Comunicación mediante Sockets UDP en Java

Redes de Comunicaciones - Curso 2023/24

Introducción

El objetivo de esta sesión es comenzar con la programación de comunicaciones basadas en UDP. En el caso de la práctica de NanoFiles, dichas comunicaciones se producen con el servidor de directorio. En particular, los pares enviarán al directorio varios tipos de solicitudes, por ejemplo, para iniciar sesión (*login*) usando un *nickname*, consultar los ficheros disponibles, registrarse como servidor de ficheros, publicar ficheros, etc. La mayor parte del contenido de este boletín estará relacionado con el código necesario para enviar mensajes a través de UDP en Java.

Para la realización de la práctica, se os proporciona código fuente que formará parte del proyecto Java en Eclipse llamado *NanoFiles*. Dicho proyecto lo iremos analizando a lo largo de este y posteriores boletines conforme vayamos avanzando en la práctica.

Programación con sockets UDP

En Java se programan los sockets UDP mediante las clases `DatagramPacket` y `DatagramSocket`. Tanto los clientes como servidores usan ambas clases para enviar segmentos UDP.

`DatagramPacket` y `DatagramSocket`

En UDP se intercambian mensajes llamados datagramas, los cuales se representan en Java mediante instancias de `DatagramPacket`.

- Para el envío, hay que construir un `DatagramPacket` que contenga los datos a ser enviados y pasárselo como argumento al método `send()` de `DatagramSocket`.
- Para recibir, hay que construir una instancia de `DatagramPacket` que tenga un espacio de memoria preasignado (un `byte[]`) en el cual se copiarán los contenidos del mensaje recibido (cuando llegue, si es que llega). Esta instancia hay que pasársela al método `receive()` de `DatagramSocket`.

Además de los datos, cada instancia de `DatagramPacket` también contiene información sobre la dirección y el puerto, cuyo significado realmente depende de si el datagrama se está enviando o recibiendo. Cuando se envía entonces la dirección y el puerto se refieren al destinatario; cuando se recibe entonces se refieren al origen del mensaje. Por tanto, un servidor podría recibir una instancia de `DatagramPacket`, modificar el contenido de los datos y después enviar la misma instancia mediante el método `send()` de `DatagramSocket`. Ello daría lugar a que el mensaje modificado volviera de nuevo a su origen.

`DatagramSocket` es la clase que permite llevar a cabo el envío de datos a través de UDP. Puesto que este protocolo no es orientado a conexión, no es necesario establecer una conexión y cada datagrama podría ser enviado a distintas fuentes o podría ser recibido desde distintos orígenes.

Una consecuencia de usar UDP es que los datagramas pueden perderse. En tal caso podría suceder que un receptor se quedara bloqueado en un método `receive()` debido a que nunca llegan a recibirse los datos.

Para evitar este problema se puede usar el método `setSoTimeout()` de `DatagramSocket` para especificar la máxima cantidad de milisegundos que el método `receive()` se va a quedar bloqueado.

Ejemplo de servidor UDP

Ilustramos ahora, mediante un ejemplo de código, la utilización de sockets UDP a través de la clase `DatagramSocket` para implementar el envío y recepción de datagramas UDP. Como podemos ver a continuación, el servidor simplemente crea un socket UDP en el puerto 6868 y se bloquea hasta recibir un datagrama en dicho puerto. A su recepción, el servidor envía un datagrama en respuesta al emisor, para ello debe obtener la dirección y puerto origen del datagrama recibido previamente, y finalmente termina la comunicación. En este caso se envían un array de bytes sin contenido concreto.

```
  public final int PORT = 6868;
  public final int MAX_MSG_SIZE_BYTES = 1024;

  public void UDPServer() {
   // Create datagram socket
   socket = new DatagramSocket(PORT);

   // Allocate buffer to receive message
   byte[] buf = new byte [MAX_MSG_SIZE_BYTES];

   // Create datagram packet
   DatagramPacket pckt = new DatagramPacket(buf, buf.length);
   // Receive request message
   socket.receive(pckt);

   // Do something with message received in “buf”

   byte[] resp = ... // Create response message

   // Send response back to client at address
   InetSocketAddress clientAddr = (InetSocketAddress)
 pckt.getSocketAddress();
   pckt = new DatagramPacket(resp, resp.length, clientAddr);
   socket.send(pckt);

   // Closes this datagram socket
   socket.close();
  }
```

Ejemplo de cliente UDP

El código del cliente es muy similar al del servidor, salvo en la forma en que se crea el socket UDP y la dirección de socket destino del datagrama. En este caso, al crear el socket no es necesario proporcionar el puerto origen, sino que se elegirá cualquier puerto origen UDP local que esté disponible, mientras que para crear el datagrama habrá que especificar tanto la dirección (o el nombre) del servidor como el puerto UDP destino donde éste escucha.

```
private final int PORT = 6868;
public final int MAX_MSG_SIZE_BYTES = 1024;

public void UDPClient(String serverName) throws IOException {

 // get a datagram socket
 DatagramSocket socket = new DatagramSocket();

 // allocate buffer and prepare message to be sent
 byte[] req = new byte [MAX_MSG_SIZE_BYTES];

 // send request
 InetSocketAddress addr =
  new InetSocketAddress(InetAddress.getByName(serverName), PORT);
 DatagramPacket packet = new DatagramPacket(req, req.length, addr);
 socket.send(packet);

 // receive response
 byte[] buf = new byte [MAX_MSG_SIZE_BYTES];
 response = new DatagramPacket(buf, buf.length);
 socket.setSoTimeout(1000);
 socket.receive(response);

 // Do something with response in “response”

 socket.close();
}
```

El código anterior debe tomarse como base, no literal, sino adaptado, para empezar a intercambiar información con el servidor de directorio.

Ejecución y depuración de aplicaciones Java cliente-servidor en Eclipse

En esta sección veremos un código de ejemplo que implementa un servidor y un cliente que se comunican mediante *sockets* UDP. Utilizando dicho código, veremos cómo podemos ejecutar varios programas simultáneamente dentro del IDE de Eclipse, así como tracear (depurar) su ejecución paso a paso. De esta forma podremos observar su comportamiento en conjunto, ya que ambos son dependientes entres sí ya que las acciones que cada uno de ellos realiza depende de los valores que otros programas le comunican a través de la red. Gracias al depurador, podremos observar el valor de las variables en cada instante tanto en el programa cliente como en el servidor, y así detectar errores en nuestro código.

> **Aprender el uso de las herramientas de depuración de Eclipse es esencial para afrontar el proyecto de programación de *NanoFiles* con garantías de éxito**.

Ilustraremos la depuración de programas Java en Eclipse usando el proyecto `boletinUDP` cuyo código esta disponible en el Aula Virtual como anexo a este boletín de prácticas (fichero `boletinUDP.zip`). Sigue los siguientes pasos:

1. Abre Eclipse e importa el proyecto `boletinUDP`, seleccionando en el menú `File` → `Import` → `Existing projects into workspace` y eligiendo la carpeta donde has descomprimido el fichero descargado del AV.

2. Lanza el programa servidor y a continuación el programa cliente. Para ello, en el panel explorador de paquetes (`Package Explorer`) puedes simplemente hacer *clic* en el botón derecho sobre el fichero .java correspondiente y seleccionar `Run as..Java Application`.

3. Observa la salida de ambos programas; necesitas cambiar entre la consola de uno y otro mediante el botón `Display Selected Console` que aparece en la barra superior de la ventana de la consola (por defecto, el panel inferior de la ventana).

    1. Puedes eliminar las consolas correspondientes a programas que ya han terminado mediante el botón `Remove all terminated launches`.
    2. Puedes terminar la ejecución del programa en cualquier momento mediante el botón `Terminate`.

4. Establece un punto de ruptura (*breakpoint*) en la primera línea del código de cada uno de los dos programas, cliente y servidor.

    1. Para poner un *breakpoint* en una línea de código, basta con hacer doble clic en el margen izquierdo del panel donde se muestra el código (aparecerá un pequeño círculo azul).
    2. Para eliminar un *breakpoint*, sigue el mismo procedimiento. A veces es útil desactivarlos en lugar de eliminarlos (botón derecho sobre el margen izquierdo, `Toggle/Disable breakpoint`).

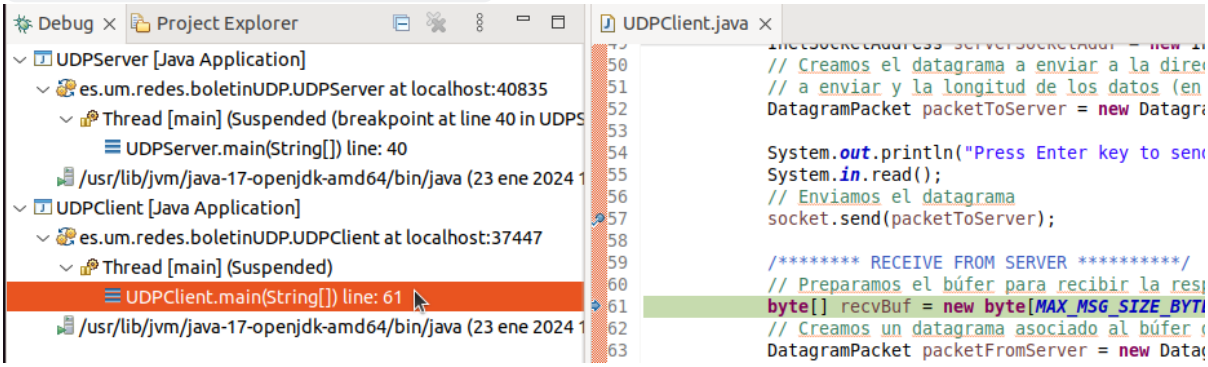
5. Ejecuta ahora el servidor en modo depuración, haciendo *clic* en el botón derecho sobre el fichero .java y eligiendo `Debug as..Java Application`).

    1. Si Eclipse te pregunta, confirma que quieres pasar a la perspectiva de depuración (puedes decirle que recuerde tu decisión).
    2. En el panel izquierdo de esta perspectiva de depuración verás que tienes junto a la pestaña `Debug` otra de `Project Explorer`. Puedes usarla para ver las clases de tu programa, abrir su

código y ejecutar/lanzar en modo depuración sin cambiar de perspectiva.

6. Ejecuta línea a línea el código del servidor, mediante el botón `Step over` (o la tecla F6), hasta llegar a la línea donde se ejecuta `socket.receive(..)` (sin llegar a ejecutarla).

    1. Pincha en `Project Explorer` y luego depura ahora el programa cliente, tal y como has hecho con el servidor: pincha en `UDPClient` y luego `Debug as.. Java Application`.
    2. Vuelve a la pestaña `Debug` del panel izquierdo; verás que ahora tienes no una sino dos `Java Applications` en ejecución al mismo tiempo: `UDPServer` y `UDPClient`, ambas detenidas en sendos *breakpoints*. Puedes alternar la depuración paso a paso entre diferentes aplicaciones Java, o incluso entre diferentes hilos de una misma aplicación, con tan solo pinchar en el desplegable correspondiente al programa/hilo/método cuya ejecución está detenida (p.ej. `UDPClient.main(String[]) line: 61`), como ilustra la siguiente figura:



7. En panel de la consola, abre una nueva consola (botón `Open Console..New Console View`). Arrastrando con el ratón desde la pestaña de la consola, mueve la nueva ventana para obtener una disposición dividida, de forma que aparezcan simultáneamente las dos consolas (servidor y cliente).

    1. Prueba a fijar cada consola a un programa distinto usando el botón `Pin console`, de forma que puedas observar en diferentes ventanas la salida de cada programa.

8. Ejecuta línea a línea el código del cliente, mediante el botón `Step over` (o la tecla F6), hasta llegar a la línea donde se ejecuta `socket.send(..)` (sin llegar a ejecutarla). Verás que ambas consolas muestran la misma salida (la del programa cliente). Esto se debe a que por defecto en Eclipse cada vez que se imprime un mensaje por consola se cambian automáticamente todas las vistas a dicho programa.

9. Ejecuta la siguiente línea del programa servidor (`socket.receive`), y fíjate en lo que ocurre.

10. Ejecuta la siguiente línea del programa cliente (`socket.send`), y acto seguido, fíjate en lo que ha ocurrido en el programa servidor.

11. Continúa ejecutando el programa servidor paso a paso (con F6) para observar cómo envía un datagrama de respuesta al cliente y finalmente termina.

12. Regresa a la depuración del programa cliente para ejecutar paso a paso (F6) hasta el final. Fíjate que en este caso la operación `socket.receive` se comporta de manera distinta, ya que no ha de esperar a que llegue el datagrama de respuesta (ya fue enviado por el servidor).

13. Repite la depuración de ambos programas como en los pasos anteriores, pero ahora avanza la ejecución del cliente hasta ejecutar `socket.receive` antes de que el servidor haya llegado a

ejecutar `socket.send`. Verás que en el cliente, transcurridos 2 segundos desde que ejecutó `socket.receive`, se recibe una excepción `SocketTimeoutException`, ya que se estableció previamente un temporizador (*timeout*) para evitar que el cliente pueda quedar bloqueado indefinidamente esperando la recepción de un datagrama a través del socket UDP creado.

Ejercicios a realizar durante la sesión

A partir del código de partida proporcionado del proyecto *NanoFiles*, modifica los archivos `NFDirectoryServer.java` y `DirectoryConnector.java` para implementar la funcionalidad que se pide a continuación.

NOTA : Con el fin de facilitar la tarea del alumnado, el código de *nanoFiles* contiene comentarios anotados con la cadena `TODO`, con instrucciones sobre la funcionalidad que falta por implementar.

1. Basándote en el código de `UDPClient`, añade el código necesario en la clase `DirectoryConnector.java` (paquete `es.um.redes.nanoFiles.udp.client`) para poder enviar mensajes a un servidor mediante UDP. Para ello, siguiendo las instrucciones de los `TODO`'s que hay en el código, has de terminar la implementación de estos métodos:

    1. Constructor de `DirectoryConnector`: recibe como parámetro una cadena con el nombre o IP de la máquina donde se ejecuta el programa *Directory* (servidor UDP en *NanoFiles*). El programa *NanoFiles* se comunicará siempre con el mismo *Directory* (parámetro del constructor) usando un objeto de esta clase.
    2. Método `sendAndReceiveDatagrams`: recibe como parámetro un array de bytes que será enviado en un datagrama, y devuelve un array de bytes con la respuesta recibida por parte del directorio (servidor UDP). **NOTA**: Por ahora, céntrate en la implementación de la funcionalidad básica y asume que no se pierden datagramas; dejaremos para más adelante la implementación del mecanismo de retransmisión.

2. Implementa el método `testSendAndReceive` de la clase `DirectoryConnector.java`, haciendo uso del método `sendAndReceiveDatagrams` que has implementado en el punto anterior para enviar la cadena "login" al directorio y comprobar que la respuesta recibida es la cadena "loginok". **NOTA**: En el código del proyecto `boletinUDP` visto anteriormente, puedes consultar la forma de convertir un objeto `String` a array de bytes, y viceversa.

3. Comprueba si el código del cliente UDP que has programado hasta ahora es correcto utilizando el depurador de Java. Como todavía no tenemos un servidor UDP funcional en *NanoFiles* que nos pueda responder (la clase `NFDirectoryServer.java` del paquete `es.um.redes.nanoFiles.udp.server` está sólo parcialmente esbozada), enviaremos datagramas al servidor implementado por `UDPServer` visto anteriormente en este boletín.

    1. Ejecuta el programa `UDPServer` del proyecto `boletinUDP` para dejar un servidor UDP escuchando en el puerto 6868 de la máquina.

    2. Establece sendos puntos de ruptura en `DirectoryConnector.java`, al inicio de cada uno de los métodos que acabas de programar: constructor y `testCommunicationWithDirectory`.

    3. Asegúrate de que el atributo de clase `testMode` de la clase `NanoFiles.java` (paquete `es.um.redes.application`) está establecido a verdadero (debería estarlo por defecto).

    4. En la misma máquina donde has ejecutado `UDPServer`, ejecuta el programa `NanoFiles` y teclea el comando `login localhost alumno`. La ejecución debería detenerse en el *breakpoint* que has puesto en `testCommunicationWithDirectory`. **NOTA**: Puesto que tanto cliente como

servidor están en la misma máquina, ambos pueden comunicarse a través interfaz de red virtual *loopback*. Por tanto, pasamos `localhost` como argumento a `login` indicando al cliente *NanoFiles* que el servidor de directorio se encuentra escuchando en la propia máquina local.

5. Ejecuta paso a paso (tecla F6) el código que has programado, colocando el puntero sobre las diferentes variables, atributos, etc. para ver su valor en cada instante. Si tu código es correcto, correcto, el servidor UDP debería recibir tu mensaje con la cadena "login" y devolverte el mismo mensaje al que se le ha añadido la hora actual, lo cual no cumple el valor esperado ("loginok").

4. Ahora, basándote en el código de `UDPServer`, añade el código necesario en la clase `NFDirectoryServer.java` (paquete `es.um.redes.nanoFiles.udp.server`) para poder recibir mensajes a través de UDP. Para ello, siguiendo las instrucciones de los `TODO`'s que hay en el código, has de terminar la implementación de estos métodos:

    1. Constructor de `NFDirectoryServer`: Debe inicializar todos los atributos de instancia de la clase, incluyendo el socket UDP y las estructuras de datos donde se guardarán los datos de los clientes (*nicknames*, etc.).
    2. Método `run`: Es el método que se ejecuta cuando se lanza el programa *Directory*, encargado de recibir datagramas de los clientes y enviar sus correspondientes respuestas.

5. Comprueba si el código del servidor UDP que has programado hasta ahora es correcto utilizando el depurador de Java.

    1. Termina la ejecución del servidor `UDPServer.java`, si es que todavía está en ejecución. De no hacerlo, el puerto 6868 seguirá estando ocupado y por tanto el programa *Directory* no podrá escuchar en dicho puerto.

    2. Establece un *breakpoint* en `NFDirectoryServer.java` al comienzo del método `run`. Depura el programa *Directory*, cuyo método `main` está en el fichero `Directory.java` del paquete `es.um.redes.nanoFiles.application`. Recuerda: botón derecho sobre el fichero `Directory.java` y luego `Debug as...Java Application`). Ejecuta paso a paso hasta comprobar que el servidor se queda bloqueado indefinidamente a la espera de recibir datagramas.

    3. Depura el programa *NanoFiles* e introduce el comando `login localhost alumno` hasta que la ejecución alcance de nuevo el método `testCommunicationWithDirectory`.

    4. En este punto, deberías tener los dos programas en ejecución controlada mediante el depurador. Asegúrate de utilizar dos consolas independientes, una para cada programa, y que ambas sean visibles al mismo tiempo (usa `New console view` y el botón `Pin console`).

    5. Ejecuta paso a paso tanto *NanoFiles* como *Directory*. Alterna la ejecución controlada en un programa y en el otro mediante el panel `Debug` (izquierda) de la perspectiva de depuración, de manera similar a como se describió anteriormente con los programas `UDPServer` y `UDPClient`. Repite tantas veces sea necesario hasta comprobar que tu código es correcto y el servidor recibe el datagrama enviado por el cliente y lo muestra por pantalla.

6. Termina de implementar la funcionalidad que falta en servidor UDP (`NFDirectoryServer.java`) para responder al cliente con un datagrama que contenga la cadena de caracteres "loginok".

7. Termina de implementar la funcionalidad que falta en el cliente UDP (`DirectoryConnector.java`), comprobando en `testCommunicationWithDirectory` que la cadena de caracteres recibida del servidor es "loginok".

Ejercicios propuestos para trabajo autónomo

1. Tras verificar que la comunicación entre cliente y servidor es correcta, prueba a arrancar el programa *Directory* pasándole como parámetro `-loss 0.3` (`Run Configurations..New Java Application..Arguments`). Arranca el programa `UDPClient` para enviar mensajes al *Directory* y comprobar que en ocasiones no response.

2. Con esta configuración que simula un enlace no confiable mediante la pérdida de segmentos UDP, amplía la funcionalidad del método `sendAndReceiveDatagrams` para implementar un mecanismo de reenvío. Para ello, debes utilizar `socket.setSoTimeout(TIMEOUT)` y capturar la excepción `SocketTimeoutException` que lanzará `socket.receive` en caso de no recibir una respuesta en un intervalo de tiempo razonable (p.ej. 1 segundo). En tal caso, se debe proceder a reenviar el mismo mensaje de nuevo y de nuevo volver esperar la respuesta.

3. Modifica el código del apartado anterior para que no se reintente el reenvío indefinidamente sino que tras un cierto número de reintentos, el programa muestre un error y aborte.