Group communication:

- **Required readings:** Tanenbaum Section 7.2.1 (pp. 368-370) and 7.4
  Mullender Chapter 5.3.-5.4.

- **Key Idea:** Distributed system design based on the notion of process groups and multicast communication.

- **What is a group?** A collection of processes that cooperate to provide a service.

- Point-to-point vs. one-to-many communication primitives
  What is wrong with RPC or BSD sockets?

---

Group Communication

- **Motivation?**
  - Replicate servers for fault-tolerance
  - Replicate servers for performance
  - Programming abstraction

- **Key Attributes:**
  - View a collection of related processes as a single abstraction.
  - A message sent to the group is delivered to all members.
  - Groups are dynamic: Membership changes due to joins, departures or failures.
  - Frequency of group creation/deletion vs. membership changes within a group
  - Network communication support:
    - Broadcasting
    - Multicasting
    - Unicasting
Design Issues for Developing Group Communication Primitives

- Closed vs. open groups
- Symmetric (peer) vs. asymmetric (hierarchical) groups
- Group membership problem
- Group addressing
- Atomicity: all-or-nothing property of group communication
- Message ordering
- Scalability

Objective of this lecture:
- Motivation for group multicasts
- Specification of various types of multicasts
- Introduction to multicast protocols

Reading: Mullender Sections 5.3 - 5.4.
Contemporary O.S. offer three types of communication services:

- Unreliable Datagrams
- Reliable Data Streams
- Remote Procedure Calls

Who needs yet another type of communication service?

Programming support for building distributed systems from groups of cooperating processes!

Example 1: Replicated Queues

Suppose a replicated queue is maintained by three server processes $s_1$, $s_2$, and $s_3$. Updates to these queue are issues from client processes often concurrently.

- What if there is a message loss?
- What if sender crashes after some, but not all, deliveries.
- What if messages delivered in different order?

- Broadcast Delivery
- Reliable Delivery
- Ordered Delivery
Example 2: Parallel Database Search

Suppose a database resides on $n$ nodes in a system. A client process $c_1$ that wants to search the database in parallel sends a multicast message to a group of server processes $s_1$, $s_2$, and $s_n$. The members of this server group partition the work and each searches $1/n$ of the database.

Suppose $s_3$ fails while $c_1$ is sending a multicast message $m$.

What if failure of $s_3$ is seen by some before the receipt of the message $m$ and by other after its receipts?

- Failure Atomicity
Non-Blocking Three-Phase Atomic Commit

Skeen's protocol, 1982.

1. Sender sends a message to destinations which acknowledge.
2. Sender tells all destination processes that the first round ended and each destination acknowledges.
3. Sender informs all destinations to discard information about message.

Classification of Broadcasts

- Reliable broadcast
- FIFO broadcast
- Causal broadcast
- Atomic broadcast
- FIFO atomic broadcast
- Causal atomic broadcast
- Timed broadcast (real-time)
- Timed broadcast (local-time)
- Uniform broadcasts
Layering of Protocols

\[ p \leftarrow \text{sender}(m) \]
\[ q \leftarrow \text{receiver}(m) \]
\[ i \leftarrow \text{seq}(m) \]
Reliable Broadcast

Informal specification:
- All correct processes deliver the same set, and all messages by correct processes will be delivered and no false messages.

Reliable broadcast is a broadcast with the following properties:
- **Validity**: If a correct process broadcasts a message \( m \), then all correct processes eventually deliver \( m \).
- **Agreement**: If a correct process delivers a message \( m \), then all correct processes eventually deliver \( m \).
- **Integrity**: For any message \( m \), every correct process delivers \( m \) at most once, and only if some process broadcasts \( m \).

What if a process \( p \) crashes while broadcasting to other processes?
- Crashes immediately after it calls and no delivery takes place.
- Crashes in the middle and others must ensure delivery.
FIFO Broadcast

Informal specification: All broadcasts from a correct sender must be delivered in the order sent.

FIFO broadcast is a reliable broadcast that satisfies the following property:

- **FIFO Order**: If a process broadcasts a message \( m \) before it broadcasts a message \( m' \), then no correct process delivers \( m' \) unless it delivered \( m \).

Example:

- \( p \) sends a broadcast message a group of replicated directory servers to create an entry in the table.
- \( p \) sends a message updating processor name and ip address in the entry.

A faulty specification: "all messages broadcast by the same process are delivered to all processes in the order they are sent."

What is wrong?

- A process \( p \) suffers a transient failure while sending message \( m_2 \) in the sequence \( m_1, m_2, \) and \( m_3 \).
- A correct process \( q \) delivers \( m_1 \) and \( m_3 \) in the right order.

Another problem?

- It requires a faulty process to deliver a message (putting aside the order in which it must do the delivery)!
Causal Broadcast

Why causal broadcast?

Sometimes the context of a message is dependent on the earlier messages delivered to sender.

An example: e-mail broadcast message and its reply ...

Definition: Based on Lamport’s ‘happen before’ relation, we define \( e \) causally precedes \( e' \) for broadcast messages. \( e \rightarrow e' \) iff:

1. a process executes \( e \) before \( e' \), or
2. \( e \) is the broadcast of some message \( m \) and \( e' \) is the delivery of \( m \), or
3. there is an event \( e'' \) such that \( e \rightarrow e'' \) and \( e'' \rightarrow e' \)

Causal broadcast is a reliable broadcast that satisfies the following property:

- Causal Order: If the broadcast of a message \( m \) causally precedes the broadcast of a message \( m' \), then no correct process delivers \( m' \) unless it has already delivered \( m \).
An incorrect formulation: if a broadcast \( m \) causally precedes a broadcast \( m' \), then every correct process that delivers both must deliver message \( m \) before message \( m' \).

What is wrong?

---

Atomic Broadcast

Causal broadcast allow unrelated messages to be delivered in any order.

Some applications may require that correct processes deliver the same sequence of messages.

Example: Replicated database with two copies of account info

- Initially, \( x = 100 \).
- User deposits \( 20: m_1 = [\text{add 20 to } x] \)
- Bank updates interest earned: \( m_2 = [\text{add 10\% to } x] \)
Atomic broadcast is a reliable broadcast that satisfies the following property:

- Total Order: If correct processes \( p \) and \( q \) both deliver messages \( m \) and \( m' \), then \( p \) delivers \( m \) before \( m' \) iff \( q \) delivers \( m \) before \( m' \).

FIFO Atomic Broadcast

Is atomic broadcast stronger than FIFO?

Consider a scenario:

- \( p \) suffers an omission failure while sending a broadcast message \( m \) (no message gets delivered).
- \( p \) subsequently broadcasts \( m' \) and correct processes deliver \( m' \).

Allowed by atomic broadcast. Not allowed by FIFO broadcast.

FIFO Atomic broadcast satisfies both FIFO and total order.
Causal Atomic Broadcast

- Stronger than FIFO atomic broadcast.
- Basis for State Machine approach to fault-tolerance.
- **Causal atomic** satisfies causal and total order.
Timed Broadcast

\( \Delta \)-Timeliness: Deliver a broadcast (if it is delivered at all), within a bounded time after it was broadcast.

Two interpretations:

**Real-Time \( \Delta \)-Timeliness:** There is a known constant such that if a broadcast of \( m \) is initiated at real-time \( t \), no correct process delivers \( m \) after real-time \( t + \Delta \).

**Local-Time \( \Delta \)-Timeliness:** There is known constant \( \Delta \) such that no correct process \( p \) delivers a message \( m \) after local time \( ts(m) + \Delta \) on \( p \)'s clock.

Observations:
- Timed reliable broadcast: is not implementable in asynchronous system.
- Timed reliable broadcast that satisfies local-time version is implementable in synchronous systems.

Inconsistency:
A faulty process can become inconsistent by failing with respect to the delivery of a message.

Contamination:
A correct process can become contaminated by a faulty process that had an inconsistent state after a previous incorrect delivery.

Example:
- Three processes \( p, q \) and \( r \) maintain a replicated variable \( x \).
- \( x = 5 \) initially.
- \( p \) broadcasts a message \( m = [\text{increment } x] \).
- \( q \) broadcasts a message \( m' = [\text{double } x] \).
- Both \( p \) and \( q \) deliver both messages resulting in the state \( x = 12 \).
- \( r \) is faulty; omits to deliver \( m \); delivers \( m' \) resulting in the state \( x = 10 \).
Observations:
- $r$ has an inconsistent state. But its execution satisfies uniform atomic broadcast.
  - $y = 3x$ is broadcast by $r$.
  - $p$ and $q$ deliver the message resulting in the state $y = 30$.

What to do about contamination?
Prevent delivery of messages by processes who past deliveries are not compatible with its own!
Filter messages to prevent faulty processes from interacting with correct ones.

**Group membership problem!**
*(more next week)*
Broadcast Protocols

In asynchronous systems, reliable broadcast, FIFO broadcast, and causal broadcast can be implemented.

Distributed Consensus = Atomic Broadcast

No deterministic algorithm for atomic broadcast exists for asynchronous system with a single process crash even if communication network is reliable and fully connected!