivered in the same order at all group members.
    - Can be implemented using logical clocks or a centralized sequencer.

• **Atomic multicast** is virtually synchronous reliable multicast offering totally-ordered delivery of messages.
  - Atomic multicast can be causally-ordered or FIFO-ordered.
<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>

Unordered multicast

<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></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 m2</td>
<td></td>
</tr>
<tr>
<td>receives m4</td>
<td>receives m4</td>
<td>receives m4</td>
<td></td>
</tr>
</tbody>
</table>

FIFO-ordered multicast

<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 m1</td>
<td>sends m3</td>
</tr>
<tr>
<td>sends m2</td>
<td>receives m3</td>
<td>receives m3</td>
<td>sends m4</td>
</tr>
<tr>
<td>receives m2</td>
<td>receives m2</td>
<td>receives m2</td>
<td></td>
</tr>
<tr>
<td>receives m4</td>
<td>receives m4</td>
<td>receives m4</td>
<td></td>
</tr>
</tbody>
</table>

FIFO atomic multicast

---

**Implementing Virtual Synchrony (1)**

- Multicasting a message $m$ to a group of processes is implemented by reliably sending $m$ to each group member using TCP
  - The sender may fail before having transmitted $m$ to each member
  - Messages from the same source are received in the order they were sent
- The main problem is to guarantee that all messages sent to view $G$ are delivered to all nonfaulty processes in $G$ before the next group membership change takes place
Implementing Virtual Synchrony (2)

- Let \( m \) be a message sent to view \( G_i \); \( m \) is said to be **stable** if it has been received by all members in \( G_i \).
- A process can install view \( G_{i+1} \) only when all messages sent to view \( G_i \) are stable.
- Assume the current view is \( G_i \), and it’s necessary to install the next view \( G_{i+1} \):
  - When a process \( P \) receives a view-change message for \( G_{i+1} \), it forwards a copy of any unstable message to every process in \( G_i \) and then marks it as being stable.
  - Next, \( P \) multicasts a flush message for \( G_{i+1} \) to indicate that it no longer has any unstable messages.
  - After \( P \) has received a flush message for \( G_{i+1} \) from each other process, it installs the new view.
- When process \( Q \), still operating in view \( G_i \), receives a message \( m \):
  - If it has already received \( m \), it discards it as a duplicate.
  - Else it delivers \( m \) (using message ordering constraints as necessary).

Implementing Virtual Synchrony (3)

(a) Process 4 notices that process 7 has crashed, sends a view change.
(b) Process 6 sends out all its unstable messages, followed by a flush message.
(c) Process 6 installs the new view when it has received a flush message from everyone else.
Error Recovery

• An error is a part of a system’s state that may lead to a failure
  – E.g., packet corrupted during transmission
• Error recovery is to replace an erroneous state with an error-free state
  – Forward recovery: bring the system into a correct new state from which it can continue to execute
    • E.g., use error correction code to correct bit errors in a packet
    • Need to know which errors may occur and correct errors in the system’s state → impossible in many cases
  – Backward recovery: bring the system back to a previous error-free state
    • E.g., retransmit a corrupted packet
    • Need to record the system state from time to time and restore such a recorded state when an error occurs
    • Widely used in distributed systems

Checkpointing

• In backward recovery, each process periodically saves its state on stable storage; this is called checkpointing
  – High checkpointing frequency increases the overhead
  – Low checkpointing frequency increases the recovery cost in terms of lost computation
• We need to record a consistent global state: if a process P has recorded the receipt of a message M, then there should be a process Q that has recorded the sending of M
• To recover after a failure, roll back to a recovery line, which is the most recent collection of checkpoints that form a consistent global state
  – If processes use the distributed snapshot algorithm to coordinate checkpointing, then the latest snapshot form a recovery line