30221/39521 - Sistemas Distribuidos
**Universidad de Zaragoza, curso 2023/2024**

# Práctica 2: **Problema de los Lectores y Escritores Distribuido**

## Objetivos y requisitos

### Objetivos

Esta práctica tiene como objetivo la implementación del problema de los lectores / escritores en distribuido, de manera que no exista ningún proceso coordinador, sino que todos los procesos participantes tengan la misma responsabilidad en el sistema. En particular estos son los objetivos de la práctica:

- Implementar en Go el algoritmo de Ricart-Agrawala generalizado para la resolución del problema de Lectores y Escritores distribuido.
- Familiarizarse con las pruebas en un sistemas distribuido.

### Requisitos

- Go versión >= 1.18.1 linux/amd64[1]
- El protocolo *ssh* para ejecutar comandos remotos
- El esqueleto de ćodigo fuente de apoyo que se adjunta a este guión.
- El algoritmo original de Ricart-Agrawala implementado en Algol [2]
- Utilización de la herramienta GoVector para logging de sistemas distribuidos [3]
- Utilización de la herramienta de visualización de logs ShiViz. [4]

[1] Disponible en `https://golang.org/doc/install`
[2] `https://dl.acm.org/citation.cfm?id=358537`
[3] `https://github.com/DistributedClocks/GoVector`
[4] `https://bestchai.bitbucket.io/shiviz/`

## Ejercicio 1.

Diseñar (diagramas de secuencia) e Implementar (en Go) el sistema de lectores y escritores descrito en este guion, utilizando el Algoritmo de Ricart-Agrawala Generalizado:

- Modificar el Algoritmo de Ricart-Agrawala para que funcione con relojes vectoriales.
- Generalizar el Algoritmo de Ricart-Agrawala para que permita implementar patrones de concurrencia más complejos, tales como lectores-escritores.
- implementar una aplicación de lectores y escritores (según se describe en el guion) que acceden a sus ficheros de texto utilizando el Algoritmo de Ricart-Agrawala generalizado, para garantizar que el contenido de todos los ficheros de texto, tras una escritura, siempre sea el mismo. **Se recomienda realizar una traducción del código en Algol y no fundamentar exclusivamente la implementación en las Figura 1. El código de la Figura 1 podría tener condiciones de carrera.**
- Además, la implementación tiene que utilizar el módulo *ms* proporcionado para implementar la comunicación entre procesos. Así como seguir el esquema y las estructuras de datos planteadas en el fichero ra.go (fichero donde se debe implementar el algoritmo de Ricart-Agrawala Generalizado).
- Finalmente, deberéis redactar una memoria donde deberéis describir cómo habéis diseñado el Algoritmo de Ricart-Agrawala (arquitecura de componentes, máquinas de estado y/o redes de petri, y diagramas de secuencia) y aspectos relevantes de su implementación en Go.

*Entrega: código fuente completo del modulo "practica2" y memoria*

## Ejercicio 2.

**OPCIONAL**

Diseñar (diagramas de secuencia) e Implementar (en Go) programas de prueba con varios lectores / escritores para probar el Ejercicio 1. Para ello, tendréis que utilizar un servicio de logging distribuido: las herramientas GoVector y ShiViz, así como la funcionalidad de testing de Go. Adicionalmente, deberéis añadirlo en la memoria donde deberéis describir cómo habéis implementado el Algoritmo de Ricart-Agrawala en Go y cómo habéis probado la corrección de vuestro sistema (distinto número de participantes, baterías de pruebas, etc.).

*Entrega: ficheros de prueba y memoria*

# 1. Aplicación de lectores y escritores

En esta práctica diseñaremos una aplicación distribuida que consiste en N procesos lectores y M procesos escritores. Tanto los lectores como los escritores acceden a un fichero de texto, para mejorar las prestaciones, todos los procesos tienen una copia del fichero. Múltiples procesos lectores pueden acceder a su fichero en lectura simultáneamente. No obstante, a veces un proceso escritor puede escribir. La operación de escritura se tiene que hacer de tal forma que el proceso escritor escribe en su fichero, pero también actualiza los ficheros de todos los procesos lectores y escritores, de manera que todas las copias serán iguales.

Para ello, cada fichero de texto tendrá un proceso gestor con dos operaciones:

- func LeerFichero() string
- func EscribirFichero(fragmento string)

La operación LeerFichero devuelve el contenido completo del fichero de texto. Mientras que la operación EscribirFichero añade al final del fichero de texto un fragmento. Se pueden implementar las variantes que se deseen de estas dos operaciones, por ejemplo, que la operación de lectura del fichero devuelva solo una parte del contenido del fichero o que la operación de escritura permita escribir en cualquier posición.

Cada proceso lector o escritor, contará con un fichero y, por tanto, con un proceso gestor que proporcionará las operaciones de lectura y escritura a través del módulo *ms*.

Para coordinar a los lectores y a los escritores, una solución centralizada podría usar un secuenciador, que nos diera un número de secuencia incremental y globalmente único para cada mensaje. Un proceso no podría enviar su mensaje hasta que no le hubieran llegado todos los mensajes con un número de secuencia menor. Sin embargo, la solución no sería escalable. Una posible alternativa escalable consistiría en utilizar el algoritmo de Ricart- Agrawala generalizado para lectores y escritores. El algoritmo original de Ricart-Agrawala [1] es una implementación del mutex distribuido y puede verse en la Figura 1. Un proceso distribuido que desee acceder a la sección crítica tiene que ejecutar el preprotocol, realizado por las instrucciones (1) a (6). El preprotocol para un Proceso $P_i$ consiste en enviar a los N - 1 procesos distribuidos una petición de acceso a la sección crítica. Cuando se reciba la respuesta en (5) de los N – 1 entonces el Proceso $P_i$ accede a la sección crítica. Una vez terminado el acceso a la sección crítica, el Proceso $P_i$ realiza el postprotocol, instrucciones (7)-(9) donde se envía el ACK a los procesos cuyas peticiones habían sido postergadas. El Proceso $P_i$, mientras está ejecutando el preprotocol, acceso a la sección crítica, postprotocol y sección no crítica (si la hubiera), tiene que atender simultáneamente las peticiones de otros procesos $P_j$. Por ello cada proceso $P_i$ es consta de al menos 2 procesos concurrente, que ejecutan las instrucciones (10)-(15). **La esencia del algoritmo reside en los relojes lógicos** que se van incrementando y que permiten ordenar las peticiones de acceso a la sección crítica. No obstante, **cabe notar que el algoritmo solo considera dos tipos de eventos, a efectos de relojes lógicos, los eventos de envío y recepción de acceso a la sección crítica**. Además, este

**operation** acquire_mutex() **is**
(1) $cs\_state_i \leftarrow trying$;
(2) $\ell rd_i \leftarrow clock_i + 1$;
(3) $waiting\_from_i \leftarrow R_i$; % $R_i = \{1, \ldots, n\} \setminus \{i\}$
(4) **for each** $j \in R_i$ **do** send REQUEST($\ell rd_i, i$) to $p_j$ **end for**;
(5) **wait** ($waiting\_from_i = \emptyset$);
(6) $cs\_state_i \leftarrow in$.

**operation** release_mutex() **is**
(7) $cs\_state_i \leftarrow out$;
(8) **for each** $j \in perm\_delayed_i$ **do** send PERMISSION($i$) to $p_j$ **end for**;
(9) $perm\_delayed_i \leftarrow \emptyset$.

**when** REQUEST($k$, $j$) **is received do**
(10) $clock_i \leftarrow \max(clock_i, k)$;
(11) $prio_i \leftarrow (cs\_state_i \neq out) \wedge (\langle \ell rd_i, i \rangle < \langle k, j \rangle)$;
(12) **if** ($prio_i$) **then** $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$
(13) **else** send PERMISSION($i$) to $p_j$
(14) **end if**.

**when** PERMISSION($j$) **is received do**
(15) $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$.

Figura 1: Esquema del Algoritmo de Ricart-Agrawala, para su implementación es recomendable acceder al código en Algol

algoritmo se puede generalizar fácilmente para resolver problemas de sincronización más complejos.

## 2. Algoritmo de Ricart-Agrawala Generalizado: de un mutex simple a un mutex de tipos de operación

De manera más general, existen problemas de sincronización que, en lugar de tener una única operación de acceso, consideran varios tipos de operaciones y reglas de exclusión entre ellas, por ejemplo, el problema de Lectores / Escritores. El algoritmo de Ricart-Agrawala puede extenderse fácilmente para implementar otros problemas más complejos, mediante dos cambios:

- construyendo una matriz que defina las reglas de exclusión de las operaciones del problema de sincronización
- modificando las instrucciones (4) y (11) del Algoritmo de Ricart-Agrawala, de manera que incluya dicha matriz.

En general, en caso de que el problema conste de dos operaciones, op1() y op2(). Se puede definir una matriz simétrica booleana de exclusión de operaciones exclude (simétrica quiere decir que exclude[op1, op2] = exclude [op2, op1]). De manera que:

- si dos operaciones op1 y op2 no pueden ejecutarse simultáneamente, la matriz exclude[op1, op2] = true.
- Si dos operaciones pueden ejecutarse simultáneamente entonces exclude[op1, op2] = false.

Por ejemplo, consideremos el problema de los lectores / escritores. Existen dos operaciones read() y write(). La matriz de exclusión es tal que exclude[read, read] = false, exclude[write, read] = exclude [write, write] = true.

En esta práctica, se proporciona un esquema en Go para la implementación de este algoritmo y que hay que completar. El esquema describe un módulo en Go con dos operaciones que se exportan:

```
1 // Pre: Verdad
2 // Post: Realiza el PreProtocol para el algoritmo de
3 //          Ricart-Agrawala Generalizado
4 func (ra *RASharedDB) PreProtocol()
5
6 // Pre: Verdad
7 // Post: Realiza el PostProtocol para el algoritmo de
8 //          Ricart-Agrawala Generalizado
9 func (ra *RASharedDB) PostProtocol()
```

# 3. Implementación: Modelo Actor

Si bien en la práctica anterior, para la comunicación entre procesos, utilizamos directamente las operaciones de TCP read y write y el paquete gob para serializar las estructuras de datos; en esta práctica vamos a incorporar algunas características del modelo actor. Por otra parte, en asignaturas anteriores, habéis podido comprobar la dificultad que conlleva verificar la corrección de un algoritmo concurrente o distribuido. En esta práctica, vamos a introducir mecanismos para realizar pruebas unitarias de los componentes de un sistema distribuido, así como de sus sincronizaciones.

El modelo de actor [5] [6] es un modelo de computación concurrente que considera al *actor* como el elemento fundamental de computación. Un actor es un proceso que puede enviar y recibir mensajes. En respuesta a un mensaje recibido, un actor puede: tomar decisiones locales, crear más actores, enviar más mensajes y determinar cómo responderá a los siguientes mensajes recibidos. Los actores pueden modificar su propio estado local, pero solo pueden afectarse unos a otros a través del intercambio de mensajes. La comunicación entre los actores es *directa y asíncrona*, esto es, los actores se conocen entre sí y se envían mensajes directamente y, además, para que se produzca la comunicación entre dos o varios actores, no tienen que coincidir simultáneamente en el acto de comunicación. Esto se traduce en que los actores cuentan con dos primitivas de comunicación: *send* y *receive* para el envío y recepción de mensajes, respectivamente. La operación *send* no bloquea al emisor y una vez que se envía un mensaje este llega al buzón del destinatario. El proceso actor destinatario puede consultar los mensajes del buzón en cualquier momento, si no hay mensajes, entonces la invocación a *receive* bloquea hasta que llegue un mensaje (*bajo grado de acoplamiento temporal*).

*En esta práctica*, no vamos a utilizar el modelo actor en su totalidad, pero *sí vamos a utilizar las dos operaciones de comunicación send y receive*, con su semántica habitual (asíncrona y directa) y con el buzón. Vamos a asumir que todos los procesos del sistema distribuido pueden enviar un mensaje a cualquier otro. Para ello, vamos a asignar un identificador único a cada proceso (un número natural comenzando en 1) y de esta forma enmascararemos la ubicación en la red (IP y puerto). Así ocultaremos los detalles de comunicación, para centrarnos en el desarrollo de nuestro objetivo: la resolución del problema de lectores y escritores completamente distribuido.

Para esta práctica se adjunta el paquete *ms*, que debéis utilizar, y que nos proporciona las siguientes operaciones, con la semántica descrita en esta sección:

```
1 // Pre: pid en {1..n}, el conjunto de procesos del SD
2 // Post: envía el mensaje msg a pid
3 func (ms *MessageSystem) Send(pid int, msg Message) {
4     conn, err := net.Dial("tcp", ms.peers[pid - 1])
5     checkError(err)
6     encoder := gob.NewEncoder(conn)
7     err = encoder.Encode(&msg)
8     conn.Close()
```

[5] Artículo sobre el Modelo Actor `https://tinyurl.com/zckvnnds`
[6] Vídeo sobre el modelo actor por Hewitt, Meijer y Szyperski: `https://youtu.be/7erJ1DV_Tlo`

```
9  }
10
11 // Pre: True
12 // Post: el mensaje msg de algún Proceso P_j se retira
13 //           del mailbox y se devuelve
14 //       Si mailbox vacío, Receive bloquea hasta que
15 //           llegue algún mensaje
16 func (ms *MessageSystem) Receive() (msg Message) {
17     msg = <-ms.mbox
18     return msg
19 }
```

# 4. Servicio de Logging Distribuido

## 4.1. Verificación y Pruebas en Go

La verificación del software es uno de los aspectos más importantes de la construcción de sistemas computacionales y trata de comprobar que el software realiza exactamente la funcionalidad para la que fue diseñado y que, además, la realiza cumpliendo los requisitos no funcionales. Para la verificación, existen fundamentalmente dos aproximaciones: la verificación formal, utilizando métodos formales, y las pruebas (testing), que consiste en realizar ejecuciones del software sobre un subconjunto de datos de entrada que se considere relevante. Dentro de las pruebas de software, las pruebas unitarias representan un método mediante el cual el objeto de las pruebas es un conjunto de componentes software.

El lenguaje Go tiene integrados un conjunto de mecanismos que permiten diseñar y ejecutar un conjunto de pruebas unitarias. El paquete *testing* de Go proporciona la funcionalidad y las herramientas para construir los test unitarios y el comando *go test* ejecuta las pruebas. Para más información, ejemplos y documentación sobre las pruebas unitarias en Go, podéis acceder a estas páginas [7] [8]

## 4.2. Logging o registro de eventos en el código

En Golang, hay un módulo de logging (paquete log `https://pkg.go.dev/log`) que puede ser útil para depurar programas. Un ejemplo de su uso podéis encontrarlo aquí:

`https://medium.com/@pradityadhitama/simple-logger-in-golang-f72dadf2c8c6`

El log puede utilizarse para estampillar eventos con tiempos físicos. Un ejemplo de utilización del log podría ser mediante su configuración para que los mensajes salgan con estampillas temporales en microsegundos, nombre de fichero y línea. Por ejemplo, para ello, la configuración del paquete log sería la siguiente:

[7] `https://gobyexample.com/testing`
[8] `https://golang.org/doc/tutorial/add-a-test`

```
// configurar para que logs salgan con estampillas en
// microsegundos, nombre de fichero y línea de los prints
log.SetFlags(log.Lshortfile | log.Lmicroseconds)
```

Luego en el código:

```
log.Println("Client send Request with Id and Interval : ", request)
```

Para evitar problemas de sincronización entre los relojes físicos, se pueden introducir pequeños retardos en la función "send" ( Time.sleep(1 * Time.Millisecond) ) para asegurar el úmbral de relevancia de relojes físicos.

## 4.3. Verificación en Sistemas Distribuidos: GoVector y ShiViz

Verificar la corrección un sistema concurrente o distribuido es mucho más complejo que un sistema formado por un único proceso secuencial: incluso habiendo superando las pruebas unitarias, puede haber entrelazados y ejecuciones concurrentes que den lugar a condiciones de carrera que solo se manifiestan en determinadas circunstancias.

Además, los procesos de un sistema distribuido no comparten un reloj global y se sincronizan a través del intercambio de mensajes. Por ese motivo, en general, se considera que los eventos de envío y recepción de mensajes pueden modificar el estado de un sistema distribuido, además de los eventos (instrucciones) internas a cada proceso. Existen herramientas en Go, GoVector y ShiViz, para visualizar gráficamente el intercambio de mensajes, de manera que se pueda comprobar visualmente si la sincronización es correcta. Por un lado, GoVector permite introducir relojes vectoriales y asignarlos a eventos en el código. A partir de ahí, una vez terminada la ejecución del sistema distribuido, se puede establecer una relación de causalidad y esta puede visualizarse con la herramienta ShiViz.

GoVector es un módulo para Golang que permite generar un fichero de log de eventos (propios, envío y recepción, aunque es el programador el que tiene que decidir qué eventos registra). GoVector permite asociar a cada evento una estampilla temporal vectorial, de manera que al final de la ejecución, se pueden ordenar *parcialmente* todos los eventos registrados. Para registrar un evento, se utilizan fundamentalmente 3 funciones:

- *PrepareSend* (antes del envío) codifica los mensajes para el transporte de red, actualiza el reloj vectorial del proceso y registra un evento de envío.
- UnpackReceive (en la recepción) y
- LogLocalEvent (evento propio). UnpackReceive decodifica los mensajes de la red, fusiona el reloj local de GoVectors con el reloj recibido y registra un evento de recepción. El evento LogLocalEvent marca el tiempo y registra un mensaje.

La idea intuitiva tras GoVector es que cada proceso en un sistema distribuido genere su propio fichero de log de eventos (estampillados con un reloj vectorial). Posteriormente, una vez terminada la ejecución del sistema, GoVector permite generar un fichero unificado, de sistema, a partir de todos los logs. Ese fichero será la entrada para ShiViz. Ambas herramientas cuentan con ejemplos para comprender fácilmente su uso.

# 5. Algoritmo de Ricart-Agrawala con relojes vectoriales

Los relojes del logging distribuido son vectoriales y para que solo haya un tipo de reloj en el código, vamos a modificar el Algoritmo de Ricart-Agrawala para que funcione con relojes vectoriales.

Una posible relación de orden total para el Algoritmo de Ricart-Agrawala con relojes vectoriales puede definirse de esta manera. Sea un conjunto de procesos $P_1 \dots P_n$ de un sistema distribuido, de manera que el pid $i$ de cada proceso coincide con su subíndice $P_i$. Sean dos eventos $e_i$ y $e_j$ en el sistema distribuido y $date(e_i)$, $date(e_i)$ sus respectivos relojes vectoriales. Una relación total happens-before sobre la relación parcial definida, puede obtenerse de la siguiente forma:

$$e_i \xrightarrow{to\_ev} e_j \stackrel{def}{=} \begin{cases} (date(e_i) < date(e_j)) & \text{si las fechas son comparables} \\ (i < j) & \text{si las fechas no son comparables} \end{cases}$$

Por tanto, se puede modificar el Algoritmo de Ricart-Agrawala fácilmente para reemplazar el Algoritmo de Ricart-Agrawala mediante el uso de relojes vectoriales.

# 6. Instrucciones de Entrega

Deberéis entregar un fichero zip que contenga: (i) los fuentes para la resolución de la práctica y (ii) la memoria en pdf. La entrega se realizará a través de moodle en la actividad habilitada a tal efecto. **La fecha de entrega será no más tarde del día anterior anterior al comienzo de la siguiente práctica**.

## 6.1. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rúbrica en la Tabla 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

- A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, sin errores. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje, así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.

Tabla 1: Rúbrica para la Práctica

| Calificación | Código | Memoria |
|---|---|---|
| 10 | A+ | A+ |
| 9 | A | A |
| 8 | A | A |
| 7 | A | A |
| 6 | B | B |
| 5 | B- | B- |
| suspenso | | |

- A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, con ciertos errores no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funcionan correctamente. En el caso del código, este se ajusta casi exactamente a las guías de estilo propuestas.

- B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero con errores. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.

- B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero con errores de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son manifiestamente mejorables, el lenguaje presenta serias deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

## Referencias

[1] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.