DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 2

24 de junio de 2014 — Sistemas Operativos



| Integrante | LU | Correo electrónico |
|---|---|---|
| Silvio Vileriño | 106/12 | svilerino@gmail.com |
| Ezequiel Gambaccini | 715/13 | ezequiel.gambaccini@gmail.com |
| Martin Arjovsky | 683/12 | martinarjovsky@gmail.com |



| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |





**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) 4576-3300
http://www.exactas.uba.ar

# Índice

**Resumen**

INTRO ABSTRACT

# 1. Diseño: Modelo propuesto para servidor multihilo

Implementamos el servidor multi en base al servidor mono.

## 1.1. Funcionamiento del servidor mono

El servidor mono atiende un cliente a la vez en base al orden de llegada, y hasta que no termina con el cliente que está atendiendo, no pasa a atender al próximo.

El protocolo cliente-servidor que se utiliza es el siguiente:

1. El cliente indica su nombre y posición inicial. Para mantener la simplicidad esta posición no se chequea, por lo que será el cliente el encargado de enviar una posición válida.
2. El cliente indica el próximo movimiento, enviando ARRIBA, ABAJO, IZQUIERDA, DERECHA. El servidor recibe el pedido y responde OK u OCUPADO según corresponda.
3. Las partes repiten este ciclo hasta que el cliente haya salido del lugar. Una vez que esto ocurre, el servidor espera por un rescatista para el alumno y cuando lo obtiene, le coloca la máscara y envía al cliente LIBRE!

Si la conexión se interrumpe en medio de la huida, el servidor da de baja al cliente automáticamente. Ejemplo:

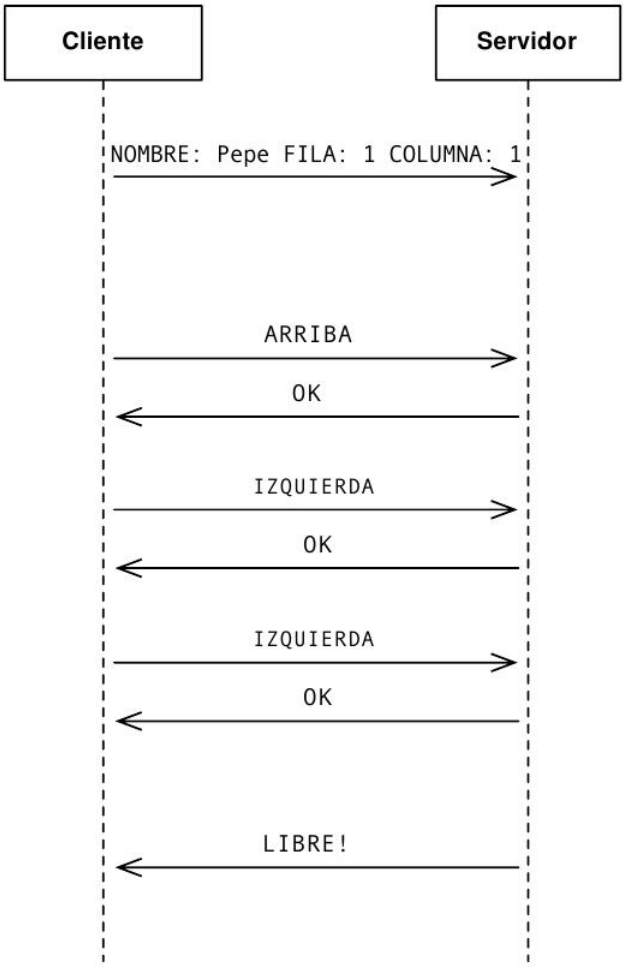


Figura 1: Ejemplo de escape exitoso (con puerta en posicion 0, -1).

## 1.2. Implementación del servidor multi

Nuestro servidor multi consiste de un thread principal y un thread worker por cada cliente conectado, separando en hilos diferentes las operaciones que realizaba el servidor mono, como se explica a continuación:

- El thread principal se encarga de detectar la conexión de nuevos clientes, tras lo cual crea un worker para que los atienda.
- El worker se encarga de atender a los clientes usando el protocolo cliente-servidor especificado anteriormente, recibiendo como parámetros el socket donde se conectó el cliente, y la estructura compartida tipo `t_aula` aula.

Para sincronizar los accesos a los diferentes recursos compartidos del aula, y prevenir race conditions, se utilizaron diferentes estructuras de sincronización del estándar pthread, `pthread_mutex_t` y `pthread_cond_t`, en las funciones `t_aula_ingresar`, `t_aula_liberar`, `intentar_moverse`, y `atendedor_de_alumno`. La variable de condición solo fue utilizada para la función `atendedor_de_alumno`.

Una race condition es cuando varía la ejecución dependiendo del orden de acceso a un recurso compartido, si este orden no es el esperado. Para evitar esto, se fuerza un orden determinado mediante diferentes estructuras de sincronización, como mutex y semáforos.

Se utilizan en total 3 mutex y 1 variable de condicion:

- Se utiliza un mutex para sincronizar el acceso a la matriz de posiciones del aula.
- Se utiliza otro mutex para sincronizar el acceso a la cantidad de personas del aula.
- Por último, se utilizan en conjunto un mutex y una variable de condición para sincronizar el acceso a los rescatistas del aula.

Las diferentes funciones especificadas anteriormente acceden a recursos compartidos con todos los clientes, por lo que se usan las estructuras antes especificadas para que el acceso al recurso sea hecho de manera atómica y ordenada.

Una consideración especial que tomamos para nuestro servidor multi fue validar que la posición inicial este dentro del rango de la matriz, y que la celda correspondiente a esta posición tenga lugar disponible para el nuevo cliente. Si no se cumpliera alguna de las condiciones anteriores, se le envía un mensaje de error al cliente, se cierra la conexión y finaliza la ejecución de ese worker particular. Para realizar esto también se tuvo que tener en cuenta posibles race conditions, por lo que se sincronizo el acceso con los mutex especificados anteriormente.

## 1.3. Funcionamiento de los threads

Las operaciones que realiza el hilo principal son las siguientes:

1. Crear el socket servidor. bindearlo y empezar a escuchar.
2. Inicializar las estructuras de sincronización.
3. Quedarse en un loop infinito aceptando conexiones entrantes, creando workers en modo DETACHED para que atiendan las nuevas conexiones realizadas, pasando como parámetro el socket de la conexión y un puntero al aula. Se los crea en modo DETACHED ya que no importa cuando terminen ni su valor de retorno, son independientes al thread principal.

Las operaciones que realiza un hilo worker son las siguientes:

1. Recibir el nombre y posición del cliente.
2. Tratar de hacer entrar al alumno (nuevo cliente), al aula. Si lo logra, se prosigue con la operación normal, en caso contrario, como se especificó anteriormente, se envía un mensaje de error al cliente, se cierra la conexión, y se finaliza la ejecución de ese worker particular.

3. Luego, el cliente manda una dirección y el servidor le dice si puede avanzar o no. Se repite este ciclo hasta que el cliente sale. Las funciones de acceso al aula son de acceso atómico gracias a los mutex.

4. Mediante la combinación mutex/variable de condición, el worker se queda esperando hasta que la cantidad de rescatistas disponibles sea mayor a 0, para luego marcar que el alumno tiene la mascara puesta.

5. Finalmente, el worker emite un broadcast a la variable de condición para indicar que terminó, incrementa en 1 la cantidad de rescatistas, avisa al cliente que es libre, cierra la conexión, y finaliza su ejecución.

Dado que en la implementacion monothread, podia haber un único cliente moviendose por la matriz, salvo que la cantidad de personas por posicion fuera menor que uno, no había problemas de saturación al momento de entrar. Ahora, dado que puede haber muchos clientes moviendose por la matriz, se debe tener en cuenta la cantidad de personas actualmente en una posicion $(x, y)$ ante un pedido de movimiento o de ingreso al aula. En esta situación, se debe validar que la posición destino a donde va a terminar el cliente que solicito el pedido, este en rango y ademas que haya lugar para esa persona. Esto provocó tener que crear un nuevo tipo de mensaje de respuesta del servidor (sin modificar el protocolo ya existente) que ante el pedido de ingreso al aula de una posicion $(x, y)$ saturada, se le niegue el ingreso devolviendo dicho nuevo estado `POSICION_LLENA_O_FUERA_DE_RANGO`. Esto no modifica de ninguna forma el protocolo existente, dado que el cliente que vino en el bundle no valida errores. Si el servidor original devolvía errores (el mono devuelve error y el cliente no valida) el cliente no tiene comportamiento definido ante esto. Una forma de solucionar este problema es definir que `POSICION_LLENA_O_FUERA_DE_RANGO` sea equivalente a `ERROR`, cosa que ya estaba contemplada en el protocolo original presentado.

El tester python otorgado, tampoco validaba si el servidor devolvía `ERROR`, y se limitaba a enviar cíclicamente mensajes de forma serial a todos los sockets abiertos. Cuando el servidor devuelve error, cierra el socket y cuando el tester envia datos en el proximo ciclo a un socket cerrado hay un error. Para evitar esto, se validaron las respuestas del servidor, de forma tal que cuando un cliente era liberado/había error, se eliminaba del ciclo de proceso de peticiones.

Para testear paralelismo, se corren varios testers de python en paralelo usando bash. En este archivo se pueden encontrar variables de ajuste de parametros tales como cantidad de clientes por thread y cantidad de threads, lo que da una carga del servidor igual a *cantidad de threads* $\times$ *cantidad de clientes por thread*

## 2. Testing: Analisis de escalamiento

## 3. Apéndice: Entregable

Dado que la unica modificacion fue la creacion de un servidor paralelo servidor_multi. Con solo compilar tipeando `make clean all` se obtendran todos los ejecutables necesarios para llevar a cabo cualquier prueba o ejecucion del sistema. Para ejecutar el servidor se debe tipear `./servidor_multi` y para realizar las pruebas `python server_tester.py`. Los parametros modificables que permiten varios tipos de pruebas del servidor se encuentran en `biblioteca.h` y los de testing en `python server_tester.py`. Se realizo una automatizacion de los tests en scripts para agilizar el proceso de analisis, los mismos se encuentran en la carpeta `codigo`.