

Ejercicios Optativos 2

Cristina Díaz García

Diciembre 2018



Índice

Índice general	1
1. I. Desarrollo de servicios no estándar	2
1.1. Ejercicio 2	2
1.2. Ejercicio 3	3
2. III. Patrones de diseño	4
2.1. Ejercicio 1	4
2.2. Ejercicio 2	4

1. I. Desarrollo de servicios no estándar

1.1. Ejercicio 2

2. Implementar el mismo servicio de “eco” que en el ejercicio anterior pero esta vez sobre TCP. Deberán tenerse en cuenta los siguientes requisitos de implementación:

- Usar la interfaz *Executor* para implementar el servidor concurrente.
- Opcionalmente implementar la otra opción de grupo de hebras y comparar ambas implementaciones
- El cliente le notificará al servidor el final del servicio mediante el cierre del *socket*.

En este ejercicio se han implementado tres clases:

- **TCPCliente:** Es el cliente, al que se le pasan como argumentos al ejecutarlo, primero la dirección IP a la que conectarse, y segundo, el puerto. Si el servidor no estuviera disponible, el cliente se cierra automáticamente. En caso de que se realizara la conexión, envía y recibe los mensajes hasta que se envía un punto, en cuyo caso, el servidor cierra la conexión y el cliente se cierra.

```
java Client
Local connection: /127.0.0.1
Connected to 127.0.0.1:12345
This is an echo server. Type whatever you want me to echo.
Hola
Waiting for a response...
echo: Hola
.
Waiting for a response...
The connection has been closed.
```

- **TCPServer:** Es el servidor, al que se le pasa como argumento el puerto por el que escuchar. Cada vez que un cliente se intenta conectar, se crea una instancia de EchoProtocol, que es el que “gestiona” la conexión, es decir, es el que hace la función de eco.

```
java Server
Server initialization...ServerSocket[addr=0.0.0.0/0.0.0.0,localport=12345]
Waiting for a new client...
New client, socket Socket[addr=/127.0.0.1,port=48644,localport=12345]
New client, socket Socket[addr=/127.0.0.1,port=48646,localport=12345]
New client, socket Socket[addr=/127.0.0.1,port=48652,localport=12345]
Read timed out
Read timed out
Read timed out
New client, socket Socket[addr=/127.0.0.1,port=49632,localport=12345]
Waiting for a new client...
```

- **EchoProtocol:** Recibe los mensajes del cliente y se los reenvía, siempre que el mensaje no fuera un punto, en cuyo caso cerraría la conexión.

Las ventajas de la implementación con TPC es la fiabilidad de la conexión. En esta implementación, uno de los problemas que tenemos es que estamos asumiendo que los argumentos se van a introducir tanto en el orden correcto como siendo los necesarios, es decir, no vamos a meter una IP incorrecta, ni vamos a introducir una palabra en vez de la IP correspondiente.

1.2. Ejercicio 3

3. A partir del servicio de “eco” dado en clase realizar las siguientes modificaciones para probar las diferentes opciones de sockets sobre TCP:

- En el servidor, limitar el tiempo de espera de una nueva conexión dando por finalizado el servidor tras dicho tiempo (ej: varios minutos). Nota: probar a arrancar el servidor justo después de su finalización, antes de implementar la característica que se propone a continuación, ¿qué ocurre?

```
sockfd.setSoTimeout(60000);
```

Cuando se intenta volver a ponerlo en el mismo puerto justo después de quitarlo da un fallo al hacer binding en el puerto. Una vez que se activa, se puede volver a usar ese puerto.

- En el servidor establecer la opción de reutilización del puerto para facilitar el arranque del servidor tras su finalización por el temporizador.

```
sockfd.setReuseAddress(true);
```

- En el servidor, limitar el tiempo total del servicio de “eco” (ej: un minuto)

```
socket.setSoTimeout(60000);
```

- Cambiar en el cliente el tamaño del buffer de escritura a 512 bytes y comprobar si se ha cambiado visualizando el valor anterior y el nuevo.

```
Tamaño actual del buffer de escritura: 1313280
Cambiando tamaño a 512...
Tamaño actual del buffer de escritura: 2304
```

El tamaño mínimo es 2304, por lo que no se ha podido poner a 512.





- Deshabilitar el algoritmo de Nagle, que es especialmente recomendado si se va a realizar el eco a nivel de caracteres o mensajes muy pequeños.

```
sockfd.setTcpNoDelay(false);
```

2. III. Patrones de diseño

2.1. Ejercicio 1








Implementar un multi-servidor que proporcione al menos dos servicios sencillos, ambos sobre TCP. Nota: pueden ser dos versiones diferentes del servicio de “eco”.

```
▶  EchoProtocol.java
▶  EchoProtocolR.java
▶  TCPProtocol.java
▶  TCPServerSelector.java
```

Los dos servicios implementados, ambos implementados sobre TCP, son un servicio de eco otro de eco reverse, que devuelve invertido el mensaje invertido. Un problema a tener en cuenta es el tener que guardar como mínimo una de las dos Keys del ServerSocketChannel para poder diferenciar los servicios para poder responder a la hora de realizar el accept, ya que al aceptarlo se crea otro SocketChannel hay que guardar su clave para poder identificar cuál es el servicio que debe hacer read o write.

2.2. Ejercicio 2

Implementar un servidor multi-protocolo que proporcione el servicio de “eco” sobre TCP y sobre UDP en el mismo puerto usando el paquete NIO (Selector). El servicio de “eco” es el mismo que el propuesto en los ejercicios I.1 y I.2. Se valorará el grado de modularización de la solución presentada.

```
▶  ClienteTCP.java
▶  ClienteUDP.java
▶  ClientRecord.java
▶  EchoProtocol.java
▶  EchoProtocolUDP.java
▶  ServerSelector.java
▶  TCPProtocol.java
```

Los clientes usados son los mismos que en los ejercicios del primer apartado, ya descritos previamente.

El servidor tiene tanto ServerSocketChannel como DatagramChannel. Le asigna al DatagramChannel un objeto de tipo ClientRecord y se registran en el selector, guardando la clave del DatagramChannel. Se crean las dos clases, tanto el echo sobre TCP como el echo sobre UDP, y con un Iterator se recorren las claves. Accept solo se usa con las claves de TCP, así que no habría que compararlas pero en Read y Write sí hay que filtrar por clave, al guardar la clave UDP se compara esta con la obtenida en el iterator y se realizan las operaciones de TCP o las de UDP.

El echo sobre TCP crea un SocketChannel asignado al Accept de la clave y se registra un nuevo canal de lectura, en Read se consigue el canal de lectura de la clave y se lee en éste. Si al leer la longitud es -1 es que no hay nada y se ha cerrado la conexión. En Write se vuelve a obtener el canal de la clave y se manda el mensaje que se había leído en el buffer al cliente y por lo tanto una vez acabada la escritura vuelve a Read.

El echo sobre UDP obtiene el canal de la clave al ejecutar el Read, y el ClientRecord asignado a esta al hacer Receive con el buffer guarda la dirección del cliente. Si esta no es nula, se pasa a escritura, en Write se obtiene el DatagramChannel de la clave, y el ClientRecord. Para enviar se manda el contenido del buffer a la dirección almacenados en el ClientRecord, y se devuelve a leer.