Practica 0 y 1 de Redes

P0 -> Tutorial de sockets

Flujos de datos

Los programas pueden necesitar recoger información de una fuente externa y enviar información a un destino externo. ¿Qué información? -> Objetos , Caracteres, Imágenes, Sonidos.

El programa usara **flujos de datos** para leer y enviar la información necesaria.

Secuencias de bytes

Para realizar entradas/salidas. Usan FileInputStream y FileOutputStream.

Uso de secuencias de bytes

CopyBytes usa flujos de bytes para copiar un byte a la vez. Copia los contenidos de un archivo llamado "xanadu.txt" a otro llamado "outagain.txt". Pasos:

-Crear los objetos FileInputStream y FileOutputStream, inicializándolos a null.

-Abre "xanadu.txt" con FileInputStream con el método read() de este, y escribe los bytes en "outagain.txt" usando write, de FileOutputStream.

-Cierra los dos objetos para liberar los recursos usados.

El programa usa un bloque "try-finally" para asegurarse de que los objetos se cierran adecuadamente, incluso si ocurre una excepción.

CopyBytes pasa la mayor parte del tiempo en un bucle simple que lee el flujo de entrada y sigue el flujo de salida, un byte a la vez.

CERRAR SIEMPRE LOS ARROYOS: Esta práctica ayuda a evitar graves fugas de recursos. Un posible error es que no se pudo abrir algún archivo. Cuando eso sucede, la variable de flujo del archivo no cambia de su valor inicial. Es por eso que se asegura de que cada variable de flujo contenga una referencia de objeto antes de invocar

CUANDO NO USAR SECUENCIAS DE BYTES: `CopyBytes` parece un programa normal, pero en realidad representa un tipo de E/S de bajo nivel que debe evitar. Dado que contiene datos de caracteres, el mejor enfoque es usar **secuencias de caracteres**.

¿Entonces porque hablamos de flujos de bytes? -> Porque todos los demás tipos de flujo se basan en los flujos de bytes.

Secuencias de caracteres

Todas las clases de flujo de caracteres decienden de Lector y Escritor. Al igual que con las secuencias de bytes, hay clases de secuencias de caracteres que se especializan en E/S de archivos: **FileReader** y **FileWriter**. (usa **inputStream** y **outputStream**.

Hay dos secuencias "puente" de byte a carácter de uso general: **InputStreamReader** y **OutputStreamWriter**. Uselos para crear secuencias de caracteres cuando no haya clases de secuencias de caracteres preempaquetadas que satisfagan sus necesidades.

E/S ORIENTADAS A LINEAS:

La E/S de caracteres generalmente ocurre en unidades más grandes que en caracteres individuales. Una unidad común es la línea. Modifiquemos el ejemplo para usar E/S orientadas a líneas. Para ello, tenemos que usar dos clases nuevas: **BufferedReader** y **BufferedWriter**.

Flujos en buffer

Hay 4 clases de secuencias en buffer que se usan para encapsular secuencias sin bufer: **BufferedInputStream** y **BufferedOutputStream** crean secuencias de bytes almacenadas en buffer, mientras que **BufferedReader** y **BuferedWriter** crean secuencias de caracteres almacenadas en buffer.

IMPLEMENTACIÓN DE COPY: P0

Debemos desarrollar una aplicación que permita copiar ficheros de texto y ficheros binarios usando flujos de datos.

Solución:
Flujo de caracteres no podríamos usar, porque por ejemplo, una imagen no tiene un conjunto de caracteres, tiene caracteres en binario. Entonces usaremos el ejemplo de flujo de caracteres (VER FUNCIONAMIENTO DE FLUJO DE CARACTERES).

IMPLEMENTACIÓN DE INFO: P0

Debemos desarrollar un método que permita obtener las principales propiedades de un fichero:

Solución:

Almacenamos el archivo en una variable File. Si el archivo existe, comenzamos a enunciar las propiedades del fichero

-tamaño del archivo: **archivo.length()**

-fecha de ultima modificación: **archivo.lastModified()**
-nombre del archivo: **archivo.getName()**
-Extension del archivo: **archivo.getName()** y **Split(\\.)** obtenemos la extensión del archivo a la derecha del punto.
-Tipo de fichero: Files.probeContentType(archivo.toPath())
-Ruta absoluta: **archivo.getAbsolutePath()**

TCP (Transmission Control Protocol)

Protocolo orientado a conexión, que 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…

UDP (User Datagram Protocol)

Protocolo no orientado a conexión que envia paquetes de datos (datagramas) independientes sin garantías. Permite broadcast y multicast.

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 maquinas diferentes de dicha red, ¿Qué necesitamos para que dichos procesos se puedan comunicar entre si?

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

Una misma maquina puede tener una o varias conexiones físicas a la red y múltiples servidores pueden estar escuchando en dicha maquina. Si a través de una de las conexiones físicas se recibe una petición por parte de un cliente, ¿Cómo se identifica que proceso debe atender la petición? -> Aquí surge el concepto de **puerto**, que permite a TCP y UDP dirigir los datos a la aplicación correcta de entre todas las que se están ejecutando en la maquina. Todo servidor, por tanto, ha de estar registrado en un puerto para recibir los datos que a el se dirigen.

Los datos transmitidos a través de la red tendrán información para identificar a la maquina mediante su dirección IP y el puerto a los que van dirigidos.

Los puertos:

- Son independientes para TCP y UDP
- Se identifican por un numero de 16 bits (de 0 a 65535)
- Algunos de ellos están reservados (de 0 a 1023).

Por tanto, un **socket** se puede definir como un extremo de un enlace de comunicación bidireccional entre 2 programas que se comunican por la red (se asocia un numero de puerto).

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

Sockets UDP

Los sockets UDP son no orientados a conexión. Los clientes no se conectaran 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** -> Mensaje independiente, enviado a través de una red, cuya llegada, tiempo de llegada y contenido no están garantizados.

Implementación del servidor de eco UDP:

Se crea un socket no orientado a conexión. Es importante recalcar que no es necesario especificar un numero de puerto en el constructor, ya que el propio constructor se encargara de seleccionar un puerto libre.

DatagramSocket

Establecemos un tiempo de espera máximo para el socket.

.setSoTimeOut([TIMEOUT])

Se obtiene la IP de la maquina en la que se encuentra el servidor, a partir del primer argumento recibido por la línea de comando. La clase **InetAdress** representa en Java el concepto de dirección IP. Esta clase dispone del método estático **getByName()** que contiene la IP a partir del **String** que recibe como parámetro. Este parámetro puede ser un nombre o una IP.

Se obtiene el nº de puerto en el que se esta ejecutando el servidor.

Se obtiene el mensaje que queremos enviar al servidor.

Preparamos el datagrama que vayamos a enviar, indicando: el mensaje que queremos enviar, el numero de bytes a enviar, la dirección IP del destinatario y el puerto del destinatario.

Enviamos el datagrama invocando el método **send()** del socket que hemos creado inicialmente.

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 un nuevo datagrama, se indicara el array donde queremos almacenar la respuesta y el numero máximo de bytes que puede almacenar este array

Recibimos el datagrama, usando el método **receive()** de nuestro socket UDP. Este método es bloqueante, es decir, el programa se quedara esperando en este método hasta recibir algo o, como hemos establecido un timeout, hasta que venza el timeout.

Por ultimo, cerramos el socket.

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 esta establecida, el protocolo TCP garantiza que los datos son enviados correctamente y debidamente ordenados en el otro extremo.

IMPLEMENTACIÓN DEL SERVIDOR DE ECO TCP: P0

Implementación del servidor monohilo:

Se obtiene la IP de la maquina en la que se encuentra el servidor, a partir del primer argumento recibido por línea de comandos.

Se obtiene el nº de puerto en el que esta ejecutándose el servidor, a partir del segundo argumento recibido por la línea de comandos.

Se obtiene el mensaje que queremos enviar al servidor, a partir del tercer argumento recibido por línea de comandos.

Se crea un socket orientado a conexion. No se especifica ningún puerto para el cliente, ya que automáticamente se seleccionara un puerto libre (efimero). 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.

Establecemos un tiempo de espera máximo para el socket. Si pasado ese tiempo no ha recibido nada se lanzara la excepción correspondiente.

Creamos el canal de entrada para recibir los datos del servidor. Para leer los datos del servidor se usa 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.

Creamos el canal d salida para enviar datos al servidor. Se usa la clase **PrintWriter**, construida a partir del método **getOutputStream()** del socket cliente. Esta clase dispone de métodos **println()** y **print()**, equivalentes a los usados habitualemtne para imprimir por pantalla.

Enviamos el mensaje al servidor invocando el método **println()** del flujo de salida.

Esperamos y recibimos la respuesta del servido invocando el método **readLine()** del flujo de entrada

Por último, cerramos el socket.

Implementación del servidor multihilo:

Crear un ServerSocket, asociado a un numero de puerto especifico.
Establecer un tiempo de espera máximo para el socket

Crear un bucle infinito:

*Invocar el método **accept()** del socket servidor. Al establecerse la conexión, devuelve un nuevo socket que se usara para la comunicación con ese cliente.

*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.

*Iniciar la ejecución del hilo con el método **start()**.

Implementación de la clase **ServerThread**:
Preparar los flujos de entrada y salida
Recibir el mensaje del servidor
Enviar el mensaje de eco de vuelta al cliente
Cerrar los flujos y la conexion del socket creado en el método **accept()**.

P1 -> Implementación de un Servidor Web

Comprobaciones:

1. Multithread

2. HTML

3. IMAGE

4. TEXT

5. BADREQUEST

5. NOTFOUND

6. HEAD

7. HEADIMAGE

8. HEADNOTFOUND

9. IFMODIFIEDSINCE

Implementación del WebServer:

Comprobamos que solo entra un argumento

Conseguimos el número de puerto del argumento

Iniciamos el servudor con un **ServerSocket** y inicializamos un TIMEOUt.

Creamos un bucle donde cada vez que se recibe una conexión por el **ServerSocket** , se crea un nuevo hilo y se pone en marcha con un **Socket**, un **ServerThread** y un **.start()** de la clase **ServerThread**. Luego, las excepciones.

Implementación del ServerThread

Run

Inicializamos un **BufferedReader** para leer la entrada del socket.

Creamos una cadena para almacenar la primera línea de la solicitud, la cual si es vacia, se sale de la función.

Se divide la variable msg en sus partes separadas, y se crea un objeto "archivo" a partir de la segunda parte de la solicitud del cliente. También se crean dos archivos para manejar casos de error, uno para 400 y otro para 404.

Luego, la función verifica si el archivo existe, si es asi, verifica si la solicitud es un método "GET" o "HEAD". Si es "GET", la función llama a ProcessRequest y el contenido del archivo debe enviarse al cliente. SI es "HEAD", se pasa a ProcessRequest y solo se deben enviar los encabezados HTTP al cliente. Si no es ninguno se llama a processNotValidRequests con el archivo de error 400.

Si el archivo solicitado no existe, la función llama a processNotValidRequests con el archivo de error 404.

Finalmente, la función cierra el socket.

processRequest

Se encarga de procesar una solicitud HTTP y enviar la respuesta correspondiente al cliente que realizo la solicitud.

Verifica con **OutputStream** si la línea comienza con "If-Modified-Since". Si es asi, la función compara la fecha de última modificación del archivo solicitado con la fecha especificada en la solicitud HTTP. SI la fecha no es anterior a la fecha especificada, significa que no ha sido modificado y se enviara "304 Not Modified" al cliente y devuelve sin enviar el archivo.

Si la solicitud HTTP no contiene "If-Modified-Since" o si el archivo fue modificado, se envía "200 OK" y llama a la función "InformacionValores".

Si se debe enviar el archivo al cliente, se crea un buffer y un entero para almacenar la cantidad de bytes leídos del archivo de entrada en el bufer en cada iteración del ciclo while.

Finalmente, la función limpia y cierra el objeto **OutputStream** del socket del cliente.

processNotValidRequest

La función processNotValidRequests se encarga de procesar las solicitudes HTTP que no son validas.

La función comienza abriendo un **FileInputStream** para leer el contenido del archivo de error y un **OutputStream** para escribir la respuesta en el flujo de salida del cliente. Si **found** es verdadero, se envia un mensaje de error "Bad Request 400". Si es falso, se envia "Not Found 404".

Se llama a **InformacionValores.**

Si **writer** es verdadero, la función lee el contenido del archivo de error y lo escribe en el flujo de salida del cliente usando un ciclo while.

Finalmente, la función limpia el flujo de salida del cliente para asegurarse de que todos los bytes se hayan escrito.

InformacionValores

-Fecha: **ZonedDateTIme.now()**

-Server: “WebServer_695”

-Content-Length: **file.length**

-Tipo de contenido: Función auxiliar…

-Fecha de ultima modificación: **.lastModifiedTime()**

Se enseña por el **OutputStream** con la función **.write**