Practical Assignment 2 - Distributed Systems

The Problem of Distributed Readers and Writers

Authors:

Fernando Pastor Peralta (897113)
Javier Murillo Jimenez (897154)

Delivery date: 16/10/2025

Group: Pair: 1-6

Index

Problem description:

The problem to be solved in this exercise consists of implementing a distributed reader/writer system where multiple processes concurrently access a shared resource (in this case a text file, stored locally on each device), guaranteeing the following:

- Multiple readers can access the file simultaneously.
- Writers require access on a mutually exclusive basis.
- There is no coordinator, so all processes have the same responsibility in the system.
- All local copies must converge to the same state after each write.

To ensure that all of that is fulfilled, the following properties must be met:

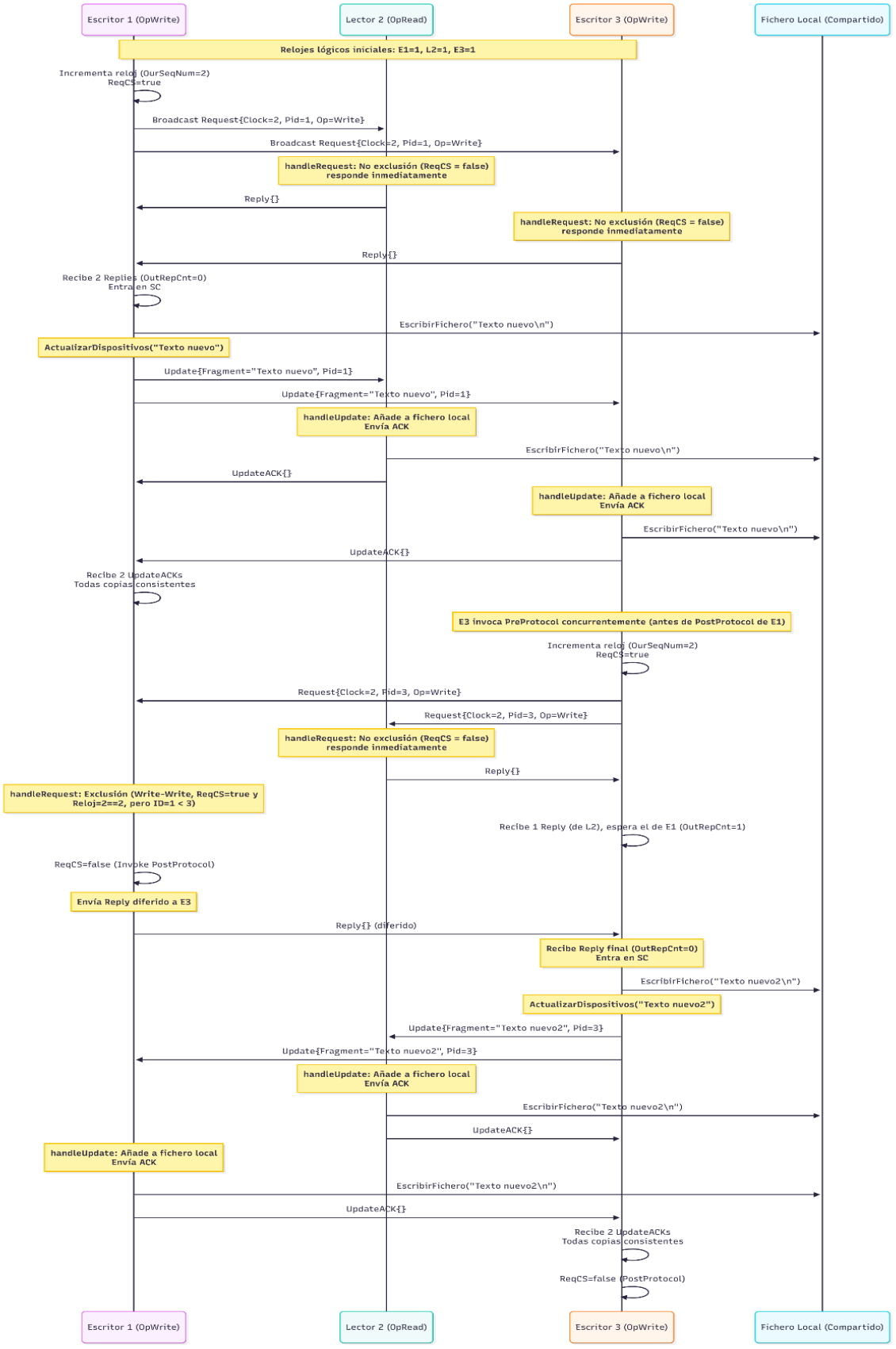
- Mutual exclusion between writers: Two writers cannot simultaneously access the critical section.
- Mutual exclusion between reader and writer: A reader and a writer cannot simultaneously access the critical section.
- Concurrency between readers: Multiple readers can concurrently read the file.

We also guarantee the following liveliness properties:

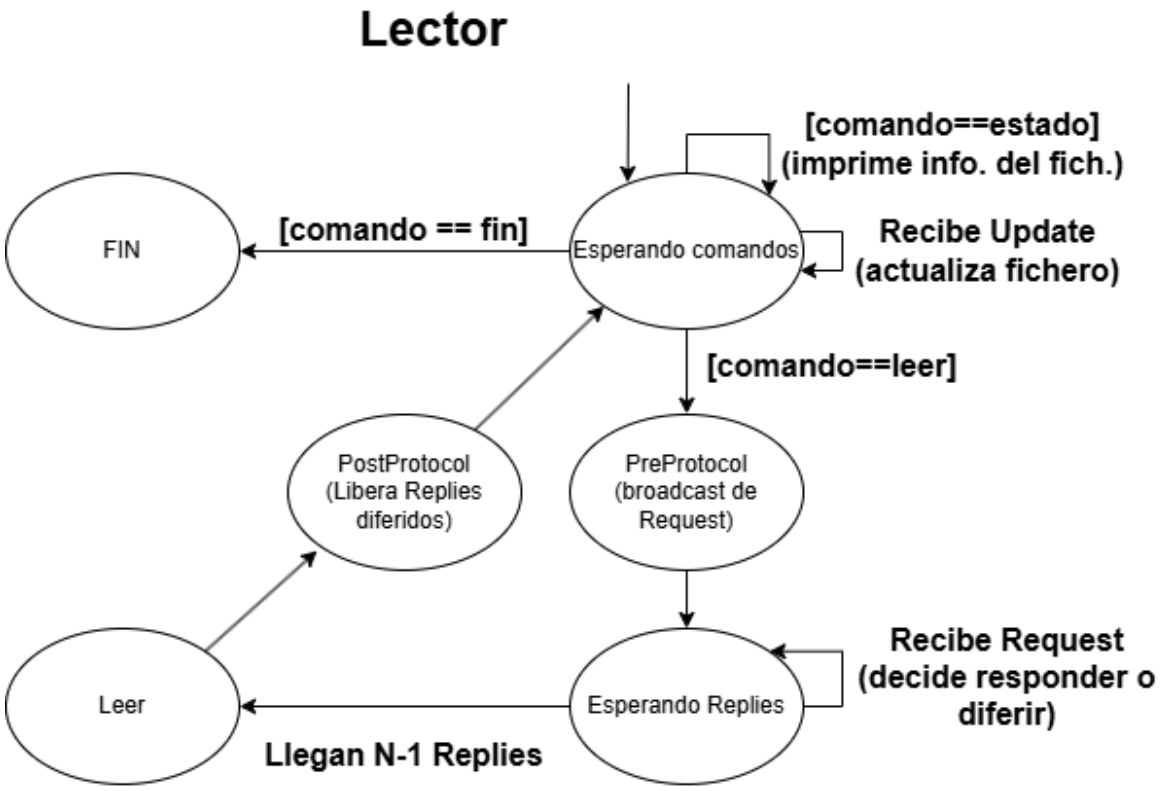
- Absence of starvation: Both readers and writers will eventually gain access to the critical section.
- Absence of deadlock: The system never enters a deadlock.

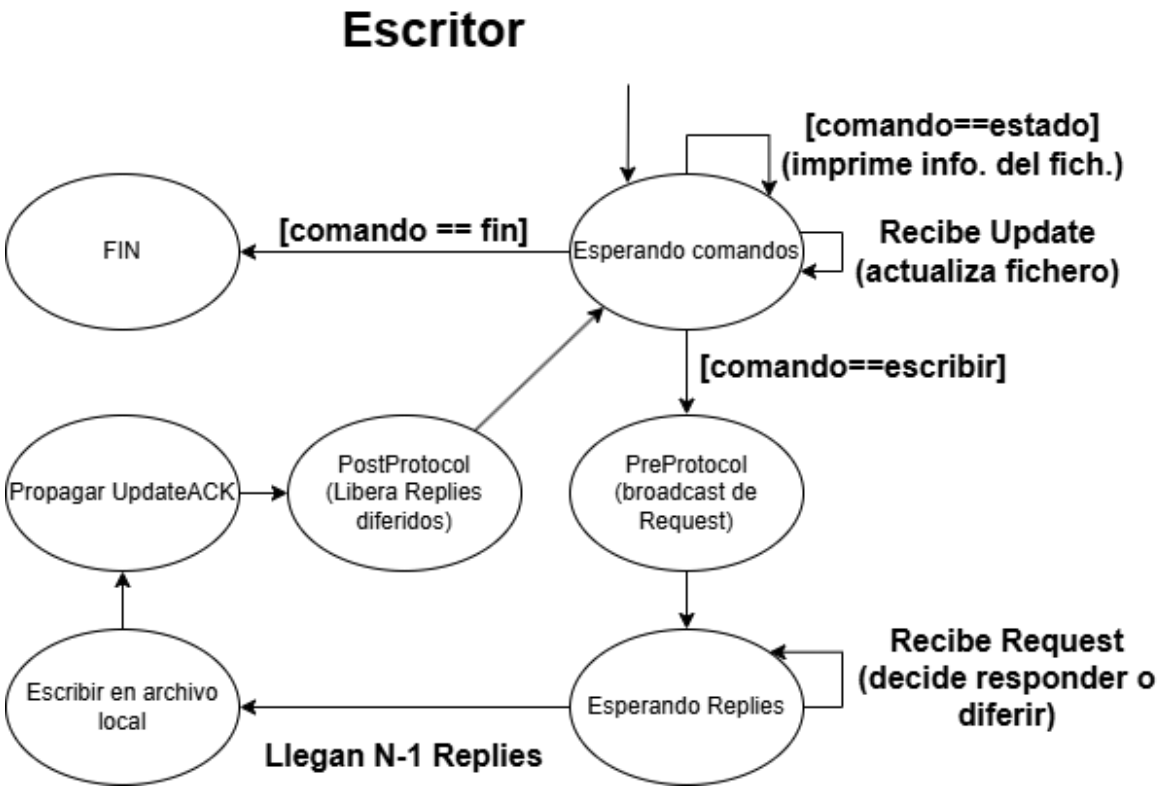
Algorithm design:

Sequence diagram:



State machine:





Implementation:

For the implementation of the algorithm, we have extended the original Ricart-Agrawala algorithm to handle multiple types of operations; for this, we use:

▸ A common database for all distributed processes, based on logical clocks, local process information, booleans to indicate requests to enter the critical section, mutexes, and channels for receiving messages:

```
Go
type RASharedDB struct {
	OurSeqNum  int              // Process sequence number
	HighSeqNum int              // Highest sequence number seen
	OutRepCnt  int              // Number of replies needed to enter the SC
	ReqCS        bool               // Indicates if the process is requesting
access to the SC
	RepDefd    []bool           // For each process, if it is owed a reply
	MS         ms.MessageSystem // Messaging system
	done       chan bool        // To end the processes
	chrep      chan bool        // Channel for receiving replies
	Mutex      sync.Mutex       // Mutex to protect concurrency on variables

	id             int        // Process ID
	MyOp           int        // Current operation (read or write)
	N              int        // Number of processes
	exclude           [2][2]bool // Mutual exclusion matrix between read and
write operations
	chUpdateACK      chan bool   // Channel to receive ACKs from devices to
confirm that they have updated their file
	file_EscLec string     // File in which to write or read
}
```

▸ An exclusion matrix, which is part of the aforementioned database, defines the rules of concurrency between readers and writers:

```
Go
exclude: [2][2]bool{
	/*         read   write*/
	/*read*/  {false, true},
	/*write*/ {true,  true},
}
```

In that matrix we can see that readers do not exclude each other, but in the other cases they do (reader-writer, writer-writer).

▸ Message receiver using a goroutine "receiver", created when the local database was created. It handles receiving messages, identifying them, and processing the information depending on the message type.

▸ Message passing:

```go
type Request struct {
	Clock int //logical clock of the process that sends the message
	Pid   int //ID of the process that sends the message
	On    int //operation (read or write)
}

type Reply struct{}

type Update struct {
	Fragment string
	Pid     int //ID of the process that sends the message
}

type UpdateACK struct{}
```

Types of messages implemented:

- Request for access to the critical section (Clock, Pid, Op).
- Reply for authorization to enter the critical section.
- Update for propagating changes in the file
- UpdateACK for confirmation of the file update (Fragment, Pid).

To request the critical section, we create the PreProtocol, which sets ReqCs to true, requesting the critical section, updates the clock, sends a Request broadcast to indicate that we want to enter the SC, and blocks the process until we receive a Reply from the rest of the devices.

Reply messages are sent automatically if we are not requesting the critical section, or if our operation is not mutually exclusive with the operation being requested, or if our clock indicates a longer time (or the same time but with a lower ID) than the clock of the process that sent the message. Otherwise, the reply is deferred and will only be sent once our PostProtocol is executed, indicating that we have exited our critical section (ReqCS = false).

On the other hand, since we have to update all local files every time we write, we have to make sure that the file update has been performed on all devices before any other device starts writing, otherwise we could have missing or disordered text in the file of some of the distributed devices.
To ensure this, in addition to the Request messages to request access to the critical section and the Reply messages to grant it, we created two more types of messages to ensure the updating of local files on all devices before exiting distributed mutual exclusion. These are: Update, which is sent to all distributed devices to tell them to update their local file with a

specific text, and another message, UpdateACK, which is sent to the device that wrote first. This device waits to receive N-1 UpdateACK messages, since when the last message of this type arrives, we have ensured that all local files are correctly updated.

The UpdateDevices function handles this entire message process for updating local files. It sends Update messages to all devices and waits for N-1 UpdateACK messages. This process is triggered every time a write operation is performed, between the Pre-Protocol and Post-Protocol stages.

▸ Next, the algorithms for the writers and readers were designed, where the databases are created, logs can be enabled, a series of commands that can be executed are shown, and PreProtocol and PostProtocol are called when reading or writing (and UpdateDevices when writing).

Evidence:

For testing the algorithm, a logging system has been created that allows detailed viewing of the system's status at any given time. This system can be activated when calling the reader or writer by adding "log" as the last argument of the execution call. This system, along with the command that can be used at any time to view the status of the shared document ("status"), allows you to know exactly what happened and whether the execution of the action sequences was successful.

Using this method to monitor the system's status, we conducted a series of tests. We added a commented-out 5-second timer after the PreProtocol call, both on the reader and the writer. By commenting and uncommenting this timer, we can see if calls made from other readers or writers are experiencing mutual exclusion as expected, or if there is any unexpected behavior.
The tests we have performed include:

- Checking for the existence of mutual exclusion when one writer tries to write while another writer is writing.
- Checking for the existence of mutual exclusion when a reader tries to read while a writer writes.
- Checking for the existence of mutual exclusion when a writer tries to write while a reader reads.
- Checking for the absence of mutual exclusion when one reader tries to read while another reader is also reading.

After verifying that the tests worked correctly, we can conclude that the system is working properly.