UNIDAD DIDÁCTICA 4:
GENERACIÓN DE SERVICIOS EN RED

Módulo profesional:
Programación de Servicios y Procesos

Índice

RESUMEN INTRODUCTORIO

En esta unidad seguiremos avanzando con la programación en red centrándonos en la última capa del protocolo TCP/IP, la capa más cercana al usuario, la capa de aplicación.

Repasaremos los conceptos dados en la UD anterior, donde introducíamos todas las librerías dentro de Java para poder realizar comunicaciones entre ordenadores y avanzaremos poniendo una capa más de funcionalidad, una capa de usuario.

Uniremos, por lo tanto, los conocimientos sobre protocolos de alto nivel como protocolos web, protocolo FTP o el SMPT, para introducir ejemplos realizados con las librerías base de Java tanto en servidor como en cliente.

A través de las librerías de Socket y ServerSocket podremos establecer esos mecanismos de comunicación del tipo cliente-servidor. Sólo nos quedará implementar el protocolo de forma adecuada una vez que tengamos el esqueleto.

Por último, realizaremos un ejemplo usando librerías específicas y especializadas para la implementación de uno de los servicios estudiados como es el de envío y recepción de correo electrónico.

INTRODUCCIÓN

Actualmente, la arquitectura cliente-servidor es la arquitectura base para la gran mayoría de comunicaciones entre ordenadores. Mediante esta arquitectura, un servicio programado en el servidor queda a la espera de una conexión por parte de un cliente. Una vez establecida, y dependiendo del protocolo establecido de capa de aplicación, comienza la comunicación extrema a extremo.

Uno de los ejemplos más conocidos y usados actualmente es el protocolo HTTP o también llamado Web, sobre el que se basan infinidad de otros servicios. Pero, además, en el día a día, también usamos otros protocolos y servicios basados en esa arquitectura cliente-servidor, como es el envío y recepción de correo electrónico o la autenticación de usuarios.

Con Java, y las librerías base dentro de .net, podemos desarrollar todos esos servicios de cliente y servidor, poniendo la diferencia en el protocolo que se desee implementar.

CASO INTRODUCTORIO

Acabamos de ser contratados en una empresa de logística para farmacias. En las farmacias disponen de un software propio con el que mantienen el stock de los fármacos, pero que también sirve para realizar pedidos a nuestra central.

Se pretende cambiar el mecanismo de pedido telefónico a mediante el mismo programa, para lo que se quiere realizar un pequeño software tipo cliente-servidor para realizar esta aplicación.

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

Al finalizar la unidad tendremos los recursos y habilidades para:

- Comprender y diseñar aplicaciones con arquitectura cliente/servidor.
- Realizar el desarrollo de aplicaciones tipo servidor.
- Realizar el desarrollo de aplicaciones tipo cliente.
- Usar de forma optimizada las clases y paquetes de Java para el desarrollo de aplicaciones tipo cliente/servidor.

1. SERVICIOS EN RED

El primer paso, es conocer protocolos existentes y formas de comunicación cliente-servidor para intentar adaptar o usar protocolos y servicios ya existentes.

A partir de ese conocimiento, ¿Podemos usar un protocolo existente? ¿Tenemos que reinventar uno propio o nuevo?

Antes de comenzar a programar, es importante responder a estas preguntas, ya que es importante para plantear los mecanismos de comunicación correctos.

El éxito de la red global no ha sido solo el desarrollo de una conectividad barata y eficiente. De hecho, desde los inicios, a la conectividad se añadieron un número creciente de aplicaciones, soportadas por los operadores de Internet, que intentaban dar respuestas genéricas a las necesidades de intercambio de información demandadas por los usuarios.

Se trata de aplicaciones distribuidas más o menos especializadas en algún tipo de intercambio, que siguen el modelo cliente-servidor y que se han popularizado con el nombre de servicios de red.

El acierto de Internet ha sido la estandarización de un conjunto de servicios que podríamos denominar "básicos", que son capaces de soportar nuevos servicios de más alto nivel, garantizando así la interoperabilidad de todos ellos, sin necesidad de nuevas instalaciones ni cambios de software en ninguno de los clientes que tenga que utilizarlos.

Entre los servicios de red que existen, cabe destacar los siguientes:

- **Servicios de archivo**: incluyen las aplicaciones de red necesarias para almacenar y recuperar datos de archivos. Además, ayudan a:

    - Hacer más eficiente el uso de hardware de almacenamiento.
    - Mover con mayor velocidad los archivos de un lugar a otro.
    - Respaldar datos.
    - Manejar varias copias del mismo archivo.
    - Transferencia de archivos y
    - Almacenamiento.

- **Servicios de impresión**: incluyen las aplicaciones de red necesarias para controlar y administrar el acceso a impresoras y fax. Además:

  - Facilitan la reducción del número de impresoras necesarias en una organización.
  - Mejoran la colocación de impresoras.
  - Reducen el tiempo de impresión.
  - Mejoran la gestión de las colas de trabajos de impresión.

- **Servicios de mensajería**: incluyen las aplicaciones necesarias para almacenar, acceder y entregar los mensajes. Además, ayudan en:

  - La organización y mantenimiento de los directorios de información.
  - El direccionamiento y compartición de datos.
  - La integración del correo electrónico con los sistemas de voz.

- **Servicios de aplicación**: incluyen las aplicaciones necesarias para correr software para clientes de red. Permiten, además de compartir los datos, compartir el poder de procesamiento de las computadoras.

- **Servicios de bases de datos**: proveen bases de datos a los clientes de la red de manera que éstos puedan almacenar y recuperar los datos para así controlar la manipulación y presentación de éstos.

  Estos servicios:
  - Optimizan las computadoras ya que les permite almacenar, localizar y recuperar registros de bases de datos.
  - Organizar los datos.
  - Reducir los tiempos de acceso a las bases de datos por parte de los clientes.

PARA SABER MÁS

En el siguiente enlace aprenderá a configurar los principales servicios de red:

https://docs.oracle.com/cd/E24842_01/html/820-2981/ipconfig-1.html

VIDEO DE INTERÉS

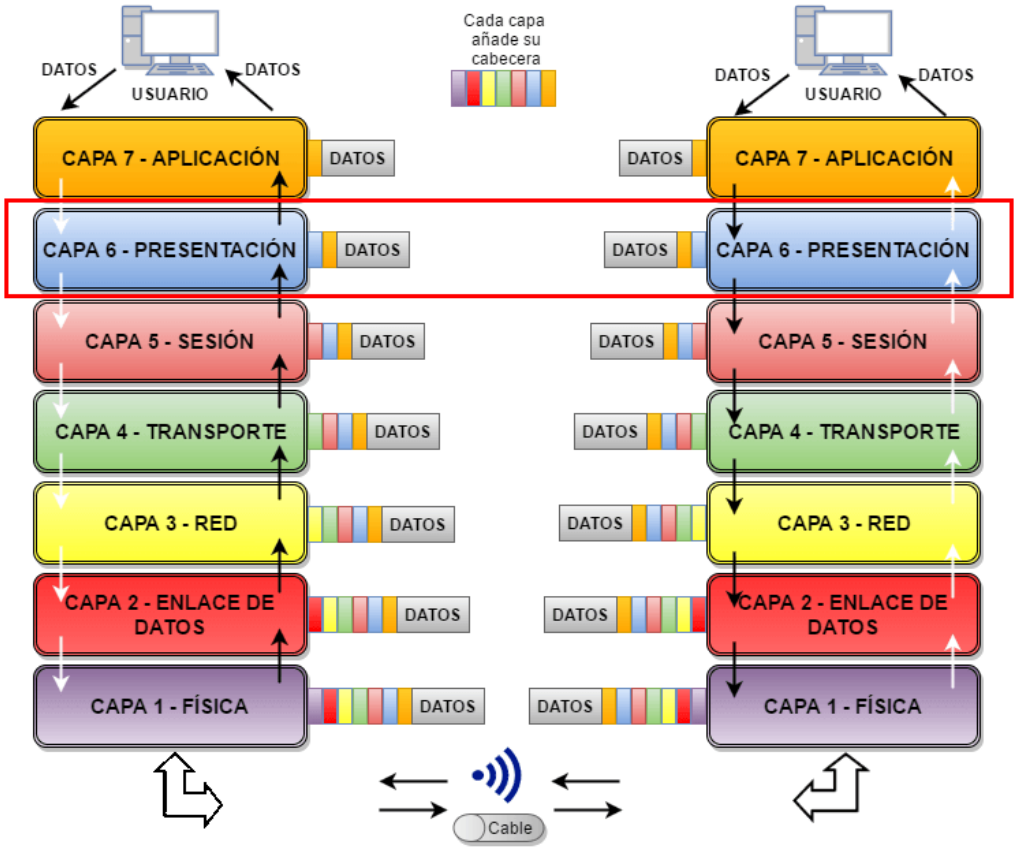
En el siguiente vídeo podrás ver los modelos basados en capas y en concreto, la capa de aplicación:

https://www.youtube.com/watch?v=oHJOo38Ts70

1.1 Protocolos estándar de comunicación en red

En la unidad anterior estudiábamos las capas del **protocolo TCP/IP**, Transfer Control Protocol/Internet Protocol. Si recordamos, vimos que 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.

Este modelo es muy similar al modelo OSI de 7 capas, que fue desarrollado por la Organización Internacional para la Estandarización (ISO) para estandarizar las comunicaciones entre equipos. En la siguiente imagen podrá observar la equivalencia de capas entre un modelo y otro.



Ejemplo de modelo de pares

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

Las capas del modelo TCP/IP 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.

- **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 la 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.

1.2 Protocolos estándar de comunicación en red a nivel de aplicación

En este punto nos vamos a centrar en la capa de aplicación y sus protocolos. Éstos proporcionan las reglas para la comunicación entre las aplicaciones, además de:

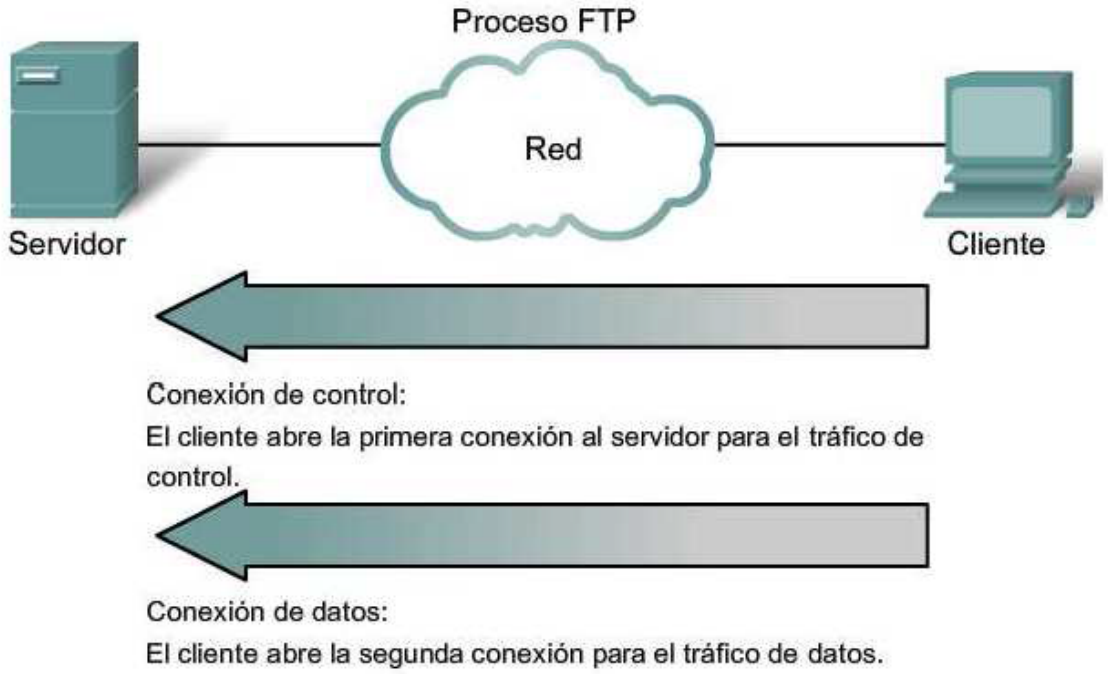
- Definir los procesos en cada uno de los extremos de la comunicación.
- Definir los tipos de mensajes.
- Definir la sintaxis de los mensajes.
- Definir el significado de los campos de información.
- Definir la forma en que se envían los mensajes y la respuesta esperada.
- Definir la interacción con la próxima capa inferior.

Los principales protocolos de la capa de aplicación son:

- **Telnet:** servicio que ofrece la dirección IP de un sitio web o nombre de dominio para que un host pueda conectarse a éste. Una vez conectado, los usuarios pueden realizar cualquier función autorizada en el servidor.

- **FTP (File Transfer Protocol):** servicio que permite descargar y subir archivos entre un cliente y un servidor.

  El protocolo de transferencia de archivos es uno de los protocolos más utilizados. Permite las transferencias de archivos entre un cliente y un servidor para lo que necesita dos conexiones: una para los comandos y respuestas y otra para la transferencia real de los archivos solicitados. Veamos un ejemplo gráfico:



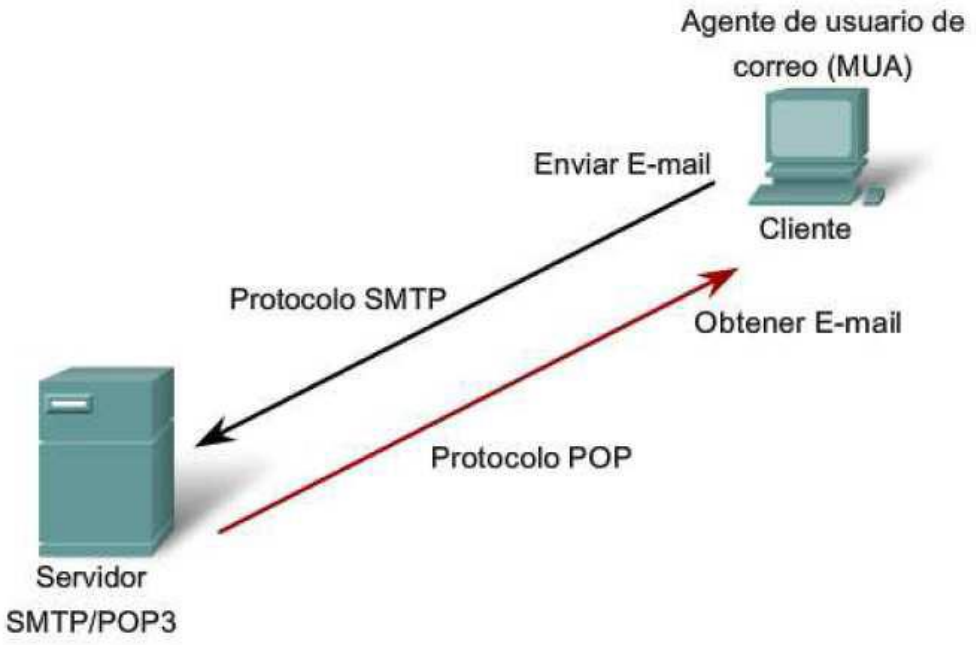
Protocolo FTP
Fuente: Coremsa

- **HTTP (Hypertext Transfer Protocol, Protocolo de transferencia de hipertexto):** protocolo que se utiliza para transferir información entre clientes web y servidores web.

  Este protocolo surgió por la necesidad de publicar y recuperar paginas HTML, pero en la actualidad se utiliza para la transferencia de datos a través de la WWW.

  Al escribir una dirección de Internet en un navegador web, el navegador realiza una conexión con el servicio web del servidor que utiliza el protocolo HTTP. Para acceder al contenido, los clientes web realizan conexiones al servidor y solicitan los recursos deseados. El servidor responde con los recursos y una vez recibidos, el navegador interpreta los datos y los presenta al usuario.

- **POP3 (Post Office Protocol, Protocolo de oficina de correos) y SMTP (Simple Mail Transfer Protocol, Protocolo simple de transferencia de correo):** protocolos utilizados para enviar mensajes de email de clientes a servidores a través de Internet. Cuando escribimos y recibimos mensajes de correo electrónico estamos utilizando estos protocolos. Por ejemplo, al enviar un email se utilizan formatos de mensajes definidos por el protocolo SMTP y al recibirlos, el cliente de correo electrónico puede utilizar el protocolo POP. Veamos una imagen ilustrativa del proceso:

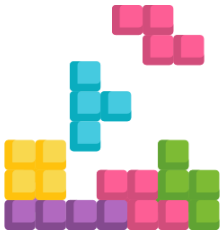


Protocolo Correo
Fuente: Coremsa

COMPRUEBA LO QUE SABES

Acabamos de estudiar diferentes servicios de aplicación, ¿serías capaz de poner dos ejemplos de uso del protocolo HTTP?

Razona el ejemplo. Coméntalo en el foro.



EJEMPLO PRÁCTICO

Queremos programar sobre la aplicación que tenemos instalada en las farmacias una ampliación que nos permita realizar pedidos de fármacos a nuestro almacén central.

Para esa comunicación necesitamos realizar el siguiente proceso de comunicación:

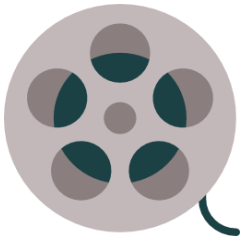
- Solicitar desde tienda nuevo pedido.
- Enviar desde tienda los productos.
- Recibir desde almacén el OK y cuando se recibirían.

¿Qué tipo de servicio elegiríamos?

Podríamos reducir nuestra elección a dos tipos, bien un servicio tipo correo electrónico, bien un servicio tipo HTTP/Web o bien un servicio propietario. Veamos los pros y contras:

- El servicio tipo correo electrónico es fácil de implementar y permitiría que el servicio funcionase incluso protegiéndose ante caídas de la red. Sin embargo, el servicio no está orientado a la conexión y necesitaríamos.
- El servicio propietario puede ser totalmente a medida, implementado el protocolo que mejor se adapta, así como los mecanismos que mejor se nos adapten. El problema es que es muy poco escalable y mantenible.
- Por último, el protocolo HTTP/Web también es muy sencillo de implementar y además está muy extendido. En este caso necesitamos de conexión para su buen funcionamiento.

Nos decantamos por el protocolo HTTP/Web. Podemos usar servidores webs ya implementados en servidor e incluso basados en otras tecnologías, y clientes Java propios.



VIDEO DE INTERÉS

En este vídeo podrás aprender más acerca del protocolo SMTP:

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

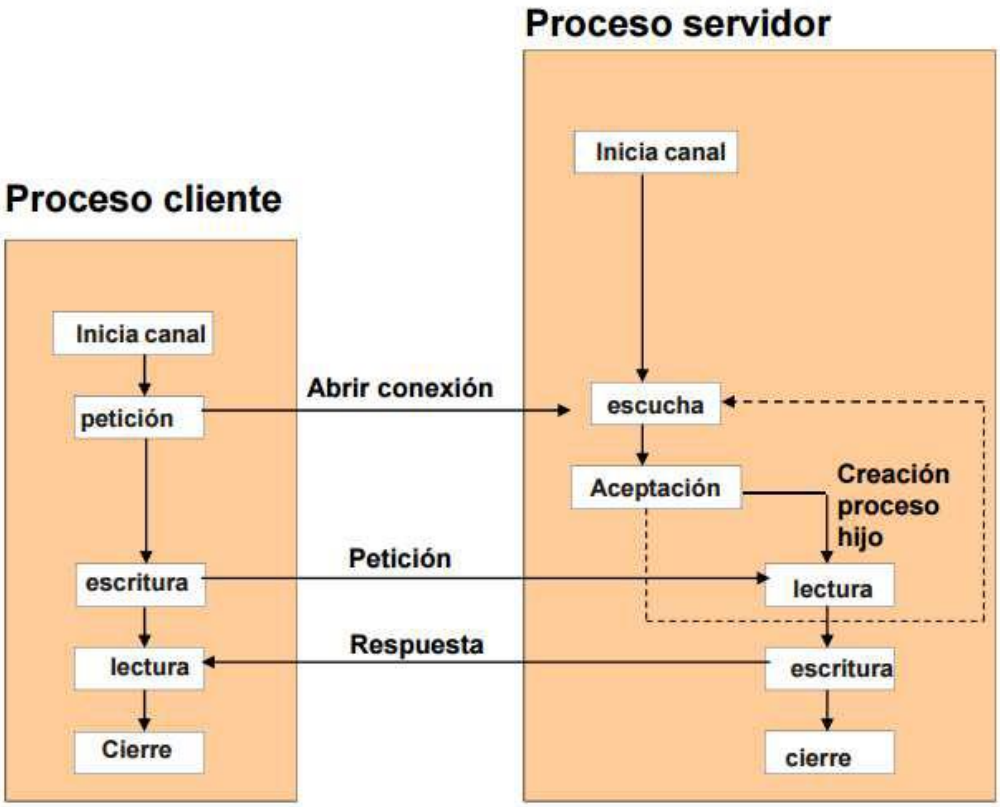
2. PROGRAMACIÓN DE SERVICIOS EN RED CON JAVA

Una vez que se ha elegido el correcto protocolo de comunicación, debemos comenzar a plantear la aplicación tanto en el lado del cliente como en el lado del servidor.

¿Tenemos librerías dentro de Java específicas? ¿Cómo desarrollamos el cliente? ¿Cómo desarrollamos el servidor?

Con las respuestas adecuadas a estas preguntas podremos realizar una correcta programación de nuestros servicios.

En la unidad anterior, estudiamos que el paradigma cliente / servidor es el más utilizado para la programación de aplicaciones de red. Si recordamos cómo se programan estas aplicaciones:



Arquitectura cliente-servidor

Fuente de la imagen: Coremsa

El servidor se ejecuta en una máquina específica y tiene un socket asociado a un número de puerto específico, donde espera a que un cliente le envíe una petición. Por la parte de los clientes, como estos saben el nombre de la máquina sobre la que se está ejecutando el servidor y el número de puerto, solicita conexión con el servidor mediante esos parámetros.

Si todo va bien, el servidor acepta la conexión y crea un nuevo socket en un puerto diferente.

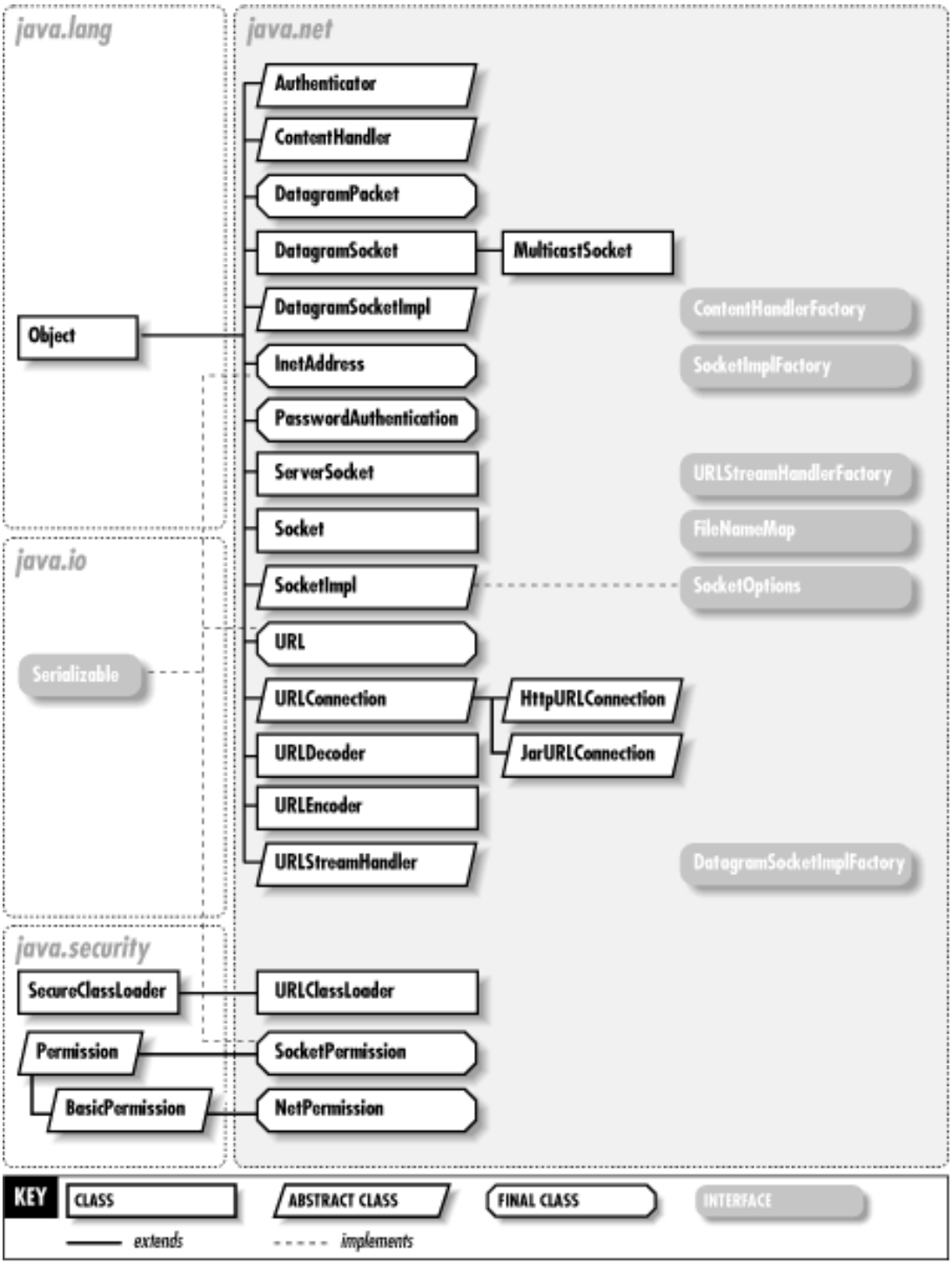
De esta manera el socket inicial se mantiene a la escucha de nuevas peticiones de clientes y este nuevo socket permite la conexión entre el servidor y el cliente en cuestión.

Para la programación de este tipo de aplicaciones es necesario conocer como programar sockets en el lenguaje de programación Java. Esto ya lo estudiamos en la unidad anterior así que no vamos a centrarnos en esto.

2.1 Librerías de clases y componentes

En la UD3, vimos que son muchas las librerías, clases, métodos y funciones que tenemos dentro de Java, y tal y como veremos a continuación. Java proporciona una colección de clases e interfaces que se encargan de los detalles de comunicación de bajo nivel entre el cliente y el servidor.

El paquete principal de librerías y clases lo encontramos dentro de java.net tal y como vemos en el árbol de clases dentro de java.net.



Paquete Java.Net

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

Los componentes más importantes se encuentran en su mayoría en el paquete java.net, por lo que debemos realizar la siguiente importación:

```
import java.net. *;
```

También necesitamos el paquete java.io que nos brinda flujos de entrada y salida para escribir y leer mientras nos comunicamos:

```
import java.io. *;
```

ENLACE DE INTERÉS

En el siguiente enlace podrás ver la Documentación oficial de java.net:

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

2.2 Utilización de objetos predefinidos

Los sockets son los objetos predefinidos y más importantes que nos permiten implementar el resto de las comunicaciones. Proporcionan el mecanismo de comunicación entre dos ordenadores que utilizan TCP. Un programa cliente crea un conector en su extremo de la comunicación e intenta conectar ese conector a un servidor, tal y como hemos visto en UD anteriores y vamos a ver en esta.

La clase ***java.net.Socket*** representa un socket y la clase ***java.net.ServerSocket*** proporciona un mecanismo para que el programa servidor escucha a los clientes y establece conexiones con ellos.

A continuación, vamos a establecer una tabla con los métodos más importantes de estas dos librerías y que después nos servirán para poder implementar diferentes tipos de ejemplos de aplicación.

2.2.1 Objetos dentro de java.net.ServerSocket

En primer lugar, veamos los constructores más importantes del ServerSocket en la tabla:

Método	Descripción
public ServerSocket(int port) throws IOException	Intenta crear un socket de servidor vinculado al puerto especificado. Se produce una excepción si el puerto ya está vinculado por otra aplicación.
public ServerSocket(int port, int backlog) throws IOException	Al igual que en el constructor anterior, el parámetro backlog especifica cuántos clientes entrantes almacenar en una cola de espera.
public ServerSocket(int port, int backlog, InetAddress address) throws IOException	Al igual que en el constructor anterior, el parámetro InetAddress especifica la dirección IP local a la que enlazar. InetAddress se utiliza para servidores que pueden tener varias direcciones IP, lo que permite al servidor especificar en cuál de sus direcciones IP aceptar las solicitudes de los clientes.
public ServerSocket() throws IOException	Crea un socket de servidor independiente. Cuando use este constructor, use el método bind () cuando esté listo para vincular el socket del servidor.

En segundo lugar, veamos los métodos más importantes del ServerSocket en la tabla:

Método	Descripción
public int getLocalPort()	Devuelve el puerto en el que escucha el socket del servidor. Este método es útil si pasó 0 como número de puerto en un constructor y dejó que el servidor encuentre un puerto libre.
Public Socket accept() throws IOException	Espera un cliente entrante. Este método se bloquea hasta que un cliente se conecta al servidor en el puerto especificado o el socket agota el tiempo de espera, asumiendo que el valor de tiempo de espera se ha establecido usando el método setSoTimeout (). De lo contrario, este método se bloquea indefinidamente.
Public void setSoTimeout(int timeout)	Establece el valor de tiempo de espera para el tiempo que el socket del servidor espera a un cliente durante el accept ().
Public void bind(SocketAddress host, int backlog)	Vincula el socket al servidor y al puerto especificados en el objeto SocketAddress. Utilice este método si se ha creado una instancia de ServerSocket utilizando el constructor sin argumentos.

2.2.2 Objetos dentro de java.net.Socket

La clase java.net.Socket representa el socket que tanto el cliente como el servidor utilizan para comunicarse entre sí. El cliente obtiene un objeto Socket creando una instancia de uno, mientras que el servidor obtiene un objeto Socket del valor de retorno del método accept() antes visto.

Haciendo un resumen del constructor:

Método	Descripción
public Socket(String host, int port) throws UnknownHostException, IOException	Este método intenta conectarse al servidor especificado en el puerto especificado. Si este constructor no lanza una excepción, la conexión es exitosa y el cliente está conectado al servidor.
public Socket(InetAddress host, int port) throws IOException	Este método intenta conectarse al servidor especificado en el puerto especificado. Si este constructor no lanza una excepción, la conexión es exitosa y el cliente está conectado al servidor.
public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException	Este método intenta conectarse al servidor especificado en el puerto especificado. Si este constructor no lanza una excepción, la conexión es exitosa y el cliente está conectado al servidor.
public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException	Este método es idéntico al constructor anterior, excepto que el host se indica mediante un objeto InetAddress en lugar de un String.
public Socket()	Crea un socket sin conectar. Se utiliza el método connect() para conectar este socket a un servidor.

Para el caso de los métodos:

Método	Descripción
public void connect(SocketAddress host, int timeout) throws IOException	Este método conecta el socket al host especificado. Este método es necesario solo cuando crea una instancia del Socket usando el constructo sin argumentos
public InetAddress getInetAddress()	Este método devuelve la dirección de la otra computadora a la que está conectado este socket.
public int getPort()	Devuelve el puerto al que está vinculado el socket en la máquina remota.
public int getLocalPort()	Devuelve el puerto al que está vinculado el socket en la máquina local.
public SocketAddress getRemoteSocketAddress()	Devuelve la dirección del socket remoto
public InputStream getInputStream() throws IOException	Devuelve el flujo de entrada del socket. El flujo de entrada está conectado al flujo de salida del enchufe remoto.
public OutputStream getOutputStream() throws IOException	Devuelve el flujo de salida del socket. El flujo de salida está conectado al flujo de entrada del enchufe remoto.
public void close() throws IOException	Cierra el socket, lo que hace que este objeto Socket ya no pueda conectarse nuevamente a ningún servidor.

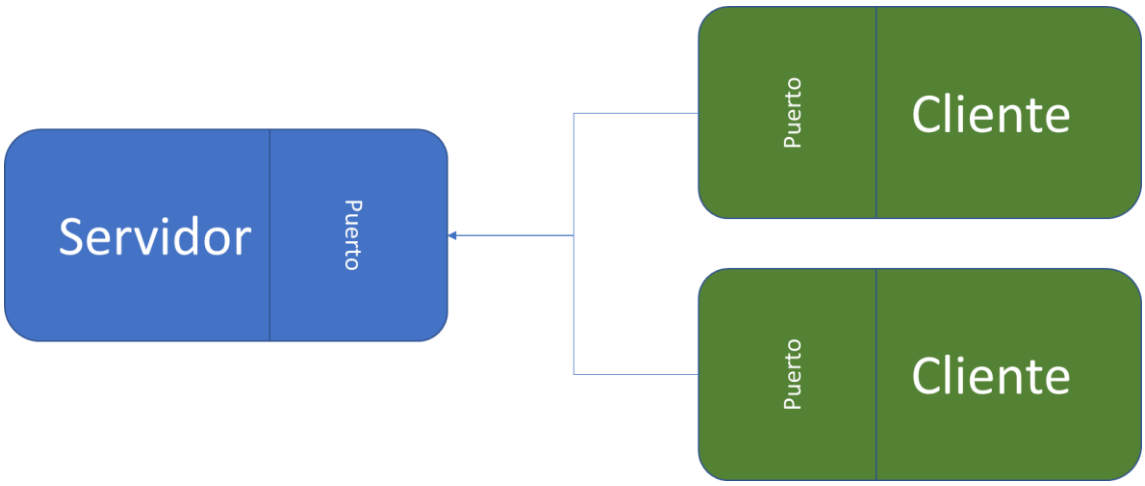
COMPRUEBA LO QUE SABES

Acabamos de estudiar Socket y ServerSocket, ¿serías capaz de indicar un flujo de comunicación entre cliente y servidor usando ambas librerías?

Coméntalo en el foro.

2.3 Establecimiento, transmisión y finalización de conexiones

En una comunicación tipo cliente servidor, que es el tipo de comunicaciones de servicios que nos vamos a centrar, un servidor se ejecuta en un ordenador con un IP específico y tiene un socket que está vinculado a un número de puerto específico. El servidor simplemente espera, escuchando el socket para que un cliente haga una solicitud de conexión.



Conexión Cliente-Servidor
Fuente: Imagen propia

En el lado del cliente, se conoce el nombre de host de la máquina en la que se ejecuta el servidor y el número de puerto en el que escucha el servidor. El cliente también necesita identificarse ante el servidor para que se vincule a un número de puerto local que utilizará durante esta conexión. Esto suele ser asignado por el sistema y de forma automática.

La mejor forma de entender todo el proceso involucrado en la conexión, transmisión de información y cierre de la misma es a través de un ejemplo con un cliente y con un servidor. Crearemos una aplicación donde intervengan cliente y servidor y donde se produzcan los pasos habituales de dicha comunicación.

Servidor

```
public class GreetServer {
   private ServerSocket serverSocket;
   private Socket clientSocket;
   private PrintWriter out;
   private BufferedReader in;

   public void start(int port) {
      serverSocket = new ServerSocket(port);
      clientSocket = serverSocket.accept();
      out = new PrintWriter(clientSocket.getOutputStream(), true);
      in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
      String greeting = in.readLine();
         if ("hello server".equals(greeting)) {
            out.println("hello client");
         }
         else {
            out.println("unrecognised greeting");
         }
   }

   public void stop() {
      in.close();
      out.close();
      clientSocket.close();
      serverSocket.close();
   }
   public static void main(String[] args) {
      GreetServer server=new GreetServer();
      server.start(6666);
   }
}
```

En el anterior servidor observamos tres partes claramente:

- La creación del servidor y su puesta en espera. Esto lo realiza el método start, lanzando un nuevo ServerSocket en un puerto en concreto.

- La comunicación entre cliente y servidor:
    - Aceptando la conexión sobre el ServerSocket.
    - Abriendo un canal de comunicación por el que se realiza la comunicación bidireccional.

- Por último, existe un método de cierre y liberación de todos los recursos denominado stop.

Para el caso del cliente tendríamos que hacer lo que muestra en la tabla:

Cliente

```
public class GreetClient {
   private Socket clientSocket;
   private PrintWriter out;
   private BufferedReader in;

   public void startConnection(String ip, int port) {
      clientSocket = new Socket (ip, port);
      out = new PrintWriter(clientSocket.getOutputStream(), true);
      in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
   }

   public String sendMessage(String msg) {
      out.println(msg);
      String resp = in.readLine();
      return resp;
   }

   public void stopConnection() {
      in.close();
      out.close();
      clientSocket.close();
   }
}
```

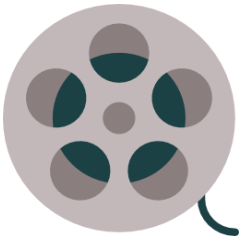
En el caso del cliente también podemos ver esos tres estados de la comunicación:

- Conexión y establecimiento de la comunicación mediante el método startConnection y donde necesitaremos tanto la ip como el puerto.
- La transferencia de información a través de sendMessage.
- La finalización de la conexión a través del método stopConnection.

Para probar este cliente con el servidor anteriormente programado, podríamos desarrollar el código de prueba:

Test

```
public void
givenGreetingClient_whenServerRespondsWhenStarted_thenCorrect() {
    GreetClient client = new GreetClient();
    client.startConnection("127.0.0.1", 6666);
    String response = client.sendMessage("hello server");
    assertEquals("hello client", response);
}
```



VIDEO DE INTERÉS

En el siguiente vídeo podrás ver una explicación clara sobre la programación de clientes TCP con Java:

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

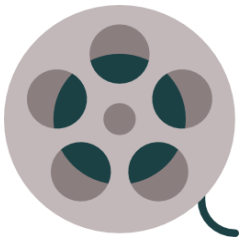
2.4 Programación de servidores y servicios

De entre la gran cantidad de servicios de Internet destacan, por su importancia, tres servicios que sin duda han marcado y todavía marcan la evolución de la red de redes. Por orden de importancia son el servicio WWW o Web, los servicios de correo electrónico y los servicios de transferencia de ficheros. La importancia se refiere sobre todo por la utilización que se les da en el desarrollo de aplicaciones más grandes, ya sea como complementos o como apoyo base de su implementación.

ENLACE DE INTERÉS

En el siguiente enlace podrás ver una lista de todos los documentos RFC existentes:

http://www.ietf.org/download/rfc-index.txt



VIDEO DE INTERÉS

En el siguiente vídeo podrás ver más detalladamente la programación de servidores TCP con Java:

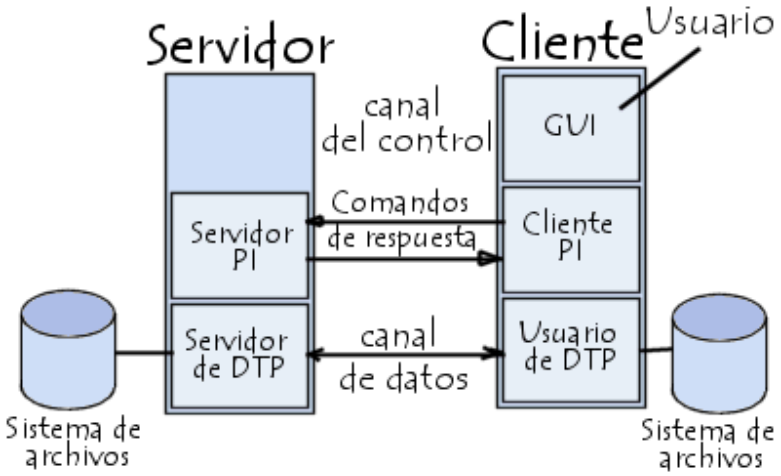
https://www.youtube.com/watch?v=bNXpF7tF10U

2.4.1 Servicio de transferencia de ficheros

Este servicio, tal como indica su nombre, permite gestionar la transferencia de ficheros entre dos lugares situados en diferentes dispositivos. El protocolo estándar utilizado aquí es el llamado FTP que se encuentra especificado al documento RFC-959.

El protocolo corre sobre TCP y se basa en un modelo cliente-servidor que contempla un conjunto de pedidos que permitirán al cliente gestionar un sistema de ficheros remoto.

En general podemos decir que se sigue el patrón petición-respuesta de la mayoría de las aplicaciones cliente-servidor. En este caso las peticiones son las peticiones que el cliente envía y las respuestas las genera el servidor asociando un código y un mensaje que informa de si la petición ha tenido o no éxito.

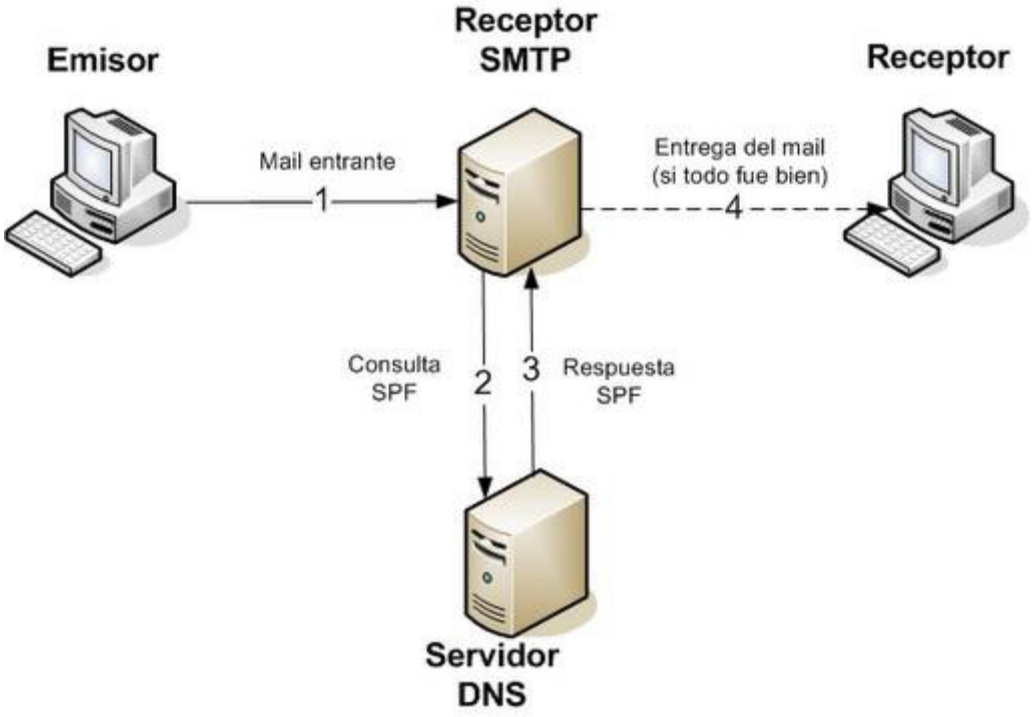


FTP

Fuente: https://es.ccm.net/contents/263-protocolo-ftp-protocolo-de-transferencia-de-archivos

2.4.2 Servicios de correo electrónico

Cuando hablamos de servicios de correo electrónico, básicamente nos referimos a dos servicios diferentes que colaboran para conseguir transmitir un mensaje digital desde el dispositivo del autor (remitente) al dispositivo del destinatario. Autores y destinatarios son particulares que no mantienen su dispositivo permanentemente abierto y conectado, cosa que podría impedir la recepción si el dispositivo destinatario se encontrara desconectado en el momento de la transmisión. La solución pasa para almacenar los mensajes en dispositivos a los cuales se pueda acceder desde el destinatario para acabar de realizar la transmisión. Haciendo un símil con la vida real, los dispositivos de almacenamiento se denominan buzones de correo electrónico. Cada destinatario debe tener un buzón diferenciado en la cual encontrar solo sus mensajes.



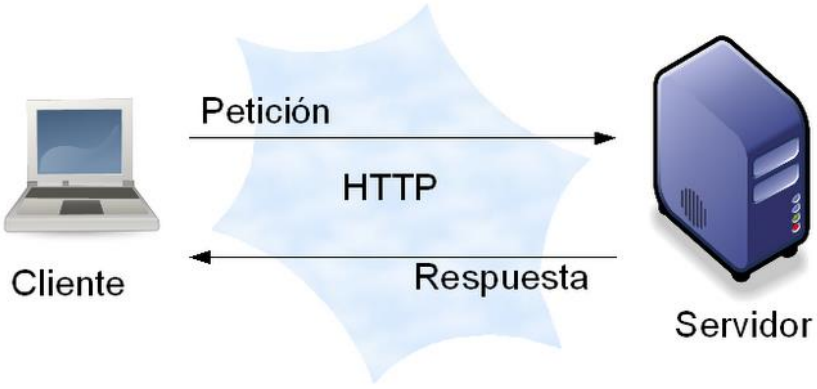
ISMTP

Fuente: https://www.gya.es/registros-spf-prevencion-de-spam/

La transmisión efectiva de los mensajes electrónicos se conseguirá pues realizando una transmisión desde el dispositivo del autor hasta el buzón del destinatario, en una primera fase, y desde el buzón hasta el dispositivo del destinatario en una segunda. La primera fase da lugar al servicio de transferencia de mensajes. La segunda al servicio de acceso a los buzones y recepción del contenido.

2.4.3 Servicio WWW o Web

Este servicio, que normalmente se encuentra asociado al puerto 80, se ha convertido en el más importante de los tres en los últimos años, puesto que ha acabado convirtiéndose en el servicio que sustenta la comunicación y presentación de los datos de una gran cantidad de aplicaciones distribuidas e incluso de nuevos servicios que se sobreponen al primero. La gran versatilidad de este servicio ha acabado desbancando totalmente otros servicios, como por ejemplo Telnet, pensados específicamente para apoyar a la ejecución de aplicaciones remotas y de servicios de nivel superior.



Servicio WWW

Fuente: http://scampos94.blogspot.com/2012/08/protocolo-http-https-ftp-www-www2-y-web.html

HTML es el acrónimo de Hipertexto Makup Language (Lenguaje de marcaje de hipertetx) y HTTP lo de Hipertexto Transfer Protocolo (Protocolo de transferencia de hipertexto).

COMPRUEBA LO QUE SABES

Acabamos de estudiar HTML como el lenguaje usado sobre HTTP, ¿serías capaz de indicar un servidor web que pudiéramos usar como ejemplo para nuestras comunicaciones?

Coméntalo en el foro.

2.5 Programación de aplicaciones cliente

En un apartado anterior hemos realizado un ejemplo de aplicación cliente. En este punto realizaremos otro ejemplo en este caso sobre uno de los servicios que acabamos de estudiar, el servicio de correo.

Para implementar clientes de correo Java, pone a nuestra disposición la biblioteca JavaMail. La biblioteca solo se encuentra disponible si usamos la plataforma J2EE que es una extensión del JDK. JavaMail es una biblioteca que trabaja sobre la mayoría de los protocolos, ya sea de transmisión o de acceso a los buzones. Por supuesto reconoce los protocolos SMTP y POP3 y consigue una programación bastante limpia en ambas funcionalidades.

ENLACE DE INTERÉS

En el siguiente enlace podrás encontrar la documentación oficial de JavaMail:

https://javaee.github.io/javamail/

La estructura de la biblioteca es de muy alto nivel y con esto consigue independizar la programación del protocolo realmente utilizado. La clase Session contribuye a esta abstracción. Se trata de la clase que da acceso a uno de los protocolos disponibles y representa una sesión de correo abierta contra un proveedor de los servicios de correo. La clase Session se configura a través de Properties puesto que resulta una forma muy versátil de asociar atributos y valores que se puede adaptar a casi cualquier necesidad.

```
Properties props = new Properties();
props.put("mail.smtp.host", hostName);
props.put("mail.smtp.port", 459);
//Crea una instancia con las propiedades
Session session = Session.getInstance(props);
```

Si hay que realizar transferencia de correos, *Session* trabajará con una instancia de la clase *Transport*. Esta es una clase abstracta materializada por una clase que se adapta a un protocolo concreto. Por defecto, la biblioteca JavaMail soporta *SMTPTransport*, pero el hecho de trabajar con la clase abstracta permite en un futuro incorporar nuevas clases. Es importante, pues, que todas las instancias que haya que crear estén declaradas como Transporte si queremos construir una aplicación escalable y de larga duración.

Las clases de la jerarquía *Transport* básicamente tienen la funcionalidad de enviar mensajes siguiendo el protocolo que soporten y usando una instancia abierta de *Session*. Para evitar crear instancias temporales que nos puedan cargar la memoria, *Transport* tiene el método *static send* para conseguir realizar el envío sin tener que crear ninguna instancia. Este método abre y cierra una conexión cada vez que lo invocamos.

```
InternetAddress[] addresses = {new
InternetAddress("direccion@dominio.com"),
new InternetAddress("direccion2@ dominio.com ")};

...

Transport.send(emailMessage1);

Transport.send(emailMessage2, addresses);
```

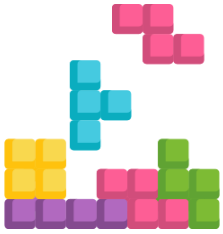
El objeto del envío, que tendremos que haber construido previamente, es una instancia de la clase *Message*, que es creado por la clase *MimeMessage*. La clase *Message* es una clase abstracta que no podamos instanciar. Usaremos siempre la clase *MimeMessage* por instanciar el objeto. *MimeMessage* soporta la especificación MIME. Es decir, podemos enviar mensajes de contenido diverso y adjuntar ficheros, de acuerdo con esta especificación.

```
Message emailMessage = new MimeMessage(session);

...

emailMessage.setFrom(new InternetAddress("Midireccion@dominio.com"));
InternetAddress[] address = {new InternetAddress("secretaria@
dominio.com " ),
new InternetAddress("direcciofp@dominio.com")};
emailMessage.setRecipients(Message.RecipientType.TO, address);
emailMessage.setRecipient(Message.RecipientType.CC,
new InternetAddress("secretaria@ dominio.com " ) );
emailMessage.setSubject("Prova de construcció amb JavaMail");
emailMessage.setSentDate(new Date());

emailMessage.setText("Contenido del mensaje texto plano " +
"Atentamente Paco");
```



EJEMPLO PRÁCTICO

Dentro de nuestra comunicación entre equipo de tienda y servidor central queremos implementar en el cliente conocer si en la central disponen de un determinado producto.

Para ello en la central hay un servidor con un servicio que está esperando diferentes peticiones y sobre la cual podemos conocer el estado de un producto.

- En primer lugar, crearemos la clase que se encargue en el cliente de conectarse contra el servidor:

```
// EstadoProducto.java
import java.net.*;
import java.io.*;

public class EstadoProducto {

  private Socket client                    ;
  public EstadoProducto(String servidor, int puerto) {
    String serverName = servidor;
    int port = puerto;
    try {
      System.out.println("Conectando a " + serverName + " en el puert " +
port);
      this.client = new Socket(serverName, port);

      System.out.println("Conectado a " +
this.client.getRemoteSocketAddress());

    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

- Una vez que tenemos el cliente conectado con el servidor realizaremos la petición a nuestro servidor, mediante el código de producto:

```
 int public producto (int producto) {
   try {
     OutputStream outToServer = this.client.getOutputStream();
out.writeUTF("Producto: " + producto);

     DataOutputStream out = new DataOutputStream(outToServer);
```

```
        InputStream inFromServer = this.client.getInputStream();
        DataInputStream in = new DataInputStream(inFromServer);

        System.out.println("Estado de producto: " + in.readUTF());
        this.client.close();


      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

RESUMEN FINAL

Los servicios cliente servidor son hoy en día el mecanismo de comunicación entre aplicaciones ya que hay herramientas, librerías y protocolos muy maduros y diversos que nos permiten elegir para diferentes usos y situaciones.

Como hemos visto, tenemos muchos servicios dentro de esa capa de aplicación, y dentro de estos, tres servicios sobresalen por encima de los demás: FTP, correo y web. La transferencia de ficheros es un servicio que poco a poco está siendo desplazado por otro tipo de servicios sobre todo por encima del protocolo HTTP. El servicio de correo ha evolucionado también introduciendo seguridad y mecanismos comunicación también basado en HTTP. Por último, el servicio web o HTTP es uno de los servicios que mayormente ha evolucionado y más se está utilizando, por su sencillez y su versatilidad.

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.

También encontramos librerías especializadas, tal y como hemos visto en la última parte. Un ejemplo lo tenemos con JavaMail, la cual nos proporciona librerías para el desarrollo de un cliente de correo que podrá usar diversos protocolos de una forma transparente.