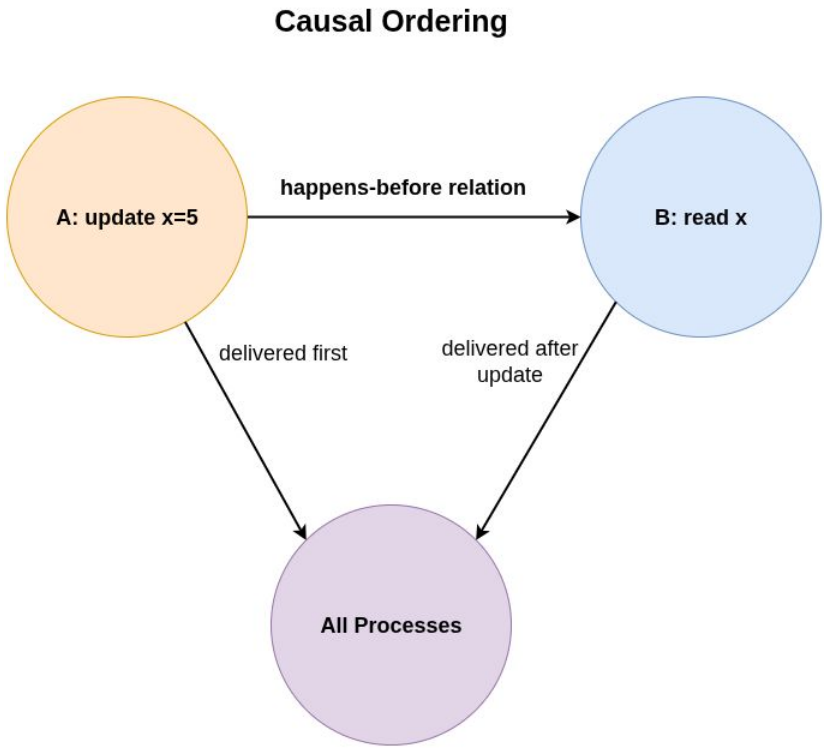
Distributed Computing

Lesson 6

Message Ordering in Distributed Systems

- In a distributed system, messages travel independently through the network.
- To keep the system consistent, the *order of messages matters* just as much as the messages themselves.
- Example: If a replica receives “update x=5” before “read x,” everyone else should also see this order. Otherwise, results diverge.



Total Order

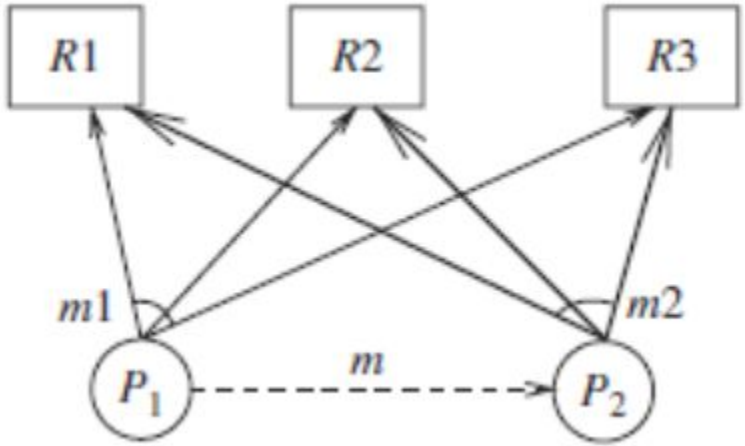
- Imagine you have several replicas of a single data item d.
- If updates arrive in different orders at different replicas, the replicas may disagree.
- **Total order** ensures *all replicas* see messages in the *same order*, regardless of whether the updates are causally related or not.
- Benefit:
  - Removes coherence and consistency problems.
  - Makes “read” operations simple (all replicas agree).
  - Improves fault tolerance (replicas can be queried interchangeably).
- In short: *Total order = Everyone sees the same global sequence of events.*

Example Setup

- P1 sends a message `m1` to the replicas (`R1, R2, R3`).
- P2 sends a message `m2` to the replicas.
- There's also a message `m` exchanged between P1 and P2 (maybe a coordination or trigger message).

Condition for Total Order

- In a distributed system, total order requires that all processes (or replicas) deliver messages in the same sequence.
- It doesn't matter if m1 comes before m2, or m2 before m1.
- What matters is every replica must see the same order.



Example violation Scenario 1

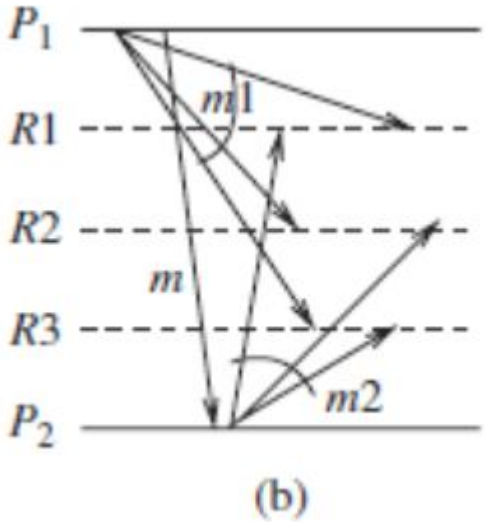
The orders in which messages are received at the replicas are the following:

R1: m2, m1

R2: m1, m2

R3: m1, m2

As the order of receiving the messages are not the same in each of these replicas it violates Total Order



Example satisfying Scenario 2

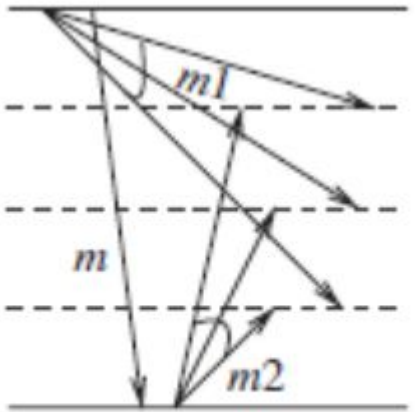
The orders in which messages are received at the replicas are the following:

R1: m2, m1

R2: m2, m1

R3: m2, m1

As the order of receiving the messages are the same in each of these replicas it satisfies Total Order



(c)

Three-Phase Distributed Algorithm

Purpose

- Ensures all processes in a group **see messages in the same order** (total order).
- Also respects **causal relations** between messages (causal order).
- Works for **closed groups** (fixed set of processes).

Phase 1 – Send the message

- The sender wants to broadcast a message `M`.
- It tags the message with:
  - A **unique ID** (so there's no confusion with other messages).
  - Its **local timestamp** (a first guess at ordering).
- Then it **multicasts** this message to all group members.

Phase 2 – Collect proposals

- Each receiver replies with a **tentative proposed timestamp** for when it could safely deliver the message to its application.
- The sender waits until **all replies** arrive. It then takes the **maximum of these proposed timestamps** to ensure **no process delivers the message too early**, and that maximum becomes the **final agreed timestamp**.

Phase 3 – Announce the final time

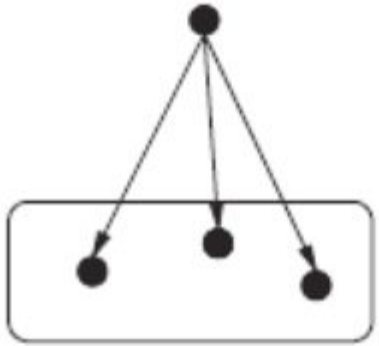
- The sender multicasts the **final timestamp** to all group members.
- Now everyone knows exactly when to deliver the message.

Nomenclature for multicast

Single Source Single Group (SSSG) multicast.

- **Single Source** → There is exactly one sender (the process at the top).
- **Single Group** → That sender transmits messages to exactly one group of receivers (the three processes inside the box).
- This is the simplest multicast case: **one sender, one group**.

Example: A live video stream from a single sports broadcaster (source) sent only to subscribers of one sports channel (group).



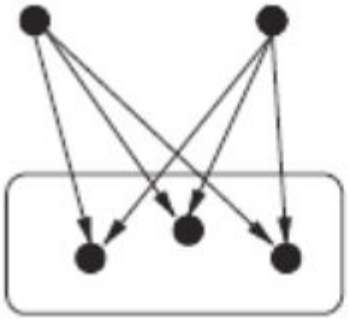
(a) Single source single group (SSSG)

Nomenclature for multicast

Multiple Sources Single Group (MSSG): Here, multiple senders (sources) multicast their messages to a *single* common group of receivers.

Example: In a collaborative chat room or a multiplayer game, several participants (sources) are all sending updates, and everyone in the same group receives them.

The challenge here is ensuring ordering and consistency of messages, since different sources can send concurrently.

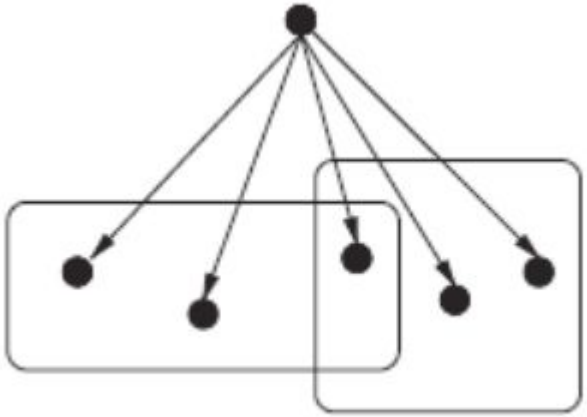


(b) Multiple sources single group (MSSG)

Nomenclature for multicast

Single Source Multiple Groups (SSMG):
One sender transmits to more than one group of receivers, possibly with some overlap between groups.

Example: A news broadcaster (source) streams to two separate groups — one group subscribed to *politics* and another to *business*. Some users may belong to both groups and thus receive the stream twice.



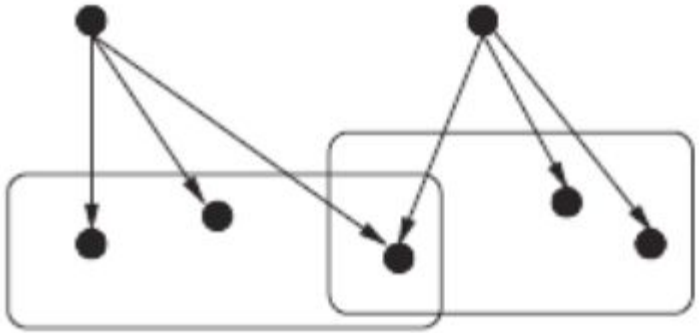
(c) Single source multiple groups (SSMG)

Nomenclature for multicast

Multiple Sources Multiple Groups (MSMG):
In this setup, there are several senders (sources) and several receiver groups. Each source can multicast to one or more groups.

Example: Think of multiple news agencies (sources), each broadcasting different categories of news (sports, politics, finance). Subscribers (receivers) join the groups they are interested in. Some receivers may overlap across groups (e.g., someone who follows both sports and finance).

This is the most general and complex multicast model, since it must handle both **multiple senders** and **multiple groups** simultaneously, with challenges in scalability, ordering, and delivery guarantees.



(d) Multiple sources multiple groups (MSMG)

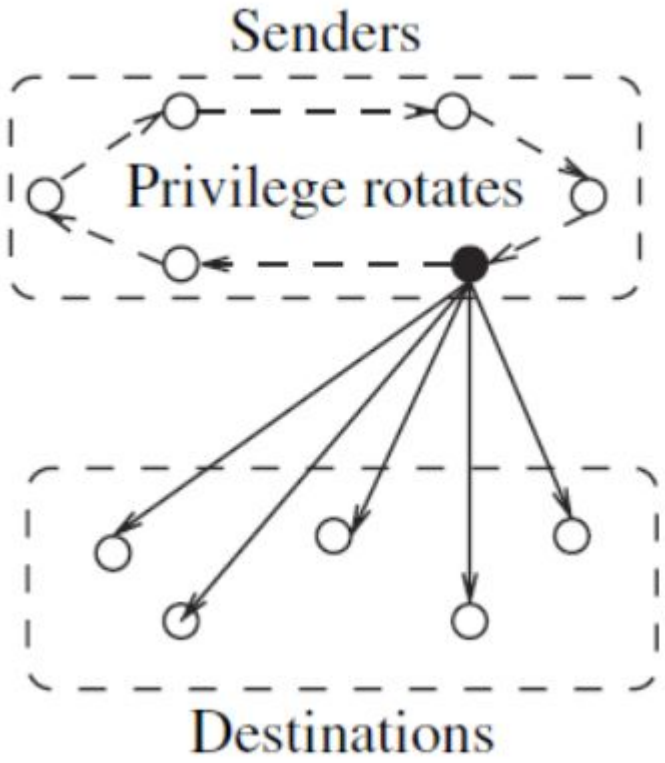
Classification of application-level multicast algorithms

Communication history-based algorithms

These use past communication to enforce ordering. They don’t track fixed groups, so they fit open-group multicasts. Example: **Lamport’s algorithm**, where scalar timestamps ensure a message is delivered only if no earlier time stamped message is pending.

Privilege-based algorithms

A token is passed among senders, and only the token-holder can multicast. The token carries the sequence number, ensuring receivers deliver in increasing order. This guarantees **total and causal ordering** but only in **closed groups**. The drawback is poor scalability since only one sender can multicast at a time.

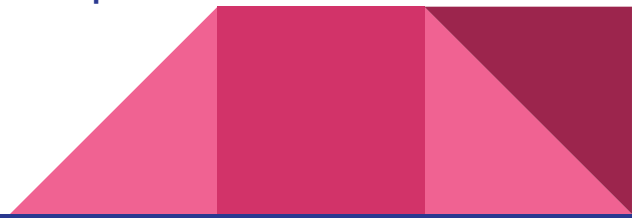


Classification of application-level multicast algorithms

Moving sequencer algorithms

Here, special processes called **sequencers** are responsible for assigning sequence numbers. Senders multicast their messages to all sequencers. The sequencers circulate a **token** that carries the next sequence number and a list of already sequenced messages.

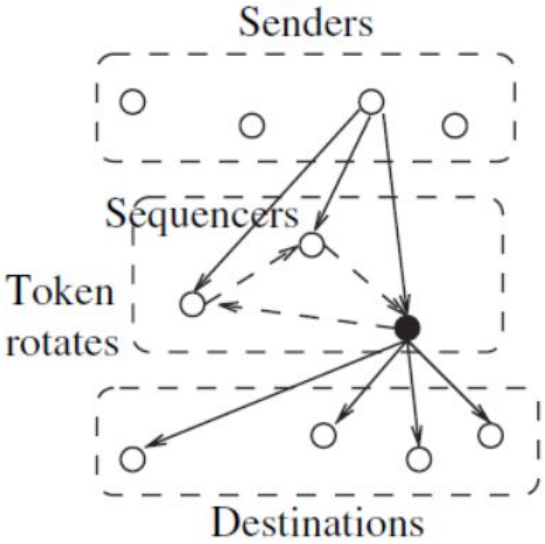
When a sequencer gets the token, it assigns sequence numbers to pending messages, sends them to destinations, updates the token, and passes it on. Receivers then deliver messages strictly in order of these sequence numbers → guaranteeing **total ordering**.



Classification of application-level multicast algorithms

This diagram shows the **Moving Sequencer Algorithm** in action:

- **Senders** (top box) generate messages.
- These messages are sent to the **Sequencers** (middle box).
- Sequencers coordinate using a **rotating token** (dashed arrows). The token carries the next sequence number and history of already sequenced messages.
- The sequencer holding the token assigns sequence numbers to pending messages, then multicasts them to the **Destinations** (bottom box).
- Destinations deliver messages strictly in increasing sequence number order → ensuring **total order** across the system.

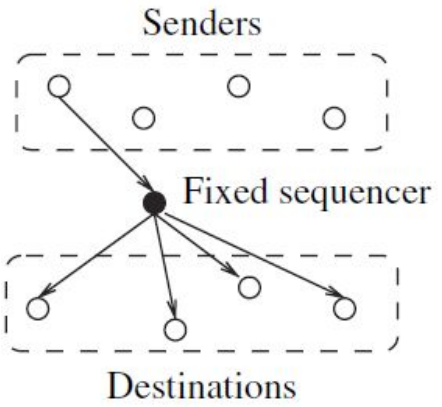


Senders → sequencers (with rotating token) → destinations (ordered delivery)

Classification of application-level multicast algorithms

Fixed Sequencer Algorithms

A fixed sequencer algorithm uses a **single sequencer process** to assign sequence numbers to all multicast messages. This centralizes control and ensures **total ordering**, since every message passes through one sequencer.



Senders (top box): Multiple processes want to multicast their messages.

Fixed Sequencer (middle black dot):

- Every sender forwards its message to this sequencer.
- The sequencer assigns a **sequence number** to each incoming message (e.g., 1, 2, 3…).

Destinations (bottom box):

- The sequencer multicasts the message (now with a sequence number) to all destinations.
- Destinations deliver messages strictly in the order of these assigned sequence numbers.

Destination agreement algorithms

In this class of algorithms, the **destinations themselves decide the delivery order**. They first receive the messages (with limited ordering info like timestamps), then **exchange information** to agree on a consistent order before delivering.

Destinations do not just accept the sender's order; they coordinate among themselves.

Two main approaches:

1. **Timestamp-based:** Use timestamps attached to messages to decide order.
2. **Consensus-based:** Use agreement protocols (like majority voting or Paxos-style consensus) to settle on the order.

The advantage is **decentralization** (no single sequencer).

The downside is **extra communication overhead**, since receivers must interact to finalize the order.

Termination Detection

In distributed systems, a single problem is often solved collectively by many processes running on different machines. To make progress, these processes must cooperate and exchange messages.

A key question is: **how do we know when the entire distributed computation has finished?**

This is important because:

- Results can only be used once all parts of the computation are done.
- Many problems are broken into smaller **subproblems**. A new subproblem cannot begin until the previous one has fully completed.
- Without detecting completion correctly, processes may either stop too early (missing results) or wait forever (wasting resources).

Thus, a **fundamental challenge** in distributed systems is deciding whether a distributed computation has terminated.

Local vs Global Termination

- **Locally terminated**: A process has finished its part of the work and will remain idle unless it receives a new message.
- **Globally terminated**: Every process in the system is locally terminated **and** there are no messages in transit between processes.

The difficulty comes from the fact that:

- No single process has a complete view of the entire system.
- There is no global clock or global state to check directly.
  So, termination must be inferred using carefully designed algorithms.

Two Computations Running in Parallel

Whenever we try to detect termination, we are effectively running **two computations at once**:

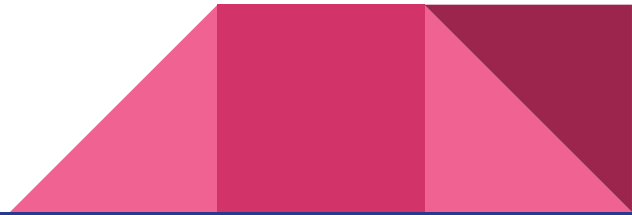
1. **The underlying computation** – the actual work being done (e.g., solving a distributed problem).
2. **The termination detection algorithm** – the extra mechanism that checks whether the work is finished.
    - Messages used for the actual work are called **basic messages**.
    - Messages used by the detection algorithm are called **control messages**.

Requirements of Termination Detection Algorithms

Any termination detection (TD) algorithm must satisfy:

1. **Non-intrusiveness** – it should not freeze or indefinitely delay the actual computation. The system must continue working while detection runs in the background.
2. **No new infrastructure** – it should not require creating new communication channels beyond what the system already uses.

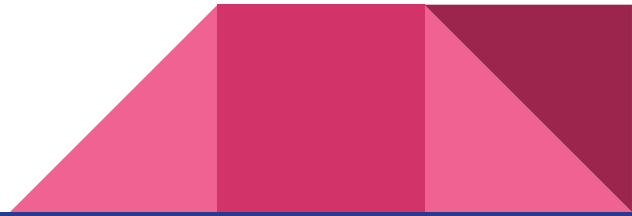
Termination detection is about ensuring that *all processes are done* and *no messages are flying around* in a distributed system. Since no process can see the global state directly, specialized algorithms use local states, message exchanges, and control mechanisms to infer global termination safely and efficiently.



Main Termination Detection Approaches

1. Using Distributed Snapshots

- A snapshot records the *state of processes* and *messages in channels* at one moment.
- Since *termination* is a **stable property** (once true, it stays true), a snapshot taken after termination will capture it.
- **How it works**:
  - When a process becomes idle, it requests everyone to take a local snapshot.
  - If all processes agree and take snapshots, we combine them into a global snapshot.
  - If all processes are idle and no messages are in transit → termination detected.



Main Termination Detection Approaches

2. Weight Throwing Method

- Here, a special **controlling agent** manages the computation.
- The agent starts with **weight = 1**; processes start with **weight = 0**.
- When it activates a process, it gives some of its weight along with the task.
- As messages flow, weight is split and carried around.
- When processes finish, they return their weight to the controller.
- **Termination is detected when the controller regains weight = 1** (meaning all work is done, no weight is left outside).

Main Termination Detection Approaches

3. Spanning-Tree Based Detection

- Processes are arranged in a **spanning tree**.
- Leaf nodes report to their parents when they finish.
- Parents wait until *both they and all their children* are done, then report upward.
- Eventually, the root gets reports from everyone and declares **termination**.
- To ensure correctness, two **waves of signals** (inward and outward) keep repeating until the system stabilises.