Práctica 2:
SISTEMAS DISTRIBUIDOS

LECTORES Y ESCRITORES DISTRIBUIDOS

Jorge Hernández Aznar (872838) y Paula Blasco Díaz (874320)

ÍNDICE

EJERCICIO 1

El problema planteado para esta práctica número dos es diseñar, implementar y verificar la validez del algoritmo de ricart-agrawala con relojes vectoriales, generalizado para el problema de lectores y escritores.

MODIFICACIONES ALGORITMO RICART-AGRAWALA

Inicialmente, para implementar el algoritmo Ricart Agrawala con relojes vectoriales, en vez de con relojes lógicos se requería una estructura de datos auxiliar para su representación. Para ello se hizo uso de la biblioteca *GoVec* que permite trabajar con un tipo de datos *vClock* que representan los relojes vectoriales y ofrece ciertas operaciones para su uso.

A continuación sustituimos las variables OurSecuenceNumber y MaxSequenceNumber que originalmente eran enteros (relojes lógicos) por el tipo de datos vClock, siendo *VClock* y *VClockMax* respectivamente . En el *Preprotocol()* se aumenta el reloj de *VClockMax* mediante el método *Tick(pid)* y se asigna a *VClock* el valor del reloj máximo recién modificado haciendo uso de la función *Copy()*.

Asimismo para que el algoritmo RA este generalizado para el problema de lectores y escritores diseñamos una matriz de exclusión en la que dependiendo del tipo de proceso que quiera acceder a SC puede o no. Para ello también se añade un string en la base de datos para identificar qué tipo es cada proceso. De esta forma implementamos el tipo *Pair* que se conforma por dos strings, y la matriz de exclusión forma parte de los datos compartidos de RA siendo un map con clave *Pair* y el valor *Bool*. Dando como resultado las siguientes combinaciones:

```
ra.Exclusion[Pair{"Reader", "Reader"}] = false
ra.Exclusion[Pair{"Reader", "Writer"}] = true
ra.Exclusion[Pair{"Writer", "Reader"}] = true
ra.Exclusion[Pair{"Writer", "Writer"}] = true
```

De esta manera si quieren acceder dos lectores no ofrece problema ya que no están modificando los datos, y ambos pueden acceder a sección crítica. En el tipo de mensaje *Request* se envía el pid, el reloj y el tipo de proceso, “Reader” o “Writer”.

Como resultado el proceso que se encarga de recibir las Requests hace *Tick* del *VClockMax* y llama al método *Merge()* para quedarse con los valores máximos de reloj entre el suyo máximo y el recibido. Asimismo se implementó la función *happensBefore* que verifica que un reloj sea anterior estrictamente a otro. Para ello se hace uso de la función *Compare()* que verifica que todas las componentes de un reloj sean anteriores a las de otro. Y en caso contrario se comparan los PIDs de los procesos para desempatar posibles situaciones de concurrencia.

Finalmente la condición de *deferIt* de una solicitud de acceso a SC se reformula, para que en caso de que solo quieran acceder a SC procesos lectores quede permitido.

```
deferIt := ra.ReqCS && happensBefore(ra.VClock, msg.Clock, ra.Me, msg.Pid)
&& ra.Exclusion[Pair{ra.Op, msg.Op}]
```

Por otro lado, para llevar a cabo los módulos de lectores y escritores, se implementó inicialmente un módulo barrier, similar a la práctica 1 pero simplificada y adaptada para ser usada por el Message System proporcionado. Se requiere de una barrera debido a que si unos procesos comienzan a mandar requests para acceder a SC, antes de que otros tengan activo el proceso que las recibe, se pueden perder mensajes en ese intercambio y puede haber inanición a un proceso.

Finalmente el esquema de los procesos escritores y lectores es muy similar. En ambos inicialmente se crea el fichero fichero*Pid*.txt sobre el que realizarán las lecturas y escrituras. A continuación se crea el msg system en el que se añaden los tipos de mensajes *Reply, Upgrade* y *Barrier*. El tipo *Barrier* es usado para la comunicación con el proceso barrera. Los otros dos tipos son para actualizar los ficheros cuando un proceso escritor modifica su contenido.

Hay que destacar la creación del módulo msgManager para lidiar con la recepción de mensajes por parte de estos procesos. En él se implementa la función *msgManage()* que mediante canales se comunica con el proceso lector o escritor correspondiente. Se manda true al canal *okBarrier* cuando se recibe la confirmación de la barrera. También actualiza el fichero cuando llegan los mensajes *Upgrade*. Por último, manda la confirmación al canal *okUpgrade* cuando todos los procesos han actualizado sus ficheros.

Tras invocar al constructor del modulo RA, se ejecuta en bucle el PreProtocolo y PostProcolo para acceder a SC. En el caso de los lectores se lee el contenido del fichero y se muestra por pantalla. En el caso contrario de los escritores, se escribe el fichero, se mandan los N - 1 mensajes de *Upgrade* y se espera la confirmación por parte del resto de procesos.

PRUEBAS DE VERIFICACIÓN

Para comprobar la validez del algoritmo implementado se realizaron varias pruebas en las máquinas asignadas de las raspberries.

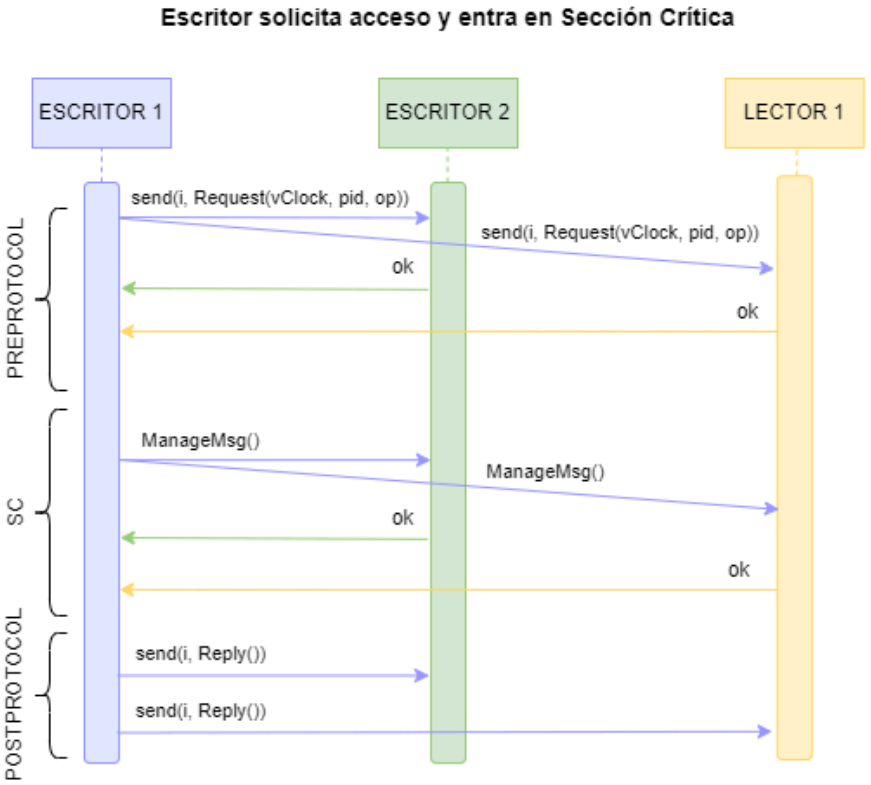
Primeramente se ejecutaron dos lectores en las máquinas 192.169.3.5 y 192.168.3.6, un escritor en la 192.168.3.7 y se destinó la 192.168.3.8 para la barrera. En esta prueba se pudo ver perfectamente que dos lectores podían acceder a SC sin problemas aunque si el escritor solicitaba acceder y tenía un reloj previo su petición era correspondida de forma más temprana. Del mismo modo comprobamos que los ficheros resultantes en todas las máquinas eran iguales.

A continuación hicimos una segunda prueba en la que verificamos la misma situación anterior aunque ahora sustituyendo un lector por un escritor dando lugar a dos escritores y un lector. En ella se pudo observar la ausencia de inanición ya que ambos procesos escritores podían acceder a SC de manera proporcional.

Este diseño también asegura ausencia de bloqueos ya que es imposible que todos los procesos estén esperando a entrar en SC debido a que siempre hay uno que tiene un reloj anterior o un pid inferior al resto y difiera las peticiones de otros procesos para acceder el mismo a SC, y antes de volver a solicitar entrar conduce a los otros procesos el acceso a SC.

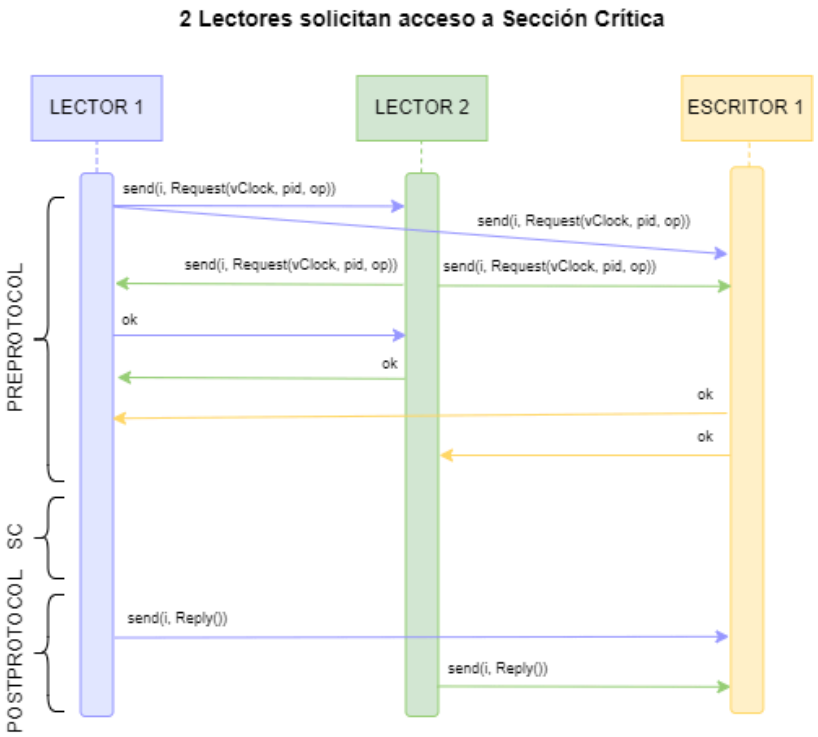
DIAGRAMAS DE SECUENCIAS Y DE ESTADOS

En este primer diagrama de secuencia se ha querido representar la situación en la que solamente un escritor solicita acceso y entra en sección crítica, bloqueando al resto de procesos. Como se muestra en la imagen, el escritor no muestra ningún problema para entrar ya que es el único que quiere acceder. Cuando entra en Sección Crítica, modifica el fichero y manda mensajes para que el resto de procesos también lo hagan, esperando a que estos le envíen la confirmación de que han recibido los cambios.



El segundo diagrama representa la situación en la que dos lectores quieren acceder a la sección critica simultáneamente. Esto es posible dado que sus operaciones no son excluyentes, tal y como se indica en la matriz de exclusión:

```
ra.Exclusion[Pair{"Reader", "Reader"}] = false
```



Por tanto, a partir de los diagramas anteriores, hemos obtenido las máquinas o diagramas de estados de tanto los lectores como los escritores.