Práctica 2. Lectores y Escritores Distribuidos

1. Objetivos y Requisitos

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.

1.1 Objetivo

- Familiarización de los relojes lógicos de Lamport y el algoritmo de Ricart-Agrawala
- Implementación de los relojes lógicos de Lamport en un SSDD programado en Go.
- Uso de ordenación de eventos y de instrumentación de código para depuarar sistemas distribuidos.

1.2 Requisitos

- Go y su entorno de desarrollo
- Utilización de la herramienta GoVector para logging de sistemas distribuidos

  https://github.com/DistributedClocks/GoVector
- Utilización de la herramienta de visualización de logs ShiViz.

  https://bestchai.bitbucket.io/shiviz/
- El algoritmo original de Ricart-Agrawala implementado en Algol

  https://dl.acm.org/citation.cfm?id=358537

2. Conceptos Subyacentes

2.1 El Modelo Actor[1,2]

El modelo de actor es un modelo de computación concurrente que considera al actor como el elemento fundamental de computación concurrente. 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 privado, pero solo pueden afectarse unos a otros indirectamente 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 lo bloquean hasta que llegue un mensaje.

---

[1] https://heather.miller.am/teaching/cs7680/pdfs/Concurrent-Object-Oriented-Programming.pdf

[2] https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

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 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). De esta manera ocultaremos los detalles de comunicación, para centrarnos en el desarrollo de nuestro objetivo: la resolución del problema de lectores y escritores completamente distribuido. Estas son las operaciones que nos va a proporcionar el módulo messagesystem:

```
func New(whoIam int, usersFile string) (ms MessageSystem)
// Pre: whoIam es un número natural que identifica a un proceso de forma unívoca en el sistema
// usersFile es un fichero de texto que tiene en cada línea IP:puerto, de manera que la línea
iésima especifica la dirección IP y el puerto donde del proceso iésimo del sistema recibe
mensajes.
// Post: inicializa el proceso whoIam del Sistema MessageSystem
//NOTA: por cada proceso del Sistema Distribuido, hay que invocar New(proc, usersfile)

func (ms *MessageSystem) Receive() (msg Message)
// Pre: true
// Post: devuelve el primer mensaje del buzón, bloquea si no había mensajes

func (ms MessageSystem) Send(pid int, msg Message)
// Pre: pid es un número natural válido en el sistema distribuido
// Post: envía el mensaje msg al proceso pid

func (ms *MessageSystem) Stop()
// Pre: true
// Post: cierra el buzón
```

2.2 Depuración en un Sistema Distribuido: Las herramientas GoVector[3] y ShiViz[4]

Depurar un sistema distribuido es mucho más complejo que un sistema formado por un único proceso secuencial. Las razones ya las hemos estudiado, los procesos de un sistema distribuido no comparten un reloj global. Una forma de abordar el reto de la depuración de un sistema distribuido consiste en considerar qué tipos de eventos pueden modificar el estado del sistema distribuido, en general, vamos a considerar tres: eventos propios de un proceso, evento de envío y evento de recepción. Después, se trata de establecer una relación de causalidad entre los eventos, para ello, podemos utilizar relojes lógicos. Finalmente, podríamos realizar distintos experimentos y comprobar que las relaciones de causalidad entre los eventos son las esperables.

En esta práctica, vamos a utilizar dos herramientas: GoVector y ShiViz. GoVector nos va a permitir generar un fichero de log para cada proceso de nuestro sistema distribuido. En su fichero de log, cada proceso va a registrar los eventos que sucedan durante su ejecución (propios, recepción y envío). ShiViz nos va a permitir visualizar gráficamente la ejecución y nos permitirá analizar si la ejecución fue correcta o no.

GoVector es un módulo para Golang que permite generar un fichero de log de eventos (propios, envío y recepción) y asociar a cada uno de ellos una estampilla temporal vectorial, de manera que al final de la ejecución, se pueden ordenar parcialmente todos los eventos registrados. Se utilizan fundamentalmente

[3] https://github.com/DistributedClocks/GoVector

[4] https://bestchai.bitbucket.io/shiviz/

3 funciones: PrepareSend (antes del envío), UnpackReceive (en la recepción) y LogLocalEvent (evento propio). PrepareSend codifica los mensajes para el transporte de red, actualiza el reloj vectorial del proceso y registra un evento de envío. 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.

3. Ejercicio

En esta práctica vamos a implementar una infraestructura para alojar un repositorio para el enunciado de un trabajo de asignatura, siguiendo el esquema clásico de sincronización de lectores y escritores. La infraestructura permitirá escribir en el repositorio a un profesor y leer del repositorio a un alumno. En todo momento, solo un usuario profesor (rol de escritor) puede acceder al repositorio, mientras que puede haber múltiples alumnos (rol de lector) leyéndola simultáneamente.

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 pre-protocol, realizado por las instrucciones (1) a (6). El pre-protocol para un Proceso Pi 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 Pi accede a la sección crítica. Una vez terminado el acceso a la sección crítica, el Proceso Pi realiza el post-protocol, instrucciones (7)-(9) donde se envía el ACK a los procesos cuyas peticiones habían sido postergadas. El Proceso Pi, mientras está ejecutando el pre-protocol, acceso a la sección crítica, post-protocol y sección no crítica (si la hubiera), tiene que atender simultáneamente las peticiones de otros procesos Pj. Por ello cada proceso Pi 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 un tipo de evento, el evento de acceso a la sección crítica.** Además, este algoritmo se puede generalizar fácilmente para resolver problemas de sincronización más complejos.

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: Algoritmo de Ricart Agrawala.

3.1 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:

1. i) construyendo una matriz que defina las reglas de exclusión de las operaciones del problema de sincronización
2. ii) 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:

1. (i) si dos operaciones op1 y op2 no pueden ejecutarse simultáneamente, la matriz *exclude*[op1, op2] = true.
2. (ii) 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.

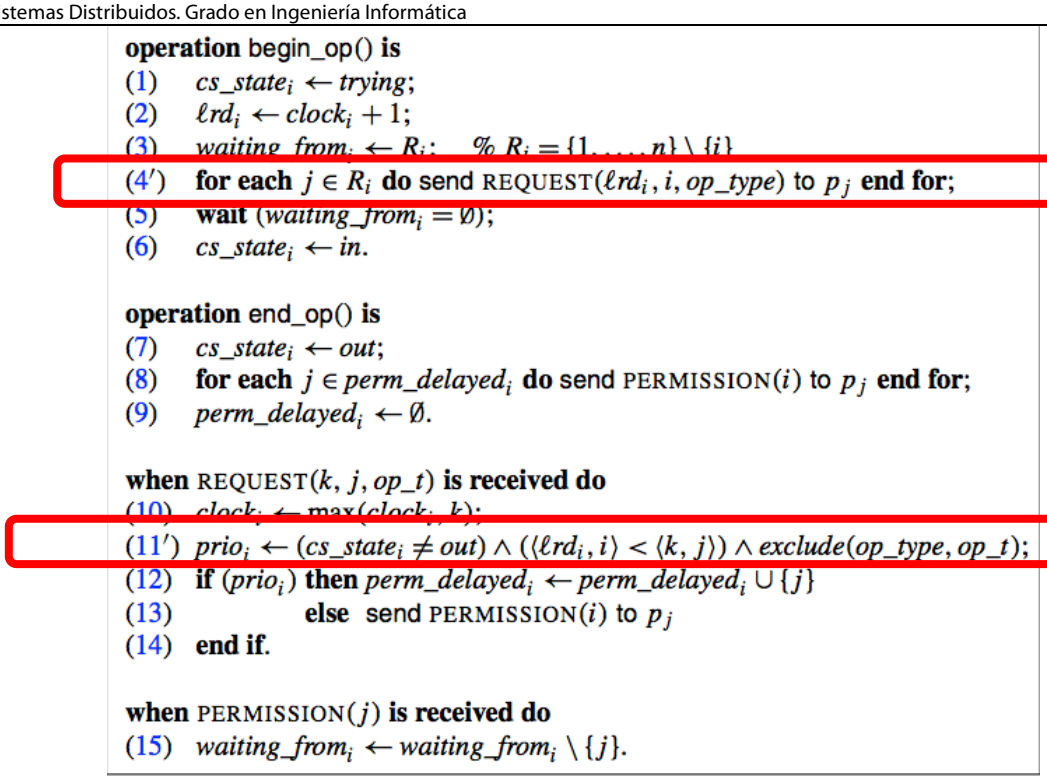

Figura 2: Algoritmo de Ricart Agrawala Generelizado.

4. Ejercicio

Se pide:

1) implementar el Algoritmo de *Ricart-Agrawala Generalizado*, para permitir el acceso al repositorio del enunciado del trabajo de asignatura siguiendo el esquema de sincronización de lectores y escritores. Se recomienda realizar una traducción del código en Algol y no fundamentar exclusivamente la implementación en las Figuras 1 y 2. El código de las Figuras 1 y 2 podría tener condiciones de carrera. Además, la implementación tiene que utilizar el módulo messagesystem proporcionado para implementar la comunicación entre procesos.
2) Además, deberéis diseñar programas de prueba con varios lectores / escritores para probarlo. Para ello, tendréis que utilizar las herramientas GoVector y ShiViz.
3) Adicionalmente, deberéis redactar una 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.).

5. Evaluación

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general estos son los criterios de evaluación:

- Deben entregarse *todos* los programas, se valorará de forma negativa que falte algún programa.
- Todos los programas deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas deben funcionar correctamente como se especifica en el problema.

- Todos los programas tienen que seguir el manual de estilo de Elixir, disponible en[5] (un 20% de la nota estará en función de este requisito). Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
# AUTORES: nombres y apellidos
# NIAs: números de identificaci'on de los alumnos
# FICHERO: nombre del fichero
# FECHA: fecha de realizaci’on
# TIEMPO: tiempo en horas de codificación
# DESCRIPCI'ON: breve descripci'on del contenido del fichero
```

5.1. 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 rubrica 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.

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

[5] https://github.com/christopheradams/elixir_style_guide/blob/master/README.md

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.

C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

Calificación	Código	Memoria
10	A+	A+
9	A	A
8	A	A
7	A	A
6	B	B
5	B-	B-
suspenso	1C	

Tabla 1. Rúbrica

6. Entrega

Deberéis entregar un fichero zip que contenga: (i) los fuentes: lector.go, escritor.go, agrawala.go, y cualquier otro fuente que desarrolléis (ii) la memoria en pdf (4 páginas). La entrega se realizará a través de moodle2 en la actividad habilitada a tal efecto. La fecha de entrega será no más tarde del día anterior al comienzo de la siguiente práctica. Durante la siguiente sesión de prácticas se realizará una defensa "in situ" de la práctica.

Bibliografía

[1] Ricart, G., & Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM, 24(1), 9-17. https://dl.acm.org/citation.cfm?id=358537