Mutual Exclusion Algorithms for Distributed Systems

Neeraj Mittal

The University of Texas at Dallas

September 24, 2015

Centralized Algorithm

- Coordinator Based Algorithm

Permission Based Algorithms

- Lamport's Algorithm
- Ricart and Agrawala's Algorithm
- Roucairol and Carvalho's Algorithm

Quorum Based Algorithms

- Maekawa's Algorithm

Token Based Algorithms

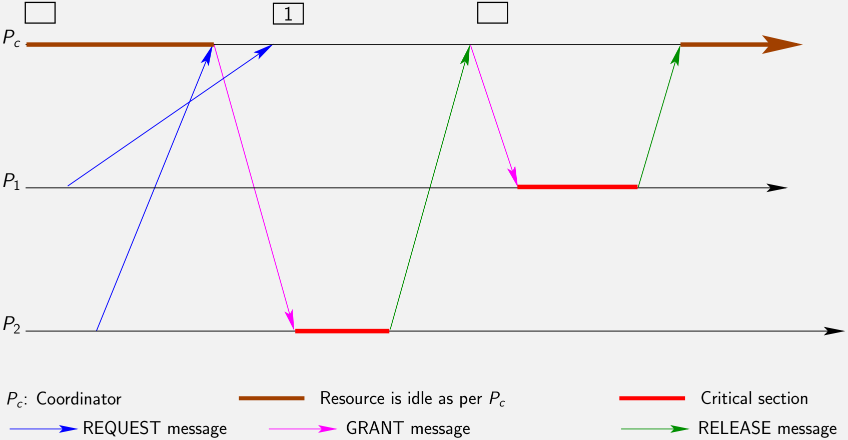
- Raymond's Algorithm

Table of Contents

The Main Idea

- One of the processes in the system is selected as the coordinator.
  - The coordinator is responsible for deciding the order in which critical section requests are fulfilled.
- Every process sends its request for critical section to the coordinator and waits to receive permission from it.
  - Requests are fulfilled in the order in which they arrive at the coordinator.
- The coordinator grants permission to requests one at a time.
  - All other requests are queued in a FIFO queue.

An Illustration



Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
- Message complexity: 3
  - 1 REQUEST message + 1 GRANT message + 1 RELEASE message
- Message-size complexity: $O(1)$
- Response time (under light load): $2T + E$
- Synchronization delay (under heavy load): $2T$

Table of Contents

The Main Idea

- Assumes that all channels are FIFO.
- Processes implement Lamport's logical clock.
- Requests are timestamped using logical clock.
- Requests are fulfilled in the order of their timestamps.
- Each process maintains a priority queue of all requests that are still outstanding as per its knowledge.

Steps for Process P_i

- On generating a critical section request:
  - Insert the request into the priority queue.
  - Broadcast the request to all processes.

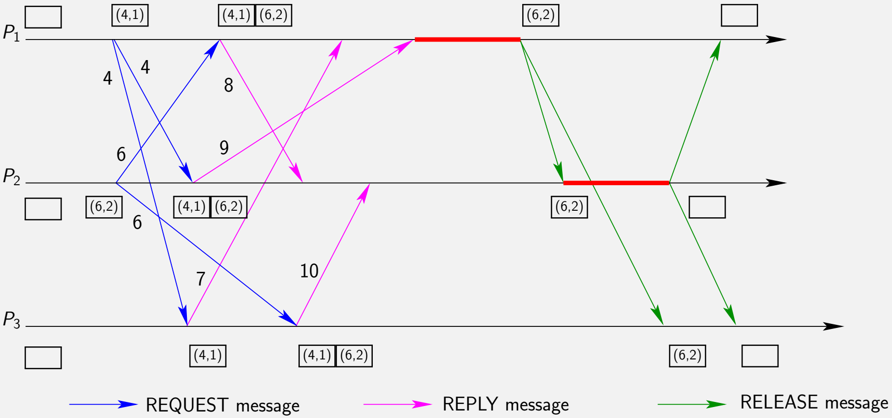
- On receiving a critical section request from another process:
  - Insert the request into the priority queue.
  - Send a REPLY message to the requesting process.

- Conditions for critical section entry (unoptimized version):
  - **L1':** P_i has received a REPLY message from all processes.
    - Any request received by P_i in the future will have timestamp larger than that of P_i's own request.
  - **L2:** P_i's own request is at the top of its queue.
    - P_i's request has the smallest timestamp among all requests received by P_i so far.

Steps for Process P_i (Contd.)

- On leaving the critcal section:
  - Remove the request from the queue.
  - Broadcast a RELEASE message to all processes.
- On receiving a RELEASE message from another process:
  - Remove the request of that process from the queue.

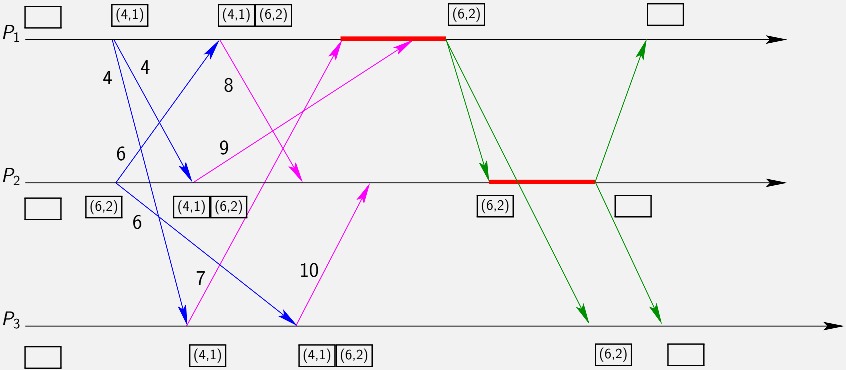
An Illustration



Optimizing the Performance

- Replace L1' with the following condition:
  - **L1:** P_i has received a message with timestamp larger than that of its own request from all processes.
    - Message can be any message: REQUEST, REPLY, RELEASE or even application
- Once L1 becomes true, we can still conclude that:
  - Any request received by P_i in the future will have timestamp larger than that of P_i's own request.

An Illustration with Optimization



Conditions for entry: L1' and L2

Conditions for entry: L1 and L2

Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
- Message complexity: $3(N-1)$
  - $N-1$ REQUEST messages $+$ $N-1$ REPLY messages $+$ $N-1$ RELEASE messages
- Message-size complexity: $O(1)$
- Response time (under light load): $2T+E$
- Synchronization delay (under heavy load): T

Table of Contents

Inefficiencies in Lamport's Algorithm

- Scenario 1: Assume P_i and P_j concurrently generate requests for critical section and P_i's request has smaller timestamp than P_j's request.
  - Lamport's algorithm behavior: P_i first sends a REPLY message to P_j and later sends a RELEASE message to P_j. P_j enters its critical section only after it has received the RELEASE message from P_i.
  - Improvement: P_i's REPLY message can be omitted.

- Scenario 2: P_i generates a request for critical section but P_j does not generate any request for some time.
  - Lamport's algorithm behavior: P_i sends a RELEASE message to P_j on leaving the critical section.
  - Improvement: If P_j generates a critical section request in the future, it will anyway contact P_i via a REQUEST message. So, there is no need for P_i to send a RELEASE message to P_j.

The Main Idea

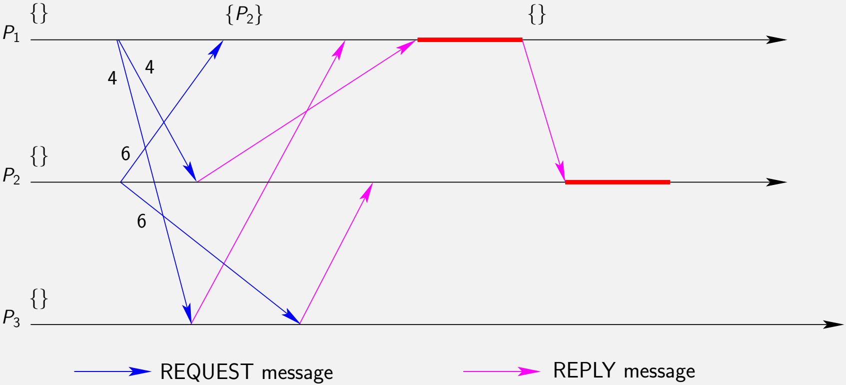
- Combine REPLY and RELEASE messages.
- On leaving the critical section, only send a REPLY/RELEASE message to those processes that have unfulfilled requests for critical section.
- Eliminate priority queue.

Steps for Process P_i

- On generating a critical section request:
  - Broadcast the request to all processes.
- On receiving a critical section request from another process:
  - Send a REPLY message to the requesting process if:
    - P_i has no unfulfilled request of its own, or
    - P_i's unfulfilled request has larger timestamp than that of the received request.

  Otherwise, defer sending the REPLY message.
- Condition for critical section entry:
  - P_i has received a REPLY message from all processes.
- On leaving the critical section:
  - Send all deferred REPLY messages.

An Illustration



Ricart and Agrawala's Algorithm: Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
- Message complexity: $2(N-1)$
  - $N-1$ REQUEST messages + $N-1$ REPLY messages
- Message-size complexity: $O(1)$
- Response time (under light load): $2T+E$
- Synchronization delay (under heavy load): T

Table of Contents

Inefficiency in Ricart and Agrawala's Algorithm

- Every process handles every critical section request.

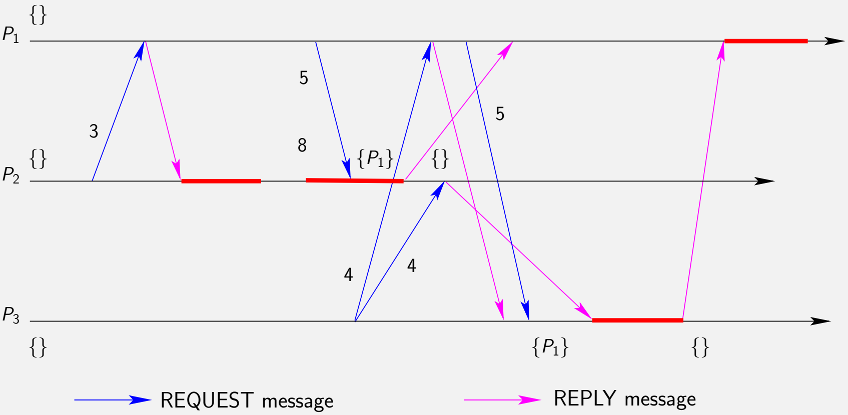
- Objective: Modify the algorithm such that only those processes that are "actively" generating critical section requests are involved in conflict resolution.
  - Processes that do not generate any requests for a long time eventually stop receiving and sending mutual exclusion messages until the time they generate requests.

The Main Idea

- When P_j sends a REPLY message to P_i, it means that P_j is granting permission to P_i to execute its critical section.
- P_i continues to have this permission from P_j until the time P_i sends a REPLY message to P_j.
- Modification to Ricart and Agrawala's Algorithm:
  - To execute its critical section, P_i needs to ask for permission from another process, say P_j, if:
    - P_i has sent a REPLY message to P_j but
    - P_i has not received a REPLY message from P_j since then.

  OR
    - It is P_i's first request and $i > j$.

An Illustration



Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
- Message complexity:
  - Best case: 0
  - Worst case: $2(N-1)$
    - $N-1$ REQUEST messages + $N-1$ REPLY messages
- Message-size complexity: $O(1)$
- Response time (under light load):
  - Best case: E
  - Worst case: $2T + E$
- Synchronization delay (under heavy load): T

Table of Contents

The Main Idea

- Each process is associated with a subset of processes referred to as its quorum.
- A process multicasts its request for critical section only to its quorum members.
- Each quorum member behaves like a coordinator in the centralized algorithm.
  - It grants permission to enter critical section to only one request at a time.
- A process enters its critical section once it has received permission to enter from all its quorum members.

Achieving Mutual Exclusion

- How do we ensure that at most one process is executing its critical section at any time?
  - It is necessary and sufficient to ensure the following intersection property:
    - Quorums of any two processes have at least one process in common.

- Algorithm described so far is prone to deadlocks.

An Illustration of Deadlock

- Three processes: P_1, P_2 and P_3
  - Quorum of $P_1 = \{Q_1, Q_2\}$
  - Quorum of $P_2 = \{Q_2, Q_3\}$
  - Quorum of $P_3 = \{Q_3, Q_1\}$
- All three processes concurrently generate requests for critical section:
  - Q_1 queues P_2's request and grants permission to P_1's request.
  - Q_2 queues P_3's request and grants permission to P_2's request.
  - Q_3 queues P_1's request and grants permission to P_3's request.
- No process is able to enter the critical section and, moreover, the condition persists forever.

Achieving Starvation Freedom

- Uses pre-emption to avoid deadlocks.
  - Quorum members can be viewed as resources that can only be held in mutually exclusive manner.
  - When a quorum member Q grants a process P permission to enter critical section, it means that P has locked Q.
- Processes implement Lamport logical clock.
- Request are assigned timestamps using the logical clock.
- Each quorum member maintains a priority queue of still unfulfilled requests it knows so far.

Achieving Starvation Freedom (Contd.)

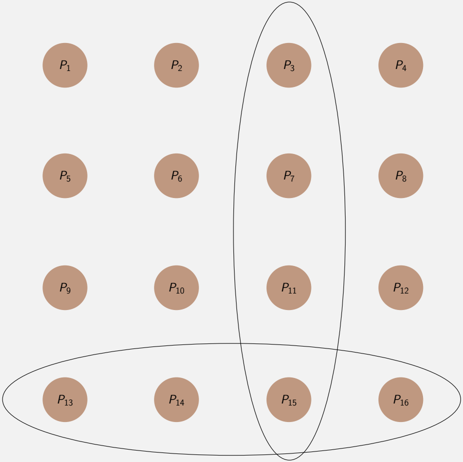
- Suppose a quorum member Q has granted P_i permission to enter its critical section. It then receives a request from P_j.
  - Case 1: P_i's request has smaller timestamp than P_j's request.
    - Q sends a FAILED message to P_j informing P_j that its request cannot be immediately satisfied.
  - Case 2: P_i's request has larger timestamp than P_j's request.
    - Q sends an INQUIRE message to P_i.
    - On receiving the INQUIRE message, P_i relinquishes its lock on Q if P_i has received a FAILED message from another quorum member.

Constructing Quorums

- How to construct quorums of small size that distribute the load uniformly?

- Use grid quorum system:
  - Arrange processes in a square grid.
  - Quorum of a process = all processes in its row + all processes in its column.
  - Size of a quorum $\approx 2\sqrt{N} - 1$.
    - N: Number of processes in the system

An Illustration of Grid Quorum System



Sixteen processes; P_1, P_2, ..., P_{16}

Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
  S: Maximum quorum size
- Message complexity: $7S$
  - At most $2S$ GRANT messages.
  - All other messages are bounded by S each.
- Message-size complexity: $O(1)$
- Response time (under light load): $2T + E$
- Synchronization delay (under heavy load): $2T$ (ignoring deadlock avoidance messages)

Centralized Algorithm

Permission Based Algorithms

Quorum Based Algorithms

Token Based Algorithms

General Idea

- Token based algorithms use a special entity called token.
- There is exactly one token in the system.
- A process can execute its critical section only if it is holding the token.
- Token has to move around to prevent starvation.

Table of Contents

The Main Idea

- Processes are arranged in a tree topology.
- Each process maintains a variable that points in the direction of the token.
  - Basically, the variable contains the identifier of the neighbor that has to be contacted in order to reach the token.
- Each process also maintains a FIFO queue of still unfulfilled requests for token that it has seen so far.

Steps for Process P_i

- On generating a critical section request:
  - If have the token, then enter the critical section. Otherwise:
    - Insert the request into the queue.
    - Send a request for token towards the token holder.
- On receiving a request for token from a neighbor:
  - If have the token and the token is idle, then send the token to the neighbor.
  - If have the token and the token is in use, then insert the request into the queue.
  - If do not have the token, then:
    - Insert the request into the queue.
    - If the queue was empty before, then send a request for token towards the token holder.

Steps for Process P_i (Contd.)

- On receiving the token:
  - Remove the first entry from the queue.
  - If the entry belongs to a neighbor, then send the token to the neighbor.
    Otherwise, enter the critical section.

- On leaving the critical section:
  - If the queue is non-empty, then:
    - Remove the first entry from the queue (should belong to a neighbor).
    - Send the token to the requesting neighbor.
    - If the queue is still non-empty, send request for token to the neighbor to which the token was sent.

Complexity Analysis

- Parameters:
  N: Number of processes in the system
  T: Message transmission time
  E: Critical section execution time
  D: Diameter of the tree
- Message complexity:
  - Best case: 0
  - Worst case: $2D$
    - D REQUEST messages $+$ D TOKEN messages
- Message-size complexity: $O(1)$
- Response time (under light load):
  - Best case: E
  - Worst case: $2D \times T + E$
- Synchronization delay (under heavy load): $D \times T$