DISTRIBUTED SYSTEMS CASE STUDY

Implementation of Maekawa's Algorithm in Go

Team

S.No	Roll.No	Name
1	CB.EN.U4CSE22430	PRIYADARSHINI RAJESH
2	CB.EN.U4CSE22432	PRAHALYAA A
3	CB.EN.U4CSE22460	SAMYUKTHA B M
4	CB.EN.U4CSE22461	B. SHANMUKA VARDHAN

Introduction

Maekawa’s Algorithm is a quorum-based mutual exclusion algorithm designed for distributed systems.
It optimizes resource access by reducing the number of messages exchanged compared to classical centralized or Ricart-Agrawala algorithms.

Key Features:

- Quorum Concept – Each node belongs to multiple quorums, ensuring minimal communication overhead.
- Faster Consensus – Instead of broadcasting to all nodes, a node only communicates with its quorum.
- Scalability – Reduces message complexity to O(√N) per request, making it efficient for large systems.

How It Works:

- Requesting Access – A node sends a request to all members of its quorum.
- Granting Permission – Quorum members decide based on availability.
- Entering Critical Section – The node enters once it has a majority approval.
- Releasing Access – After execution, it notifies quorum members to free resources.

Implementation

This project presents an implementation of Maekawa's mutual exclusion algorithm within a distributed computing environment using the Go programming language.
The implementation includes both a standard version and an optimized variant, showcasing message-based coordination among distributed nodes to achieve mutual exclusion efficiently.

Key Features:

- Dual implementations: standard and optimized
- Message-driven mutual exclusion utilizing request, grant, and release mechanisms
- Heartbeat-based failure detection in the optimized variant
- Quorum-based decision-making to ensure fairness and efficiency
- Configurable simulation parameters for performance evaluation

Algorithmic Workflow

1. Initialization Phase

1. Read the total number of nodes from the config.TotalNodes constant.
2. Create an array nodes to store numNodes instances of the Node struct.
3. Initialize each Node:
    - Assign it a unique ID (i).
    - Retrieve its quorum size from config.Quorums[i].
    - Call node.NewNode(i, quorumSize) to create the node.

2. Setting Up Quorums & Node References

4. For each node:
    - Assign its quorum (a subset of nodes responsible for granting permission to enter the critical section).
    - Provide it with a reference to all other nodes (so they can communicate).

3. Starting Nodes

5. Iterate over the nodes list:
    - Start each node’s message handling logic in a separate Goroutine (go n.Start(&wg)).
    - Use sync.WaitGroup (wg) to synchronize node execution.

4. Critical Section Requests

6.Each node will attempt to enter the critical section after a random delay:

- time.Sleep(utils.RandomDuration()) introduces randomness.
- Log that the node is requesting the critical section.
- Call n.RequestCriticalSection() to begin the request process.

5. Mutual Exclusion Handling (Inside RequestCriticalSection)

7.When a node requests the critical section, it:

- Sends a request to all nodes in its quorum.
- Waits for a majority (quorum) to grant permission.
- Enters the critical section if enough approvals are received.
- After finishing, releases the critical section and notifies other nodes.

6. Synchronization & Termination

8.The sync.WaitGroup ensures that all nodes finish execution before terminating.

9.Once all nodes have completed their operations, the program logs "Simulation finished." and exits.

OPTIMIZATION MADE

IMPROVEMENT	PERFORMANCE IMPACT	DETAILED EXPLANATION
Linked List-Based Queue	O(1) request handling, no array shifting	The old slice-based queue required **O(n) time** to remove elements due to array shifting. The new **linked list implementation** ensures **O(1) enqueue and dequeue**, eliminating unnecessary memory reallocation and improving efficiency.
Pending Requests Map	Avoids duplicate processing, saves memory	Introduces a map (pendingMap) to track requests. This prevents duplicate enqueuing, reducing redundant work and saving memory by keeping only necessary requests in the queue.
Condition Variable (sync.Cond)	Faster response times, no busy-waiting	Replaces CPU-intensive polling with sync.Cond, which allows the process to sleep until it receives enough Grant messages. This leads to lower CPU usage and faster response times.

IMPROVEMENT	PERFORMANCE IMPACT	DETAILED EXPLANATION
Reduced Mutex Locking	Better concurrency, avoids bottlenecks	Locks are now only applied when modifying shared state, instead of wrapping entire functions. This minimizes critical section duration, allowing more concurrent execution and reducing contention.
Efficient Critical Section Entry	Less CPU usage, faster access	Instead of inefficient polling (time.After()), nodes now wait using sync.Cond, which triggers immediate wake-up when conditions are met. This reduces CPU waste and allows faster entry into the critical section.
Reduced Logging Overhead	Improves execution speed	Unnecessary logging operations were removed, especially those inside mutex locks. This reduces I/O latency and improves performance, particularly in large-scale deployments.

CONCLUSION

The optimized Maekawa Algorithm significantly enhances distributed mutual exclusion by introducing efficient request handling, reduced contention, and improved concurrency which leads to :

- More Scalable System
- Faster Response Time
- Improved Concurrency
- Lower CPU & Memory Overhead
- Enhanced System Efficiency