




PRÁCTICA FINAL – JUNIO 2020


SISTEMAS DISTRIBUIDOS.


GRADO EN INGENIERÍA TELEMÁTICA – ETSIT, URJC

 **Profesor:** Felipe Ortega.

 **Versión:** 1.0.

 **Lenguaje de programación:** Go.

 **Fecha tope de entrega:** Miércoles, 22 de julio de 2020.

 **Envío:** Código fuente **comentado**, a través del espacio de entrega en Aula Virtual.

PREÁMBULO

1 OBJETIVO

El propósito de esta práctica es que el alumno/a demuestre sus conocimientos prácticos sobre varios problemas fundamentales de programación concurrente y sistemas distribuidos estudiados durante el curso.

2 ESTRUCTURA DE LA PRÁCTICA

La práctica consta de dos partes:

1. En la Parte **I**, se debe resolver un problema tradicional de programación concurrente, empleando el lenguaje Go: gorutinas, elementos de comunicación, sincronización y control vistos en clase.
2. En la Parte **II**, se debe resolver una extensión del problema anterior, que introduce conceptos de coherencia entre componentes de un sistema distribuido.

La calificación de ambas partes se realiza de forma independiente, *cada una sobre 10 puntos*. Posteriormente, para dar la nota final sobre un máximo de 10 puntos, se obtendrá la **suma ponderada** de ambas partes. **Cada parte aportará un 50% a la nota final.**

Inevitablemente, la realización de la segunda parte tiene ciertas dependencias respecto a la primera. En todo caso, se aplicarán criterios de corrección que valoren el trabajo en esa segunda parte con los elementos específicos evaluados (coherencia y sincronización en sistemas distribuidos), intentando separar fallos en la estructura o diseño de la aplicación que se arrastren desde la primera parte, en la medida en que sea posible.

PROBLEMA DE LA TRIBU CANÍBAL

3 ENUNCIADO DEL PROBLEMA

Se propone implementar mediante el **lenguaje Go** de programación el clásico problema de la **tribu caníbal**. Nos encontramos ante una tribu de caníbales que posee cierta reserva de **exploradores** para consumir. Los exploradores se cocinan en una **marmita** y hay un **cocinero** que es el único capaz de preparar más raciones de estofado cuando se acaban. Mientras que no se requieren sus servicios, el cocinero está descansando en su choza, esperando a que le avisen la próxima vez.

3.1 VARIABLES INICIALES

Con el fin de que nuestro programa **finalice** tras un número finito de iteraciones, supondremos que el **número de exploradores** es $N = 7$ (esta variable debe parametrizarse para poder cambiar fácilmente su valor).

Al ser cocinado en una marmita, cada explorador genera M **raciones** de estofado para los caníbales. En la práctica, suponemos que $M = 5$ (también debe parametrizarse). Finalmente, suponemos que la tribu cuenta con un **número de caníbales** $C = 9$ (que también se debe parametrizar).

3.2 DINÁMICA DEL PROGRAMA

Para comer, cada caníbal es capaz de servirse él solo de la marmita. Cada vez que un caníbal se sirve estofado, el número de raciones de la marmita decrece en una unidad. Cuando ya no quedan más raciones, el caníbal que ha ido a servirse y se encuentra la marmita vacía llama al cocinero para que vuelva a llenarla, cocinando otro explorador. Por simplicidad, vamos a suponer que el **tiempo** que se tarda en **cocinar cada explorador** es un valor *aleatorio*, entre 1 y 2 segundos.

En el **estado inicial** la marmita está vacía, por lo que el primer caníbal tiene que ir a llamar al cocinero para que la rellene. De nuevo por simplificar la situación, supondremos que el cocinero de la tribu es *vegetariano* y, por tanto, **no consume raciones** de la marmita durante toda la ejecución de la práctica. Una vez que cada caníbal se ha servido de la marmita, tardan **un tiempo aleatorio en comer**, que oscila entre 0.5 y 1 segundo. Por último, tras haber comido vamos a simular que **los caníbales trabajan** en otras tareas otro intervalo de tiempo *aleatorio*, que oscila entre 0.5 y 2 segundos, antes de volver a la marmita a por otra ración de estofado.

Cuando el cocinero ha acabado de rellenar la marmita, al ser esta una tarea muy cansada se retira a dormir a su choza, hasta que un caníbal que encuentre la marmita vacía le vuelva a llamar. Finalmente, cuando **se acaben los exploradores**, el *cocinero* debe **avisar a todos los caníbales** mediante algún mecanismo de que se han agotado las existencias, **finalizando** así la ejecución del programa.

3.3 RECOMENDACIONES DE IMPLEMENTACIÓN

La práctica consiste en implementar mediante gorutinas, canales y mecanismos de sincronización y control de ejecución concurrente vistos en clase, el escenario descrito en el enunciado anterior.

Se recomienda como siempre estructurar el programa en varias gorutinas, cada una de las cuales tenga un cometido claramente definido. Como *mínimo*, deberían implementarse los siguientes tipos de gorutinas (aunque nada impide que se hagan más):

- ▶ Una gorutina **caníbal**, que debe implementar todas las tareas propias de éste, como acudir a la marmita para tomar su ración, comer, despertar al cocinero cuando proceda o realizar otras tareas entre comida y comida. Debe finalizar su ejecución cuando el *cocinero* avise que ya no quedan más exploradores que consumir.
- ▶ Una gorutina **cocinero**, encargado de rellenar la marmita cuando le despierten, así como de avisar a los caníbales cuando ya no queden más provisiones de exploradores. Duerme en su choza mientras no se precisan sus servicios.
- ▶ La gorutina **main**, debe lanzar una gorutina por cada uno de los caníbales que tenemos que modelar, y otra gorutina que ejecuta el comportamiento del cocinero. El programa principal debe esperar a que terminen todas las restantes gorutinas antes de acabar.

Tanto la **marmita** como la **despensa** de exploradores pueden modelarse mediante canales del tipo apropiado. Su acceso se debe controlar de forma correcta para evitar resultados inconsistentes en la ejecución del programa.

Por último, hay ciertas condiciones importantes de ejecución del programa que se deben controlar, como por ejemplo:

- ▶ Evitar que se produzcan situaciones de interbloqueo entre las gorutinas.
- ▶ Evitar que se consuman indebidamente raciones de la marmita (por ejemplo, que dos o más caníbales consuman la misma ración).
- ▶ Asegurar que solo se llame al cocinero cuando resulte estrictamente necesario.

4 CRITERIOS DE EVALUACIÓN

Los criterios de evaluación para esta parte de la práctica serán los siguientes:

- ▶ **Corrección funcional** en la ejecución del programa, es decir, que las diversas tareas y agentes se ejecuten ajustándose a las especificaciones indicadas en el enunciado: **5 puntos**.
- ▶ **Elementos de programación concurrente** utilizados correctamente, para implementar las restricciones de acceso y sincronización especificadas en el enunciado: **3 puntos**.
- ▶ **Modularidad y estructura** general del código, incluyendo comentarios y desglosando funciones específicas en métodos individualizados: **2 puntos**.

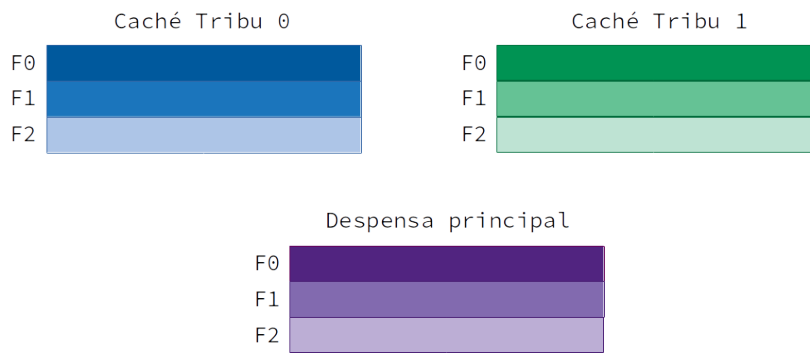
COHERENCIA ENTRE VARIAS TRIBUS

5 ENUNCIADO DEL PROBLEMA

En esta segunda parte, vamos a considerar un **problema de coherencia** que involucra a **dos tribus de caníbales**, cada una con idénticas condiciones y restricciones de funcionamiento a las ya indicadas en la Parte I.

Existe una **despensa principal** de exploradores capturados por las dos tribus. La despensa cuenta con 24 espacios para encerrar exploradores, repartidos en tres filas de 8 espacios cada una. La Figura 1 representa esquemáticamente este entorno.

Figura 1: Esquema de las *cachés* de cada tribu, para que cada uno de los dos cocineros lleve el control de los exploradores consumidos por su tribu, y la *despensa principal*, para mantener a los exploradores capturados por ambas tribus de caníbales.



5.1 DINÁMICA DEL PROGRAMA

El procedimiento que siguen los cocineros de cada tribu es el siguiente:

- ▶ Al comienzo, la *caché* de cada tribu está vacía. Así que el cocinero elige aleatoriamente cargar información de una de las tres filas de la despensa principal. Seguidamente, el **cocinero elige** aleatoriamente uno de los exploradores disponibles en esa fila para cocinarlo.
- ▶ Inmediatamente después de elegir un explorador concreto, avisa de que se ha consumido dicho explorador en la posición seleccionada, siguiendo un

protocolo *write through*. El aviso debe actualizar la información tanto en la *despensa principal* como en la *caché* de la otra tribu, tal y como se muestra en la Figura 2.

- ▶ En la siguiente iteración en la que el cocinero de esa tribu necesite otro explorador, carga aleatoriamente información de otra de las dos filas **que aún no ha usado**. Seguidamente, se repite el procedimiento descrito en los puntos anteriores para consumir un explorador elegido al azar y actualizar la despensa principal y la caché de la otra tribu, de modo que reflejen el cambio de valor para la posición seleccionada.
- ▶ En la siguiente iteración del cocinero de esa tribu, carga la información de la **fila restante que no se ha usado todavía** en esa tribu, y repite las operaciones antes descritas.
- ▶ A partir de que las tres filas estén ya cargadas en la caché de la tribu, en las siguientes iteraciones el cocinero elige una fila al azar para extraer, también al azar, un explorador al azar de una posición de la misma. Se sigue de nuevo el protocolo *write through* para actualizar la despensa principal y las otras cachés.
- ▶ El programa acaba cuando ya no quedan más exploradores en la despensa principal.

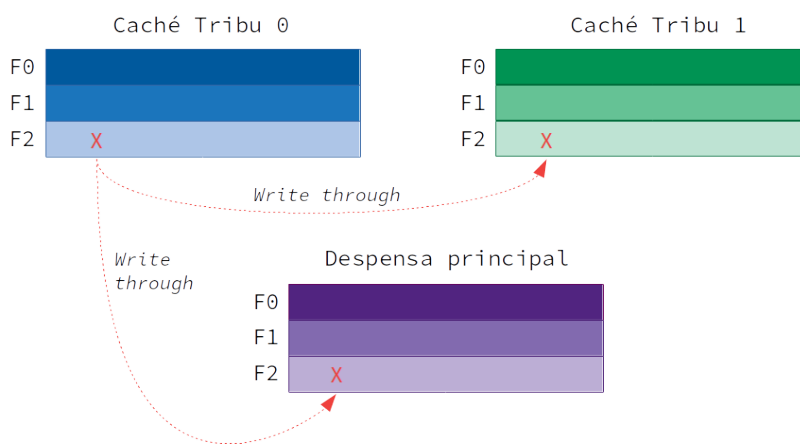


Figura 2: Esquema del procedimiento de actualización *write through* que deben seguirse para mantener la coherencia entre las cachés locales de cada tribu y la despensa principal.

5.2 RECOMENDACIONES DE IMPLEMENTACIÓN

Se pide implementar mediante el lenguaje Go el funcionamiento descrito para la despensa principal y las cachés de las dos tribus, de modo que se eviten inconsistencias de datos entre las diversas copias de las filas, tales como: no consumir un explorador dos veces, no consumir un explorador que acaba de ser consumido previamente por la otra tribu, etc.

Para ello, será necesario utilizar canales o cualquier otro medio de sincronización entre las gorutinas de cada cocinero para garantizar en todo momento la consistencia de datos. Se puede crear, por ejemplo (aunque no es obligatorio), una

gorutina *controlador* que centralice la lógica de avisos, sincronización y control de consistencia entre las diversas copias.

6 CRITERIOS DE EVALUACIÓN

Los criterios de evaluación para esta parte de la práctica serán los siguientes:

- ▶ **Corrección funcional** en la ejecución del programa, es decir, que las diversas tareas y agentes se ejecuten ajustándose a las especificaciones indicadas en el enunciado: **5 puntos**.
- ▶ **Elementos de control de coherencia** utilizados correctamente, para implementar el algoritmo *write through* de actualización de las diferentes copias de datos: **3 puntos**.
- ▶ **Modularidad y estructura** general del código, incluyendo comentarios y desglosando funciones específicas en métodos individualizados: **2 puntos**.

