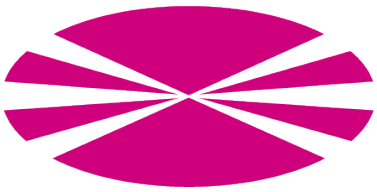

UNIVERSIDADE DA CORUÑA

LABORATORIO DE REDES.
TUTORIAL DE SOCKETS EN JAVA

Índice

1. Presentación....3
2. Conceptos básicos....4
   2.1. Entorno de Desarrollo Integrado (IDE)....4
   2.2. Flujos de datos....4
   2.3. Threads....4
   2.4. Ejercicio 1: Implementación de Copy....5
   2.5. Ejercicio 2: Implementación de Info....5
3. Pila de protocolos TCP/IP....5
   3.1. TCP....6
   3.2. UDP....6
4. Sockets....6
5. Sockets UDP....7
   5.1. Ejercicio 3: Implementación del servidor de eco UDP....8
6. Sockets TCP....9
   6.1. Ejercicio 4: Implementación del servidor de eco TCP....10
7. Evaluación....12

1. Presentación

El laboratorio de Redes constará de un conjunto de prácticas **no obligatorias**. Todas las prácticas han de ser realizadas **individualmente**.

Cada práctica constará de un enunciado en donde se plantearán las tareas a realizar, y podrá incluir información complementaria para el desarrollo de la misma. Las prácticas que requieran el empleo de un lenguaje de programación deberán ser realizadas en **Java.** Para cualquier duda sobre este lenguaje, se recomienda consultar la documentación del API de Java. La URL para acceder a la documentación de la última versión es https://docs.oracle.com/en/java/javase/23/docs/api/.

Los ficheros con el código de las prácticas deberán ser subidos al repositorio correspondiente a cada alumno a través de una URL con la siguiente estructura: https://github.com/GEI-Red-614G010172425/java-labs-<team-name>. Este repositorio se crea de manera automática tras seguir los siguiente pasos:

1. Abrir el navegador web, ir a https://github.com/ y autenticarse. Se debe **verificar el email de la UDC en la cuenta de Github** (https://github.com/settings/emails) en caso de que no esté ya verificado.
2. Hacer clic en el enlace con el *assignment* https://classroom.github.com/a/aB0GobM0.
3. Crear un nuevo **equipo** (*team*) cuyo nombre (*team-name*) tenga la siguiente estructura: **<tu grupo de prácticas>-<tu UDC login>** (por ejemplo, 1.3-john.doe).
4. Aceptar el java-labs *assignment*.
5. Refrescar el navegador tras unos segundos.
6. Buscar un nuevo email de Github donde se os invita a uniros a la organización GEI-Red-6146010172425.
7. Hacer clic en el enlace que figura en el email para uniros a la organización.
8. Autenticarse con el proveedor *single sign-on* (SSO) de la UDC.
9. Unirse a la organización.
10. Hacer clic en el enlace al repositorio. NOTA: Github cambiará automáticamente los "." del *team-name* por "-" en el nombre del repositorio (por ejemplo, java-labs-1-3-john-doe).

Este tutorial proporciona una introducción a los sockets en Java y representa la primera práctica de programación de la asignatura (p0 a partir de ahora). Para la realización de este tutorial se empleará como base el repositorio indicado. En concreto, se modificarán clases disponibles en el paquete **es.udc.redes.tutorial**. Además, el proyecto incluye también un fichero **README.md** con los distintos pasos para la configuración del entorno. **Este tutorial será evaluado en el primer examen de prácticas.**

Las prácticas se ejecutarán en el IDE IntelliJ IDEA.

2. Conceptos básicos

Para la realización de este tutorial es necesario recordar ciertos conceptos básicos.

2.1. Entorno de Desarrollo Integrado (IDE)

Se deberá utilizar un entorno de desarrollo integrado (IDE). En particular, sugerimos usar IntelliJ IDEA. Para familiarizarse con este IDE se recomienda revisar la ayuda disponible en https://www.jetbrains.com/help/idea/discover-intellij-idea.html. El objetivo es aprender a crear un proyecto Java, compilar, ejecutar y depurar un programa.

Además, se utilizará **git** como herramienta de control de versiones. En concreto, **se deberán seguir las instrucciones facilitadas en el fichero README.md del repositorio**, y se podrán emplear como ayuda la hoja de referencia de git disponible en https://training.github.com/downloads/es_ES/github-git-cheat-sheet.pdf y la guía de Github accesible vía https://guides.github.com/activities/hello-world/.

Como mecanismo de seguridad, es recomendable subir a Github las diferentes versiones de la práctica según se va avanzando en el desarrollo, y no solo la versión final. Además, también es recomendable ir haciendo commits locales de forma recurrente.

2.2. Flujos de datos

Los programas pueden necesitar recoger información de una fuente externa y enviar información a un destino externo. Esta información puede estar en ficheros, disco, en la red, memoria, programas, etc., y puede ser de cualquier tipo (objetos, caracteres, imágenes, sonidos...).

El programa usará los flujos de datos para leer y enviar la información necesaria.

En la documentación de Java se incluye un tutorial completo para el manejo de los flujos de datos (http://docs.oracle.com/javase/tutorial/essential/io/index.html), del que, cómo mínimo se deben seguir los siguientes apartados para poder realizar la práctica:

- http://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/buffers.html

2.3. Threads

Normalmente los programas están formados por un único hilo de ejecución y su comportamiento es secuencial.

Cuando en un programa se quiere realizar más de una tarea simultáneamente se utilizan técnicas de ejecución concurrente, que permiten que diversos puntos del programa se estén ejecutando a la vez.

Existen dos aproximaciones para ejecutar tareas de forma concurrente en un programa: Procesos y Threads. Mientras que todos los threads dentro de un proceso comparten recursos (como la memoria), los procesos no comparten recursos entre sí (por lo que un objeto creado en un proceso no es visible desde otro proceso distinto).

Ambas aproximaciones se pueden combinar en un mismo programa, que puede tener varios procesos, y cada uno de ellos con varios threads simultáneos. Un programa concurrente ejecuta múltiples threads simultáneamente, cada uno realizando diferentes tareas.

Java es multithread, por lo que permite la realización de programas de flujo múltiple. Las aplicaciones Java básicas están formadas normalmente por un único proceso con un único thread, que invoca al método “main” de la clase que se ejecuta.

Para más información leer el apartado destinado a threads en:

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

El uso de los threads se utilizará en el desarrollo de un servidor de eco TCP.

2.4. Ejercicio 1: Implementación de Copy

Utilizando IntelliJ IDEA, modificar la clase **es.udc.redes.tutorial.copy.Copy.java** disponible en el repositorio **java-labs** para desarrollar una aplicación que permita copiar ficheros de texto y ficheros binarios utilizando flujos de datos. Para su ejecución recibirá dos parámetros de la siguiente forma:

```
java es.udc.redes.tutorial.copy.Copy <fichero origen> <fichero destino>
```

Para que la ejecución desde la línea de comandos (no recomendable) pueda realizarse correctamente se deberá llevar a cabo desde el directorio *java-labs-<team-name>/out/production/java-labs-<team-name>*.

2.5. Ejercicio 2: Implementación de Info

Utilizando IntelliJ IDEA, modificar la clase **es.udc.redes.tutorial.info.Info.java** disponible en el repositorio **java-labs** para desarrollar un método que permita obtener las principales propiedades de un fichero: tamaño, fecha de última modificación, nombre, extensión, tipo de fichero (image, text, directory, unknown), ruta absoluta. Para su ejecución recibirá como parámetro la ruta relativa de ese fichero respecto a p0-files de la siguiente forma:

```
java es.udc.redes.tutorial.info.Info <ruta relativa>
```

Para más información, se recomienda leer el siguiente apartado:

https://docs.oracle.com/javase/tutorial/essential/io/fileio.html.

3. Pila de protocolos TCP/IP

La pila de protocolos TCP/IP permite la transmisión de datos entre redes de computadores. Dicha pila consta de una serie de capas, tal y como se puede apreciar en el diagrama de la página siguiente.

Normalmente, cuando se escriben aplicaciones Java en red trabajaremos con el nivel de aplicación, y utilizaremos además protocolos del nivel de transporte. Por este motivo es preciso recordar las principales diferencias entre los dos protocolos básicos del nivel de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).

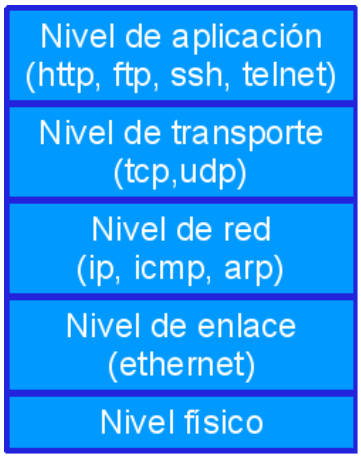

Figura 1: Pila de protocolos TCP/IP

3.1. TCP

- Es un protocolo orientado a conexión
- Provee un flujo de bytes fiable entre dos ordenadores (llegada en orden, correcta, sin perder nada). Para conseguirlo realiza control de flujo, control de congestión...
- Protocolos de nivel de aplicación que usan TCP: telnet, HTTP, FTP, SMTP...

3.2. UDP

- Es un protocolo no orientado a conexión
- Envía paquetes de datos (datagramas) independientes sin garantías
- Permite broadcast y multicast
- Protocolos de nivel de aplicación que usan UDP: DNS, TFTP…

4. Sockets

Cuando estamos trabajando en una red de ordenadores y queremos establecer una comunicación (recibir o enviar datos) entre dos procesos que se están ejecutando en dos máquinas diferentes de dicha red, ¿qué necesitamos para que dichos procesos se puedan comunicar entre sí?

Supongamos que una de las aplicaciones solicita un servicio (cliente), y la otra lo ofrece (servidor).

Una misma máquina puede tener una o varias conexiones físicas a la red y múltiples servidores pueden estar escuchando en dicha máquina. Si a través de una de las conexiones físicas se recibe una petición por parte de un cliente ¿cómo se identifica qué proceso debe atender la petición? Es aquí donde surge el concepto de **puerto**, que permite tanto a TCP como a UDP dirigir los datos a la aplicación correcta de entre todas las que se están ejecutando en la máquina. Todo servidor, por tanto, ha de estar registrado en un puerto para recibir los datos que a él se dirigen (veremos en los ejemplos cómo se hace esto).

Los datos transmitidos a través de la red tendrán, por tanto, información para identificar la máquina mediante su dirección IP (si IPv4 32 bits, y si IPv6 128 bits) y el puerto (16 bits) a los que van dirigidos.

Los puertos:

- son independientes para TCP y UDP
- se identifican por un número de 16 bits (de 0 a 65535)
- algunos de ellos están reservados (de 0 a 1023), puesto que se emplean para servicios conocidos como HTTP, FTP, etc., y no deberían ser utilizados por aplicaciones de usuario.

Por tanto, un **socket** se puede definir como un extremo de un enlace de comunicación bidireccional entre dos programas que se comunican por la red (se asocia a un número de puerto).

- Se identifica por una dirección IP de la máquina y un número de puerto.
- Existe tanto en TCP como un UDP.

Java incluye la librería `java.net` para la utilización de sockets, tanto TCP como UDP. Este tutorial se basa exclusivamente en clases y métodos de esta librería, por lo que será necesario importarla en todos los ejemplos y prácticas.

Como lectura recomendada aconsejamos: The Java Tutorials. Lesson: All About Sockets: **http://docs.oracle.com/javase/tutorial/networking/sockets/index.html**

5. Sockets UDP

Los sockets UDP son no orientados a conexión. Los clientes no se conectarán con el servidor sino que cada comunicación será independiente, sin poderse garantizar la recepción de los paquetes ni el orden de los mismos. Es en este momento en donde podemos definir el concepto de **datagrama**, que no es más que un mensaje independiente, enviado a través de una red, cuya llegada, tiempo de llegada y contenido no están garantizados.

El código fuente del cliente de eco UDP se ha implementado en la clase **es.udc.redes.tutorial.udp.client.UdpClient.java**. Esta clase recibe tres parámetros: `máquina_servidor puerto_servidor` y `mensaje`, en donde:

- `máquina_servidor` será el nombre (o la dirección IP) de la máquina en donde se está ejecutando un servidor de eco UDP.
- `puerto_servidor` será el puerto en el que está escuchando el servidor de eco UDP.
- `mensaje` será el mensaje que queremos enviar.

De manera más detallada, estos son los pasos que se ejecutan en el cliente:

1. Se crea un socket no orientado a conexión. Es importante recalcar que no es necesario especificar un número de puerto en el constructor, ya que el propio constructor se encargará de seleccionar un puerto libre (puerto efímero). Para más información ver la documentación de la clase `DatagramSocket (`en especial la sección de constructores).
2. Establecemos un tiempo de espera máximo para el socket. Si pasado ese tiempo no ha recibido nada se lanzará la excepción correspondiente.

3. Se obtiene la dirección IP de la máquina en la que se encuentra el servidor, a partir del primer argumento recibido (`máquina_servidor`). La clase `InetAddress` representa en Java el concepto de dirección IP. Esta clase dispone del método estático `getByName()` que obtiene la dirección IP a partir del `String` que recibe como parámetro. Este parámetro puede ser un nombre o una dirección IP.
4. Se obtiene el número de puerto en el que está ejecutándose el servidor, a partir del segundo argumento recibido (`puerto_servidor`).
5. Se obtiene el mensaje que queremos enviar al servidor, a partir del tercer argumento recibido (`mensaje`). En caso de querer enviar varias palabras es necesario utilizar comillas dobles (p.e. “Esto es una prueba”).
6. Preparamos el datagrama, indicando: el mensaje (como un array de bytes), el número de bytes a enviar, la dirección IP del destinatario y el puerto del destinatario. Ver la clase `DatagramPacket`.
7. Enviamos el datagrama invocando el método `send()` del socket que hemos creado inicialmente.
8. Preparamos un nuevo datagrama para recibir la respuesta del servidor. Para ello, es necesario crear previamente un array de bytes que va a almacenar la respuesta que vayamos a recibir. Al crear el nuevo datagrama, se indicará el array en donde queremos almacenar la respuesta y el número máximo de bytes que puede almacenar este array.
9. Recibimos el datagrama, utilizando el método `receive()` de nuestro socket UDP. Este método es bloqueante, es decir, el programa se quedará esperando en este método hasta recibir algo o, como hemos establecido un timeout, hasta que venza el timeout.
10. Por último, cerramos el socket.

5.1. Ejercicio 3: Implementación del servidor de eco UDP

En base al cliente de eco UDP, implementa un servidor de eco UDP modificando la clase **es.udc.redes.tutorial.udp.server.UdpServer.java**. Esta clase recibe como parámetro un único argumento, el número de puerto en el que escucha el servidor.

El servidor deberá seguir los siguientes pasos:

1 Crear un `DatagramSocket`, pero asociado a un número de puerto específico, el que el servidor recibe como parámetro.

2 Establecer un tiempo de espera máximo para el socket.

3 Crear un bucle infinito:

3.1 Preparar un datagrama para recibir mensajes de los clientes. Es recomendable crear un nuevo objeto `DatagramPacket` para cada mensaje que se vaya a recibir.

3.2 Recibir un mensaje.

3.3 Preparar el datagrama de respuesta para enviar al cliente. Recuerda que en cada datagrama recibido queda registrado la dirección IP y el número de puerto del remitente.

3.4 Enviar el mensaje de respuesta.

Una vez finalizado el servidor de eco UDP, para comprobar que funciona correctamente deberás ejecutarlo desde el IDE (pasándole como parámetro el número de puerto), el cual se quedará esperando a que los clientes le envíen peticiones. También desde el IDE, deberás ejecutar el cliente UDP (pasándole como parámetro la dirección IP o nombre del servidor (por ejemplo, localhost), el puerto en el que escucha el servidor (por ejemplo, 5000) y el mensaje (por ejemplo, “Testing my UDP server”).

El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor:

Mensajes en la consola del servidor en el IDE

```
SERVER: Received Testing my UDP server from /127.0.0.1:35286
SERVER: Sending Testing my UDP server to /127.0.0.1:35286
```

Mensajes en la consola del cliente en el IDE

```
CLIENT: Sending Testing my UDP server to
localhost/127.0.0.1:5000
CLIENT: Received Testing my UDP server from /127.0.0.1:5000
```

6. Sockets TCP

Los sockets TCP son orientados a conexión y fiables. Esto implica que antes de poder enviar y recibir datos es necesario establecer una conexión entre el cliente y el servidor. Una vez que la conexión está establecida, el protocolo TCP garantiza que los datos enviados son recibidos correctamente y debidamente ordenados en el otro extremo.

En primer lugar, se incluye en java-labs un cliente de eco TCP ya implementado. Esta clase recibe tres parámetros: `máquina_servidor puerto_servidor` y `mensaje`, en donde:

- `máquina_servidor` será el nombre (o la dirección IP) de la máquina en donde se está ejecutando el servidor de eco TCP.
- `puerto_servidor` será el puerto en el que está escuchando el servidor de eco TCP.
- `mensaje` será el mensaje que queremos enviar.

De manera más detallada, estos son los pasos que se ejecutan en el cliente:

1. Se obtiene la dirección IP de la máquina en la que se encuentra el servidor, a partir del primer argumento recibido (`máquina_servidor`).
2. Se obtiene el número de puerto en el que está ejecutándose el servidor, a partir del segundo argumento recibido (`puerto_servidor`).
3. Se obtiene el mensaje que queremos enviar al servidor, a partir del tercer argumento recibido (`mensaje`).
4. Se crea un socket orientado a conexión. No se especifica ningún puerto para el

cliente, ya que automáticamente se seleccionará un puerto libre (efímero). En el constructor se especifica la dirección IP y el puerto del servidor, estableciéndose la conexión con el servidor inmediatamente después de la creación del socket. Para más información ver la documentación de la clase `Socket`, en especial la sección de constructores.

5. Establecemos un tiempo de espera máximo para el socket. Si pasado ese tiempo no ha recibido nada se lanzará la excepción correspondiente.
6. Creamos el canal de entrada para recibir los datos del servidor. Para leer los datos del servidor se utiliza la clase `BufferedReader`, construida a partir del método `getInputStream()` del socket cliente. Esta clase dispone del método `readLine()` que permite la lectura línea a línea.
7. Creamos el canal de salida para enviar datos al servidor. Se utiliza la clase `PrintWriter`, construida a partir del método `getOutputStream()` del socket cliente. Esta clase dispone de los métodos `println()` y `print()`, equivalentes a los utilizados habitualmente para imprimir por pantalla.
8. Enviamos el mensaje al servidor invocando el método `println()` del flujo de salida.
9. Esperamos y recibimos la respuesta del servidor invocando el método `readLine()` del flujo de entrada.
10. Por último, cerramos el socket.

6.1. Ejercicio 4: Implementación del servidor de eco TCP

En base al cliente de eco TCP, implementa un servidor **multihilo (es.udc.redes.tutorial.tcp.server.TcpServer)** de eco TCP. Habitualmente, los servidores TCP son multihilo para poder procesar múltiples conexiones simultáneamente. Para facilitar su implementación se realizará una primera versión del servidor monohilo (**es.udc.redes.tutorial.tcp.server.MonoThreadTcpServer**), y comprobar su funcionamiento con el cliente de eco TCP. Una vez la versión monohilo funcione correctamente, se puede realizar la versión multihilo.

El servidor monohilo deberá seguir los siguientes pasos:

1 Crear un `ServerSocket`, asociado a un número de puerto específico, el que el servidor recibe como parámetro.

2 Establecer un tiempo de espera máximo para el socket.

3 Crear un bucle infinito:

3.1 Invocar el método `accept()` del socket servidor. Este método se queda esperando hasta recibir la petición de conexión de un cliente. En cuanto se establece la conexión con el cliente, devuelve un nuevo socket que se utilizará para la comunicación con ese cliente.

3.2 Preparar los flujos de entrada y salida, a partir del nuevo socket.

3.3 Recibir el mensaje del servidor.

3.4 Enviar el mensaje de eco de vuelta al cliente.

3.5 Cerrar los flujos y la conexión del socket creado en el método `accept()`.

Una vez finalizado el servidor monohilo de eco TCP, para comprobar que funciona correctamente deberás ejecutarlo desde el IDE (pasándole como parámetro el número de puerto), el cual se quedará esperando a que los clientes le envíen peticiones. También desde el IDE, deberás ejecutar el cliente TCP (pasándole como parámetro la dirección IP o nombre del servidor (por ejemplo, localhost), el puerto en el que escucha el servidor (por ejemplo, 5000) y el mensaje (por ejemplo, “Testing my TCP server”).

El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor:

Mensajes en la consola del servidor en el IDE

```
SERVER: Received Testing my TCP server from /127.0.0.1:36725
SERVER: Sending Testing my TCP server to /127.0.0.1:36725
```

Mensajes en la consola del cliente en el IDE

```
CLIENT: Sending Testing my TCP server to localhost/127.0.0.1:5000
CLIENT: Received Testing my TCP server from /127.0.0.1:5000
```

Para convertir el servidor monohilo en multihilo es necesario implementar una clase nueva (`ServerThread`) que extienda la clase `Thread`. Esta clase será la que se encargue de atender una conexión con un cliente. Los pasos en el servidor TCP serían los siguientes:

1. Crear un `ServerSocket`, asociado a un número de puerto específico, el que el servidor recibe como parámetro.
2. Establecer un tiempo de espera máximo para el socket.
3. Crear un bucle infinito:
   - 3.1 Invocar el método `accept()` del socket servidor. Al establecerse la conexión, devuelve un nuevo socket que se utilizará para la comunicación con ese cliente.
   - 3.2 Crear un nuevo objeto `ServerThread`, pasando como parámetro el nuevo socket de la conexión. De esta manera, la conexión es procesada con este nuevo socket en un hilo de ejecución independiente, quedando el socket servidor preparado para recibir nuevas peticiones de otros clientes.
   - 3.3 Iniciar la ejecución del hilo con el método `start()`. Importante: si se invoca el método `run()` en lugar del método `start()`, se realiza una ejecución secuencial, no multihilo.

Los pasos en la clase `ServerThread` son los siguientes (método `run()`):

1. Preparar los flujos de entrada y salida.
2. Recibir el mensaje del servidor.
3. Enviar el mensaje de eco de vuelta al cliente.
4. Cerrar los flujos y la conexión del socket creado en el método `accept()`.

Una vez finalizado el servidor multihilo de eco TCP, para comprobar que funciona correctamente deberás ejecutarlo desde el IDE (pasándole como parámetro el número de puerto), el cual se quedará esperando a que los clientes le envíen peticiones.

Desde un terminal ejecuta un comando `nc` que se conecte con el servidor, y déjalo conectado:

```
nc localhost 5000
```

Desde el IDE, deberás ejecutar el cliente TCP: (pasándole como parámetro la dirección IP o nombre del servidor (por ejemplo, localhost), el puerto en el que escucha el servidor (por ejemplo, 5000) y el mensaje (por ejemplo, “Testing my TCP server”).

En este caso, el servidor lanzará un hilo para atender la conexión realizada con el `nc`. Al ser multihilo, lanzará otro hilo de ejecución en paralelo para atender la petición realizada con el cliente de eco. El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor, tal y como sucedía en el caso monohilo. Para finalizar la conexión del `nc`, simplemente introduce algo por teclado y pulsa ENTER: esto enviará un mensaje al servidor, que responderá con el eco y cerrará la conexión.

7. Evaluación

El contenido de este tutorial será evaluado en el primer examen de prácticas que, en su conjunto, supondrá hasta 1.25 puntos de la calificación final. Este examen contendrá preguntas sobre las prácticas de programación (p0 y p1).