04

Programación de Servicios y Procesos



Comunicaciones en Red

Generación de servicios en Red

* Índice

[ Introducción 4](#_Toc536044498)

[.1. Comunicaciones en Red 5](#_Toc536044499)

[.1.1. Fundamentos de Sockets 7](#_Toc536044500)

[.1.2. El protocolo de conexión de los Sockets 16](#_Toc536044501)

[.2. Generación de Servicios en Red 22](#_Toc536044502)

[.2.1. Java Sockets 22](#_Toc536044503)

[.2.2. Apertura de Sockets 27](#_Toc536044504)

[.2.3. Creación de Streams y Cierre de Sockets 31](#_Toc536044505)

[.3. Ejemplo de aplicación 36](#_Toc536044506)

[.3.1. Cliente 36](#_Toc536044507)

[.3.2. Servidor 38](#_Toc536044508)

[.3.3. Ejecución 42](#_Toc536044509)

* Introducción

En unidades anteriores vimos cómo crear hilos y la importancia de la sincronización de los mismos en un sistema multiprogramación actual.

En este tema veremos cómo implementar las comunicaciones entre estos hilos, de modo que puedan compartir datos ya no sólo a través de memoria compartida, sino incluso estando cada uno de estos hilos en una máquina distinta. Estas comunicaciones en red las realizaremos de forma práctica mediante sockets utilizando las clases Socket y ServerSocket que nos proporciona JAVA, analizando el protocolo completo de comunicación en una arquitectura cliente/servidor e implementándola a través de los distintos métodos de las clases Socket y ServerSocket.

Veremos también mediante proyectos sencillos cómo crear servicios en red a través de la programación de servidores monocliente y multicliente, utilizando toda la potencia que nos proporciona la multiprogramación, de modo que nuestros servicios de red puedan dar servicio simultáneo a una gran cantidad de clientes que los soliciten.

No podremos dejar de lado los problemas del acceso a datos compartidos, que en la parte del servidor nos llevará a que si varios clientes de manera simultánea quieren acceder a modificar unos datos compartidos tengamos la necesidad de implementar secciones críticas de modo que garanticemos zonas de exclusión mutua como vimos en el tema anterior.

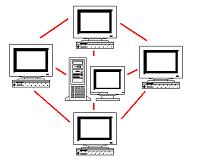
* 1. Comunicaciones en Red

Como ya conocemos la programación multihilo nos lleva irremediablemente a que, al programar un sistema mediante hilos colaborativos, nuestros hilos tengan la necesidad de compartir datos.

Cuando escalamos nuestros proyectos, es posible que estos hilos se encuentren en diferentes máquinas, de modo que la colaboración entre ellos se debe hacer a través de una red, de modo que cada hilo tendrá que conocer la dirección IP de la máquina donde está el proceso con el que quiere comunicar y el puerto de la máquina dedicado para esa comunicación.

Existen multitud de protocolos para realizar la comunicación en red a nivel de aplicación (telnet, ftp, http, pop3, smtp, entre otros), pero a nosotros en este curso nos interesará crear nuestras propias comunicaciones a través de JAVA, partiendo de cero y creando desde los propios canales de comunicación hasta nuestro propio protocolo, de modo que podamos ser capaces de controlar completamente la comunicación.

|  |  |
| --- | --- |
| Actividad | **Entra en las siguientes direcciones web y profundiza en el conocimiento de los protocolos de comunicación en red a nivel de aplicación:**  [**https://es.ccm.net/contents/708-uso-de-telnet**](https://es.ccm.net/contents/708-uso-de-telnet)  [**http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-es-4/ch-ftp.html**](http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-es-4/ch-ftp.html)  [**https://blog.mailrelay.com/es/2017/04/25/que-es-el-smtp**](https://blog.mailrelay.com/es/2017/04/25/que-es-el-smtp) |
| Definicion | **Un protocolo de comunicación en red** es el conjunto de reglas que especifican cómo debe ser el intercambio de datos durante la comunicación entre las diferentes entidades que forman parte de una red.  Establece la **semántica** y la **sintaxis** de este intercambio de información a través de la red así como los turnos de cada uno de las entidades implicadas en la comunicación |



Las entidades que forman parte de la comunicación a través de la red a su vez pueden adoptar distintos roles. De este modo puede haber una comunicación de igual a igual como por ejemplo pueden ser los protocolos de comunicación entre routers, en los que cada uno de los cuales utiliza mecanismos propios para conseguir la convergencia en la red y para determinar el mejor camino que puede seguir un paquete de datos en su viaje hasta la máquina destino (protocolos de gateway interior (IGP), de gateway exterior (EGP) o de frontera).

Otro tipo de comunicación a través de red es la arquitectura cliente/servidor, en la que una o varias entidades toman el papel de Servidor que harán pública sus direcciones IP y los puertos desde los que atenderán las peticiones de conexión y el resto de entidades actuarán como clientes que enviarán peticiones de conexión y cuando sea atendida su petición por el servidor comenzará la comunicación.

|  |  |
| --- | --- |
| Actividad | **Entra en las siguientes direcciones web y profundiza en el conocimiento de los protocolos de comunicación en red a nivel de aplicación:**  [**https://es.ccm.net/contents/148-entorno-cliente-servidor**](https://es.ccm.net/contents/148-entorno-cliente-servidor) |

* + 1. Fundamentos de Sockets

En la mayoría de Sistemas Operativos, ha existido la necesidad de que los procesos intercambien datos entre sí de una manera más compleja y con un volumen mayor que el que pueden soportar las zonas de memoria compartida. Además, con el desarrollo de las redes de comunicaciones aparece la necesidad de comunicar procesos que se ejecutan en diferentes máquinas, por lo que la solución de compartir datos a través de una zona de memoria compartida se hace inviable.

Es entonces cuando desde la universidad norteamericana de Berkeley a través de la Berkeley Software Distribution aparece y se populariza el concepto de socket. En 1983 se integran los sockets en la distribución 4.2BSD de Unix, lo que ayuda enormemente a su distribución y uso por parte de la comunidad de programadores del momento.

|  |  |
| --- | --- |
| Definicion | **Los Sockets** son mecanismos de comunicación entre procesos que permite el intercambio de datos bidireccionales entre procesos que se ejecutan en la misma máquina.  Los sockets IP (especialmente los sockets TCP / IP) son un mecanismo que permite la comunicación entre procesos a través de la red. En algunos casos, puede usar sockets TCP / IP para hablar con los procesos que se ejecutan en la misma computadora mediante la interfaz de bucle invertido (utilizando como IP la dirección de localhost 127.0.0.1). |

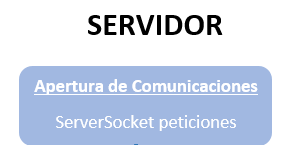
|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de la Universidad de Berkeley donde se desarrollaron y popularizaron los primeros sockets y realiza la siguiente lectura recomendada:  <http://inst.eecs.berkeley.edu/~ee122/fa09/notes/03-SocketProgrammingx6.pdf> |

Así pues, los sockets nos permiten implementar de manera muy sencilla una arquitectura del tipo cliente/servidor, en la cual uno de nuestros procesos estará en modo escucha esperando recibir las peticiones de conexión de uno o varios clientes simultáneos a los que dará servicio basándose en las peticiones que estos de realicen.

Los sockets están basados en un protocolo que establece el punto de conexión para conectar a una máquina remota a través de una dirección IP y un puerto determinado que estará dedicado para esta comunicación en modo escucha. A partir de ahí servidor acepta la comunicación con el cliente y comienza el intercambio de lectura y escritura de información de manera bidireccional. La comunicación finaliza cuando cliente o servidor se desconectan del socket.

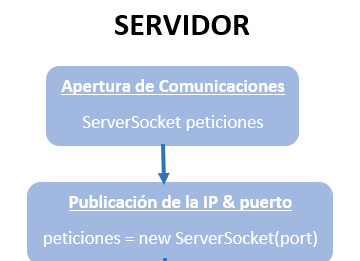
A través de este protocolo de comunicación definido se puede crear un sistema de comunicaciones muy completo que vamos a analizar detenidamente utilizando sockets de Java

En un principio tendremos al proceso o hilo servidor ejecutándose en su máquina, lo primero que tiene que hacer el servidor es abrir en el canal de comunicaciones definiendo un socket especial para servidores que será de la clase ServerSocket.



Definición del Socket de servidor de la clase ServerSocket

Acto seguido, el servidor creará un objeto al que le pasará en su constructor el número de puerto que quiere dedicar para la comunicación. Es interesante ver como en la construcción del objeto no se le pasa la dirección IP del servidor ya que esta dirección no es necesaria a la hora de crear un socket de escucha, aunque sí que deberán conocerla los hilos cliente que quieran conectar con el servidor.



Creación del objeto Socket de servidor y publicación del puerto

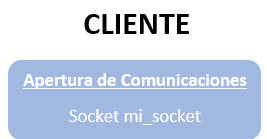
Una vez publicada en la red la dirección del canal de comunicaciones, nuestro servidor entra en un bucle en el que simplemente espera recibir las solicitudes de comunicación. esto se realiza a través del método "accept()" de nuestro ServerSocket.

Esta llamada por parte del servidor al método "accept()" implica que el servidor quede bloqueado hasta que llegue una solicitud de comunicación con él por parte de algún cliente.



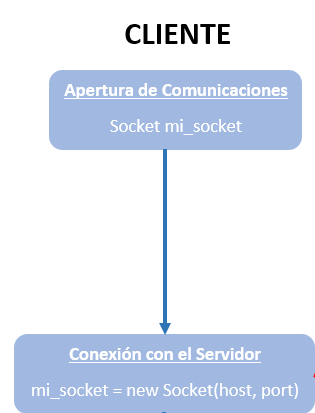
Creación del objeto Socket de servidor y publicación del puerto

Por su parte el cliente cuando quiera conectar con el servidor lo primero que tiene que hacer es abrir también un canal de comunicaciones a través de un socket cliente perteneciente a la clase Socket de Java.



Definición del objeto Socket de Cliente

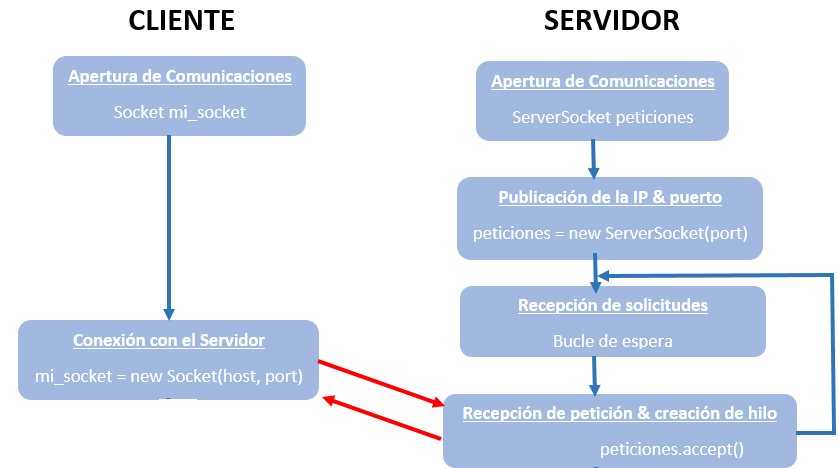
Cuando creemos el objeto de la clase Socket, debemos indicarle a su constructor con quién queremos conectar, de modo que tendremos que pasarle la dirección IP en el primer parámetro y el puerto de conexión en el segundo.



Creación del objeto Socket de Cliente

|  |  |
| --- | --- |
| Atencion | **Si estamos haciendo una conexión de un cliente a un servidor dentro de mi propia máquina,** en la dirección IP puedo utilizar la constante "localhost" o la dirección de bucle interno 127. 0. 0.1 |

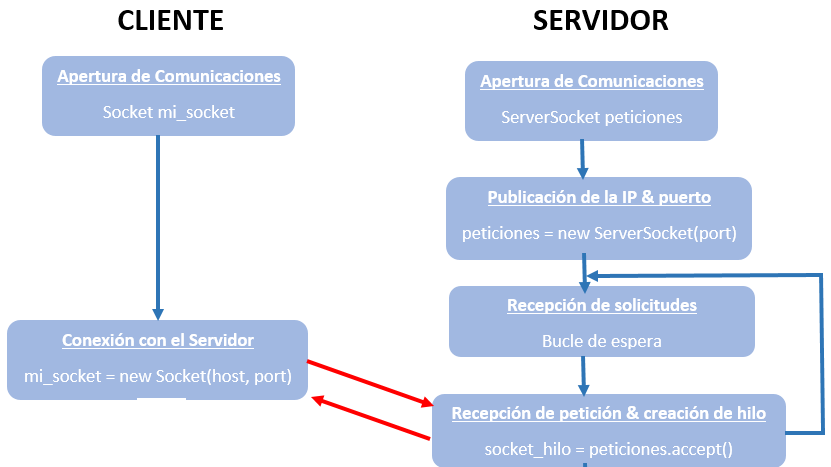
Al crear un socket de cliente con la dirección IP y el puerto del servidor donde queremos conectar se genera en red la petición de conexión que llegará hasta la máquina servidor y entrará por el puerto especificado haciendo qué el servidor se desbloquee del "accept()" donde estaba esperando la petición de comunicación. A su vez el servidor dará su visto bueno a la conexión, de modo que el cliente sabrá que su petición de conexión ha sido atendida con éxito.



Conexión de Cliente y Servidor

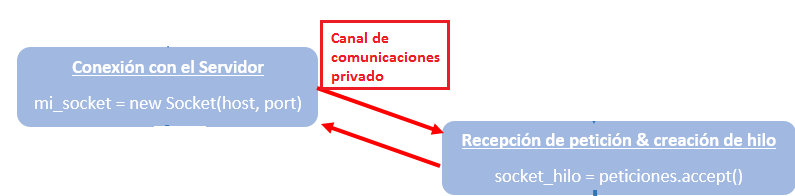
|  |  |
| --- | --- |
|  | Atencion**Si estamos intentando realizar una conexión de un cliente a un servidor que no existe** (por IP o puerto erróneo o no dedicado) o que ha cerrado ya su puerto de escucha,el cliente inmediatamente sabrá que no existe nadie al otro extremos del canal de comunicación y abortará la conexión presentando el correspondiente error. |

A partir de este punto el servidor, si es un servidor multi cliente, creará un hilo nuevo que atenderá de manera exclusiva esa comunicación con el cliente, de modo que el servidor puede volver a su bucle de espera de solicitudes y volver a ejecutar un nuevo "accept()" esperando la solicitud de comunicación de un segundo cliente.



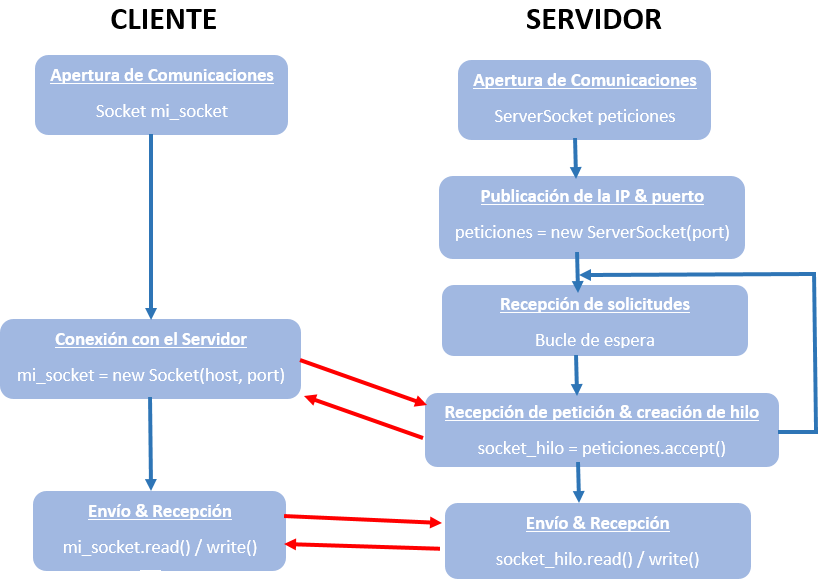
Creación del hilo Servidor dedicado para el Cliente

Como veremos más adelante la recepción de la petición de comunicación en el servidor generará un canal de comunicaciones privado entre el hilo nuevo del servidor que va a atender al cliente y el propio cliente. Está creación de un canal de comunicaciones exclusivo para el cliente Está creación de un canal de comunicaciones exclusivo para el cliente implica que el servidor utilizará un nuevo puerto dedicado distinto al de escucha y es completamente transparente al cliente es completamente transparente al cliente, que simplemente se da por satisfecho sabiendo que su petición de conexión ha sido aceptada por el servidor.



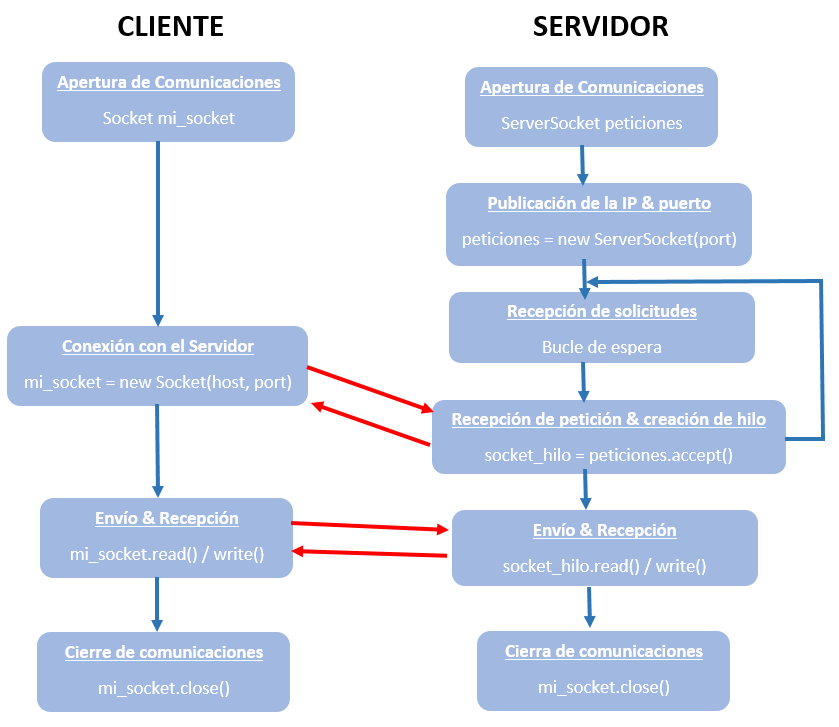
Detalle del canal de comunicaciones privado

La comunicación ya está establecida de modo que el hilo servidor y el cliente puede empezar a enviar y recibir datos del uno al otro a través de este canal de comunicaciones privado en base al protocolo de comunicaciones que tengan establecido entre ambos.



Comunicación a través del canal de comunicaciones privado

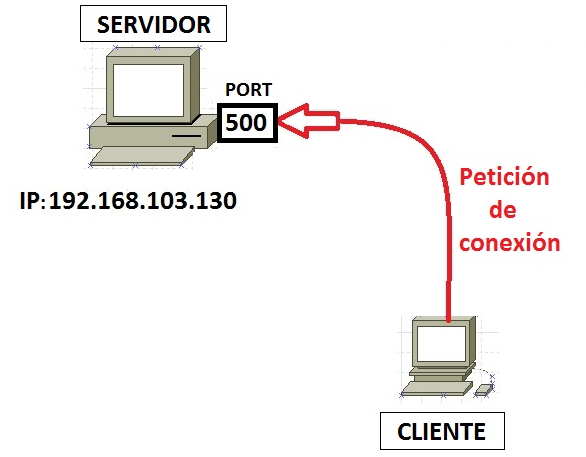
Finalmente, cuando uno de los dos dé por concluida la comunicación cerrará su extremo del canal de comunicaciones, y el sistema notificará al otro extremo el canal de comunicaciones ha sido cerrado, por lo que también se cerrará el otro extremo del canal de comunicaciones y se dará por concluida la comunicación.



Cierre del canal de comunicaciones

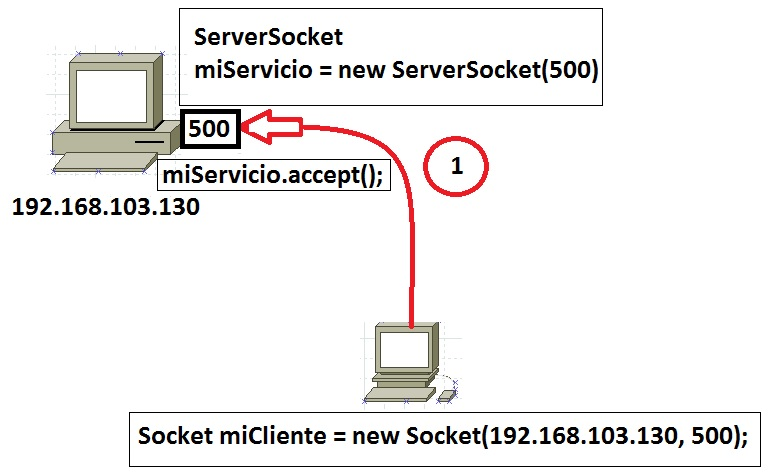
* + 1. El protocolo de conexión de los Sockets

Como ya hemos visto, el hilo servidor se ejecutará en un equipo con una dirección IP pública y un puerto público y realizará sus comunicaciones en red a través de un ServerSocket cuando esté programado en Java.



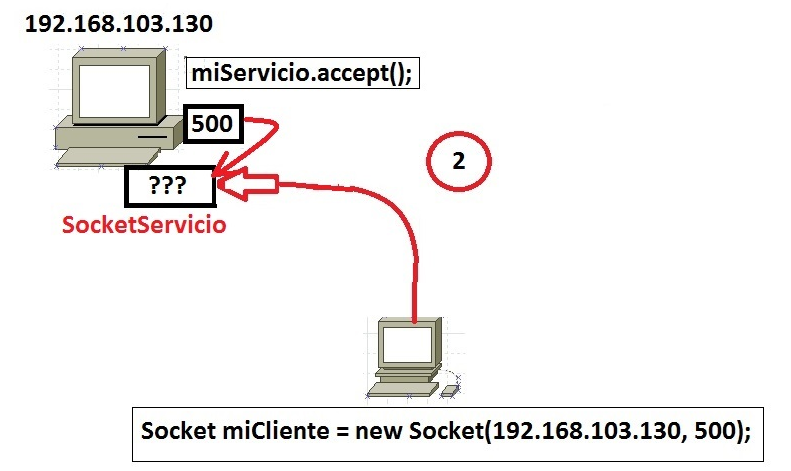
Petición de servicio

La clase ServerSocket tiene implementado el método "accept()" que es el que pone al hilo del servidor en espera escuchando a las solicitudes de comunicación de los clientes sobre el puerto especificado. El hilo cliente conoce la dirección IP y el número de puerto en el que el hilo servidor estar a la escucha de las peticiones de servicio.



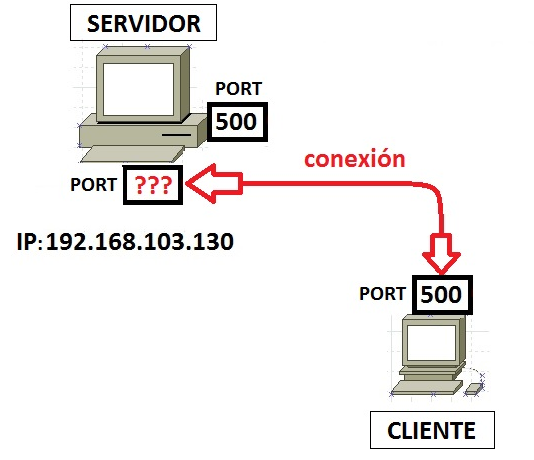
El Servidor acepta la petición de servicio

Como hemos visto en el protocolo anterior, cuando el servidor acepta la conexión se creará un nuevo socket que estará asociado a un número de puerto diferente al de escucha. El hilo servidor podría crear un nuevo hilo al cual le pasaría el socket recién creado (de servicio) para que atendiese en exclusiva al cliente.



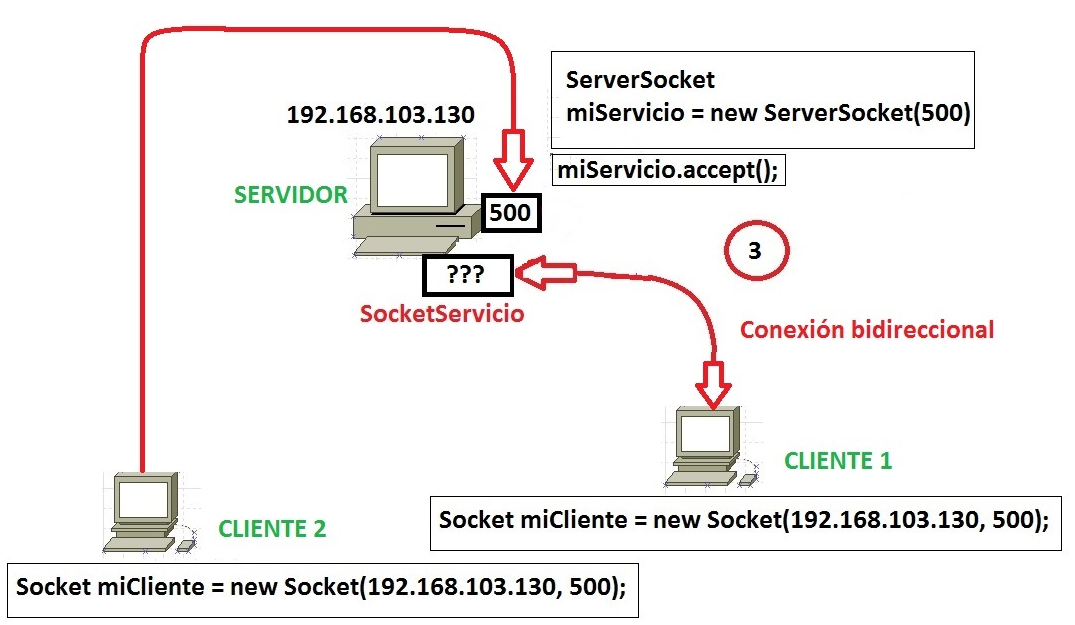
Creación del canal de comunicaciones (socket) privado

De este modo, conseguimos que el hilo servidor pueda seguir atendiendo peticiones de conexión en el puerto público, mientras que el nuevo hilo hará las veces de servidor para el cliente al cual atenderá a través del nuevo puerto del socket creado al aceptar la petición de conexión.



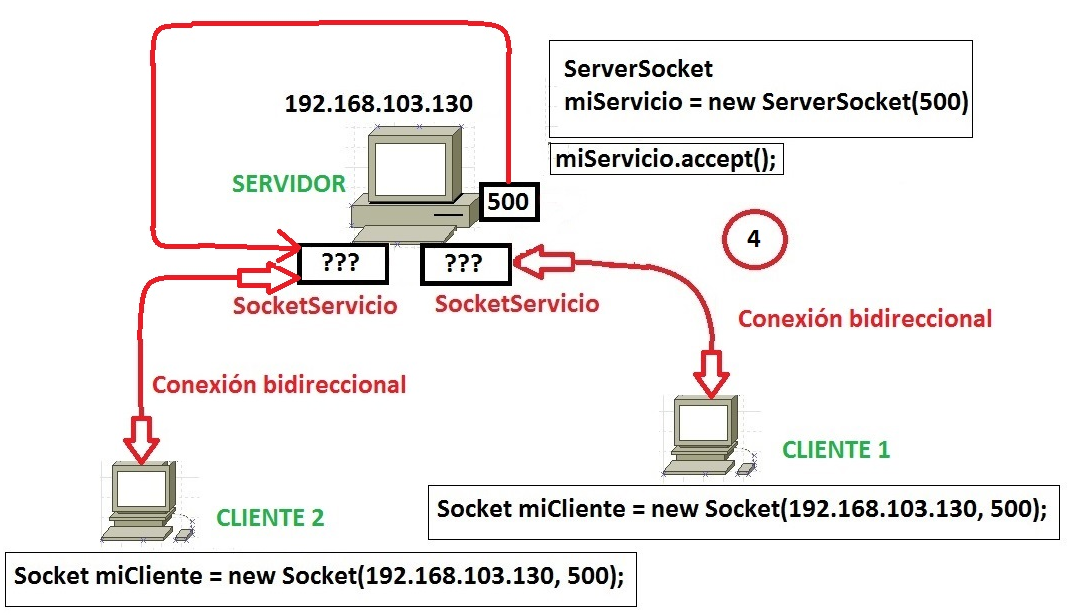
Conexión del canal de comunicaciones privado

Por su parte del cliente cuando ve que la conexión es aceptada por el servidor recibe un socket dedicado para esa conexión que estará conectado al nuevo puerto que el servidor ha abierto y podrá realizar una comunicación bidireccional con él basada en algún tipo de protocolo preestablecido entre el cliente y el servidor.



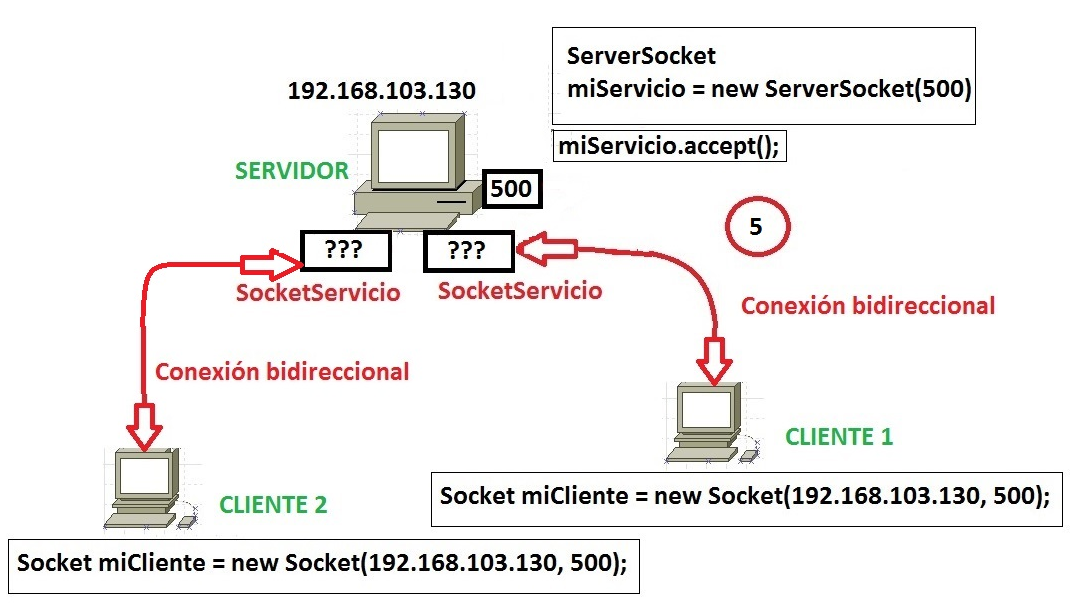
Nueva petición de servicio por un segundo cliente

Así pues, todo este cambio de puertos en la parte del servidor es completamente transparente al cliente, que comunicará con el servidor a través de su socket creado con la dirección IP y el puerto público del servidor de manera normal, escribiendo y leyendo datos del socket.



Creación de nuevo canal de comunicaciones privado

De igual manera a la explicada el sistema ya se puede extender a N clientes simultáneos, ya que el servidor desde su bucle de escucha y aceptación de clientes seguirá creando un socket de servicio y un hilo dedicado por cliente que se quiera conectar con él y liberará así el puerto establecido de escucha para seguir atendiendo a futuros clientes a los que atender.



Servicio simultáneo a dos clientes

|  |  |
| --- | --- |
|  | **Atencion En un sistema real** **Cliente/ Servidor** de este tipo, el servidor podría no poner límite al número de clientes que se atenderán de manera simultánea, de modo que la única limitación a la escala de la solución estaría en los recursos físicos del la propia máquina que aloje al servidor (CPU y memoria).  **En la realidad** y aunque son los sistemas operativos los que determinan la carga del sistema, podríamos programar de manera muy sencilla como podremos ver más adelante, el número de clientes simultáneos que estoy atendiendo simplemente teniendo un contador que aumente con el número de clientes arrancados y se decremente con el de clientes finalizados. |

* 1. Generación de Servicios en Red
     1. Java Sockets

Una vez que conocemos al detalle el protocolo completo de conexión de sockets, es hora de pasar a implementarlos. En nuestro caso lo implementaremos con lenguaje Java a través de Java sockets.

Afortunadamente para nosotros contamos con el paquete **java.net** proporcionado por el entorno de desarrollo Java, el cual contiene ya definida la clase Socket, con toda la implementación necesaria para conseguir una comunicación bidireccional entre un programa desarrollado en Java y otro programa desarrollado en cualquier otro lenguaje, incluyendo Java que utilice para sus comunicaciones el protocolo de sockets que henos definido en el punto anterior.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web documentación del paquete **java.net** donde encontrarás el Class Sumary y analiza todas las clases que contiene con el resumen de lo que cada una de esas clases realiza:  https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html |

Básicamente la clase **Socket**, a través de sus métodos que estarán implementados directamente dependiendo de la plataforma en la que estemos desarrollando, se encargará de ocultar al programador los detalles de la conexión, proporcionando simplemente una interfaz común para los programadores y que será independiente de la plataforma en la que estén desarrollando su código.

De este modo al utilizar la clase **java.net.Socket** lo que conseguimos es no tener la necesidad de programar en el código nativo de la plataforma concreta en la que estoy desarrollando. De este modo nuestros programas desarrollados en Java serán capaces de comunicarse con cualquier otro programa que utilice el mismo protocolo a través de una red de comunicaciones de manera completamente independiente a la plataforma en la que estén desarrollados tanto ellos como los programas con los que se desea conectar.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de documentación del paquete **java.net** donde encontrarás el resumen de la clase Socket y analiza todas los constructores y métodos que implementa señalando los que consideres más importantes:  https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html |

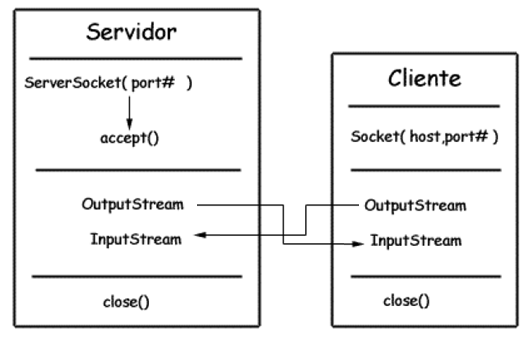
Un beneficio extra proporcionado por el paquete **java.net** es que nos permite utilizar una segunda clase de socket específicamente diseñada e implementada para dar servicio a servidores.

Esta clase denominada **ServerSocket** implementa un socket pensado para ser utilizado en servidores que es capaz de escuchar y aceptar las peticiones de conexión de clientes y que, como veremos más adelante, nos permite tener un canal de comunicación privado entre el cliente y el servidor, de modo que podemos a través de esta clase de socket atender de manera simultánea a N clientes dedicando para cada uno de ellos un canal privado de comunicación con el servidor. Estos n canales de comunicación privados para los n clientes permiten una atención simultánea a los clientes sin que sus datos se entremezclan ni aumente la complejidad del servidor a la hora de tratarlos y distinguir de qué cliente o conexión es cada dato.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de documentación del paquete **java.net** donde encontrarás el resumen de la clase ServerSocket y analiza todas los constructores y métodos que implementa señalando los que consideres más importantes:  https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html |

**Modelo de comunicaciones con Java**

El modelo de conexión a través de Java socket más sencillo que podemos representar es el siguiente:

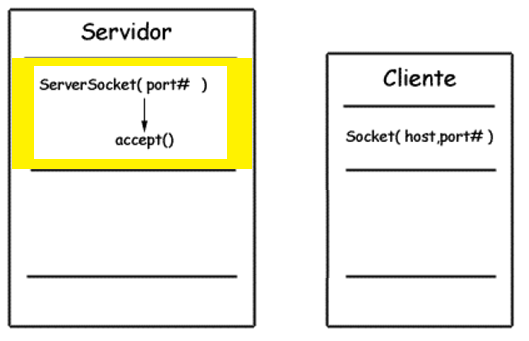
****

Modelo de comunicación con un cliente

Cómo podemos ver, este modelo simplificado, está basado en el protocolo estándar de comunicaciones a través de sockets qué hemos visto al principio de este mismo tema.

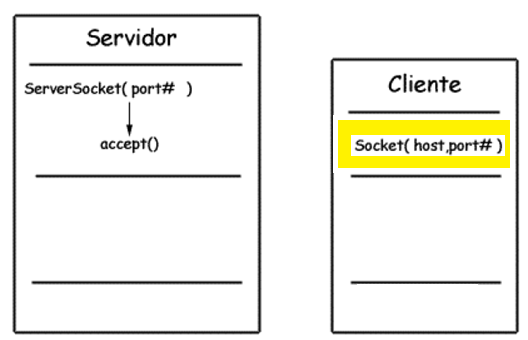
En un primer momento es necesario tener arrancado el servidor, el cual a través de la clase ServerSocket establecerá un puerto dedicado para la atención de peticiones de conexión de los clientes.

A partir de aquí el servidor quedará bloqueado en la llamada accept(), en la que esperar a la solicitud de conexión de algún cliente que deberá conocer la dirección IP del servidor y el número de puerto donde el servidor ha publicado su ServerSocket.



Creación del ServerSocket

Por otro lado, el cliente para establecer la conexión que desea con el servidor, deberá conocer la IP y el puerto en el que el servidor está esperando las peticiones de conexión, de modo que el cliente creará un objeto de la clase Socket inicializado con esa dirección IP del host destino y el número de puerto.

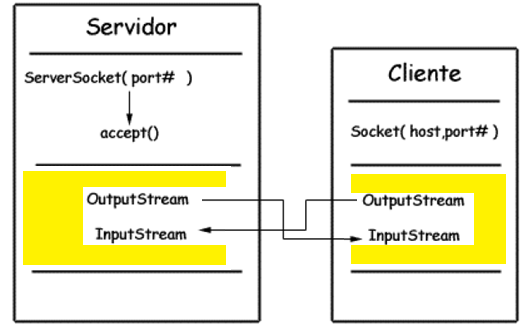


Creación del Socket

Simplemente por crear ese objeto de la clase Socket, en el constructor de la clase se realiza el intento de conexión con el servidor en la dirección IP y puerto definidos. Sí en la dirección IP y número de puerto dónde intenta conectar no hay ningún servidor a la escucha la creación del objeto socket fallará.

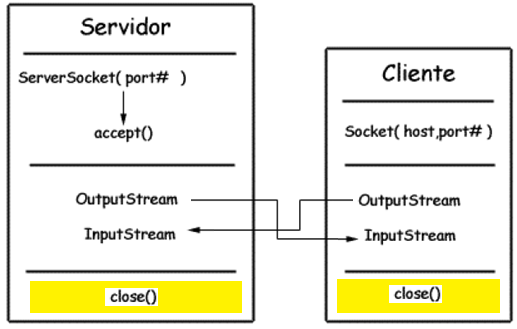
Si el servidor estaba esperando esa solicitud de conexión en el cliente se creará el objeto de la clase Socket y deberemos configurar los flujos de entrada y salida para ese objeto.

De manera simétrica, en el servidor también definiremos los flujos de entrada y salida que conectarán con el cliente. Los flujos de entrada y salida de cada uno de los sockets del servidor y del cliente estarán cruzados con el de su contrario, de manera que cada dato que escribamos en el OutputStream del cliente llegará al InputStream del servidor y viceversa.



Definición de los flujos de entrada/salidas de los Sockets

Finalmente, cuando uno de los dos decida dar por finalizada la comunicación desconectará su extremo cerrando su Socket mediante el método close. En ese momento el otro extremo al detectar el cierre del canal de comunicaciones cerrará de igual modo su extremo de la conexión.



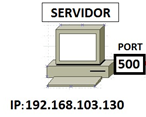
Cierre de los Sockets

* + 1. Apertura de Sockets

**Apertura del Socket de Servidor**

Tal y como hemos visto el primer paso para lograr la comunicación cliente servidor es crear el ServerSocket por parte del servidor. Para esto definimos un objeto de tipo ServerSocket al que le pasamos el número de puerto en su constructor. De esta manera queda definido eso qué es que atender a las peticiones de clientes en el número de puerto indicado.

A la hora de elegir un número de puerto válido para realizar la conexión con el servidor hay que tener en cuenta que los puertos dentro del rango de 0 a 1023 están reservados para Super usuarios o administradores de sistema. En estos puertos suelen ubicarse en los servicios estándar de sistema como FTP, HTTP, email etcétera, por lo que debemos asegurarnos de que utilizamos alguno de los puertos que podamos ver libres en nuestro sistema superiores al 1023.

Creación del ServerSocket

Es interesante ver que esta creación del ServerSocket debe ir integrada en una estructura de tipo "try catch", ya que puede producir excepciones cómo que por ejemplo el número de puerto donde quiero crear el server socket no esté disponible

**try {**

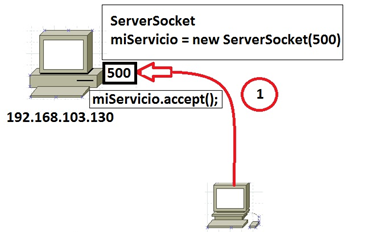
**ServerSocket skServidor = new ServerSocket(Numero\_PUERTO);**

**} catch (IOException e) {**

**System.out.println(e.getMessage());**

**}**

Una vez creado el ServerSocket será necesario aceptar las peticiones de servicio del cliente, de modo que a la vez que aceptamos la petición del cliente creemos un canal de comunicación privado para ese cliente con el servidor. Por supuesto el cliente a su vez habrá que ha creado y abierto el canal de comunicación como veremos más adelante.



El Servidor acepta la petición de servicio

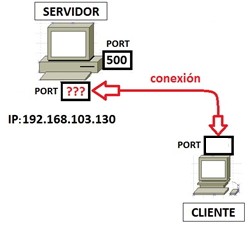
**try {(Socket skCliente = skServidor.accept()) // Crea objeto**

**} catch (IOException e) {**

**System.out.println(e.getMessage());**

**}**

El canal de comunicación privado entre el cliente y el servidor, es por supuesto un socket nos crea el propio método acepta del ServerSocket del servidor y que devuelve cuando acepta la petición de conexión con el cliente, de modo que esté socket normal que utilizaremos para comunicar con el cliente estará en un número de puerto diferente al ServerSocket en el que haciendo las peticiones, dejando libre el puerto del ServerSocket para que este siga recibiendo peticiones de conexión y no interferir con él y que pueda seguir atendiendo a nuevos clientes potenciales.

****

El Servidor crea un canal privado de comunicación con el cliente

**Apertura del Socket de Cliente**

La parte de la programación del cliente es mucho más sencilla ya que el cliente únicamente tiene que ser conocedor de la dirección IP de la máquina dónde se encuentra el servidor y el número de puerto donde quiere conectar y de este modo crear un nuevo objeto de la clase Socket pasándole a su constructor estos datos.

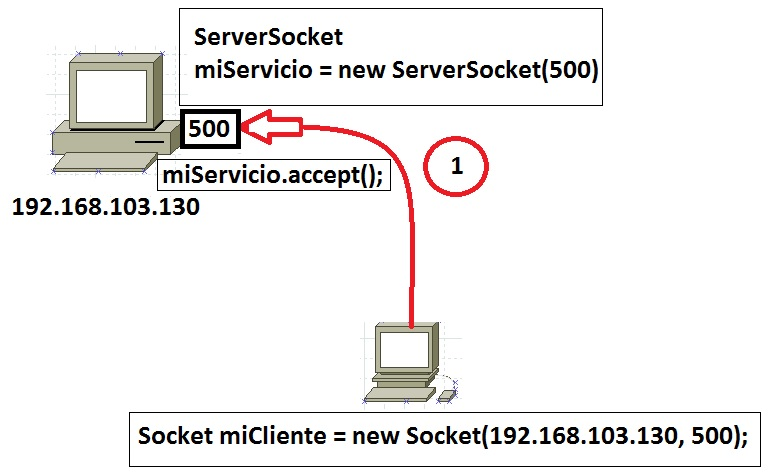
Al igual que sucedía a la hora de crear el socket del servidor (el ServerSocket), está creación del objeto puede generar excepciones ya que es posible entre otras cosas que no exista el host destino o su puerto no se encuentre abierto, por lo que es necesario englobar está creación del nuevo socket en una estructura de captura de excepciones del tipo "try catch".

**try {Socket skCliente = new Socket(HOST, PUERTO) {**

**} catch (IOException e) {**

**System.out.println(e.getMessage());**

**}**



Cliente conectado a través del canal privado con el servidor

**Resumen**

Cómo resumen podemos ver qué nuestro servidor para cada una de las solicitudes de conexión que reciba de un cliente generará un nuevo socket en el lado del servidor al que desviar a la conexión solicitada por el cliente de manera transparente a él, de modo que los clientes seguirán creciendo que están enviando sus datos al número de puerto donde estaba el server socket que atendía las peticiones de conexión cuando en realidad están enviando los a un nuevo puerto dónde estará el socket privado que se ha creado explícitamente para ese cliente en concreto que realizó la petición de conexión.

Sí el servidor acepta mi solicitud de conexión no se generará ninguna excepción y el objeto Socket se creará normalmente, pudiendo utilizar sus métodos, como enseguida veremos, para configurar los flujos de entrada y salida y poder enviar y recibir datos a través de él.

* + 1. Creación de Streams y Cierre de Sockets

**Creación de Streams de Entrada**

En la parte del servidor utilizaremos la clase DataInputStream para crear un flujo de entrada desde el que se acepten todas las peticiones que lleguen de los clientes.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de documentación del paquete **java.io** donde encontrarás el resumen de la clase DataInputStream y analiza todas los constructores y métodos que implementa señalando los que consideres más importantes:  https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html |

La clase DataInputStream a través de sus métodos nos permitirá realizar lecturas de todo tipo de datos primitivos de Java incluyendo líneas de texto de una manera estándar. Podemos ver dentro de esta clase diversos tipos de métodos para realizar multitud de lecturas, como por ejemplo read(), readChar(), readInt(), readLine(), readUTF(), etc.

Para poder trabajar con un flujo de datos de la clase DataInputStream, necesitaremos como paso previo obtener el flujo de entrada del socket y posteriormente, basándonos en él, crear el objeto de la clase DataInputStream, pasándole a su constructor el flujo que hemos obtenido del socket.

**Parte Servidor**

En el caso del servidor, el socket que utilizaremos para la conexión será el socket privado que ha devuelto el método accept() del ServerSocket.

try {Socket skCliente = skServidor.accept() // Crea objeto

{

**InputStream aux = skCliente.getInputStream();**

**DataInputStream flujo\_entrada = new DataInputStream(aux);**

}

} catch (IOException e) {

System.out.println(e.getMessage());

}

**Parte Cliente**

En el lado del cliente obtendremos el flujo de entrada directamente del socket definido por él, utilizando como en el caso del servidor un flujo de datos auxiliar intermedio qué será el que utilizamos para crear el objeto de la clase DataInputStream.

try {

try (Socket skCliente = new Socket(HOST, PUERTO)) {

**InputStream aux = skCliente.getInputStream();**

**DataInputStream flujo = new DataInputStream(aux);**

}

} catch (IOException e) {

System.out.println(e.getMessage());

}

**Creación de Streams de Salida**

A la hora de crear los flujos de salida para los sockets por los que enviaremos información al otro extremo de la conexión podemos utilizar dos clases diferentes.

La primera que es la opción que utilizaremos en este curso es la clase DataOutputStream, ya que es la clase completamente simétrica a la clase DataInputStream que hemos visto en el punto anterior de creación de streams de entrada.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de documentación del paquete **java.io** donde encontrarás el resumen de la clase DataOutputStream y observa la similitud con respecto a la clase DataInputStream:  https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html |

Otra posibilidad que podemos contemplar para la creación de un flujo de salida de datos, es utilizar la clase PrintStream, qué contiene métodos para la representación textual de todos los datos primitivos de Java punto sus métodos principales son métodos write y println.

|  |  |
| --- | --- |
| Actividad | Entra en la siguiente dirección web de documentación del paquete **java.io** donde encontrarás el resumen de la clase PrinttStream y analiza todas los constructores y métodos que implementa señalando los que consideres más importantes:  https://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html |

En nuestro caso y como hemos comentado anteriormente, utilizaremos la clase DataOutputStream, que nos permitirá escribir a través de sus métodos en el flujo de datos de salida del socket cualquiera de los tipos primitivos de Java. Los métodos más utilizados de esta clase suelen ser el método writeBytes() y el método writeUTF() que analizaremos más adelante.

**Parte Servidor**

try {Socket skCliente = skServidor.accept() // Crea objeto

{

**OutputStream aux = skCliente.getOutputStream();**

**DataOutputStream flujo\_entrada = new DataOutputStream(aux);**

}

} catch (IOException e) {

System.out.println(e.getMessage());

}

**Parte Cliente**

try {

try (Socket skCliente = new Socket(HOST, PUERTO)) {

**OutputStream aux = skCliente.getOutputStream();**

**DataOutputStream flujo = new DataOutputStream(aux);**

}

} catch (IOException e) {

System.out.println(e.getMessage());

}

**Cierre de Sockets**

A la hora de cerrar nuestros sockets de entrada y salida, utilizaremos el método close de nuestro sockets. Es importante tener en cuenta la importancia del orden de los cierres ya que primero debemos cerrar los chulis relacionados con el socket y acto seguido el propio socket. De este modo evitaremos posibles errores de lectura o escritura sobre descriptores que ya se encuentran cerrados.

En la parte del cliente:

**try {**

**flujo\_salida.close();**

**flujo\_entrada.close();**

**skCliente.close();**

**} catch( IOException e ) {**

**System.out.println( e );**

**}**

Y en la parte del servidor:

**try {**

**flujo\_salida.close();**

**flujo\_entrada.close();**

**sk\_Cliente.close();**

**skServidor.close();**

**} catch( IOException e ) {**

**System.out.println( e );**

**}**

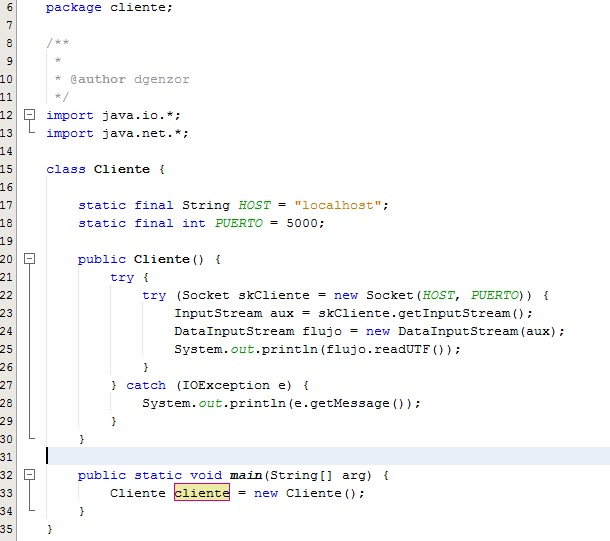
* 1. Ejemplo de aplicación

Realizaremos ahora un ejemplo de aplicación que engloba los conceptos que hemos visto en este tema. En concreto para comprender el funcionamiento de los sockets realizaremos un programa servidor y otro programa cliente que ejecutaremos varias veces para simular que hay varios clientes que quieren contactar con él.

El servidor mantendrá una pequeña conversación con el cliente de manera que el servidor saludará al Cliente con el número de cliente que le corresponde por orden de llegada al servidor.

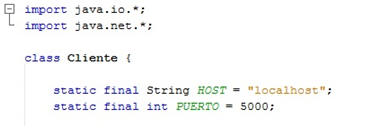
* + 1. Cliente

Comencemos por el código del cliente ya que es más sencillo que el del servidor:



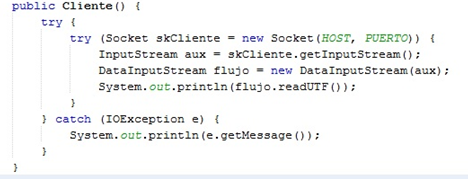
Código completo del Cliente

Vemos cómo en el código del cliente tras importar los paquetes java.io y java.net, dónde se encuentran las definiciones de las clases que vamos a utilizar, declaramos la clase cliente que únicamente contiene a su constructor y define un par de atributos que son la dirección IP del host que en nuestro caso vamos a utilizar el local host o dirección IP 127.0.0.1, y el número de puerto donde queremos conectar con el servidor que en nuestro caso utilizaremos a el 5000.



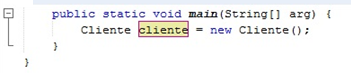
Definición atributos de la Clase Cliente

En el constructor del cliente simplemente creamos el objeto de tipo socket a cuyo constructor pasaremos la dirección IP del host y el número de puerto. Acto seguido damos formato al flujo de entrada de datos del socket para finalmente leer a través del flujo asociado al socket DataInputStream y ayudados del método readUTF, una cadena que directamente imprimiremos por pantalla, y que será el saludo que nos devuelve el servidor cuando conectamos con él.



Definición del constructor de la Clase Cliente

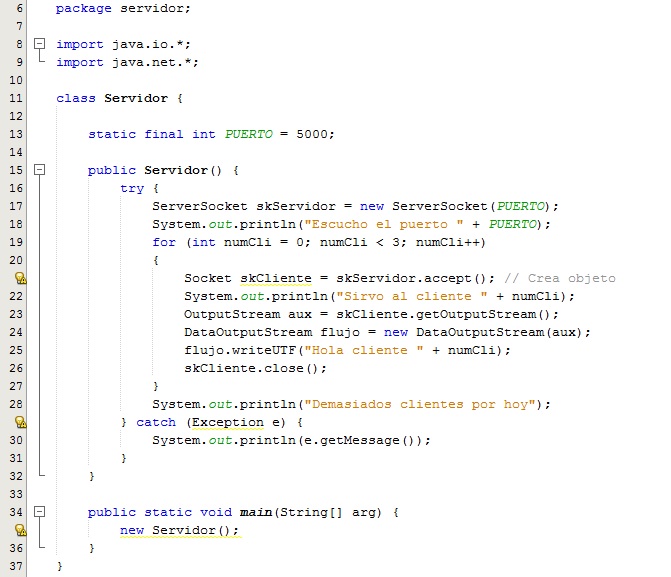
Es interesante ver como el programa de cliente tiene su propio main, dónde únicamente para poner en marcha el funcionamiento del cliente debemos definir un objeto de la clase cliente, es decir nuestro mail en realidad no contiene nada de código salvo esa declaración qué es la que pondrá a funcionar el constructor de la clase cliente donde se ejecutará todo el protocolo de comunicación con el socket del servidor.



Definición del main del Cliente

* + 1. Servidor

El código completo del programa que implementa al Servidor es el siguiente:



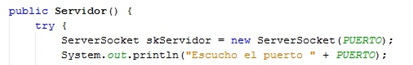
Código completo del Servidor

El programa del servidor, después de importar los paquetes java.io y java.net, declara la clase servidor, dónde se declara el atributo puerto con el número 5000 que será el puerto donde realizará las peticiones al cliente. El servidor, al igual que el cliente, realiza todas las labores de apertura de sockets y conexión con el cliente en su método constructor.



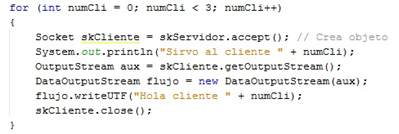
Definición de atributos del Servidor

Vemos como el servidor lo primero que realiza es crear el objeto de la clase ServerSocket pasándole a su constructor el número de puerto donde queremos ponerlo a escuchar las peticiones de conexión de los clientes.



Definición del ServerSocket

Una vez creado el ServerSocket realiza un pequeño bucle para atender a tres clientes, y una vez atendido esos tres clientes escribirá por pantalla que ya ha tendido a demasiados clientes por hoy y finalizará su ejecución.



Definición del main del Cliente

Dentro del bucle de atención a los clientes se realiza la escucha de solicitudes de conexión de los clientes a través del método accept() del ServerSocket. Hay que tener en cuenta que este método es bloqueante, de modo que cuando el servidor lo invoca sin ningún cliente ha realizado una petición de conexión el servidor se quedará esperando bloqueado en él hasta que un cliente realice su petición de conexión.



Aceptación de la petición de conexión del Cliente

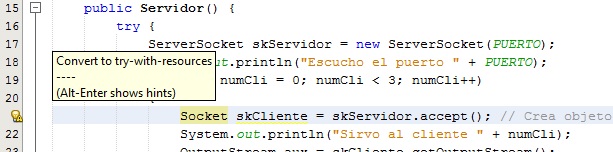
Una vez atendida la petición de conexión como más visto se crea el socket cliente que se será el canal privado de comunicación con el cliente, y es precisamente en ese socket donde formatear hemos los flujos de datos para crear el objeto de la clase DataOutputStream a través del cual escribiremos el mensaje de saludo para que lo reciba el cliente.

Una vez realizado esto cerraremos nuestro extremo del socket y volvemos a iniciar el bucle para atender al siguiente cliente.



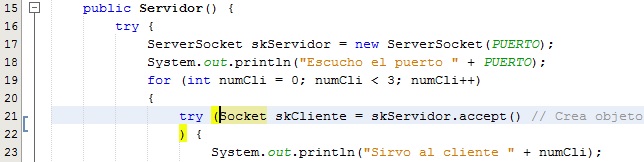
Formateo de los flujos de salida del Socket

Vemos que en el código que hemos realizado aparecen varios warnings que podremos resolver incluyendo las llamadas a los métodos que pueden generar excepciones en estructuras de captura de excepción de tipo "try catch"



Definición del main del Cliente

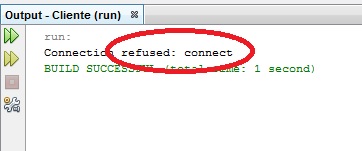
Sí estamos utilizando el entorno de desarrollo NetBeans podemos simplemente ponernos encima del warning y darle a la tecla Alt + Enter y nos generará automáticamente el código necesario para eliminar el warning.



Definición del main del Cliente

Esta estructura de atención a clientes es una estructura secuencial de manera que nuestro servidor atenderá a los tres clientes uno detrás de otro y no de forma paralela.

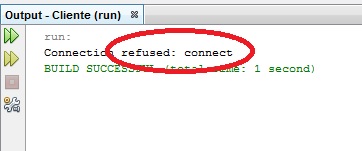
Está claro pues, que nuestro ejemplo de aplicación no es un único programa, sino que son dos. Por un lado, el programa cliente y por otro lado el programa servidor que veremos a continuación. Para probar nuestro ejemplo de aplicación de veremos a ejecutar primero el servidor y después el cliente, ya que en caso contrario veremos que el cliente al intentar crear el socket falla al no haber nadie escuchando su petición de conexión.



Definición del main del Cliente

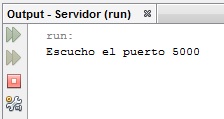
* + 1. Ejecución

A la hora de ejecutar el ejemplo como hemos dicho es importante lanzar primero el servidor para que esté atento a las peticiones de conexión antes de que los clientes las lancen, en caso contrario cuando ejecutamos un cliente nos encontraremos con este error en tiempo de ejecución ya que no hay nadie al otro lado del socket que está intentando crear



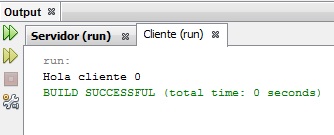
Ejecución del Cliente antes que el Servidor

Cuando ejecuto primeramente al servidor veré por pantalla un mensaje de información que me indica el puerto donde he puesto a la escucha al servidor a la espera de que los clientes realizan sus peticiones de conexión



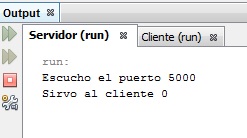
Ejecución del Servidor

Es ahora el momento de ejecutar al primero de nuestros clientes, al hacerlo directamente creará el socket cliente que conectará con el servidor despertando al servidor del bloqueo de espera en el que se encontraba en su método accept(), de modo que el servidor crea el canal de conexión privado y a través de él nos devuelve el mensaje que vemos por la pantalla del cliente



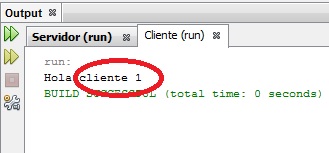
Ejecución del Cliente

A su vez si vamos a la salida de datos del servidor veremos que ya nos ha avisado por pantalla que está sirviendo al primer cliente, en nuestro caso el cliente 0, y habrá vuelto a entrar al bucle de atención a clientes por lo que de nuevo se encontrará bloqueado en la llamada al método accept() de su ServerSocket



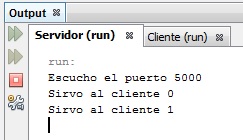
Atención del Servidor al primer Cliente

Es el momento ahora de lanzar al segundo cliente qué tendrá el mismo comportamiento que el cliente 1, solo que ahora recibirá el mensaje te saludo con la numeración del cliente aumentada, es decir el servidor le está diciendo hola cliente uno ya que es el segundo cliente al que atiende.



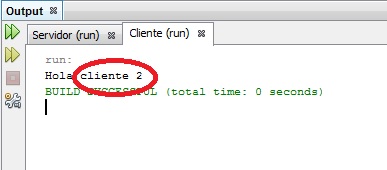
Ejecución del Cliente 2

Si volvemos a mirar la pantalla de salida de datos del servidor podremos ver que ya ha aparecido el mensaje de sirvo al cliente 1 y nuevamente habrá entrado en el bucle de escucha para recibir a un nuevo cliente y estará de bueno bloqueado en el método accept () des ServerSocket.



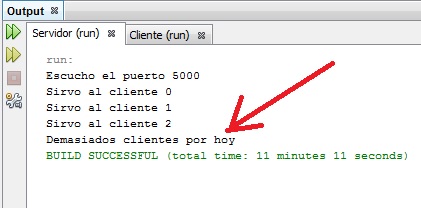
Atención del Servidor al segundo Cliente

Finalmente ejecutamos el tercer cliente y vemos que funciona exactamente igual que sus compañeros anteriores, recibiendo en este caso el mensaje de saludo "hola cliente 2"



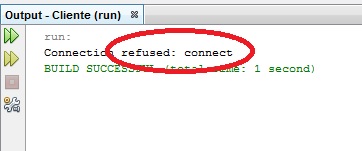
Ejecución del Cliente 3

Finalmente, el servidor ha atendido a los tres clientes que quería atender y si miramos su pantalla de salida de datos habrá indicado que ha servido al cliente dos y habrá puesto el mensaje demasiados clientes por hoy terminando además su ejecución.



Terminación de Ejecución del Servidor

A partir de ahí ya no existe servidor en el puerto 5000 escuchando las peticiones de los clientes, por lo que cualquier nuevo cliente que intentemos ejecutar a partir de este momento fallará a la hora de intentar conectar el socket y dará el error en tiempo de ejecución conexión rehusada por el servidor



Ejecución del Cliente sin Servidor

En nuestras pruebas hemos utilizado como dirección IP del host la dirección local, pero podríamos haber utilizado cualquier IP pública o de nuestra red local para hacer la prueba. En este sentido es interesante ejecutar el servidor en una máquina y el cliente en otra y probar la conexión en una pequeña red local.

|  |  |
| --- | --- |
| Actividad | Modifica el código del cliente para ponerle una dirección IP real en la que esté alojado el servidor y que no sea la local.  Ejecuta el servidor en la dirección IP real e intenta conectar con él desde otra máquina en la que arranques el cliente.  **NOTA**: Para la realización de esta actividad necesitarías tener al menos dos equipos dentro de una red local o con una dirección IP pública al menos uno de ellos. |

* + 1. Resumen

Para comunicar dos programas cualesquiera que estén en la misma o en diferentes máquinas necesitaremos un protocolo preestablecido de comunicación.

En nuestro caso utilizaremos sockets en su implementación concreta de Java.

A través del paquete java.net somos capaces de definir objetos de la clase ServerSocket y objetos de la clase Socket. Los objetos de la clase ServerSocket son una especialización del objeto Socket específicamente diseñados para servidores que permiten aceptar peticiones de conexión de los clientes en un puerto determinado. A la vez que se acepta la petición de conexión del cliente se abrirá un canal de conexión privado entre el cliente y el servidor que permitirá mantener las comunicaciones privadas con ese cliente. Podemos paralelizar este mecanismo para dar servicio simultáneo a N clientes.

Al final del tema podemos realizar un ejercicio guiado que ilustra el caso de aplicación de una estructura cliente-servidor sencilla en la que se mantiene una pequeña conversación entre el servidor y los clientes. El servidor en este ejemplo es un servidor multi cliente secuencial, de modo que el servidor atiende a más de un cliente pero no de manera simultánea sino encolando los y atendiendo uno detrás de otro al número de clientes que él determine.