DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 2

## Sistemas distribuidos

Sistemas Operativos
Primer Cuatrimestre de 2018


| Integrante | LU | Correo electrónico |
|---|---|---|
| Nicolas Bukovits | 546/14 | nicko_buk@hotmail.com |
| Kevin Frachtenberg | 247/14 | kevinfra94@gmail.com |
| Laura Muiño | 399/11 | lauramuino2@gmail.com |





Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

**Resumen**

En el siguiente trabajo práctico, se realizó una implementación de un protocolo de envío de mensajes entre procesos concurrentes. En el mismo se desarrolló un sistema de blockchain.

# Índice

# 1. Analisis del protocolo

## 1.1. ¿Puede este protocolo producir dos o más blockchains que nunca converjan?

En este ejercicio se completó la implementación del método push_front de la lista atómica provista por la cátedra. El método fue crear un nuevo nodo. Al nuevo nodo se le cargó como nodo siguiente el nodo inicial actualmente. Se compara atomicamente el valor del actual nodo inicial con el siguiente del nuevo nodo inicial. Si son iguales reemplaza el valor del actual nodo incial con el nuevo nodo creado. Si no son iguales se vuelve a recuperar el nodo inicial y se intenta nuevamente.

Luego se procedió a implementar la clase ConcurrentHashMap, la cual contiene un constructor que crea una tabla con 26 entradas (una por cada letra del abecedario) en la cual en cada una hay una lista de pares `<string, entero>` para que la tabla sea de hashing abierto y en caso de colisión se agregue la nueva palabra al principio de la lista. La función de Hash es simplemente tomar la primera letra de la palabra, por lo que para cada existe una entrada en la tabla. Además de crearse e inicializarse la tabla, se crea un array de 26 mutex llamando a la función `pthread_mutex_init` para que durante el uso de threads, podamos asegurarnos que solo uno escribe la tabla al mismo tiempo.

Si bien el constructor no era parte de la especificación de ConcurrentHashMap, para darle consistencia a la clase se optó por implementarlo, aunque luego de terminar de desarrollar la mayor parte de los métodos solicitados, de notó que habia un problema en las referencias al devolver una copia del ConcurrentHashMap asi que por sugerencia de los docentes se definió quitar el destructor.

La implementación de `addAndInc(string key)` consiste de obtener, en primer lugar, la posición en la tabla usando la función de hash. Luego, se pide el mutex de la entrada correspondiente y si la clave no esta definida se agrega al principio de la lista el par (key,0). Se realiza un unlock de la entrada y se recorre toda la lista de la posición en la tabla usando el iterador. Cuando se encuentra a la clave se pide el mutex y se incrementa en uno el valor entero del par (clave,valor). Luego se vuelve a desloquear la entrada.

El método `member(string key)` simplemente recupera la posición en la tabla y con el iterador recorre la lista. Si encuentra la clave definida devuelve verdadero.

El último método perteneciente a ConcurrentHashMap, `maximum(unsigned int nt)` : devuelve el par (k, m) tal que k es la clave con máxima cantidad de apariciones y m es ese valor. Los `nt` threads procesaran una fila de la tabla. Si no tienen filas por procesar terminarán su ejecución. La implementación consistió en definir un entero atómico y un Elem atómico que se van a usar para llevar registro de la cantidad de posiciones de la tabla a analizar y el elemento máximo encontrado. Se crean los nt threads, se pide el mutex de todas las entradas de la tabla, se cargan todas las estructuras auxiliares que van a usar los threads y se lanzan a correr los threads a los cuales se le pasa como paŕametro una función max_thread. La misma incrementa en uno atómicamente a la variable siguiente y mientras siguiente sea menos que 26, se recorre la lista en cada posicion y se va actualizando atomicamente el valor del Elem atómico que representa el máximo. Se espera a que todos los threads terminen y se hace un unlock de todas las entradas de la tabla.

## 1.2. ¿Cómo afecta la demora o la pérdida en la entrega de paquetes al protocolo?

En este ejercicio se implementó una función ConcurrentHashMap `count_words(string arch)` que toma un archivo de texto y devuelve un ConcurrentHashMap cargado con las palabras del archivo. Las palabras se consideran separadas por espacio. Por especificación, no es concurrente. Su funcionamiento consiste en abrir el archivo y llamar a la función getline que toma el arhivo, un string y un caracter delimitador. Mientras haya palabras para leer se cargan en el string y por cada una de ellas se agregan al ConcurrentHashMap usando la función addAndInc implementada en el ejercicio anterior. Una vez que se hayan leido todas las palabras se cierra el archivo. Se puede asegurar que esta funcion no es concurrente ya que si bien addAndInc soporta el uso de multithreading, en esta función no se crean threads.

## 1.3. ¿Cómo afecta el aumento o la disminución de la dificultad del Proof-of-Work a los conflictos entre nodos y a la convergencia?

Se agregó el archivo test-1.cpp el cual incluye tests particulares sobre ConcurrentHashMap. El archivo tiene tres partes en el cual se prueban las funciones member,addAndInc y maximum del ConcurrentHashMap, las cuales para poder ejecutarse prueban también el constructor del mismo. Para poder realizar los tests fue necesario agregar funciones auxiliares que se usan para este fin las cuales son add, Inc y count_word. Estas funciones fueron agregadas para testear member,addAndInc y maximum sin asumir que las demas funcionan excepto el maximum. Los tests se desarrollaron de manera incremental empezando con member, luego una vez que se testeo correctamente member, se testeó addAndInc y finalmente maximum.