Universidad de Buenos Aires
Facultad De Ingeniería
Año 2015 - Primer Cuatrimestre

75.59 - Técnicas de Programación Concurrentes I

FECHA: 02/07/2015
TEMA: Trabajo Práctico número 2

Apellido y Nombre	Padrón	Correo
Aceto, Ezequiel Leonardo	84316	ezequiel.aceto@gmail.com
Opromolla, Giovanni	87761	giopromolla@gmail.com

Índice

Objetivo

El objetivo de esta aplicación es crear una arquitectura cliente / servidor que permita a los clientes consultar e ingresar un registro dentro de una base de datos. La comunicación entre procesos se realizará usando sockets, y el servidor tendrá la capacidad de atender a varios clientes simultáneamente..

Análisis del problema

Se plantea el desarrollo del proyecto dividiéndolo en tres partes.

1. **Biblioteca compartida:** Contiene el código compartido por ambas plataformas. Principalmente lo referente al modelo, la base de datos y a clases auxiliares para el manejo del proyecto.
2. **Cliente:** Aplicación cliente que se comunica con el servidor mediante un socket y provee una consola para realizar operaciones de consulta e inserción con instrucciones similares a las del lenguaje SQL.
3. **Servidor:** Aplicación servidor que escucha conexiones entrantes a un socket y atiende las peticiones de los clientes. Realiza el acceso a la base de datos.

El alcance de la implementación de la base de datos se limita a un registro del tipo Person con las especificaciones que abajo se mencionan. El mismo sistema puede ser extendido para manejar otro tipo de datos.

Person

nombre	dirección	teléfono
varchar(61)	varchar(120)	varchar(13)

Resolución

Implementación del servidor

El servidor fue implementado utilizando un único proceso y sin threads. Para conseguir que el mismo tenga la capacidad de atender a múltiples clientes simultáneamente se siguió los lineamientos utilizados por el servidor NGINX.

El principio utilizado para atender varios sockets en un solo proceso es el de *non-blocking I/O Multiplexing*. Las operaciones de I/O bloqueantes son aquellas en las que la ejecución de un programa se detiene a la espera de que la operación se complete.

Una operación de I/O asincrónica no bloqueante devuelve el control al proceso que realizó la llamada de forma inmediata. Luego el sistema envía de algún modo una notificación al proceso que realizó la llamada, para indicarle que la operación de I/O se ha completado.

Por ejemplo, una llamada a *read* bloquea el programa hasta que cierta entrada esté presente en el *file descriptor* sobre el que se realizó la operación. Cualquier operación en otro *file descriptor* no desbloqueará a esa llamada. Podría intentar entonces setear el modo no bloqueante y hacer polling sobre los distintos *file descriptors* pero esto es totalmente ineficiente.

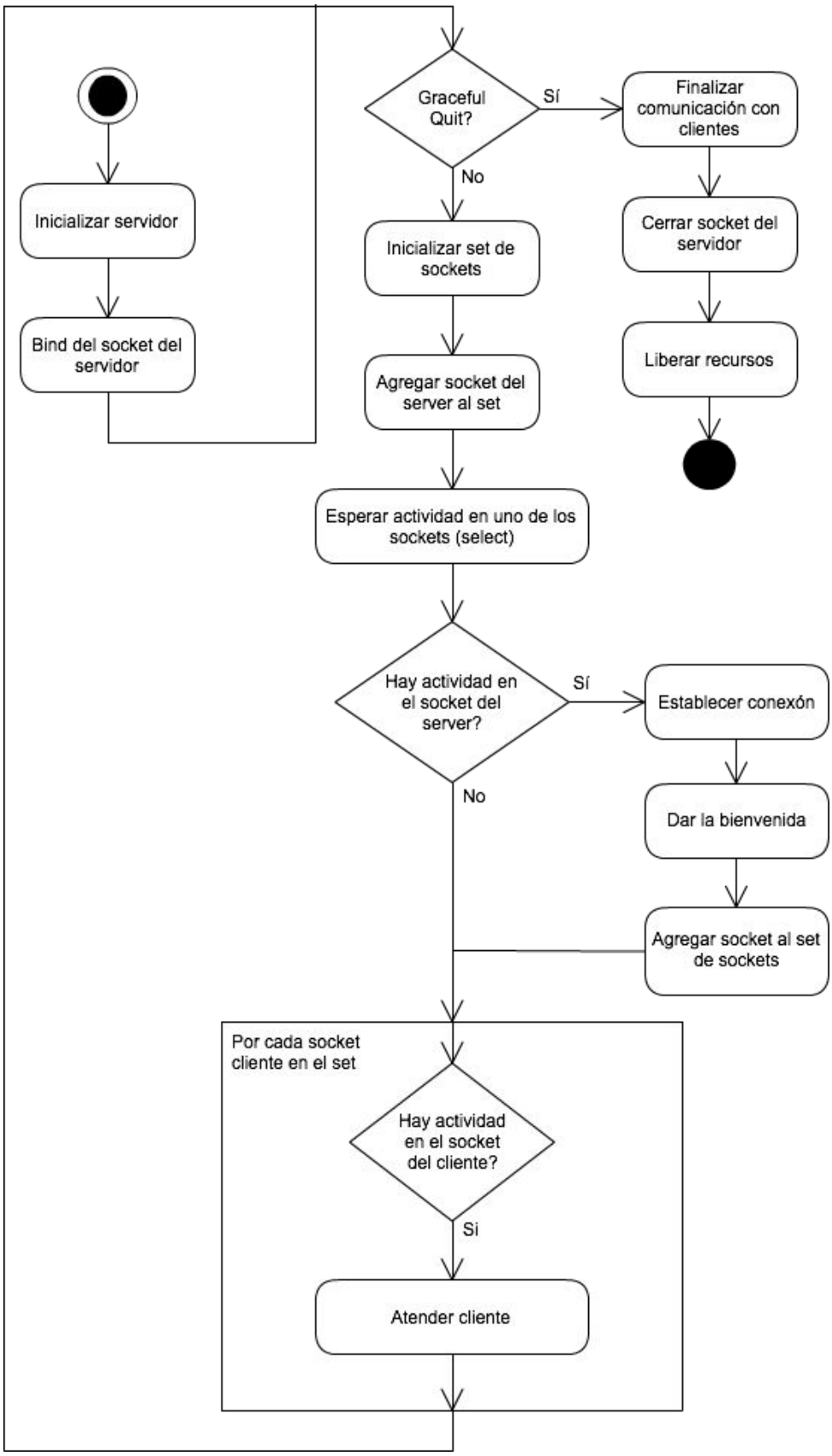
Una mejor solución es usar la función *select*. Esta llamada bloquea la ejecución hasta que una operación de I/O esta lista en algúno de varios *file descriptors*, o hasta que un timeout expire (lo que suceda primero).

Lo que hacemos es utilizar *select* en conjunto con *accept* (acepta una conexión de un socket cliente). La estrategia es armar un pool de *file descriptors* de sockets que vamos a monitorear. Ese pool está compuesto en primer lugar con el *file descriptor* del socket sobre el cual es server. Entonces, cualquier actividad de I/O sobre ese socket (que serán las conexión entrantes) será recibida asincrónicamente. Además, a ese pool le iremos agregando los *file descriptors* de todos los sockets (clientes) que tienen una conexión establecida con el server, para poder recibir información de ellos.

Para ejecutar el servidor se creo un script denominado ***runServer.sh*** el cual levante el server y lo deja listo para atender clientes:

```
$ ./runServer.sh
```

Diagrama del funcionamiento del servidor



Implementación del cliente

El cliente fue implementado de forma similar al servidor, pero haciendo uso de un único socket para comunicarse con el servidor. Si hay actividad en el socket, el cliente atiende la comunicación y envía a la salida estándar el resultado de la misma.

El cliente a su vez puede ser reemplazado por el uso del protocolo de red telnet ya que la comunicación con el servidor responde ante las consultas de igual manera:

```
$ telnet <host> <port>
```

```
$ telnet localhost 5000
```

Los mensajes que envía el cliente deben finalizar con ';' ya que el formato es SQL. Como se puede ver en el apartado siguiente de Implementación de la DB.

Para ejecutar el cliente se creo un script denominado ***runClient.sh*** el cual levante el cliente e incia la comunicación.

```
$ ./runClient.sh
```

Hay dos formas de **salir** del cliente:

1. Enviando el mensaje ***exit();*** al servidor.
2. Utilizando la señal SIGINT a través de las teclas CTRL + C.

Implementación de la DB

Lenguaje de consulta

Cada sentencia debe terminar con ; (punto y coma).
Las sentencias no son case-sensitive

Consulta

Las consultas se realizan sobre la tabla *Person* utilizando la siguiente sintaxis:

```
select * from Person;
```

Inserción

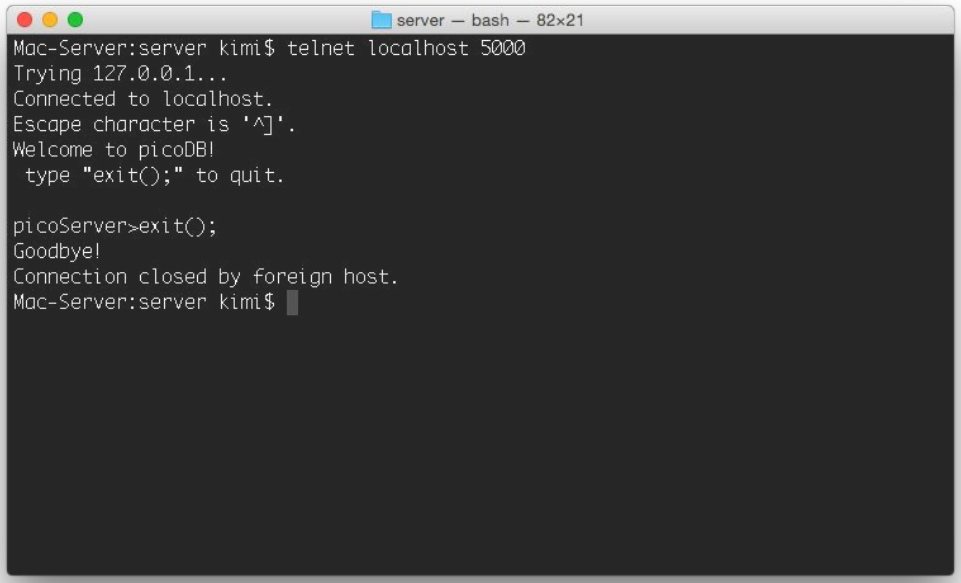
Las inserciones se realizan sobre la tabla *Person* utilizando la siguiente sintaxis:

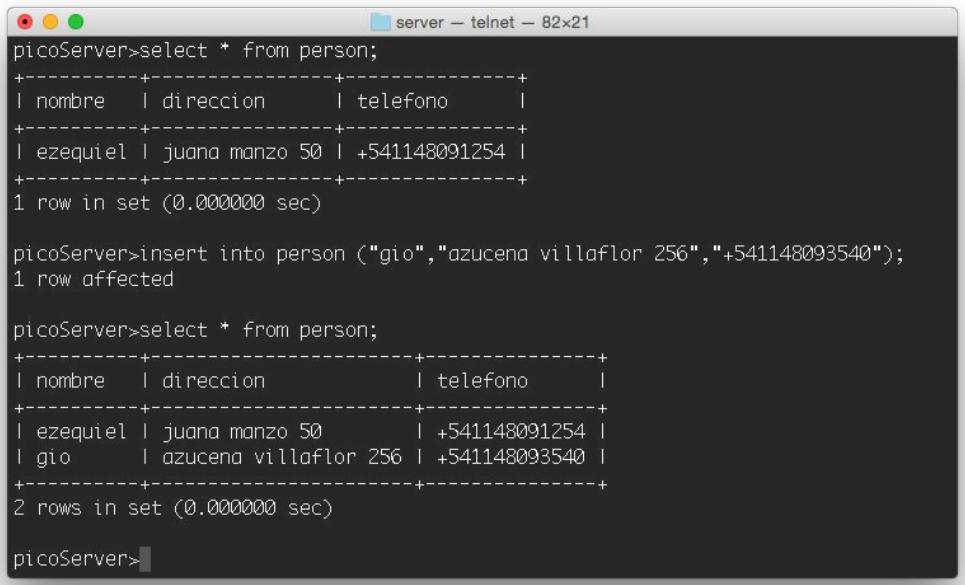
```
insert into Person (<nombre>,<dirección>,<teléfono>);
```

Pruebas

Servidor

Se realizaron varias pruebas sobre el servidor utilizando el programa Telnet.





Compilación y ejecución

Compilación del proyecto

Desde el directorio del proyecto ejecutar “./build.sh”. Este script invoca a la compilación usando cmake. El orden de compilación es:

1. shared library
2. client app
3. server app

Ejecución del servidor

Para iniciar una instancia de servidor ejecutar “./runServer.sh”. Se iniciará el servidor en el puerto 5000 (default).

Para ejecutar el servidor en otro puerto es necesario indicar el argumento port de alguna de las dos maneras que se muestran a continuación

“./picoServer -p 5001”

“./picoServer --port 5001”

Ejecución del cliente

Para iniciar una instancia de cliente ejecutar “./runClient.sh”. Se iniciará la conexión con el servidor ubicado en 127.0.0.1 en el puerto 5000 (default).

Para conectar el cliente en otro puerto y/o host, ejecutar de la siguiente manera

“./picoClient -t 192.168.1.43 -p 5001”

“./picoClient --host 192.168.1.43 --port 5001”

“./picoClient -p 5001” (localhost:5001)

“./picoClient --port 5001” (localhost:5001)

Referencias bibliográficas

1. http://www.gnu.org/software/libc/manual/html_node/Waiting-for-I_002fO.html
2. http://www.terabit.com.au/docs/Comparison.htm
3. http://www.artima.com/articles/io_design_patternsP.html