DISTRIBUTED SYSTEMS

The Distributed Readers-Writers Problem

26/10/2023

GUILLERMO BAJO LABORDA, 842748@unizar.es

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

1. Introduction

In this practice, we have implemented the generalized Ricart-Agrawala algorithm to solve the problem of accessing the critical section (CS) in a distributed system. In this case, the system implements the readers/writers problem, where all processes have the same responsibility.

We have modified the Ricart-Agrawala algorithm to use vector clocks to establish the "happens-before" relationship between a process requesting access to the critical section and the process receiving the request.

2. Ricart Agrawala Algorithm Implementation

The Ricart-Agrawala algorithm assumes a network in which N distributed processes do not share memory and communicate exclusively through **message passing**. The communication network is error-free, although messages may arrive out of order.
Like all solutions for accessing the **critical section**, the algorithm includes a pre-protocol and a post-protocol:

Pre-protocol

Access is requested by sending a message to all processes. Once all processes have granted access, the requesting process can enter the critical section:
A process will grant access if:

- It does not want to access the critical section (CS).
- It receives a request with a lower timestamp than its own.
- The exclusion matrix indicates that it can grant access without jeopardizing the system's state.

The last condition is not part of the original algorithm but was specifically implemented for our problem. If these conditions are met, the process will immediately send a message to grant access. Otherwise, it will store the request from the process asking for access.

Post-protocol

Once the process leaves the critical section, it will check if it has postponed access for any process and will respond to allow them to enter the critical section.

In our implementation of the algorithm, we used two elements that are not included in the original algorithm: vector clocks and the exclusion matrix.

Vector Clocks

Vector clocks are essential for maintaining a partial order of events in a distributed system, which is crucial for ensuring synchronization of concurrent processes. In this context, vector clocks have been used to establish a total "happens-before" relationship between events in different processes.

The difference between vector clocks and scalar clocks lies in the ability of vector clocks to capture the order of events across multiple processes, allowing them to establish a precise partial order and detect "happens-before" relationships in distributed systems. Scalar clocks, on the other hand, can only provide information about events within a single process and lack the ability to establish a total order between events across different processes.

For the implementation, we used the "GoVector" vector clock library, which allows us to record and compare events across different processes in our system. The clock is updated at various parts of the code, such as when preparing events and receiving messages. When an event occurs or a message is received, the corresponding component in the local vector clock is incremented, reflecting the temporal order of events.

This is where we made the most mistakes. Initially, we did not save the clock we had sent to other processes during the pre-protocol, which caused inconsistencies in the "happens-before" relationship. This happened because every time we received a message, the clock was incremented, causing the process’s clock to change constantly. To fix this, every time we prepared a pre-protocol message, we saved the clock we used to do so, and used it for comparison.

Exclusion Matrix

In our readers/writers system, we have two operations: reading and writing. The read operation does not change the system's state, unlike the write operation. To minimize the time processes spend in the critical section, we only block processes that could cause state inconsistencies. These are writers when another writer is already in the critical section. For all other operation combinations, we allow access to the critical section.
We built an exclusion matrix in which the only excluded operation combination is "Read", "Read".

3. System Description

For N workers, we will launch N/2 writers and N/2 readers. We will not need any process to arbitrate the execution. The distributed readers/writers system is implemented using three distinct modules: the mutual exclusion algorithm (ra), a message exchange system (ms), and a file manager for reading and writing (gf).

The functionality of the mutual exclusion module, ra, has already been detailed. Below, the other two components of the system will be explained.

File Manager

The file manager provides three operations: *LeerFichero(), EscribirFichero(fragmento),* and *New(nom_fich)*. Once we create an instance of gf, we can read and write to the file "nom_fich" using these functions.

Message Exchange System

We use the provided message exchange module to communicate with the other workers. Since the Receive() operation it provides is blocking (i.e., execution does not continue until a message is received), we launch a goroutine to manage the mailbox. This routine uses a separate channel for each type of message to redirect them accordingly.

For the process of sending messages, we used the Send() operation from the module throughout the program, including ra.go. We know this is not ideal, as it ties the implementation to a specific message exchange system. However, given the small scale of the problem and to speed up development, we decided to implement it this way.

In our general system, we will exchange four types of messages: Request, Reply, FileChange, and Barrier. The first two types are used in the Ricart-Agrawala algorithm to request and grant access to the critical section. The last two types are used for exchanging messages between processes. FileChange messages contain changes made to the file by the writer processes and are sent to all processes in the system to update their respective files. The Barrier message is used to start the execution in a synchronized manner.