Universidad Rey Juan Carlos

Memoria P2:
El Problema de Santa Claus

Presentado por: *Fernando González Ramos*
Titulación: *Ingeniería Telemática*
Fecha: *11/01/2021*

- Índice

1.- Introducción

El programa planteado entraña un problema de concurrencia debido a que encontramos tres “entes”, los cuales son los renos, Santa Claus y los elfos, realizando tareas de manera concurrente. Pero… ¿por qué motivo decimos que son concurrentes y no paralelas?

Cierto es que dichas tareas se ejecutan a la vez, como posteriormente se detalla; sin embargo, en este caso, necesitamos de mecanismos que permitan la comunicación entre estas diferentes tareas. En este caso, este mecanismo se basa en la compartición de memoria, bien sea por medio de “canales” de Golang o de otras variables compartidas.

Con respecto a los canales que el lenguaje Golang proporciona, algunos de los mismos (concretamente los *Buffered Channels*) garantiza coherencia en los datos para cada acceso, bien sean de lectura o de escritura.

Sin embargo, para garantizar que no se produzcan condiciones de carrera en aquellas variables compartidas, es decir, que son comunes a diferentes hilos de ejecución (“Go routines”, en este caso), es necesario el uso de *Locks*.

Pero... ¿Qué es un *Lock*?

Llamamos, comunmente, *Lock*, a lo que formalmente se conoce como *cierre de exclusión mutua.* Esto es, un mecanismo que permite a los programadores sincronizar los accesos de lectura y escritura a variables compartidas, de manera que no se produzca una lectura inmediatamente posterior a una escritura cuando ésta aún no ha finalizado, etc.

Acciones de este tipo, pueden dar lugar a inconsistencia en los datos y, a menudo, los resultados en un programa son catastróficos, en el caso mejor; o impredecibles, en el caso peor, ya que la aparente aleatoriedad de los errores que esto ocasiona, dificulta notablemente la labor de depuración.

A continuación, paso a detallar el planteamiento general del programa para abordar la solución, mediante el lenguaje de programación Golang, al problema planteado.

2.- Estructura General del Programa

El programa se plantea de la siguiente forma: En primer lugar, hay un programa principal, al que he llamado *main.go.* Este fichero únicamente contiene el punto de entrada (la función *main*) y una función adicional que se encarga de esperar a que todas las *Go routines* terminen de hacer su trabajo. Es necesario hacerlo así debido a que las rutinas que ofrece Go no están implementadas como demonios y, por tanto, si el hilo de ejecución principal termina, el resto de rutinas mueren al instante y finaliza el programa.

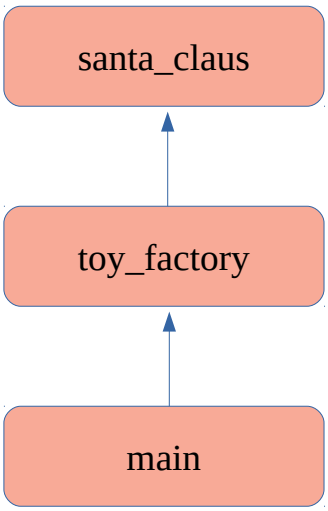
Por otra parte, he estructurado el código en dos paquetes que paso a mencionar brevemente a continuación:

En primera instancia, tenemos el paquete *toy_factory,* implementado mediante el fichero *toy_factory.go*, que implementa los tipos de datos necesarios para gestionar la

fábrica de juguetes de Santa Claus, así como todos sus métodos. Este paquete entraña la mayor parte de la complejidad del programa.

Por último, se encuentra el paquete *santa_claus*, implementado mediante el fichero *santa_claus.go*, que contiene el tipo de datos y los métodos necesarios para modelar el comportamiento de Santa Claus.

De este modo, podemos establecer un “grafo de dependencias” de la siguiente manera:



Creo necesario destacar que en todo momento se ha tratado de llevar a cabo el programa de la forma más escalable y eficiente que me ha sido posible. De manera que si se quisiera extender la funcionalidad del programa, esta implementación extra sea de mayor sencillez.

3.- Estructura detallada del Programa

A continuación, se detalla el funcionamiento de cada una de las funciones y métodos que conforman el programa, desglosado por paquetes.

3.1.- santa_claus:

En primer lugar, encontramos el tipo de datos *SantaClausType*, el cual nos ofrece una serie de atributos*, que procedo a explicar:

1º is_working: Es un atributo de tipo booleano que nos indicará si Santa Claus se encuentra trabajando en un momento dado o no. Se trata de un atributo **privado**.

**atributo/método:* Utilizo esta nomenclatura consciente de que Golang no puede tratarse como un lenguaje puramente orientado a objetos. Sálvese el rigor que impediría utilizar este término, pues no forma parte de la discusión en esta memoria.

2º Elv_ch: Se trata de un canal de tipo struct{}, mediante el cual, los elfos podrán comunicar a Santa Claus que necesitan su ayuda con los juguetes. Es un atributo **público**, debido a que ha de ser accesible desde fuera de este paquete (en este caso, desde *toy_factory*).

3º Reind_ch: Se trata de otro canal de tipo struct{}. En este caso, servirá para que los renos puedan avisar a Santa Claus de que ya están listos para marchar a repartir los regalos. Del mismo modo que el anterior, y por los mismos motivos, este atributo es **público**.

4º mutex: Atributo **privado**, de tipo puntero a *sync.RWMutex*, es decir, almacena la dirección de memoria donde se almacena una variable de tipo *sync.RWMutex*. Su función es la de sincronizar los accesos a la variable *is_working* anteriormente mencionada.

A continuación, procedo a la explicación detallada de cada uno de sus métodos:

3.1.1.- Init: Este método se encarga de inicializar la estructura de santa claus. Inicializa ambos canales, el atributo *is_working*, y el mutex.

3.1.2.- WakeUp: Mediante este método, los elfos tienen la posibilidad de "despertar" a Santa Claus para que se ponga a trabajar. Este método, primero comprueba si Santa Claus está descansando y, si es así, envía un *flag* (un struct vacío) por el canal proporcionado a tal efecto (*Elv_ch)*.

3.1.3.- WaitForFinish: Este método, esencialmente, permite esperar a que Santa Claus finalice el trabajo que está realizando si así fuere. Es importante destacar que se trata de una función **bloqueante**, ya que si Santa Claus se encuentra trabajando en el momento que se llama a este procedimiento, el programa se quedará esperando hasta que éste finalice su trabajo.

3.1.4.- SetWorking: Mediante este método, es posible "configurar" a Santa Claus como "trabajando" o "descansando", recibiendo esta configuración en forma de parámetro booleano que se asignará al atributo *is_working*. Dicha asignación se encuentra protegida, como no puede ser de otra manera, haciendo uso del mutex interno que SantaClausType proporciona y que hice mención anteriormente.

3.1.5.- IsWorking: Se trata de un método que devuelve si Santa Claus se encuentra trabajando en el momento de su llamada. Devolverá "true" si está trabajando; y, "false", si no.

Cabe destacar que todos estos métodos mencionados son públicos.

3.2.- toy_factory:

Este paquete contiene la mayor parte de la implementación del programa.

En primer lugar, encontramos una serie de constantes que sirven para parametrizar al máximo el programa y que de este modo sea lo más escalable posible.

Para empezar, tenemos constantes que contienen el número de elfos y el número de renos. Posteriormente, se definen constantes que indican los intervalos de tiempo mínimo y máximo (en segundos) que ha de tardar cada reno en llegar después del anterior.

Después, se definen los tiempos mínimo y máximo (en segundos) que Santa Claus tarda en ayudar a un grupo de elfos, sucedido de los tiempos mínimo y máximo (en segundos) que se tarda en construir un juguete.

A continuación, nos encontramos con el porcentaje de fallo de un juguete, esto es, el porcentaje de que la fabricación de un juguete experimente un error.

Por último, tenemos otra constante que parametriza cuántos elfos conforman loq eu llamamos un “grupo”.

Lo que vemos a continuación son los tipos de datos necesarios en este paquete, los cuales detallo a continuación:

1º ElfType: Se trata del tipo de dato que modela a un elfo. Contiene los siguientes atributos:

- **problems:** booleano que indica si se encuentra en problemas (su juguete ha producido un error) o no.
- **is_working:** también de tipo bool, que nos indica si el elfo se encuentra trabajando a un momento dado o no (este atributo no es estrictamente necesario para el enunciado concreto de esta práctica pero me pareció adecuado para tener una estructura “Tipo Elfo” más robusta).
- **mutex:** un puntero a *sync.RWMutex* que permitirá mantener consistentes el resto de atributos.

2º ToyFactoryType: Es el tipo de dato que modela la fábrica de juguetes. Contiene los siguientes atributos:

- **santa_claus:** De tipo *santa_claus.SantaClausType.* Santa Claus, como es lógico, pertenece a la fábrica de juguetes.
- **elves:** Se trata de un array de *NElves* número de elfos de tipo ElfType.
- **reindeer_available:** Atributo de tipo int32 que representa la cantidad de renos que han llegado de sus vacaciones y que, por tanto, están disponibles.
- **elves_with_problems:** Atributo de tipo int32 que representa la cantidad de elfos que tienen problemas con su juguete.
- **Time_to_deal_ch:** Canal mediante el cual es posible notificar al programa principal que es el momento de repartir los regalos y, por tanto, hay que finalizar. Éste es el único atributo público debido a que ha de ser accesible desde el fichero *main.go*.

Procedo a la explicación de los métodos más relevantes:

3.2.1.- Init: Método público. Inicializa la fábrica de juguetes. Por ejemplo, los renos disponibles empieza siendo 0, al igual que los elfos con problemas; también se inicializa el canal *Time_to_deal* y cada uno de los NElves elfos mediante la llamada a *initElves*. Posteriormente, se lanzan dos "Go routines": *runSantaBehavior*, que se ocupa de ejecutar las tareas relacionadas con el comportamiento de Santa Claus; y *runReindeerBehavior*, que se encarga de llevar a cabo las tareas de los renos.

Como parámetro, recibe un mutex de tipo *sync.RWMutex*. Es importante explicar que este mutex no se trata del que está definido en la estructura ElfType sino que es un mutex "global" que se instancia en el programa principal (*main.go) y gracias al cual se mantienen coherentes los datos almacenados en los atributos pertenecientes a ToyFactoryType, tales como* elves_with_problems o *reindeer_available*.

3.2.2.- runElfBehavior: Método que se ejecuta como una rutina de Go desde *initElves*. Por tanto, habrá NElves rutinas de Go ejecutando este método.
Como argumentos, recibe un puntero al elfo sobre el cual se va a operar y el mutex global anteriormente mencionado. Su funcionalidad es la siguiente:

Si no está trabajando, si es la primera vez que comienza a trabajar y el juguete no da problemas, continúa trabajando hasta que pase el tiempo necesario de fabricación del juguete. Si, por el contrario, el juguete ha dado problemas, el elfo dejará de trabajar y se mantendrá esperando la ayuda de Santa Claus.

Es de relevancia indicar que es el propio elfo el que despierta a Santa Claus en el caso de que sea el último de su grupo de elfos en experimentar problemas con su juguete.

Por tanto, mientras el elfo permanece a la espera de la ayuda de Santa Claus, su atributo *is_working* se mantiene a *false* y su atributo *problems* permanece a *true*. Por consiguiente, esperar la ayuda de Santa Claus conlleva esperar a que el atributo *problems* se restablezca a *false* (tarea de la cual se encarga Santa Claus una vez que haya acabadod e ayudarle). Y, de este modo, el elfo volvería al trabajo (Se imprime por salida estándar la traza "[ELF] Back to work!").

3.2.3.- runReindeerBehavior: Método que consta de un bucle for que da Nreindeer vueltas (es decir, el número de renos) y a cada pasada ejecuta *reindeerBehavior*, que paso a explicar a continuación.

3.2.4.- reindeerBehavior: Su funcionamiento es simple. Genera un tiempo aleatorio entre los valores que definen el intervalo de llegada de un reno (recordemos que eso está definido en dos constantes) y se duerme durante ese tiempo. Posteriormente, si él es último reno, avisa a Santa Claus de que ya han llegado todos los renos. ¿Cómo lo hace? Introduciendo un struct vacío al canal Reind_ch que proporciona la estructura de datos SantaClausType (esto se hace mediante el atributo santa_claus definido en la estructura ToyFactoryType). Esto explica la necesidad de que este canal sea un atributo exportable, ya que si no lo fuera, no podríamos hacer uso de él desde este paquete.

3.2.5.- runSantaBehavior: Este método consta de un “select” dentrod e un “for”, para poder estar escuchando durante todo elr ato en los canales *Reind_ch* y *Elv_ch*.

Si llega un flag por *Reind_ch*, lo que se hace es esperar a que Santa Claus termine de hacer sus tareas y, posteriormente, notificar que es tiempo de repartir los regalos (es decir, de finalizar el programa). Esto se hace notificándolo mediante el canal *Time_to_deal_ch*.

Si llega un flag por *Elv_ch*, se ejecuta el método *helpToElves* como una Go routine, el cual se encarga de ayudar a los elfos y que a continuación explico.

3.2.6.- helpToElves: En primer lugar, se espera a que Santa termine sus tareas e inmediatamente después, se configura como “trabajando”. Después, se ejecuta un “Sleep” para dormir la ejecución durante el tiempo que dura la “reparación” de los juguetes y lo que hace a contiuación, una vez que este tiempo ha finalizado, es recorrer el array de elfos (almacenado en el atributo *elves* de *ToyFactoryType*) y “quitar” los problemas a los tres primeros que se encuentren; esto es, configurar su atributo *problems* a false, para que cada uno de esos elfos pueda continuar con la fabricación de juguetes. El acceso a esta variable se sincroniza con el mutex interno que posee la estructura ElfType.

Por útimo, se decrementa en tres unidades el número de elfos con problemas (*elves_with_problems*) y se configura a Santa Claus como “no trabajando” (*is_working* = false).

3.3.- main:

3.3.1.- main: Se declara e inicializa la variable *toy_factory* de tipo puntero a *toy_factory.ToyFactoryType*. También se declara el mutex al que he llamado “global” que posteriormente se pasa como argumento en la llamada al método Init de *ToyFactoryType*.

Se ejecuta *toy_factory.Init(&mutex)*, que comienza a realizar como el trabajo que expliqué anteriormente (esto concierne al paquete *toy_factory*).

Se ejecuta la fución *wait_for_end*, que detallo a continuación.

3.3.2.- wait_for_end: Esta función recibe como parámetro el canal *toy_factory.Time_to_deal_ch*, que es el canal por el cual se notifica que el programa debe finalizar. Se escucha de canal mediante la sentencia “select” y cuando llega el flag, se imprime la traza “[INFO] TIME TO DEAL!” y el programa acaba con estado de éxito.