Sistemas Distribuídos

Group Communication

Eurico Pedrosa
António Rui Borges

Universidade de Aveiro - DETI

2025-05-07

Group Communication

Characterization

- **Group Communication**
  - All participating **processes are peers**, with **no special roles** or privileges.
  - This is known as **communication among equals**.
- **Access to Shared Resources**
  - Requires **mechanisms to prevent race conditions**.
    - **Race conditions** can cause **inconsistent data** if processes access shared resources unsafely.

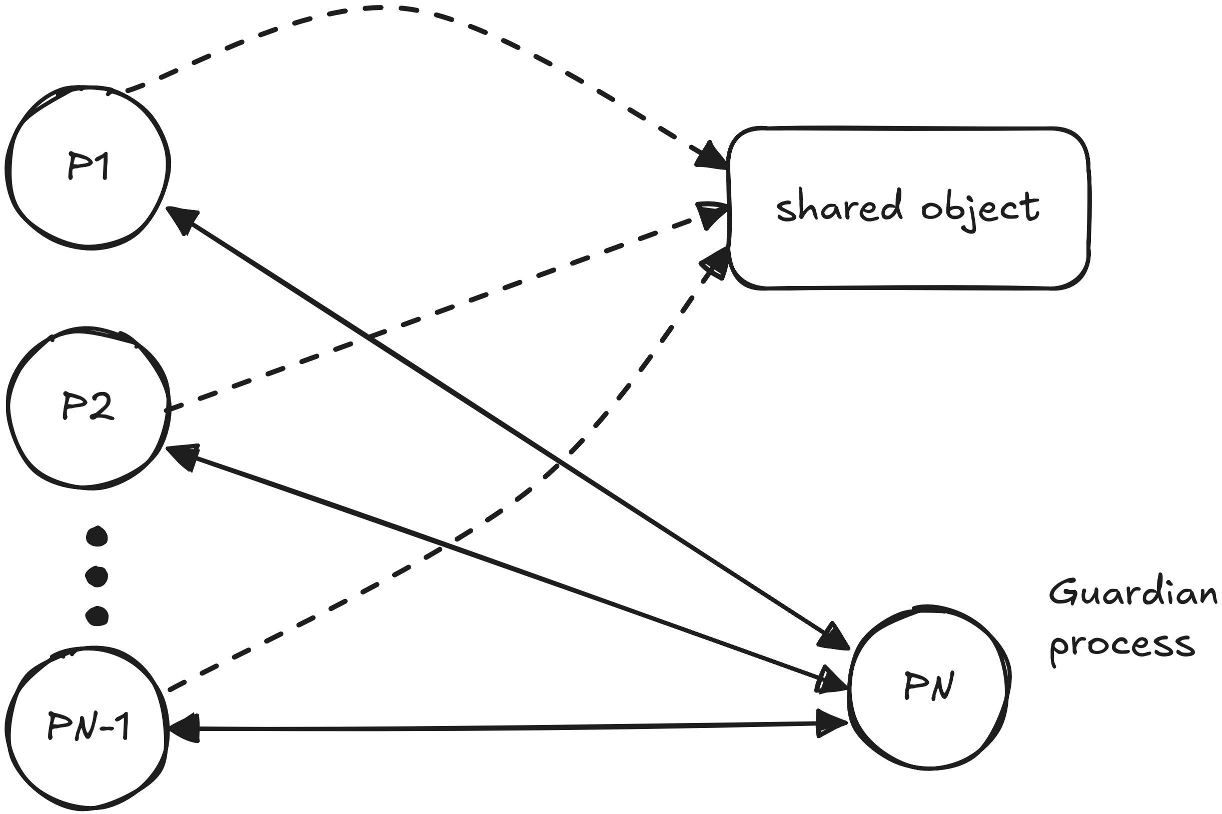
Characterization

- **No Shared Memory Space**
  - Processes **do not share an address space**.
  - **Synchronization** must be achieved through **message passing**.
- **Assumptions**
  - **Message transmission time** is **finite** but has **no upper bound**.
  - **No message loss** is assumed during communication.

Centralized Access Permission

- **Client-Server Adaptation**
  - Represents an **almost direct extension** of the **client-server model**.
    - Includes **request serialization** to handle multiple access attempts in a controlled manner.
- **Guardian Process** or **Coordinator**
  - A **dedicated process**—the **guardian process**—manages access to a **shared object**.
  - It **monitors all access attempts**.
  - Grants access **individually**, **based on incoming requests**.

Centralized Access Permission



Centralized Access Permission

Access Protocol Overview

Applies to **peer processes**- p_i, where $i = 0, 1, \ldots, N - 1$, and a **guardian process** p_N.

- **Requesting Access**
  - When a process p_i wants to **access the shared object**:
    - It sends a **"request access"**- message to the **guardian proc.** p_N.
    - Then it **waits**- for a **"grant access"** message in response.
- **Guardian Response**
  - If **no process is currently accessing**- the shared object:
    - p_N **grants access immediately**.
  - If the object is **in use**:
    - p_N **queues the request**- in a **waiting list**.

Centralized Access Permission

- **Granting Access**
  - When p_i **receives the "grant access"**- message:
    - It may **proceed to access**- the shared object.
- **Releasing Access**
  - Once finished, p_i sends a "**release access**"- message to p_N.
  - If there are **pending requests**:
    - p_N **removes the first request**- from the queue.
    - It sends a "**grant access**"- message to the corresponding process.

Centralized Access Permission

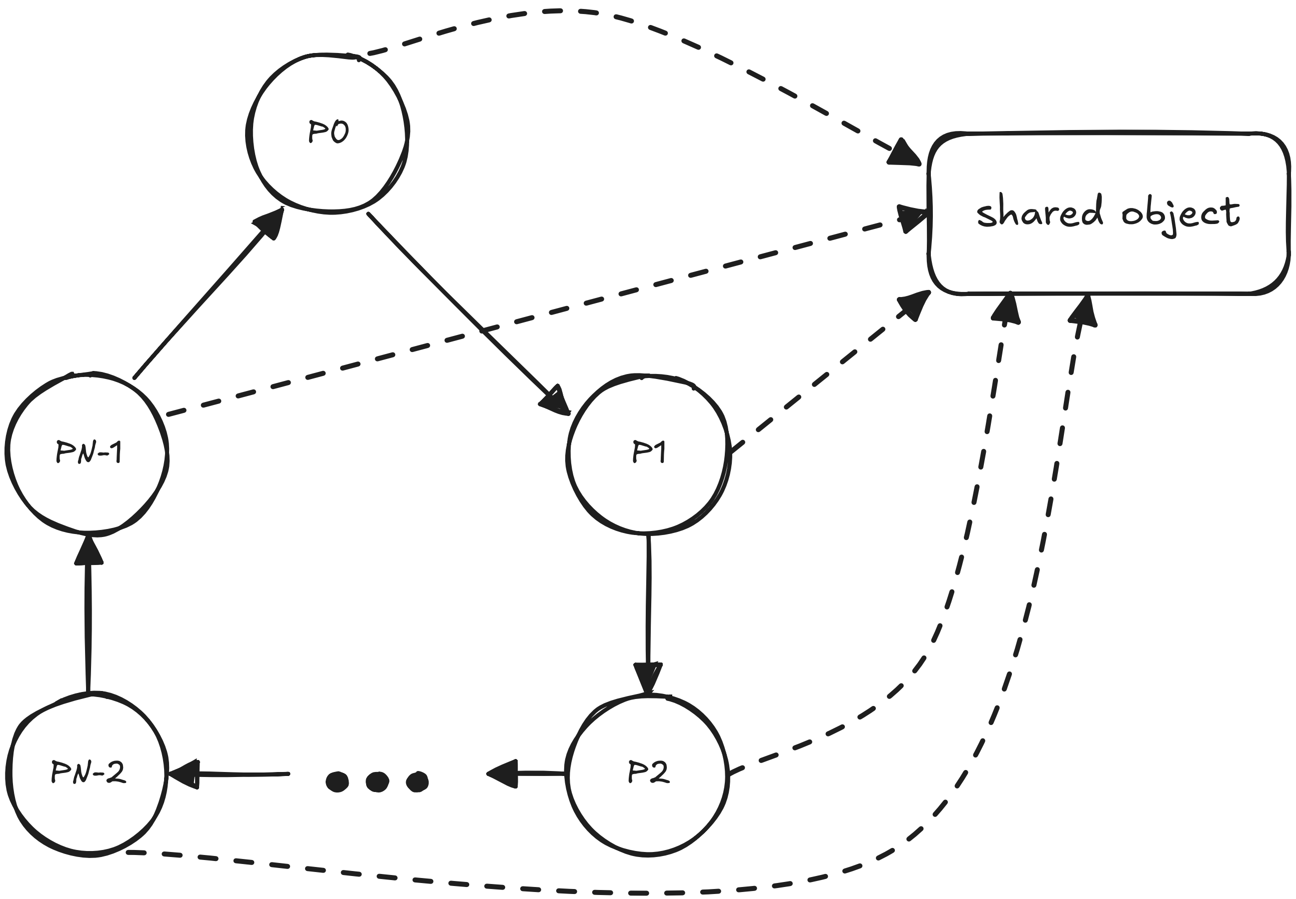
Comments

- **Message Overhead**
  - Each access to the shared object involves **three messages**:
    - **request access**, **grant access** and **release access**.
- **Architectural Limitation**
  - This is **not a fully peer-to-peer solution**.
    - It depends on a **dedicated process**—the **guardian process**—to control access.
- **Single Point of Failure**
  - The guardian process p_N is a **critical component**.
    - If p_N **fails**, the **entire system halts**, as no process can access the shared object.

Logic Ring

- **Logical Ring Topology**
  - Processes are arranged in a **closed communication loop**.
- **Communication Constraints**
  - Each process p_i, where $i = 0, 1, ..., N - 1$:
    - **Receives messages**- only from $p_{(i-1) \bmod N}$,
    - **Sends messages**- only to $p_{(i+1) \bmod N}$.
- **Token Passing Mechanism**
  - A **token message circulates**- continuously among the processes.
  - **Access to the shared object**- is restricted to the process **holding the token**.

Logic Ring



Logic Ring

Access Protocol Overview

- **Token-Based Access Protocol**
  - **Requesting Access**
    - If a process p_i **needs to access** the shared object:
      - It **waits to receive the token**.
      - Once it **holds the token**, it proceeds to **access the object**.
      - After finishing, it **sends the token** to the **next process** in the ring.
  - **No Access Needed**
    - If a process p_i **does not need access**:
      - It **forwards the token immediately** to the **next process** in the ring upon receiving it.

Logic Ring

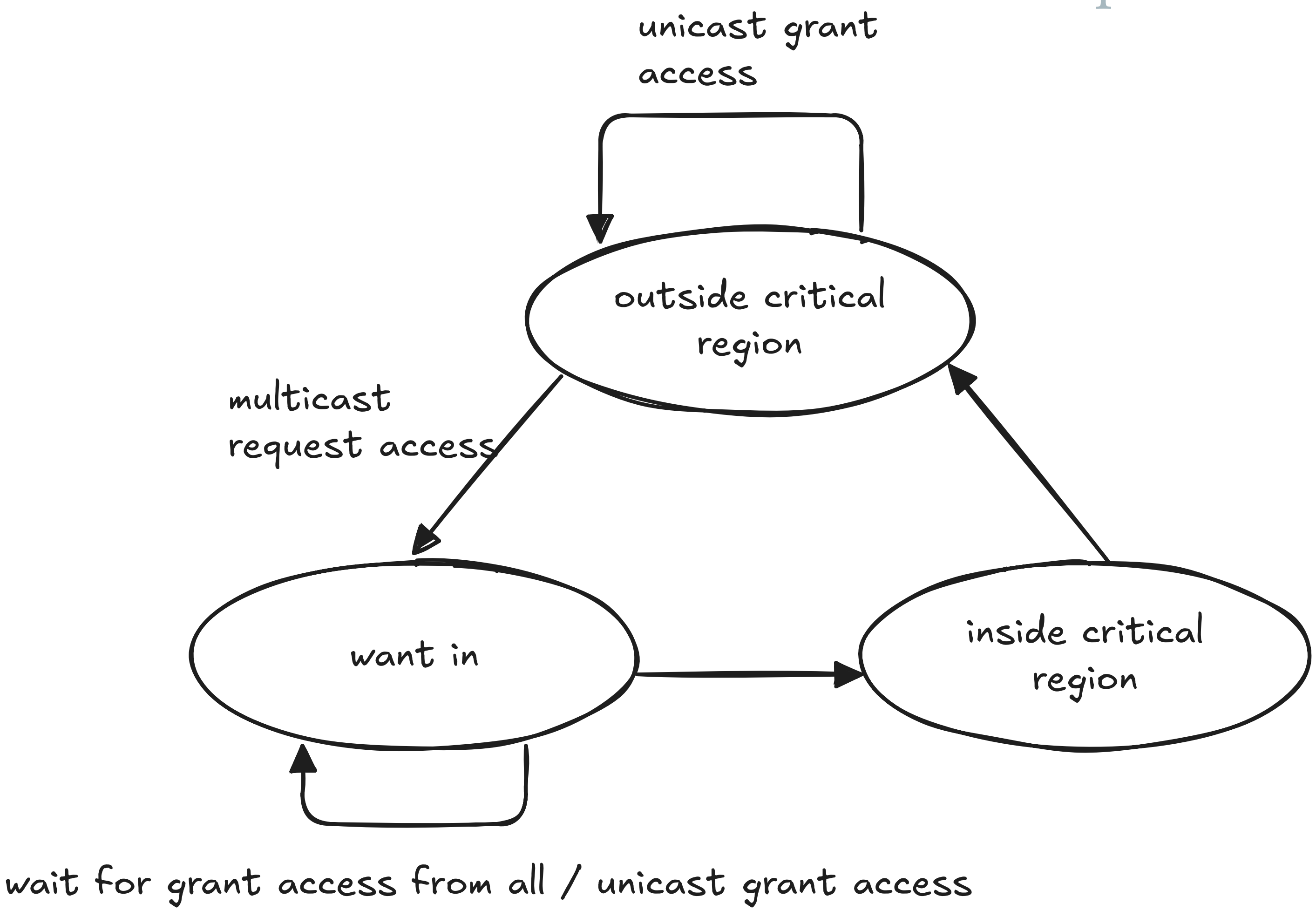
Comments

- **Message Overhead**
  - **One message is always exchanged**, whether or not the shared object is accessed.
- **Efficiency in Small Groups**
  - The protocol is **highly efficient** when the **number of processes is small**.
- **Scalability Limitation**
  - In **large groups**, a process may have to **wait a long time** to access the object.
    - This delay can occur **even if no other process is currently using the object**, due to the **fixed token circulation order**.

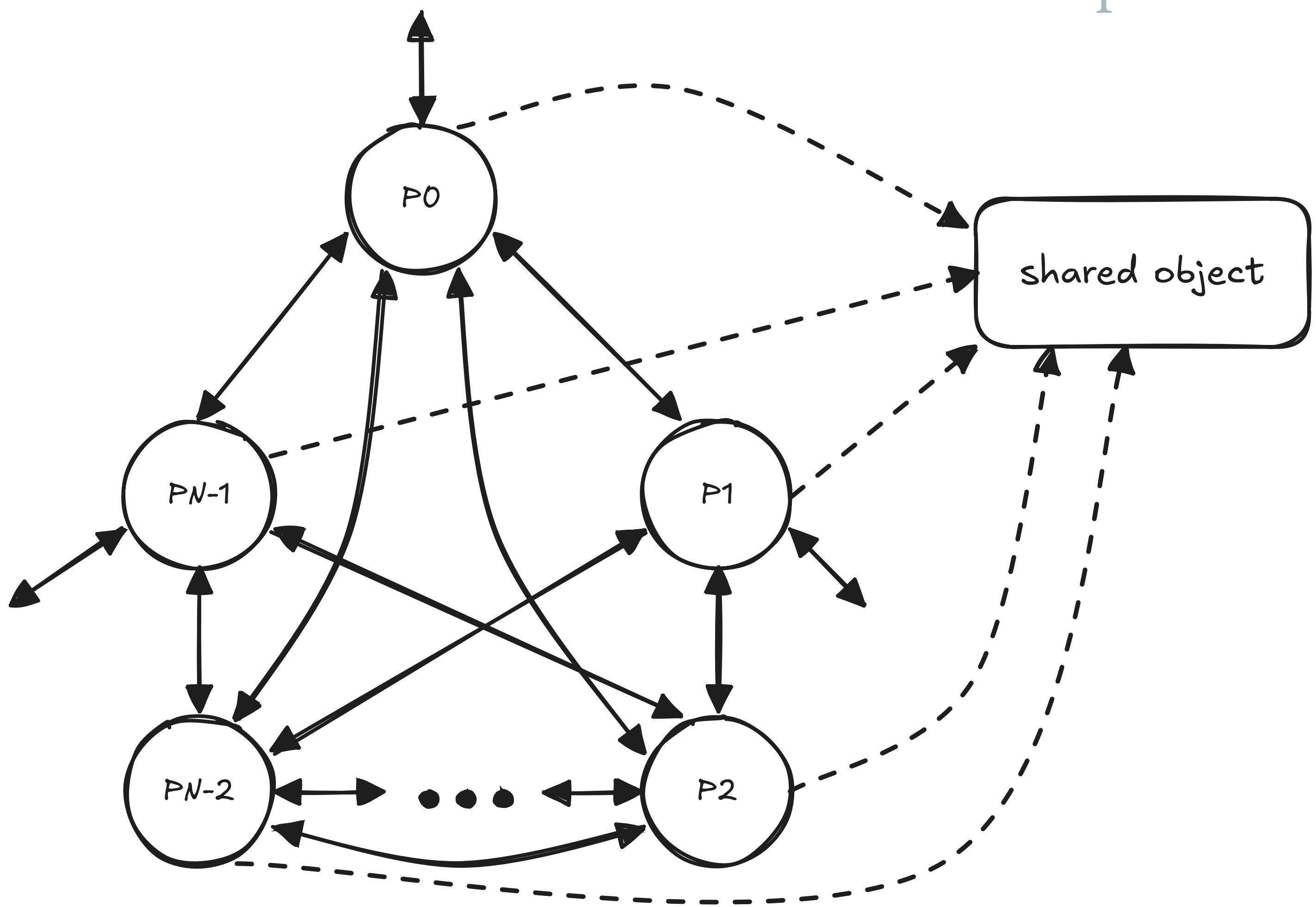
Total Ordering of Events

- **Mutual Exclusion with Logical Clocks**
  - **Ricart and Agrawala (1981)** proposed a method for **ensuring mutual exclusion** among N processes.
  - Access requests are **totally ordered** using **Lamport logical clocks**.
- **Consensus on Access Order**
  - All processes **agree on the order of access requests** to the shared object.
  - This results in an **overall consensus** for granting access.

Total Ordering of Events



Total Ordering of Events



Total Ordering of Events

- **Timestamped Messages**
  - Messages include the **timestamp of the sending event**.
  - Upon reception, a process **adjusts its local logical clock** according to **Lamport's rules**.
- **Extended Timestamps for Total Order**
  - Total ordering is achieved by associating each access request with an **extended timestamp**:
    - $(\mathrm{ts}(m), \mathrm{id}(m))$, where:
      - $\mathrm{ts}(m)$ is the **logical timestamp**,
      - $\mathrm{id}(m)$ is the **sender's process ID**.
    - This ordered pair is used to **break ties** and **totally order** the events.

Total Ordering of Events (Algorithm)


```
// initialization
state = outsideCR;
requestMessageReady = false;
// p_i enters the critical region
state = wantIn;
numberOfRequestsGranted = 0;
myRequestMessage = multicast (requestAccess);
requestMessageReady = true;
wait until (numberOfRequestsGranted == N-1);
state = insideCR;
//pi exits the critical region
state = outsideCR;
requestMessageReady = false;
while (!empty (requestQueue)) {
  id = getId (queueOut (requestQueue));
  unicast (id, accessGranted);
}
```

Total Ordering of Events (Algorithm)


```
//p_i receives an access request message from p_j
if (state == outsideCR) {
  id = getId (requestMessage));
  unicast (id, accessGranted);
} else if (state == insideCR) {
  queueIn (requestQueue, requestMessage);
} else {
  wait until (requestMessageReady);
  if (getExtTimeStamp (myRequestMessage) < getExtTimeStamp (requestMessage))
    queueIn (requestQueue, requestMessage);
  else { id = getId (requestMessage));
    unicast (id, accessGranted);
  }
}

//p_i processes access permissions
numberOfRequestsGranted += 1;
```

Total Ordering of Events (Algorithm)

Comments

- **Message Overhead**
  - Each access to the critical region requires **2(N−1) messages**:
    - **(N−1) request messages** sent by the requesting process.
    - **(N−1) grant messages** sent in response by other processes.
- **Efficiency in Small Groups**
  - The protocol is **highly efficient** when the **number of processes is small**.
- **Scalability Limitation**
  - For **large process groups**, the protocol becomes **communication-intensive**, resulting in the exchange of **a large number of messages per access**.

Minimizing the Number of Messages

- **Group-Based Permission Model**
  - **Maekawa (1985)** proposed that **mutual exclusion** can be achieved without requiring **permission from all processes**.
  - Instead, each process is part of a **smaller subset (group)** of processes.
  - A process must obtain permission **only from all members of its group** to access the shared object.

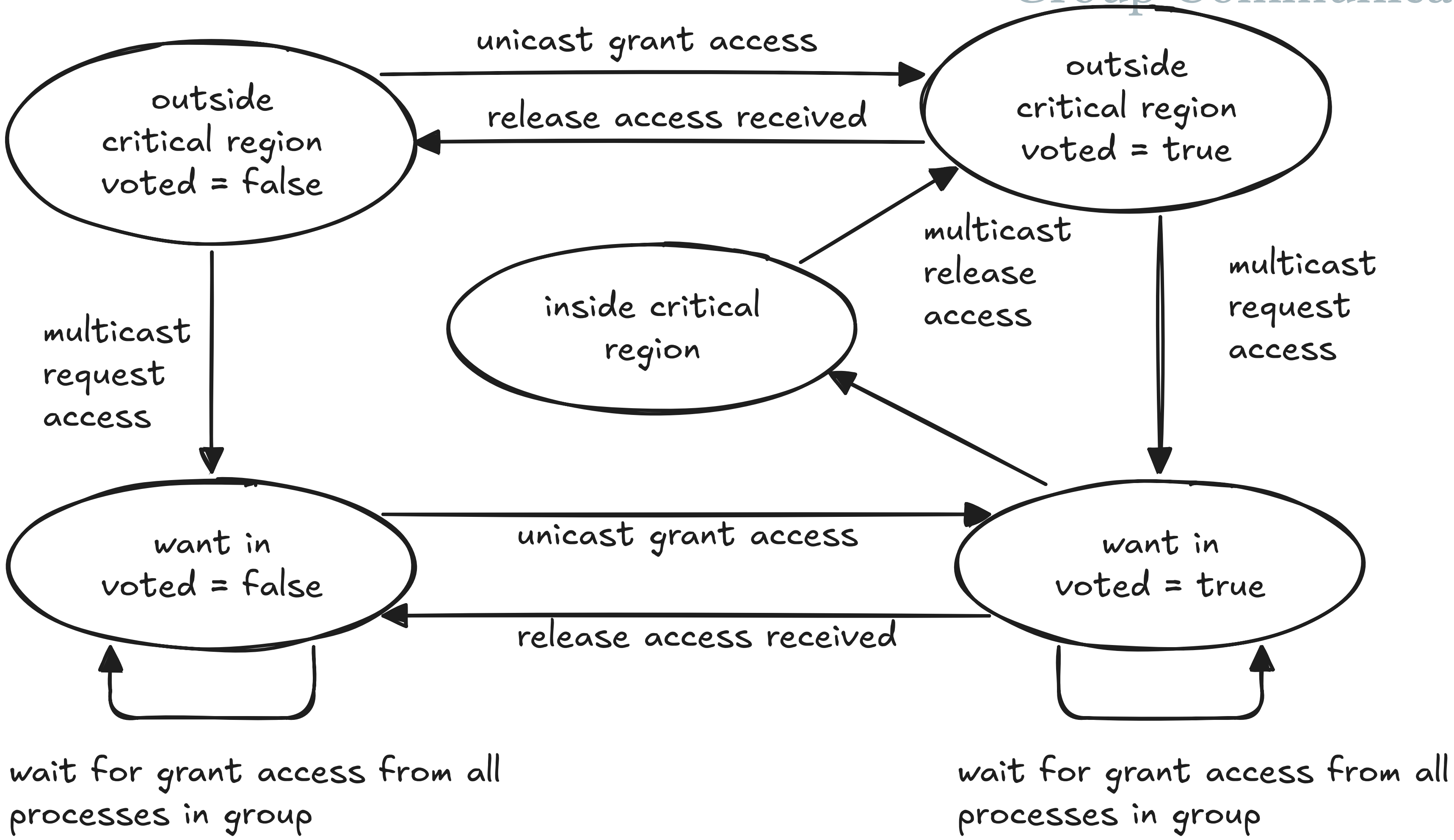
Minimizing the Number of Messages

- **Ensuring Mutual Exclusion**
  - **Groups must intersect**—they are **not mutually exclusive**.
  - This intersection guarantees that **no two processes** can enter the critical region **simultaneously**.
- **Permission by Voting**
  - Access is granted through a **voting mechanism**:
    - A process can access the shared object **if and only if** it has received **permission (votes)** from **all members of its group**.

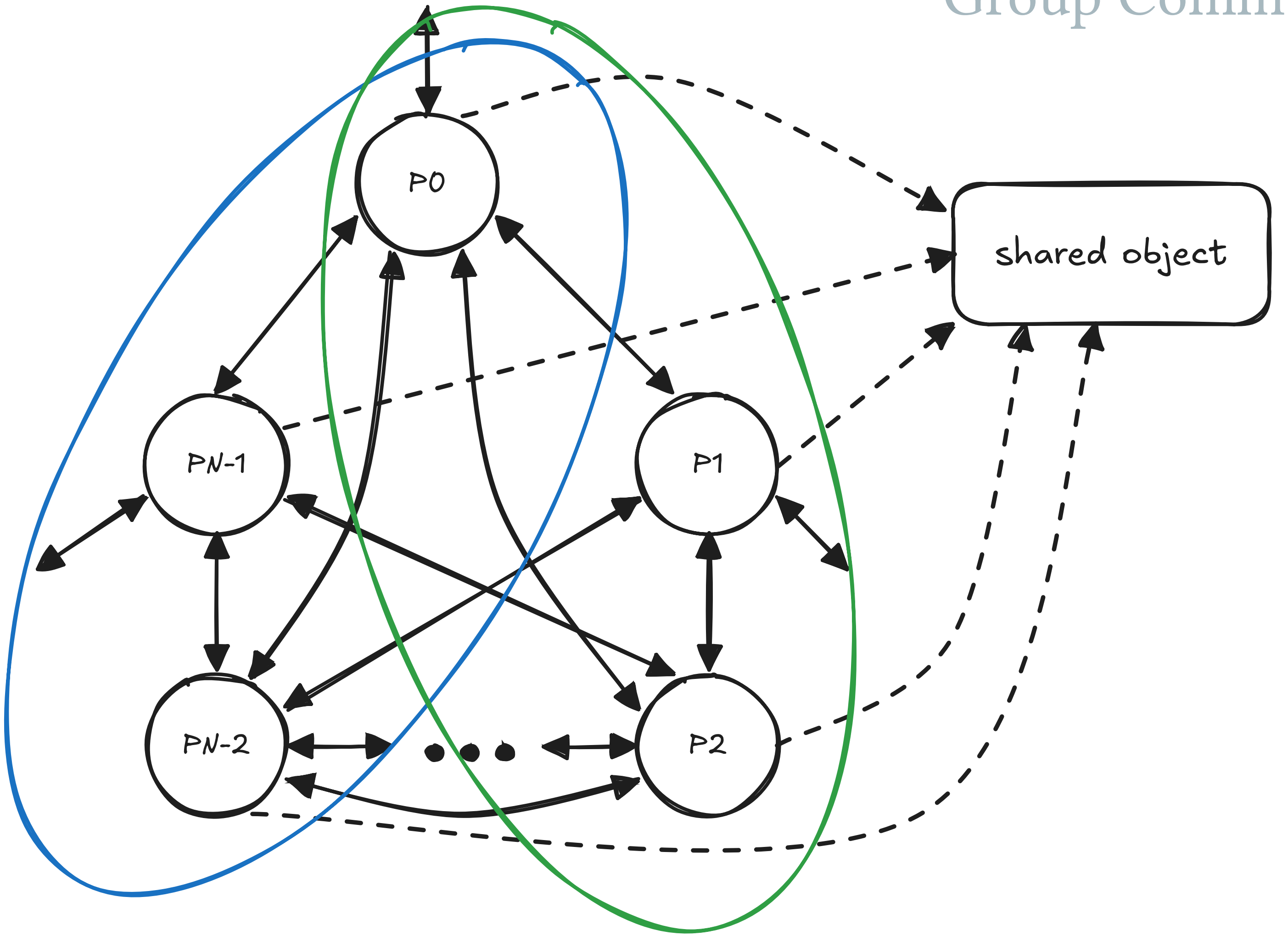
Minimizing the Number of Messages

- **Voting Group Structure**
  - Each process p_i, for $i = 0, 1, \ldots, N-1$, assign a **voting group** V_i.
- **Group Definition Properties**
  - $V_i \subseteq \{p_0, p_1, \ldots, p_{\{N-1\}}\}$
    - Each group is a **subset of all processes**.
  - $p_i \in V_i$
    - Every process is a **member of its own voting group**.
  - $V_i \cap V_j \neq \emptyset \quad \forall 0 \leq i, j < N$
    - **All voting groups must intersect**—ensures **mutual exclusion**.
  - $\#(V_i) \approx \#(V_j) \quad \forall i, j$
    - All groups should be of **approximately equal size** to balance the load.
  - $\exists M \in \mathbb{N}$ such that $\forall i, p_i$ belongs to M groups V_*
    - Each process is a **member of exactly M** voting groups.

Minimizing the Number of Messages



Minimizing the Number of Messages



Minimizing the Number of Messages

- **Voting Group Construction Complexity**
  - Determining the exact composition of each voting group V_i, for $i = 0, 1, \dots, N - 1$, is **non-trivial**.
  - However, a **simple approximation** exists that satisfies:
    - $\#(V_i) \approx \#(V_j)$ (groups are of approximately equal size),
    - $M \approx \sqrt{N}$, with **group membership and size both in** $\mathcal{O}\left(\sqrt{N}\right)$.

Minimizing the Number of Messages

- **Approximate Voting Group Structure**
  - Processes are arranged in a $\sqrt{N} \times \sqrt{N}$ matrix.
  - Each voting group consists of:
    - All processes in a **specific row**,
    - All processes in a **specific column**.
  - This ensures:
    - Every group intersects with every other,
    - Each process belongs to exactly **2 voting groups** (1 row + 1 column),
    - Total message complexity becomes $\mathcal{O}\left(\sqrt{N}\right)$ per access.

Minimizing the Number of Messages

- **Example Structure for Group V_1**
  - $V_1 = \{0, 1, ..., K-1, K+1, ..., (R-1)K+1\}$
  - This includes all processes in **row 1** and **column 1**, skipping the diagonal overlap to avoid duplication.

0	1	...	$K-2$	$K-1$
K	$K+1$	...	$2K-2$	$2K-1$
...	...	...	...	...
$(R-1)K$	$(R-1)K+1$	...	$RK-2$	0

Minimizing the Number of Messages

Algorithm

```
//initialization
state = outsideCR;
voted = false;
//p_i enters the critical region
state = wantIn;
numberOfRequestsGranted = 0;
multicast (requestAccess)to all processes in V_i;
wait until (numberOfRequestsGranted == #(V_i));
state = insideCR;
//p_i exits the critical region
state = outsideCR;
multicast (releaseAccess)to all processes in V_i;
```

Minimizing the Number of Messages


```
//p_i receives an access request message from p_j
if ((state != insideCR) && !voted) {
  id = getId (requestMessage));
  unicast (id, accessGranted);
  voted = true;
} else queueIn (requestQueue, requestMessage);
//pi receives a release access message from pj
if (!empty (requestQueue)){
  id = getId (queueOut (requestQueue));
  unicast (id, accessGranted);
  voted = true;
} else voted = false;
//p_i processes access permissions
numberOfRequestsGranted += 1;
```

Minimizing the Number of Messages

Comments

- **Message Complexity**
  - Each access to the shared object involves $\approx \mathbf{3} \times \mathcal{O}(\sqrt{N}$ messages:
    - **Request**, **grant**, and **release** messages are exchanged within each process's **voting group**.
- **Scalability and Efficiency**
  - The protocol is **highly efficient** for **large process groups**.
    - By reducing the number of participants per access, it significantly lowers the **communication overhead** compared to full peer-to-peer solutions.

Minimizing the Number of Messages

Correctness Issue: Potential Deadlock

- Although **Maekawa's algorithm** reduces message complexity, it is **not deadlock-free**.

Minimizing the Number of Messages

- **Example scenario**:
  - Voting groups:
    - $V_0 = \{0, 1\}$
    - $V_1 = \{1, 2\}$
    - $V_2 = \{2, 0\}$
  - If processes p_0, p_1, p_2 **request access simultaneously**, the following can occur:
    - p_0 receives a vote from p_1,
    - p_1 receives a vote from p_2,
    - p_2 receives a vote from p_0.
  - Each has **one vote**, but **no one receives all votes required** to enter the critical region.
  - The system reaches a **deadlock**: no process can proceed or release its vote.

Minimizing the Number of Messages

- **Solution to Prevent Deadlock**
  - Introduce a **total ordering of requests** to break circular wait conditions:
    - Use **Lamport timestamps** or **logical clocks** to assign a global order to each access request.
    - Each voter grants its vote only to the **request with the earliest timestamp**.
  - Implement a **priority queue** in each process to manage incoming requests by timestamp.
    - When a process finishes its critical section, it **releases its vote** to the **next lowest-timestamped request** in its queue.
  - Alternatively, enforce a **centralized or token-based fallback** mechanism:
    - Use a **coordinator** or **token circulation** in high-contention scenarios to guarantee progress.

Election Procedure

- **Leader Election in Symmetric Process Groups**
  - In some situations, a **single process must be selected** from a group to perform a **specific task at a specific time**.
  - All processes are **conceptually identical** (i.e., peers), so **any process may be selected** in principle.
- **Key Requirements for the Election Procedure**
  - **Termination:**
    - The election must **complete in a finite number of steps**.
  - **Unambiguity:**
    - The outcome must be **unique—only one process** is selected.
  - **Consensus:**
    - **All involved processes must agree** on the elected process.

Election Procedure (Assumptions)

System Assumptions for Election Protocols

- **Fixed Process Group**
  - The **number of processes is fixed** and **known in advance**.
- **Process States**
  - Each process is either:
    - **In execution**, or
    - In **catastrophic failure** (i.e., completely non-responsive or crashed).

Election Procedure (Assumptions)

- **Message Transmission Timing**
  - Message delivery has a **finite upper bound**.
    - Therefore, **timeouts can be defined** and used to detect failures or non-responsiveness.
- **Reliability of Communication**
  - **Messages may be lost**, requiring **fault-tolerant mechanisms** such as retransmission or acknowledgments.

Election Procedure in a Logic Ring

- **Election Initialization**
  - Initially, **no election is in progress**.
    - All processes are in the **`no participant`** state.
  - **Election Trigger**
    - **Any process** may **initiate the election**.
      - It **sets its state to `participant`**.
      - Sends a **`start election`** message to the **next process** in the ring.
      - The message contains its own **process ID**.

Election Procedure in a Logic Ring

- **Handling `start election` Messages**
  - When a proc. receives a `start election` msg, it compares the **msg ID** with its **own ID**:
    - **Case 1:**
      - If the message ID is **less than its own**:
        - **Forwards the message** to the next process.
        - **Becomes a `participant`** (if not already).
    - **Case 2:**
      - If the message ID is **greater than its own** and the process is **not a participant**:
        - **Replaces the message ID** with its own ID.
        - **Forwards the updated message** to the next process.
        - Changes its state to **`participant`**.
      - If it **is already a participant**, it **discards the message** to **reduce unnecessary traffic**.
    - **Case 3:**
      - If the message ID **equals its own ID**:
        - The election is **complete**.
        - The process has been **elected leader**.

Election Procedure in a Logic Ring

- **Leader Announcement**
  - Once a process is **elected as leader**, it sends an **`elected` message** containing its **own ID** to the **next process in the ring**.
- **Handling `elected` Messages**
  - Upon receiving an `elected` message, a proc. does the following:
    - **Reset Participation State**
      - Sets its state to **`no participant`**.
    - **Case 1: Leader ID ≠ Own ID**
      - **Records the leader's ID**.
      - **Forwards the `elected` message** to the next process.
    - **Case 2: Leader ID = Own ID**
      - **Discards the message**, marking the **end of the election process**.

Election Procedure in a Logic Ring

- **Chang and Roberts Algorithm – Failure-Free Assumption**
  - The **original algorithm (1979)** assumes:
    - A **static ring**,
    - No **process failures**,
    - No **message loss**.

Election Procedure in a Logic Ring (Failure and Recovery)

- **Failure Detection**
  - Implement **timeouts**: if a process doesn't receive a response from the next in the ring within a bounded time, assume **catastrophic failure**.
- **Bypassing Failed Processes**
  - Upon detecting a failed neighbor:
    - **Skip over** it and forward the message to the **next reachable alive process** in the ring.
    - Maintain a **locally updated view** of the ring or rely on a **failure detector service**.

Election Procedure in a Logic Ring (Failure and Recovery)

- **Rejoining After Recovery**
  - When a previously failed process recovers:
    - It must **register** or **announce its re-entry** to the group.
    - The ring structure must be **reconstructed** to include the process again.
    - This may require **suspending the current election** and restarting it with the **updated ring topology**.

Election Procedure in a Logic Ring (Failure and Recovery)

- **Reliable Communication Layer**
  - Introduce an **acknowledgment (ACK)** system for every election-related message.
  - If an ACK is not received within the timeout window:
    - **Resend the message** (with a retry limit to avoid infinite loops).

Election Procedure in a Logic Ring (Failure and Recovery)

- **Election Timeout and Restart**
  - If a process **suspects message loss or stalling**:
    - It can **restart the election** after a timeout using a **higher ID** to avoid conflicts.
    - Care must be taken to **avoid concurrent elections** leading to inconsistencies.

Election in an unstructed group

- **Election Initialization**
  - Initially, **no election is in progress**, and all processes are in the **`no participant`** state.
  - **Election Trigger**
    - **Any process** may initiate the election:
      - Sets its state to **`participant`**.
      - Sends a **`start election`** message (with its own ID) to **all processes with lower IDs**.

Election in an unstructed group

- **Handling `start election` Messages**
  - When a process receives a `start election` message:
    - **Replies with an `acknowledge`** message to the sender.
    - If its state is **`no participant`**:
      - Sets its state to **`participant`**.
      - Sends a **`start election`** message (with its own ID) to **all processes with lower IDs**.
- **Handling `acknowledge` Messages**
  - Upon receiving an `acknowledge` message:
    - The sender **waits for an `elected` message** to learn the identity of the leader.

Election in an unstructed group

- **Timeout and Leadership Assumption**
  - If a `participant` process **does not receive any `acknowledge` messages within a predefined timeout**:
    - It assumes it is the **lowest-ID process still alive**.
    - It **declares itself the leader**.
    - Sends an **`elected` message** (with its own ID) to **all processes** to announce the result.

Election in an unstructed group

Garcia-Molina's Election Algorithm

The **original algorithm (1982)**- assumes:

- A **failure-free environment** during the election,
- A **known set of processes**,
- **Reliable communication**.

To extend it to handle **dynamic reconfiguration**, the algorithm must be enhanced as follows:

Election in an unstructured group

Process Failure (Catastrophic Crash)

- **Failure Detection:**
  - Use **timeouts** and **heartbeat mechanisms** to detect non-responsive processes.
  - If a coordinator (leader) fails:
    - A **new election is triggered** by the highest-ID process that detects the failure.
    - Peers with higher IDs are contacted; if none respond, the detecting process elects itself.
- **Failure Propagation:**
  - On detecting a failure, processes **broadcast the updated membership** to maintain group consistency.

Election in an unstructured group

Process Recovery

- A recovered process must:
  - **Announce its return** to the group.
  - **Query for the current leader** (e.g., by sending a “who is leader?” message).
  - If the recovered process has a **higher ID than the current leader**, it may **initiate a new election** (if allowed by policy).

Election in an unstructed group

Handling Message Loss

- **Acknowledgment & Retransmission:**
  - Use **ACKs** for all critical messages (`start election`, `acknowledge`, `elected`).
  - If an ACK is **not received within a timeout**, **retransmit** the original message.
- **Election Recovery:**
  - If a process waits too long for an `elected` message (after sending `start election`) without success:
    - It **retries the election**.
    - To avoid repeated elections, include a **round number or election ID** to track and avoid duplication.

Election in an unstructed group

Additional Enhancements for Robustness

- **Membership Service:**
  - Use a **distributed or centralized membership protocol** to track active nodes.
  - Keeps all processes informed about the **current system configuration**.
- **Persistent State:**
  - Processes may store election state (e.g., participant status, known leader) in **persistent storage** to recover consistently after crashes.

Suggested Reading

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.
- Distributed Systems: Concepts and Design, 4th Edition, Coulouris, Dollimore,

Kindberg, Addison-Wesley