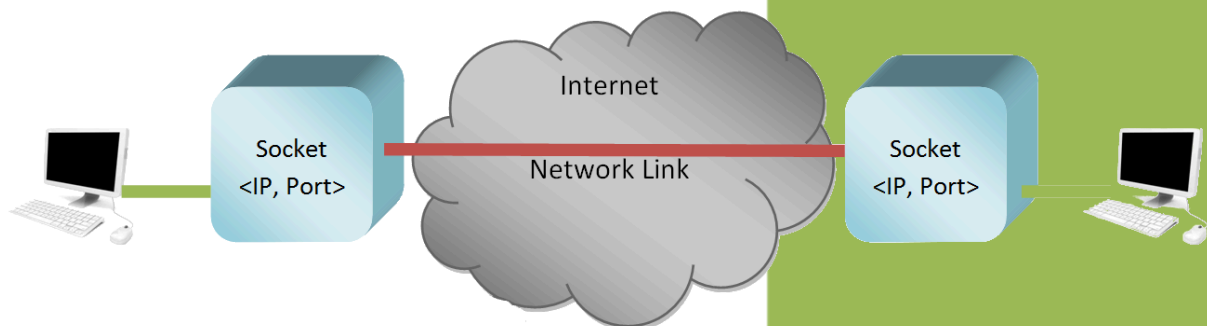


PROGRAMACIÓN DE SERVICIOS Y PROCESOS

UT3: Sockets. Comunicación entre Procesos



Contenido

1. ARQUITECTURA DE INTERNET: UNA ESTRUCTURA DE CAPAS	2
1.1. Protocolos de comunicaciones	2
1.1.1. <i>Conceptos básicos</i>	2
1.2. Arquitecturas por niveles o capas	2
1.2.1. Conceptos	2
1.2.2. <i>Beneficios del uso de un modelo en capas</i>	3
1.2.3. <i>Modelo de referencia OSI</i>	5
2. CAPA DE TRANSPORTE	7
2.1. Funciones de la Capa de Transporte	7
3. SOCKETS TCP	18
3.1. Clase ServerSocket	18
3.2. Clase Socket	19
3.3. Gestión de sockets TCP	20
3.4. Conexión de múltiples clientes. Hilos	28
3.5. Envío de Objetos a través de sockets	32

1. ARQUITECTURA DE INTERNET: UNA ESTRUCTURA DE CAPAS

La comunicación de dos equipos conectados a una red de datos es un proceso complejo. Por tanto, el estudio y desarrollo de estas redes son tareas complicadas. Para facilitar estas tareas suelen emplearse estrategias modulares basadas en el principio conocido como “divide y vencerás”. Así, el problema complejo se divide en partes más simples que son más sencillas de comprender y desarrollar. Cada parte se encarga de realizar de manera transparente determinada función dentro del proceso de comunicación, de tal forma que las demás partes se desprecupan de dicha función. El conjunto de todas las funciones realizadas por estas partes representa el proceso de comunicación.

1.1. Protocolos de comunicaciones

1.1.1. Conceptos básicos

La información que en un instante dado viaja a través de una red desde un emisor hasta un receptor determinado se conoce genéricamente como **paquete** o **paquete de datos**.

Para que los paquetes de datos puedan viajar desde el origen hasta su destino a través de una red, es importante que **todos los dispositivos de la red** hablen el mismo lenguaje o protocolo.

Un **protocolo** es un conjunto de normas perfectamente definidas y convenidas de mutuo acuerdo entre los participantes de una comunicación que hacen que esta comunicación sea posible ya que regulan estrictamente la conexión, la comunicación y la transferencia de datos entre dos o más sistemas.

Los protocolos de comunicación a través de la red describen la función de todos los elementos de la red (sean físicos o lógicos) implicados en la intercomunicación.

Los protocolos pueden estar implementados tanto en **hardware** (tarjetas de red y demás dispositivos de red) como en **software** (drivers o programas).

Los protocolos pueden ser creados por **empresas de comunicaciones**, por **asociaciones de estándares** o por **simples usuarios** y en función de cómo los definan estos, pueden ser **privados** (solo lo usa su creador) o **públicos** (quedando a disposición de todo aquel que lo quiera usar).

1.2. Arquitecturas por niveles o capas

1.2.1. Conceptos

Para reducir tanta complejidad, las redes se conciben utilizando una serie de **niveles** o **capas** jerarquizadas. Así, en lugar de tener un solo protocolo gigante que especifique todos los detalles de la comunicación, el problema es dividido en pequeñas partes.

Este sistema se caracteriza por los siguientes aspectos:

- Cada capa se ocupa internamente de llevar a cabo **una pequeña parte** de todo el proceso de comunicación y ofrece a la capa superior una serie de **servicios** o **funciones**.
- Cada capa, para llevar a cabo su cometido, utiliza los **servicios** que le presta su **capa inferior**.

- Cada capa de la máquina emisora se comunica indirectamente **con la misma capa** de la máquina receptora a través de un **protocolo**.
- Al conjunto de capas y protocolos utilizados en una red lo conocemos como la **arquitectura de red** o **pila de protocolos**.

Esta arquitectura de red debe estar lo suficientemente **clara** como para que los fabricantes de software y hardware puedan diseñar sus productos con la garantía de que funcionarán en comunicación con otros equipos que sigan la misma arquitectura de red.

1.2.2. Beneficios del uso de un modelo en capas

Un modelo en capas describe el funcionamiento de los protocolos que se produce en cada capa y la interacción de los protocolos con las capas que se encuentran por encima y por debajo de ellas.

Hay **beneficios** por el uso de un modelo en capas para describir protocolos de red y operaciones. Uso de un modelo en capas:

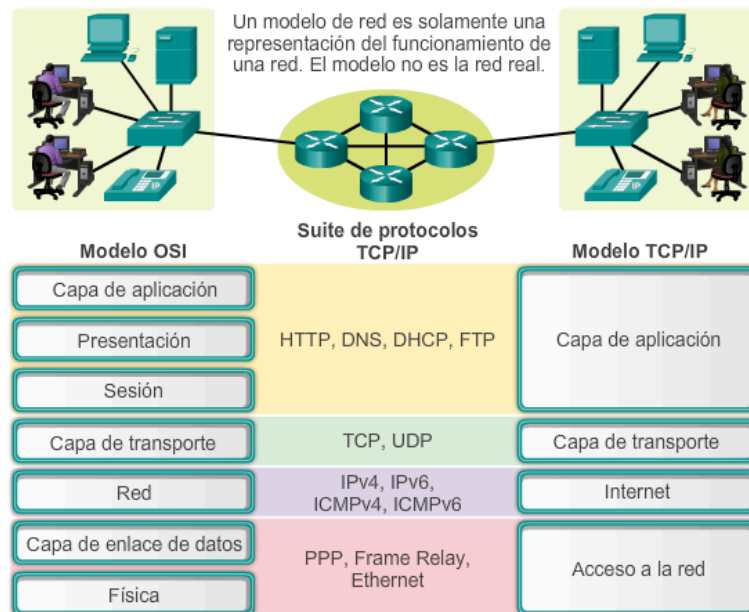
- Ayuda en el diseño de protocolos, ya que los protocolos que operan en una capa específica tienen información definida según la cual actúan, y una interfaz definida para las capas superiores e inferiores.
- Fomenta la competencia, ya que los productos de distintos proveedores pueden trabajar en conjunto.
- Evita que los cambios en la tecnología o en las capacidades de una capa afecten otras capas superiores e inferiores.
- Proporciona un lenguaje común para describir las funciones y capacidades de redes.

Existen dos tipos básicos de modelos de redes:

- **Modelo de protocolo:** este modelo coincide con precisión con la estructura de una suite de protocolos determinada. El modelo TCP/IP es un modelo de protocolo, porque describe las funciones que tienen lugar en cada capa de protocolos dentro de una suite TCP/IP.
- **Modelo de referencia:** este modelo es coherente con todos los tipos de servicios y protocolos de red al describir qué es lo que se debe hacer en una capa determinada, pero sin regir la forma en que se debe lograr. Un modelo de referencia no está pensado para ser una especificación de implementación ni para proporcionar un nivel de detalle suficiente para definir de forma precisa los servicios de la arquitectura de red. El objetivo principal de un modelo de referencia es ayudar a lograr un mejor entendimiento de las funciones y procesos involucrados.

El modelo OSI es el modelo de referencia de internetwork más conocido. Se usa para diseño de redes de datos, especificaciones de funcionamiento y resolución de problemas.

Como se muestra en la ilustración, los modelos TCP/IP y OSI son los modelos principales que se utilizan al hablar de funcionalidad de red. Los diseñadores de protocolos, servicios o dispositivos de red pueden crear sus propios modelos para representar sus productos. Por último, se solicita a los diseñadores que se comuniquen con la industria asociando sus productos o servicios con el modelo OSI, el modelo TCP/IP o ambos.



ACTIVIDAD

Busca información sobre el modelo de Comunicación OSI.

Para cada una de las capas que forman parte del modelo encuentra información específica sobre:

- Su nombre.
- Su función.
- Ejemplos concretos de cada una de estas capas.

¿Se comunican las diferentes capas de OSI entre sí?

1.2.3. Modelo de referencia OSI

El problema de trasladar información entre computadores se divide en **siete problemas más pequeños** y de tratamiento más simple en el modelo de referencia OSI.

Cada uno de los siete problemas más pequeños está representado por su propia capa. Las siete capas del modelo de referencia OSI son:

1. La capa física.
2. La capa de enlace de datos. (Trama)

3. La capa de red. (Paquete)
4. La capa de transporte. (Segmento)
5. La capa de sesión.
6. La capa de presentación.
7. La capa de aplicación.

Para que los datos viajen desde el origen hasta el destino, cada capa del modelo OSI del origen debe comunicarse con la misma capa del destino. Esta comunicación se llama **comunicación de igual a igual** y está basada en un **protocolo** específico.

La información que intercambian las capas se llama genéricamente **unidad de datos del protocolo** (PDU). Las PDU, dependiendo de la capa, reciben un nombre específico. A la PDU de la capa 4 se le llama **Segmento**, a la PDU de la capa 3 se le llama **Paquete** y a la PDU de la capa 2 se le llama **Trama**.

Cuando se desean enviar datos, estos van atravesando en el ordenador origen en sentido descendente las distintas capas del modelo y se van añadiendo **encabezamientos** sucesivos que contienen información de control (esto se denomina **encapsular los datos** de la capa superior).

El conjunto resultante se transmite por el canal y en el sistema de destino se seguirá el camino inverso (**desencapsulando los datos**).

Capa 7. Capa de aplicación

Los procesos informáticos que trabajan en este nivel están incluidos dentro de los programas que utiliza el usuario final y se comportan siguiendo alguno de los **protocolos de alto nivel** definidos para esta capa.

Hay **muchísimos protocolos**, tantos como aplicaciones distintas y puesto que continuamente se desarrollan nuevas aplicaciones el número de protocolos crece sin parar.

Cabe aclarar que el usuario normal no interactúa directamente con el nivel de aplicación. Suele interactuar con programas que a su vez interactúan con el nivel de aplicación pero que nos ocultan el nivel de complejidad de esta capa.

Entre los protocolos más conocidos de esta capa destacan: **HTTP, FTP, SMTP, POP, SSH, Telnet, DNS**, etc.

Capa 6. Capa de presentación

La capa de presentación garantiza que la información que envía la capa de aplicación de un sistema pueda ser leída por la capa de aplicación del sistema receptor. De ser necesario, la capa de presentación **traduce** entre varios formatos de datos utilizando un **formato común**.

Así, aunque distintos equipos puedan tener diferentes representaciones internas de caracteres (ASCII, Unicode, ISO-8859-1), números, sonido o imágenes los datos llegan de manera reconocible al receptor.

Para conversaciones que requieren confidencialidad, este nivel se encarga de aspectos relativos a la seguridad, como pueda ser **codificar** y **encriptar** los datos para hacerlos incomprensibles a posibles escuchas ilegales.

Otra función de la capa de presentación es la de **comprimir los datos** enviados a la red para que la comunicación sea más eficiente.

Capa 5. Capa de sesión

Esta capa permite el diálogo entre emisor y receptor estableciendo una **sesión**. **Sincroniza** el diálogo entre los hosts y administra su **intercambio de datos**.

Capa 4. Capa de transporte

Su función básica es encargarse o no de la **confiabilidad**, es decir, de que los fragmentos de una comunicación lleguen todos al otro extremo de la comunicación.

Además, en el host emisor debe aceptar los datos enviados por la capa 5, **dividirlos** en pequeñas unidades si es necesario, y pasarlos a la capa 3. En el host receptor la capa de transporte se encarga de reensamblar los fragmentos y devolverle a la capa superior una corriente de datos.

Los protocolos más importantes de la capa de transporte son: **TCP y UDP**.

Capa 3. Capa de red

El cometido de la capa de red es hacer que los datos lleguen desde el origen al destino, aun cuando ambos no estén conectados directamente. Es decir que se encarga de **encontrar un camino** manteniendo una tabla global de enrutamiento y atravesando los equipos que sean necesarios, para hacer llegar los datos al destino.

Los equipos encargados de realizar este encaminamiento se denominan routers (encaminadores o enrutadores).

Los protocolos más utilizados en la capa de red son: **IP** (v4 y v6), **ARP**, **ICMP**, **RARP**, **RIP**, etc.

Capa 2. Capa de enlace

Su tarea principal es **detectar y corregir** todos los errores que se produzcan en la línea de comunicación, proporcionando un canal **confiable** (libre de errores) a la capa 3.

También se encarga de **repartir el medio de transmisión** en aquellas redes donde existe un único medio compartido por todas las estaciones (control de acceso al medio).

Los protocolos más empleados en la capa 2 son: **Ethernet**, **WiFi**, etc.

Capa 1. La capa física

Es la capa encargada de **transmitir los bits de información a través del medio físico** utilizado para la transmisión transformando los datos provenientes del nivel de enlace en una señal adecuada al medio físico utilizado en la transmisión.

La capa física define las especificaciones eléctricas, mecánicas y de procedimiento para activar, mantener y desactivar el enlace físico entre dos nodos. Las características tales como niveles de voltaje, temporización de cambios de voltaje, velocidad de datos físicos, distancias de transmisión máximas, conectores físicos y otros atributos similares son definidas por las especificaciones de la capa física.

Los protocolos más utilizados son: **10Base5**, **10BaseT**, **codificación Manchester**, etc.

2. CAPA DE TRANSPORTE

2.1. Funciones de la Capa de Transporte

La capa de transporte es responsable de establecer una sesión de comunicación temporal entre dos aplicaciones y de transmitir datos entre ellas. Como se muestra en la ilustración, la capa de transporte es el enlace entre la capa de aplicación y las capas inferiores que son responsables de la transmisión a través de la red.

La capa de transporte proporciona un método para entregar datos a través de la red de una manera que garantiza que estos se puedan volver a unir correctamente en el



extremo receptor. La capa de transporte permite la segmentación de datos y proporciona el control necesario para rearmar estos segmentos en los distintos streams de comunicación.

En el protocolo TCP/IP, estos procesos de segmentación y rearmado se pueden lograr utilizando dos protocolos muy diferentes de la capa de transporte: el *Protocolo de Control de Transmisión (TCP)* y el *Protocolo de Datagramas de Usuario (UDP)*.

Las principales responsabilidades de los protocolos de la capa de transporte son las siguientes:

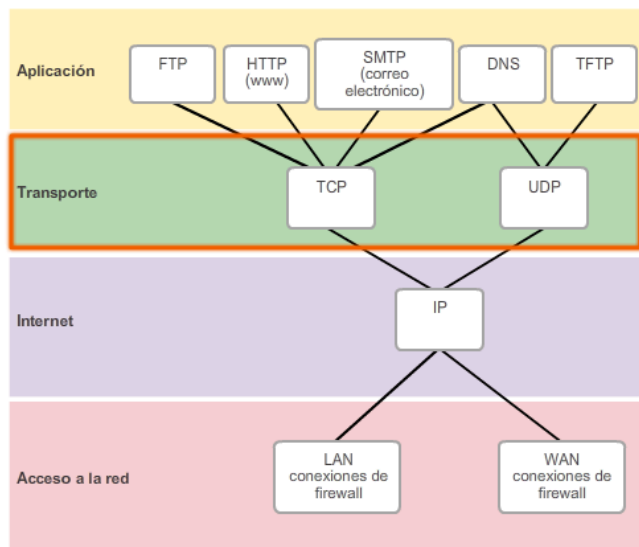
- Rastreo de comunicación individual entre aplicaciones en los hosts de origen y destino
- División de los datos en segmentos para su administración y reunificación de los datos segmentados en streams de datos de aplicación en el destino
- Identificación de la aplicación correspondiente para cada stream de comunicación.

2.1.1. Confiabilidad de la Capa de Transporte

La capa de transporte también es responsable de administrar los requisitos de confiabilidad de las conversaciones. Las diferentes aplicaciones tienen diferentes requisitos de confiabilidad de transporte.

TCP/IP proporciona dos protocolos de la capa de transporte: el **Protocolo de Control de Transmisión (TCP)** y el **Protocolo de Datagramas de Usuario (UDP)**, como se muestra en la ilustración. IP utiliza estos protocolos de transporte para habilitar la comunicación y la transferencia de datos entre los hosts.

TCP se considera un protocolo de la capa de transporte confiable y completo, lo que garantiza que todos los datos lleguen al destino. En cambio, UDP es un protocolo de la capa de transporte muy simple que no proporciona confiabilidad.



2.1.2. Protocolo TCP

TCP también proporciona las siguientes funciones:

a) Establecimiento de una sesión

TCP es un protocolo orientado a la conexión. Un protocolo orientado a la conexión es uno que negocia y establece una conexión (o sesión) permanente entre los dispositivos de origen y de destino antes de reenviar tráfico. El establecimiento de sesión prepara los

dispositivos para que se comuniquen entre sí. Mediante el establecimiento de sesión, los dispositivos negocian la cantidad de tráfico que se puede reenviar en un momento determinado, y los datos que se comunican entre ambos se pueden administrar detenidamente. La sesión se termina solo cuando se completa toda la comunicación.

b) Entrega confiable

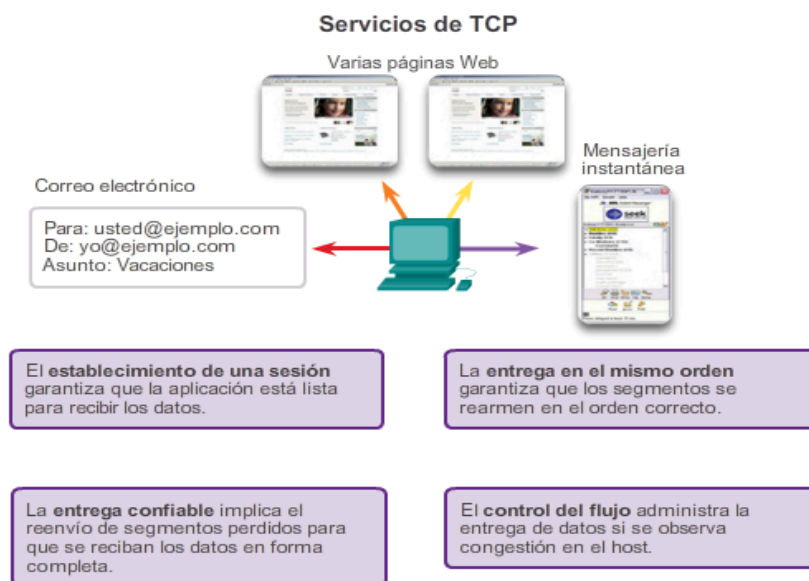
TCP puede implementar un método para garantizar la entrega confiable de los datos. En términos de redes, confiabilidad significa asegurar que cada sección de datos que envía el origen llegue al destino. Por varias razones, es posible que una sección de datos se corrompa o se pierda por completo a medida que se transmite a través de la red. TCP puede asegurar que todas las partes lleguen a destino al hacer que el dispositivo de origen retransmita los datos perdidos o dañados.

c) Entrega en el mismo orden

Los datos pueden llegar en el orden equivocado, debido a que las redes pueden proporcionar varias rutas que pueden tener diferentes velocidades de transmisión. Al numerar y secuenciar los segmentos, TCP puede asegurar que estos se rearmen en el orden correcto.

d) Control de flujo

Los hosts de la red cuentan con recursos limitados, como memoria o ancho de banda. Cuando TCP advierte que estos recursos están sobrecargados, puede solicitar que la aplicación emisora reduzca la velocidad del flujo de datos. Esto lo lleva a cabo TCP, que regula la cantidad de datos que transmite el origen. El control de flujo puede evitar la pérdida de segmentos en la red y evitar la necesidad de la retransmisión.

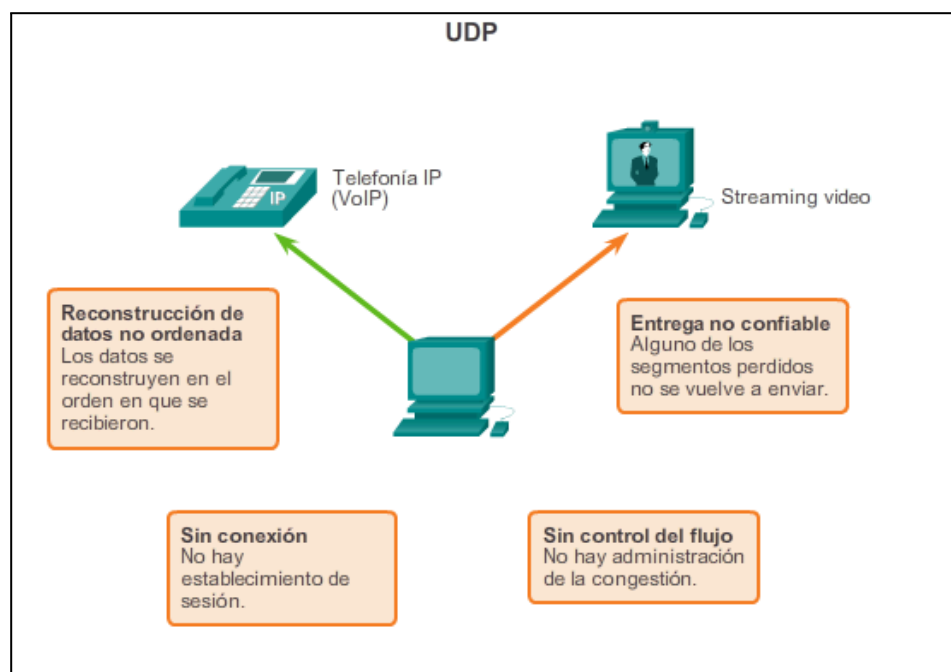


2.1.3. Protocolo UDP

UDP se considera un protocolo de transporte de máximo esfuerzo, descrito en RFC 768. UDP es un protocolo de transporte liviano que ofrece la misma segmentación y rearmado de datos que TCP, pero sin la confiabilidad y el control del flujo de TCP. UDP es un protocolo tan simple que, por lo general, se lo describe en términos de lo que no hace en comparación con TCP.

UDP presenta las siguientes características:

- a) **Sin conexión:** UDP no establece una conexión entre los hosts antes de que se puedan enviar y recibir datos.
- b) **Entrega no confiable:** UDP no proporciona servicios para asegurar que los datos se entreguen con confianza. UDP no cuenta con procesos que hagan que el emisor vuelva a transmitir los datos que se pierden o se dañan.
- c) **Reconstrucción de datos no ordenada:** en ocasiones, los datos se reciben en un orden distinto del de envío. UDP no proporciona ningún mecanismo para rearmar los datos en su secuencia original. Los datos simplemente se entregan a la aplicación en el orden en que llegan.
- d) **Sin control del flujo:** UDP no cuenta con mecanismos para controlar la cantidad de datos que transmite el dispositivo de origen para evitar la saturación del dispositivo de destino. El origen envía los datos. Si los recursos en el host de destino se sobrecargan, es probable que dicho host descarte los datos enviados hasta que los recursos estén disponibles. A diferencia de TCP, en UDP no hay un mecanismo para la retransmisión automática de datos descartados.



2.1.4. Protocolo de la capa de transporte correcto para la aplicación adecuada

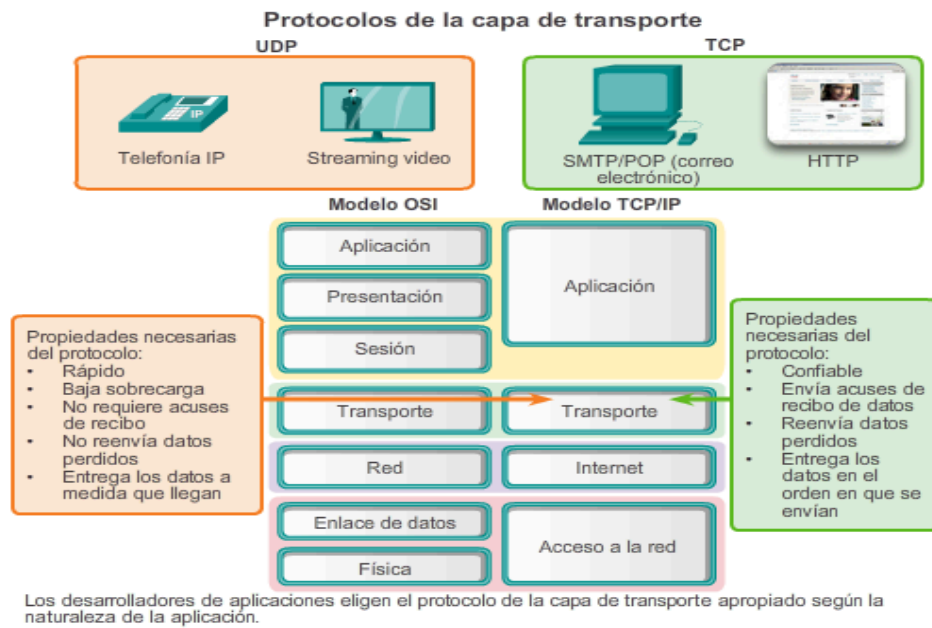
Tanto TCP como UDP son protocolos de transporte válidos. Según los requisitos de la aplicación, se puede utilizar uno de estos protocolos de transporte y, en ocasiones, se pueden utilizar ambos. Los desarrolladores de aplicaciones deben elegir qué tipo de protocolo de transporte es adecuado según los requisitos de las aplicaciones.

Para algunas aplicaciones, los segmentos deben llegar en una secuencia muy específica para que se puedan procesar correctamente. Con otras aplicaciones, todos los datos se deben recibir en forma completa para poder considerarse útiles. En ambos casos, se utiliza TCP como protocolo de transporte. Por ejemplo, las aplicaciones, como las bases de datos, los exploradores Web y los clientes de correo electrónico, requieren que todos los datos que se envían lleguen a destino en su formato original. Todos los datos perdidos pueden corromper una comunicación y dejarla incompleta o ilegible. Por lo tanto, estas aplicaciones están diseñadas para utilizar TCP. Los gastos de red adicionales se consideran necesarios para estas aplicaciones.

En otros casos, una aplicación puede tolerar cierta pérdida de datos durante la transmisión a través de la red, pero no se admiten retrasos en la transmisión. UDP es la mejor opción para estas aplicaciones, ya que se requiere menos sobrecarga de red. Con aplicaciones como streaming audio, video y voz sobre IP (VoIP), es preferible utilizar UDP. Los acuses de recibo reducirían la velocidad de la entrega, y las retransmisiones no son recomendables.

Por ejemplo, si uno o dos segmentos de un stream de video no llegan al destino, se interrumpe momentáneamente el stream. Esto puede representar distorsión en la imagen, pero quizá ni el usuario lo note. Por otro lado, la imagen en un streaming video se degradaría en gran medida si el dispositivo de destino tuviera que dar cuenta de los datos perdidos y demorar el stream mientras espera las retransmisiones. En este caso, es mejor producir el mejor video posible con los segmentos recibidos y prescindir de la confiabilidad.

La radio a través de Internet es otro ejemplo de aplicación que utiliza UDP. Si parte del mensaje se pierde durante su transmisión por la red, no se vuelve a transmitir. Si se pierden algunos paquetes, el oyente podrá escuchar una breve interrupción en el sonido. Si se utilizara TCP y se volvieran a enviar los paquetes perdidos, la transmisión haría una pausa para recibirlos, y la interrupción sería más notoria.



Ejercicio 1

Selecciona el método de entrega de la capa de transporte (TCP, UDP o ambos) más idóneo para cada uno de los protocolos de la capa de aplicación.

Protocolos de la capa de aplicación	TCP	UDP	AMBOS
DHCP			
DNS			
HTTP			
IPTV			
TFTP			
Telnet			
FTP			
SMTP			

VoIP			
------	--	--	--

? Ejercicio 2

Relaciona cada característica al método de entrega TCP o UDP:

Característica	TCP	UDP
Establecimiento de sesión		
Sin conexión		
Entrega ordenada		
Entrega no ordenada		
Entrega garantizada		
Requisitos de transmisión rápida		
Segmentos de mensaje en secuencia		
Sin acuse de recibo		
Control de flujo		
Menos sobrecarga		

2.1.5. Direccionamiento de puertos TCP y UDP

Un equipo necesita 3 direcciones para conectarse a una red a través de los protocolos de Internet: dirección de datos (**MAC**), dirección de red (**IP**) y dirección de transporte (**puerto**).

En el encabezado de cada segmento o datagrama, hay un **puerto origen** y **uno de destino**. El número de puerto de origen es el número para esta comunicación asociado con la aplicación que origina la comunicación en el host local. Como se muestra en la ilustración, el número de puerto de destino es el número para esta comunicación relacionada con la aplicación de destino en el host remoto.



Cuando se envía un mensaje utilizando TCP o UDP, los protocolos y servicios solicitados se identifican con un número de puerto. Un puerto es un identificador numérico de cada segmento, que se utiliza para realizar un seguimiento de conversaciones específicas y de servicios de destino solicitados. Cada mensaje que envía un host contiene un puerto de origen y un puerto de destino.

a) Puerto de destino

El cliente coloca un número de puerto de destino en el segmento para informar al servidor de destino el servicio solicitado. Por ejemplo: el puerto 80 se refiere a HTTP o al servicio Web. Cuando un cliente especifica el puerto 80 en el puerto de destino, el servidor que recibe el mensaje sabe que se solicitan servicios Web. Un servidor puede ofrecer más de un servicio simultáneamente. Por ejemplo, puede ofrecer servicios Web en el puerto 80 al mismo tiempo que ofrece el establecimiento de una conexión FTP en el puerto 21.

b) Puerto de origen

El número de puerto de origen es generado de manera aleatoria por el dispositivo emisor para identificar una conversación entre dos dispositivos. Esto permite establecer varias conversaciones simultáneamente. En otras palabras, un dispositivo puede enviar varias solicitudes de servicio HTTP a un servidor Web al mismo tiempo. El seguimiento de las conversaciones por separado se basa en los puertos de origen.

Los puertos de origen y de destino se colocan dentro del segmento. Los segmentos se encapsulan dentro de un paquete IP. El paquete IP contiene la dirección IP de origen y de destino. El socket se utiliza para identificar el servidor y el servicio que solicita el cliente. Miles de hosts se comunican a diario con millones de servidores diferentes. Los sockets identifican esas comunicaciones.

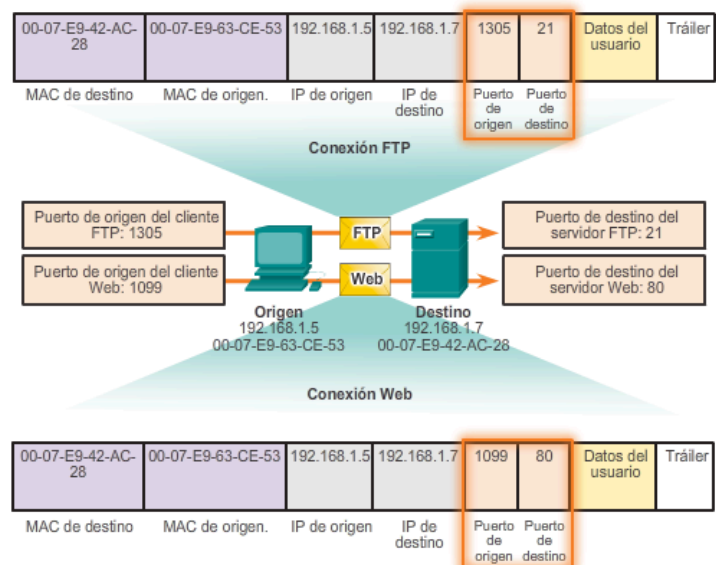
La combinación del número de puerto de la capa de transporte y de la dirección IP de la capa de red del host identifica de manera exclusiva un proceso de aplicación en particular que se ejecuta en un dispositivo host individual. Esta combinación se denomina socket. Un par de sockets, que consiste en las direcciones IP de origen y destino y los números de puertos, también es exclusivo e identifica la conversación específica entre los dos hosts.

Un socket de cliente puede ser parecido a esto, donde 1099 representa el número de puerto de origen: **192.168.1.5:1099**

El socket en un servidor Web podría ser el siguiente: **192.168.1.7:80**

Juntos, estos dos sockets se combinan para formar un par de sockets: **192.168.1.5:1099, 192.168.1.7:80**

Con la creación de sockets, se conocen los extremos de la comunicación, de modo que los datos puedan moverse desde una aplicación en un host hacia una aplicación en otro host. Los sockets permiten que los procesos múltiples que se ejecutan en un cliente se distingan entre sí. También permiten la diferenciación de múltiples conexiones a un proceso de servidor. El puerto de origen de la solicitud de un cliente se genera de manera aleatoria. El número de puerto actúa como dirección de retorno para la aplicación que realiza la solicitud. La capa de transporte hace un seguimiento de este puerto y de la aplicación que generó la solicitud de manera que cuando se devuelva una respuesta, esta se envíe a la aplicación correcta. El número de puerto de la aplicación que realiza la solicitud se utiliza como número de puerto de destino en la respuesta que vuelve del servidor.



2.1.5.1. Tipos de Puertos

La **Agencia de asignación de números por Internet (IANA)** asigna números de puerto. IANA es un organismo normativo responsable de asegurar diferentes estándares de direccionamiento.

Existen diferentes tipos de números de puerto:

- **Puertos bien conocidos (números del 0 al 1023):** estos números se reservan para servicios y aplicaciones. Se utilizan comúnmente para aplicaciones como HTTP (servidor Web), protocolo de acceso a mensajes de Internet (IMAP) o protocolo simple de transferencia de correo (SMTP) (servidor de correo electrónico) y Telnet. Al definir estos puertos bien conocidos para las aplicaciones de los servidores, las aplicaciones cliente se pueden programar para solicitar una conexión a ese puerto en particular y el servicio relacionado.

Nº de puerto	Protocolo	Servicio de Aplicación
20	TCP	FTP datos
21	TCP	FTP control
23	TCP	TELNET
25	TCP	SMTP
53	TCP/UDP	DNS
69	UDP	TFTP
80	TCP	HTTP
110	TCP	POP3
443	TCP	HTTPS

- **Puertos registrados (números del 1024 al 49.151):** estos números de puerto se asignan a procesos o aplicaciones del usuario. Principalmente, estos procesos son aplicaciones individuales que el usuario elige instalar en lugar de aplicaciones comunes que recibiría un número de puerto bien conocido. Cuando no se utilizan para un recurso del servidor, un cliente puede seleccionar estos puertos de forma dinámica como su puerto de origen.

Nº de puerto	Protocolo	Servicio de Aplicación
2049	TCP	NFS
8080	TCP/UDP	Puerto alternativo HTTP

- **Puertos dinámicos o privados (números 49.152 a 65.535):** también conocidos como puertos efímeros, generalmente se los asigna de forma dinámica a las aplicaciones cliente cuando el cliente inicia una conexión a un servicio. El puerto dinámico suele utilizarse para identificar la aplicación cliente durante la comunicación, mientras que el cliente utiliza el puerto bien conocido para identificar el servicio que se solicita en el servidor y conectarse a dicho servicio. No es común que un cliente se conecte a un servicio mediante un puerto dinámico o privado (aunque algunos programas de intercambio de archivos punto a punto lo hacen).

? Ejercicio 3

Completa la siguiente tabla:

Nº de puerto	Tipo de puerto	Protocolo	Servicio de aplicación
20	<i>Bien conocido</i>	<i>TCP</i>	<i>FTP de datos</i>
8080			
25			
53			
4662			

3. SOCKETS TCP

Dentro del paquete **java.net** nos encontramos con dos clases “**ServerSocket**” y “**Socket**” para trabajar con sockets TCP. Como ya sabemos, TCP es un protocolo orientado a conexión, por lo que para establecer una comunicación es necesario especificar una conexión entre un par de sockets. Uno de los sockets, el cliente, solicita una conexión, y el otro socket, el servidor, atiende las peticiones de los clientes. Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

3.1. Clase ServerSocket

La clase **ServerSocket** se utiliza para implementar el extremo de la conexión que corresponde al servidor, es el conector que escucha las peticiones de conexión de los clientes. Veamos algunos de los constructores de la clase:

CONSTRUCTOR	FUNCIÓN
ServerSocket()	Crea un socket de servidor sin ningún puerto asociado
ServerSocket(int port)	Crea un socket de servidor, que se enlaza al puerto especificado
ServerSocket(int port, int máximo)	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro máximo especifica el número máximo de peticiones de conexión que se pueden mantener en cola.

Algunos métodos importantes de **ServerSocket** son:

MÉTODOS	FUNCIÓN
Socket accept()	El método accept() escucha una solicitud de un cliente y la acepta cuando se recibe. <u>Una vez que se ha establecido la conexión con el cliente, devuelve un objeto de tipo Socket, a través del cual se establecerá la comunicación del cliente.</u> Tras esto, el ServerSocket sigue disponible para realizar nuevos accept(). Puede lanzar IOException.
close()	Se encarga de cerrar el ServerSocket.
int getLocalPort()	Devuelve el puerto local al que está enlazado el ServerSocket.

Pequeño ejemplo de cómo se crearía un socket de servidor (**ServerSocket**) y se obtendrían los **Socket** de comunicación con los clientes:

```
int puerto = 8000; //puerto
ServerSocket Servidor = new ServerSocket(puerto);
System.out.println("Escuchando en "+ Servidor.getLocalPort());

Socket cliente1 = Servidor.accept(); //Esperando a un cliente
//Realizar acciones con cliente1.

Socket cliente2 = Servidor.accept(); //Esperando a un segundo
cliente
//Realizar acciones con cliente2.

Servidor.close(); //cierro socket servidor.
```

3.2. Clase Socket

La clase Socket implementa un extremo de la conexión TCP. Algunos de sus constructores son (pueden lanzar la excepción IOException):

CONSTRUCTOR	FUNCIÓN
Socket()	Crea un socket sin ningún puerto asociado
Socket(InetAddress address, int port)	Crea un socket y lo conecta al puerto y dirección IP especificados
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Permite además especificar la dirección IP local y el puerto local a los que se asociará el socket.
Socket(String host, int port)	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar <code>UnknownHostException</code> , <code>IOException</code> .

Algunos métodos importantes son:

MÉTODOS	FUNCIÓN
InputStream getInputStream()	Devuelve un <code>InputStream</code> que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <code>IOException</code> .
OutputStream getOutputStream()	Devuelve un <code>OutputStream</code> que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <code>IOException</code> .
close()	Se encarga de cerrar el socket.
InetAddress getAddress()	Devuelve la dirección IP y el puerto a la que el socket está conectado. Si no está devuelve null.
int getLocalPort()	Devuelve el puerto local al que está enlazado el socket, -1 si no está enlazado a ningún puerto.

<code>int getPort()</code>	Devuelve el puerto remoto al que está conectado el socket, 0 si no está conectado a ningún puerto.
----------------------------	--

El siguiente ejemplo crea un socket cliente y lo conecta al host local al puerto 8000 (tiene que haber un **ServerSocket** escuchando en ese puerto). Después visualiza el puerto local al que está conectado el socket, y el puerto, host y dirección IP de la máquina remota a la que se conecta (en este caso es el host local):

```
String Host = "localhost";
int Puerto = 8000; //puerto
//abrir socket
Socket Cliente= new Socket(Host, Puerto); //conectamos

//hacer cosas con el socket cliente

InetAddress i = Cliente.getInetAddress();
System.out.println("Puerto local:  "+ Cliente.getLocalPort());
System.out.println("Puerto remoto:  "+ Cliente.getPort());
System.out.println("Nombre host remoto:  "+ i.getHostName());
System.out.println("IP host remoto:  "+ i.getHostAddress());
Cliente.close(); //cerramos el socket
```

La salida que se genera es la siguiente:

```
Puerto local:8784
Puerto remoto: 6000
Nombre host remoto: localhost
IP host remoto: 127.0.0.1
```

3.3. Gestión de sockets TCP

- El programa servidor crea un socket de servidor definiendo un puerto, mediante el método `ServerSocket(port)`, y espera mediante el método `accept()` a que el cliente solicite la conexión.
- Cuando el cliente solicita una conexión, el servidor abrirá la conexión al socket con el método `accept()`.
- El cliente establece una conexión con la máquina host a través del puerto especificado mediante el método `Socket(host,port)`.
- El cliente y el servidor se comunican con manejadores **InputStream** y **OutputStream**. El cliente escribe los mensajes en el **OutputStream** asociado al socket y el servidor leerá los mensajes del cliente de **InputStream**. Igualmente, el servidor escribirá sus mensajes para el cliente en un **OutputStream** y el cliente los leerá del **InputStream**.

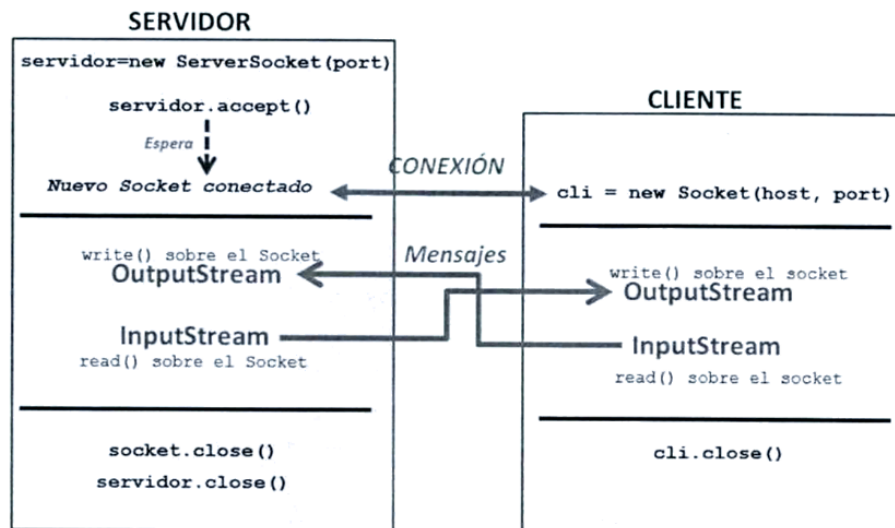


Figura 3.5. Modelo de Socket TCP.

3.3.1. Apertura de sockets

⇒ En el **programa servidor** se crea un **ServerSocket** invocando al método `ServerSocket(puerto)` en el que indicamos el número de puerto por el que el servidor escucha las peticiones de conexión de los clientes (cuidado con las excepciones).

```
ServerSocket servidor = null;
try{
    servidor = new ServerSocket(numeroPuerto);
}catch (IOException io){
    io.printStackTrace();
}
```

Necesitamos también crear un objeto **Socket** desde el **ServerSocket**. Para aceptar conexiones se usa el método `accept()`;

```
Socket clienteConectado = null;
try{
    clienteConectado = servidor.accept();
}catch (IOException io){
    io.printStackTrace();
}
```

⇒ En el **programa cliente** es necesario crear un objeto **Socket**; el socket se abre de la siguiente manera:

```
Socket cliente= null;
try{
    cliente = new Socket("máquina", numPuerto);
}catch (IOException io){
    io.printStackTrace();
}
```

Donde *"máquina"* es el nombre de la máquina a la que nos queremos conectar y *numPuerto* es el puerto por el que el programa servidor está escuchando las peticiones de los clientes.

3.3.2. Creación de streams de entrada

En el **programa servidor** podemos usar **DataInputStream** para recuperar los mensajes que el cliente escriba en el socket, previamente hay que usar el método `getInputStream()` para obtener el flujo de entrada del socket del cliente:

```
InputStream entrada = null;
try{
    entrada= clienteConectado.getInputStream();
}catch (IOException e){
    e.printStackTrace();
}
DataInputStream flujoEntrada = new DataInputStream(entrada);
```

En el programa Cliente podríamos realizar la misma operación para recibir los mensajes procedentes del programa servidor

Recuerda: La clase **DataInputStream** permite la lectura de líneas de texto y tipos primitivos de java. Algunos de sus métodos son: `readInt()`, `readDouble()`, `readLine()`, `readUTF()`, etc

3.3.3. Creación de streams de salida

En el programa cliente podemos usar **DataOutputStream** para escribir los mensajes que queremos que el servidor reciba, previamente hay que usar el método `getOutputStream()` para obtener el flujo de salida del socket del cliente:

```
OutputStream salidaCliente = null;
try{
    salidaCliente = cliente.getOutputStream();
}catch (IOException e){
    e.printStackTrace();
}
DataOutputStream flujoSalidaCliente = new
DataOutputStream(salidaCliente);
```

En el programa servidor podríamos realizar la misma operación para enviar mensajes al cliente.

Recuerda: La clase **DataOutputStream** permite la escritura de líneas de texto y tipos primitivos de java, Algunos de sus métodos son: `writeInt()`, `writeDouble()`, `writeBytes()`, `writeUTF()`, etc

3.3.4. Cierre de sockets

El orden de cierre de los sockets es relevante, primero cerramos los streams relacionados con el socket antes que el propio socket:

```
//Ejemplo de cierre para nuestro servidor que sólo tiene un flujo de
entrada
try{
    flujoSalida.close();
    flujoEntrada.close();
    clienteConectado.close();
    servidor.close();
} catch (IOException e){
    e.printStackTrace();
}
```

Ejemplo completo de Socket Servidor y Cliente:

```
package es.iesvjp;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {
    private static int puerto = 8000;
    private Socket cliente;
    private ServerSocket servidor;
    private DataInputStream flujoEntrada = null;
    private DataOutputStream flujoSalida = null;

    public static void main(String[] args) {

        Servidor servidor= new Servidor();
        servidor.setConexion();
        servidor.setMensajeCliente("Saludos al Cliente de parte del Servidor");
        servidor.getMensajeCliente();
        servidor.closeStreamsSockets();

    }

    public void setConexion() {

        try {
            servidor = new ServerSocket(puerto);
            System.out.println("Esperando conexión entrante en el puerto " + puerto +
"...");
            cliente = servidor.accept();
            System.out.println("Conexión establecida con: " +
                cliente.getInetAddress().getHostName() + "\n\n");
        } catch (IOException e) {

            System.out.println("Error en establecerConexion " + e.getMessage());
        }
    }
    /**
     * Establecemos un flujo de entrada para recibir mensajes desde el Cliente
     */
    public void getMensajeCliente()
```



```

{
    try {
        flujoEntrada= new DataInputStream(cliente.getInputStream());
        //El cliente me envía un mensaje
        System.out.println("Recibiendo del cliente: " + flujoEntrada.readUTF());
    } catch (IOException e) {
        System.out.println("setFlujoEntrada " + e.getMessage());
    }

}

/**
 * Creamos un flujo de salida hacia el Cliente
 * @param msj
 */
public void setMensajeCliente(String msj)
{
    try {
        flujoSalida= new DataOutputStream(cliente.getOutputStream());
        flujoSalida.writeUTF(msj);

    } catch (IOException e) {
        System.out.println("setFlujoSalida " + e.getMessage());
    }

}

/**
 * Cerramos los flujos de entrada/salida y los Sockets Cliente y Servidor
 */
public void closeStreamsSockets()
{
    try {
        flujoEntrada.close();
        flujoSalida.close();
        cliente.close();
        servidor.close();
    } catch (IOException e) {
        System.out.println("closeStreamSockets " + e.getMessage());
    }

}

}
}

```

```

package es.iesvjp;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

public class Cliente {
    private static int puerto = 8000;
    private Socket cliente;
    private DataInputStream flujoEntrada = null;
    private DataOutputStream flujoSalida = null;

    public static void main(String[] args) {

        Cliente cliente = new Cliente();
    }
}

```

```
        cliente.setConexion();
        cliente.getMensajeServidor();
        cliente.setMensajeServidor("Saludos al Servidor desde el Cliente ");
        cliente.closeStreamsSockets();
    }

    public void setConexion() {
        try {
            cliente = new Socket("localhost", puerto);
            System.out.println("Socket cliente iniciado...");

        } catch (IOException e) {

            System.out.println("Error en establecerConexion " + e.getMessage());
        }
    }

    /**
     * Establecemos un flujo de entrada para recibir mensajes desde el Servidor
     */
    public void getMensajeServidor() {
        try {
            flujoEntrada = new DataInputStream(cliente.getInputStream());
            // El servidor me envía un mensaje
            System.out.println("Recibiendo del servidor: " +
                               flujoEntrada.readUTF());
        } catch (IOException e) {
            System.out.println("setFlujoEntrada " + e.getMessage());
        }
    }

    /**
     * Creamos un flujo de salida hacia el Servidor
     *
     * @param msj
     */
    public void setMensajeServidor(String msj) {
        try {
            flujoSalida = new DataOutputStream(cliente.getOutputStream());
            flujoSalida.writeUTF(msj);

        } catch (IOException e) {
            System.out.println("setFlujoSalida " + e.getMessage());
        }
    }

    /**
     * Cerramos los flujos de entrada/salida y los Sockets Cliente y Servidor
     */
    public void closeStreamsSockets() {
        try {
            flujoEntrada.close();
            flujoSalida.close();
            cliente.close();

        } catch (IOException e) {
            System.out.println("closeStreamSockets " + e.getMessage());
        }
    }
}
```

}

Para probar los programas del Cliente y el Servidor, es necesario arrancar primero el Servidor sino nos dará el siguiente error si arrancamos el Cliente sin haber arrancado previamente el Servidor:

```
Error en establecerConexion Connection refused: connect
Exception in thread "main" java.lang.NullPointerException
    at es.iesvjp.acadt.Cliente.setFlujoSalida(Cliente.java:59)
    at es.iesvjp.acadt.Cliente.main(Cliente.java:19)
```

Una vez arrancado en el orden correcto los programas, la salida sería:

```
<terminated> Servidor [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

Recibiendo del cliente: Saludos al Servidor desde el Cliente
```

```
<terminated> Cliente [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
Socket cliente iniciado...
Recibiendo del servidor: Saludos al Cliente de parte del Servidor
```

Actividades

4. Crea una aplicación Servidor que envíe un mensaje a una aplicación Cliente y esta aplicación Cliente devolverá el mismo mensaje del Servidor, pero transformado todo en mayúsculas.

Modificación-por parejas: Una vez que hayas probado que funciona perfectamente la aplicación en local, prueba ahora con tu compañero que puedes realizar la actividad anterior ejecutando el programa cliente y el servidor en ordenadores diferentes.

5. Modifica el ejercicio anterior para que los mensajes ahora sean:
 - "PIM" (El cliente).
 - "PAM" (Responde el servidor)
 - "PUM" (Responde el cliente)
 - "FUEGO" (Responde el servidor)

Y cambia también para que haya un retraso de 3 segundos entre cada comunicación.

```
Console
<terminated> Servidor (1) [Java Application] C:\Program Files\Java\
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

[Servidor] recibe: PIM
[Servidor] envía: PAM
[Servidor] recibe: PUM
[Servidor] envía: FUEGO
```

```
Console
<terminated> Cliente (1) [Java Application] C:\Program Files
Socket cliente iniciado...
[Cliente] envía: PIM
[Cliente] recibe: PAM
[Cliente] envía: PUM
[Cliente] recibe: FUEGO
```

6. Crear un programa en el que haya un cliente y un servidor TCP. El servidor estará escuchando en el puerto 8000.

El cliente podrá enviar mensajes al servidor, estos mensajes serán escritos por la entrada estándar de teclado.

El servidor capturará dichos mensajes y simulará el eco de dichos mensajes, para ello, cambiará todas las letras de orden para que aparezcan de manera inversa y se lo enviará de nuevo al cliente, que lo mostrará por pantalla.



Nota:

- Intenta lograr la cadena transpuesta por ti mismo.
- Como el eco, nuestro programa no para nunca de trabajar.

```

Console
Servidor [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (20 ene. 2019 13:59:44)
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

Cliente [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (20 ene. 2019 13:59:44)
Socket cliente iniciado...

```

```

[Recibido]Hola
[Enviando]ALOH
[Recibido]roma ciudad del amor
[Enviando]ROMA LED DADUIC AMOR
[Recibido]La zorra comia arroz
[Enviando]ZORRA AIMOC ARROZ AL

```

Introduce una cadena, por favor:

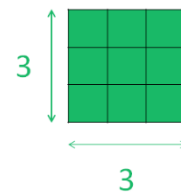
Hola

[Recibido]ALOH

Introduce una cadena,

roma ciudad del amor

[Recibido]ROMA LED DAI



7. Crea un programa en el que haya un cliente y un servidor TCP. El servidor estará escuchando en el puerto 8000.

El cliente podrá enviar números enteros y el servidor se dedicará a devolver el cuadrado del número entero que reciba.

Una vez envíe el cuadrado del número se cerrará la conexión.

Prueba que funciona, para ello el cliente capturará el número que le pase el usuario y luego lo mandará al servidor.

Introduce una cadena,

La zorra comia arroz

[Recibido]ZORRA AIMOC ARROZ AL

Área = $3^2 = 3 \times 3 = 9$

```

Problems Console
<terminated> Servidor (6) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (20 ene. 2019 13:59:44)
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

```

```

[Recibido]:8
[Enviando]:64

```

```

Problems Console
<terminated> Cliente (6) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (20 ene. 2019 13:59:44)
Socket cliente iniciado...
Introduce un número mayor que 0, para calcular su cuadrado
8
[Enviando]:8
[Recibiendo] Cuadro de 8=64

```

8. Vamos a crearnos un **chat por turnos**, para ello nos tenemos que crear un cliente y un servidor TCP. El servidor estará escuchando en el puerto 8000. El cliente y el servidor

podrán estar mandándose mensajes constantemente, pero por turnos. Es decir, primeramente, habla el cliente, luego el servidor, luego el cliente, luego el servidor...

Realiza el programa de tal modo que la comunicación pueda ser infinita, es decir nunca habrá un último turno de palabra.

Nota: no es necesario que haya una interfaz.

9. **Descubre el servidor (preguntar al profesor antes de realizar este ejercicio):** Hemos inicializado un servidor en la red del Instituto.

Dicho servidor realiza una tarea muy simple, escucha una entrada de texto (una cadena cualquiera) y responde con una frase.

Tu misión consistirá en realizar un programa que sea capaz de localizar dicho servidor, para ello debes descubrir cuál es su dirección IP y su puerto de conexión.

PISTA 1: El puerto está en el rango del 15000 al 20000.

PISTA 2: La dirección IP está dentro del rango "192.168.1XX.XXX".



IMPORTANTE:

Por defecto, el *timeout* de intento de conexión es muy alto, por lo que cada comprobación de conexión con un servidor se hace muy lenta. Para cambiar ese timeout, revisa la siguiente respuesta:

<http://stackoverflow.com/questions/5632279/how-to-set-timeout-on-client-socket-connection>

¡Intenta ser el primero en localizar el servidor para obtener la frase que devuelve!

3.4. Conexión de múltiples clientes. Hilos

Hasta ahora los programas servidores que hemos creado solo son capaces de atender a un cliente en cada momento, pero lo más típico es que un programa servidor pueda atender a muchos clientes simultáneamente. La solución para poder atender a múltiples clientes está en el **multihilo**, cada cliente será atendido en un hilo.

El esquema básico en sockets TCP sería construir un único servidor con la clase `ServerSocket` e invocar al método **`accept()`** para esperar las peticiones de conexión de los clientes. Cuando un cliente se conecta, el método **`accept()`** devuelve un objeto `Socket`, éste se usará para crear un hilo cuya misión es atender a este cliente. Después se vuelve a invocar a **`accept()`** para esperar a un nuevo cliente; habitualmente la espera de conexiones se hace dentro de un bucle infinito.

Vamos a realizar un programa en la que los clientes enviarán al servidor cadenas de texto introducidas por teclado; y éste le devolverá la misma cadena pero en mayúsculas. Cuando le envíe un asterisco el servidor finalizará la comunicación con el cliente.

```

package es.iesvjp.acadt;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Cliente {
    private static int puerto = 8000;

    public static void main(String[] args) {
        try {
            Socket cliente = new Socket("localhost", puerto);
            // creo flujo de salida al servidor
            PrintWriter flujoSalida = new PrintWriter(cliente.getOutputStream(), true);
            // creo flujo de entrada al servidor
            BufferedReader flujoEntrada = new BufferedReader(new InputStreamReader
                                                                (cliente.getInputStream()));

            BufferedReader teclado = new BufferedReader(new InputStreamReader
                                                        (System.in));

            String cadena, eco = "";
            System.out.println("Introduce cadena:...");
            cadena = teclado.readLine();
            while (!cadena.trim().equals("")) {
                flujoSalida.println(cadena); // envío cadena al servidor
                eco = flujoEntrada.readLine(); // recibo cadena del servidor
                System.out.println(" ==> ECO: " + eco);
                System.out.println("Introduce cadena: ");
                cadena = teclado.readLine();
            }
            flujoSalida.close();
            flujoEntrada.close();
            teclado.close();
            cliente.close();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                System.out.println(e.getMessage());
            }
        }
    }
}

```

```

package es.iesvjp.acadt;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

```

```

public class HiloServidor extends Thread {
    BufferedReader fentrada;
    PrintWriter fsalida;
    Socket socket = null;

    public HiloServidor(Socket socket) {
        this.socket = socket;
        try {
            // inicializamos los flujos de entrada y de salida
            fsalida = new PrintWriter(socket.getOutputStream(), true);
            fentrada = new BufferedReader(new InputStreamReader
                                         (socket.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace(); }
    }

    @Override
    public void run() {
        // tarea a realizar con cada cliente
        try {
            String cadena = "";
            while (cadena!=null && !cadena.trim().equals("")) {
                System.out.println("COMUNICO CON: " + socket.toString());
                cadena = fentrada.readLine();// obtenemos la cadena
                if (cadena != null) {
                    // enviar mayúscula
                    fsalida.println(cadena.trim().toUpperCase());
                }

                System.out.println("FIN CON: " + socket.toString());
                fsalida.close();
                fentrada.close();
                socket.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                System.out.println(e.getMessage());
            }
        }
    }
}

```

```

package es.iesvjp.acadt;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {
    private static int puerto = 8000;

    public static void main(String[] args) {

        try {
            ServerSocket servidor = new ServerSocket(puerto);
            System.out.println("Servidor iniciado ... ");
            while (true) {
                Socket cliente = new Socket();
                cliente = servidor.accept();// esperando cliente
            }
        }
    }
}

```

```

        HiloServidor hilo = new HiloServidor(cliente);
        hilo.start(); // se atiende al cliente
    }
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
}

```

En las siguientes capturas de pantalla se muestra un momento de la ejecución, primero se ejecuta el programa servidor y a continuación el programa cliente; se puede observar cómo los 2 clientes conectados están siendo atendidos por el servidor de forma simultánea.

The screenshot shows two console windows. The left window, titled 'Servidor (3) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (23 dic. 2018)', shows the server starting and receiving multiple connections. The right window, titled '<terminated> Cliente (3) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe', shows a client interacting with the server, sending messages like 'ola soy el cliente 1' and 'ola soy el cliente 2'.

```

Servidor (3) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (23 dic. 2018)
Servidor iniciado ...
COMUNICO CON: Socket[addr=/127.0.0.1,port=53289,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53289,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53290,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53290,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53289,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53289,localport=8000]
FIN CON: Socket[addr=/127.0.0.1,port=53289,localport=8000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=53290,localport=8000]
FIN CON: Socket[addr=/127.0.0.1,port=53290,localport=8000]

<terminated> Cliente (3) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
Introduce cadena:...
ola soy el cliente 1
==> ECO: HOLA SOY EL CLIENTE 1
Introduce cadena:
n, dos, tres
==> ECO: UN, DOS, TRES
Introduce cadena:
robando, probando
==> ECO: PROBANDO, PROBANDO
Introduce cadena:
*

```

Actividades

- Modifica el ejercicio 8 para que no haya turnos al hablar y escribir, es decir, una parte de la comunicación puede hablar con la otra parte tantas veces como desee.

La comunicación nunca termina.

Pista: crea una clase que extienda de *Thread* para recibir y otra para enviar

The screenshot shows two console windows. The left window, titled 'Servidor (2) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (11 dic. 2018 14:16:11)', shows the server waiting for a connection and then receiving a connection from 127.0.0.1. The right window, titled '<terminated> Cliente (2) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe', shows a client sending messages like 'Hola Pocoyo!!' and 'Muy bien, de vacaciones de Navidad'.

```

Servidor (2) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (11 dic. 2018 14:16:11)
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

[ELI] Hola Pocoyo!!
Hola Eli!!¿qué tal estás?
[ELI] Muy bien, de vacaciones de Navidad

<terminated> Cliente (2) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
Socket cliente iniciado...
Hola Pocoyo!!
[POCOYO] Hola Eli!!¿qué tal estás?
Muy bien, de vacaciones de Navidad

```

- Debemos diseñar para nuestra red del Instituto un Servidor Central que proporcione el servicio DayTime, es decir, fecha actual del sistema a través del puerto 13. Cualquier estación Cliente que desee consultar la fecha actual del sistema solo será necesario establecer una conexión con el Servidor Central y puerto indicado, a lo que el servicio

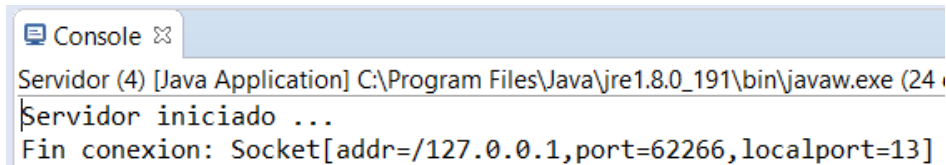
The screenshot shows a console window titled '<terminated> Cliente (4) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe'. It shows a client receiving the system date: '==> Fecha del Sistema: 2018-12-24T14:24:16.742'.

```

<terminated> Cliente (4) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
==> Fecha del Sistema: 2018-12-24T14:24:16.742

```


responde enviando una cadena de caracteres con la fecha y hora del sistema y cierra la conexión.



```

Servidor (4) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (24
Servidor iniciado ...
Fin conexion: Socket[addr=/127.0.0.1,port=62266,localport=13]

```

3.5. Envío de Objetos a través de sockets

Hasta ahora hemos estado intercambiando cadenas de caracteres entre programas cliente y datos primitivos, caracteres localizados y objetos.

En este apartado veremos cómo se pueden intercambiar objetos entre programas emisor y receptor o entre programas cliente y servidor usando sockets.

3.5.1. Objetos en Sockets TCP

Las clases `ObjectInputStream` y `ObjectOutputStream` nos permiten enviar objetos a través de sockets TCP. Utilizaremos los métodos `readObject()` para leer el objeto del stream y `writeObject()` para escribir el objeto al stream. Usaremos el constructor que admite un `InputStream` y un `OutputStream`. Para preparar el flujo de salida para escribir objetos escribimos:

```

ObjectOutputStream outObjeto = new ObjectOutputStream
(socket.getOutputStream());

```

Para preparar el flujo de entrada para leer objetos escribimos:

```

ObjectInputStream inObjeto = new ObjectInputStream
(socket.getInputStream());

```

Las clases a las que pertenecen estos objetos deben implementar la interfaz `Serializable`. Por ejemplo, sea la clase `Persona` con 2 atributos, nombre y edad, 2 constructores y los métodos `get` y `set` correspondientes:

```

package es.iesvjp.acadt;

import java.io.Serializable;

public class Persona implements Serializable {
    private static final long serialVersionUID = 6622024139761433636L;
    String nombre;
    int edad;

    public Persona() {}

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {

```

```

        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

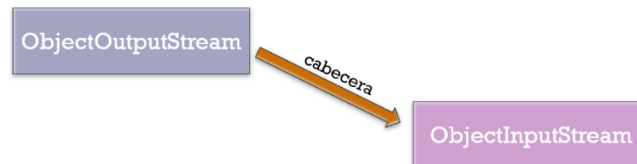
    public void setEdad(int edad) {
        this.edad = edad;
    }

@Override
    public String toString() {
        return "Persona [nombre=" + nombre + ", edad=" + edad + "]";
    }
}

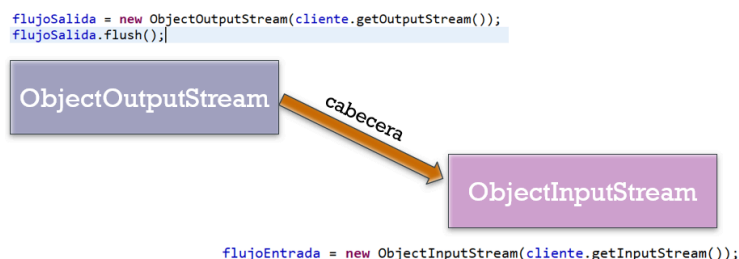
```

Para el correcto funcionamiento de los flujos de objetos tenemos que tener en cuenta ciertos puntos:

- 1) Los **ObjectOutputStream** de ambos puntos de la comunicación se tienen que abrir antes que los **ObjectInputStream**. ¿Por qué? Porque la creación del **ObjectInputStream** es bloqueante, está esperando una cabecera que se recibe cuando se abre el **ObjectOutputStream** correspondiente en el lado opuesto de la comunicación.



- 2) Como es necesario leer esas cabeceras, es una buena idea realizar un **flush()** una vez se crea el **ObjectOutputStream**. Flush vacía el buffer, por lo tanto, fuerza a enviar el flujo de datos desde un punto al otro, en este caso a enviar la cabecera.



- 3) En los **ObjectOutputStream** se vuelve a enviar el mismo objeto que se enviaba anteriormente, pese a que estoy instanciando un objeto nuevo y enviándolo. Problema, en el **ObjectOutputStream** guarda referencia de los objetos y aunque tú los

cambios y pienses que los estás enviando correctamente, se mantiene la referencia anterior del objeto.

La solución es llamar a **ObjectOutputStream.reset()**. Que fuerza a eliminar las referencias antiguas que tuviese el ObjectOutputStream del objeto que vas a enviar. En resumen, es necesario llamar a reset() si estás enviando el mismo objeto una y otra vez por los streams.

```
flujoSalida.writeObject(persona);  
flujoSalida.flush();  
flujoSalida.reset();
```

Ejemplo 3:

Vamos a intercambiar objetos *Persona* entre un cliente y un servidor usando sockets TCP. Por ejemplo, el programa servidor crea un objeto *Persona*, dándole valores y se lo envía al programa cliente, el programa cliente realiza los cambios oportunos en el objeto y se lo devuelve modificado al servidor. La clase Servidor es la siguiente:

```
package es.iesvjp.acadt;  
  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class ServidorObjeto {  
    private static int puerto = 8000;  
    private Socket cliente;  
    private ServerSocket servidor;  
    private ObjectInputStream flujoEntrada = null;  
    private ObjectOutputStream flujoSalida = null;  
  
    public static void main(String[] args) {  
  
        ServidorObjeto servidor= new ServidorObjeto();  
        servidor.setConexion();  
        //nos creamos una persona Pepe de edad 30 y se lo enviamos al Cliente  
        Persona persona=new Persona("Pepe", 30);  
        System.out.println("[Servidor enviando:]" + persona);  
        servidor.enviarMensajeCliente(persona);  
        //recibimos del cliente la persona  
        Persona person=servidor.recibirMensajeCliente();  
        System.out.println("[Servidor recibiendo:" + person);  
        servidor.closeStreamsSockets();  
  
    }  
  
    public void setConexion() {  
  
        try {  
            servidor = new ServerSocket(puerto);
```

```

        System.out.println("Esperando conexión entrante en el puerto " + puerto + "...");

        cliente = servidor.accept();
        System.out.println("Conexión establecida con: " +
                           cliente.getInetAddress().getHostName() + "\n\n");
    } catch (IOException e) {

        System.out.println("Error en establecerConexion " + e.getMessage());
    }
}
/**
 * Establecemos un flujo de entrada para recibir mensajes desde el Cliente
 */
public Persona recibirMensajeCliente() {
    Persona person=null;
    try {
        flujoEntrada = new ObjectInputStream(cliente.getInputStream());
        // El cliente nos envía un objeto
        person = (Persona) flujoEntrada.readObject();

    } catch (IOException | ClassNotFoundException e) {
        System.out.println("getMensajeServidor " + e.getMessage());
    }
    return person;
}
/**
 * Creamos un flujo de salida hacia el Cliente
 * @param persona
 */
public void enviarMensajeCliente(Persona persona) {
    try {

        flujoSalida = new ObjectOutputStream(cliente.getOutputStream());
        flujoSalida.writeObject(persona);
        flujoSalida.flush();
        flujoSalida.reset();

    } catch (IOException e) {
        System.out.println("enviarMensajeCliente " + e.getMessage());
    }
}
/**
 * Cerramos los flujos de entrada/salida y los Sockets Cliente y Servidor
 */
public void closeStreamsSockets()
{
    try {
        flujoEntrada.close();
        flujoSalida.close();
        cliente.close();
        servidor.close();
    } catch (IOException e) {
        System.out.println("closeStreamSockets "+ e.getMessage());
    }
}
}
package es.iesvjp.acadt;

import java.io.IOException;
import java.io.ObjectInputStream;

```

```
import java.io.ObjectOutputStream;
import java.net.Socket;

public class ClienteObjeto {
    private static int puerto = 8000;
    private Socket cliente;
    private ObjectInputStream flujoEntrada = null;
    private ObjectOutputStream flujoSalida = null;

    public static void main(String[] args) {

        ClienteObjeto cliente = new ClienteObjeto();
        cliente.setConexion();
        //obtenemos la persona que nos envía el servidor y cambiamos lo datos
        Persona person=cliente.recibirMensajeServidor();
        System.out.println("[Cliente recibiendo]:"+ person);
        person.setEdad(60);
        person.setNombre("Pepe");
        System.out.println("[Cliente enviando]:"+ person);
        //una vez modificada la persona la volvemos a enviar
        cliente.enviarMensajeServidor(person);
        cliente.closeStreamsSockets();

    }

    public void setConexion() {

        try {
            cliente = new Socket("localhost", puerto);

            System.out.println("Socket cliente iniciado...");

        } catch (IOException e) {

            System.out.println("Error en establecerConexion " + e.getMessage());
        }

    }

    /**
     * Establecemos un flujo de entrada para recibir mensajes desde el Servidor
     */
    public Persona recibirMensajeServidor() {
        Persona person=null;
        try {
            flujoEntrada = new
ObjectInputStream(cliente.getInputStream());
            // El servidor nos envía un objeto
            person = (Persona) flujoEntrada.readObject();

        } catch (IOException | ClassNotFoundException e) {
            System.out.println("getMensajeServidor " + e.getMessage());
        }
        return person;
    }

    /**
     * Creamos un flujo de salida hacia el Servidor
     */
}
```

```

    * @param persona
    */
    public void enviarMensajeServidor(Persona persona) {
        try {

            flujoSalida = new ObjectOutputStream(cliente.getOutputStream());
            flujoSalida.writeObject(persona);
            flujoSalida.flush();
            flujoSalida.reset();

        } catch (IOException e) {
            System.out.println("setFlujoSalida " + e.getMessage());
        }

    }

    /**
     * Cerramos los flujos de entrada/salida y los Sockets Cliente y Servidor
     */
    public void closeStreamsSockets() {
        try {
            flujoEntrada.close();
            flujoSalida.close();
            cliente.close();

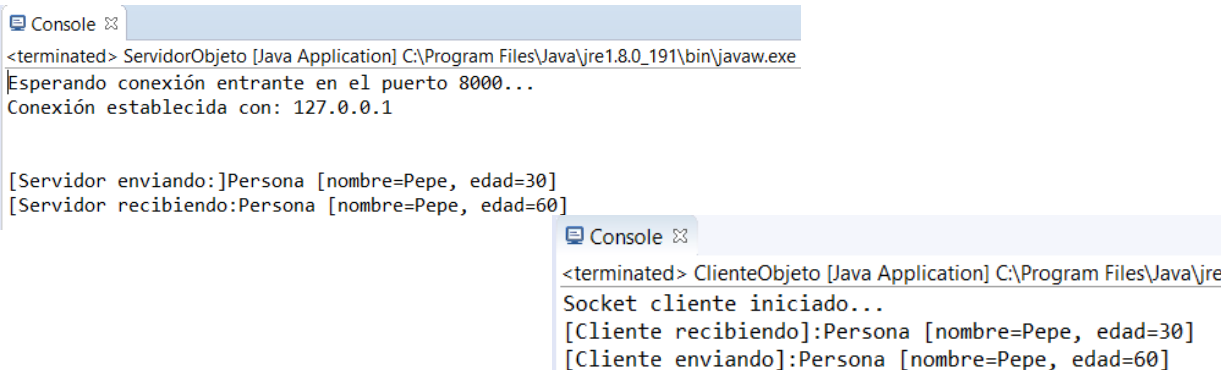
        } catch (IOException e) {
            System.out.println("closeStreamSockets " + e.getMessage());
        }

    }

}

```

Ejemplo de ejecución del programa Servidor y del Cliente:



```

Console
<terminated> ServidorObjeto [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1

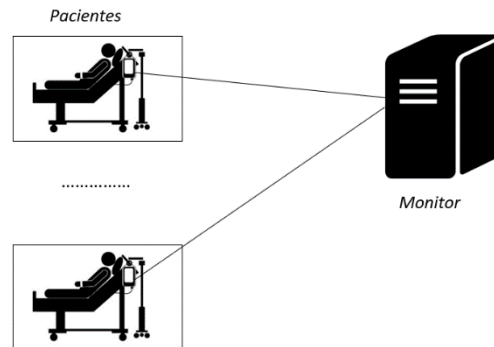
[Servidor enviando:]Persona [nombre=Pepe, edad=30]
[Servidor recibiendo:]Persona [nombre=Pepe, edad=60]

Console
<terminated> ClienteObjeto [Java Application] C:\Program Files\Java\jre
Socket cliente iniciado...
[Cliente recibiendo:]Persona [nombre=Pepe, edad=30]
[Cliente enviando:]Persona [nombre=Pepe, edad=60]

```

Actividades

12. Un hospital desea construir una aplicación que le permita monitorizar las constantes vitales de los pacientes. Para ello dispone de un sistema como el que se muestra en la siguiente figura.



Cada paciente está conectado a un computador que toma cada 10 sg la temperatura (37º-42º), la frecuencia cardíaca (60-100), la presión arterial sistólica (120-140) y la presión arterial diastólica (70-90) de dicho paciente.

El funcionamiento de la aplicación es el siguiente: cuando llega un paciente a la habitación se le conectan los aparatos de medición al cuerpo y estos al computador; una vez está todo conectado el computador comienza a registrar sus constantes vitales y cada 10 sg envía dicha información al monitor. El monitor recibirá cada 10sg la información de las constantes vitales de todos los pacientes conectados, si alguna de estos valores es igual o superior al máximo mandará un mensaje de aviso al computador del respectivo paciente sobre el valor o los valores alarmantes, para que los auxiliares del hospital acudan en su ayuda.

***** PELIGRO, los siguientes valores son muy altos: [TEMPERATURA: XX] [PULSO: XX] [SISTOLICA: XX] [DIASTOLICA: XX]

Cuando el paciente abandona la habitación, el computador se desconecta

Recomendaciones:

- Para poder almacenar en una única variable varios valores (temperatura alta, pulso alto, ...) utiliza enteros y la operación AND (&) a nivel de bit.
- Tendrás que utilizar hilos: para gestionar todos los clientes que se conectan al servidor, para enviar las constantes vitales al servidor y recibir avisos de alerta.
- Utiliza una clase *ConstantesVitales* para guardar todos los valores.
- Para simular la medición de las constantes vitales, genera de manera aleatoria los valores de cada paciente teniendo en cuenta los valores mínimos y máximos proporcionados.

```

Console
Servidor (5) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (26 dic. 2018 17:20:33)
Servidor iniciado ...
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=39.86, pulso=89.02, presion_sistolica=134.64, presion_diastolica=92.73]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=40.59, pulso=61.66, presion_sistolica=125.84, presion_diastolica=81.27]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=38.5, pulso=64.0, presion_sistolica=139.94, presion_diastolica=94.05]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=36.02, pulso=93.17, presion_sistolica=148.41, presion_diastolica=95.39]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=37.21, pulso=97.36, presion_sistolica=140.54, presion_diastolica=74.69]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=36.62, pulso=68.14, presion_sistolica=136.65, presion_diastolica=73.86]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=41.18, pulso=98.07, presion_sistolica=132.35, presion_diastolica=72.32]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=36.74, pulso=106.67, presion_sistolica=140.16, presion_diastolica=95.34]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=40.46, pulso=69.48, presion_sistolica=130.41, presion_diastolica=91.18]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=37.53, pulso=77.44, presion_sistolica=131.88, presion_diastolica=79.0]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54786,localport=8000]
ConstantesVitales [temperatura=35.67, pulso=77.29, presion_sistolica=139.03, presion_diastolica=77.6]
Recibo del cliente: Socket[addr=/127.0.0.1,port=54787,localport=8000]
ConstantesVitales [temperatura=37.59, pulso=72.56, presion_sistolica=139.53, presion_diastolica=81.43]

```

Ejemplo de avisos a los computadores de los pacientes con valores alarmantes (se han manipulado para que se salgan de los rangos normales):

```
Console
Cliente (5) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (26 dic. 2018 17:20:39)
***** PELIGRO, los siguientes valores son muy altos: [DIASTOLICA:92.73]
***** PELIGRO, los siguientes valores son muy altos: [DIASTOLICA:94.05]
***** PELIGRO, los siguientes valores son muy altos: [SISTOLICA:140.54]
***** PELIGRO, los siguientes valores son muy altos: [DIASTOLICA:91.18]
***** PELIGRO, los siguientes valores son muy altos: [PULSO:105.25] [SISTOLICA:140.79]
```

```
Console
Cliente (5) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (26 dic. 2018 17:20:47)
***** PELIGRO, los siguientes valores son muy altos: [SISTOLICA:148.41] [DIASTOLICA:95.39]
***** PELIGRO, los siguientes valores son muy altos: [PULSO:106.67] [SISTOLICA:140.16] [DIASTOLICA:95.34]
***** PELIGRO, los siguientes valores son muy altos: [SISTOLICA:146.07] [DIASTOLICA:96.71]
```

13. Crea una aplicación cliente-servidor que permita llevar el recuento de las personas que se apunten a un evento.

El servidor, tendrá un arrayList de personas que se rellena con las personas que se han apuntado a un determinado evento.



El servidor permitirá:

- Recibir solicitudes para apuntar gente al evento.
- Recibir solicitudes sobre la gente que está apuntada en el evento, en cuyo caso se enviará todos los que están apuntados.

Los clientes pueden:

- Enviar personas para apuntarlas al evento.
- Pedir información sobre las personas apuntadas en el evento. Todas esas personas se listarán por pantalla.

IMPORTANTE: Puedes crear el sistema de comunicación que creas más conveniente, pero es necesario documentar correctamente dicho sistema, justificando el porqué de cada decisión tomada.

Nosotros nos hemos creado las siguientes clases para nuestro sistema de comunicación:

Clase Persona

Variables:

- String DNI
- String nombre
- String apellidos
- int teléfono

- String email

Métodos:

- Implementa serializable
- Los get y set.

Clase Petición

Variables:

- int peticion; //Indica el mensaje de envío. Un 0 significa que solicitas información sobre las personas apuntadas. Un 1 significa que añades una persona.
- Persona persona; //La persona que subes, en caso que la opción sea 0, el valor de esta variable será indiferente.

Métodos:

- Implementa serializable.
- Los get y set.

Clase Respuesta

Variables:

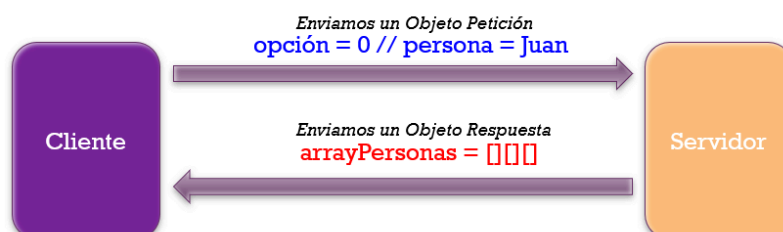
- MiArray arrayPersonas; //Un arrayList de las personas que se han incluido en el evento.

Métodos:

- Implementa serializable.
- Método para mostrar todas las personas inscritas en el evento
- Método para comprobar si una determinada persona está inscrita en el evento
- Los get y set.

Lado Cliente:

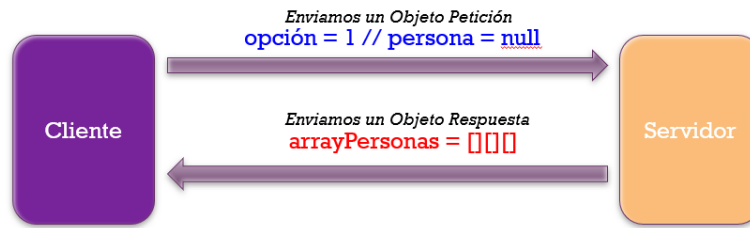
opción 1: inscripción de una persona al evento



opción 2: mostrar listado de personas inscritas al evento

```
Console
Cliente (8) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (3 feb. 2019 12:43:05)
Socket cliente iniciado...

¿Qué deseas hacer:
1)Inscribir a una persona al evento
```



```

¿Qué deseas hacer:
1)Inscribir a una persona al evento
2)Mostrar el listado de personas inscritas
3)Salir.
2
Solicitud de listado completada.
El listado de personas es el siguiente:
-----
Persona [nombre=Sergio, apellidos=Garcia Vaquero, telefono=927456789, email=sergio_vaquero@gmail.c
Persona [nombre=Pepe, apellidos=Gotera, telefono=926453445, email=pepe_gotera@gmail.com]
-----

¿Qué deseas hacer:
1)Inscribir a una persona al evento
2)Mostrar el listado de personas inscritas
3)Salir.
  
```

Lado Servidor:

```

Esperando conexión entrante en el puerto 8000...
Conexión establecida con: 127.0.0.1
  
```

```

Nuevo inscrito. La lista queda:
[Persona [nombre=Sergio, apellidos=Garcia Vaquero, telefono=927456789, email=sergio_vaquero@gmail.com]]
  
```

```

Nuevo inscrito. La lista queda:
[Persona [nombre=Sergio, apellidos=Garcia Vaquero, telefono=927456789, email=sergio_vaquero@gmail.com],
Persona [nombre=Pepe, apellidos=Gotera, telefono=926453445, email=pepe_gotera@gmail.com]]
  
```

4. CLASE INETADDRESS

La clase **InetAddress** es la abstracción que representa una dirección IP. Tiene dos subclases: *Inet4Address* para direcciones IPv4 e *Inet6Address* para IPv6. En la mayoría de los casos **InetAddress** aporta la funcionalidad necesaria y no es necesario recurrir a ellas.

Principalmente se utiliza para obtener un “nombre de dominio” de una dirección IP y viceversa.

MÉTODOS	FUNCIÓN
InetAddress <code>getLocalHost()</code> [Método Static]	Devuelve un objeto <i>InetAddress</i> que representa la dirección IP de la máquina donde se está ejecutando el programa.
InetAddress <code>getByName(String host)</code> [Método Static]	Devuelve un objeto <i>InetAddress</i> que representa la dirección IP de la máquina que se especifica como parámetro (<i>host</i>). Este parámetro puede ser el nombre de la máquina, nombre de dominio o una dirección IP.
InetAddress <code>getAllByName(String host)</code> [Método Static]	Realiza la misma funcionalidad que <code>getByName</code> pero devuelve todas las IPs asociadas a un host en particular.
String <code>getHostAddress()</code>	Devuelve la dirección IP de un objeto <i>InetAddress</i> en forma de cadena.
String <code>getHostName()</code>	Devuelve el nombre del host de un objeto <i>InetAddress</i>
String <code>getCanonicalHostName()</code>	Devuelve el nombre real (completo el FQDN).

Ejemplo 4. Vamos a realizar una prueba de la clase *InetAddress* para ver cómo funciona. Para ello creamos un proyecto nuevo con una clase nueva.

El funcionamiento del programa va a ser el siguiente:

- 1) Se le va a pedir al usuario que introduzca un nombre de dominio o una dirección IP:
- 2) Se va a mostrar la información asociada a esa *InetAddress*:
 - 1) Dirección IP.
 - 2) Nombre de Host.
 - 3) Nombre canónico.
- 4) Se van a mostrar todas las IP's diferentes para dicho host.

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

public class PruebaInetAddress {

    public static void main(String[] args) {
        InetAddress inetAddress = null;
        try {
            inetAddress = InetAddress.getByName(pedirCadena("Introduzca un
                nombre de máquina ó una dirección IP:"));
            mostrarInfoInetAddress(inetAddress);
        } catch (UnknownHostException e) {
```

```
        System.out.println("Host no válido");
    }
}

public static String pedirCadena(String msj) {
    Scanner teclado = new Scanner(System.in);
    System.out.println(msj);
    return teclado.nextLine();
}

public static void mostrarInfoInetAddress(InetAddress inetAddress) {
    System.out.println("----- Información para: \"" +
        inetAddress.getHostByName() + "\" -----");
    System.out.print("- Dirección IP:");
    System.out.println(inetAddress.getHostAddress());
    System.out.print("- Nombre del host:");
    System.out.println(inetAddress.getHostByName());
    System.out.print("- Nombre canónico:");
    System.out.println(inetAddress.getCanonicalHostName());

    System.out.println("Las direcciones asociadas a este host son:");

    InetAddress[] direccionesAsociadas = null;
    try {
        direccionesAsociadas = InetAddress.getAllByName(inetAddress.getHostByName());
    } catch (UnknownHostException e) {
        System.out.println("Error al intentar sacar las IPs del host.");
    }

    for (int i = 0; i < direccionesAsociadas.length; i++) {
        System.out.print("\t\tDirección IP:");
        System.out.println(direccionesAsociadas[i].getHostAddress());
    }
}
}
```

Ejemplo de ejecución con `iesvalledeljertepla.educarex.es`:

```
Introduzca un nombre de máquina ó una dirección IP:
iesvallejertepla.educarex.es
|---- Información para: "iesvallejertepla.educarex.es" ----
- Dirección IP:178.255.108.26
- Nombre del host:iesvallejertepla.educarex.es
- Nombre canónico:178.255.108.26
Las direcciones asociadas a este host son:
Dirección IP:178.255.108.26
```

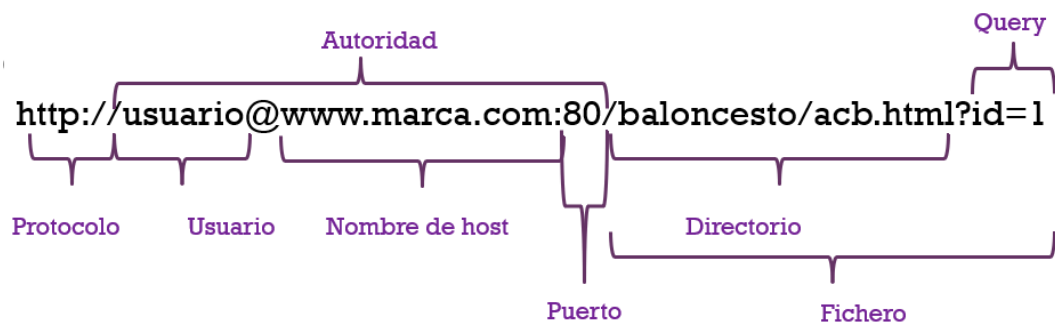
Prueba también los siguientes ejemplos:

- `pc1010.valencia.admon.com`
- `google-public-dns-b.google.com`

5. CLASE URL

La clase **URL** (Uniform Resource Locator) representa un puntero a un recurso en la Web. Un recurso puede hacer referencia a algo tan simple como un fichero o un directorio (suele ser lo habitual cuando accedemos a páginas web), pero puede ser algo más complicado, por ejemplo, una consulta a una base de datos o a un motor de búsqueda.

Una URL se divide generalmente en diferentes partes:




Los constructores más importantes para la clase URL son los siguientes:

CONSTRUCTOR	FUNCIÓN
<code>URL (String url)</code>	Crea un objeto URL a partir del String url
<code>URL(String protocolo, String host, String fichero)</code>	Crea un objeto URL a partir de los parámetros protocolo, host y fichero
<code>URL(String protocolo, String host, int puerto, String fichero)</code>	Crea un objeto URL en el que se especifica el protocolo, host, puerto y ficheros representados.

Algunos métodos importantes para la clase URL son:

MÉTODOS	MISIÓN
<code>String getAuthority()</code>	Obtiene la autoridad del objeto URL. La autoridad es: datos usuario + nombre de host+puerto
<code>int getPort()</code>	Devuelve el número de puerto de la URL, -1 si no se indica
<code>String getHost()</code>	Devuelve el nombre de la máquina
<code>String getQuery()</code>	Devuelve la cadena que se envía a una página para ser procesada (es lo que sigue al signo ? del a URL)
<code>String getPath()</code>	Devuelve una cadena con la ruta hacia el fichero desde el servidor y el nombre completo del fichero
<code>String getFile()</code>	Devuelve lo mismo que <code>getPath()</code> , además de la concatenación del valor de <code>getQuery()</code> si lo hubiese.
<code>String getUserInfo()</code>	Devuelve la parte con los datos del usuario de la dirección URL o nulo si no existe
<code>InputStream openStream()</code>	Devuelve un objeto <code>InputStream</code> con el cual podemos leer el contenido de una URL.

? Actividades

 14. Crea una clase que pruebe la funcionalidad de URL. Para ello le pediremos al usuario que introduzca una URL completa ó por partes (protocolo, host, puerto y fichero).

Una vez hayamos inicializado la URL, se obtendrá:

- 1) Protocolo
- 2) Nombre de la máquina
- 3) Puerto
- 4) Fichero
- 5) Información del usuario
- 6) El "path" (la ruta del fichero en el servidor).
- 7) La autoridad de la URL.
- 8) La query

NOTA: Prueba con la url:
<http://jesus@stackoverflow.com:80/questions/9401647/how-to-parse-user-credentials-from-url-in-c?id=24>


Ejemplo de ejecución:

```

Elige una opción:
1) URL completa
2) URL parcial
1
Introduce la URL completa:
http://jesus@stackoverflow.com:80/questions/9401647/how-to-parse-user-credentials-from-url-in-c?id=24
--- Información sobre la url: "http://jesus@stackoverflow.com:80/questions/9401647/how-to-parse-user-credential
Protocolo:
    http
Host
    stackoverflow.com
Puerto
    80
Fichero
    /questions/9401647/how-to-parse-user-credentials-from-url-in-c?id=24
Información de usuario
    jesus
Path
    /questions/9401647/how-to-parse-user-credentials-from-url-in-c
Autoridad
    jesus@stackoverflow.com:80
Query
    id=24

Elige una opción:
1) URL completa
2) URL parcial
2
Introduce protocolo:
http
Introduce host:
stackoverflow.com
Introduce puerto:
80
Introduce fichero:
questions/9401647/how-to-parse-user-credentials-from-url-in-c?id=24
|--- Información sobre la url: "http://stackoverflow.com:80questions/9401647/how-to-parse-user-crede
Protocolo:
    http
Host
    stackoverflow.com
Puerto
    80
Fichero
    questions/9401647/how-to-parse-user-credentials-from-url-in-c?id=24
Información de usuario
    null
Path
    questions/9401647/how-to-parse-user-credentials-from-url-in-c
Autoridad
    stackoverflow.com:80
Query
    id=24

```

 **15. El navegador más simple del mundo.** En esta actividad intentaremos recrear el navegador de páginas web más sencillo posible, para que los mayores de nuestra ciudad puedan utilizar sus dispositivos móviles.

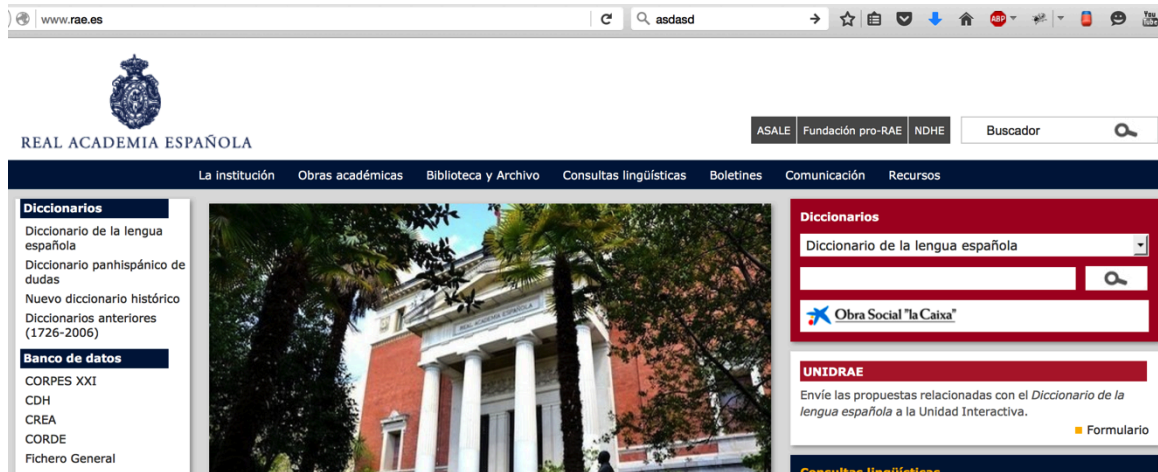
¿Qué hará nuestro navegador?:

- Le pedirá al usuario una URL por teclado.
- Abrirá esa URL y extraerá todo el texto de la página web (consulta el método `openStream` en <http://lineadecodigo.com/java/leer-una-url-con-java/>)
- Mostrará por pantalla solamente la cadena que sea el título de la página web.

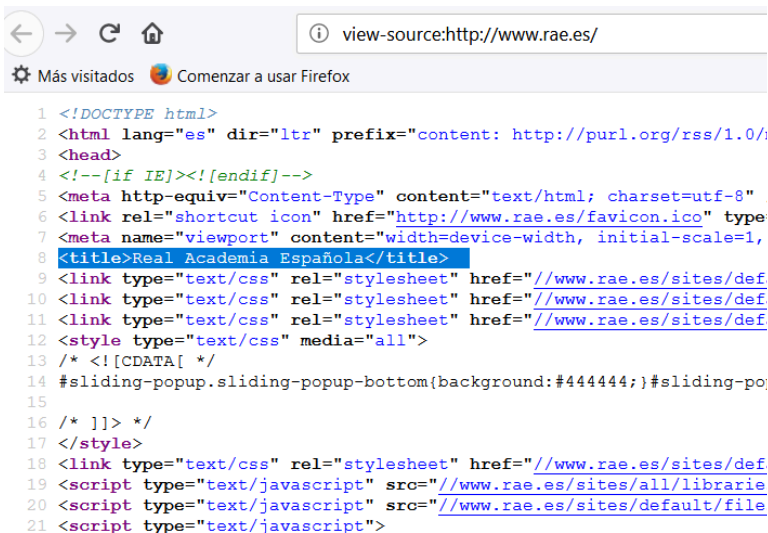


Ejemplo de funcionamiento:

La página de la RAE:



Código fuente:



Dentro del código fuente podemos apreciar cómo nos quedamos con la parte que nos interesa, que es el título de la página web.

Recomendaciones:

Para buscar unas determinadas cadenas podemos utilizar expresiones regulares (clases `Pattern` y `Matcher`) del paquete `java.util.regex`.

15b. ¿Funciona nuestro navegador simple con la siguiente página?: <http://www.marca.es>. Si no es así...Identifica por qué no está funcionando. Determina una solución para que nuestro programa funcione con la página de marca.

16. El **descargador de imágenes**. Vamos a crear un mini-navegador que nos permita revisar las imágenes que hay en una determinada página web y bajarnos una de ellas.

Para ello, el primer paso es que el usuario introduzca la URL de la web que quiere consultar, por ejemplo:

<http://www.rae.es>

Se listarán las 10 primeras URLs de las imágenes que haya en la página, solamente capturaremos las imágenes con rutasy absolutas y del tipo *jpg*, *jpeg*, *png* y *gif*. Por ejemplo, para el RAE:


```
Introduce la URL completa:
http://www.rae.es/
--- El titulo de la página web "http://www.rae.es/" es:
Real Academia Española
--- Los Links de imágenes que existen son:
0) http://www.rae.es/sites/default/files/rae.png
1) http://www.rae.es/sites/default/files/styles/sidebar_first/public/imagenes/portada/Anuario_2019_lateral_i
2) http://www.rae.es/sites/default/files/styles/sidebar_first/public/imagenes/portada/Escudo_RAE_1.jpg
3) http://www.rae.es/sites/default/files/styles/sidebar_first/public/YO_EL_SUPREMO_PORTADA_WEB_0.jpg
4) http://www.rae.es/sites/default/files/styles/medium_patrocinador/public/imagenes/portada/ELH.jpg
5) http://www.rae.es/sites/default/files/styles/sidebar_first/public/imagenes/portada/Nuevo_logo_de_Lope.jpg
6) http://www.rae.es/sites/default/files/styles/home_slide_large/public/imagenes/articulos/Foto.jpg
7) http://www.rae.es/sites/default/files/styles/home_slide_large/public/imagenes/articulos/_BP86025.jpg
8) http://www.rae.es/sites/default/files/styles/home_slide_large/public/imagenes/portada/_DSC5643.jpg
9) http://www.rae.es/sites/default/files/styles/home_slide_large/public/imagenes/portada/Serrano_2.jpg
Pulse una tecla del 0-9 para seleccionar el enlace:
```

El usuario, elegirá una de las imágenes que tiene la página web, y esta se descargará en su equipo. Ten en cuenta que la imagen tome el nombre de fichero que tiene en la URL. Por ejemplo si el usuario elije:

0) <http://www.rae.es/sites/default/files/rae.png>

La imagen se guardará como **rae.png**

