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.

Drawbacks of Maekawa’s Algorithm

- **Deadlock Possibility** – Circular wait conditions can lead to deadlock, requiring additional mechanisms like timeouts or priority-based conflict resolution.
- **Higher Message Complexity** – Requires O(√N) message exchanges per request, making it less efficient than Ricart-Agrawala in smaller systems.
- **More Complex Quorum Management** – Selecting and maintaining quorums is challenging, especially in dynamic systems.
- **Increased Synchronization Overhead** – Each process needs to manage multiple quorum sets, leading to additional processing overhead.
- **Potential Starvation** – A process may be repeatedly denied access if it does not gain majority approval in its quorum.

OPTIMIZATION MADE

IMPROVEMENT	PERFORMANCE IMPACT	DETAILED EXPLANATION
Buffered Channel for Messages	Reduces blocking on message sending	A larger buffer size prevents nodes from blocking while waiting for messages to be processed, improving concurrency.
Linked List-Based Request Queue	Efficient enqueue/dequeue operations	Avoids using slices or arrays, which require costly resizing operations, making request handling faster.
Condition Variable (sync.Cond) for Grant Wait	Reduces CPU usage while waiting	Instead of busy-waiting, the node sleeps until it receives enough Grant messages, improving efficiency.

IMPROVEMENT	PERFORMANCE IMPACT	DETAILED EXPLANATION
Concurrent Heartbeat Monitoring (monitorHeartbeats)	Detects failures with minimal overhead	Uses a separate goroutine with a ticker to periodically check last heartbeat timestamps, reducing blocking operations.
Efficient Mutex Locking in processMessage	Minimizes contention and deadlocks	Each case (Request, Grant, Release, Heartbeat) locks only when needed, avoiding unnecessary serialization of operations.
Deferred Unlocking (defer n.mu.Unlock())	Ensures proper resource release	Prevents potential deadlocks by guaranteeing that the mutex is always unlocked after critical operations.
Optimized Message Logging (utils.Log)	Reduces unnecessary logging overhead	Logs only essential events, avoiding excessive console I/O, which can slow down execution.
Heartbeat Mechanism for Failure Detection	Enhances system reliability	Helps identify failed nodes quickly and efficiently without needing an external failure detection system.

Algorithmic Workflow of the Optimized Code

1. Initialization

- Each Node is created using NewNode(id, quorumSize).
- Initializes the message queue (Incoming), mutex (mu), condition variable (cond), and heartbeat tracking.
- Starts goroutines for heartbeat monitoring (monitorHeartbeats) and sending heartbeats (sendHeartbeats).

2. Node Setup

- SetQuorum(q []int): Defines the quorum (subset of nodes needed for mutual exclusion).
- SetNodes(nodes []*Node): Links the node to all other nodes in the system.

3. Message Handling Loop

- Start(wg *sync.WaitGroup): Continuously listens for messages from Incoming channel.
- Calls processMessage(msg), which directs the message to appropriate handlers based on its type:
  - Request → Calls handleRequest(msg).
  - Grant → Calls handleGrant(msg).
  - Release → Calls handleRelease(msg).
  - Heartbeat → Calls handleHeartbeat(msg).

4. Requesting Critical Section

- RequestCriticalSection():
  - Resets grant counter (grantCount = 0).
  - Sends Request messages to all nodes in the quorum.
  - Waits (using cond.Wait()) until it receives enough Grant messages (grantCount == quorumSize).
  - Calls enterCriticalSection() → Performs operations inside the critical section.
  - Calls releaseCriticalSection() → Releases access by sending Release messages to quorum members.

5. Handling Mutual Exclusion

- **handleRequest(msg):**
  - If the node is not holding the resource, it immediately grants access by sending a Grant message.
  - If the node already holds the resource, it queues the request (waitingQueue.Enqueue(msg.From)).
- **handleGrant(msg):**
  - Increments grantCount.
  - Signals the waiting process if the node has received enough Grant messages.
- **handleRelease(msg):**
  - Marks the node as not holding the resource.
  - Checks the queue and grants access to the next waiting node.

6. Heartbeat Mechanism for Failure Detection

- **sendHeartbeats():**
  - Periodically sends Heartbeat messages to quorum members.
- **handleHeartbeat(msg):**
  - Updates lastHeartbeat time when a Heartbeat message is received.
- **monitorHeartbeats():**
  - Periodically checks the lastHeartbeat timestamps.
  - If a heartbeat is not received within the timeout, logs a failure detection warning.

7. Message Communication

- **sendMessage(msg, target *Node):**
  - Logs the message transmission.
  - Sends the message to the target node via its Incoming channel.
- **msgTypeToString(msgType MessageType):**
  - Converts message types (Request, Grant, Release, Heartbeat) into human-readable strings for logging.

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