Sistemas Distribuidos - Práctica 2
Problema de los Lectores y Escritores Distribuido

Selivanov Dobrisan, Cristian Andrei - 816456@unizar.es
Wozniak, Dorian Boleslaw - 817570@unizar.es

Índice

1 Introducción

En esta práctica, se trata de implementar un sistema que resuelve el problema de los lectores-escritores de forma distribuida, utilizando un algoritmo de exclusión mútua distribuida y descentralizada, como lo es, el algoritmo de *Ricart-Agrawala*.

El problema consiste en una serie de procesos que pueden caer en dos categorías: los procesos **lectores**, que leen un fichero compartido; y los **escritores**, que escriben en dicho fichero. En su versión de concurrencia local, el problema radica en que los lectores no pueden leer el fichero cuando un escritor está modificando el fichero, y múltiples escritores no deben escribir simultáneamente sobre el mismo archivo. Por otro lado, múltiples lectores pueden acceder para leer el fichero mientras un escritor no desee entrar en sección crítica. Este problema se resuelve normalmente mediante el uso de semáforos binarios.

En la versión distribuida del problema, se añaden ciertas condiciones. Todos los procesos tienen una copia local del fichero compartido. Para poder acceder a la sección crítica se requiere el uso del algoritmo de *Ricart-Agrawala*, donde se utiliza un *Pre Protocolo* para obtener acceso a la sección crítica con el permiso del resto de procesos, y un *Post Protocolo* para salir de este, mandando las respuestas pendientes a otras solicitudes postergadas. Los procesos que no desean entrar en sección crítica responden al solicitante dando permiso para entrar; quienes la desean, responden sólo si no tienen prioridad, determinada si su número de secuencia es menor que el del solicitante (o si su identificador es menor en caso de estar en la misma secuencia), reteniendo la respuesta en caso contrario hasta salir de la sección crítica. Finalmente, si un escritor modifica su fichero, debe mandar las modificaciones realizadas al resto de procesos, y estos deben actualizar su copia.

El sistema de mensajería ya está implementado mediante el módulo *ms*: una implementación parcial del modelo Actor, donde a cada conexión se le asigna un identificador único que se utiliza para los envíos, y un buzón donde cada nodo del sistema recibe los mensajes de otros procesos.

Puesto que los lectores no necesitan retener peticiones de sección crítica de otros lectores incluso si estos mismos quieren entrar en ella, se implementa una matriz operaciones excluyentes, el cual determina dicho comportamiento según el tipo de procesos involucrados. Por otro lado, se sustituye la implementación de relojes escalares por relojes vectoriales, de tal manera que el algoritmo en cuestión no se vea afectado.

El objetivo de esta práctica es diseñar e implementar dicho sistema, haciendo uso de una versión generalizada del algoritmo *Ricart-Agrawala*.

2 Diseño del sistema

2.1 Arquitectura de componentes

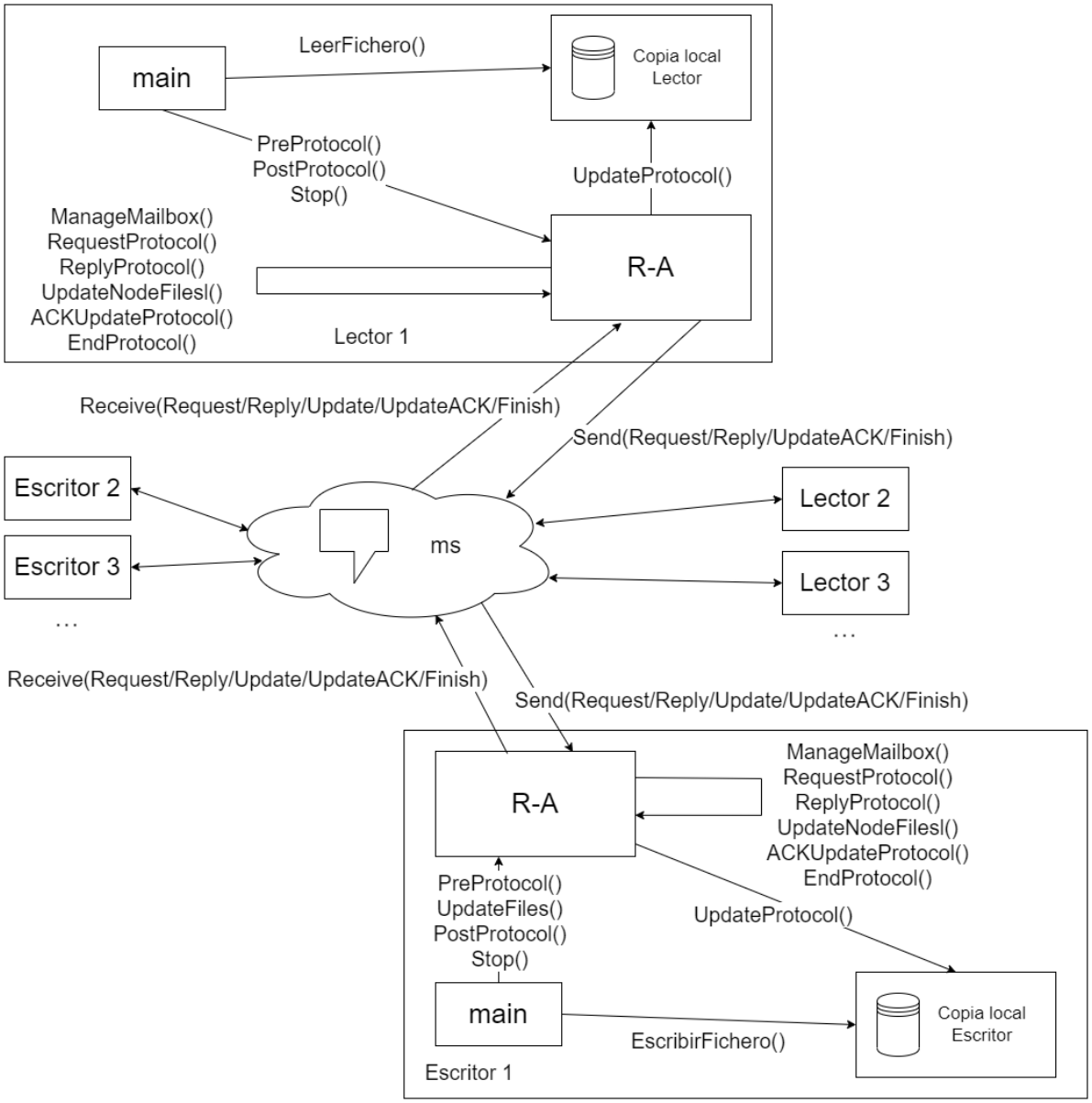

Fig. 1: Diagrama de la arquitectura de los componentes del sistema Lectores-Escritores

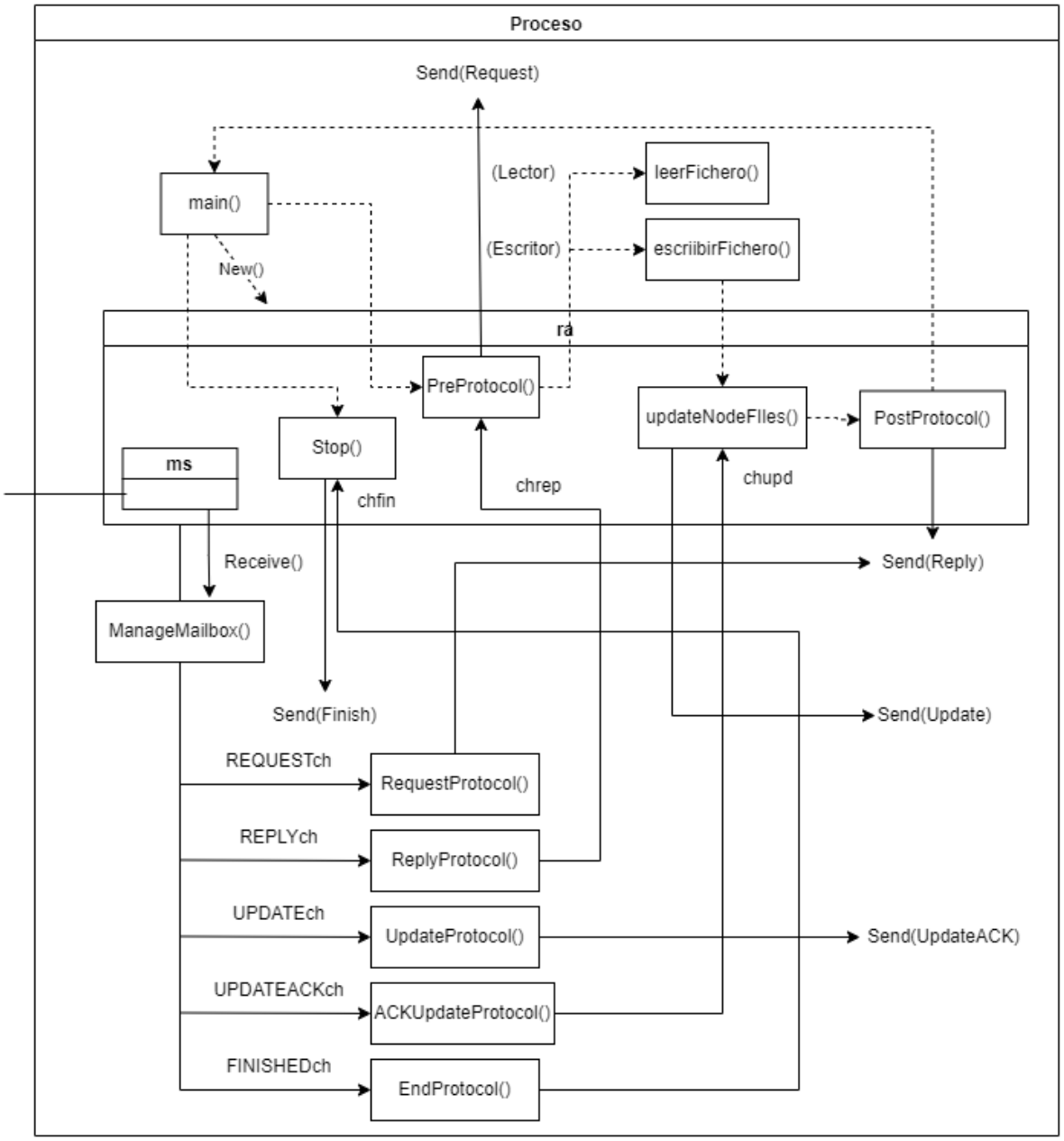

Fig. 2: Diagrama interno de comunicación de un proceso local

2.2 Máquinas de estados

2.2.1 Proceso lector

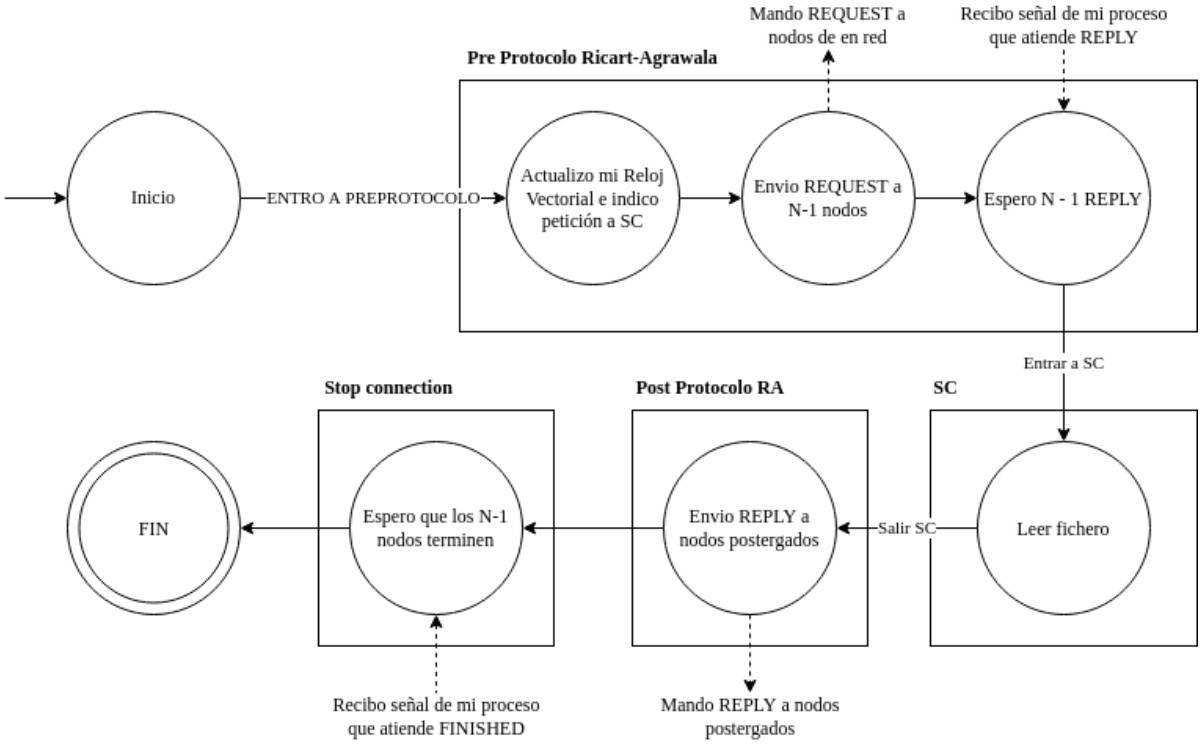

Fig 3. : Máquina de estados de proceso lector que accede solo una vez a SC

El proceso lector tan solo se dedicara a leer el fichero en bucle, el número de iteraciones que desee, pidiendo acceso a SC crítica para ello, y no postergando a los demás procesos lectores con prioridad que también quieran acceder al mismo tiempo. Todo esto tras haber inicializado las variables compartidas junto a los subprocesos concurrentes, a la hora de configurar el buzón del nodo.

Cabe destacar que tal y como se indica en la documentación oficial de *Ricart-Agrawala*, cuando un nodo desea apagarse porque ha concluido su trabajo, deberá notificar a los demás de su finalización, pero siempre a la espera de que estos terminen también, durante esta espera, sus subprocesos seguirán atendiendo las peticiones de los nodos *"activos"* para siempre garantizarles accesos a SC.

Una vez todos los nodos hayan terminado y notificado al actual, se concluirá la ejecución del proceso.

2.2.2 Proceso escritor

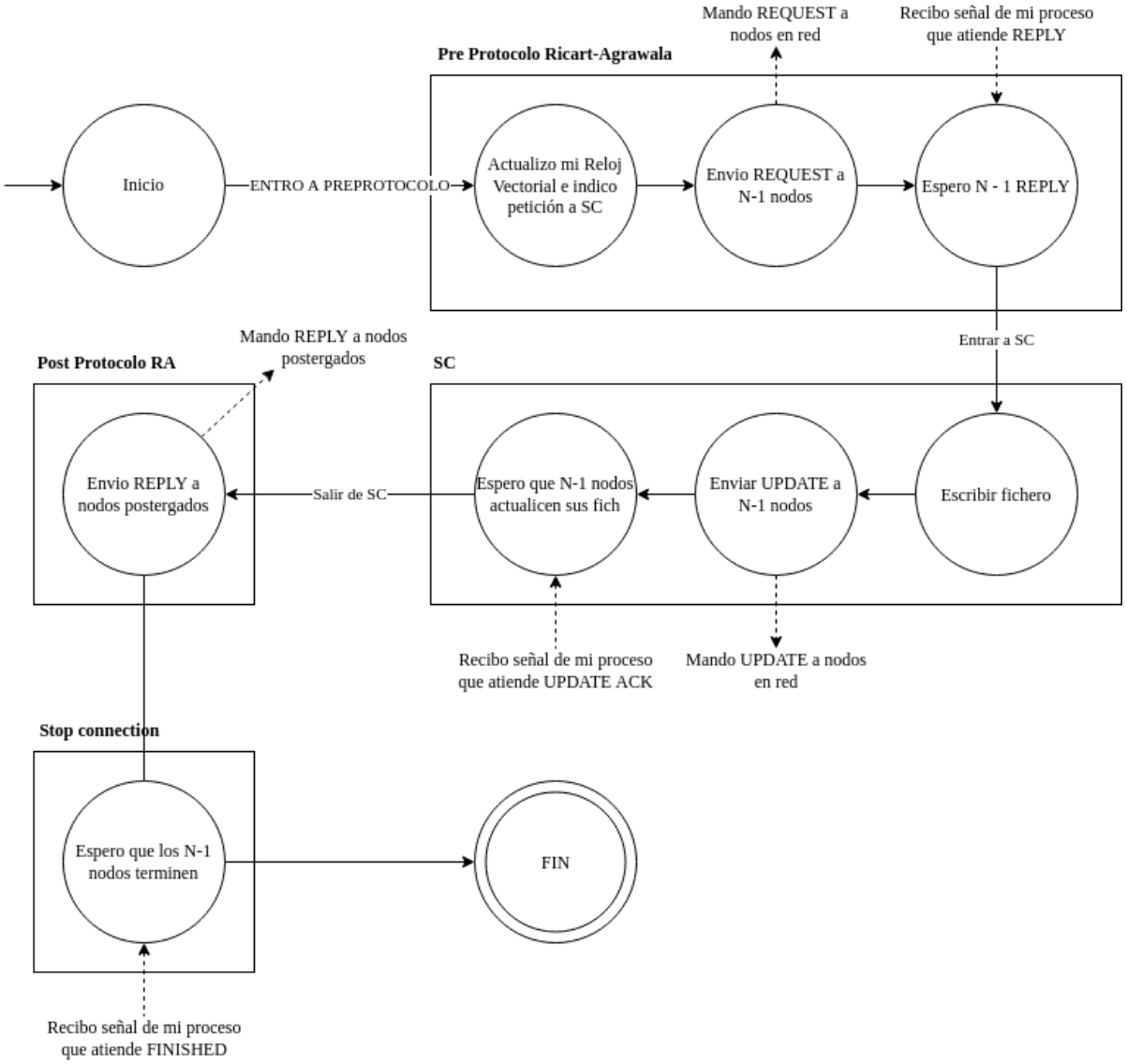

Fig. 4: Máquina de estados de proceso escritor que accede solo una vez a SC

El proceso escritor tan solo se dedicara a escribir en el fichero, el número de veces que desee, un fragmento determinado. Para realizar dicha escritura deberá pedir acceso a *SC*, y una vez dentro, procederá a escribir la cadena de caracteres oportuna. Acto seguido avisará a los demás nodos de la red, notificando los cambios realizados en el fichero, para que estos procedan a actualizarlos. Tras concluir con las alteraciones al fichero, saldrá de *SC* realizando el *Post Protocolo*, al igual que los procesos lectores. Además, el protocolo de finalización controlada será idéntico al de los lectores, al igual que la inicialización de buzón, subprocesos y variables compartidas.

Es importante remarcar que, a diferencia de los procesos lectores, solo puede haber un único proceso escritor en *SC*, ya que sus acciones alteran el dato compartido de forma distribuida, por tanto, no deberá aceptar la petición de ningún tipo de proceso durante su estancia en *SC*.

2.2.3 Subproceso *"ManageMailbox"*

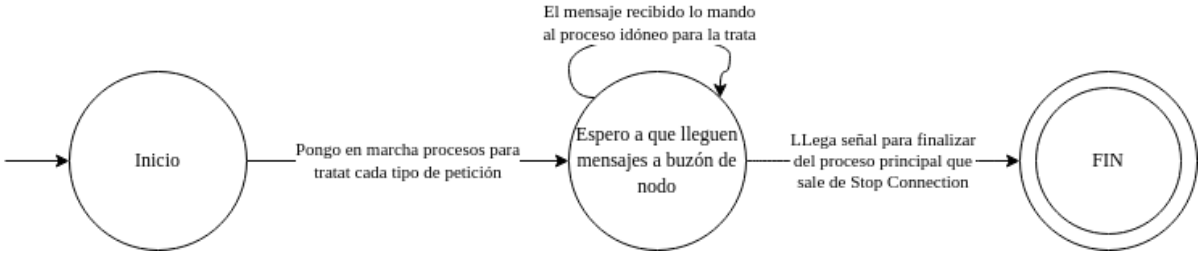

Fig. 5: Máquina de estados de sub-proceso "ManageMailbox" ejecutado al inicializar nodo

El subproceso *"ManageMailbox"* será creado cuando los nodos hayan configurado sus buzones. Su único propósito será el de poner en marcha tantos procesos de trata de mensajes, como tipos de mensajes se compartan, para después ponerse a la escucha del buzón del nodo, remitiendo las notificaciones recibidas a los procesos oportunos.

La secuencia anterior perdurará siempre que el proceso principal, que en nuestro caso sería un escritor / lector, no mande una señal de finalización, lo cual significa que todos los nodos han concluido con su trabajo, de tal manera que ya no hace falta mantener el subproceso en marcha.

2.2.4 Subproceso *"RequestProtocol"*

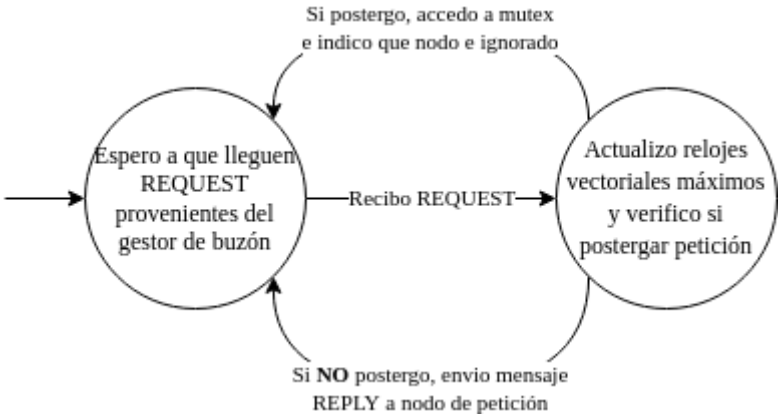

Fig. 6: Máquina de estados de sub-proceso "RequestProtocol"

El subproceso *"RequestProtocol"* es creado por el *"ManageMailbox"*. Su objetivo será tratar las peticiones entrantes al nodo del proceso lector / escritor, actualizando el reloj vectorial máximo y verificando si el nodo solicitante tiene prioridad sobre el nodo del subproceso para acceder a *SC*. Dicha prioridad se dará si el reloj vectorial solicitante es estrictamente menor que el local, o en caso de ser iguales (nodos concurrentes en misma secuencia), si el identificador del nodo solicitante es menor al del subproceso, además se los dos nodos realizar acciones no excluyentes entre sí.

En caso de otorgar prioridad al solicitante, se le responderá inmediatamente mandando un *REPLY* al nodo. En caso contrario, dicha petición será postergada.

2.2.5 Subprocesos *"ReplyProtocol"* y restantes

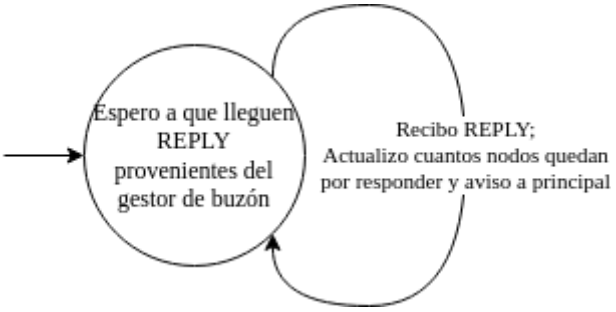

Fig. 7: Máquina de estados de sub-proceso "ReplyProtocol", similar a los demás protocolos restantes

Al igual que los subprocesos *"RequestProtocol"* es creado por el *"ManageMailbox"*. Este subproceso se limitará a recibir los mensajes provenientes del gestor del buzón, actualizando el número de procesos que quedan por responder al nodo local y avisando al proceso principal, por cada respuesta recibida.

Las máquinas de estados de los subprocesos restantes *"UpdateProtocol"*, *"ACKUpdate"* y *"EndProtocol"*, son muy similares al del subproceso encargado de los *REPLY*, diferenciándose únicamente en la acción realizada. Pero por lo general se limitan a hacer una acción y volver al estado de espera.

2.3 Diagramas de secuencia

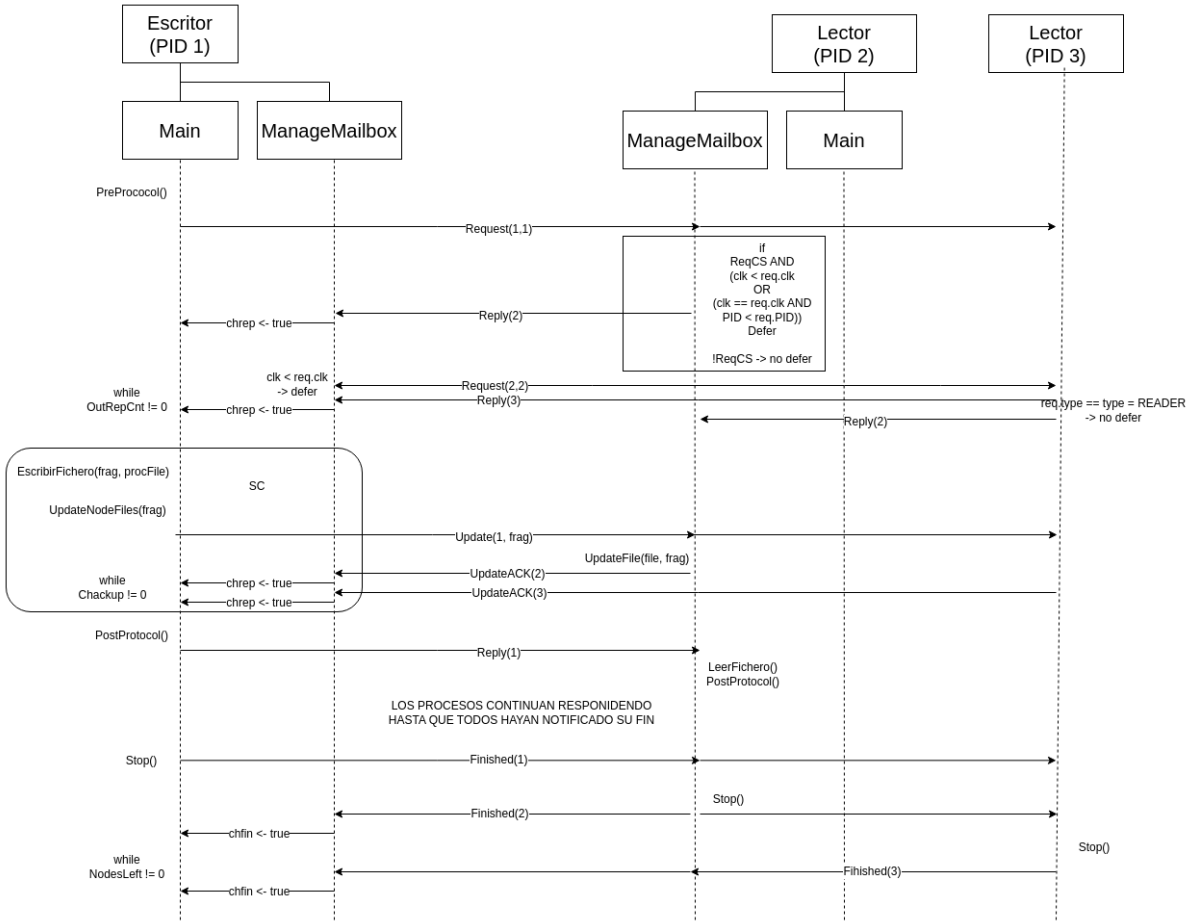

Fig. 8: Diagrama de secuencia para un grupo de tres procesos: dos lectores y un escritor. Notar cómo el proceso 1 posterga la petición de entrada a la sección crítica del proceso 2 al requerir el proceso 1 está y tener prioridad por un reloj menor. Por otro lado, un proceso lector nunca posterga peticiones de SC de otro proceso lector, incluso si está en la SC

3 Implementación del sistema

Tal y como se exige en la asignatura, el sistema distribuido descrito en la introducción se implementará en el lenguaje de programación *GO*, aprovechando los diversos elementos que este ofrece, como los canales compartidos entre procesos locales entre otras cosas.

3.2 Aspectos relevantes

A continuación, se describirán los cambios necesarios a realizar tanto en el algoritmo *Ricart-Agrawala,* como en las fuentes dadas por el profesorado para hacer posible el funcionamiento del sistema, además de los métodos añadidos para conseguir un final controlado del mismo.

3.2.1 Mutex de tipos de operación

Como los procesos que emplean nuestro sistema distribuido serán procesos lectores y escritores, se deberá tener en cuenta las distintas acciones que estos podrán realizar sobre su copia de fichero local coordinado. Por tanto, las acciones de lectura podrán ser ejecutadas concurrentemente, en cambio, las de escritura deberán ser procesadas con exclusión mutua. Esto se implementa mediante una matriz simétrica de booleanos, que permitirá identificar acciones excluyentes entre distintos tipos de procesos a la hora de tratar peticiones.

Dicha matriz aportará información adicional sobre las prioridades de un nodo sobre otro.

```
// Variable compartida dentro del struct RASharedDB
Exclude [2][2]bool // Vector de operaciones excluyentes

...

// Variable que inicializara variable compartida
exc := [2][2]bool{{false, true}, {true, true}}

...

// Aplicación a la hora de verificar prioridad durante el tratamiento de un REQUEST,
// para postergar si hace falta
defer_it := ra.ReqCS && ra.Exclude[ra.NodeType][req.Type] &&
                (lowerThanCLKs(ra.OurSeqNum, req.Clocks) ||
                        (!(lowerThanCLKs(req.Clocks, ra.OurSeqNum)) && req.Pid > ra.Me))
```

3.2.2 Nuevos tipos de mensajes

Como además de las peticiones y respuestas para acceso a *SC*, también se han añadido protocolos de actualización de ficheros y finales controlados, hará falta añadir nuevos tipos de mensajes.

```
// Mensaje para avisar actualización de fichero local
type Update struct {
    Pid       int
    Fragment string
}

// Mensaje para confirmar cambio de fichero local
type UpACK struct {
    Pid int
}

// Mensaje para indicar fin de peticiones de acceso a SC
type Finished struct {
    Pid int
}
```

3.2.3 Cambio de relojes escalares a vectoriales

Se ha modificado el algoritmo de *Ricart-Agrawala* para sustituir el número de secuencia, por relojes vectoriales tal y como pide el enunciado de la práctica. Esto implica cambiar algunas operaciones de actualización de secuencia, donde ahora se modificarán los relojes vectoriales tal y como se ha enseñado en las clases de teoría.

```
// Variables en RASharedDB
HigSeqNum []int // Mayor reloj vectorial visto de cualquier REQUEST recibido
OurSeqNum []int // Reloj vectorial de REQUEST originados desde este nodo

...

// Cómo actualizan en Pre Protocolo
func (ra *RASharedDB) updateClksPre() {
    for i, val := range ra.HigSeqNum {
        if (ra.Me - 1) == i {
                ra.OurSeqNum[i] = val + 1
        } else {
                ra.OurSeqNum[i] = val
        }
    }
}

// Cómo actualizarán máximos durante peticiones
func (ra *RASharedDB) updateClksReq(clks []int) {
    for i, val := range ra.HigSeqNum {
        if val < clks[i] {
```

```
                ra.HigSeqNum[i] = clks[i]
        }
    }
}
```

3.2.4 Implementación para final controlado

Al igual que el mecanismo de espera de respuestas para acceder a *SC* del algoritmo *Ricart-Agrawala*, simplemente se ha implementado una barrera donde se bloquean los procesos principales hasta que todos quieran salir. Pero tal y como se ha indicado anteriormente, deberán seguir atendiendo las peticiones de los nodos que todavía quieran entrar a *SC*.

```
// En función STOP

// Se avisa del final a los demás nodos
    for j := 0; j < ra.N; j++ {
        if j+1 != ra.Me {
                ra.Ms.Send(j+1, Finished{Pid: ra.Me})
        }
    }

    // Bloqueo de nodo hasta que todos terminen, sigue atendiendo REQUEST para
    // dejarles acceder a SC
    for ra.NodesLeft != 0 {
        <-ra.Chfin
    }

// Cómo se tratan dichos mensajes de fin
for {
        eoc := <-Finch
        ra.NodesLeft--
        log.Printf("[%d] Nodo %d concluye, quedan %d nodos que acceden a SC\n",
                ra.Me, eoc.Pid, ra.NodesLeft)

        ra.Chfin <- true
    }
```