Modes of communication

- **One-to-One**
  - Unicast
  - 1-to-1
  - Point-to-point
- **Anycast**
  - 1-to-nearest 1 of several identical nodes
  - Introduced with IPv6; used with BGP routing protocol
- **One-to-many**
  - Broadcast
  - 1-to-all
  - Multicast
  - 1-to-many = group communication

Groups

Groups allow us to deal with a collection of processes as one abstraction

Send message to one entity
  - Deliver to entire group

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

Primitives:
- `join_group`
- `leave_group`
- `send_to_group`
- `query_membership`
  (sometimes)

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)
    - Root coordinator: forwards message to appropriate subgroup coordinators
- **Managing membership & group creation/deletion**
  - Distributed vs. centralized
- **Leaving & joining must be synchronous**
- **Fault tolerance**
  - Reliable message delivery? What about missing members?

Failure considerations

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
- **Partition failure**
  - The network may get segmented, dividing the group into two or more unreachable sub-groups

Implementing Group Communication Mechanisms
Hardware multicast

If we have hardware support for multicast
  – Group members listen on network address

Diffusion group: send to all clients & then filter
  – Software filters incoming multicast address
  – May need to use auxiliary address to identify the group
    (not in the network address header)

Hardware multicast & broadcast

• Ethernet supports both multicast & broadcast
• Limited to local area networks

Software implementation: multiple unicasts

Sender knows group members

Software implementation: hierarchical

Multiple unicasts via group coordinator
  – Coordinator knows group members
  – Coordinator iterates through group members
  – May support a hierarchy of coordinators

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
    - Use a sequence # on each message
    - Receiver requests retransmission of a missed message
    - More efficient but requires sender to buffer messages indefinitely
  - TCP (not multicast) does the first three of these... but with groups we must do this for each recipient

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
  - 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
  - Send message to all group members
  - Write message to log
  - 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 will restart where they left off
Achieving atomicity – example 2

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

Good Ordering

Message ordering

Bad Ordering

Good ordering = consistent order
If a node sends a sequence of messages, all group members will receive the messages in the same order

Bad ordering ≠ Some group members receive the messages in a different order than others

Good Ordering

Bad Ordering

Good ordering ≠ consistent order
If a node sends a sequence of messages, all group members will receive the messages in the same order

Bad ordering ≠ Some group members receive the messages in a different order than others
Sending vs. Receiving vs. Delivering

- 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/discarded (duplicate or earlier message that we no longer want)

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
- Not viable

Causal ordering

Also known as partial ordering

- Messages sequenced 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'.

Total ordering

- Consistent ordering at all receivers
- All messages are delivered at all group members in the same order
  - They are sorted in the same order in the delivery 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'.

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

Causal ordering example

- m is causally dependent on the receipt of m'. Hence, m must be delivered after m' has been delivered.
- m and m' have no causal relationship (they are concurrent). Any process can deliver them in any order.
Causal ordering – implementation

Implementation: Pa receives a message from Pb

- Each process keeps a precedence vector
  (similar to vector timestamp)

- Vector is updated on multicast send and receive events
  - Each position in the vector = sequence number of latest message from
    the corresponding group member that causally precedes the event

\[(P_0, P_1, P_2, \ldots)\]

Precedence Vector \( V_a \)
Precedence Vector \( V_b \)

Algorithm

- When Pa 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 Pb receives a message from Pa
  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 must be ≤ the one \( P_b \) has.
  - If both conditions are satisfied, \( P_a \) will deliver the message to the application

  - At \( P_a \), update the precedence vector: \( V_a[a] = V_a[a] + 1 \)

  - Otherwise, hold the message until these conditions are satisfied

Causal Ordering: Example

P0 receives message \( m_1 \) from P1 with \( V_1 = (1, 1, 0) \)

1. Is this in sequential order from P1?
   - Compare current \( V \) on P2:
     \( V_2 = (0, 0, 0) \)
   - with received \( V \) from P1, \( V_1 = (1, 1, 0) \)
   - Yes: \( V_2[1] = 0, \) received \( V_1[1] = 1 \)
   - ⇒ sequential order
     - message 1 follows message 0

2. Is \( V_1[i] \leq V_2[i] \) for all other \( i \)?
   - No, because \( V_1[0] = 1 > V_2[0] = 0 \)
   - this means \( P_2 \) has seen msg #1 from \( P_0 \) that \( P_2 \) has not yet received

Therefore: hold back \( m_1 \) at \( P_2 \)

Next, \( P_2 \) receives message \( m_0 \) from \( P_0 \) with \( V_0 = (1, 0, 0) \)

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 \( V_0[0] = 1 \)
   - ⇒ sequential order

2. Is \( V_0[i] \leq V_2[i] \) for all other \( i \)?
   - Yes. Element 0: \( (0 \leq 0) \), Element 2: \( (0 \leq 0) \)

Therefore: deliver \( m_0 \) on \( P_2 \)

Now check hold-back queue. Can we deliver \( m_1 \)?

Check the message in the hold-back set

1. Is the hold-back message \( m_0 \) in sequential order from \( P_1 \)?
   - Compare element 1 on current \( V \) on \( P_2 \):
     \( V_2 = (0, 0, 0) \)
   - with held-back \( V \) from \( P_0 \), \( V_0 = (1, 1, 0) \)
   - Yes: \( (current \ V_2[1] = 0) \) vs. \( (received \ V_0[1] = 1) \) = 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 \( m_0 \) 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 arrive in any order
- Special message type
  - Synchronization primitive = 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' if 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

<table>
<thead>
<tr>
<th>Reliability</th>
<th>Atomic</th>
<th>Reliable</th>
<th>Unreliable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sync</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Causal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unordered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IP multicast routing

- Deliver messages to a subset of nodes
  - Send to a multicast address
- How do we identify the recipients?
  - Enumerate them in the header?
  - What if we don't know?
  - What if we have thousands of recipients?
- Use a special IP address to identify a group of receivers
  - A copy of the packet is delivered to all receivers associated with that group
    - IPv4: Class D multicast IP address
      - 32-bit address that starts with 1110 (224.0.0.0 – 239.255.255.255)
    - IPv6: 128-bit address with high-order bits 8 bits all 1
      - ff00:0:0:0:0:0:0:0/8
      - Host group = set of machines listening to a particular multicast address
    - A copy of the message is delivered to all receivers associated with that group
IP multicasting

- Can span multiple physical networks
- Dynamic membership
  - Machine can join or leave at any time
- No restriction on number of hosts in a group
- Machine does not need to be a member to send messages
- Efficient: Packets are replicated only when necessary
- Like IP, no delivery guarantees

IP multicast addresses

- Addresses chosen arbitrarily for an application
- Well-known addresses assigned by IANA

Internet Assigned Numbers Authority
IPv4 addresses: [http://www.iana.orgassignments/multicast-addresses/multicast-addresses.xhtml](http://www.iana.orgassignments/multicast-addresses/multicast-addresses.xhtml)
IPv6 addresses: [https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml](https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml)

- Similar to ports – service-based allocation
- For ports, we have:
  - FTP: port 21, SMTP: port 25, HTTP: port 80
- For multicast, we have:
  - 224.0.0.1: all systems on this subnet
  - 224.0.0.2: all multicast routers on subnet
  - 224.0.23.173: Philips Health
  - 224.0.23.52: Amex Market Data
  - 224.0.12.0-63: Microsoft & MSNBC
  - FF02::1:0:0:0:9: RIP routers

IGMP

Internet Group Management Protocol (IGMP)
- Operates between a host and its attached router
- Goal: allow a router to determine to which of its networks to forward IP multicast traffic
- IP protocol (IP protocol number 2)

Three message types
1. Membership_query
   - Sent by a router to all hosts on an interface (i.e., on the LAN) to determine the set of all multicast groups that have been joined by the hosts on that interface
2. Membership_report
   - Host response to a query or an initial join or a group
3. Leave_group
   - Host indicates that it is no longer interested
   - Optional: router infers this if the host does not respond to a query

Multicast Forwarding

IGMP allows a host to subscribe to receive a multicast stream

What about the source?
- There is no protocol for the source!
- It just sends one message to a class D address
- Routers have to do the work

IGMP & Wide-Area Multicast Routing

Multicast Forwarding

- IGMP: Internet Group Management Protocol
  - Designed for routers to talk with hosts on directly connected networks

- PIM: Protocol Independent Multicast
  - Multicast Routing Protocol for delivering packets across routers
  - Topology discovery is handled by other protocols
  - Two forms:
    1. Dense Mode (PIM-DM)
    2. Sparse Mode (PIM-SM)
PIM-DM: Dense Mode Multicast – *flooding*

**Forward multicast packet to all connected routers**

- Use a spanning tree and reverse path forwarding (RPF) to avoid loops
- Feedback & cut off if there are no interested receivers on a link
  - A router sends a `prune` message.
  - Periodically, routers send messages to refresh the prune state
- Flooding is initiated by the sender’s router

- Use Reverse path forwarding (RPF): avoid routing loops
  - Packet is duplicated & forwarded ONLY IF it was received via the link that is the shortest path to the sender
  - Shortest path is found by checking the router’s forwarding table to the source address

**Advantage:**
- Simple
- Good if the packet is desired in most locations

**Disadvantage:**
- Wasteful on the network, wasteful extra state & packet duplication on routers

---

PIM-SM: Sparse Mode Multicast

**Initiated by the routers at each receiver.** Each router requests a multicast feed with a PIM `Join` message

- Initiated by a router at the destination that gets an IGMP `join`
- Rendezvous Point: meeting place between receivers & source
  - `Join` messages propagate to a defined rendezvous point (RP)
  - Sender transmits only to the rendezvous point
  - RP announcement messages inform edge routers of rendezvous points
- A `Prune` message stops a feed

**Advantage**
- Packets go only where needed
- Creates extra state in routers only where needed

---

IP Multicast in use

**Initially exciting:**
- Internet radio, NASA shuttle missions, collaborative gaming

**But:**
- Few ISPs enabled it
- For the user, required tapping into existing streams
  - (not good for on-demand content)
- Industry embraced unicast instead

---

IP Multicast in use: IPTV

- IPTV has emerged as the biggest user of IP multicast
  - Cable TV networks have migrated (or are migrating) to IP delivery

- Cable TV systems: aggregate bandwidth ~ 4.5 Gbps
  - Video streams: MPEG-2 or MPEG-4 (H.264)
  - MPEG-2 HD: ~30 Mbps ⇒ 150 channels = ~4.5 Gbps
  - MPEG-4 HD: ~6-9 Mbps; DVD quality: ~2 Mbps

- Multicast
  - Reduces the number of servers needed
  - Reduces the number of duplicate network streams
The end