Programación en red: Sockets I

En aplicaciones **cliente-servidor**, el servidor proporcional algún servicio, mientras que el cliente hace uso del servicio proporcionado por el servidor. La comunicación que debe existir entre el cliente y servidor debe ser confiable, es decir, no se puede descartar datos, y estos deben llegar al cliente en el mismo orden en el que fueron enviados por el servidor.

El **protocolo TCP** proporciona un canal de comunicación punto a punto confiable que las aplicaciones cliente-servidor en el internet utilizan para comunicarse mutuamente. Para comunicarse a través de TCP, el programa cliente y el programa servidor establecen una conexión entre ellos. Cada programa vincula un socket a su extremo de la conexión, y escriben y leen el socket vinculado para comunicarse.

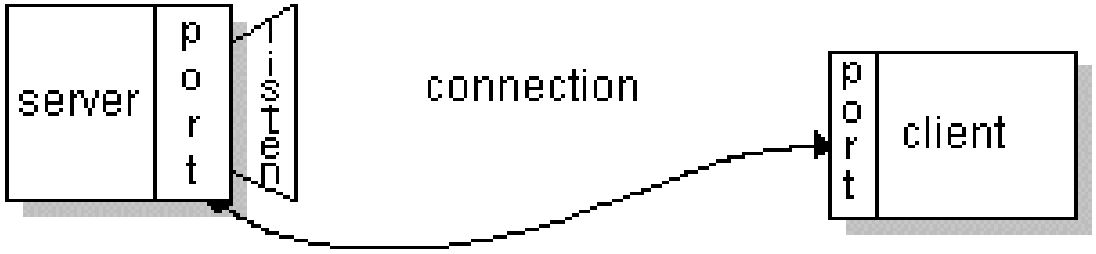
- El cliente debe conocer el hostname de la máquina en la que se ejecuta el servidor y el puerto al cual se encuentra escuchando.

- Para hacer una solicitud de conexión, el cliente intenta encontrarse con la máquina y puerto del servidor.

- El cliente también necesita identificarse a sí mismo con el servidor por lo que vincula un número de puerto local que utilizará durante la conexión.

- Si todo va bien, el servidor acepta la conexión.

- Una vez aceptada la conexión, el servidor necesita un nuevo socket para que pueda seguir escuchando al socket original por solicitudes de conexión mientras atiende las necesidades del cliente conectado. Para ello vincula un nuevo socket al mismo puerto local, y también tiene su endpoint remoto establecido a la dirección y puerto del cliente.



- Si la conexión es aceptada, el socket es creado exitosamente y el cliente lo puede utilizar para comunicarse con el servidor.

InetAddress

InetAddress es una clase que representa una dirección IP (IPv4 o IPv6) y encapsula también el nombre de dominio de dicha dirección.

Dado que esta clase no cuenta con constructores, es necesario utilizar los métodos de fábrica de esta clase para crear objetos InetAddress, es decir, los métodos estáticos que devuelvan una instancia de la misma clase. Entre estos métodos se tienen:

- static InetAddress getLocalHost( )
  - Retorna la dirección del host local.
- static InetAddress getLoopbackAddress( )
  - Retorna la dirección de loopback.

- static InetAddress getByName(String host)
  - Determina la dirección IP de un host dando su hostname.
  - **Obs:** El nombre de host ingresado puede ser un nombre de máquina como “nombre.com” o una representación textual de su dirección IP.
  - Si se entrega null como argumento, se retornará un InetAddress representando la interfaz de loopback.

- static InetAddress getByAddress(byte[ ] addr)
  - Retorna un objeto InetAddress dado una dirección IP sin procesar.
  - El argumento se debe encontrar en el orden de red, es decir, el byte de mayor orden se debe encontrar en la primera entrada.
  - Las direcciones IPv4 deben ser de 4 bytes de longitud, mientras que las IPv6, de 16.

Métodos de InetAddress

- byte[ ] getAddress( )
  - Retorna la dirección IP sin procesar de este objeto InetAddress.
- String getHostAddress( )
  - Retorna la dirección IP de forma textual a través de un String.
- String getHostName( )
  - Retorna el nombre de host para esta dirección IP.
- int hashCode( )
  - Retorna un código hash para la dirección IP del objeto InetAddress.

Sockets

Definición:

- Un socket es un **endpoint** de un enlace de comunicación bidireccional entre dos programas que se ejecutan en la red. Un socket esta vinculado a un número de puerto para que la capa TCP pueda identificar la aplicación a la que están destinados los datos.

Un endpoint es una combinación de una dirección IP y un número de puerto. Cada conexión TCP puede ser únicamente identificada por sus dos endpoints.

El paquete **java.net** incluye un par de clases para implementar una conexión bidireccional entre un programa Java y otro programa en la red:

- **ServerSocket**
- **Socket**

Adicionalmente, se puede usar las clases **URL** y otras clases relacionadas (**URLConnection** y **URLEncoder**) preferiblemente si deseamos conectarnos a la web

ServerSocket

Esta clase implementa un **socket de servidor**. Un socket de servidor espera a que lleguen solicitudes a través de la red, realiza algunas operaciones en base a la solicitud, y posiblemente retorne un resultado al solicitante.

Para instanciar un ServerSocket basta con indicar el número de puerto a vincular.

- ✓ServerSocket(int port)
  - ➢Arroja un IOException si ocurre algún error de I/O durante la creación del socket.
  - ➢Arroja un SecurityException si existe un adminisrador de seguridad que no permita este método.
  - ➢Arroja un IllegalArgumentException si el puerto está fuera del rango aceptado, i.e. entre 0 y 65535.

Observaciones:

- ✓Si se entrega 0 como puerto, se tomará un número de puerto asignado automáticamente.
- ✓La longitud máxima de la cola de solicitudes de conexión entrante es 50. Si una solicitud llega cuando la cola se encuentra llena, la conexión será rechazada.

Los dos método más importantes, y esenciales para el establecimiento de la conexión, de un Server socket son:

- Socket accept( )
  - Inicia un estado en el que el socket escucha por una conexión a ser realizada y la acepta. Este método bloquea el hilo hasta realizar la conexión.
  - Puede arrojar múltiples tipos de excepciones.
  - Retorna un objeto Socket vinculado a la nueva conexión.

- synchronized void close( )
  - Cierra el socket. Cualquier hilo bloqueado por el método accept( ) arrojará una SocketException y liberará el bloqueo.

Socket

Esta clase simplemente implementa un socket de cliente para una comunicación bidireccional entre dos máquinas.

Para instanciar un Socket necesitamos por lo menos la dirección IP del servidor o su hostname, y el puerto al cual se encuentra escuchando. Por tanto podemos hacer uso de los dos siguientes constructores:

- ✓Socket(InetAddress address, int port)
  - ➢Crea un stream socket y lo conecta al puerto especificado de la dirección IP especificada.
- ✓Socket(String host, int port)
  - ➢Crea un stream socket y lo conecta al puerto especificado de la host mencionado.

Para poder efectuar la comunicación a través de sockets necesitamos identificar los medios a través de los cuales se realizará. Para ello, la clase Socket cuenta con dos métodos básicos que retornan los stream de entrada y salida relacionados a la conexión establecida.

- InputStream getInputStream( )
  - Puede arrojar un IOException si ocurre algún error de I/O al crear stream de entrada, el socket está cerrado, el socket no está conectado, o la entrada del socket ha sido apagada usando shutdownInput( ).

- OutputStream getOutputStream( )
  - Puede arrojar un IOException si ocurre algún error de I/O al crear el stream de salida o si el socket no se encuentra conectado.

Observación:

Los streams retornados únicamente pueden ser usados para leer y escribir arrays de bytes por lo que es necesario instanciar otras clases en base al stream obtenido para trabajar con otros objetos o tipos de variables.

Por otra parte, en cierto punto será necesario cerrar la conexión para poder liberar cualquier recurso del sistema que este siendo utilizado con los stream de entrada y salida. Para ello se cuenta con el siguiente método:

- synchronized void close( )
  - Cualquier hilo que se encuentre bloqueado en alguna operación I/O en relación a este socket arrojará una SocketException.
  - Una vez que un socket ha sido cerrado ya no puede ser usado en otros trabajos de red, es decir, no puede ser reconectado, se debe crear un nuevo socket.
  - Cerrar el socket cierra también el InputStream y OutputStream.
  - **El método puede arrojar un IOExeption.**

Nota:

Los streams de entrada y salida también pueden cerrarse haciendo uso de un método con este mismo nombre. Además, todas las clases para stream de entrada o salida que se verán a continuación también pueden invocar al método close.

Al manejar un socket por el lado del cliente, se tienen 5 pasos básicos para realizar la comunicación:

1. Abrir un Socket.

2. Abrir un stream de entrada y un stream de salida al socket.

3. Leer o escribir a través del stream acorde a los protocolos del servidor.

4. Cerrar los stream.

5. Cerrar el Socket.

Sólo el tercer paso puede diferir de cliente en cliente debido a la complejidad del servidor con el cual se este estableciendo la conexión.

InputStream con Scanner

Para instanciar la clase Scanner de java necesitamos entregarle como argumento un objeto InputStream (e.g. System.in). Por tanto, también podemos crear un objeto Scanner en base al InputStream de un socket.

```
ServerSocket s = new Socket( 8181 );
Socket cliente = s.accept( );

InputStream streamIn = cliente.getInputStream( );
Scanner in = new Scanner( streamIn );
```

Luego, podemos usar cualquiera de los métodos ya conocidos de la clase Scanner para esperar por un tipo de entrada en particular.

```
String msg = in.nextLine( );
System.out.println(msg);
```

PrintWriter

Esta clase imprime representaciones formateadas de objetos en un stream de salida de texto. Por tanto, podemos instanciar esta clase en base al objeto OutputStream de un socket.

```
ServerSocket s = new Socket( 8181 );
Socket client = s.accept( );

PrintWriter out = new PrintWriter( client.getOutpuStream( ) );
PrinWriter out2 = new PrintWriter( client.getOutputStream , true );
```

La segunda forma de instanciar la clase especifica un booleano indicando si se realizará un flush automático en el buffer de salida luego de ejecutar los métodos println, prinft o format.

Para enviar un mensaje a través del socket, simplemente se deben invocar los métodos print, println o printf. Los dos primeros pueden recibir distintos tipos de variable, entre ellos caracteres, números u objetos.

InputStreamReader

Un **InputStreamReader** es un puente entre stream de bytes y stream de caracteres: lee bytes y los decodifica en caracteres usando un charset especificado o el charset por defecto.

Un InputStreamReader debe ser construído tomando como base un InputStream ya existente.

- ✓InputStreamReader(InputStream in)
  - ➢Crea un InputStreamReader que utiliza el charset por defecto.

Para una máxima eficiencia, es recomendable envolver un InputStreamReader con un BufferedReader

BufferedReader

Esta clase lee texto desde un stream de entrada de caracteres, almacenando caracteres en el búfer para proporcionar una lectura eficiente de caracteres, arrays y líneas.

Los BufferedReader deben ser instanciados tomando como argumento un objeto Reader ya existente (InputStreamReader es una subclase de la clase Reader).

- ✓ BufferedReader(Reader in)
  - ➢ Crea un stream de entrada de caracteres con un búfer con un tamaño por defecto.

El tamaño del búfer también puede ser especficado.

- ✓ BufferedReader(Reader in, int sz)
  - ➢ Puede arrojar un IllegalArgumentException si sz es menor o igual a 0.

Para poder leer a través de un BufferedReader se pueden utilizar los siguientes métodos :

- int read( )
  - Retorna el carácter leído como un entero o -1 si se ha llegado al final del stream.
  - Puede arrojar IOException.
- String readLine( )
  - Retorna una línea de texto sin incluir el carácter de terminación de línea (salto de línea o carriage return), o null si se ha llegado al final del stream.
  - Puede arrojar IOException.

```
ServerSocket s = new Socket( 8181 );
Socket client = s.accept( );

BufferedReader in = new BufferedReader(
            new InputStreamReader( client.getInputStream( ) ) );
String line = in.readLine( );

in.close( );
```

ObjectOutputStream

Un **ObjectOutputStream** escribe tipos de datos primitivos y grafos de objetos de java en un OutputStream. Los objetos pueden ser leídos usando un ObjectInputSteam

```
ServerSocket s = new Socket( 8181 );
Socket client = s.accept( );

OutputStream outStream = client.getOutputStream( );
ObjectOutputStream out = new ObjectOutputStream( outStream );
```

El método writeObject( ) es usado para escribir un objeto en el stream. Cualquier objeto, incluyendo Strings y Arrays, son escritos con este método. Los objetos deben ser leídos por el ObjectInputStrem correspondiente con los mismo tipos y el mismo orden en el que fueron escritos.

```
out.writeInt( 12345 );
out.writeObject( "Today“ );
out.writeObject( new Date() );

out.close( );
```

ObjectInputStream

Un **ObjectInputStream** deserializa los datos primitivos y objetos previamente escritos usando un ObjectOutputStream.

```
ServerSocket s = new Socket( 8181 );
Socket client = s.accept( );

OutputStream inStream = client.getInputStream( );
ObjectInputStream in = new ObjectInputStream( inStream );
```

El método readObject( ) es usado para leer los objetos del stream. Los datos pasados son tratados como objetos durante la serialización, por lo que es necesario realizar un casteo hacia el tipo esperado. Los tipos primitivos puede ser leídos haciendo uso de los métodos apropiados. Del ejemplo anterior, tenemos la siguiente secuencia de sentencias:

```
int i = in.readInt( );
String today = ( String ) in.readObject( );
Date date = ( Date ) in.readObject( );

in.close( );
```

FilterOutputStream

Esta clase es una superclase de todas las clases que filtran streams de salida. Estos streams se sientan sobre streams de salida ya existentes que utilizan como un sumidero básico de datos, pero posiblemente transformando los datos a lo largo del camino, o proporcionando funcionalidad adicional.

La clase **FilterOutputStream** simplemente sobrescribe todos los métodos de OutputStream con versiones que pasen todos los requisitos del stream de salida subyacente.

Las subclases de FilterOutputStream podrían sobrescribir aún más estos métodos, así como también proporcionar métodos y campos adicionales.