# **27 DE MAYO DE 2020**



# **PRÁCTICA 2:**

# **EXCLUSIÓN MUTUA**

## **SISTEMAS DISTRIBUIDOS 2019-2020**

DAMIÁN MALENO GONZÁLEZ

GEOVANNY RISCO CARRERA

# Decisiones de diseño

# Análisis de los gráficos

# Reparto de trabajo

Se utilizó un repositorio en Github para trabajar de manera conjunta. Se puede acceder mediante mediante el siguiente link <https://github.com/geovalexis/task1-SD.git>, aunque tal vez no tengas acceso porque es un repositorio privado. En caso de que sea necesario lo podemos hacer público o bien darte acceso como contribuidor.

En todo momento se trabajó de forma coordinada discutiendo y trabajando el diseño del programa asi como elaborando el código en Python utilizando *Visual Studio Code*. En general se puede decir que cada uno participó mayoritariamente en las siguientes funciones:

* Damian Maleno: función de inicialización y análisis de resultados.
* Geovanny Risco: funciones map y reduce, y decisiones de diseño.

# Preguntas

Respuestas a las siguientes preguntas sobre la exclusión mutua.

* Muchos algoritmos distribuidos como el propuesto es esta práctica requieren del uso de un proceso ***master***o **coordinador**. ¿En qué medida se pueden considerar realmente estos algoritmos como distribuidos? Justificar.

Cuando hablamos de algoritmos centralizados, normalmente existe un proceso fijo que actúa como *master* o coordinador. El hecho de que los procesos sean ejecutados en diferentes máquinas hace que el algoritmo pueda decirse distribuido. En los algoritmos distribuidos que no disponen de un proceso coordinador fijo, este es elegido mediante algoritmos de elección de líder entre los diferentes procesos que forman parte del algoritmo, entre estos podemos encontrar algoritmos de elección de líder aleatorios, en anillo, en malla… Por ello podemos afirmar que la presencia de un proceso *master* o coordinador en un algoritmo no hace que este sea más o menos distribuido.

* Ahora supón que la función ***master*** falla. ¿Esto siempre conlleva la violabilidad de la exactitud del algoritmo? ¿Si no, bajo qué circunstancias ocurre? ¿Existe alguna forma de evitar el problema y hacer el sistema tolerante a fallos del coordinador? Justificar.

Supongamos que el proceso *master* o coordinador participa en un algoritmo en el cual los *slaves* solicitan acceso a uno o varios recursos y estos son respondidos inmediatamente con un mensaje de solicitud aceptada o denegada. Si nos encontramos en una fase del algoritmo en el que no existen procesos *slave* que estén accediendo a los recursos y no haya solicitudes pendientes o encoladas, entonces no conllevará la violabilidad de la exactitud del algoritmo.

En cambio, si nos encontramos en una fase del algoritmo en la que se siguen realizando solicitudes de acceso a los recursos, entonces un fallo en el coordinador si que implicaría la violabilidad de la exactitud del algoritmo.

Como hemos indicado en la pregunta anterior, muchos sistemas distribuidos poseen algoritmos de elección de líder. Por ello, si un proceso solicita el recurso al coordinador y no recibe respuesta, este puede iniciar la elección de un nuevo coordinador. Es sistema podría hacer incluso más tolerante a fallos del coordinador si este antes de conceder/denegar el recurso guardase las solicitudes entrantes en disco, de esta manera en caso de fallo el nuevo líder podría reconstruir la lista de solitudes y de recursos en accedidos leyendo del disco.

* En el algoritmo propuesto, los permisos de escritura se otorgan en el orden en que son solicitados, de manera que no hay *slaves* que esperan infinitamente (*no starvation*)*.* ¿Si la función *master* eligiese las funciones *slave* de manera aleatoria, podrían las funciones *slave* sufrir *starvation*?

Entendemos el concepto de *starvation* como la incapacidad de poder asignar un recurso a un proceso dado. Supongamos que la función *master* elige las funciones de manera aleatoria, puesto que la función aleatoria posee una distribución uniforme (todos los *slaves* tienen la misma probabilidad de ser elegidos) las funciones *slave* no podrán sufrir de inanición.

Pero surge el siguiente problema; si los procesos que esperan acceder al recurso no so encolados con o sin prioridad y un proceso aleatorio es otorgado con el acceso al recurso a cada vez, es posible que ciertos procesos esperen durante mucho tiempo. Realmente el problema con esta estrategia no es que los procesos tengan que esperar infinitamente, sino que no se sabe cuando los procesos tendrán accesos al recurso. Un ejemplo de esto podría ser la comunicación Ethernet, en la que los procesos se comunican mediante un medio común sin una sincronización general, los procesos solicitan el medio y si este no está libre han de esperar durante un intervalo arbitrario y volver a solicitar el medio (distribución parecida a la aleatoria), cuando el medio es muy solicitado se pueden observar grandes retrasos.

# Referencias

* Funcionamiento e implementación de pywren: <https://github.com/pywren/pywren-ibm-cloud>
* Programación en Python: <https://runestone.academy/runestone/books/published/pythonds/index.html>
* Librería de Python para las operaciones matemáticas: <https://docs.scipy.org/doc/numpy/index.html>
* Diapositivas, fórum y apuntes de clase.

# ANEXO I: Gráficos