Trabajo práctico N. 1
Sistemas Operativos - I.T.B.A

Marcantonio, Nicolás
Raies,Tomás A.
Saqués,M. Alejo

Resumen

En el presente informe, describiremos brevemente el problema que decidimos abordar, junto a aquellas decisiones de implementación que hemos tomado a los efectos de construir nuestro trabajo.

1. *Buscaminas* multijugador

El objetivo del trabajo fue desarrollar una versión del conocido juego *Buscaminas*, en la que potencialmente N jugadores puedan participar concurrentemente. En estas condiciones, un jugador gana en el caso que $N - 1$ jugadores pierdan, o cuando la ventaja del primer jugador no puede ser superada por los demás.

Cada jugador no solo puede ver en su pantalla aquellas zonas descubiertas por él mismo, sino que verá también aquellas descubiertas por otros jugadores, cada una de ellas en un color distintivo. De esta forma, se ha logrado un efecto secundario en la dinámica del juego: la presión para tomar decisiones rápidas y correctas es mucho mayor que en el juego original. En el último, el tiempo corre como una medida para la superación personal. En nuestro juego, si no se es lo suficientemente rápido, se corre el riesgo de que otro gane antes que uno.

2. Esquema de funcionamiento

A continuación, describiremos los detalles del proceso de comunicación entre un cliente y el servidor, tras lo cual argumentaremos el porqué de las decisiones tomadas.

2.1. Explicación

Para que un cliente pueda comunicarse con el servidor y viceversa, hemos diseñado un esquema mediante el cual cada cliente posee dos *agentes* para comunicarse con el servidor, uno que le proporciona los datos enviados por el mismo, y otro que le toma sus datos para remitirlos al servidor.

Inicialmente, el servidor abre su canal de comunicaciones y escucha N conexiones entrantes en dicho canal. El cliente, al conocer la dirección en la cual el servidor está escuchando, procede a entablar una conexión con el mismo. El servidor, al recibir el intento de conexión por parte de un cliente, crea dos *threads*: un *Attender*, que correrá la rutina `attend()`, y un *Informer*, que ejecutará la rutina `inform()`. Ambos *threads* se comunican con el servidor maestro mediante *pipes* que el último crea para tal propósito. Este mismo proceso sucede $\forall x, 1 \leq x \leq N$, siendo x el identificador de un cliente.

Un *Attender* es un *thread* que espera mensajes del cliente a quien está atendiendo. Al recibir un mensaje, el *Attender* lo redirecciona al servidor maestro, el cual está constantemente esperando actualizaciones de los *Attenders* utilizando la *syscall* `select()` sobre los pipes en los cuales los mismos notifican al servidor.

Por otro lado, un *Informer* es otro *thread* encargado de enviarle a su cliente los mensajes que el servidor maestro tiene para el último. De esta forma, dicho *Informer* está constantemente esperando actualizaciones del servidor maestro, utilizando la *syscall* `select()`.

Además, un *Greeter* se encarga de recibir las conexiones entrantes cuando una partida ya esta en juego. Simplemente creara la conexion para luego enviar un mensaje que le avise al cliente que el servidor esta ocupado.

Por su parte, para un momento dado, el cliente puede o bien haber realizado alguna acción, o bien estar esperando a que el servidor le envíe alguna actualización. En el primero de los casos, el cliente pedirá al servidor que actualice el contexto de juego en base a la casilla que solicitó descubrir. En este caso, el servidor le responderá con las casillas que descubrió tras realizar la acción, o bien informándole que la casilla que seleccionó contenía una bomba. Por otro lado, si el jugador en dicho momento no realizó ninguna acción, continuará escuchando mensajes del servidor. Por ejemplo, el servidor le informará a cada cliente de los descubrimientos de otros jugadores.

Nótese que un cliente no conoce en absoluto que no se está comunicando directamente con el servidor, sino que dos *threads* están actuando de mediadores. Él utiliza la conexión que estableció en un principio con el servidor maestro, y nunca advierte ninguna diferencia al respecto.

2.1.1. Argumento

Al separar las tareas de recibir y enviar mensajes de un servidor para cada uno de los clientes conectados al mismo, creamos un escenario para un buen uso de una herramienta que provee el sistema operativo, la *syscall* `select()`. Es decir, estamos delegando en el sistema operativo la responsabilidad de detectar la existencia de nuevos mensajes no solamente del lado del servidor, sino también del lado del cliente. Otras alternativas, tales como crear un único *thread* para atender a un cliente, hubieran significado para un cliente la necesidad de pedir cada un *delta* de tiempo actualizaciones al servidor, el cual podría o no tenerlas. Con la estrategia utilizada, la llamada a `select()` por parte del cliente evita tener que hacer *busy waiting*, ya que una invocación exitosa de dicha *syscall* confirma la existencia de un mensaje entrante.

Para la IPC entre el servidor y sus *threads*, optamos por utilizar *pipes* en lugar de otras alternativas como compartir direcciones de memoria en el *heap* porque nos permiten lograr una correcta sincronización *multithreading* sin necesidad de *mutex* o *semáforos*. Si bien es cierto que esto nos lleva a duplicar los *buffers*, lo que es caro tanto en memoria como en tiempo de ejecucion, preferimos esta alternativa para mantener el código simple y reducir el riesgo de caer en los errores tipicos de la programacion concurrente, como codiciones de carrera y *deadlocks*.

Finalmente, sincronizamos el servidor con el *Greeter* usando dos *mutex* para evitar que ambos realizen llamadas a `mm_accept()` al mismo tiempo, y solo se rechacen las conexiones cuando el servidor este ocupado.

3. Capa de comunicación

A continuación, enunciaremos aquellas características que merezcan observarse a propósito de la implementación de la capa de comunicaciones.

Para evitar ser reiterativos, aclaramos que la capa de comunicaciones asume que quien desea conectarse conoce la dirección en donde debe hacerlo.

3.1. Implementación con *FIFOs*

En la implementación de la librería de comunicaciones con *named pipes*, la llamada a `mm_listen()` retorna un `Listener` que contiene el *file descriptor* correspondiente al *FIFO* en el cual el invocador esperará conexiones entrantes. Esto último se realiza invocando la función `mm_accept()`.

La primera función recibe como argumento el *path* en el cual dicho *FIFO* debe crearse, donde, asimismo, otro proceso que quiera establecer una conexión con el primero deberá utilizar. La segunda, solicita que se le pase como argumento el `Listener` que la primera ha creado.

Al realizar la llamada a `mm_connect()`, el proceso que desea conectarse abre el *FIFO* pasado como argumento con permisos de escritura, y genera a su vez dos *FIFOs* adicionales que serán utilizados en la nueva conexión: uno de ellos será en el que uno de los procesos escribe y el otro lee, mientras que en el otro *FIFO*, los roles se encuentran invertidos. De esta forma, quien invoca a `mm_connect()` le envía al proceso que está aceptando conexiones entrantes dichos *FIFOs*, escribiendo sus direcciones en el *named pipe* en el que el segundo está escuchando nuevas conexiones. De esta forma, cada nueva comunicación posee únicos *file descriptors* de lectura y escritura, un par para cada uno de los dos procesos que participan de dicha conexión.

3.2. Implementación con *Sockets*

Hemos considerado realizar implementaciones utilizando *sockets* tanto `AF_UNIX` como `AF_INET`. Ambas implementaciones son similares en naturaleza, en el sentido que ambas utilizan las mismas llamadas al *API* de *sockets* de UNIX. La diferencia radica en que el *binding* de la primera se realiza sobre un nombre en el *filesystem*, mientras que la segunda se realiza sobre un puerto determinado. Al final, nos decidimos por los sockets `AF_INET` porque nos permite correr la aplicacion como estuvo pensada: en tiempo real y en máquinas diferentes.

Para la llamada de `mm_listen()`, la función espera que se le pase un *string* conteniendo el número del puerto sobre el que se debe realizar el *binding*. Por otro lado, la invocación a `mm_connect()` espera un string que contenga tanto la dirección IP del *host*, como así también el puerto en el cual se debe establecer la conexión. Por ejemplo, si un cliente corriendo en la misma máquina en la que se encuentra el servidor quiere conectarse al mismo, y conoce que el puerto en el que se encuentra el último es el 27456, deberá pasar como argumento de `mm_connect()` el *string* `localhost:27456` o, lo que es lo mismo, `127.0.0.1:27456`.

4. Instrucciones de uso

Generar los ejecutables de servidor y cliente con `make` (asegurarse de poseer un directorio llamado `bin` previamente). Ejecutar `./bin/server.out n`, siendo `n` la cantidad de jugadores que van a participar en la partida (por defecto, 1). Luego, ejecutar `./bin/client.out`.

Debe notarse que, por defecto, el archivo `config` contiene la información necesaria para conectarse por *sockets* `AF_INET` en la red local. Si se quisiese cambiar esto, se debería especificar en el tercer campo de la segunda línea de dicho archivo la dirección de la computadora *host*.