SISTEMAS DISTRIBUIDOS

PRÁCTICA 2

Pablo Moreno Muñoz 841972
Andrés Yubero Segura 842236

ÍNDICE

Introducción

Esta práctica consiste en diseñar una arquitectura distribuida que gestiona N procesos lectores y M procesos escritores, donde cada uno tiene un fichero de texto, del cual cada proceso tiene una copia. Varios procesos pueden leer simultáneamente, sin embargo, a la hora de escribir, solo uno puede hacerlo a la vez, y mientras se escribe tampoco se puede leer; por lo que habrá que garantizar la exclusión mutua al escribir en el fichero.

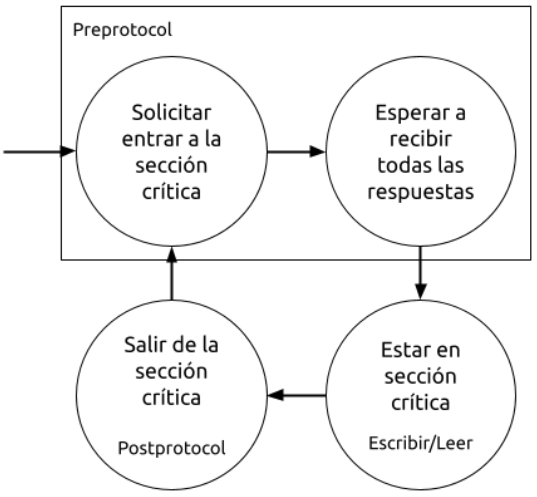
Una vez escrito el proceso escritor, éste comunicará al resto de procesos lo que ha escrito para que actualicen sus copias del fichero locales, después esperaremos antes de salir de la sección crítica a que envíen un mensaje de confirmación de que han actualizado el fichero correctamente.

Implementación del Algoritmo de Ricart-Agrawala

Para lograr la sincronización y coordinación entre los lectores y escritores, se ha empleado el algoritmo de Ricart-Agrawala generalizado. De esta manera, cualquier proceso que desee acceder a la sección crítica (ya sea para leer o escribir) debe realizar un preprotocolo (equivalente a los dos primeros estados en la ilustración de abajo), que implica enviar una solicitud de acceso a los otros N-1 procesos distribuidos. El que consiga todas las confirmaciones entrará en la sección crítica.

Una vez finalizada la sección crítica, se lleva a cabo un postprotocolo, en el cual se envía una confirmación (ACK) a los procesos cuyas solicitudes habían sido pospuestas. Durante este proceso, los procesos deben estar atentos a las solicitudes de los demás procesos.

La máquina de estados del algoritmo sería la siguiente:



En este caso el algoritmo original para la exclusión mutua se basa en relojes lógicos, los cuales se van incrementando de manera ordenada para gestionar la entrada a la sección crítica, este algoritmo se ha traducido el algoritmo original escrito en Algol. No obstante, como comentaremos en el apartado de “Relojes vectoriales”, el algoritmo se ha generalizado para permitir trabajar con relojes vectoriales.

El módulo "ra.go" incluye operaciones PreProtocol y PostProtocol que equivalen al código original. Además, se han creado tres procesos concurrentes, "handleRequests" y "handleReplies", "receiveMessages" para recibir los mensajes y poder manejar las solicitudes y respuestas.

Matriz de exclusión

Este algoritmo original no permitía múltiples tipos de operaciones, además había ciertas restricciones de exclusión específicas que hay que tener en cuenta, como que dos o más lectores pueden leer a la vez, pero un lector no puede leer si un escritor está escribiendo o dos escritores no pueden escribir a la vez.

Para ello se creó un mapa que define las reglas de exclusión para las operaciones, implementadas que devuelve un valor booleano (verdadero si las operaciones son excluyentes, falso en caso contrario). Este mapa que devuelve un booleano es de un tipo de dato *Exclusion* que está conformado por dos strings que codifican las operaciones “read” y “write”.

El mapa queda como se muestra en la siguiente tabla:

String 1	**String 2**	**Booleano**
“read”	“read”	false
“read”	“write”	true
“write”	“read”	true
“write”	“write”	true

Relojes vectoriales

Para esta implementación, se reemplazaron los relojes lógicos del algoritmo original de Ricart-Agrawala por dos relojes vectoriales, guardando “OurSeqNum” y “HighSeqNum” en las posiciones correspondientes al id del proceso en el mapa que codifica cada reloj vectorial.

Inicialmente se utilizó el paquete “GoVector” para introducir estos relojes vectoriales y asignarlos a eventos en el código. El paquete “GoVector” asocia una estampilla temporal vectorial a cada evento y registra eventos internos mediante funciones como "PrepareSend" (utilizada antes de enviar un mensaje) y “UnpackReceive” (utilizada después de recibir un mensaje). “GoVector” permite guardar en logs las relaciones de causalidad mediante el registro de eventos que se ha comentado, y visualizarlos posteriormente con la herramienta ShiViz.

Los gráficos de comunicación ShiViz que se muestran en su correspondiente apartado han sido generados con “GoVector”. No obstante, para obtener un código más limpio y fiel al algoritmo de Ricart-Agrawala, se ha optado por usar exclusivamente el paquete “VClock” que viene dentro de “GoVector”.

“VClock” incluye una función *Compare* que devuelve un booleano si dos relojes vectoriales cumplen una determinada condición.

Las condiciones que se han empleado son “Descendant” para comprobar si date(ei) < date(ej), y “Concurrent” para comprobar si no son comparables, y por tanto proceder a comprobar si (i<j).

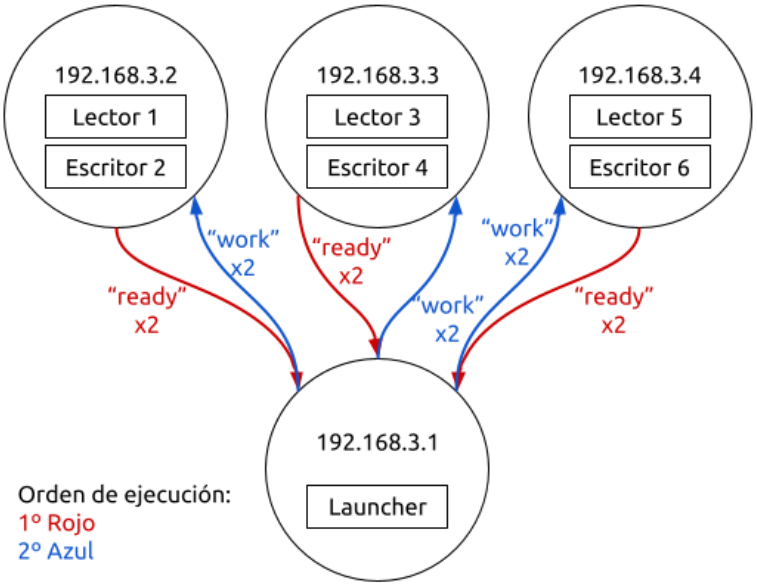
$$e_i \xrightarrow{\text{to}_e\text{V}} e_j \stackrel{\text{def}}{=} \begin{cases} (\text{date}(e_i) < \text{date}(e_j)) & \text{si las fechas son comparables} \\ (i<j) & \text{si las fechas no son comparables} \end{cases}$$

De esta manera, si el reloj vectorial del proceso actual es inferior al de la petición, o sino son comparables, el id del proceso actual es inferior, devolveremos true para indicar que la respuesta a la petición debería ser postergada. Aunque para determinar si se aplaza la respuesta intervienen también dos factores: el hecho de haber solicitado entrar a sección crítica y si hay que cumplir algún tipo de exclusión.

Descripción de la aplicación diseñada

Distribución de los procesos

Los distintos procesos que intervienen en nuestras pruebas del algoritmo son los procesos lectores y escritores (en total hay 3 de cada uno, por lo tanto 6 procesos lectores/escritores) y un proceso al que podemos llamar lanzador o *Launcher* que equivale al start.go. Están repartidos por las máquinas que se nos han asignado de la siguiente manera:



Los mensajes que aparecen en rojo y azul, son los encargados de sincronizar los procesos esperando a que todos lleguen a la barrera. En esencia, los lectores y escritores envían un mensaje como que están preparados “Ready” al *Launcher*, éste espera a recibir los 6 mensajes “Ready”, cuando ha recibido todos, envía de vuelta un mensaje “Work” para que los escritores y lectores se pongan a trabajar.

Nota: Se mencionan 6 procesos escritor/lector, pero podríamos generalizar a N procesos.

Barrera

Es importante tener en cuenta que, una vez que la aplicación se inicia, existe la posibilidad de que el primer proceso inicie su fase de preparación para el protocolo antes de que todos los demás procesos estén listos para recibir sus mensajes. Esto podría dar lugar a la pérdida de mensajes, ya que algunos procesos podrían no estar disponibles para recibirlos.

Para abordar este problema, se ha implementado una barrera centralizada en el fichero start.go que garantiza que ningún proceso comience hasta que todos los procesos estén listos para enviar y recibir mensajes.

A la hora de implementar la barrera hemos creado los siguientes métodos dentro del paquete *com* para los procesos lectores y escritores: “WaitWorkOrder” y “SendReady”. Para el proceso coordinador *Launcher* definido en start.go, hemos empleado otras dos funciones llamadas “waitProcesses” y “putProcessesToWork”:

- **Función *SendReady***:
  Esta función se encarga de enviar un mensaje que indica que el proceso está listo. Para ello, intenta establecer una conexión TCP con la dirección especificada en barrierEndpoint. Una vez establecida la conexión, se envía un mensaje que contiene el ID del proceso y su estado "listo". Esta función se utiliza para notificar a otros procesos que este proceso ha completado su preparación y está listo para continuar.

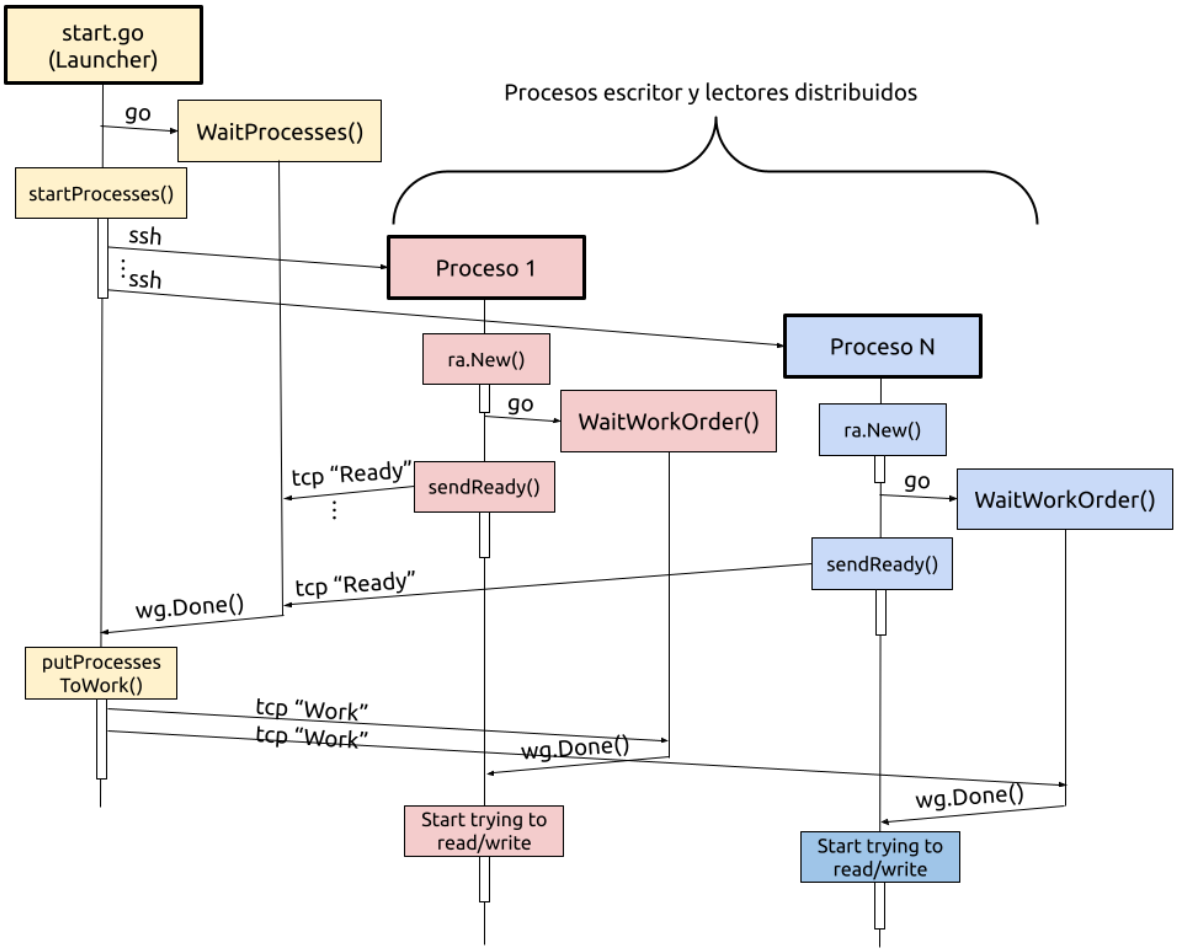
- **Función *WaitWorkOrder:***
  La función "WaitWorkOrder" está diseñada para esperar una orden de trabajo a través de una conexión TCP. Primero, crea un listener en la dirección especificada en endpoint. Luego, entra en un bucle de escucha donde acepta conexiones entrantes. Cuando se recibe una conexión, se verifica el contenido del mensaje. Si el mensaje indica una "orden de trabajo" (status = "work"), la función sale del bucle y permite que el proceso continúe con su tarea. Esta función es esencial para coordinar el inicio de tareas entre procesos en un sistema distribuido.

- **Función *waitProcesses***:
  La función "waitProcesses" se ejecuta desde start.go y se encarga de esperar a que otros procesos notifiquen que están listos para continuar. Esto se hace mediante un bucle que espera a recibir 6 mensajes de tipo *BarrierMessage* con el Status de “Ready”. Una vez se haya salido del bucle, se pone como *Done()* al *wait group* que se pasa como puntero, para que *putProcessesToWork()* se ejecute.

- **Función *putProcessesToWork***:
  La función “putProcessesToWork” se encarga de enviar un *BarrierMessage* a cada proceso escritor/lector con el Status “Work” para indicarles que ya se pueden poner a trabajar.

Abajo se muestra un diagrama de secuencia que describe cómo funciona el algoritmo de barrera. En primer lugar, al arrancar el proceso *Launcher*, se crea una gorutina de *WaitProcesses* que se pone a la escucha de los mensajes de lectores y escritores. Después arrancamos los procesos lectores y escritores, cuando estos hayan creado una gorutina para ponerse a la escucha de la orden de trabajar “Work” del start.go e inicializado el

algoritmo de Ricart-Agrawala con ra.New(), entonces estarán listos y enviarán un “Ready” al Launcher. Cuando el Launcher haya recibido todos los “Readies”, uno por cada proceso escritor/lector, procederá a darles la orden de trabajar “Work”. Finalmente cuando cada escritor o lector reciba la orden de trabajar, intentará entrar a su sección crítica para leer o escribir.



Gestión del fichero

Los procesos escritores y lectores acceden a un paquete que se ha implementado que se llama *fileManager.go* donde se encuentran las distintas operaciones que se pueden realizar sobre el fichero:

- func Create(filename string) File {}
- func (f *File) Write(data string) {}
- func (f *File) Read() string {}

En este paquete se ha definido un tipo de dato **File** sobre el cuál operan las funciones Write() y Read(). Es por ello que, a la hora de crear un fichero con Create(), esta función devuelve un tipo de dato File.

Estructura del algoritmo

Estructuras de Datos

- **Request, Reply, Update, UpdateReply:** Estas estructuras definen los mensajes utilizados para la comunicación entre los procesos.
- **RASharedDB:** La estructura RASharedDB representa la base de datos compartida entre los procesos y es esencial para el funcionamiento del algoritmo. Contiene las siguientes variables y componentes significativos:
    - *OurSeqNum y HigSeqNum*: Relojes vectoriales utilizados para llevar un registro de eventos.
    - *OutRepCnt*: Contador de respuestas pendientes.
    - *ReqCS*: Indicador de solicitud de acceso a la sección crítica.
    - *RepDefd*: Registra respuestas diferidas.
    - *ms (MessageSystem)*: Sistema de mensajes utilizado para la comunicación entre procesos.
    - *done y chrep*: Canales para la sincronización de procesos.
    - *chlog*: Canal para pasar los logs a la gorutina que los guardará en el fichero de log.
    - *Mutex*: Mutex para la protección de variables compartidas.
    - *exclude*: Mapa de exclusiones que define las reglas de exclusión entre operaciones.
    - *request, reply y updateReply*: Canales para la gestión de solicitudes, respuestas y actualizaciones.
    - *Function*: Indica si el proceso es un lector o escritor.
    - *File*: Archivo utilizado para almacenar datos compartidos.
    - *meAsString*: Identificación del proceso como cadena.

Inicialización del Sistema

La función New() se encarga de crear una instancia de RASharedDB y devolverla como puntero para ser usada desde otras partes del código. En esta función, se inicializan los tipos de mensajes posibles que serán gestionados más adelante, al igual que los relojes vectoriales.

Además se crea la copia local del fichero distribuido y se establecen los valores iniciales de la matriz de exclusiones. Finalmente, la función crea cuatro gorutinas para guardar los logs, gestionar los mensajes que llegan, gestionar peticiones y gestionar respuestas.

Fases del Algoritmo

- **PreProtocol:** Implementa la fase de preprotocol del algoritmo de Ricart-Agrawala. Realiza las siguientes acciones:
    - Obtiene el mutex para proteger la sección crítica.
    - Establece el indicador de solicitud.
    - Incrementa el número de secuencia propio siguiendo el algoritmo.
      `Our_Sequence_Number := Highest_Sequence_Number + 1`
    - Libera el mutex.
    - Envía solicitudes a otros procesos.

- **PostProtocol:** En este apartado se pone como falso la solicitud de acceso a la sección crítica y se envían mensajes a todos los procesos

Funciones de apoyo

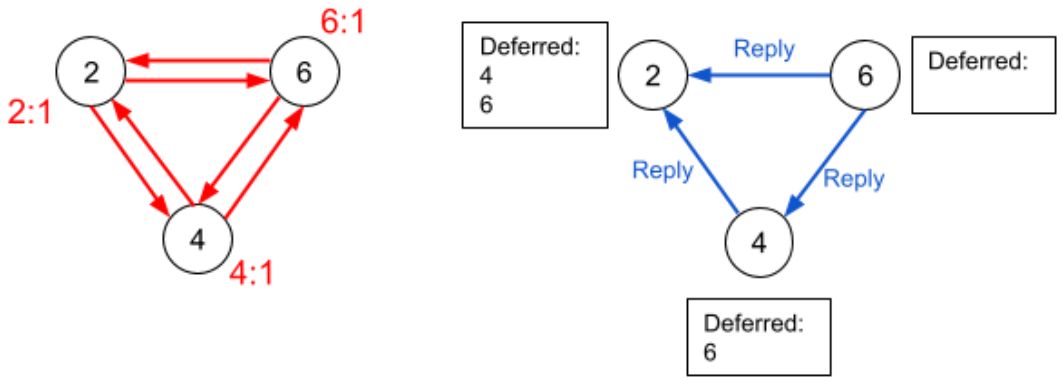
- **isPriorityMine**: Determina la prioridad entre dos relojes vectoriales, es esencial para la decisión de postergar una solicitud, usando funciones de comparación comentadas anteriormente de la librería *Vclock*.
- **receiveMessages**: Gestiona la recepción de mensajes y redirecciona los mensajes a las funciones adecuadas dependiendo del tipo de mensaje determinado en el struct msgs.
- **handleLogs**: Crea un fichero de log, recibe logs a través de un canal y los escribe en el fichero.
- **handleRequests**: Procesa las solicitudes recibidas y toma decisiones basadas en las prioridades de los relojes usando isPriorityMine y la matriz de exclusión para determinar si postergar la petición o no. Posteriormente se actualizan el reloj vectorial *HigSeqNum* con la función merge.
- **handleReplies**: Gestiona las respuestas y permite la entrada a la sección crítica cuando todas las respuestas han sido recibidas.
- **sendUpdate**: Consiste en un bucle del número total de procesos donde se envía un mensaje al resto de procesos que no coinciden con el id del proceso actual, (un mensaje de tipo update). Luego se hace un bucle donde recibimos la respuesta a la recepción del update por parte del resto de procesos para asegurarse de que todos han recibido e implementado la actualización correctamente en su copia del fichero local.

Resultado del fichero de texto

En cada fichero de texto local, el resultado obtenido es el siguiente:

```
I am 2 and this is my contribution number 1
I am 4 and this is my contribution number 1
I am 6 and this is my contribution number 1
I am 2 and this is my contribution number 2
I am 4 and this is my contribution number 2
. . .
I am 4 and this is my contribution number 99
I am 6 and this is my contribution number 99
I am 2 and this is my contribution number 100
I am 4 and this is my contribution number 100
I am 6 and this is my contribution number 100
```

Como se puede ver en las líneas, se sigue un patrón, primero escribe el escritor 2, luego el 4 y finalmente el 6. Esto es debido a que los 3 escritores empiezan a trabajar a la vez, envían sus relojes y, como son iguales, se le da prioridad al proceso de id menor, que sería el 2, quedando 4 y 6 aplazados para entrar a la sección crítica, como se puede ver en la imagen de abajo. Cuando 2 sale de la SC, responderá al 4 y al 6 como que pueden entrar, como el 4 ya tenía la confirmación del 6, entrará a su sección crítica. El 2 volverá a su preprotocol actualizando *OurSeqNum* a un valor más alto que la petición del 6, por lo que el 6 será el siguiente en entrar a la sección crítica. Este proceso se repetirá indefinidamente, dado que no entran nuevos procesos a escribir, y provoca que los escritores siempre entren en el mismo orden, creando el patrón del que se ha hablado al principio de este párrafo.



Logging

El canal "chlog" se crea en la función ra.New y se pasa a la estructura RASharedDB. Este canal se utiliza para comunicar mensajes de registro entre las diferentes partes del programa.

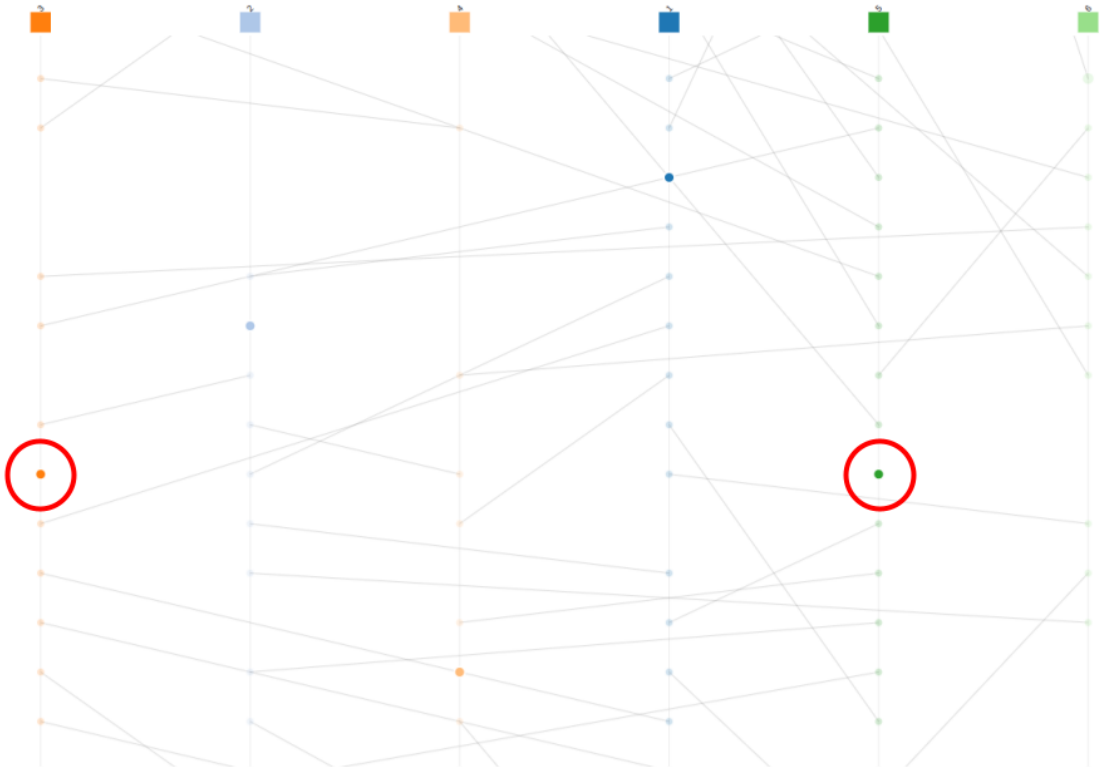
Luego hemos creado la función "handleLogs" que se encarga de manejar la generación y escritura de registros. En primer lugar, la función crea o abre un archivo de registro llamado "id.log" (donde "id" es el identificador del proceso), esto se logra con os.OpenFile.

Además, dentro de “handleLogs”, se configura el formato del registro para incluir el microsegundo y la línea donde ocurre el evento. Esto ayuda a identificar la fuente de los registros y proporciona una marca de tiempo precisa. La función continúa ejecutándose indefinidamente en un bucle. Cuando llega un nuevo mensaje de registro al canal "chlog", se captura y se escribe en el archivo de registro utilizando log.Println().

En cualquier punto del programa donde se desee realizar un registro, se puede escribir un mensaje en el canal "chlog". Este mensaje se procesará por la función "handleLogs" y se almacenará en el archivo de registro correspondiente.

Verificación del sistema con Govector y ShiViz

Como se ha comentado en el apartado sobre relojes vectoriales, nos permite registrar los eventos de envío y recepción de mensajes en un fichero de log. Cada escritor/lector genera un fichero de log propio, que posteriormente tras la ejecución del programa, se han combinado para generar un fichero de log unificado listo para subir a la herramienta ShiViz.



Esta es una captura de una parte del gráfico de comunicación entre procesos, en color oscuro se pueden ver los instantes en los que los procesos entran en la sección crítica. Como se puede observar, existe un momento en el que dos procesos (el 3 y el 5) están a la vez en la sección crítica, pero no existe ningún problema ya que son lectores y no hace falta que haya exclusión entre ambos.