Week 4: Part 1
Group Communication
Modes of communication

• One-to-One
  – Unicast
    • 1↔1
    • Point-to-point
  – Anycast
    • 1→nearest 1 of several identical nodes
    • Introduced with IPv6; used with BGP routing protocol

• One-to-many
  – Broadcast
    • 1→all
  – Multicast
    • 1→many = group communication
Groups allow us to deal with a collection of processes as one abstraction.

Send a message to one entity
- Deliver to the entire group

Groups are dynamic
- Created and destroyed
- Processes can join or leave
  - May belong to 0 or more groups

**Primitives:**
- `create_group*`
- `delete_group*`
- `join_group`
- `leave_group`
- `send_to_group`
- `query_membership*`

*Optional
Design Issues

• Closed vs. Open
  – Closed: only group members can send messages

• Peer vs. Hierarchical
  – Peer: each member communicates with the entire group
  – Hierarchical: go through coordinator(s), which relay messages to the group
    • Root coordinator: forwards the message to appropriate subgroup coordinators

• Managing membership & group creation/deletion
  – Distributed vs. centralized

• Leaving & joining — must be synchronous

• Fault tolerance & message order
  – Do we need reliable message delivery? What about missing or unreachable group members?
  – Do messages need to be received in the order they were sent?
The same things bite us with unicast communication

- **Crash failure**
  - Process stops communicating

- **Omission failure** (typically due to network)
  - Send omission: A process fails to send messages
  - Receive omission: A process fails to receive messages

- **Byzantine failure**
  - Some messages are faulty

- **Partitions**
  - The network may get segmented, dividing the group into two or more unreachable sub-groups
  - Some group members may not get the message
Failure considerations

The same things bite us with unicast communication … with extra problems

• Client dies before the multicast is complete
  – A set of group members might not get the message

• Server dies during a multicast
  – It may not receive the message while other group members do
  – Receive omission: A process fails to receive messages

• A member leaves or joins a group during a multicast
  – Will it get the message?
Implementing Group Communication Mechanisms
Hardware multicast

If we have hardware support for multicast

- Group members listen on the MAC address

send $addr = m_1$

listen $multicast addr = m_1$
Diffusion group: broadcast to all clients & then filter
- Software filters incoming broadcast or multicast address
- May need to use auxiliary group ID to identify the group (not in the network address header)
Hardware multicast & broadcast

- Ethernet & Wi-Fi support both multicast & broadcast
- Limited to local area networks
Software implementation: multiple unicasts

Sender knows group members

```
listen local addr = a_2
send(a_2)
```

```
listen local addr = a_3
send(a_3)
```

```
listen local addr = a_5
send(a_5)
```
Multiple unicasts via group coordinator

- Coordinator knows group members
- Coordinator iterates through group members
- May support a hierarchy of coordinators

```
listen local addr = a

listen local addr = a

listen local addr = a

send(a_2)

send(a_3)

send(a_5)

coordinator
```

send(c)
Publish-Subscribe (Pub/Sub)

Communication pattern – one of several for group communication

• Publishers & subscribers
  – Publishers: send messages — typically to a topic
  – Subscribers: receive messages that match certain attributes (topics)

• Message broker – service that filters, routes, & queues messages
  (also known as a message bus or event bus)
Publish-Subscribe (Pub/Sub)

The message broker is a service that is responsible for

- Message queuing
- Filtering
- Reliability (of itself and, in some cases, dealing with dead subscribers)
- Delivery guarantees and message ordering
- Scaling to handle message volume and clients
Reliability of multicasts
Unreliable multicast (best effort)

• Basic multicast
• Hope it gets to all the members
• Best-effort delivery
  – The system (computers & network) tries to deliver messages to their destinations but does not retransmit corrupted or lost data
Reliable multicast

• All non-faulty group members will receive the message
  – Assume sender & recipients will remain alive
  – Network may have glitches
    • Try to retransmit undelivered messages … but eventually give up
  – It’s OK if some group members don’t get the message

• Acknowledgements
  – Send message to each group member
  – Wait for acknowledgement from each group member
  – Retransmit to non-responding members
  – Subject to feedback implosion in group communication
    • Feedback implosion = a system sends one message but gets many back in response.
      E.g., send a message to a group of 1,000 members and get back 1,000 acknowledgements.
Optimizing Acknowledgements

- Easiest thing is to wait for an ACK before sending the next message
  - But that incurs a round-trip delay

- Optimizations
  - **Pipelining**
    - Send multiple messages – receive ACKs asynchronously
    - Set timeout – retransmit message for missing ACKs
  
  - **Cumulative ACKs**
    - Wait a little while before sending an ACK
    - If you receive other messages, then send one ACK for everything
  
  - **Piggybacked ACKs**
    - Send an ACK along with a return message

  - **Negative ACKs**
    - Receiver requests retransmission of a missed message

TCP (not multicast) does the first three of these … but with groups we must do this for each recipient
How does a receiver know it missed a message?
- Sender can attach a sequence # to each message

**Negative acknowledgment** – sent by a receiver if it misses a sequence #
- Sender must keep a buffer of old messages (possibly forever)
  - Realistically, keep either a fixed-size buffer or have a time limit
- Need to account for the receiver not sending a negative ACK because it is dead
  - E.g., Send periodic *are-you-alive* messages to check that receivers are alive

**Scalable Reliable Multicasting: feedback suppression**
- Send only negative acknowledgments
  - But multicast them – that way, other receivers will not send a NACK for the same message
  - Use a small random delay before sending the NACK to avoid lots of feedback msgs
  - Every group member is interrupted with NACK messages
Hierarchical feedback control

- Another technique for avoiding feedback implosion

- Partition group into subgroups, organized into a tree

- Sender is in the root of the tree (or sends to the root)
  - Each subgroup has a local coordinator – responsible for retransmissions within the subgroup
Atomic multicast

Atomicity – “all or nothing” property
A message sent to a group arrives at all group members.
If it fails to arrive at any member, no member will process it.

Problems

- Unreliable network
  - Each message should be acknowledged
  - Acknowledgements can be lost
- Recipient might die
- Message sender might die
Achieving atomicity

• General idea
  – Ensure that every recipient acknowledges receipt of the message
  – Only then allow the application to process the message
  – If we give up on a recipient then no recipient can process that received message

• Easier said than done!
  – What if a recipient dies after acknowledging the message?
    • Is it obligated to restart?
    • If it restarts, will it know to process the message?
  – What if the sender (or coordinator) dies partway through the protocol?
Achieving atomicity – example 1

Retry through network failures & system downtime
• Sender & receivers maintain a **persistent log**
• Each message has a unique ID so we can discard duplicates

**Sender**
- Write message to log
- Send message to all group members
- Wait for acknowledgement from each group member
- Write acknowledgement to log
- If timeout on waiting for an acknowledgement, retransmit to group member

**Receiver**
- Log received non-duplicate message to persistent log
- Send acknowledgement

**NEVER GIVE UP!**

Assume that dead senders or receivers will be rebooted and can restart where they left off
Redefine the group

- If some members failed to receive the message:
  - Remove the failed members from the group
  - Then allow existing members to process the message

- But still need to account for the death of the sender
  - Surviving group members may need to take over to ensure all current group members receive the message

- This is the approach used in virtual synchrony
Message ordering
Consistent (Good) Ordering

Single sender multicasting a stream of messages

message $a$
message $b$

order received

$a, b$

$\text{order received}$
Inconsistent (Bad) Ordering

Single sender multicasting a stream of messages

- Message a
- Message b

Order received:
- a, b
- b, a
Multiple senders multicasting a stream of messages

Consistent (Good) Ordering

- Process 0
  - Message: a
  - Order received: a, b

- Process 1
  - Message: b
  - Order received: a, b
Inconsistent (Bad) Ordering

Multiple senders multicasting a stream of messages

Consistent (good) ordering = All group members will receive the messages in the same order

Inconsistent (bad) ordering = Some group members receive the messages in a different order than others
Sending vs. Receiving vs. Delivering

• After a message is **sent**, it arrives at its destination and is **received** by the operating system.

• A **multicast receiver algorithm** decides when to **deliver** a message to the process.

• A received message may be:
  - **Delivered immediately**
    (put on a delivery queue that the process reads)
  - **Placed on a hold-back queue**
    (because we need to wait for an earlier message)
  - **Rejected/discard**
    (a duplicate or earlier message that we no longer want)
Sending, delivering, holding back

sender

Multicast sending algorithm

send

receiver

deliver

delivery queue

hold-back queue

Multicast receiving algorithm

receive

message transmission

? discard
Global time ordering

- All messages are delivered in exact order sent
- Assumes two events never happen at the exact same time!
- Difficult (impossible) to achieve
  - Multiple events may have the same timestamp
  - Clocks may not be perfectly synchronized
  - A process has no way of knowing it is still missing messages
- Not a viable approach
Total ordering

• Consistent ordering at all receivers
• All messages are delivered at all group members in the same order
  – They are sorted into the same sequence before being placed on the delivery queue

Implementation:
• Attach unique totally sequenced message ID
• Receiver delivers a message to the application only if it has received all messages with a smaller ID
• Otherwise, the message sits in the hold-back queue

1. If a process sends $m$ before $m'$
   then any other process that delivers $m'$ will have delivered $m$.
2. If a process delivers $m'$ before $m''$ then every other process will have delivered $m'$ before $m''$. 
Causal ordering

Also known as *partial ordering*

Messages sequenced by only if they are causally related
(e.g., by Lamport or Vector timestamps)

If multicast(G, m) → multicast(G, m′)
then *every* process that delivers m′ will have delivered m

If message m′ is causally dependent on message m,
all processes must deliver m before m′
Causal ordering example

$m_1$ is causally dependent on the receipt of $m_0$
\[ \Rightarrow m_1 \text{ must be delivered only after } m_0 \text{ has been delivered} \]

$m_0$ and $m_2$ have no causal relationship (they are concurrent)
\[ \Rightarrow \text{Any process can deliver these messages in any order} \]
Causal ordering – implementation

Implementation: $P_a$ receives a message from $P_b$

- Each process keeps a **precedence vector**

- Vector is updated on multicast *send* and *receive* events

Each position in the vector = sequence number of the latest message from the corresponding group member that causally precedes the event: $[P_0, P_1, P_2, \ldots]$
Causal ordering – implementation

Algorithm

- When $P_a$ sends a message, it increments its own entry and sends the vector
  $$V_a[a] = V_a[a] + 1$$
  where $a$ is the index for process $P_a$
  Send $V_a$ with the message

- When $P_b$ receives a message from $P_a$
  1. Check that the message arrived in sequential order from $P_a$:
     $$V_a[a] == V_b[a] + 1 ?$$
  2. Check that the message does not causally depend on messages $P_b$ has not received from other processes:
     $$\forall i, i \neq a: \ V_a[i] \leq V_b[i] ?$$

*The sequence # of every other message in $P_a$ must be $\leq$ the corresponding on in $P_b*$

- If both conditions are satisfied, $P_b$ will deliver the message to the application:
  At $P_b$, update the precedence vector: $V_b[a] = V_b[a]+1$

- Otherwise, *hold the message* until these conditions are satisfied
P₂ receives message m₁ from P₁ with V₁=(1,1,0)

1) Is this in sequential order from P₁?
   Compare current V on P₂: V₂=(0,0,0) with received V from P₁, V₁=(1,1,0)
   Yes: V₂[1] = 0, received V₁[1] = 1  ⇒  sequential order – message 1 follows message 0

2) Is V₁[i] ≤ V₂[i] for all other i?
   Compare the same vectors: V₁=(1,1,0)  vs.  V₂=(0,0,0)
   No, because (V₁[0] = 1) > (V₂[0] = 0)
   – this means P₂ has seen msg #1 from P₀ that P₂ has not yet received

Therefore: hold back m₁ at P₂
Next, $P_2$ receives message $m_0$ from $P_0$ with $V=(1,0,0)$

(1) Is $m_0$ in sequential order from $P_0$?

Compare current $V$ on $P_2$: $V_2=(0,0,0)$ with received $V$ from $P_0$, $V_0=(1,0,0)$

Yes: $V_2[0] = 0$, received $V_0[0] = 1 \Rightarrow$ sequential order

(2) Is $V_0[i] \leq V_2[i]$ for all other $i$?

Yes. Element 0: $0 \leq 0$, Element 1: $0 \leq 0$

**Deliver $m_0$ on $P_2$ and update precedence vector on $P_2$ from $(0, 0, 0)$ to $(1, 0, 0)$**

Now check hold-back queue. Can we deliver $m_1$?
Causal Ordering: Example

Check the message in the hold-back set

(1) Is the held-back message \( m_1 \) in sequential order from \( P_0 \)?

Compare element 1 on current \( V \) on \( P_2 \): \( V_2 = (1,0,0) \) with held-back \( V \) from \( P_0 \), \( V_0 = (1,1,0) \)

Yes: (current \( V_2[1] = 0 \)) vs. (received \( V_1[1] = 1 \)) ⇒ sequential

(2) Is \( V_0[i] \leq V_2[i] \) for all other \( i \)?

Now yes. \( (V_0[0] = 1) \leq (V_2[0] = 1) \) and element 2: \( (V_0[2] = 0) \leq (V_2[2] = 0) \)

Deliver \( m_1 \) on \( P_2 \) and update the precedence vector on \( P_2 \): \( V_2 = (1,1,0) \)
Causal Ordering

- Causal ordering can be implemented more efficiently than total ordering:
  - No need for a global sequencer
  - Expect reliable delivery but we may not need to send immediate acknowledgements
Sync ordering

• Messages can be delivered in any order

• Special message type
  – Synchronization primitive = \textit{barrier}
  – Ensure all pending messages are delivered before any additional (post-sync) messages are accepted

If \( m \) is sent with a sync-ordered primitive and \( m' \) is multicast, then every process either delivers \( m \) before \( m' \) or delivers \( m' \) before \( m \).

Multiple sync-ordered primitives from the same process must be delivered in order.
Single Source FIFO (SSF) ordering

• Messages from the same source are delivered in the order they were sent
  – Message $m$ must be delivered before message $m'$ iff $m$ was sent before $m'$ from the same host

If a process issues a multicast of $m$ followed by $m'$, then every process that delivers $m'$ will have already delivered $m$. 
Unordered multicast

- Messages can be delivered in different order to different members
- Order per-source does not matter
Multicasting considerations

- Atomic
- Reliable
- Unreliable

Message Ordering

Single-Source FIFO

unordered

sync

causal

total

global

Reliability
The End