UNIDAD 3: PROGRAMACIÓN DE COMUNICACIONES EN RED

Módulo Profesional:
Programación de Servicios y Procesos

Índice

RESUMEN INTRODUCTORIO

En esta unidad introduciremos y desarrollaremos los conceptos necesarios para la programación en red y la transmisión de información entre aplicaciones remotas.

Repasaremos conceptos sobre las redes de ordenadores, la comunicación en esas redes y, por supuesto, la importancia de conocer los estándares y arquitecturas que definen estos estándares para poder realizar estas comunicaciones.

En ese punto es interesante e importante unir los conocimientos teóricos a los conocimientos más prácticos con el lenguaje de programación Java y las diferentes librerías y clases que nos proporciona en el paquete java.net.

Por otro lado, estudiaremos los sockets, las librerías mínimas para establecer una comunicación entre dos aplicaciones, las librerías relacionadas para el envío y recepción de información o las de transmisiones orientadas o no a la conexión.

Por último, realizaremos una aproximación a la programación de clientes y servidores, uniéndola a los conceptos vistos sobre la programación multitarea como forma de hacer más eficiente la interacción entre procesos.

INTRODUCCIÓN

Actualmente es imposible concebir ningún dispositivo de procesamiento de la información (sea ordenador, móvil, libro electrónico…) sin capacidad para comunicarse. La gran mayoría de aplicaciones actuales necesitan una conexión para instalarse o para actualizarse y una buena parte de ellas la necesitan también para poderse ejecutar con normalidad.

A menudo las aplicaciones trabajan con recursos en la nube o extraen los datos de un SGBD situado en un servidor remoto. Gracias a la conectividad de nuestros dispositivos podemos ver películas sin tener que almacenarlas en un disco local, podemos sincronizar los relojes con la hora oficial, comprar entradas para ir al teatro sin movernos de casa o hacer una partida en línea de nuestro videojuego preferido. Toda esta capacidad de comunicación solo es posible gracias a las redes.

Y los lenguajes de programación nos proporcionan los mecanismos para poder realizar esta comunicación, cada uno con sus particularidades, pero en definitiva con los mismos objetivos ya que se deben cumplir los

estándares. En concreto, Java aprovecha su orientación a la programación orientada a objetos para justamente ofrecer un paquete completo de soluciones dentro de la capa de aplicación principalmente.

CASO INTRODUCTORIO

Acabamos de ser contratados por una empresa de alimentación que va a cambiar su modelo de negocio poniendo tiendas físicas de venta en pequeñas localidades.

Esta empresa necesita cambiar su aplicación de gestión de alimentos de entrada y salida que estaba únicamente en el almacén y que servía a otros clientes.

En esta ocasión, necesitan cambiar el modelo a un modelo en red en el que las diferentes tiendas y el almacén se comuniquen, tanto con la base de datos como entre ellas para realizar diferentes procesos.

¿Cómo planteamos nuestras comunicaciones? ¿Qué arquitectura elegimos para las comunicaciones? ¿Qué librerías y alteraciones necesitamos realizar para programar este cambio?

Al finalizar la unidad tendremos los recursos y habilidades para responder a esas preguntas.

1. PROTOCOLOS DE COMUNICACIONES

El primer paso es definir la red de trabajo y conocer cómo serán los procesos y las comunicaciones entre las aplicaciones. Debemos decidir qué tipos de comunicaciones necesitamos para cada caso.

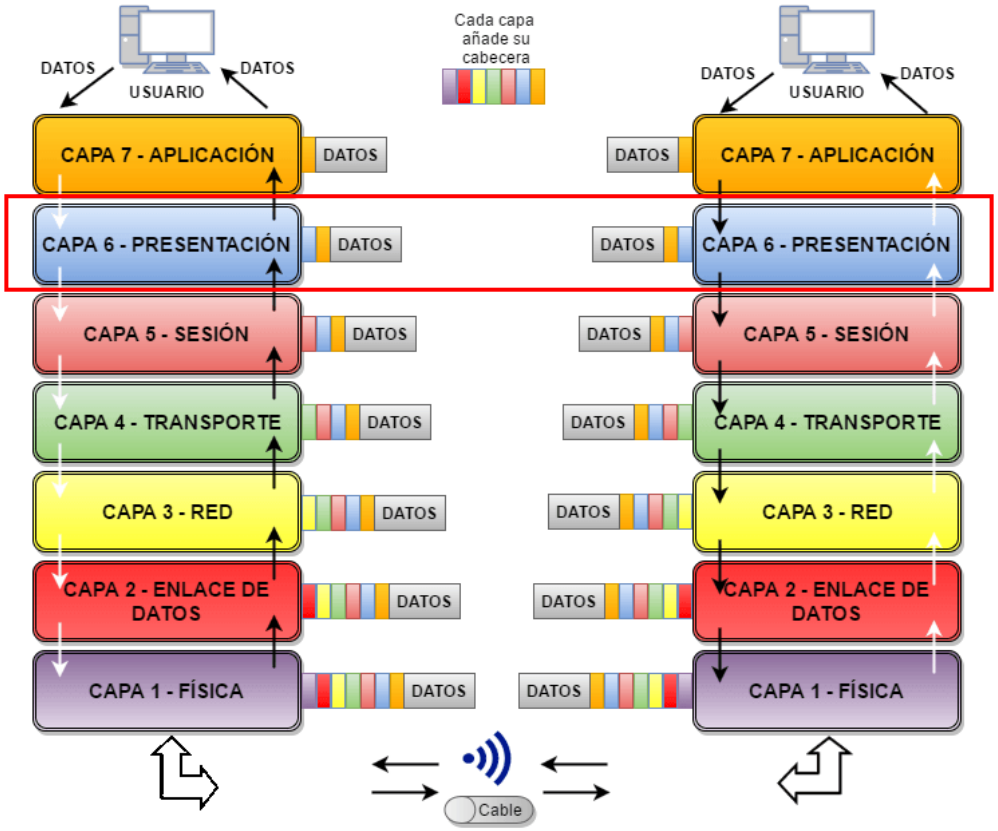
Antes de comenzar a programar es importante responder a las siguientes preguntas, ya que son esenciales para plantear los mecanismos de comunicación correctos: ¿Cómo realizaremos las comunicaciones si queremos programar un sistema de avisos? ¿Cuál será el mejor diseño en el caso de un equipo que haga de maestro esperando información del resto?

Las redes igual que cualquier comunicación entre dos o más personas necesita establecer un orden para que todos tengan la posibilidad de comunicarse correctamente. Podríamos establecer el símil entre los códigos de circulación de las carreteras para comenzar a hablar de estándares y comunicaciones.

Un protocolo de comunicación es un conjunto de normas que usan los ordenadores para gestionar la comunicación en el intercambio de información. Dos ordenadores con distinta configuración, pero el mismo protocolo de comunicación, se podrán comunicar sin problema.

Los procesos de comunicación dentro de una red son bastante complejos, por los elementos que intervienen y también por las situaciones en las que se encuentran cada uno de esos elementos en un momento concreto. Por ese motivo, para poder describir una red se utilizan **modelos de redes**, dentro de los cuales se intenta describir el modelo como un conjunto de capas o de partes, que es lo que se denomina el **modelo de capas** y en el que podemos destacar las siguientes características:

- Cada capa se centra y está especializada en un aspecto concreto de la comunicación y es responsable de un segmento del proceso total.
- Cada capa interactúa únicamente con las capas adyacentes, es decir, con la anterior y posterior si existiese.

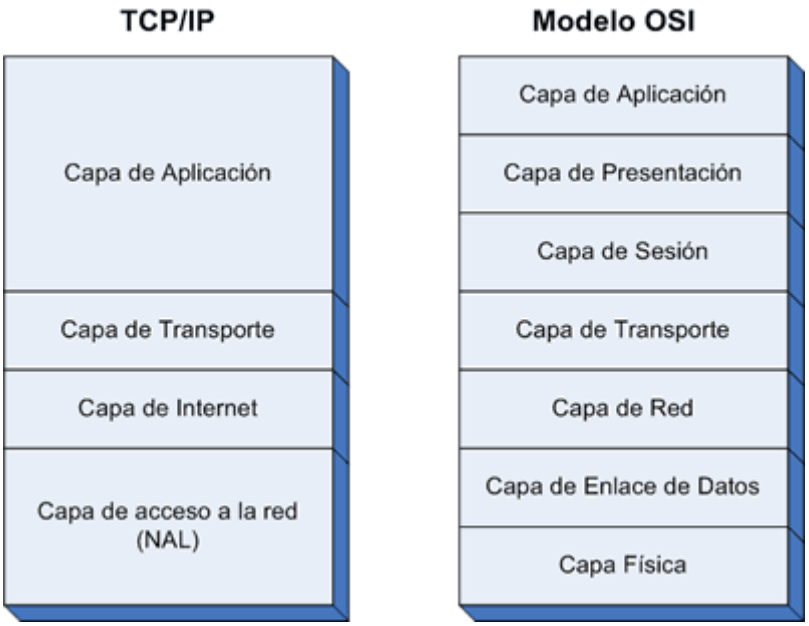


Ejemplo de modelo de pares.
Fuente: http://gigainside.com/wp-content/uploads/2017/07/modelo-osi.png

1.1 Protocolo TCP/IP

Hoy en día, el protocolo más popular es TCP/IP cuyas siglas vienen de *Transfer Control Protocol / Internet Protocol*. Este protocolo se basa en un modelo por capas que permite que se pueda utilizar en cualquier equipo independientemente del sistema operativo utilizado. El término capa se refiere al hecho de que la información viaja por la red atravesando diferentes niveles de protocolos, ya que cada capa procesa sucesivamente la información y la envía a la capa siguiente.

Según (Stallings: 2004), "la arquitectura de protocolos TCP/IP es resultado de la investigación y desarrollo llevados a cabo en la red experimental de conmutación de paquetes ARPANET, financiada por la Agencia de Proyectos de Investigación Avanzada para la Defensa (DARPA, Defense Advanced Research Projects Agency), y se denomina globalmente como la familia de protocolos TCP/IP. Esta familia consiste en una extensa colección de protocolos que se han especificado como estándares de Internet por parte de IAB (Internet Architecture Board)".



Comparativa modelo OSI y TCP/IP.
Fuente: https://www.textoscientificos.com/imagenes/redes/tcp-ip-osi.gif

BIBLIOGRAFÍA RECOMENDADA

Stallings, W. (2004). *Comunicaciones y Redes de Computadores.* (7.ª edición). Madrid: Pearson Educación.

Este libro es imprescindible, ya que en su capítulo "Arquitectura de redes" explica perfectamente el modelo de referencia OSI.

Las capas de este modelo son las siguientes:

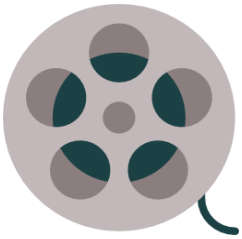
- **Capa de acceso a la red**. Se encarga de ofrecer la capacidad de acceder a cualquier red física. Contiene las especificaciones necesarias para la transmisión de datos por una red física.

- **Capa de Internet**. Se encarga de permitir que los nodos incluyan paquetes en cualquier red y viajen de forma independiente a su destino. Estos paquetes pueden llegar incluso en un orden distinto a como se enviaron. Esta capa define un formato de paquetes y un protocolo oficial de direccionamiento denominado IP.

- **Capa de transporte**. En esta capa se encuentran dos protocolos:
  - TCP (protocolo de control de la transmisión).
  - UDP (protocolo de datagrama de usuario).

La principal diferencia entre ambos es que el primero es orientado a conexión, mientras que el segundo es un protocolo sin conexión y no confiable por lo que su uso se limita a aplicaciones que no necesitan control de flujo.

- **Capa de aplicación**. Contiene los protocolos de más alto nivel como son SMTP, FTP, etc. El software de esta capa se comunica mediante los protocolos de la capa de transporte TCP o UDP.



VIDEO DE INTERÉS

En el siguiente vídeo es imprescindible para comprender totalmente el modelo de referencia TCP/IP, al igual que ocurría con el modelo OSI:

https://www.youtube.com/watch?v=JQDCL17sARA

1.1.1 Capa de transporte. Protocolos UDP y TCP

Se trata de la capa situada inmediatamente por debajo de las aplicaciones. Durante la recepción de los datos, necesitará reconocer cuál es la aplicación destinataria de la información puesto que, por supuesto, podría darse el caso de tener varias aplicaciones independientes esperando datos. Por convención, la capa de transporte usará un número para identificar las diversas aplicaciones que se encuentren escuchando. Habitualmente se conoce este número identificativo con el nombre de puerto. A cada envío, habrá que incluir siempre el valor del puerto como dato extra de la capa. Así el receptor podrá averiguar cuál es la aplicación destinataria.

Los principales protocolos usados en este nivel son UDP y TCP:

- **Protocolo UDP**: Además de los puertos origen y destino, el resto de datos extras de este nivel vendrán determinadas por el protocolo utilizado. UDP es más sencillo que TCP porque únicamente comprueba la coherencia de los datos recibidos. Utiliza la longitud de los datos y un valor de comprobación denominado checksum para realizar la verificación de la coherencia. El valor checksum se obtiene a través de un algoritmo de suma que lleva el mismo nombre.

- **Protocolo TCP**: Las aplicaciones que necesiten fiabilidad en las transmisiones tendrán que usar el protocolo TCP. Se trata de un protocolo de gran complejidad con el cual se garantiza que todos los

datos enviados desde el origen llegan al destino íntegramente y en el mismo orden en que han salido del emisor.

Para asegurar la fiabilidad de la transmisión, el protocolo TCP mantiene un diálogo permanente entre emisor y receptor en el cual ambos dispositivos se informan de aquello que van enviando y recibiendo. El diálogo se inicia con una petición de conexión y se mantiene abierto (ambos dispositivos se escuchan mutuamente) hasta que uno de los dos envíe una señal para finalizar la conexión.

COMPRUEBA LO QUE SABES

Acabamos de estudiar las diferencias entre las comunicaciones orientadas a la conexión y las no orientadas a la conexión. ¿Serías capaz de poner un ejemplo sobre cada una de ellas?

Razona el ejemplo y coméntalo en el foro.

1.1.2 Direccionamiento IP

Las direcciones IP identifican las conexiones directas de un dispositivo de manera única. Un dispositivo puede disponer de una o varias conexiones a través de las cuales enviar y recibir los mensajes. Al tratarse de un valor único en toda la red, se utilizan también para identificar los dispositivos a pesar de que en realidad un dispositivo tendrá siempre tantos identificadores como conexiones tenga.

Estas son en el fondo una secuencia de bits sobre la cual se pueden hacer operaciones binarias. Se trata de una característica muy útil porque nos permite definir rangos de direcciones y determinar de una forma muy rápida si una dirección cualquiera se encuentra dentro del rango. Para definir el rango utilizaremos una máscara y una dirección de red base.

Direccionamiento IP.

Fuente: https://sites.google.com/a/albertoruiz.es/albertoruiz/mis-articulos/asignando-direcciones-ip-a-la-red-del-centro

1.2 Comunicación entre aplicaciones

La comunicación es un proceso complejo en el cual se produce una transferencia de información entre agentes independientes (aplicaciones). Es importante subrayar el término independientes porque esto implica que cada agente dispone de su propio sistema de información que no comparte de forma directa ni simple con otro agente.

Para que la comunicación sea posible es necesario que los agentes compartan una misma manera de representar la información a pesar de que esta no tiene por qué coincidir con la representación interna que cada agente mantiene. Hace falta también que los agentes tengan órganos o dispositivos funcionales propios que les permitan, por un lado, generar representaciones comunes de una parte de la información que mantienen y de la otra, interiorizar las representaciones elaboradas por otros agentes. La parte de información que se transmite se denomina mensaje.

1.2.1 Modelos cliente/servidor. Roles

Las aplicaciones utilizan el modelo cliente/servidor para comunicarse principalmente. Se trata de un sistema distribuido donde el software está dividido en dos partes las cuales se pueden ejecutar en el mismo o diferente sistema:

- Servidor: Aplicación que ofrece servicios a clientes. Cabe destacar que un mismo servidor puede dar servicio a varios clientes y que en un mismo sistema puede haber varios servidores.

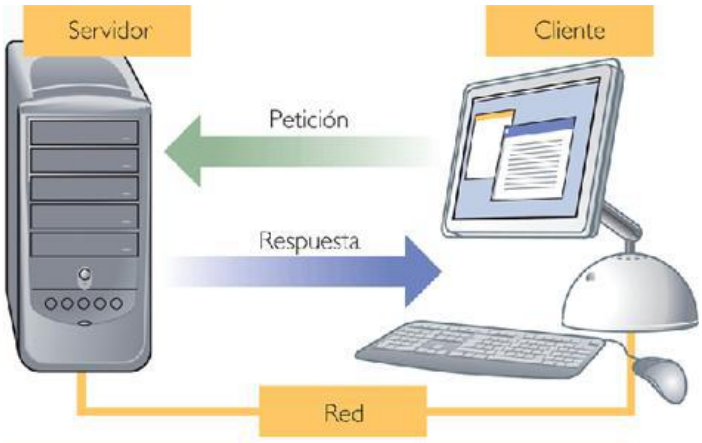
- Cliente: Aplicación que solicita servicios al servidor. Esta parte es la que interactúa con el usuario de la aplicación. Cabe destacar que un mismo cliente puede comunicarse simultáneamente con varios servidores.

Gracias a este modelo se consigue una separación clara de responsabilidades de manera que en cada una de las partes se realizan determinadas funciones.

Aunque el flujo de la comunicación puede ir en ambos sentidos, el funcionamiento más habitual para el establecimiento de la comunicación entre cliente y servidor es:

1. Un usuario invoca la parte cliente de una aplicación.
2. El cliente construye la solicitud del servicio demandado por el usuario y la envía al servidor de la aplicación.

3. El servidor recibe la solicitud, realiza el servicio solicitado y devuelve los resultados.



Comunicación cliente/servidor.

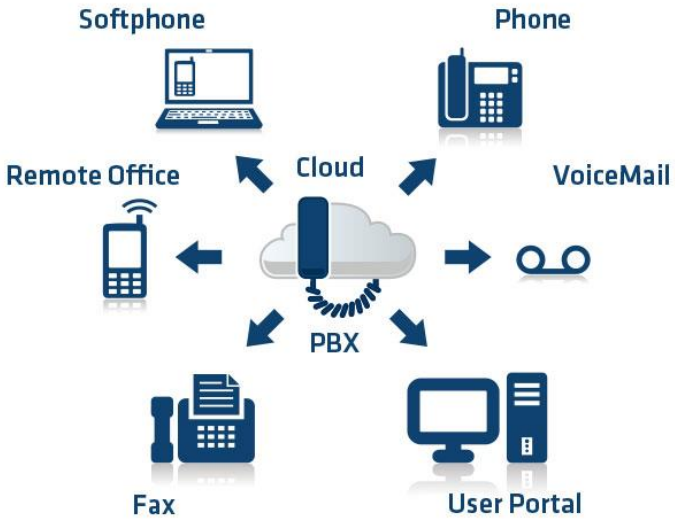
Al cliente se le suele denominar parte activa y al servidor parte pasiva, puesto que está esperando a que los clientes se comuniquen con él. Entre las principales utilidades del uso de aplicaciones cliente/servidor cabría destacar:

- Facilidad de mantenimiento debido a que todo el mantenimiento se realiza en el servidor.
- Pueden ser desarrolladas utilizando distintos lenguajes de programación.
- Ligereza debido a que la carga de trabajo y consumo de recursos se realiza en el lado del servidor.

1.2.2 p2p (peer-to-peer)

En un sistema peer to peer dos hosts están conectados entre ellos y solamente entre ellos. Durante esta conexión, que no tiene por qué ser física, el papel de servidor y de cliente pueden ir turnándoselo o incluso no existir.

Un servicio caramente peer to peer es el de telefonía IP, donde una vez que se establece la comunicación, la misma siempre se mantiene por un canal uno a uno, aunque el medio sea compartido.

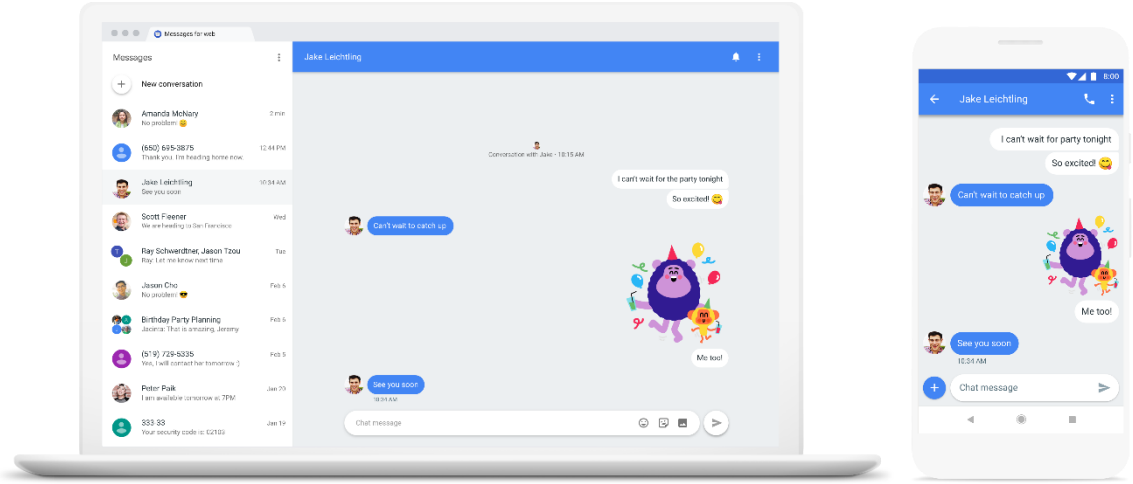


Ejemplo de telefonía IP.
Fuente: http://www.uccommunications.com.au/wp-content/uploads/2016/03/comm.jpg

1.2.3 Modelos híbridos

Los modelos híbridos los encontramos cuando aparecen combinaciones de los dos modelos anteriores, por un lado, un modelo cliente/servidor y, por otro, p2p. Este tipo de modelo cada vez es más habitual, ya que algunas de las comunicaciones necesitan de una comunicación asíncrona cliente/servidor y otras comunicaciones necesitan de la sincronía y el establecimiento de un canal p2p para la comunicación.

Un ejemplo claro es el de una programación contra un servidor web, donde claramente el modelo es cliente/servidor, pero que en un momento dado se necesita abrir un chat o el usuario se une a un chat con otros usuarios.



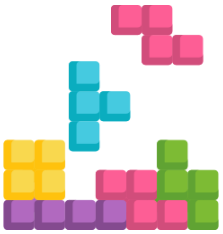
Comunicación por chat.
Fuente: https://www.antevenio.com/blog/2019/07/chat-google/

Tenemos el API de Web denominado WebSockets, cuya tecnología permite abrir una sesión de comunicación interactiva entre el navegador principalmente y un servidor.

COMPRUEBA LO QUE SABES

Acabamos de estudiar las diferentes arquitecturas de conexión y comunicación. ¿En qué apartado pondríamos las comunicaciones multicast?

Razona la respuesta y coméntala en el foro.



EJEMPLO PRÁCTICO

En la empresa de alimentación en la que trabajamos, queremos programar dos aplicaciones sin tener que utilizar bases de datos y que utilicen la programación en red para la comunicación entre las tiendas. Por un lado, queremos tener un chat privado e interno entre tiendas y almacén y, por otro, un sistema de avisos en el caso de productos en tienda con alguna necesidad especial. ¿Qué tipo de conexiones elegiríamos?

Vamos a ver que durante nuestros desarrollos softwares casi siempre podremos resolver las problemáticas planteadas mediante soluciones cliente/servidor y, de hecho, la gran mayoría de aplicaciones en red que se programan hoy en día son de ese tipo.

Para las propuestas que tenemos se pueden plantear dos soluciones:

- La solución sobre chat es claramente cliente/servidor, ya que un servicio deberá estar gestionando los mensajes, a quién van dirigidos y cómo presentarlos. Por lo tanto, será una comunicación orientada a la conexión y tipo cliente/servidor. Se puede llegar a pensar en salas privadas, entonces el desarrollo deberíamos pensarlo tipo p2p.

- En el caso del sistema de avisos, podemos usar una conexión UDP, es decir, no orientada a la conexión. La comunicación es muy sencilla, del tipo aviso y respuesta al aviso.

ENLACE DE INTERÉS

En el siguiente enlace podrás ampliar información sobre los WebSockets:

https://developer.mozilla.org/es/docs/Web/API/WebSockets_API

2. ELEMENTOS DE PROGRAMACIÓN DE APLICACIONES EN RED. API. LIBRERÍAS Y CLASES

Nos planteamos realizar dos tipos de comunicaciones: en una realizaremos una escucha de posibles alertar por parte de las tiendas para poder reaccionar ante el desabastecimiento de productos y la segunda comunicación para tener un chat interno entre empleados.

¿Cómo realizamos cada comunicación? ¿Necesitamos siempre programar un servidor? ¿Con Java tenemos todas las herramientas para poder desarrollar las comunicaciones?

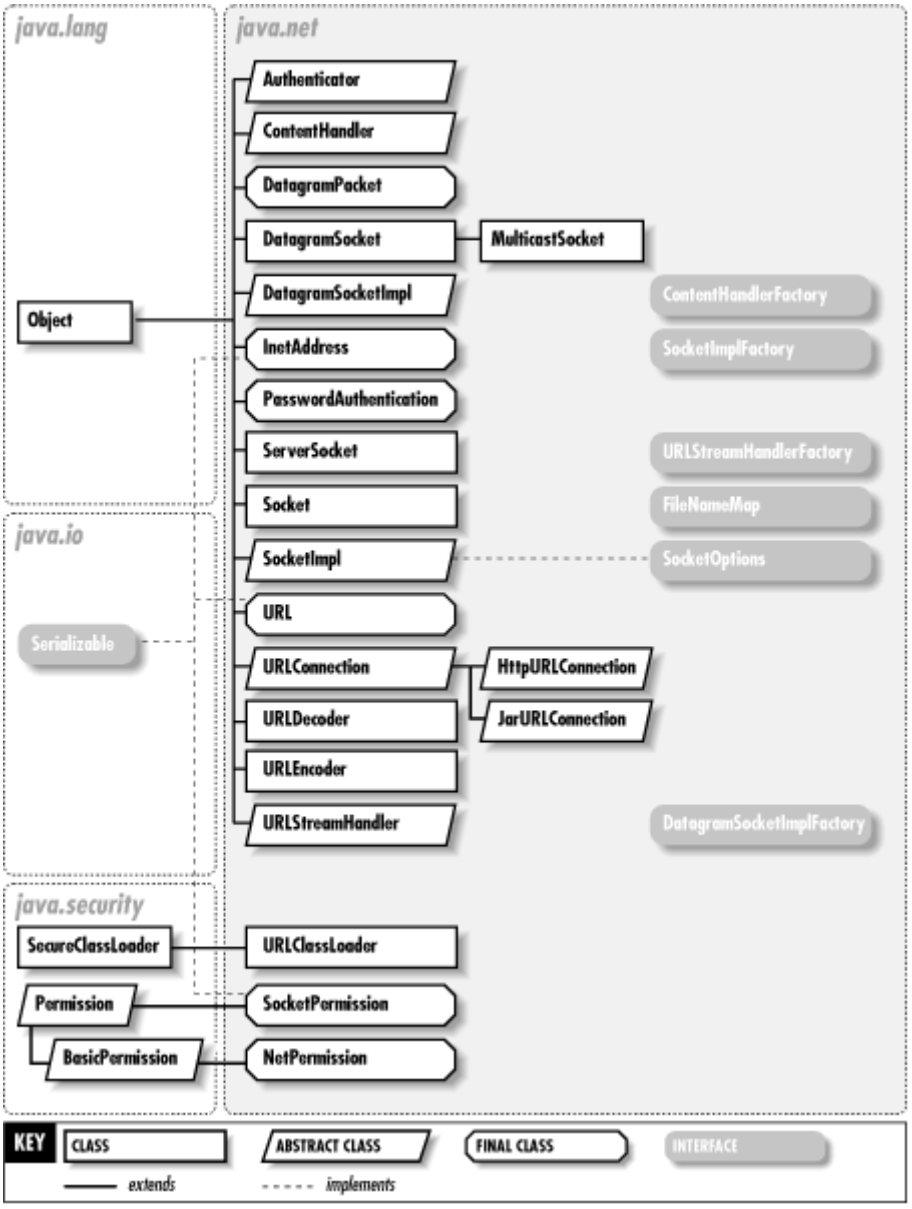
Con las respuestas a estas preguntas, será factible poder desarrollar todas las comunicaciones necesarias dentro de nuestra aplicación.

Los lenguajes de alto nivel disponen de bibliotecas especializadas en el desarrollo de aplicaciones distribuidas. A pesar de que cada lenguaje contempla sus particularidades, todos ellos presentan bastantes elementos comunes que dan respuesta a los conceptos básicos de direccionamiento, envío de información, conexión y canal de transmisión o tratamiento de recursos remotos de la forma más transparente posible.

El lenguaje Java aprovecha la riqueza que le da el paradigma orientado a objetos para definir una jerarquía de clases que envuelve toda esta potencialidad a través de métodos de muy alto nivel que facilitan el desarrollo de aplicaciones distribuidas robustas. De hecho, el API que contempla la biblioteca estándar de Java solo permite trabajar a nivel de la capa de aplicación usando la interfaz de acceso exclusivo a la capa de transporte. El JDK estándar no contempla la posibilidad de acceder a las capas más bajas (red o enlace).

Se trata de una implementación realmente eficiente que da respuesta a todas las necesidades originadas desde la capa de aplicación. Por lo tanto, no es necesario que nos planteemos si tenemos que usar otras alternativas, ni mucho menos de implementarlas nosotros.

Son muchas las librerías, clases, métodos y funciones que tenemos dentro de Java, tal y como veremos a continuación. El paquete principal de librerías y clases lo encontramos dentro de java.net, como vemos en el siguiente árbol de clases dentro de java.net:



Paquete java.net

Fuente: https://www.antevenio.com/blog/2019/07/chat-google/

Como indica la documentación oficial, el paquete java.net provee la implementación de las clases necesarias para las aplicaciones en red.

ENLACE DE INTERÉS

En el siguiente enlace encontrarás la documentación oficial de java.net:

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/net/package-summary.html

2.1 Sockets, conceptos y características

Los sockets son una Interfaz de Programación de Aplicaciones (API) que permite a dos aplicaciones intercambiar información a pesar de que se ejecuten en dispositivos diferentes. Representan la puerta de entrada y salida en la red y constituyen la base de cualquier aplicación distribuida.

Recordemos que la arquitectura de las redes define una pila de capas desde la aplicación al medio físico para hacer factible la transmisión de datos entre programas. Los sockets estarían situados en la capa de transmisión y representarían el punto de acceso de las aplicaciones a las capas inferiores del sistema.

Por motivos de seguridad y robustez, Java no dispone de ninguna utilidad que permita trabajar directamente en las capas inferiores, por lo tanto, los sockets se convierten en la utilidad de programación de más bajo nivel del lenguaje Java. Esto significa que no es posible hacer implementaciones de protocolos que se encuentren por debajo de la capa de transmisión.

Los sockets se encuentran asociados a una IP y a un puerto, de forma que sea posible dirigir información a través de la red usando alguno de los protocolos sobre IP disponibles (TCP o UDP).

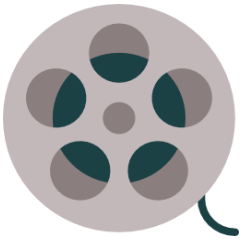
El ciclo de vida de un socket estaría compuesto por las siguientes etapas:

- Creación y apertura del socket.
- Lectura y escritura de información, es decir, recepción y envío de datos por el socket.
- Cierre del socket.

Para la creación de sockets, el lenguaje de programación Java proporciona en el paquete java.net las siguientes clases:

- Java.net.socket: Proporciona métodos para la entrada y salida a través de streams.

- Java.net.serversocket: Se utiliza en las aplicaciones servidoras para escuchar las peticiones que realizan los clientes conectados.



VIDEO DE INTERÉS

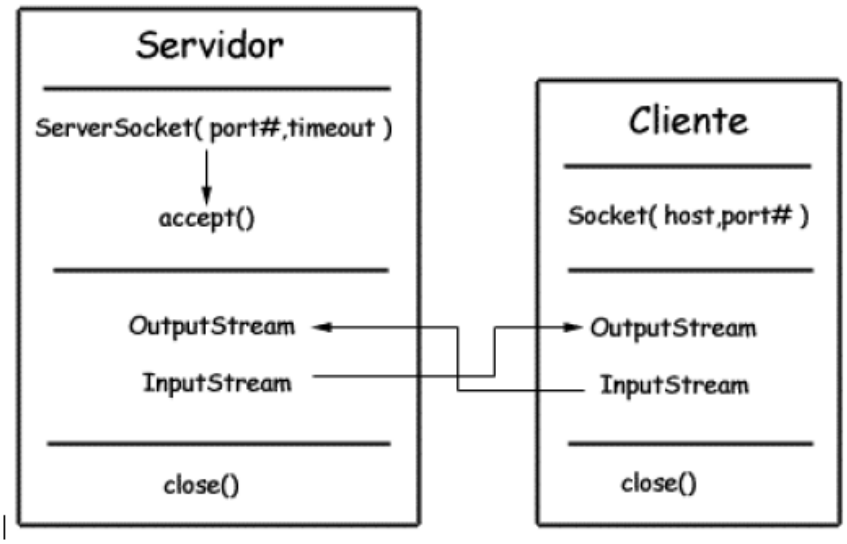
En el siguiente vídeo encontrarás una descripción sobre los sockets en comunicaciones, sus tipos y cómo identificarlos:

https://www.youtube.com/watch?v=teCS58E5YMY

2.1.1 Creación de sockets. Enlazado y establecimiento de conexiones

Para la apertura de sockets tenemos que diferenciar entre si estamos programando conexiones orientadas a la conexión o no orientadas a la conexión y dentro de las orientadas a la conexión, si estamos en la parte de cliente o la de servidor. Todas estas opciones las vamos a trabajar en los apartados siguientes.

Por ahora veamos un ejemplo de los más complejos que nos podemos encontrar en la creación, enlazado y establecimiento de las conexiones:



Conexiones con sockets.

Para la construcción de los sockets y de acuerdo a la documentación oficial, tenemos los siguientes constructores y usos:

Constructor	Uso
Socket(String servidor, int puerto) throws IOException, UnknownHostException	Crea un nuevo socket hacia el servidor utilizando el puerto indicado.
Socket(InetAdress servidor, int puerto) throws IOException	Como el anterior, solo que el servidor se establece con un objeto InetAddress.
Socket(InetAdress servidor, int puerto, InetAdress dirLocal, int puertoLocal) throws IOException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.
Socket(String servidor, int puerto, InetAdress dirLocal, int puertoLocal) throws IOException, UnknownHostException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.

Un ejemplo para la creación de un nuevo socket de comunicaciones podría ser el siguiente:

```
try{
Socket s=new Socket("time-a.mist.gov",13);
}
catch(UnknownHostException une){
System.out.println("No se encuentra el servidor");
}
catch(IOException une){
System.out.println("Error en la comunicación");
}
```

Dentro de la clase Socket nos encontramos con otros métodos que nos permiten gestionar una comunicación, y finalizarla:

Método	Uso
void setSoTimeout(int tiempo)	Establece el tiempo máximo de bloqueo cuando se está esperando entrada de datos por parte del socket. Si se cumple el tiempo, se genera una interrupción del tipo: InterruptedIOException.
void close()	Cierra el socket.
boolean isClosed()	**true** si el socket está cerrado.

2.1.2 Utilización de sockets para la transmisión y recepción de información

El uso de los sockets para realizar comunicaciones a través de la red permite tener un canal con el cual poder enviar y recibir información. La clase Socket nos proporciona dos métodos para poder realizar envío de información:

Método	Uso
InputStream getInputStream() throws IOException	Obtiene la corriente de entrada de datos para el socket.
OutputStream getOutputStream() throws IOException	Obtiene la corriente de salida de datos para el socket.

Estos dos métodos están orientados al envío de bytes, por lo que tendríamos que realizar conversiones tipo getBytes.

Otra forma de usar los streams es mediante objetos como BufferredReader para la entrada o PrintWriter para la salida, mucho más cercanos al String, pero que también son factibles su conversión a bytes.

Un ejemplo de uso y envío de información a través de esos streams lo podemos ver en el siguiente código.

```
try{
Socket socket=new Socket("servidor.dementiras.com",7633);
BufferedReader in=new BufferedReader(
new InputStreamReader(socket.getInputStream()));
PrintWriter out=new PrintWriter(
socket.getOutputStream(),true)); // el parámetro
//true sirve para volcar la salida al
//dispositivo de salida (autoflush)
boolean salir=false;
do {
s=in.readLine();
if(s!=null) System.out.println(s);
else salir=true;
}while(!salir);
}
catch(UnknownHostException une){
System.out.println("No se encuentra el servidor");
}
catch(IOException une){
System.out.println("Error en la comunicación");
}
```

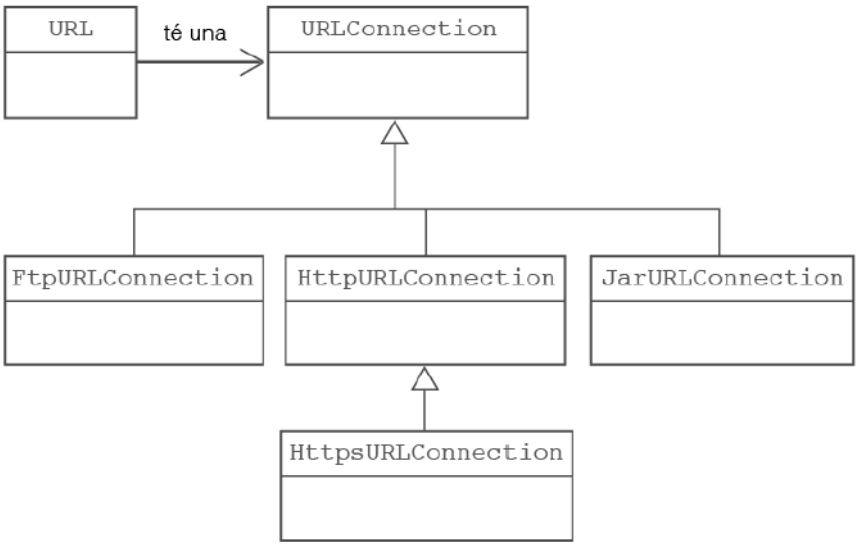
VIDEO DE INTERÉS

En el siguiente enlace encontrarás un vídeo muy explicativo sobre los sockets y los streams en Java:

https://www.youtube.com/watch?v=4AG-HHIhbl8

Para identificar un recurso de la web, podemos usar una URL (*Uniform Resource Locator*). Una URL es una cadena de caracteres única para cada recurso diferente que sigue unas determinadas reglas sintácticas y contiene información de donde se encuentra el recurso, de cómo se puede localizar. Al tratarse de una cadena única para cada recurso decimos que, además de localizar el recurso, también lo identifica. Por eso, las URL se consideran también identificadores de recursos.

El Java Development Kit (JDK) de Java dispone de dos clases para poder trabajar fácilmente con recursos remotos identificados con una URL. Nos referimos a las clases URL y URLConnection. Se trata de clases de muy alto nivel que tendríamos que situar dentro de la capa de aplicación.



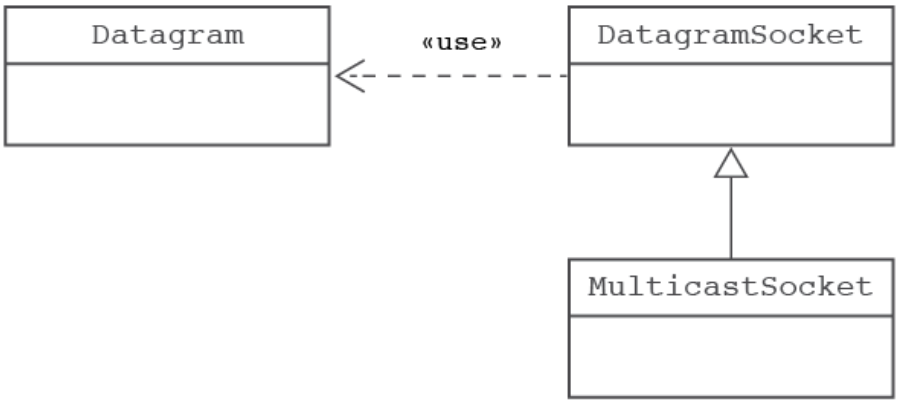
Clases sobre URLs.
Fuente: https://ioc.xtec.cat/

A nivel de aplicación normalmente trabajaremos siempre con la clase URLConnection y no con sus derivadas porque así podemos despreocuparnos del protocolo utilizado y dar un tratamiento estándar a todas las conexiones.

Algo pareciendo pasa con los objetos de tipos flujo que serán devueltos para conseguir el contenido remoto. La clase usada dependerá del tipo de contenido según sea imagen, PDF, texto plano o un documento HTML, pero por el programador todas ellas habrá que tratarlas como una instancia de la clase InputStream. Se trata de uniformizar el tratamiento de todo el contenido tanto como sea posible.

2.1.3 Implementación de sockets no orientados a la conexión

El lenguaje Java contempla 3 clases a hora de hacer implementaciones de comunicación no orientada a conexión. DatagramSocket y DatagramPacket constituyen las clases básicas de la comunicación a través de UDP. La tercera deriva de DatagramSocket y apoya específicamente a las comunicaciones de tipos multicast.



Clases sockets no orientados a la conexión.
Fuente: https://ioc.xtec.cat/

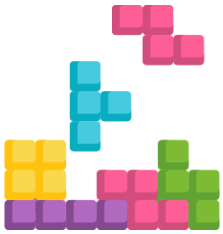
Los zócalos creados usando DatagramSocket son capaces de enviar y recibir los paquetes especificados por el protocolo UDP. Al crear una instancia podemos especificar un puerto concreto que el zócalo escuchará cuando sea necesario atender algún servicio (estándar o no) asociado al puerto:

- ***DatagramSocket***(). Genera una instancia de DatagramSocket asociada a un puerto temporal.
- ***DatagramSocket***(int puerto). Genera una instancia de DatagramSocket asociada al puerto que se indica en el parámetro.
- ***DatagramSocket***(int puerto, InetAddress). En dispositivos que tengan más de una IP se podrá especificar a través del segundo parámetro la IP que se vinculará la instancia generada.

ARTÍCULO DE INTERÉS

En el siguiente enlace encontrarás un artículo sobre la programación multicast con Java:

https://ambellido.blogspot.com/2012/08/multicast-en-java-iii.html



EJEMPLO PRÁCTICO

En las tiendas de la empresa de alimentación en la que hemos sido contratados, queremos poder tener un sistema de avisos y que cuando uno de los productos esté por debajo de los mínimos, el almacén reciba este aviso y envíe el producto a la tienda. ¿Cómo se realizaría esta tarea con conexión UDP?

1. En primer lugar, programaríamos un módulo que recibiera el producto y las unidades y enviara el mensaje.

```
//bytes del mensaje a enviar
byte[] msg = "ProductoID: 5Uds".getBytes();
//dirección IP destino
InetAddress ipDestino = InetAddress.getByName("localhost");
//puerto destino
int pruestoDestino = 5555;
//creación del paquete
DatagramPacket packet = new DatagramPacket(msg,
msg.length,
ipDestino,
pruestoDestino);
//creación socket
DatagramSocket socket = new DatagramSocket();
//ecnvio
socket.send(packet);
```

2. Por otro lado, debemos programar la recepción.

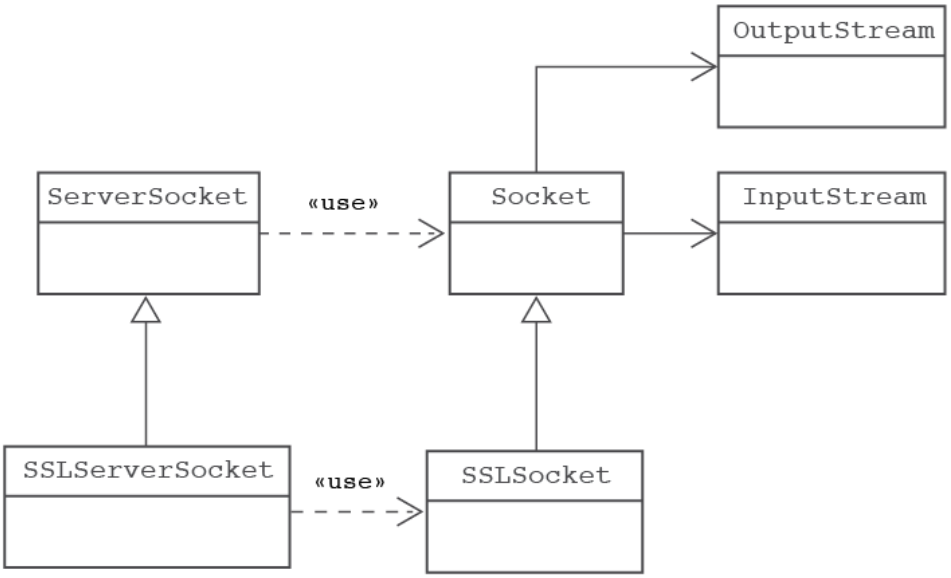
```
//el servidor se abre escuchando indefinidamente
while(true){
//creación del paquete de recepción con un máximo de 1024 bytes
DatagramPacket packet = new DatagramPacket(receivingData, 1024);
//recepcion
socket.receive(packet);
//procesamiento
sendingData = processData(packet.getData(), packet.getLength());
//ip del emisor para la respuesta
clientIP = packet.getAddress();
// puerto del emisor para la respuesta
clientPort = packet.getPort();
//paquete de respuesta
packet = new DatagramPacket(sendingData, sendingData.length,
clientIP, clientPort);
//envío
socket.send(packet);
}
```

2.1.4 Implementación de sockets orientados a la conexión

Los sockets orientados a conexión usan el protocolo TCP. Recordemos que en este caso los datos se pasan a la aplicación en el mismo orden en que han salido y se asegura que no se pierde información durante la transmisión.

El protocolo TCP define que antes de empezar la transmisión de datos hay que hacer una petición de conexión que el otro parto tendrá que aceptar. Una vez aceptada la conexión, en ambos lados se reservará un puerto de red exclusivamente para la transmisión de datos en cualquiera de los dos sentidos. Recordáis que TCP es un protocolo que define una comunicación hoja-dúplex exclusiva entre dos dispositivos.

En la imagen podemos ver el diagrama de las clases que entran en juego en una transmisión TCP de Java. Las clases SSLServerSocket y SSLSocket ofrecen la misma funcionalidad que ServerSocket y Socket respectivamente, pero están especializadas al transmitir de forma segura usando un protocolo de seguridad denominado SSL.



Clases sockets orientados a la conexión.
Fuente: https://ioc.xtec.cat/

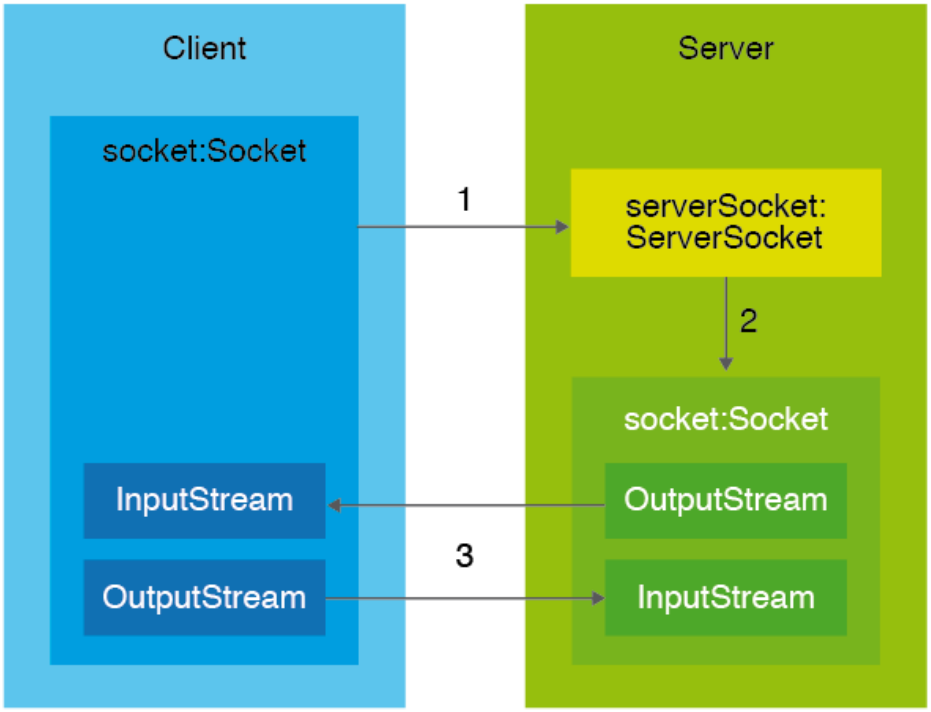
COMPRUEBA LO QUE SABES

Acabamos de estudiar las clases más básicas para establecer una comunicación entre ordenadores. ¿Por qué es necesario cerrar un Socket abierto?

Razona la respuesta y coméntala en el foro.

2.2 Sockets servidores y clientes. Programación de aplicaciones

El envío a través del flujo de salida de uno de los sockets implicará la recepción de los mismos datos enviados a través del flujo de entrada del otro socket. El esquema sería el siguiente:



Flujo de funcionamiento.
Fuente: https://ioc.xtec.cat/

1. El socket de la parte cliente pide conectarse a la dirección en la cual se encuentre el servidor y en el puerto que escuche una instancia de ServerSocket del servidor.
2. El ServerSocket acepta la conexión y crea un socket en la parte del servidor con dos canales, uno de entrada y uno de salida.

3. El InputStream (flujo de entrada) del socket de la parte cliente se encuentra conectado al OutputStream (flujo de salida) del socket de la parte servidor y su flujo de entrada conecta con el flujo de salida del socket del cliente. De este modo es posible la comunicación en ambas direcciones con la calidad exigida por el protocolo TCP.

El cliente necesitará una instancia de Socket, la cual dispondrá desde el primer momento de los recursos propios del protocolo TCP, puesto que las instancias de Socket suelen crearse indicando la dirección y el puerto con el que habrá que comunicarse y desde la creación que se realiza la petición inicial para poder generar los recursos pertinentes como los flujos de datos que simularán los dos canales de comunicación que el protocolo TCP pide, tal y como hemos visto en el apartado de creación de sockets.

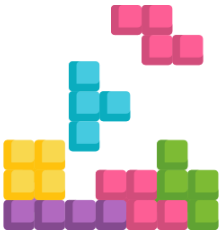
```
Socket socket = new Socket(new InetAddress.getByName("192.168.4.1"),
7777);
```

Durante la creación del zócalo del cliente se enviará al servidor la petición, el cual al recibirla la aceptará creando un socket específico para comunicarse con el cliente aceptado a la banda del servidor. Recordemos que la comunicación TCP es exclusiva por dos dispositivos, por eso el servidor tiene que ir creando un socket específico por cada cliente que haga la petición.

La creación del socket en el lado del servidor se encuentra automatizado y se obtiene como retorno del método de **ServerSocket** que inicia la escucha del puerto hasta la llegada de la petición. Nos referimos al método **accept**.

```
Socket socket = serverSocket.accept();
```

A partir de aquí ambos sockets, el de la parte cliente y el de la parte servidor, dispondrán de un canal de entrada y uno de salida adecuadamente conectados. La implementación de los canales se realizará a través de instancias internas de flujos de entrada y salida. Para obtener los flujos de un zócalo habrá que invocar los métodos getInputStream y getOutputStream para obtener respectivamente los flujos de entrada y salida.



EJEMPLO PRÁCTICO

En las tiendas de la empresa de alimentación en la que desarrollamos aplicaciones, tenemos un sistema que nos avisa cuando uno de los productos está por debajo de los mínimos. En el almacén reciben este aviso y envían el producto a la tienda. Ahora mismo está implementado entre un ordenador del almacén y un ordenador de tienda. Sin embargo, se quiere modificar la comunicación para que puedan interactuar múltiples ordenadores contra un servidor. ¿Cómo se realizaría la aplicación cliente/servidor?

1. En primer lugar, programaríamos la parte del servidor.

```
public class TcpSocketServer {
static final int PORT=9090;
private boolean end=false;

public void listen(){
ServerSocket serverSocket=null;
Socket clientSocket = null;
try {
serverSocket = new ServerSocket(PORT);

while(!end){
clientSocket = serverSocket.accept();
//processamos la petición del cliente
proccesClientMsg(clientSocket);
//cerramos el socket con el cliente
closeClient(clientSocket);
}
//cerramos el socket principal
if(serverSocket!=null && !serverSocket.isClosed()){
serverSocket.close();
}

} catch (IOException ex) {
Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
null, ex);
}
}

...
}
```

2. En segundo lugar, programaríamos el método en el servidor que procesaría la petición de un cliente.

```
public void proccesClientMsg (Socket clientSocket){
boolean farewellMessage=false;
String clientMessage="";
BufferedReader in=null;
PrintStream out=null;
try {
in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
out= new PrintStream(clientSocket.getOutputStream());
do{
//Aquí procesamos el mensaje del cliente
String dataToSend = processData(clientMessage);
out.println(dataToSend);
out.flush();
clientMessage=in.readLine();
farewellMessage = isFarewellMessage(clientMessage);
}while((clientMessage)!=null && !farewellMessage);
} catch (IOException ex) {
Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
null, ex);
}
}
```

3. En la parte del cliente tendríamos el siguiente código:

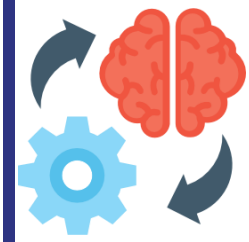
```
public class TcpSocketClient{

public void connect(String address, int port) {
String serverData;
String request;
boolean continueConnected=true;
Socket socket;
BufferedReader in;
PrintStream out;
try {
socket = new Socket(InetAddress.getByName(address), port);
in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
out = new PrintStream(socket.getOutputStream());
//el cliente debe estar conectado al puerto hasta que se cierre
while(continueConnected){
serverData = in.readLine();
//se procesa la respuesta y se envía nueva petición
request = getRequest(serverData);
out.println(request);//assegurem que acaba amb un final de línia
out.flush(); //assegurem que s'envia
//comprobamos si se finaliza
continueConnected = mustFinish(request);
}

close(socket);
} catch (UnknownHostException ex) {
reportError("Error de conexión, no existe el socket", ex);
} catch (IOException ex) {
reportError("Error de conexión indefinido", ex);
}
}
...
}
```

2.3 Utilización de hilos en la programación de aplicaciones en red

En puntos anteriores hemos estudiado que un mismo servidor puede atender a varios clientes. Para esto se utilizan los threads o hilos. En la unidad anterior estudiamos cómo se programan, así como las clases y métodos que se utilizan. Uno de estos era el método run() que en este caso contendrá las instrucciones de comunicación con el cliente.



RECUERDA

A lo largo de toda la unidad 2, trabajamos la programación y gestión de hilos. En el diseño de un servidor y clientes puede ser muy importante la incorporación de esta tecnología.

En estos casos, el servidor, como siempre, está a la espera de clientes y cuando un cliente le envía una solicitud de comunicación, este le asigna un thread para él. Por tanto, se crea una clase derivada de la clase Thread para atender a los clientes.

Un servidor teniendo en cuenta la programación multihilo tendría el siguiente código:

```
public static void main(String args[]){
try{
//Se crea el servidor de sockets
ServerSocket servidor=new ServerSocket(8347);
while(true){//bucle infinito
//El servidor espera al cliente siguiente y le
//asigna un nuevo socket
Socket socket=servidor.accept();
//se crea el Thread cliente y se le pasa el socket
Cliente c=new Cliente(socket);
c.start();//se lanza el Thread
}
}catch(IOException e){}
```

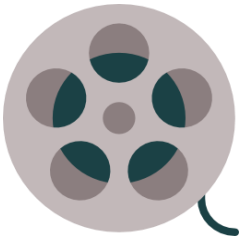
En el caso del cliente:

```
public class Cliente extends Thread{
private Socket socket;
public Cliente(Socket s) {
socket=s;
}
public void run(){
try{
//Obtención de las corrientes de datos
BufferedReader in=new BufferedReader(
new InputStreamReader(
socket.getInputStream()));
PrintWriter out=new PrintWriter(
socket.getOutputStream(),true);
out.println("Bienvenido”);
boolean salir=false;//controla la salida
while (!salir){
resp=in.readLine();//lectura
...//proceso de datos de lectura
out.println(“....”);//datos de salida
if(...) salir=true;//condición de salida
}
out.println("ADIOOOOOOS");
socket.close();
}catch(Exception e){}
}
```

COMPRUEBA LO QUE SABES

Acabamos de finalizar la unidad enlazando la programación multihilo con la de red, ¿es posible programar una comunicación cliente/servidor sin tener en cuenta el multihilo?

Enfoca las respuestas a la parte del servidor principalmente y coméntalo en el foro.



VIDEO DE INTERÉS

En el siguiente enlace encontrarás un vídeo muy interesante sobre la programación de clientes y servidores con Java:

https://www.youtube.com/watch?v=bNXpF7tF10U

RESUMEN FINAL

Las redes de computadores lo son todo hoy en día, tanto en el presente como en el futuro, ya que se presentan como el elemento de evolución principal a nivel profesional y personal.

Como hemos visto, para un correcto funcionamiento, debemos conocer los estándares que hacen funcionar esas redes, estándares de redes de OSI y, sobre todo, el estándar por niveles TCP/IP, donde protocolos de conexión (TCP) y no conexión (UDP) marcan el posterior desarrollo software.

Los lenguajes de alto nivel disponen de bibliotecas especializadas en el desarrollo de aplicaciones distribuidas. A pesar de que cada lenguaje contempla sus particularidades, todos ellos presentan bastantes elementos comunes que dan respuesta a los conceptos básicos de adrizamiento, envío de información, conexión y canal de transmisión o tratamiento de recursos remotos de la forma más transparente posible.

En Java, tenemos la librería de java.net que nos permite realizar todas estas acciones de forma transparente a la arquitectura establecida, sin necesidad de conocer las capas más físicas de una comunicación, ya que el desarrollo de los módulos de comunicación con Java se centra en la capa de aplicación y en el uso de los sockets como elementos para la definición tanto de conexiones punto a punto como las de cliente/servidor.

Mediante las clases orientadas a la conexión, además, podremos establecer una arquitectura cliente/servidor, una de las implementaciones más usadas hoy en el desarrollo de software.

En este tipo de comunicación, tal y como en la última parte hemos estudiado, la programación multihilo nos puede ayudar a ampliar las capacidades de nuestro servicio.