PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONESMULTIPLATAFORMA

Modelos de comunicaciones
Comunicación Cliente/Servidor

Arquitectura cliente/servidor

A nivel general, cuando se trata de aplicaciones de comunicaciones entre equipos, el modelo más utilizado es el **modelo Cliente/Servidor**, debido a que ofrece una enorme **flexibilidad, interoperabilidad y estabilidad** para acceder a recursos que están centralizados en un ordenador. Este modelo apareció por primera vez en los años 80, resolviendo uno de los grandes problemas de la época: un ordenador necesita un servicio que provee otro.

Aplicaciones que usan el modelo Cliente/Servidor:

- **Outlook** es un **cliente** de correo que sirve para leer los correos del servicio hotmail.
- **WhatsApp** o **Telegram** son **clientes** de servicios de chat, vídeo, audio.
- **Firefox** es un **cliente** web, **Chrome** también ("navegadores").
- **Internet Information Server** y **Apache Web Server** son **servidores** web.

Este modelo está **compuesto por los siguientes elementos**:

importante memorizar:

- **Cliente**:

  Es quien va a interactuar con el **usuario**, realizando las peticiones que sean necesarias al servidor y **mostrando** a posteriori los **resultados** a la persona con la que está interactuando. Las principales **funciones** que va a cumplir el cliente son:

  - Interactuar con el **usuario**.
  - **Procesar** todas las **peticiones** para comprobar que sean **válidas**, evitando enviar peticiones malintencionadas al servidor.
  - **Recibir** los **resultados** que le envía el servidor.
  - Dar un **formato correcto** a los **datos** recibidos para mostrárselos al usuario.

- **Servidor**:

  Este va a ser quien provea de **uno o varios servicios** a los clientes.

  Las **funciones** que deberá desempeñar son:

  - **Aceptar las peticiones** de los clientes.
  - **Procesar** en orden las **peticiones** de los clientes.
  - Dar un **formato correcto** a los **datos** enviados a los clientes.
  - Realizar **validaciones** de **datos**.
  - Debe asegurar que la **información** sea **consistente**.
  - Mantener **seguro el sistema**.

Variantes de modelo Cliente/Servidor

- ***1*** *cliente y* ***1*** *servidor.*
- ***1*** *servidor y* ***muchos*** *clientes*
- ***muchos*** *servidores y* ***muchos*** *clientes.*

Todo dependerá de los ***servicios*** *que se necesiten proporcionar, ya que esto condicionará la* ***complejidad*** *del propio sistema.*

Características modelo cliente/servidor

Características básicas:

• Existe una **combinación del cliente con los usuarios**, ya que estos son los que interactúan entre sí. También existe una **combinación entre el servidor y los recursos** que se van a ofrecer, es decir: es el **cliente** el que deberá proporcionar una **interfaz gráfica** al usuario y será el **servidor** el que permita el **acceso remoto a los recursos** que se compartirán.

• Las **operaciones** que podrán realizar los **clientes** y los **servidores** van a necesitar, claramente, unos **requisitos totalmente diferentes** en lo que respecta al tratamiento de información, por razones obvias, y será el **servidor** quien realice todo el **trabajo de procesamiento**.

• Existe una clara **relación entre el cliente y el servidor**, pudiéndose ejecutar en un **mismo ordenador o** en varios ordenadores **distribuidos en red**.

• Un servidor podrá dar **servicio a uno o muchos clientes**, realizando siempre las operaciones necesarias para que las **peticiones se ejecuten en orden** y que ningún cliente se quede sin atender.

• Los **servidores** van a tener un **rol pasivo** en este modelo, ya que estarán esperando a que los clientes les envíen sus peticiones, mientras que los **clientes** tendrán un **rol activo** en el modelo, ya que serán los que envíen las peticiones cuando las necesiten.

• La **comunicación** entre cliente y servidor **únicamente** se puede realizar a través del **envío de mensajes**, tantos como hagan falta.

• Los clientes podrán utilizar **cualquier plataforma** para conectarse a los servidores.

Cantidad necesaria de Servidores

Las aplicaciones que usan el modelo cliente/servidor deben ser capaces de dar servicio a todos los clientes que lo necesiten, da igual que sea uno o que sean millones.

Deben dar soporte a todos los clientes que lo necesiten y, para ello, **probablemente deberemos tener más de un servidor**. El uso de varios servidores en la arquitectura cliente/servidor **es la norma que predomina**, ya que es inconcebible no poder dar soporte a los clientes.

Tenemos, incluso, casos en los que los propios **servidores van a actuar como clientes** que piden soporte a otros servidores para **obtener un servicio del que no disponen** y poder dar soporte a sus clientes.

Ventajas e inconvenientes del modelo cliente/servidor

Ventajas:

• Utiliza **clientes** que son **ligeros**, lo cual significa que no van a necesitar un hardware muy potente para funcionar, sino todo lo contrario, ya que quien realizará todos los **cálculos** necesarios será el **servidor**.

• Muestra una gran **facilidad** en la **integración entre sistemas** que comparten la información, lo cual va a permitir que dispongamos de unas **interfaces gráficas muy sencillas** de usar para el usuario.

• Predominan las interfaces gráficas que son interactivas, por lo que los usuarios pueden **interactuar con el servidor**. Haciendo uso de interfaces gráficas conseguiremos **transmitir únicamente los datos**, consiguiendo una **menor congestión en la red**.

• El **desarrollo y mantenimiento** de este tipo de aplicaciones es relativamente **sencillo**.

• Este modelo es **modular**, lo que implica que va a permitir la **integración de nuevos componentes** y tecnologías de una forma muy **sencilla**, permitiendo un crecimiento **escalable**.

• Se tiene un **acceso centralizado** de los recursos en el servidor.

• Los clientes pueden **acceder** de forma **simultánea al servidor**, compartiendo los datos entre ellos de una forma **rápida y sencilla**.

Inconvenientes:

• El **mantenimiento** de los sistemas es un poco **más complejo**, ya que interactúan **diferentes hardware y software**, lo cual puede ocasionar múltiples **errores**.

• Se tienen que **controlar** absolutamente todos los **errores** posibles del sistema.

• Se tiene que garantizar una **buena seguridad** en el sistema. Si cae el servidor, cae el servicio para todos sus clientes.

• Se debe garantizar la **consistencia** de la información transmitida.

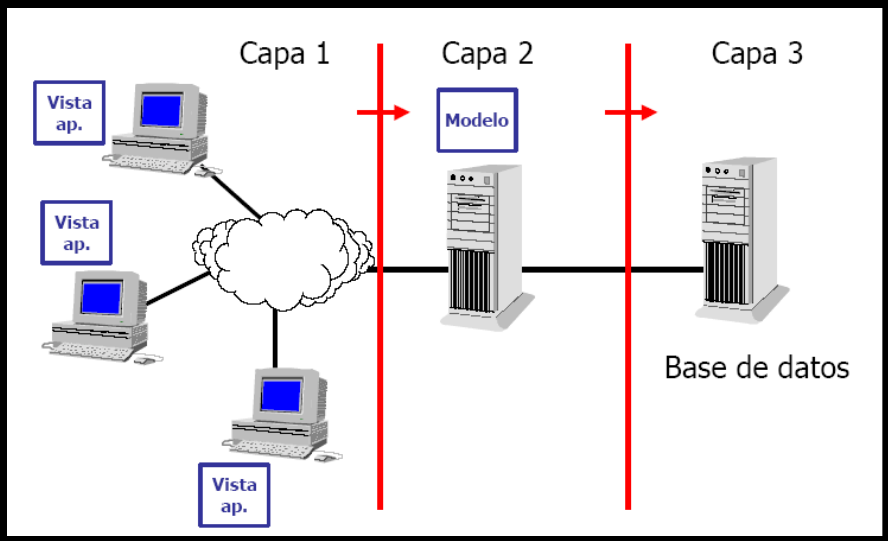
Modelos cliente/servidor

Según el número de capas (triers) que tenga, lo vamos a poder clasificar de las siguientes formas:

• Sistema de **1 capa**: Vamos a tener este tipo de modelo cuando tanto el cliente como el servidor se encuentren en el **mismo ordenador**. Este sistema **puede no llegar a considerarse** como un modelo **Cliente/Servidor**, ya que, en realidad, **no hay una conexión de red** para la comunicación del cliente y del servidor.

• Sistema de **2 capas**: Este es el **modelo tradicional** en lo que se refiere a Cliente/Servidor y es el que vamos a utilizar en nuestras aplicaciones. Aquí existen un cliente y un servidor totalmente identificables. Este modelo tiene un gran inconveniente: tiene una **bajísima escalabilidad**, pudiendo llegar a **sobrecargarse** si hay **muchos clientes** o si los clientes son pocos, pero mandan una **gran cantidad de peticiones** al servidor.

• Sistema de **3 capas**: Este modelo fue diseñado para mejorar las limitaciones del modelo de 2 capas, añadiendo, para ello, una nueva capa con un **nuevo servidor**, que puede ser o bien un **servidor de aplicaciones**, el cual se encargará de **interactuar** con los clientes y de enviar las peticiones al servidor de datos; o bien un **servidor de datos**, que será el encargado de recibir las peticiones del servidor de aplicación con lo requerido por los clientes para **resolverlas**. Sería posible agregar **tantos servidores de datos como necesitemos** sin problema alguno para mejorar el **rendimiento** del sistema.



• Sistema de **n capas**: Este modelo fue diseñado para mejorar las limitaciones del modelo de 3 capas y, a partir de aquí, podremos añadir **tantas capas de servidores como sean necesarias**, permitiendo **separar funcionalidades** y mejorar el rendimiento del sistema de una forma considerable. Un ejemplo de este tipo de sistemas son los conocidos programas o **aplicaciones P2P**, como **Napster, Gnutella, KaZaA, eMule, BitTorrent, BitCoin**...

Comunicación en el modelo cliente/servidor

El modelo Cliente/Servidor tiene un sistema de comunicación íntegro a través de intercambio de mensajes. Estos mensajes contendrán toda la información necesaria para poder ser identificados y enviados a donde se les solicite.

En este sistema, la pérdida de mensajes es un factor importantísimo a tener en cuenta, ya que podríamos estar enviando un fichero que ocupa 3 MB en varios mensajes. Si se pierden unos cuantos por el camino, porque no sean bien redireccionados, por ejemplo, el resultado de la reconstrucción del fichero de 3 MB estaría corrupto, por lo que quedaría inutilizable. Para evitar que se pierdan mensajes en este tipo de comunicaciones, **se enviará por cada mensaje recibido** un mensaje de **confirmación al emisor**. Estos son los **mensajes** conocidos como **ACK** o **acknowledgement** (reconocimiento) en inglés. En el caso de que **un mensaje no llegue** correctamente al receptor o se **demore mucho** tiempo su llegada (se considerará que se ha producido algún **error**), **no se enviará** el mensaje de ACK al emisor; en consecuencia, este entenderá que el mensaje no ha llegado, por lo que lo **volverá a enviar** y **esperará** de nuevo la llegada de su **ACK**.

El método de los envíos de ACK es muy efectivo y útil, pero la verdad es que es bastante lento, ya que deberemos esperar absolutamente todos los ACK de todos los mensajes que enviemos, además de que se **producirá más tráfico** en la red, con la posible congestión que puede implicar. Esto se puede **mejorar** de una forma muy simple, ya que, si permitimos que el receptor **envíe varios ACK a la vez**, en lugar de enviarlos de uno en uno, conseguiremos una **mejora** considerable en la **transmisión** de la información. Con este método, el emisor podrá enviar **paquetes de X mensajes** y esperará la **llegada** de sus correspondientes **ACK en grupo**. Debemos tener en cuenta que con este método de enviar varios paquetes y recibir varios ACK, puede suceder que tanto los paquetes como los ACK lleguen **desordenados**, o que, incluso, alguno de ellos **se pierda** por el camino, siendo necesario llevar un **control de errores** en esta técnica. Para gestionar estos errores, podremos utilizar un **vector de ACK**, compuesto por **pares** que indican el **número del mensaje enviado** y si se ha **recibido o no** su correspondiente **ACK**.

Procedimiento para programar una aplicación con el modelo Cliente/Servidor, por parte del **servidor**:

1. Decidir un puerto: El **puerto** elegido será utilizado tanto por el **servidor** como por los **clientes**.
2. Esperar peticiones: Será el servidor quien quede escuchando permanentemente que se conecte un cliente. Una vez **conectado** el cliente **crearemos un Socket** para la comunicación entre cliente y servidor.
3. Envío y recepción de información: Una vez conectados el cliente y el servidor por medio de un Socket deberemos **obtener los flujos** de escritura y lectura necesarios.
4. Cerrar el Socket: Una vez que se ha terminado la trnasmisión de información entre el cliente y el servidor se deberá cerrar el Socket **para evitar errores** indeseados. Este es un de los

pasos más importantes, ya que si se nos olvida cerrar el Socket nuestro programa va a estar sujeto a errores indeseados.

Procedimiento para programar una aplicación con el modelo Cliente/Servidor, por parte del **cliente**:

1. Conectarse con el servidor: El cliente deberá conectarse al servidor mediante su **IP o nombre y un puerto** previamente **conocido (el mismo que use el servidor)**. Una vez que se haya realizado correctamente dicha **conexión** se creará el **Socket** para la comunicación.
2. Envío y recepción de información: Una vez conectados el cliente y el servidor por medio de un Socket deberemos **obtener los flujos** de escritura y lectura necesarios.
3. Cerrar el Socket: Una vez que se ha terminado la transmisión de información entre cliente y servidor se deberá cerrar el Socket para **evitar posibles errores** indeseados, al igual que lo debía de cerrar el servidor.

Diseño de una aplicación cliente/servidor

El modelo cliente/servidor tiene dos partes: la del cliente y la del servidor.

Cuando se diseña una aplicación de este tipo, lo más normal es que sea bastante compleja, por lo que habrá **dos equipos de desarrollo**, uno para desarrollar la parte del **servidor** y otro que se encargue de desarrollar la parte del **cliente**. Con este tipo de desarrollo, se gana mucho tiempo, ya que lo único en lo que tendrán que **estar de acuerdo** será en la **forma en la que se comunican** los clientes con el servidor, si hay algún **mensaje de inicio, de confirmación, de parada**... y de situaciones sobre comunicación por el estilo.

Sería recomendable seguir este tipo de diseño, aunque la aplicaióc sea simple, **definiendo cada uno una parte y poniéndolas en común**.

Diagramas de estados

Cuando utilizamos el modelo Cliente/Servidor para programar aplicaciones, uno de los **fallos** principales con respecto a la **seguridad** que nos vamos a encontrar son:

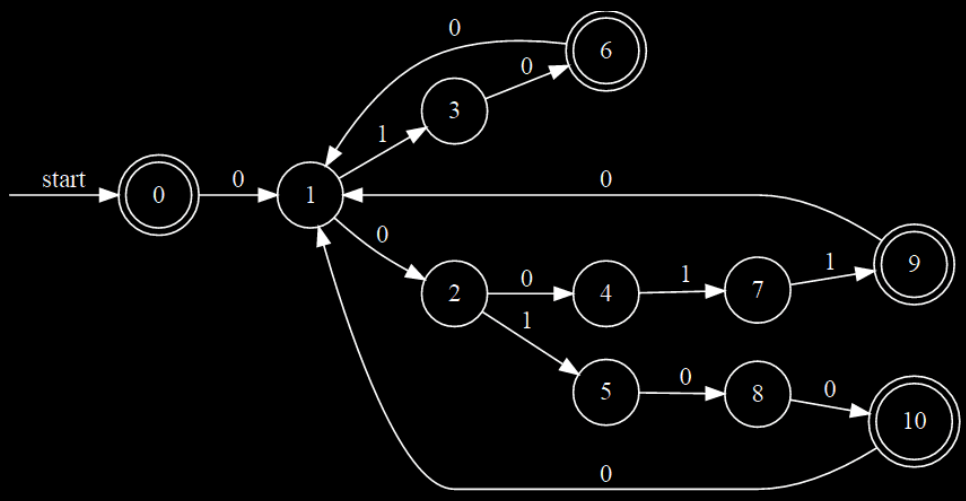
> • Que un **cliente** pueda realizar **operaciones no autorizadas**. En este caso, el servidor no va a poder procesar una petición enviada por un **cliente que no tiene privilegios para ello**, por ejemplo: un cliente que se encarga de realizar peticiones de precios de productos, de repente empieza a realizar peticiones de login.
>
> • Que los **mensajes no** estén **bien formados**. Puede ocurrir, por ciertos eventos, que un cliente envíe al servidor un mensaje y que éste no esté formado, bien porque **le falte** información necesaria o porque la información que contenga **sea errónea**.

Para evitar estos errores es de vital importancia que se utilice un **diagrama de estados o autómata**, para crear el **comportamiento que desarrollará el servidor**.

Con ellos podremos definir todos y cada uno de los **estados** o comportamientos por los que **va a pasar nuestro servidor**, definiendo qué **información debe llegar** a cada uno de ellos, hacia qué **estado se dirigirá** cuando termine su ejecución y qué **comportamientos** deberá tener cuando **llegue información errónea** a cada uno de los estados posibles en su ejecución.

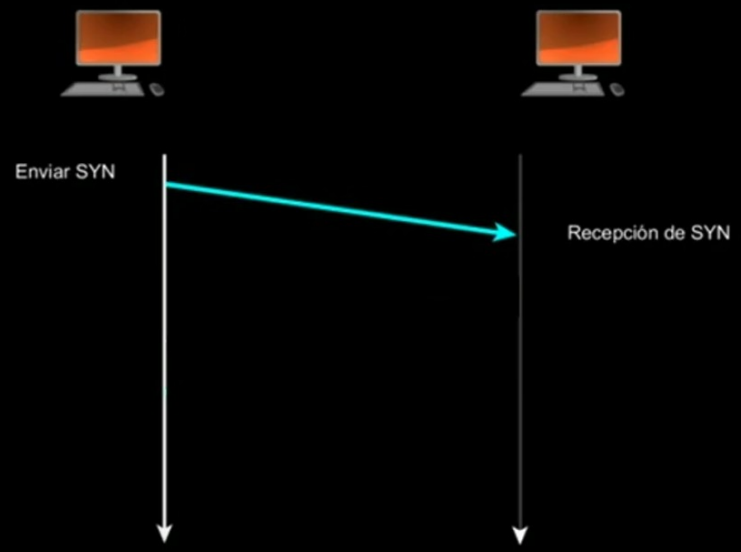
Para ello, deberemos definir en nuestro servidor **dos variables**, para almacenar:

- el **estado** en el que nos encontramos,
- el **comando** que queremos ejecutar de una lista de posibles casos.

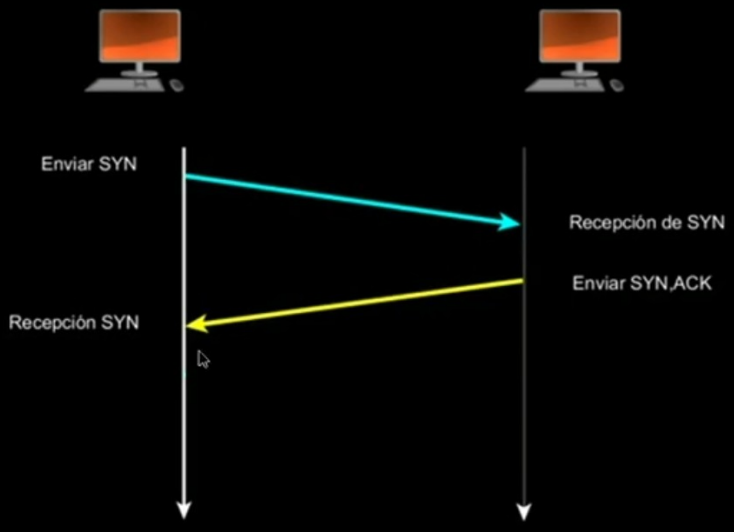


Conexión TCP simple paso a paso

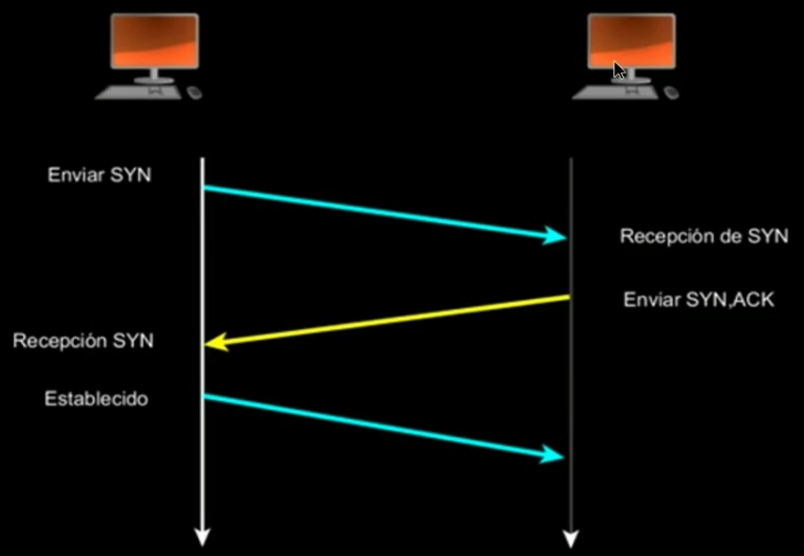
1. Sincronización del cliente y del servidor. Se envía un mensaje de sincronización, senominado SYN. Una vez realizada la sincronización podrán enviar y recibir información sin problemas. El cliente enviará un mensaje al servidor (SYN)



2. Una vez que el servidor recibe el SYN correctamente, este envía el ACK del mensaje al cliente (mensaje de confirmación al emisor-cliente de que ha llegado correctamente el paquete).



3. Una vez que el cliente recibe el ACK de la sincronización, pueden empezar a enviar paquetes. El envío de paquetes seguirá este procedimiento, se envía un mensaje y se espera su ACK.


Enviar SYN
Recepción de SYN
Enviar SYN,ACK
Recepción SYN
Establecido

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Protocolos nivel de aplicación

Vamos a recordar qué es un protocolo y vamos a estudiar **algunos de los protocolos** que se usan en comunicaciones y que actúan en la **capa de aplicación**:

- **DNS,**
- **FTP,**
- **SMTP,**
- **POP3,**
- **IMAP,**
- **HTTP**.

Vamos a analizar qué son y para qué sirve cada uno de ellos.

Servicios en red

Para programar aplicaciones que se comuniquen mediante el envío y recepción de mensajes por red, necesitaremos una red de **ordenadores interconectados** entre sí. Esta la definimos formalmente como un **sistema de telecomunicaciones interconectado** entre sí y que tienen la finalidad de **compartir información** o recursos.

Aprovechando esta distribución de datos, se podrá hacer un mejor uso de ellos por parte de todos los usuarios que se encuentren dentro de la red informática, es decir, se va a mejorar, en gran medida, el rendimiento y la eficacia del sistema.

La utilización de **redes de ordenadores** (servicios en red) presenta las siguientes **ventajas**:

- Reduce los **costes** tanto en hardware como en software.
- Podremos crear **grupos de trabajo**, los cuales nos ayudarán a tener organizados a los usuarios.
- Mejora de forma notable la **administración**, tanto de los equipos físicos como de las aplicaciones.
- Se mejora la **integridad** de los datos que hay en el sistema, que será mayor cuantos **más nodos** tenga el mismo.
- Lo mismo para la **seguridad** en los datos.
- Se facilita la **comunicación**.

Los **servicios en red se van a clasificar en**:

- Servicios de administración/configuración:

  Gracias a estos servicios, se va a facilitar tanto la **administración** como la **gestión** de **configuración de los equipos**.

  Un ejemplo son los servicios **DHCP y DNS**.

- Servicios de acceso remoto:

  Con ellos, podremos gestionar las **conexiones de los usuarios a nuestra red** desde lugares remotos.

  Un ejemplo son los servicios **Telnet y SSH**.

- Servicios de ficheros:

  Gracias a estos servicios, podremos ofrecer grandes cantidades de **almacenamiento**. Un ejemplo es el servicio **FTP**.

- Servicios de impresión:

  Nos facilitan la opción de **imprimir documentos** de forma **remota**.

- Servicios de información:

Nos permiten **obtener ficheros en función de** su **contenido**.

Un ejemplo es el servicio **HTTP**.

- Servicios de comunicación:

  Gracias a estos servicios, los usuarios podrán comunicarse a través de **mensajes**.

  Un ejemplo es el servicio **SMTP**.

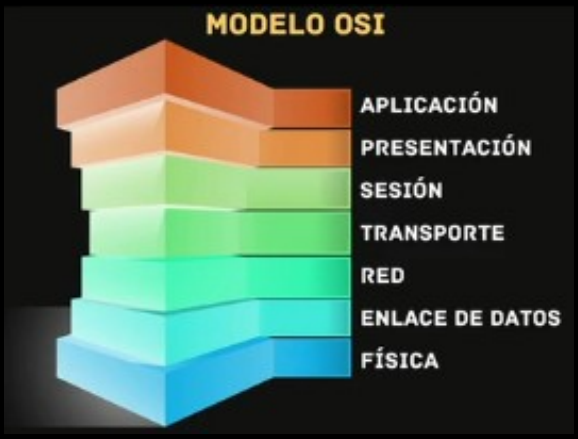
Definición de servicio en red

*Programa o **software** que va a proporcionar una determinada **funcionalidad a nuestro sistema**. Están basados por norma en una serie de **protocolos y estándares**,*

Protocolos a nivel de aplicación

Actualmente, el conjunto de **protocolos más importante** lo conforman los englobados en el modelo **TCP/IP**. Sobre este modelo están **basadas todas** y cada una de las **comunicaciones** que realizamos cotidianamente.

Vamos a destacar, de entre todas las capas que conforman el modelo TCP/IP, la **capa de Aplicación**, que, si recordamos, es la **capa** que está **más arriba** dentro de este modelo. La capa de Aplicación es la que contiene absolutamente todos los **protocolos** que van a estar **relacionados con los servicios en red**.



En la capa de transporte donde, utilizando el protocolo TCP o UDP, enviaremos los datos al destinatario.

En esta capa, se van a **definir los protocolos** que se van a utilizar por todas las **aplicaciones** que desarrollemos que necesiten **intercambiar datos**.

Cabe destacar que el **número de protocolos** que se encuentra en la capa de Aplicación **aumenta** continuamente, ya que no dejan de aparecer **nuevos servicios** que **demandan los usuarios** y que deben basarse en algún protocolo.

Algunos de los **protocolos más importantes** que nos vamos a encontrar dentro de la capa de Aplicación del modelo TCP/IP son los siguientes:

- FTP: Este protocolo se encarga de la transferencia de **ficheros**. Su nombre lo conforman las siglas de Protocolo de Transferencia de Ficheros (**File Transfer Protocol**).
- SMTP: Mediante el que podremos enviar **correos** electrónicos.
- HTTP: Es el que nos habilita la **navegación** por Internet. Es el usado por todos los navegadores **web**.
- Telnet: Este protocolo nos permitirá poder acceder de forma **remota** a un **ordenador** y **manejarlo**.

• SSH: Nos va a permitir una **gestión segura** de forma **remota** de otro ordenador. Es la **versión mejorada del Telnet**.

• NNTP: Con él podremos realizar el envío de **noticias** por la red. Su nombre lo conforman las siglas de Protocolo de Transferencia de Noticias (**Network News** Transport Protocol).

• IRC: Con el que podremos **chatear** vía Internet. Es el protocolo que siguen los chats de Internet (**Internet Relay Chat** o charla interactiva en Internet).

• DNS: Este protocolo nos permitirá **resolver direcciones de red**.

Cómo comenzar una aplicación que use servicios en red

Aunque sea posible una implementación desde cero utilizando Sockets, esto sería una tarea bastante ardua, ya que estos **protocolos no son sencillos**.

Existen **bibliotecas ya predefinidas en la JDK de Java** que nos van a permitir **trabajar con estos protocolos**, facilitándonos el trabajo, en gran medida.

Un ejemplo puede ser la biblioteca **Proyect Lombok**, la cual **automatiza el código generado de los POJO**.

También es importarte tener en cuenta que, probablemente, podremos encontrar **bibliotecas de terceros** que trabajen con estos protocolos y que, seguramente, serán más sencillas de utilizar.

Con respecto a este tipo de bibliotecas, podemos destacar los **controladores** que nos ofrecen bases de datos como MySQL para poder **conectar nuestros proyectos** a ellas.

El protocolo DNS

Todos los **dispositivos**, ya sean ordenadores, portátiles, smartphones, tablets, videoconsolas, etc., que se puedan **conectar a una red TCP/IP** se van a **identificar mediante una dirección IP** del estilo 192.168.1.1.

Las direcciones IP que se utilizan **normalmente** son del formato **IPv4**, las cuales se componen de **cuatro números**, que estarán comprendidos en el intervalo de **0 a 255**, separados por un **punto** unos de los otros.

Los **ordenadores** pueden **utilizar estos valores** muy fácilmente para **comunicarse** entre los diferentes **nodos**, aunque para los humanos puede ser bastante complicado acordarse de todas las IP de los nodos o hosts a los que queremos acceder.

El servicio **DNS**, cuyas siglas hacen referencia a Sistema de Nombre de Dominio (**Domain Name System**), se encarga de resolver el problema de identificar a un nodo mediante una dirección IP. Este servicio nos va a permitir **asignar nombres de dominio a un nodo**, por ejemplo www.google.es, lo cual, para los humanos, es más fácil de recordar.

El servicio DNS se podrá utilizar en **cualquier dispositivo** que se conecte a la **red**.

El objeto principal del servicio DNS es el de **traducir las direcciones IP a nombres de dominio** y **viceversa** en cada uno de los dispositivos que están conectados a la red, pudiendo identificarlos de una forma mucho más sencilla.

Esto no quiere decir que tengamos que hacerlo así obligatoriamente. Si queremos **acceder** a los nodos de nuestra red **mediante direcciones IP**, **podremos** hacerlo sin ningún problema.

Ventajas que nos ofrece el servicio DNS:

- Posibilita que **varios nombres de dominio** compartan una **misma dirección IP** (por ejemplo, si se tienen distintas marcas comerciales y se desarrollan distintas webs, pero se desea que todas apunten al mismo servidor).
- Permitirá que **varias IP** estén compartidas **por varios nombres de dominio**.
- Un **nodo** de la red **podrá cambiar de nombre de dominio** sin tener que cambiar de dirección IP.

El servidor **DNS** va a utilizar una **base de datos distribuida**, que es la encargada de **almacenar** toda la información asociada a **nombres de dominio** en redes como Internet.

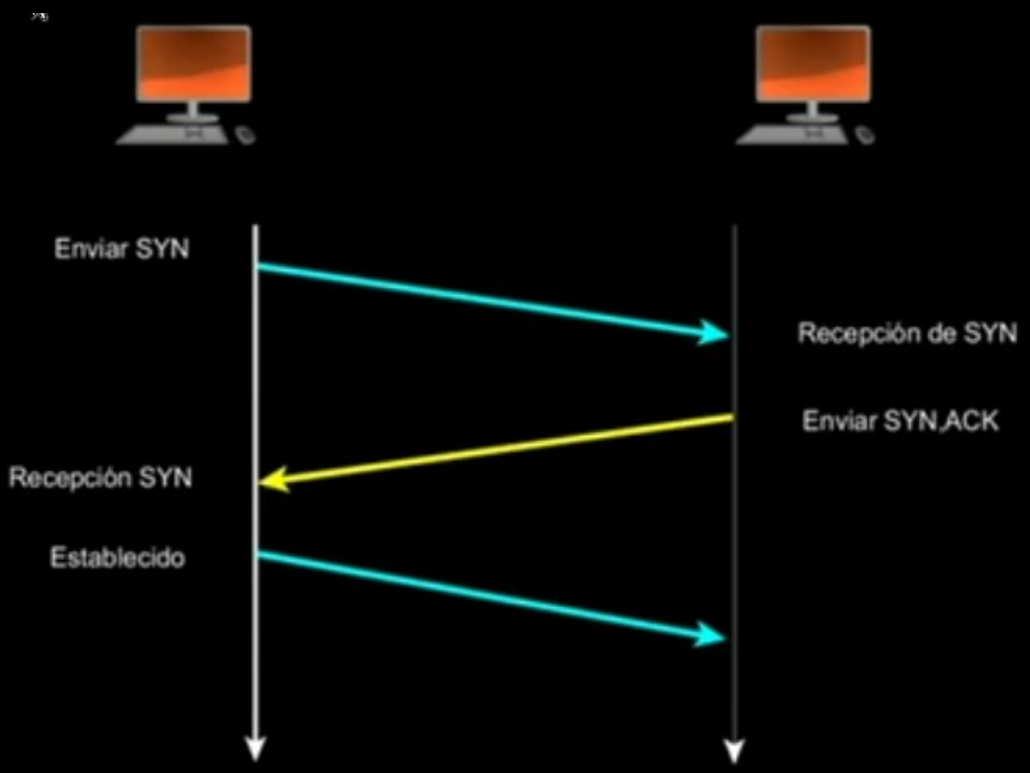
Conexión, comunicación y desconexión

La información pasa de capa en capa desde la capa Aplicación **hasta llegar a la capa de Transporte**, donde mediante una API que utilice sockets se podrá enviar la información.

Un socket es la representación que nos va a dar **java** para **realizar una transmisión** de información en red. En otros lenguajes de programación tendremos otras representaciones equivalentes a sockets.

Los **pasos para crear una comunicación** en red **mediante sockets** son:

1. Se **crean los sockets** tanto en cliente como en servidor.
2. El **servidor** establece un **puerto de escucha**.
3. El **cliente se conecta** al servidor mediante el **puerto** de escucha.
4. El **servidor acepta la conexión** del cliente.
5. Se crean los **flujos de datos** y se realiza el **intercambio** de información.
6. Se **cierran las conexiones** tanto del servidor como del cliente.



El protocolo FTP

Hoy en día, una de las funcionalidades más utilizadas en la red es la de poder **transferir ficheros**, sean del tamaño que sean, desde varios ordenadores, en una red local o desde Internet.

Para proporcionarnos este servicio surgió el protocolo FTP, que proporcionará las funcionalidades necesarias para definir un **estándar en la transferencia de ficheros** en las **redes TCP/IP**.

Principios fundamentales del protocolo FTP (son 2):

- Nos va a permitir **intercambiar ficheros** entre ordenadores remotos que estén interconectados mediante una **red**.
- Permite transferir los ficheros a una **velocidad** bastante **alta**.

La gran **desventaja** de este protocolo es que transmite toda la información en **formato de texto plano**. Esto significa que la información transmitida **no está codificada**, por lo que, si alguien realiza una **conexión intermedia**, podrá **obtener los ficheros** transmitidos de un extremo a otro. La transferencia se realiza en texto plano para proporcionar una **mayor velocidad** de transferencia de información, a costa de una **pobre seguridad**.

El problema de la seguridad será solucionado gracias a la **encriptación** de la información a través del **protocolo SFTP**, **Secure File Transfer Protocol** o Protocolo de Transferencia de Ficheros Seguro, usado **junto al protocolo SSH**.

El protocolo FTP usa el **puerto 20** para la transmisión de **datos** y el **21** para la transferencia de **órdenes**.

Algunas de las **características del protocolo FTP** son:

• Permite **conectar usuarios** en remoto al servidor FTP.

• Hay un **límite en el acceso** al sistema de archivos mediante un **sistema de privilegios** de los usuarios.

• Tiene **dos modos de conexión**:

- modo **activo**, con el que habrá **dos conexiones distintas**,
- modo **pasivo**, con el que **no habrá dos conexiones** distintas.

Protocolo SMTP

El servicio de correo electrónico permite tanto enviar como recibir mensajes con o sin archivos de una forma muy rápida a través de Internet.

El servicio de la capa de Aplicación que nos va a permitir usar el **correo electrónico** es el **protocolo SMTP**, cuyas siglas significan Protocolo para Transferencia de Correo Simple (**Simple Mail Transfer Protocol**), el cual sigue el **modelo Cliente/Servidor**. Por lo tanto, deberemos tener un servidor de correo electrónico y un cliente de correo electrónico para poder usarlo correctamente.

El **servidor** de correo electrónico **creará** una serie de **cuentas** para los usuarios, que **tendrán** lo que se denomina un **buzón**, donde estarán **almacenados todos sus correos** correspondientes.

Los **clientes** serán los encargados de **descargar y elaborar los correos** electrónicos.

El protocolo SMTP utiliza el **puerto 25** y es el encargado de realizar el transporte del correo desde la máquina del **usuario hasta el servidor**, que lo almacenará para que el destinatario del mismo pueda acceder a él.

Cuando el destinatario desee ingresar en su cuenta de correo electrónico, tendrá **dos opciones para acceder a los mensajes**:

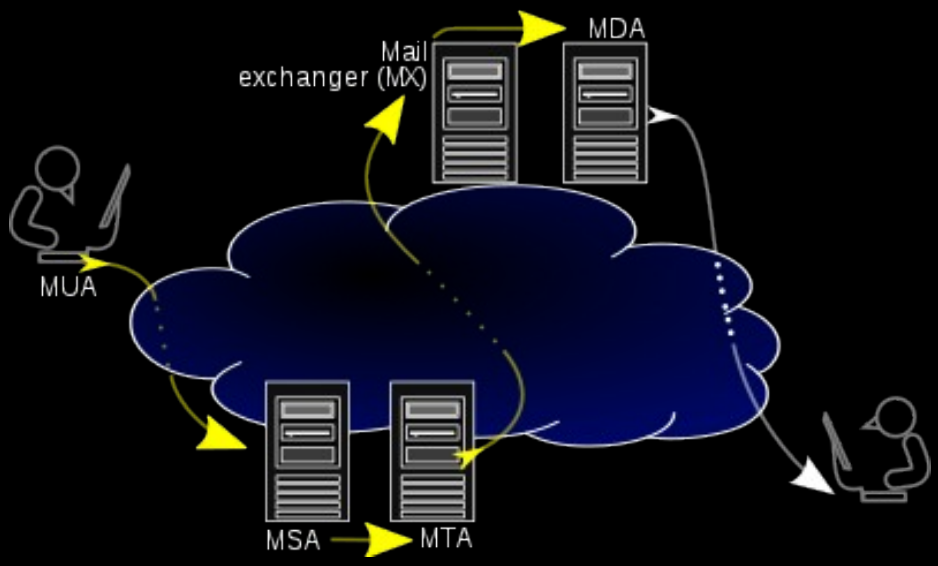
- descargarlos en su **máquina local** mediante el protocolo POP3,
- consultarlos **directamente en el servidor**, mediante el protocolo IMAP.

Estos protocolos nos permitirán no solo enviar texto en nuestros correos electrónicos, sino **cualquier tipo de documento digitalizado**.

Comandos SMTP para realizar transferencia de correo:

- **HELO (hola)**: Comando usado para **abrir una sesión** con el servidor.
- **MAIL FROM**: Comando usado para indicar **quién es el emisor** del correo.

Procesamiento del correo



MUA: agente de usuario de correo.

MSA: agente de sumisión de correo.

MTA (Mail Transfer Agent): Mail Transfer Agent, Agente de Transferencia de Correo.

Para lograr la localización del servidor objetivo, el MTA divisorio tiene que usar el sistema de nombre de dominio (DNS) para lograr la búsqueda del **registro interno de cambiado de correo** conocido como **registro MX** para la esfera del recipiente (la parte de la dirección a la derecha). Es en ese instante cuando el registro de MX devuelto contiene el nombre del anfitrión objetivo.

MDA: agente de entrega de correo.

El protocolo HTTP

El protocolo HTTP, cuyas siglas significan Protocolo de Transferencia de Hipertexto (**HyperText Transfer Protocol**), es el protocolo encargado de que podamos navegar por Internet de forma correcta. Está compuesto por una serie de **normas** que posibilitan una **comunicación y transferencia** de información entre **cliente y servidor**.

La información que transfiere este protocolo son las **páginas HTML**.

El protocolo HTTP se encarga de definir toda la **sintaxis de comunicación** que van a usar tanto el cliente como el servidor, siendo algunas de las **reglas** más importantes las siguientes:

- HTTP sigue el **modelo de petición-respuesta** que aplica al servidor y al cliente.
- El **puerto** que utiliza es el **80**, aunque ofrece la **posibilidad de cambiarlo**.
- Al **cliente** se le denomina como agente del usuario, o **user agent** en inglés.
- Toda la **información que se transmite** se conoce como **recursos** y están identificados mediante una **URL** del tipo **http://**www.google.es.
- Los **recursos también** pueden ser **ficheros**, una consulta de una **base de datos**, un **resultado de una operación** realizada por un programa, **etc**.

El protocolo HTTP **no tiene estado**. Con esto, se refiere a que **no va a recordar** absolutamente nada de **conexiones** anteriores.

Funcionamiento del protocolo HTTP:

- El **usuario accede** a una **URL**, pudiendo **indicar el puerto** en la misma.
- El **equipo cliente descompondrá la información** de la **URL**, diferenciando **todas sus partes**, como el nombre de **dominio**, la **IP**, el **puerto**, etc.
- El **cliente** establece una **conexión con el servidor**.
- El **servidor contesta** al cliente y **envía** el código **HTML**.
- El **cliente visualiza el HTML** y cierra la conexión.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red
Clases Socket y DatagramPacket
Sockets TCP y UDP

Sockets

Los Sockets son un **mecanismo de comunicación entre aplicaciones**, utilizado principalmente en redes de comunicación. Podemos entender los Sockets, en sí, como uno de los **extremos de una conexión bidireccional** entre dos **programas** que se están ejecutando en **red**. Por lo tanto, representan los **extremos del canal de comunicación** que necesitamos.

Los Sockets van a identificarse mediante una **dirección IP** y un **puerto** en el que transmitirán.

Podremos utilizar **dos tipos de Sockets**:

- Sockets TCP (orientados a conexión):
  - ➔ Este tipo de Sockets se van a utilizar para establecer una comunicación en red utilizando el protocolo TCP.
  - ➔ Esta conexión **no comenzará hasta** que los dos **Sockets** estén **conectados**.
  - ➔ La **forma de transmitir** información en una conexión con Sockets TCP será en **bytes**.
  - ➔ Para las aplicaciones que utilizan Sockets TCP, podremos utilizar las siguientes **clases en Java**: **Socket y ServerSocket**, que implementarán el cliente y el servidor de la conexión, respectivamente.
  - ➔ Vamos a utilizar la **clase Socket** tanto para **transmitir** como para **recibir datos**, mientras que la clase **ServerSocket** se encargará de implementar el servidor y su única tarea será **esperar** a que un **cliente** desee establecer una conexión con el servidor.
- Sockets UDP (no orientados a conexión):
  - ➔ Este tipo de Sockets se utilizará para establecer una comunicación en red utilizando el protocolo UDP.
  - ➔ La **forma de transmitir** información en una conexión con Sockets UDP será en **datagramas**.
  - ➔ Estos Sockets son mucho **más rápidos** que los TCP, aunque **menos seguros**.
  - ➔ Para las aplicaciones que utilizan Sockets UDP, podremos utilizar las **clases en Java DatagramSocket**.

Sockets en otros lenguajes de programación

Ya hemos visto las clases que nos proporciona el lenguaje de programación Java para la implementación de Sockets, pero ¿cuáles son en otros lenguajes?

En **C++** tenemos también la **clase socket**, en **Python** tenemos un **módulo** llamado también **socket** y en **C#** tenemos la **clase Sockets**.

Como podemos ver, los Sockets están presentes en cualquier lenguaje de hoy en día, con una **nomenclatura similar**, y ofreciendo la **misma funcionalidad**.

Servidor TCP

Pasos para crear aplicaciones que usen Sockets en Java:

- **Crear o abrir** los **Sockets** tanto en el **cliente** como en el **servidor**.
- **Crear los flujos de entrada o salida** tanto en el **cliente** como en el **servidor**.
- **Cerrar los flujos** y los **Sockets**.

Además de todo esto, como es posible que se produzcan **errores** en las transmisiones en red de información, será necesario llevar el control de excepciones mediante el **bloque try-catch**.

Vamos a crear una **clase llamada ServidorTCP**, que actuará como servidor de nuestra aplicación. En esta clase, vamos a crear un constructor en el que crearemos toda la funcionalidad que deseemos y, además, crearemos un método main en el que implementaremos nuestro servidor.

Servidor TCP simple

```
public class ServidorTCP {
	/**
	 * En esta clase, vamos a crear una variable que indicará
	 * el puerto donde escuchará nuestro servidor, por ejemplo,
	 * el puerto 6000.
	 */
	private final int PUERTO = 6000;

	public ServidorTCP() {
		try {
			/**
			 * Para implementar el servidor, crearemos un
			 * objeto de la clase ServerSocket con el puerto
			 * deseado.
			 */
			ServerSocket skServidor =
					new ServerSocket(PUERTO);
			/**
			 * Una vez hecho esto, deberemos crear un bucle
			 * infinito para atender a todos los clientes que
			 * se conecten al servidor, obteniéndolos mediante
			 * el método accept(), que devolverá el cliente
			 * conectado.
			 *
			 * Escucho a los clientes
```

```
             */

            while (Boolean.TRUE) {
                  Socket skCliente = skServidor.accept();
                  /**
                   * El siguiente paso será crear los flujos de
                   * entrada y de salida para poder llevar a
                   * cabo la comunicación, mediante los métodos
                   * readUTF() y writeUTF(), respectivamente.
                   */
                  // Obtengo el flujo de entrada del cliente
                  // (Mensajes que recibe del cliente)
                  InputStream aux2 = skCliente.getInputStream();
                  DataInputStream flujorecibir =
                              new DataInputStream(aux2);
                  // Obtengo el flujo de salida del cliente
                  // (Mensajes que envía al cliente)
                  OutputStream aux = skCliente.getOutputStream();
                  DataOutputStream flujoenviar =
                              new DataOutputStream(aux);
                  // Manda un mensaje al cliente
                  flujoenviar.writeUTF("Hola cliente");
                  // Por último, deberemos de cerrar el Socket.
                  // Cierro el socket servidor
                  skCliente.close();
            }
      } catch (IOException e) {
            System.out.println("Error en el servidor: "
      + e.getMessage());
      }
   }
}
```

Cliente TCP

Una vez creado nuestro servidor, vamos a pasar a crear un cliente, el cual tendrá que:

1. Crear un **Socket** y **conectar con el servidor** mediante su **IP y puerto**.
2. Crear los **flujos** de entrada o salida.
3. Una vez terminada la comunicación, **cerrar los flujos y los Sockets**.

De igual forma que pasaba con el servidor, es posible que se produzcan errores en las transmisiones en la red de comunicación, será necesario, por tanto, llevar el **control de excepciones** mediante el bloque try-catch.

Vamos a crear una **clase** llamada **ClienteTCP**, que actuará como uno de los **clientes que se conectarán al servidor** de nuestra aplicación. En esta clase, vamos a crear un **constructor** en el que montaremos toda la **funcionalidad** que necesitemos, y además, implementaremos un método **main** en el que **crearemos nuestro cliente**.

```
public class ClienteTCP {
	/**
	 * crearemos una variable que indicará el puerto donde
	 * escuchará nuestro servidor, por ejemplo, el puerto
	 * 6000 y otra con el host donde se aloja el servidor
	 * (en nuestro caso, como será en la misma máquina,
	 * será localhost).
	 */
	private static final String HOST = "localhost";
	private static final int PUERTO = 6000;

	public ClienteTCP() {
		try {
			/**
			 * Para crear el cliente, crearemos un objeto de
			 * la clase Socket con el host y el puerto deseado.
			 */
			Socket skCliente =
					new Socket(HOST, PUERTO);
			/**
			 * Una vez hecho esto, el servidor nos aceptará,
			 * por lo que el siguiente paso será crear los
			 * flujos de entrada y de salida para poder llevar
			 * a cabo la comunicación, mediante los métodos
			 * readUTF() y writeUTF() respectivamente.
			 */
```

```
                // Obtengo el flujo de entrada del cliente creado
                // (Mensajes que recibe el cliente del servidor)
                InputStream aux = skCliente.getInputStream();
                DataInputStream flujo =
                            new DataInputStream(aux);
                /**
                 * Por último, deberemos de cerrar el Socket.
                 */
                // Cierro el socket
                skCliente.close();

            } catch (IOException ex) {
                System.out.println("Error -> " + ex.toString());
            }

        }

}
```

Ejemplo de aplicación con Sockets TCP

¿Cómo crear una aplicación de comunicación en red?

Vamos a ver cómo podemos crear una aplicación de comunicación en red mediante el protocolo TCP, utilizando sockets TCP. Para ello vamos a crear dos clases, una que va a representar el servidor y otra que va a representar el cliente.

En la clase **ServidorTCP** vamos a tener lo siguiente:

- Una **variable int** que va a ser el **puerto** por el que va a escuchar el servidor y en el que tendrán que hablar los clientes. Por ejemplo, el puerto 6000.
- Y una variable MAX_CLIENTES que valdrá 3, que serán los clientes que va a dar soporte a nuestro servidor.

En el constructor de nuestra clase tenemos dentro de un **bloque try-catch** lo siguiente:

1. Mediante la clase ServerSocket hemos creado un socket (skServidor) al servidor con el puerto en el que escuchamos.
2. Ahora con un bucle for desde cero hasta el máximo de clientes **esperaremos a que se conecte un cliente** mediante el método accept, el cual será devuelto en un objeto del tipo socket.
3. Ahora vamos a obtener el flujo de salida del cliente por el que podremos enviarle mensajes.
4. Y una vez hecho esto mediante el método writeUTF podremos **enviar el mensaje** que queramos,
5. y cerramos el socket del cliente ya que está **servido**.
6. Dentro de la clase servidorTCP tenemos un método main que creará un servidor.

```
public class ServidorTCP {
    private static final int PUERTO = 6000;
    private static final int MAX_CLIENTES = 3;
    public ServidorTCP() {
        try {
            // Creo el socket servidor que escucha
            // en el puerto 6000
            ServerSocket skServidor =
                    new ServerSocket(PUERTO);
            System.out.println("Escucho el puerto "
                    + PUERTO);
            // Escucho a un cierto número de clientes
            for (int numCli = 0; numCli < MAX_CLIENTES; numCli++)
            {
                // Escucho a un cliente
                Socket skCliente = skServidor.accept();
                // Cuando escucha un cliente da un aviso
                System.out.println("Sirvo al cliente "
                + numCli);
```

```
                // Obtiene el flujo de salida del cliente
                // (Mensajes que envía al cliente)
                OutputStream aux = skCliente.getOutputStream();
                DataOutputStream flujo =
                          new DataOutputStream(aux);
                // Manda un mensaje al cliente
                flujo.writeUTF("Hola cliente " + numCli);
                // Cierro el socket servidor
                skCliente.close();
            }
            System.out.println("Demasiados clientes por hoy.");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
    // Método main del servidor
    public static void main(String[] args) {
        new ServidorTCP();
    }
}
```

En la clase **ClienteTCP** tendremos la representación de un cliente que se conectará a nuestro servidor TCP.

Esta clase tiene:

- una **variable** del tipo **String** que es el **host** donde se va a conectar nuestro cliente, que en este caso tiene el valor "**localhost**".
- Una **variable** del tipo **int** que es el **puerto**, que tiene como valor 6000, que es el mismo puerto en el que escucha nuestro servidor.

Dentro del constructor de nuestro cliente y dentro de un **bloque try-catch** para poder gestionar todas las acciones que se nos presenten. Tenemos lo siguiente:

1. Creamos un socket con la clase socket que se conectará a un puerto y a un host. Aquí será donde se conectará al servidor.
2. Una vez conectado y aceptado por el servidor tendremos que obtener el flujo de entrada del cliente, que serán los mensajes que recibe el cliente, es decir, los mensajes enviados por el servidor.
3. Una vez hecho esto podremos obtener el mensaje que se nos envíe por el servidor mediante el método readUTF, el cual devolverá un String.
4. Y cerraremos el socket ya que está servido.
5. Dentro de la clase cliente TCP tenemos un método main que ejecuta un cliente.

```java
public class ClienteTCP {

    private static final String HOST = "localhost";
    private static final int PUERTO = 6000;
    public ClienteTCP() {

        try {
            // Creo el socket cliente que escucha en la
            // máquina localhost y en el puerto 6000
            Socket skCliente = new Socket(HOST, PUERTO);
            // Obtengo el flujo de entrada del cliente creado
            // (Mensajes que recibe el cliente del servidor)
            InputStream aux = skCliente.getInputStream();
            DataInputStream flujo = new DataInputStream(aux);
            // Saco por pantalla el mensaje recibido del servidor
            System.out.print("Mensaje recibido del servidor: ");
            System.out.println(flujo.readUTF());
            // Cierro el socket
            skCliente.close();
        } catch (IOException ex) {
            System.out.println("Error en el cliente: " + ex.toString());
        }
    }
    // Método main del cliente
    public static void main(String[] args) {
        // Creo un cliente
        new ClienteTCP();
    }
}
```

Ejecución de la aplicación:

Primero deberemos ejecutar el **servidor**:

En la clase ServidorTCP pulsamos botón derecho y seleccionamos “run file”. Y veremos que el servidor escucha en el puerto 6000.

Y ahora ejecutaremos los clientes de la misma forma. Botón derecho en la clase cliente TCP y “run file”. Aquí vemos que el servidor nos ha enviado este mensaje:

Escucho el puerto 6000

Sirvo al cliente 0

Volvemos a ejecutar otra vez, y tenemos el cliente 1.

Volvemos a ejecutar otra vez, y tenemos al cliente 2.

Terminando el servidor su ejecución ya que solamente admite 3 clientes:

Escucho el puerto 6000

Sirvo al cliente 0

Sirvo al cliente 1

Sirvo al cliente 2

Demasiados clientes por hoy.

Sockets UDP
Clases Datagrampacket y Datagramsocket

Los Sockets UDP serán algo más sencillos de implementar que los TCP, ya que, si recordamos, el protocolo UDP no es un protocolo orientado a conexión, por lo que simplemente deberemos crear el Socket y enviar o recibir mensajes.

Para poder enviar o recibir datos a través de una conexión UDP deberemos utilizar la **clase DatagramPacket** y crear un **Socket UDP** mediante la **clase DatagramSocket**. La documentación de las clases la podemos encontrar en:

http://docs.oracle.com/javase/8/docs/api/java/net/DatagramPacket.html

https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html

La clase DatagramPacket va a representar un **datagrama**, pudiendo **enviar o recibir paquetes**, también denominados datagramas. Un datagrama no es más que un **array de bytes** enviado a una **dirección IP** y a un **puerto UDP** concreto.

Un **datagrama** tiene la siguiente forma:

datagrama

Array de bytes + Longitud + IP destino + Puerto destino

Primero implementamos el elemento DatagramSocket y luego lo agregamos al elemento contenedor o paquete DatagramPacket:

Métodos de la clase DatagramPacket:

- getData(): Este método devuelve el **array de bytes** que contiene los datos.
- getLenght(): Este método devuelve la **longitud del mensaje** a enviar o recibir.
- getPort(): Devuelve el **puerto de envío/recepción** del paquete.
- getAddress(): Devuelve la **dirección del host** remoto de envío/recepción (dirección IP).

Métodos la clase DatagramSocket:

- send(datagrama): Permite **enviar** datagramas.
- receive(datagrama): **Recibe** datagramas.
- close(): **Cierra** el Socket.
- getLocalPort(): Devuelve el **puerto donde está conectado** el Socket.
- getPort(): Devuelve el puerto **de donde procede** el Socket.

Servidor UDP

Cuando utilizamos Sockets UDP, no necesitamos realizar ningún tipo de conexión. Por este motivo, la **diferenciación entre clientes y servidores** será un poco **más complicada**.

Vamos a distinguir al servidor UDP como el que **espera un mensaje de un cliente** y lo responde, y consideraremos al cliente como el que **inicia la comunicación**.

Al igual que con el protocolo TCP, es posible que se produzcan errores en las transmisiones en red de información. Por ello, también será necesario llevar el **control de excepciones** mediante el bloque try-catch.

Vamos a crear una **clase llamada ServidorUDP**, que actuará como el servidor de nuestra aplicación. En esta clase, vamos a crear un constructor en el que montaremos toda la **funcionalidad** que deseemos, y además, implementaremos un método main, en el que **ubicaremos nuestro servidor**.

En esta clase, vamos a crear una variable que indicará el **puerto** donde escuchará nuestro servidor, por ejemplo, el puerto 6789.

Para crear el servidor, utilizaremos las **clases DatagramSocket y DatagramPacket**, para poder crear el **Socket** y los **mensajes datagramas** intercambiados. También deberemos crear un **array del tipo byte** para los datos.

Una vez hecho esto, simplemente deberemos **esperar la recepción** de un mensaje, mediante el método **receive**, pudiendo responder a la misma utilizando el método **send**.

En este caso, **no deberemos cerrar ninguna conexión**, ya que no existe.

Servidor UDP simple

```
public class ServidorUDP {
	private static final int PUERTO = 6789;

	public ServidorUDP() {
		try {
			DatagramSocket socketUDP = new DatagramSocket(PUERTO);
			byte[] bufer = new byte[1000];
			System.out.println("Escucho €el puerto " + PUERTO);
			while (true) {
				// Construimos el DatagramPacket para recibir peticiones
				DatagramPacket peticion =
						new DatagramPacket(bufer, bufer.length);
				// Leemos una peticion del DatagramSocket
				socketUDP.receive(peticion);
				// Obtengo el mensaje del cliente
				String mensajerecibido =
						new String (peticion.getData());
				// Construimos el DatagramPacket para enviar la respuesta
				String mensajerespuesta = "Hola cliente " + mensajerecibido;
				DatagramPacket respuesta =
						new DatagramPacket (
								mensajerespuesta.getBytes(),
								mensajerespuesta.length(),
								peticion.getAddress(),
								peticion.getPort()
								);
				// Enviamos la respuesta, que es un eco
				 socketUDP.send(respuesta);
			}
		} catch (IOException e) {
			System.out.println("Error: " + e.getMessage());
		}
	}
}
```

Cliente UDP

Ya hemos concretado que, en una comunicación UDP, vamos a distinguir al **cliente** como el que **inicia la comunicación**.

Nuevamente, al igual que pasaba cuando utilizábamos el protocolo TCP, es posible que se produzcan errores en las transmisiones en red de información, por lo que también será necesario llevar el **control de excepciones** mediante el bloque try-catch.

Crearemos una clase llamada **ClienteUDP**, que actuará como un cliente de nuestra aplicación. Nuevamente, en esta clase, debemos crear un constructor con toda la **funcionalidad** que deseemos y, además, crearemos el main en el que **ubicaremos el cliente UDP**.

En esta clase, vamos a crear una variable int que indicará el **puerto** donde escuchará nuestro servidor; por ejemplo, el puerto 6789, y otra variable String que indicará el **host** al que nos vamos a conectar (en nuestro caso, **localhost**), ya que tanto el cliente como el servidor van a estar en la **misma máquina**.

Para crear el cliente, utilizaremos de nuevo las **clases DatagramSocket y DatagramPacket**, para poder crear el **Socket** y los **mensajes datagramas** intercambiados. Como ocurría en el servidor UDP, deberemos crear un array del tipo **byte para los datos**.

Una vez hecho esto, simplemente deberemos **enviar un mensaje al servidor** utilizando el método send y **recibir** un mensaje del mismo mediante el método receive.

Cliente UDP simple

```
public class ClienteUDP {
	private static final int PUERTO = 6789;
	private static final String HOST = "localhost";

	public ClienteUDP() {
		try {
			String mensajeenviar = String.valueOf("Hola");
			// Creo el socket UDP
			DatagramSocket socketUDP =
					new DatagramSocket();
			byte[] mensaje = mensajeenviar.getBytes();
			// Obtengo la direccion del host
			InetAddress hostServidor = InetAddress.getByName (HOST);
			// Construimos un datagrama para enviar al servidor
			DatagramPacket peticion =
					new DatagramPacket (
							mensaje,
							mensajeenviar.length(),
							hostServidor,
							PUERTO);
			// Enviamos el datagrama
			socketUDP.send(peticion);
```

```
                    // Construimos el DatagramPacket que contendra la respuesta
                    byte[] bufer = new byte[1000];
                    // Construimos el DatagramPacket que contendra la respuesta
                    DatagramPacket respuesta =
                                new DatagramPacket(
                                            bufer,
                                            bufer.length);
                    socketUDP.receive(respuesta);
                    // Enviamos la respuesta del servidor a la salida estandar
                    System.out.println("Respuesta: " +
                    new String(respuesta.getData()));
                    // Cerramos el socket
                    socketUDP.close();

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

Ejemplo de aplicación con Sockets UDP

Vamos a ver un ejemplo de una aplicación cliente-servidor programada con Sockets UDP.

Vamos a crear **dos clases**, una que va a representar al **servidor UDP** y otra que va a representar al **cliente UDP**.

En el **ServidorUDP** vamos a tener lo siguiente:

- ✔ La variable int PUERTO va a representar el puerto donde se conectará nuestro cliente, que será el mismo puerto que tendremos en el servidor.
- ✔ En el constructor de la clase es donde vamos a poner toda la **funcionalidad** de nuestra aplicación cliente. Si nos fijamos tenemos un **bloque try-catch** ya que todo el tratamiento con Sockets, tanto TCP como UDP, puede dar algún fallo y hay que tratarlos.
- ✔ Lo primero que hacemos será crearnos un datagram socket que será un socket de UDP. Lo crearemos con el puerto al que nos conectamos.
- ✔ Crear un array del tipo byte que nos va a permitir tanto el envío como la recepción de datagramas UDP.
- ✔ En un bucle infinito, para poder dar servicio a todos los clientes que se nos conecten, tendremos lo siguiente:
    1. Creamos un objeto de la clase DatagramPacket que nos permitirá realizar las operaciones. A él le vamos a pasar tanto el buffer como la longitud del mismo.
    2. Esperamos a recibir una petición de un cliente.
    3. Una vez recibida, mostraremos toda la información del mismo, tanto como el **mensaje**, el **puerto** de donde se recibe y el **host** de donde se recibe.
    4. Crearemos un mensaje respuesta y lo **enviaremos** mediante un objeto DatagramPacket.
    5. Tenemos un método main que será el encargado de ejecutar nuestro servidor.

```
public class ServidorUDP {

	private static final int PUERTO = 6789;

	public ServidorUDP() {
		try {
			DatagramSocket socketUDP = new DatagramSocket(PUERTO);
			byte[] bufer = new byte[1000];
			System.out.println("Escucho el puerto " + PUERTO);
			/**
			 *  Creo un bucle infinito para atender a todos los
			 *  clientes que se conecten sin límite
```

```
                 */
                while (true) {
                // Construimos el DatagramPacket para recibir peticiones
                      DatagramPacket peticion =
                                  new DatagramPacket(
                                              bufer,
                                              bufer.length);
                      // Leemos una petición del DatagramSocket
                      socketUDP.receive(peticion);
                      // Otengo el mensaje del cliente
                      String mensajerecibido =
                                  new String(peticion.getData());
                      System.out.print("Datagrama recibido del host: " +
                                  peticion.getAddress());
                      System.out.print(" desde el puerto remoto: " +
                                  peticion.getPort());
                      System.out.println(" con el mensaje: " +
                                  mensajerecibido);
                    // Construimos el DatagramPacket para enviar la respuesta
                      String mensajerespuesta = "Hola cliente " +
                      mensajerecibido;
                      DatagramPacket respuesta =
                                  new DatagramPacket(
                                              mensajerespuesta.getBytes(),
                                              mensajerespuesta.length(),
                                              peticion.getAddress(),
                                              peticion.getPort());
                      // Enviamos la respuesta, que es un eco
                      socketUDP.send(respuesta);
                }
          } catch (IOException e) {
                System.out.println("Error: " + e.getMessage());
          }
    }

    public static void main(String[] args) {
          new ServidorUDP();
    }
}
```

En el **ClienteUDP** tenemos lo siguiente:

- ✔ Una variable int que guarda el puerto que va a ser el mismo que el cliente.
- ✔ Una variable String para conectar con el servidor que, como estamos en la misma máquina, va a ser localhost.
- ✔ Y una variable Random, ya que vamos a mandar números aleatorios al servidor.

1. Lo primero que haremos será en el constructor, en un bloque try-catch, crear un **número aleatorio** que será el mensaje que vamos a enviar.
2. Ahora, mediante un objeto de DatagramSocket, vamos a realizar la conexión.
3. Creamos el mensaje en un array de bytes.
4. Mediante el método getBytes de string, obtenemos la **URL del servidor**.
5. Creamos un DatagramPacket con la **petición** que será lo que vamos a enviar. En ella tenemos:
    - el mensaje,
    - la longitud,
    - el host donde está el servidor
    - y el puerto.
6. Lo enviamos mediante el método send.
7. Construimos el DatagramPacket que va a contener la respuesta, para recibirla del servidor mediante el método receive.
8. Y la mostramos por pantalla.
9. Aquí podemos ver que también tenemos un método main que será el encargado de ejecutar el cliente.

```
public class ClienteUDP {

    private static final int PUERTO = 6789;
    private static final String HOST = "localhost";
    private Random aleatorio;

    public ClienteUDP() {
        try {
            /**
             *  Creo el mensaje a enviar al servidor
             *  con un número aleatorio
             */
            aleatorio = new Random();
            int numerocliente = aleatorio.nextInt(100) + 1;
            String mensajeenviar = String.valueOf(
                    numerocliente);
            // Creo el socket UDP
            DatagramSocket socketUDP =
```

```
                new DatagramSocket();
            byte[] mensaje = mensajeenviar.getBytes();
            // Obtengo la dirección del host
            InetAddress hostServidor = InetAddress.getByName(HOST);
            /**
             *  Construimos un datagrama para enviar el mensaje
             *  al servidor
             */
            DatagramPacket peticion
                    = new DatagramPacket(
                                mensaje,
                                mensajeenviar.length(),
                            hostServidor,
                            PUERTO);
            // Enviamos el datagrama
            socketUDP.send(peticion);
            /**
             *  Construimos el DatagramPacket que
             *  contendrá la respuesta
             */
            byte[] bufer = new byte[1000];
            DatagramPacket respuesta = new DatagramPacket(
                    bufer,
                    bufer.length);
            socketUDP.receive(respuesta);
            /**
             *  Enviamos la respuesta del servidor
             *  a la salida estandar
             */
            System.out.println("Respuesta: " +
             new String(respuesta.getData()));
            // Cerramos el socket
            socketUDP.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        new ClienteUDP();
    }
}
```

Para ejecutar esta aplicación:

Primero deberemos ejecutar el servidor y luego los clientes.

Para ello, en el servidor, pulsamos botón derecho y seleccionamos “Run File”.

Y en el cliente, pulsamos botón derecho y seleccionamos “Run File”.

Tenemos que el servidor está escuchando en el puerto 6789.

Escucho el puerto 6789

Ahora podemos ejecutar un cliente de la misma forma. Botón derecho, “Run File”.

Respuesta: Hola cliente 21

En el servidor vemos que se ha recibido un datagram de esta dirección del puerto remoto este con un mensaje que es 21.

Escucho el puerto 6789
Datagrama recibido del host: /127.0.0.1 desde el puerto remoto: 60766 con el mensaje: 21

Y se ha devuelto la respuesta al cliente.

Respuesta: Hola cliente 21

Si volvemos a ejecutar otro cliente, pasará lo mismo. Solo que en este caso es 91.

en la consola del servidor tenemos:
Datagrama recibido del host: /127.0.0.1 desde el puerto remoto: 60767 con el mensaje: 91

Y en la consola del cliente tenemos:
Respuesta: Hola cliente 91

Y así con todos los clientes que queramos.

Usar Sockets TCP o UDP

Lo más recomendable será usar los **Sockets TCP**, ya que ofrecen **garantía de llegada** de los paquetes; además, los **envía y recibe en orden**, por lo que no deben realizarse operaciones de reestructuración de los mismos.

Será **mucho más rápido y seguro TCP** (aunque, por definición, sea más lento, con tan poca información a intercambiar, no va a existir apenas diferencia) que UDP.

La forma de comunicación, es muy simple. Una vez el cliente se conecte con el servidor, deberá enviar sus datos, el servidor lo recibirá y realizará la funcionalidad programada, devolviéndola al cliente mediante su flujo de salida correspondiente:

Pseudocódigo que resuelve el ejercicio práctico

```
// El servidor espera un cliente
cliente = obtenerCliente()
// Una vez obtenido el cliente obtiene sus flujos de comunicación
entrada = obtenerFlujoEntrada()
salida = obtenerFlujoSalida()
// Obtiene el número del DNI
numero = entrada.obtenerMensaje()
// Realizar funcionalidad
resultado = calcularResultado (dato)
// Envía el resultado al cliente
salida.enviarMensaje (dato)
```

Sincronizar dos aplicaciones que usen sockets

La sincronización de aplicaciones se hace, precisamente, **para** llevar a cabo un **cálculo del tiempo que cada aplicación cliente está ocupando** la aplicación servidor.

Una forma muy sencilla de realizar la implementación de la sincronización, tanto del cliente que se conecta como del servidor es través del **envío de mensajes vía red**.

Podemos hacer que el **cliente** envíe una **petición de conexión** al **servidor** y que este, al recibirla, le **envíe** al cliente, de vuelta, la **hora del sistema**, que será la **hora en la que se han conectado** uno al otro y que, por lo tanto, se han **sincronizado**.

Una vez hecho esto, cuando el **cliente decida desconectarse**, el **servidor** podrá volver a obtener la **hora del sistema**, sabiendo así el tiempo que el cliente ha estado conectado a él.

Esto se podría realizar tanto mediante Sockets TCP o UDP.

Ejemplo sencillo de cómo podría llevarse a cabo la **sincronización entre cliente y servidor** mediante **pseudocódigo**:

```
// Cuando el cliente se conecte al servidor
        hora = cliente.obtenerHoraConexion ()
// Una vez obtenida la hora de conexión del cliente
// el servidor comprueba que la hora es correcta
        horaser = obtenerHoraSistema ()
        diferencia = horaser - hora
// Dejamos un pequeño margen por la actuación del servidor
        si diferencia < 0.05
// sincronizacion realizada
        sino
// enviar error al usuario
```

Eficiencia en las aplicaciones que usan sockects

Para comprobar si una aplicación es más eficiente que otra se pueden tener en cuenta varios factores, y los más importantes serían:

- Tiempo de respuesta de cada una de ellas: Ya que unas milésimas de diferencia pueden significar mucho tiempo cuando hay muchas peticiones. Para ello, habría que medir el tiempo que tarda cada una de ellas en dar el resultado esperado.
- Tráfico de red que genera cada una, siendo la **mejor** opción la que **menor tráfico** genere, ya que saturará menos la red.
- Espacio de memoria que ocupen durante su ejecución. Habría que analizar qué RAM necesita cada una durante su funcionamiento.
- Recursos de CPU requeridos: A **menor uso de CPU** durante la ejecución, **mejor rendimiento** presentarán.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red
Clientes HTTP, FTP, Telnet y SMTP
Sockets con hilos

Programación de un cliente HTTP

La implementación del protocolo HTTP o HTTPS se basa en un proceso sencillo que implica una serie de **solicitudes y respuestas** a esas solicitudes por parte tanto del **cliente**, como del **servidor**:

- En primer lugar, un **cliente** establecerá una conexión con un servidor, enviando para ello un **mensaje de solicitud** con los datos pertinentes.
- Cuando el **servidor** reciba dicho mensaje, le **responderá con un mensaje** muy similar, conteniendo éste el resultado de la operación solicitada por el cliente.

El lenguaje de programación **Java** dispone de dos clases que nos van a permitir programar aplicaciones donde tengamos **tanto servidores como clientes HTTP**. Estas clases son:

- La clase URL: Esta clase nos va a permitir **representar una dirección** de una página web, por ejemplo, https://www.google.es, de forma que el programa pueda realizar operaciones con ella.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

  https://docs.oracle.com/javase/7/docs/api/java/net/URL.html.

- La clase URLConnection: Esta clase es la que nos va a permitir **realizar operaciones con la dirección** web que hemos creado mediante URL.

  Podremos lanzar **operaciones tipo GET** y obtener las respuestas de éstas de una forma muy sencilla.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

  https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html.

Al utilizar estas clases estamos programando un **servicio HTTP de alto nivel**, esto quiere decir que todas las operaciones que harán posible la comunicación no se visualizarán, y para trabajar con ello, será tan sencillo como **programar un servidor o cliente** mediante el uso de **sockets**.

Procesamiento de peticiones HTTP

Los servidores HTTP deben procesar de alguna forma las peticiones de sus clientes. Este tipo de servidores van a dar un servicio **según el método que utilicen los clientes HTTP** al invocarlos.

Lo habitual es tener un **método llamado GET**, en el que según un **parámetro**, se pueda **indicar qué servicio** de los ofrecidos por el servidor **queremos**.

Programación de un cliente FTP

Las siglas del protocolo FTP significan File Transfer Protocol, o Protocolo de Transferencia de Ficheros, y es el protocolo que debemos usar siempre que deseemos **transferir ficheros entre un servidor y un cliente** o viceversa.

Los **pasos para crear un cliente FTP** son los siguientes:

- Realizar una **conexión** al servidor.
- **Comprobar** que la **conexión** que se ha realizado con éxito.
- **Validar el usuario** FTP que se ha conectado. En caso de que el usuario no sea válido deberemos abortar la conexión y enviar un **mensaje de error**.
- Realizar las **operaciones** pertinentes **con el servidor**.
- **Desconectar del servidor** una vez terminemos de realizar las operaciones requeridas.

No hay que olvidar que todo el proceso de conexiones y realización de operaciones puede generar excepciones, concretamente puede lanzar las **excepciones:**

- **SocketException,**
- **IOException**.

El lenguaje de programación **Java no dispone de clases específicas para el uso del protocolo FTP**, pero la fundación Apache creó la API:

org.apache.commons.**net.ftp**

para **trabajar con clientes y servidores FTP**.

Esta API tiene las siguientes clases que nos permiten operar de una manera sencilla con el protocolo FTP:

- Clase FTP:

  Esta es la clase que nos va a proporcionar todas las **funcionalidades básicas** para poder realizar un cliente FTP básico. Esta clase **hereda de SocketClient**.

- Clase FTPReplay:

  Esta es la clase que nos va a permitir **operar con los valores devueltos** por las consultas FTP **del servidor**.

- Clase FTPClient:

  Esta clase es la encargada de dar **soporte a las funcionalidades del cliente** FTP. **Hereda de SocketClient**.

- Clase FTPClientConfig:

Esta clase nos va a permitir realizar las **configuraciones** oportunas de los clientes FTP de una forma sencilla.

• Clase FTPSClient: Esta clase nos va a permitir utilizar el **protocolo FTPS**, que es la versión segura del protocolo FTP. Esta clase utiliza el **protocolo SSL** y **hereda de FTPClient**.

Comprobaciones de seguridad en red

La seguridad en los servidores es algo totalmente necesario, ya que puede exponerse código o documentos confidenciales. Con cualquier tipo de servicio deberíamos de tener unas mínimas comprobaciones de seguridad.

Para una mínima seguridad en el sistema podríamos realizar las siguientes **comprobaciones una vez nos llegue una petición de un cliente**:

• Comprobar que el cliente es un **cliente válido** del sistema. Para esto deberemos mantener el **listado de clientes** que tenga el sistema en una pequeña base de datos o, en su defecto, en un fichero.

• Comprobar que la operación que nos pide el cliente es una **operación válida** en el sistema. Para esto basta con saber qué operaciones hemos diseñado en nuestro sistema y cuáles son sus parámetros.

• Comprobar que el cliente tiene **permisos** para realizar dicha operación. De igual forma que deberemos tener un **registro de clientes válidos** en el sistema, le podemos agregar una serie de datos indicando qué **operaciones pueden realizar**.

Pseudocódigo Pasos a seguir: comprobaciones en petición de cliente

```
// Comprobamos que el cliente es valido
c_valido = comprobarCliente (nombre)
// Comprobamos que la operación solicitada
// es valida
op_valida = comprobarOperacion (operación)
// Comprobamos que el cliente tiene permiso para
// realizar dicha operación
permisos = comprobarPermisos(nombre, operación)
si c_valido = verdad Y op_valida = verdad Y permisos = verdad
        Realizamos la operación solicitada
sino
        Devolvemos un mensaje de error
```

Programación de un cliente Telnet

El protocolo Telnet, cuyo nombre significa TELecomunication NETwork, nos va a permitir acceder a otros equipos conectados a nuestra red, pudiendo realizar así una **administración de forma remota**, como si estuviéramos sentados frente al equipo que estamos administrando remotamente.

Gracias a este protocolo, se hace muy simple la administración de equipos que no tengan pantalla ni teclado, es decir, que sean simplemente un servidor que se arranca y lanza todas sus tareas y servicios automáticamente.

Bases del protocolo Telnet:

- Tiene el **esquema** básico del **protocolo cliente/servidor**.
- El **puerto** que utiliza es el **23**.
- Funciona mediante **comandos** en modo texto.

Aunque sea un protocolo que nos ayudará y simplificará la administración de equipos, **no ofrece mucha seguridad**, y es por esta misma razón por la que apenas se utiliza en las grandes empresas. Esto se debe a que toda la **información se transmite en texto plano**, incluidos los datos y contraseñas de los usuarios, cuando toda la información debería intercambiarse cifrada.

La biblioteca que Apache nos ofrece en su paquete org.apache.commons.net.telnet para poder trabajar con este protocolo es:

Clase **TelnetClient**: Esta es la clase que nos va a permitir **implementar un terminal** para usar el protocolo Telnet. Esta clase **hereda de la clase SocketClient**.

Entre los métodos más útiles de esta clase tenemos:

- SocketClient.connect(), que nos permitirá realizar una **conexión** al servidor.
- TelnetClient.getInputStream() y TelnetClient.getOutputStream(), que nos permitirán obtener los **flujos de comunicación**.
- TelnetClient.disconnect(), que nos permitirá desconectar.

Programación de un cliente SMTP

El protocolo SMTP es el que se utiliza para **enviar y recibir correos electrónicos**.

Para poder implementar un cliente SMTP vamos a utilizar la API javax.mail que nos proporciona el lenguaje de programación Java.

Dentro de esta API tenemos las siguientes **clases** que nos van a permitir realizar una gestión de correos electrónicos de una manera fácil y sencilla:

- Session:

  Esta clase representa una **sesión de usuario** para correo electrónico. Aquí vamos a tener agrupada **toda la configuración por defecto** que utiliza la API javax.mail para la gestión de correos electrónicos. Mediante el método getDefaultInstance() podremos obtener **una sesión por defecto**, con toda su configuración correspondiente.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/jms/Session.html.

  https://docs.oracle.com/javaee/6/api/javax/mail/Session.html.

- Message:

  Esta clase representa un **mensaje de correo electrónico**. Podremos configurar:

  - **hacia quién va dirigido** mediante el método setFrom(), (en realidad es quién lo envía, es un error del temario...).
  - el **asunto** del correo electrónico, mediante el método setSubject(),
  - el **texto** del mismo, mediante el método setText().

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://javaee.github.io/javamail/docs/api/javax/mail/Message.html.

- Transport:

  Esta clase es la que representa el envío de los correos electrónicos. **Hereda de la clase Service**, que es una clase que proporciona las **funcionalidades comunes** a todos los servicios de **mensajería**.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/mail/Transport.html.

Mediante las clases anteriores podemos crear fácilmente una **aplicación que envíe correos electrónicos** mediante nuestra cuenta de Gmail, por ejemplo.

Sockets e hilos I

Lo ideal es que cada servidor pueda **atender a muchos clientes** al mismo tiempo, haciendo así una **implementación concurrente de los servicios**.

La concurrencia la conseguimos utilizando hebras o hilos, haciendo así que se lance una hebra por cada tarea concurrente que queramos realizar, dando la sensación de que se estaban ejecutando todas al mismo tiempo.

A la hora de programar servicios con sockets también vamos a hacer uso de las hebras, haciendo que **cada servidor pueda atender a varios clientes a la vez**.

Cuando el servidor detecte que un cliente le ha hecho una petición éste deberá aceptarla, procesarla y crear una hebra que sea capaz de atender la petición del cliente.

De esta forma, cada cliente será atendido por una hebra diferente y el servidor podrá volver a escuchar una petición nueva de otro cliente, inmediatamente después de lanzar la hebra que atenderá al primer cliente.

De esto debemos deducir que las **hebras se ejecutarán en la parte del servidor**, ya que es este el que atiende la petición del cliente, de forma que el cliente no será consciente de si lo está atendiendo el propio servidor o una hebra en el mismo.

Boceto de utilización de hilos con sockets

```
// Crear socket
while (Boolean.TRUE) {

      /**
       * 1. Aceptar una solicitud de cliente
       * 2. Código
       * 3. Crear una hebra independiente en el servidor
       *    para procesar la solicitud
       */
}
```

Ejemplo de Monitorización de tiempos de respuesta

Para monitorizar el tiempo de respuesta del servidor:
Se debe tener en cuenta el tiempo de proceso del servidor **y el tiempo de transmisión**.

```
public class Prueba {

	/**
	 * Esta función calcula la diferencia en segundos entre dos fechas
	 *
	 * @param tiempoinicio Tiempo de inicio
	 * @param tiempofin    Tiempo final
	 * @return Diferencia en segundos entre tiemponicio y tiempofinal
	 */
	private static float diferenciaSegundosTiempo(
			Date tiempoinicio, Date tiempofin) {
		// Calculamos la diferencia de las fechas en segundos
		float segundos =
				(float) ((tiempofin.getTime() / 1000)
						- (tiempoinicio.getTime() / 1000));

		if (segundos > 60) {
			segundos = segundos / 60;
		}

		return segundos;
	}

	/**
	 * @param args the command line arguments
	 */
	public static void main(String[] args) {
		try {
			// Obtenemos la fecha inicial
			Date inicio = Calendar.getInstance().getTime();

			// Dormimos el programa 2 segundos
			Thread.sleep(2000);

			// Obtenemos la fecha final
			Date fin = Calendar.getInstance().getTime();

			// Mostramos la diferencia en segundos entre las dos fechas
			System.out.println("La diferencia en segundos es: "
```

```
                + diferenciaSegundosTiempo(inicio, fin));
        } catch (InterruptedException ex) {
                Logger.getLogger(
                        Prueba.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Vamos a ver cómo podemos monitorizar el tiempo de respuesta de una serie de procesos u operaciones a los que queramos medir lo que tardan en ejecutarse:

- En primer lugar, deberemos realizar las siguientes operaciones que son tres:
  1. La primera será obtener la fecha antes de ejecutar las tareas que queremos monitorizar en tiempo. Para ello, podemos utilizar la clase Date que mediante la clase Calendar y el método getInstance().getTime() nos permitirá obtener el momento de tiempo en el que se ejecuta.

```
// Obtenemos la fecha inicial
Date inicio = Calendar.getInstance().getTime();
```

  2. Una vez hecho esto, ejecutaremos todas las instrucciones u procesos que se necesitan para obtener la fecha antes de ejecutar las tareas que queremos monitorizar en tiempo. Una vez hecho esto, ejecutaremos todas las instrucciones u procesos que queramos monitorizar. En este caso, al ser un ejemplo sencillo, lo único que vamos a hacer es hacer que el **programa se duerma durante dos segundos** mediante el método Thread.sleep().

```
// Dormimos el programa 2 segundos
Thread.sleep(2000);
```

  3. Una vez que terminemos de ejecutar todas las operaciones u procesos que queremos monitorizar, deberemos volver a obtener el instante de tiempo que va a ser diferente al instante de tiempo en el que cogimos el inicio. Una vez hecho esto, ya tenemos el instante de tiempo de inicio y el instante de tiempo de fin.

```
// Obtenemos la fecha final
Date fin = Calendar.getInstance().getTime();
```

Hemos creado una función que se llama diferenciaSegundosTiempo que tiene como parámetros los dos tiempos, el de inicio y el del fin, y que nos va a devolver un float con los segundos de diferencia.

```
private static float diferenciaSegundosTiempo(
                  Date tiempoinicio, Date tiempofin) {
            // Calculamos la diferencia de las fechas en segundos
            float segundos =
                        (float) ((tiempofin.getTime() / 1000)
                                    - (tiempoinicio.getTime() / 1000));

            if (segundos > 60) {
                  segundos = segundos / 60;
            }

            return segundos;
      }
```

Teniendo nuestra función diferenciaSegundosTiempo, lo que haremos será después de obtener el tiempo de fin del proceso, obtener la diferencia en segundos y mostrarla por pantalla.

```
            // Mostramos la diferencia en segundos entre las dos fechas
            System.out.println("La diferencia en segundos es: "
            + diferenciaSegundosTiempo(inicio, fin));
```

Vamos a ejecutar este programa para ver cómo funciona.

```
La diferencia en segundos es: 2.0
```

Como podemos ver, la diferencia en segundo es de 2.0 ya que el programa duerme dos segundos.

También podríamos dar esta diferencia en milisegundos.

Ejemplo de servicio concurrente

Vamos a ver la forma de realizar un **servidor TCP** que una vez que acepte la solicitud con un número de DNI de un cliente, lance una **hebra que atienda la solicitud**.

La hebra que lancemos **se ejecutará en el servidor**, y gracias a esto podremos pasar los datos que necesitemos del cliente a la misma, ya que el servidor en sí los tiene.

Para resolver este problema deberemos:

1. crear **una clase que sea una hebra**, pudiendo hacerlo de los dos métodos posibles, heredando de Thread o implementando Runnable.
2. A esta clase le tendremos que implementar un **método** que proporcione el servicio deseado, siendo en este caso una función que **calcule y devuelva la letra del DNI**.
3. Dentro de los parámetros de la clase le vamos a pasar:
    1. el número del DNI del cliente,
    2. el flujo de salida del cliente,

    para que de esta forma podamos enviar la letra al cliente una vez calculada.

Si hemos seguido estos pasos podremos atender a tantos clientes como necesitemos de forma concurrente, aunque siempre **es muy recomendable** establecer un **tope máximo de clientes**, dependiendo esto del servicio que queramos dar y de las prestaciones de nuestro servidor.

Pseudocódigo del servidor concurrente

```
// Aceptamos el cliente
cliente = esperarCliente()
// Obtenemos el flujo de entrada y el número
entrada = cliente.obtenerFlujoEntrada()
numerodni = entrada.obtenerMensaje()
// Obtenemos el flujo de salida del cliente
salida = cliente.obtenerFlujoSalida()
// Creamos el objeto de la hebra y la lanzamos
hebraservidor = Hebra(numerodni, salida)
hebraservidor.lanzar()
```

Sockets e hilos

Vamos a ver cómo podemos crear un programa **cliente/servidor concurrente** con **TCP** mediante hilos.

Para esto deberemos crear las siguientes **clases**:

• Clase Servidor: Esta clase será la que va a **representar el servidor** de nuestra aplicación.

• Clase ServidorHebra: Esta clase será la que va a representar una **hebra que se lanzará en el servidor** y que será la encargada realmente de **dar servicio al cliente** conectado, haciendo posible la **concurrencia**.

• Clase Cliente: Esta clase será la que va a **representar un cliente** de nuestra aplicación. Se podrán ejecutar **tantos clientes como se quiera**.

➔ Dentro de la clase servidor deberemos crear un **método main**, en el que:

- crearemos un **ServerSocket** con el que esperaremos que **se conecte un cliente**.
- Una vez **conectado el cliente** a nuestro servidor, deberemos crear un **objeto de la clase ServidorHebra**, al que le pasaremos como mínimo, los **flujos de entrada y salida** del cliente, para poder así poder establecer una **comunicación con él**. Después de crear y lanzar la hebra, nuestro servidor **volverá a escuchar** para que se conecte otro cliente al que dar servicio.

➔ Dentro de la clase cliente deberemos crear un **método main**, en el que:

- creamos un **Socket** que conectaremos al servidor.
- Una vez hecho, nos **comunicamos con el servidor normalmente**.

➔ Dentro de la clase ServidorHebra deberemos implementar el **método run** y el método que ejecutará la hebra, implementando en dicho método toda la **funcionalidad que debe dar el servidor normalmente**, es decir, realizando una **comunicación normal con el cliente**, tanto para **recibir** como para **enviarle** información.

- Será dentro de la clase ServidorHebra donde implementemos **todas las funciones** que necesitemos para poder dar los **servicios** requeridos a los clientes.

Ejemplo de sockets con hilos paso a paso

Vamos a ver cómo podemos crear una **aplicación cliente-servidor** utilizando **Sockets TCP** pero **concurrente**, es decir, que un servidor va a poder dar servicio a más de un cliente simultáneamente. Para ello necesitamos crear mínimo tres clases:

- la clase Cliente,
- la clase Servidor,
- la clase ServidorHebra.

La **clase cliente** será la que va a representar un cliente de nuestra aplicación y tendrá un **método main** para poder ejecutarse que hará lo siguiente:

1. Lo primero será configurar nuestro cliente. Para ello, nos declaramos:
   a) la **dirección IP** que será localhost ya que el ejercicio se ejecutará en nuestra propia máquina,
   b) el **puerto** al que nos vamos a conectar.
   c) Obtenemos la **IP real**,
   d) creamos un **socket** mediante la IP y el puerto.

   Aquí nuestro cliente se conectará a nuestro servidor y daremos el mensaje de que la conexión se ha establecido.

```
String direccionIP = "localhost";
int puerto = 5056;
// Optengo la IP real
InetAddress ip = InetAddress.getByName(direccionIP);
System.out.println("CLIENTE: Conectando con "
+ direccionIP + ":" + puerto + "...");
// Establezco la conexión con la IP y el puerto
Socket socket = new Socket(ip, puerto);
System.out.println("CLIENTE: Conexión establecida.");
```

2. Ahora obtenemos los flujos tanto de **entrada** como de **salida** para poder realizar una conversación con nuestro servidor:

```
// Obtengo los flujos de entrada y salida
DataInputStream entrada =
new DataInputStream(socket.getInputStream());
DataOutputStream salida =
new DataOutputStream(socket.getOutputStream());
```

y realizamos un bucle que nos va a permitir realizar dos tareas:

a) Si escribimos un 1, el servidor nos enviará un número aleatorio,

b) si escribimos un 2, saldremos.

```
// Este es el bucle que va a permitir la comunicación entre el
// cliente y el cliente hebra
boolean salir = false;
Scanner teclado_String = new Scanner(System.in);
while (!salir)
{
    // Imprimo el mensaje del cliente
    System.out.println(entrada.readUTF());
    // Leo la respuesta y la envío
    // (Si quiero número aleatorio o salir)
    String textoenviar = teclado_String.nextLine();
    salida.writeUTF(textoenviar);

    // Según lo que le haya enviado al servidor...
    switch(textoenviar)
    {
        case "1": // Opción mostrar número aleatorio
            // Imprimo el mensaje del número aleatorio
            // del cliente
            String mensajerecibido = entrada.readUTF();
            System.out.println(mensajerecibido);
            break;
        case "2": // Opción salir
            System.out.println(""
                    + "CLIENTE: Cerrando la conexión...");
            socket.close();
            System.out.println(""
                    + "CLIENTE: Conexión cerrada.");
            salir = true;
            break;
        default:
            System.out.println(""
                    + "CLIENTE: Opción incorrecta.");
            break;
    }
}
```

Antes de seguir con la clase cliente, vamos a ver la **clase servidor** que será la que represente un servidor de nuestra aplicación:

En ella tendremos un **método main** que hará lo siguiente:

Nos creamos un ServerSocket para crear nuestro servidor en el mismo puerto al que se conectan los clientes.

```
ServerSocket socketservidor = new ServerSocket(5056);
System.out.println(""
+ "SERVIDOR: Escuchando en localhost:5056...");
```

Con un bucle infinito realizamos la funcionalidad de nuestro servidor que será la siguiente:

Esperaremos que se nos conecte un cliente y una vez conectado y aceptado obtendremos sus flujos de entrada y salida para poder establecer una conversación con él.

```
// Espero a que se conecte un nuevo cliente
socketcliente = socketservidor.accept();
System.out.println(""
        + "SERVIDOR: Cliente nuevo conectado: "
        + socketcliente);
// Obtengo los flujos de entrada y salida del socket cliente
DataInputStream entrada =
        new DataInputStream(
                socketcliente.getInputStream());
DataOutputStream salida =
        new DataOutputStream(
                socketcliente.getOutputStream());
```

Una vez hecho esto, nos deberemos crear una hebra que atenderá a ese cliente y lanzarla. Para ello tenemos la clase ServidorHebra.

Aquí podemos ver que nos hemos creado un **objeto ClienteHebra del tipo thread** que es del tipo servidor hebra al que le pasamos el nombre de nuestro servidor. Y una vez lanzado nuestro servidor, se despreocupa y vuelve a escuchar el siguiente cliente, siendo la hebra quien atiende al cliente.

```
System.out.println(""
        + "SERVIDOR: Creando una hebra nueva "
        + "para el cliente nuevo...");
```

```
            // Creo una hebra para el cliente que se ha conectado
            Thread clientehebra = new ServidorHebra(
                      socketcliente, entrada, salida);
              // Lanzo la hebra del cliente
              clientehebra.start();
              System.out.println(""
                      + "SERVIDOR: Hebra del cliente nuevo "
                      + "creada (yo ya me despreocupo).");
```

La clase **servidor hebra** hereda de Thread y tiene como datos el socket del cliente y los flujos de entrada y salida. Aquí podemos ver el **constructor** de nuestra clase:

```
public class ServidorHebra extends Thread {

private DataInputStream entrada;
private DataOutputStream salida;
private Socket socketcliente;

public ServidorHebra(
        Socket socketcliente,
        DataInputStream entrada,
        DataOutputStream salida) {
    this.socketcliente = socketcliente;
    this.entrada = entrada;
    this.salida = salida;
}
```

Y el **método run** que será el que haga la funcionalidad.

Este método será el que **hable verdaderamente con el cliente** y hace lo siguiente:

Lo primero que hará será mandarle un mensaje a nuestro cliente preguntado qué quiere hacer: si generar un número aleatorio o salir. En este momento se ejecutará esta orden de la clase Cliente:

```
salida.writeUTF(textoenviar);
```

y se mostrará ese mensaje pidiendo al cliente una opción: si uno o dos y enviándola a el servidor. En este caso realmente se envía a la hebra.

Una vez que ha llegado, se ejecuta esta instrucción de la clase ServidorHebra:

```
// Recibo la respuesta del cliente
mensajerecibido = entrada.readUTF();
```

y se hará una distinción, en el caso de que sea uno o de que sea dos.

En el caso de que la instrucción **sea uno**, generamos un número aleatorio y lo enviamos a el cliente.

```
// Según el mensaje recibido...
switch (mensajerecibido) {
    case "1":
        int aleatorio = generador.nextInt(500);
        salida.writeUTF(""
              + "SERVIDOR: El número aleatorio generado es "
              + aleatorio);
        break;
    case "2":
        System.out.println(""
                  + "SERVIDOR: El cliente "
                  + this.socketcliente + " envía salir...");
        System.out.println(""
                  + "SERVIDOR: Cerrando la conexión...");
        this.socketcliente.close();
        System.out.println(""
                  + "SERVIDOR: Conexión cerrada.");
        salir = false;
        break;
      }
```

En este caso, se ejecutará este trozo de aquí, de la clase Cliente, recibiendo el mensaje e imprimiéndolo por pantalla y volviéndonos a preguntar qué queremos hacer.

```
case "1": // Opción mostrar número aleatorio
          // Imprimo el mensaje del número aleatorio
          // del cliente
          String mensajerecibido = entrada.readUTF();
          System.out.println(mensajerecibido);
          break;
```

En el caso de que introduzcamos **un dos**, se enviará ese dos a nuestro servidor aquí (en la clase ServidorHebra):

```
// Recibo la respuesta del cliente
mensajerecibido = entrada.readUTF();
```

y se ejecutará el trozo correspondiente a salir, que lo que hace es cerrar todos los flujos (como vemos en el código de más arriba en case 2...).

En el **cliente**, se ejecutará la opción salir y también se cerrarán todos los flujos.

```
case "2": // Opción salir
    System.out.println(""
            + "CLIENTE: Cerrando la conexión...");
    socket.close();
    System.out.println(""
            + "CLIENTE: Conexión cerrada.");
    salir = true;
    break;
```

Vamos a ver cómo funciona nuestra aplicación:

En primer lugar, deberemos ejecutar el servidor:

```
SERVIDOR: Escuchando en localhost:5056…
```

El cual nos dice que está escuchando en el puerto 5056.

Ahora deberemos ejecutar un cliente.

```
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=52364,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
```

Aquí en el servidor se nos indica de que se ha creado un cliente nuevo y se ha lanzado la hebra.

Si volvemos a ejecutar otro cliente:

```
CLIENTE: Conectando con localhost:5056...
CLIENTE: Conexión establecida.
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
```

los dos clientes se están ejecutando perfectamente y en el servidor se indica que se han lanzado dos hebras.

```
SERVIDOR: Escuchando en localhost:5056...
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=52364,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=47260,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
```

Si en el cliente 1 le damos la orden de generar un número aleatorio:

```
CLIENTE: Conexión establecida.
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
1
SERVIDOR: El número aleatorio generado es 466
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
```

nos dirá que el número es 377 y volverá a preguntar que queremos hacer.

Si le decimos que queremos salir:

```
1
SERVIDOR: El cliente
Socket[addr=/127.0.0.1,port=52364,localport=5056] envía salir...
SERVIDOR: Cerrando la conexión...
SERVIDOR: Conexión cerrada.
```

se cerrará nuestro cliente. El servidor nos dirá que se ha cerrado un cliente.

Pero el cliente lo seguirá ejecutando hasta que introduzcamos la instrucción de salir, que cerrará el cliente, se destruirá la hebra,

```
SERVIDOR: El cliente Socket[addr=/127.0.0.1,port=47260,localport=5056] envía salir...
```

```
SERVIDOR: Cerrando la conexión...
SERVIDOR: Conexión cerrada.
```

pero el servidor sigue funcionando a la espera de más clientes.

https://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

El servicio web
Arquitectura SOA
Proyectos que usan servicios web

SOAP es un protocolo de comunicación basado en XML que permite que aplicaciones en diferentes plataformas se comuniquen entre sí de manera independiente del lenguaje de programación y el sistema operativo utilizado.

Servicios web

Un servicio web, o web service, es un “programa” que proporciona una forma de **comunicación** entre **aplicaciones** software que se están **ejecutando en distintas plataformas**.

Lo forman componentes de aplicaciones distribuidas que deben estar disponibles de forma externa, a los que se podrá acceder mediante una serie de protocolos web estándar, utilizando **lenguajes de programación independientes de la plataforma** en la que se ejecuten para el **intercambio de mensajes**.

Características de los servicios web:

- Se debe poder acceder a ellos a través de la web**.** Para esto se debe utilizar:
  - el **protocolo HTTP**,
  - y el lenguaje **XML**.
- Al tratarse de un estándar de representación de datos independiente de plataformas, el hecho de utilizar **XML** como **formato de intercambio** de datos entre el servicio Web y el cliente **permite la comunicación** entre ambos.
- Todos los servicios web realizan una serie de funciones bien definidas:

  Deben de contener una **descripción de ellos mismos**, de forma que una **aplicación conozca fácilmente** cuál es la función del servicio web.
- Han de poder localizarse:

  Debe existir algún **mecanismo que permita encontrar un servicio** web que realice una determinada función.
- Son componentes independientes que se pueden **integrar**, formando así **sistemas distribuidos complejos**.
- Ofrecen ínter-operabilidad:

  El hecho de usar HTTP y XML hace que sea posible que **distintas aplicaciones puedan utilizar** los servicios web.

Estas son las características más importantes que podemos destacar de los servicios web, pero te invitamos a que investigues por la red para descubrir algunas más.

Arquitectura de los Servicios Web SOAP

Estándares y especificaciones web

A continuación, vamos a conocer cuáles son algunas de las...

Organizaciones que definen los estándares que han de seguir los servicios web.

- W3C: (Consorcio para la World Wide Web):

  Esta organización fue fundada en **octubre de 1994** con el objetivo de llevar a Internet a su **máximo potencial**. Para ello desarrollaron **protocolos** de uso común que han marcado la evolución y han asegurado la **interoperabilidad** de internet. Algunos de estos protocolos son:

  - **HTML**,
  - **HTTP**,
  - **XML**,
  - **SOAP**,
  - **WS**,
  - entre otros.

  Para saber más puedes visitar su página web:

  https://www.w3.org/standards/webofservices/description.html

- OASIS:

  Esta organización es un **consorcio sin fines de lucro** que se dedica a impulsar el **desarrollo, la convergencia y la adopción de los estándares** abiertos para la sociedad de la información global. OASIS siempre ha **promovido el consenso** de la industria y ha producido:

  - una serie de **normas internacionales para la arquitectura orientada a servicios** (SOA, Service Oriented Architecture),
  - las **funciones y calidad** de servicios web,
  - la **seguridad** de la web,
  - el **cloud computing**,
  - la **publicación electrónica** de documentos, etc.

  Los estándares OASIS abiertos han permitido:

  - **reducir costes**,
  - fomentar la **innovación**,
  - y proteger el derecho de **libre elección** de la tecnología,
  - entre otras muchas acciones.

- JPC:

  Esta organización trabaja con **cada versión de JAVA**, incluyendo un conjunto de **especificaciones de diferentes tecnologías**, definidas por este mismo organismo, y que son nombradas como **JSR** (Java Specification Request, petición de especificación de Java), seguidas de un número.

  Especificación es información proporcionada por el fabricante de un producto, la cual describe sus componentes, características y funcionamiento.

  Algunos ejemplos de los **principales estándares que se utilizan en los servicios web** son:

  - JSR 370: define la API JAX-RS 2.1.
  - JSR 224: define la API JAX-WS 2. 0 para servicios web basados en XML.
  - JSR 342: define las especificaciones para java EE 7.
  - JSR 366: define las especificaciones para java EE 8.

Servicios REST

REST es una alternativa **más simple a SOAP**. Varios grandes proveedores de Web 2.0 usan esta tecnología, incluyendo:

- Yahoo,
- Google,
- Amazon,
- Facebook...

Se utiliza frecuentemente en **aplicaciones móviles**.

Las ventajas de REST son las siguientes:

- tienen mejores tiempos de respuesta,
- disminución de sobrecarga tanto en cliente como en servidor,
- mayor estabilidad frente a futuros cambios,
- gran sencillez en el desarrollo de clientes, éstos sólo han de ser capaces de realizar **interacciones HTTP** y **codificar información en XML**.

Ejemplo de proyecto con servicio web: Conversor de medidas

Este proyecto onsiste en convertir una cierta cantidad de centímetros a pulgadas.

Para ello habrá que realizar un **servicio SOAP** donde el cliente podrá indicar la cantidad de centímetros a convertir a pulgadas. También se deberán realizar todas las **comprobaciones** oportunas para garantizar que el servicio SOAP no falle.

Nos centraremos en la fase de análisis del problema, que nos servirá para saber cómo enfocar posteriormente el desarrollo del servicio SOAP.

La conversión de centímetros a pulgadas puede realizarse con una fórmula muy simple, ya que basta con conocer cuántas pulgadas equivalen a un centímetro: concretamente 1 centímetro es igual a 0,393701 pulgadas.

Con respecto a las **comprobaciones** previas al ejercicio:

- podemos cerciorarnos que el **dato** que haya enviado el usuario **no sea** un campo **vacío**, ya que esto podría ocasionar una **excepción en el servidor**, con el respectivo error.
- Otro de los aspectos que podríamos comprobar es que el dato que envíe el cliente para convertir **sean números**, ya que si envía una letra o palabra se producirá una **excepción en el cálculo**.
- Por último, podríamos comprobar que la **cantidad** enviada **no fuese negativa**, que, aunque no es tan importante como las comprobaciones anteriores, no tiene mucho sentido una distancia negativa.

Ejemplo de servico web: Calculadora SOAP

Consiste en realizar un servicio web SOAP que permita **ejecutar la funcionalidad de una calculadora en el servidor**, pudiendo realizar las operaciones de suma, resta, multiplicación y división. El **cliente podrá elegir qué operación realizar**, **pasándole además los datos** con los que operar.

Siempre que tengamos que pensar cómo vamos a resolver un problema en programación, debemos ser conscientes de que existen muchas formas de resolverlo, y este no es una excepción.

Como tenemos que poder elegir **la opción en la parte del cliente**, es decir, en la parte donde vamos a interactuar, ya que ésta está programada mediante **HTML**, podremos crear, por ejemplo, una **lista con las posibles opciones** de la calculadora. Otra forma de resolver el problema podría ser mediante unos **botones de tipo radio**, en los que podamos elegir únicamente una opción.

Con respecto a la **implementación SOAP** tenemos que tener en cuenta que hemos de distinguir claramente qué operación queremos realizar, la cual, habría sido elegida en la parte descrita anteriormente, pudiéndolo hacer fácilmente con un **switch**.

Observa en el siguiente código cómo se ha implementado para la operación, siendo la forma de resolverlo exactamente igual para el resto de operaciones.

Método para el servicio de la ‘suma’

```
@WebMethod(operationName = "operacion")
public Double calcularOperacion (
            @WebParam(name = "operacion") String operacion,
            @WebParam(name = "dato1") Double dato1,
            @WebParam(name "dato2") Double dato2) {
      Double resultado = 0;
      Switch (operacion)
      {
            case "suma":
                  resultado = dato1 + dato2;
                  break;
            // Demas casos de igual forma }
      }
      return resultado;
}
```

Arquitectura orientada a servicios: SOA

La arquitectura SOA es una tecnología que nos permite el diseño de aplicaciones basadas en **peticiones a un servicio**. De esta forma podemos **crear pequeños elementos software reutilizables** y, además, **independientes del lenguaje** con el que fueron creados.

Esto ha servido para dar lugar a un nuevo **tipo de programación**, la llamada **Software as a Service** (software como servicio o **SaaS**). Este tipo de programación se basa en que las aplicaciones no se diseñan para ser instaladas en el ordenador del cliente (como ocurre en la programación de aplicaciones clásica), sino que **se instalan en un servidor** al que los **clientes** realizan una serie de **peticiones**. De este modo, tenemos un **servicio que está disponible** desde cualquier punto del planeta, siempre que se pueda tener **acceso a Internet**.

Los **servicios web** son el punto fuerte de las aplicaciones SOA, debido a que la **tecnología SOA** trabaja con un conjunto de **características** que hacen posible que se **ejecuten los servicios web a la perfección**.

Existen **dos tipos de arquitecturas orientadas a servicios**:

1. Arquitectura Orientada a Servicios Tradicional (SOA tradicional):

   Este tipo de arquitectura utiliza los **principios y las tecnologías básicas** de los servicios web, pudiendo utilizar:

   - **SOAP** como lenguaje de intercambio de datos,
   - **WSDL** (Web Services Description Language) como **lenguaje para la descripción de los servicios** utilizados,
   - y **UDDI** para la publicación. La especificación UDDI (Universal Description, Discovery, and Integration) define un **modo de publicar y encontrar información** sobre servicios Web.

   Estas arquitecturas son muy utilizadas, pero **no incluyen características** tan básicas como pueden ser:

   - **seguridad**,
   - **transaccionalidad**,
   - garantía de **entrega**,
   - **direccionamiento**,
   - entre otras.

2. Arquitectura Orientada a Servicios de segunda generación:

   Un servido **SOA de segunda generación** consigue ampliar la funcionalidad de los anteriores. Para ello, se añaden nuevas características que buscan **mejorar la calidad** del servicio según los **estándares WS** (Servicios Web) aprobados por **OASIS**. Algunas de estas características que se mejoran son:

   - política de **seguridad**,

- **transacción**,
- **gestión**...

En la actualidad, se reciben **actualizaciones frecuentes** en las características de los servicios SOA, esto es debido a que constantemente están saliendo a la luz **nuevos/as:**

- **fallos de seguridad**,
- **formas de realizar transacciones** de datos,
- etc.

Los servicios SOA, según la fuente, nos los podremos encontrar también como SOAP (Simple Objet Access Protocol), y no debemos confundirlos.

➔ **SOA** es el **modelo de la arquitectura**,

➔ **SOAP**, una **forma de comunicación** (protocolo) **que se permite en SOA**.

La arquitectura **SOA** es una tecnología que nos permite el diseño de aplicaciones basándose en peticiones a un servicio.

Aplicaciones distribuidas

https://bit.ly/3kLSCpa

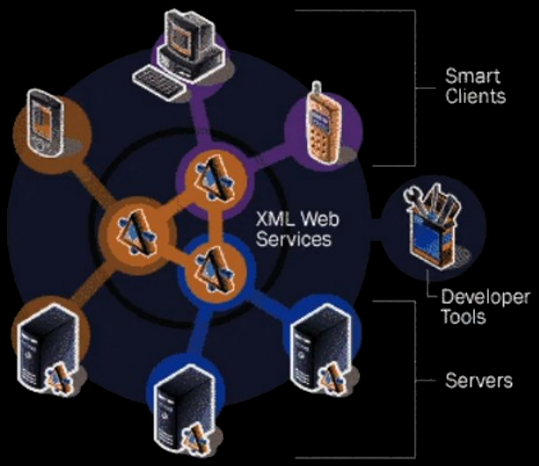
Las aplicaciones distribuidas son aplicaciones con **distintos componentes** que se ejecutan en **entornos separados**, normalmente en **diferentes plataformas** conectadas a través de una **red**. Las aplicaciones distribuidas pueden tener distinto numero de niveles:

1. Dos niveles (son las más sencillas):
    - cliente,
    - servidor.

      Estas son las más sencillas.
2. Tres niveles (suelen ser las más normales):
    - **cliente**,
    - **middleware** (o software de conectividad),
    - **servidor**/es.

      Las más normales suelen ser las de tres niveles.
3. Multinivel que suele ser 4 o mas:
    - cliente,
    - web de **presentación**,
    - empresarial o de **negocio,**
    - y de **acceso a datos**.

Ejemplo de aplicaciones distribuidas



Aquí tenemos **varios servidores** que están ofreciendo el servicio de la aplicación. Podemos ver los **servicios web xml** que permitirán la comunicación y podemos ver varios tipos de clientes que se conectan a los servidores mediante el xml.

Esta imagen que tenemos aquí sería un servicio de tres niveles, donde tenemos:

- los clientes,
- el software de conectividad,
- los servidores.

Fuente de la imagen:
https://arquitecturaortentadaalservicioyenis.wordpress.com/2011/10/19/%C2%BFque-es-soa-la-arquitectura-orientada-a-servicios/

Creación de servicios web SOAP

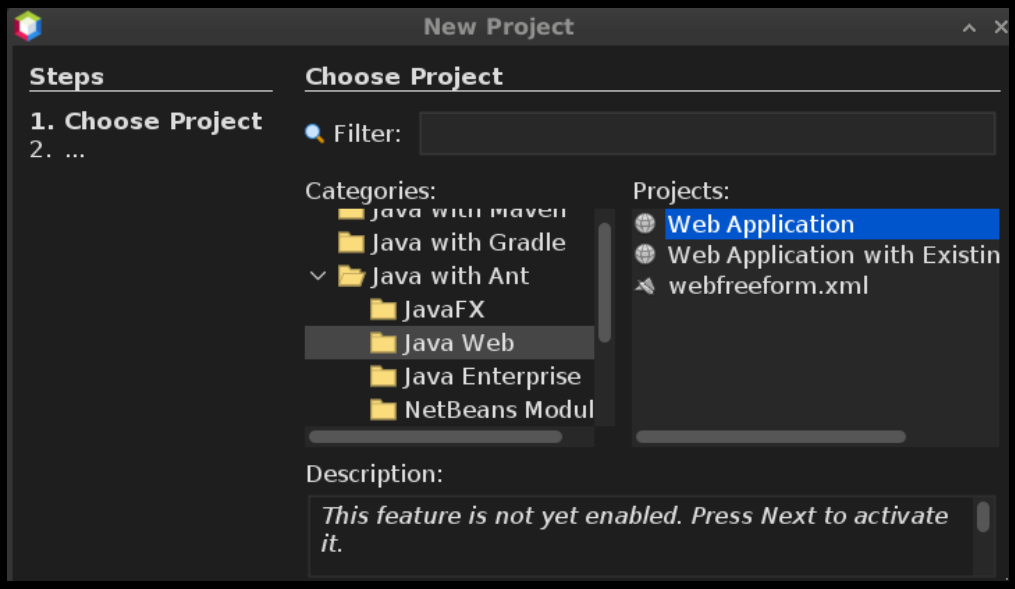
Vamos a ver paso a paso cómo podemos crear un servicio web SOAP, que nos permita calcular el volumen de una esfera.

Una vez hecho esto **crearemos un cliente web** que haciendo uso del servicio anterior sea capaz de calcular dicho volumen.

(**Nota**: Antes de nada, mencionar que este proyecto usará el servidor Glashfish, el usa el JDK 8, de modo que debemos de tener configurado el JDK 8 para que Netbeans lo detecte y pueda trabajar con Glassfish. Además deberá estar configurado el servidor Glassfish antes de la ejecución de la aplicación, en el siguiente enlace explica cómo hacerlo: https://www.youtube.com/watch?v=v0f1XMFc2RY)

SERVIDOR:

• Lo primero es crear una **aplicación web en Java**, para ello seleccionamos Java Web y **Web Application**.



• En la siguiente pantalla deberemos indicar el nombre del proyecto, en este ejemplo los llamaremos WebServiceEsfera.

• Aceptado, nos aparecerá la pantalla de **configuración de GlassFish**, que será el servidor que vamos a utilizar para nuestros proyectos con NetBeans. Este viene integrado por defecto en la versión completa de NetBeans 8.2. Dejamos la configuración de GlassFish como está por defecto.

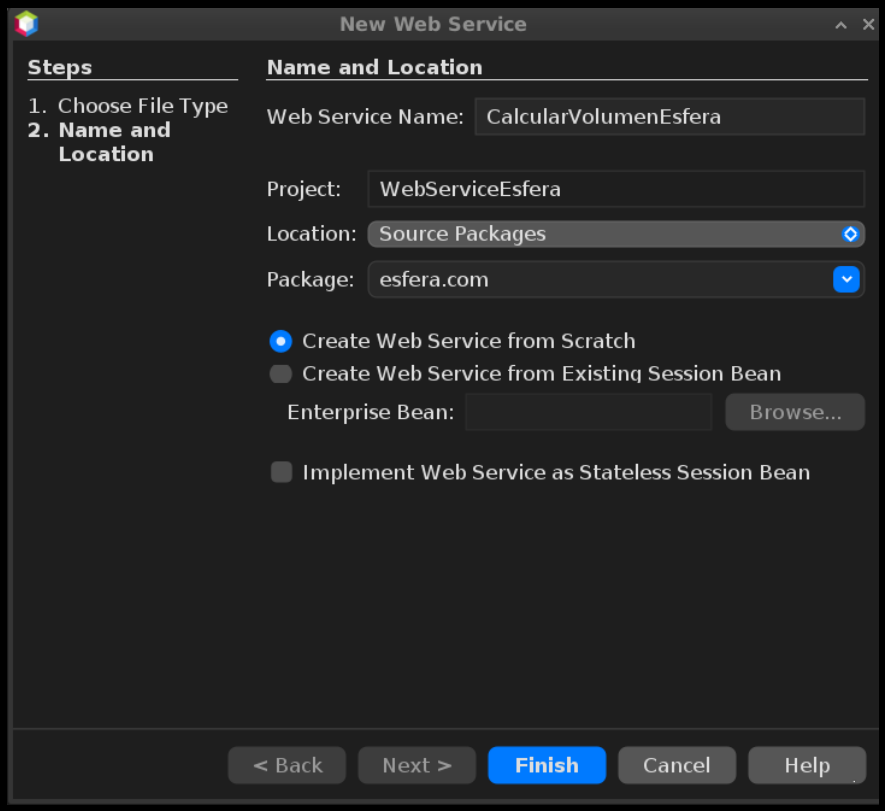
• Una vez creado el proyecto, vamos a crear un **paquete nuevo**, para ello pulsamos nuevo y **paquete Java**.

• Llamaremos al paquete esfera.com.

• El siguiente paso será el de **agregar un nuevo web service**, para ello pinchamos en:

Nuevo → web service

y lo llamamos CalcularVolumenEsfera. Este estará dentro del paquete esferas.com.

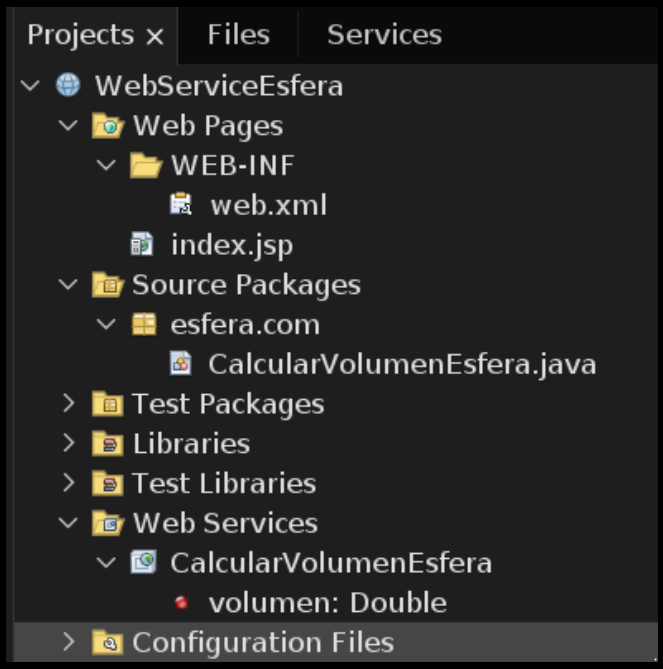


• Ahora dentro de nuestro paquete tendremos un fichero llamado CalcularVolumenEsfera.java con un método *hello*. Borramos el método hello y escribimos el método que nos permitirá calcular el volumen de la esfera.

Método para calcular el volumen de la esfera

```
@WebMethod(operationName = "volumen")
public Double calcularVolumen(
            @WebParam(name = "radio") Double radio) {
      return (4.0 / 3.0) * Math.pow(radio, 3);
}
```

• Si nos fijamos en la **carpeta Web Services**, el **servicio** que creamos previamente se nos ha **actualizado** y ahora mostrará el **web service calcularVolumen**.



• El siguiente paso sería **lanzar el proyecto**, para ello, pulsamos el botón de ejecutar y se iniciarán todos los servicios necesarios (la primera vez suele tardar un poco y además, puede que el firewall lance algún tipo de aviso, si esto ocurre, habría que permitir el acceso de la aplicación).

Si hemos seguido todas las indicaciones podremos acceder a nuestro servicio mediante la dirección:

http://localhost:8080/WebServiceEsfera/

• Ahora podremos **modificar el fichero HTML** como deseemos para darle la interfaz que más nos guste, pudiendo utilizar CSS para ello.

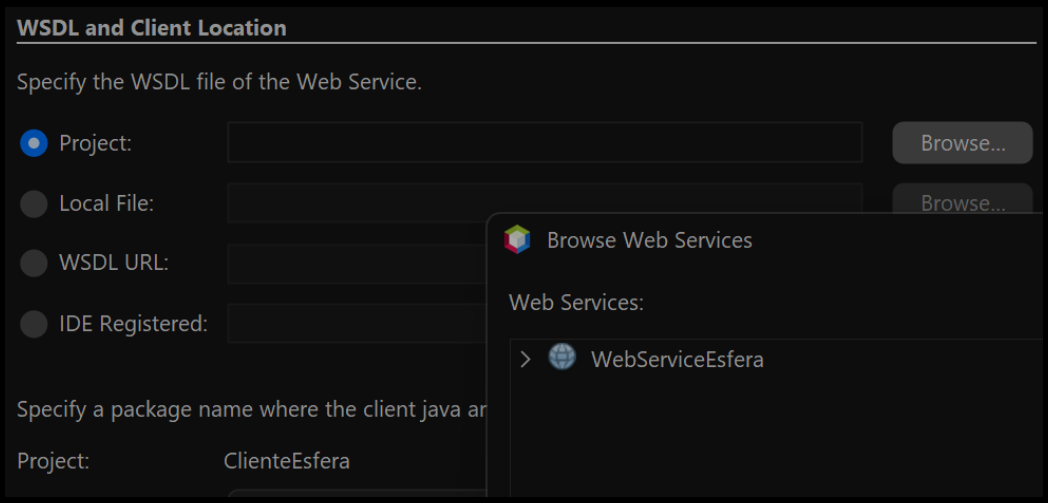
CLIENTE:

Con el servicio ya implementado, crearemos **otro proyecto Java web** que nos servirá de **cliente de nuestro servicio**, el cual llamaremos **ClienteEsfera**:

• Para ello, **creamos un Web Service Client** (nota: no está esa opción, creo otro web application, y al tenerlo creado ya sí aparece la opción al hacer click derecho sobre el proyecto), pinchando en:

Nuevo → web service client.

Accedemos a su configuración y **seleccionamos el proyecto de calcular el volumen de la esfera**.



• Una vez realizado esto vamos a **crear un jsp** para que se ejecute nuestro cliente:

Pulsamos en

web pages → nuevo → jsp

Lo llamaremos **index**. Por ahora en el proyecto disponemos de dos index:

- un html,
- un jsp.

Procederemos a crear nuestro cliente en el jsp, que nos permitirá escribir código Java, no obstante, si ejecutamos el cliente tal cual está ahora, se ejecutará el html. Para cambiarlo, accederemos a las propiedades del proyecto cliente y seleccionaremos **run**. Una vez aquí, observamos una opción denominada:

Relative URL

donde escribiremos /index.jsp para que, al ejecutarse, lance el fichero index.jsp.

• A continuación, haciendo uso de la **paleta de elementos** disponibles crearemos un **formulario** con un input para introducir el radio y un botón para realizar los cálculos.

• El siguiente paso sería escribir el código Java de nuestro cliente. Para ello, nos dirigiremos al proyecto cliente, a la carpeta

Web Services References → CalcularVolumenEsfera → CalcularVolumenEsferaPort

donde observaremos un punto rojo en el servicio de calcular el volumen, lo seleccionamos y lo arrastramos dentro de nuestro jsp, concretamente debajo del formulario y dentro del body. Para terminar, habría que **completar el código tal y como está implementado en el fichero index.jsp** que podrás encontrar en la sección de Recursos del tema.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <div>Volumen de la una esfera</div>
        <form action="index.jsp" method="POST">
            <input type="text" name="radio" value="1" />
            <input type="submit" value="Calcular" />
        </form>
    </body>
</html>
```

• Finalmente, compilamos nuestro proyecto servidor, y posteriormente el cliente, y ya podremos ejecutar nuestro servicio web.

“Ejemplo SOAP completo”

https://bit.ly/3nEFGmW

Conclusión:

Cuando estamos acostumbrados a desarrollar programas de escritorio en Java es normal que nos sorprenda la idea de que con Java se puedan crear páginas web, aunque sería más correcto indicar que son servicios web, no páginas web en sí.

Hay multitud de lenguajes de programación que nos permiten crear páginas y servicios web, como pueden ser **Python** (que ya conocemos de Sistemas de Gestión Empresarial) con el framework **Django**, o incluso el ya citado y más que conocido por nuestra parte Java, que nos va a permitir crear servicios web que se implementarán en un servidor y que los clientes podrán consumir. Aunque seguro que lo sabes, no obstante, cabe destacar que no hay que confundir Java con Javascript, ya que son lenguajes diferentes, y se parecen únicamente en su nombre.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

La programación segura
Amenazas y ataques
Vulnerabilidades
Expresiones regulares.

Introducción a la seguridad

Podemos definir seguridad como el hecho de estar **libre de todo daño**. En informática, la seguridad es **imposible** de conseguir, es decir, ningún programa software va a estar al 100% seguro ante amenazas, ya que día a día surgen **nuevos tipos de riesgos** que desconocemos.

No obstante, en los sistemas informáticos, sean cual sean su composición (programas, aplicaciones, webs, sistemas operativos...), podemos decir que son seguros, o que se pueden acercar a disponer de la máxima seguridad, siempre y cuando cumplan las siguientes **características**:

1. Confidencialidad: Esta característica va a requerir que únicamente las personas **autorizadas** accedan al sistema.
2. Integridad: Requerirá que únicamente las personas autorizadas puedan **modificar** la información existente en el sistema, entendiendo por modificación de la información la **escritura, lectura, modificación, creación y envío** de mensajes.
3. No repudio: Esta cuestión hará que un usuario **no pueda negar que ha enviado** un mensaje. Con el no repudio se va a **proteger al receptor del mensaje** si el emisor niega que ha enviado dicho mensaje. Una forma de no repudio son las **firmas** y los **certificados digitales**.
4. Disponibilidad: Requerirá que todos los recursos del sistema estén **siempre** disponibles para el uso de los usuarios **autorizados**.

Para poder tratar cualquier problema con la seguridad, todas las empresas deben tener obligatoriamente una **política de seguridad** firmemente establecida. La función de la política no es otra que la de dejar claro cuáles son las **responsabilidades y reglas** a seguir **para evitar amenazas**.

Existen una serie de **organismos oficiales** a nivel mundial, que son los encargados de asegurar los servicios de prevención de riesgos y de la asistencia a los tratamientos de cualquier posible incidencia. Uno de ellos es el Computer Emergency Response Team Coordination Center del Software Engineering Institute de la Universidad Carnegie Mellon, el cual es un **centro de alerta y reacción** frente a los ataques informáticos.

En España tenemos el Instituto Nacional de Ciberguridad, o por sus siglas, INCIBE.

Elementos susceptibles de amenazas

Desde el punto de vista de la informática, existen tres tipos de elementos que son los que pueden sufrir amenazas, los cuales son el **hardware**, el **software** y los **datos**. Para cada uno de estos elementos se deben crear medidas de seguridad para no ponerlos en riesgo.

Amenazas de seguridad

Una amenaza de seguridad es una **condición del entorno** de un sistema de información, que una vez dada una oportunidad, **puede producir una violación** de seguridad en el sistema.

Las condiciones del entorno pueden referirse tanto a **personas**, como a **máquinas** que realicen un ataque.

Existen los siguientes **tipos de amenazas**:

Amenazas por el **origen**:

> Estas se dan por el hecho de **conectar un sistema a cualquier entorno**, ya que esto propicia que haya atacantes que puedan entrar en nuestro sistema o alterar el funcionamiento normal de la red. Hay que remarcar que no por no estar conectados a Internet signifique que estamos a salvo. La amenaza puede provenir de **cualquier punto de la red aún sin tener salida** a internet.

Amenazas por el **efecto**:

> Este tipo de amenazas pueden ser:
>
> - **robos** de información,
> - **anulación** de los sistemas informáticos,
> - **suplantación** de identidad,
> - robo de **dinero**,
> - **venta** de datos personales,
> - **destrucción** de información,
> - **estafas**,
> - etc.

Amenazas por el **medio utilizado**:

> Estas pueden ser:
>
> - **virus**,
> - ingeniería **social**,
> - ataques de **denegación** de servicio,
> - **phishing**,
> - etc.

Todas estas amenazas las podemos clasificar principalmente en **cuatro grandes grupos**:

1. Interrupción: En este grupo tenemos todas las amenazas que consiguen **destruir recursos** del sistema informático, pudiendo dejarlo totalmente inservible, provocando así grandes **pérdidas** de información y, posiblemente, de dinero.

2. Intercepción: En este grupo tenemos todas las amenazas que consiguen **acceder a un recurso** de otra persona, pudiendo lucrarse del mismo pidiendo un **rescate** por los datos robados.

3. Modificación: En este grupo tenemos todas las amenazas que intentan **acceder a un recurso** de otra persona, pudiendo llegar a **modificarlo**. (Según un test: aquí se clasifica un programa para escuchar el tráfico de la red).

4. Fabricación: En este grupo tenemos todas las amenazas que impliquen un ataque **contra la autenticidad** de los datos, insertados **datos faltos** en los originales.

Tipos de virus

Un virus informático es un sistema de software dañino, escrito intencionadamente para entrar en una computadora sin permiso o conocimiento del usuario. Tiene la capacidad de replicarse a si mismo, continuando así su propagación.

Algunos virus no hacen mucho mas que replicarse, mientras que otros pueden causar graves danos o afectar negativamente el rendimiento de un sistema.

Un virus nunca debe ser considerado como inofensivo y dejarlo en un sistema sin tomar medidas.

1. Virus de **acción directa**:

   El objetivo principal de estos tipos de virus informáticos es replicarse y actuar cuando son ejecutados. Cuando se cumple una condición especifica, el virus se pondrá en acción para infectar a los ficheros en el directorio o carpeta que se especifica en el archivo autoexec.bat.

2. Virus de **sobreescritura**: Estos tipos de virus informáticos se caracterizan por el hecho de que borran la información contenida en los ficheros que infectan, haciéndolos parcial o totalmente inútiles. Una vez infectados, el virus reemplaza el contenido del fichero sin cambiar su tamaño.

3. Virus de **sector de arranque**: Este tipo de virus afecta al sector de arranque del disco duro. Se trata de una parte crucial del disco en la que se encuentra la información que hace posible arrancar el ordenador desde disco.

4. Virus **polimórfico**: Estos tipos de virus informáticos se encriptan o codifican de una manera diferente, utilizando **diferentes algoritmos y claves de cifrado** cada vez que infectan un sistema. Esto hace **imposible** que el software **antivirus** los encuentre utilizando búsquedas de cadena o firma porque **son diferentes cada vez**.

Técnicas seguras en Java

Un concepto propio de la seguridad en los sistemas es, por ejemplo, el **tratamiento de excepciones** mediante los **bloques try-catch**. Si repasamos para qué servían estos bloques recordaremos que su finalidad era la de controlar las posibles excepciones, errores o **ejecuciones no deseadas** que pudiesen ocurrir durante la ejecución de un programa.

Ya estudiamos que una excepción era un evento que podía ocurrir en tiempo de ejecución de una aplicación y que interrumpiría el flujo normal de las instrucciones de la misma, provocando que la aplicación se detuviera. Si analizamos esto, podemos ver que para luchar contra las ejecuciones no deseadas podemos utilizar un bloque try-catch, lanzando una **excepción que pare el proceso no deseado**. Como sabemos, en el lenguaje de programación Java podemos encontrar varios tipos de excepciones:

- Error. Estas son excepciones que van a indicar **problemas muy graves**, los cuales por norma general **no suelen ser recuperables** y que no deben casi nunca ser capturadas.
- Exception. Estas son excepciones que no son definitivas, pero que vamos a poder **detectar en tiempo de codificación**.
- RuntimeException. Estas son excepciones que se dan durante el **tiempo de ejecución** del programa o aplicación.

Otro ejemplo de técnica de seguridad es la de las **validaciones** de entradas de datos mediante **expresiones regulares**.

Ataques a un sistema informático

Ya sabemos que todos los sistemas informáticos están expuestos a ser atacados de una forma u otra, y que no hay una forma de estar libre al 100% de esta posibilidad, aunque si cumplimos con ciertos requisitos de seguridad, como los vistos anteriormente, podremos **interponer obstáculos** y que a los atacantes **no les resulte fácil atacarnos**.

Los ataques a los sistemas informáticos los podemos definir como ciberataques. Es muy importante tener en cuenta que hoy en día, algunos ciberataques, dependiendo de dónde se realice, a quién o cuándo, pueden llegar a formar parte de una **guerra informática** o de un ataque de **ciberterrorismo**.

Los ataques que podemos recibir se pueden clasificar en **dos grandes grupos** principalmente:

1. Ataques **pasivos**:

   Esta categoría engloba a los tipos de ataques en los que el atacante **no necesita alterar la comunicación**, este únicamente se dedicará **escuchar o monitorizar** el tráfico de información en la red para intentar **obtener información** que está siendo transmitida. A este tipo de ataques se les suele dar uso para intentar ver el **tráfico de red**, pudiendo conseguir de esta forma **credenciales** de acceso, como usuarios y contraseñas, **páginas web** que están siendo visitadas, etc.

2. Ataques **activos**:

   Esta categoría engloba a los tipos de ataques en los que el atacante realiza algún tipo de **modificación de los datos** que están siendo transmitidos en la red, o incluso pudiendo llegar a crear un **flujo de datos falso** que se transmitirá por la red como si de un verdadero se tratase.

   Los objetivos de estos ataques son:

   a. Intentar una suplantación de identidad, haciéndose pasar el atacante por el usuario suplantado.

   b. Una reactuación, pudiendo **reenviar una serie de mensajes** determinados para producir un efecto no deseado.

   c. Ataques de denegación de servicio, pudiendo de esta forma **apagar sitios** web.

Vulnerabilidades en el software

Las vulnerabilidades del software surgieron en el mismo momento en el que surgió Internet, y aún a día de hoy sigue siendo uno de los mayores problemas a los que nos debemos enfrentar.

Éstas son un **fallo o hueco en la seguridad** de un software o sistema informático, que ha sido **detectado** por algún otro sistema, o persona que monitoriza ese sistema malicioso, el cual, puede ser utilizado para **entrar** en el sistema destino **de forma no autorizada**, pudiendo realizar operaciones indeseadas.

El mayor inconveniente de las vulnerabilidades de software reside en la **dificultad en la forma en la que son detectadas**. De hecho, cuando se trata de un programa con una gran **envergadura**, las **auditorías de vulnerabilidades** normalmente son encargadas a **empresas externas** que se dedican a buscar **fallos** de software, las cuales, cuentan con **expertos** en la materia.

Cuando se crea una aplicación software, **antes de lanzarla al mercado** se hace un estudio meticuloso en el que se intentan descubrir todos los **fallos de seguridad** de la misma, pudiendo **identificarlos y solucionarlos** sin haber puesto la aplicación en producción. En este proceso se pueden descubrir una gran cantidad de fallos de seguridad, pero desgraciadamente, estos no serán todos los que surgirán, por lo que aún con la aplicación en el mercado, se deberá seguir con este proceso ininterrumpidamente, monitoreando de forma continua el programa, y lanzando **actualizaciones y parches** que solventen los nuevos **fallos** de seguridad **encontrados**.

Como en este módulo estamos centrados en el **lenguaje de programación Java**, vamos a ver los **dos pilares de la seguridad** en los que se basa éste:

1. Seguridad interna de la aplicación.

   Esto se refiere a que cuando se programe una aplicación han de seguirse unos **criterios de tratamiento de errores**.

   Un ejemplo serían los **bloques try-catch-finally** para el tratamiento de excepciones.

2. Políticas de acceso.

   Las políticas de acceso se refieren a las **acciones** que puede realizar la **aplicación en nuestro equipo**, es decir, el hecho de dar o no **permiso** a la aplicación para que pueda **utilizar ciertos recursos** del sistema.

Mecanismos de control de acceso

En el ámbito de la seguridad informática existen lo que se denominan políticas de seguridad:

> Las podemos definir como una serie de **documentos de muy alto nivel** que van a ser **fundamentales** para llevar a cabo el compromiso con la **ciberseguridad**.
>
> Estos documentos contienen:
>
> - la definición de la seguridad de la información,
> - las medidas a adoptar tanto por la **empresa**, **trabajadores**, así como el **departamento** de sistemas, para aplicar todos los procedimientos necesarios que garanticen que el trabajo se desarrolla en un entorno seguro.

Las políticas de seguridad por sí solas no son suficientes y deben de ser utilizadas **con otras medidas** que van a depender de las mismas, como pueden ser:

- los objetivos de seguridad,
- los procesos.

Las políticas de seguridad deben ser **fácilmente accesibles** para que **todo el personal** de la empresa **esté al tanto** de su existencia y **entiendan** su contenido.

Seguridad en la aplicación mediante tokens

Si debemos hacer que las peticiones de realización de las operaciones que los usuarios mandan al servidor de la empresa sean seguras, intentando **validar a los usuarios** de alguna forma, una de las formas más sencillas y efectivas de hacer que sea seguro es mediante una 'tokenización' de los usuarios.

La 'tokenización' consiste en asignar un **token único** a cada uno de los **usuarios**, teniendo que **verificarlo en todas las peticiones** que realice al servidor de la empresa.

En un primer momento, esto puede parecer que va a ralentizar el sistema, pero, al fin y al cabo, no va a ser más que un bloque if-else, lo cual, no va a influir demasiado en el tiempo de las operaciones.

En caso de que el **token enviado no sea correcto** o no exista, se **lanzará un error** desconocido.

Validación de entradas

Una forma de entrada de errores en nuestras aplicaciones que puede afectar a la seguridad, son las entradas de datos que introducen los usuarios, por ejemplo, en el momento del control de acceso a los sistemas, o simplemente, cuando nos encontramos desarrollando un nuevo programa. Uno de los fallos de seguridad más comunes se trata de los conocidos como buffer overflow, los cuales, se producen cuando se **desborda el tamaño de una determinada variable**.

Para evitar parte de estos errores podemos usar la **validación de datos mediante expresiones regulares**, la cual nos va a **permitir**:

1. Mantener la consistencia de los datos, pudiendo comprobar que los datos insertados cumplen ciertos **requisitos**.
2. Evitar desbordamientos de memoria buffer overflow, pudiendo **comprobar la longitud** de los campos.

Para poder usar las expresiones regulares, Java usa la biblioteca java.util.regex, la cual nos va a permitir usar la clase Patter, que permite la **definición de expresiones regulares**. Para la validación de entrada podemos utilizar los siguientes elementos y operadores:

Elementos y operadores de expresiones regulares

Elementos:

- x → El carácter x.
- [abc] → Los caracteres a, b o c.
- [a-z] → Una letra minúscula.
- [A-Z] → Una letra mayúscula.
- [A-Za-z] → Una letra minúscula o mayúscula.
- [0-9] → Un número entre 0 y 9.
- [A-Za-z0-9] → Una letra minúscula, mayúscula o un número.

Operadores:

- [a-z]{2} → Dos letras minúsculas.
- [a-z]{2,5} → Entre dos y cinco letras minúsculas.
- [a-z]{2,} → Más de 2 letras minúsculas.
- hola|adios → Solo se pueden introducir las palabras hola o adiós. | (barra vertical) funciona como la operación OR.
- XY → Solo se pueden introducir dos expresiones. Esta es la operación AND.
- e(n|l)campo → Los **delimitadores** ( ) permiten hacer expresiones más complejas. En el ejemplo se debe introducir el texto “en campo” o “el campo”.

Ejemplo de validación de datos

Validación de DNI mediante Expresiones Regulares en Java

Exploraremos cómo validar la entrada de un Documento Nacional de Identidad (DNI) mediante expresiones regulares en Java.

En primer lugar, hemos creado **dos expresiones regulares** que nos permitirán validar un DNI:

1. La primera expresión regular indica que necesitamos dígitos en una **cantidad de 8**, seguidos por una **letra mayúscula**.
2. En la segunda expresión regular, lo único que **cambia** es la **forma de indicar que necesitamos dígitos**.

Ambas expresiones regulares **funcionan de la misma manera**.

Posteriormente, hemos creado **cuatro variables** con cuatro posibles **DNIs a comprobar**.

- El único DNI **válido es el primero**, ya que tiene ocho dígitos y una letra mayúscula.
- En el segundo, la letra es minúscula, lo cual debería dar falso.
- En el tercero, falta un dígito,
- y, en el cuarto, hay dos letras.

Para comprobar la expresión regular, utilizamos la clase `Pattern`, que debemos importar del paquete `java.util.regex`. En el método `matches`, podemos verificar si una expresión regular coincide con una cadena. En este caso, le pasaremos la expresión regular 1 y luego podremos probar los otros tres DNIs.

Si ejecutamos nuestro programa, observamos que nos indica que el DNI es correcto. Si cambiamos la expresión 1 por la 2, el DNI sigue siendo correcto. Ahora, al probar el DNI 2, muestra que no es correcto, el DNI 3 tampoco es correcto y el DNI 4 tampoco es correcto.

```
    public static void main(String[] args) {
            // Expresiones regulares para el DNI
            String dniexpresionreg1 = "\\d{8} [A-Z] {1}";
            String dniexpresionreg2 = "(0-9] {8} [A-Z]{1}";

            // DNIs para comprobar
            String dni1 = "25841796T"; // true
            String dni2 = "25841796t"; // false, la letra es minúscula
            String dni3 = "2584179T"; // false, falta un digito
            String dni4 = "25841796TT"; // false, tiene dos letras

            // Comprobamos si es correcto el DNI a comprobar
```

```
        if (Pattern.matches(dniexpresionreg1, dni1))
        {
              System.out.println("El DNI es correcto.");
        }
        else
        {
              System.out.println("El DNI no es correcto.");
        }
  }
```

Reflexión

El hecho de que se revisen aspectos de seguridad en un momento determinado, no exime de volver a revisar antes de entregar la aplicación al cliente, o lanzarla a producción. Ya que, por ejemplo, durante el periodo en el que no se ha revisado, un usuario puede ejecutar un programa de terceros e instalar un software malicioso en el PC del trabajo, sin ser siquiera consciente de ello.

Por esta razón y por muchas otras, cuando creamos una aplicación debemos tener revisado en todo momento, el aspecto de la seguridad, sobre todo, en la transmisión de los datos, en el almacenamiento de los datos, en las entradas de información a nuestra aplicación, en las entradas al sistema de los usuarios, etc.

Si el tiempo no permite hacer las revisiones oportunas sobre seguridad, habría que negociar con el cliente la fecha de entrega de los trabajos, ya que será más perjudicial entregar un proyecto a tiempo inseguro, que tarde, pero totalmente seguro.

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

La criptografía
Clave pública y privadas
Firmas y Certificados digitales
Funciones HASH

Vamos a ver cómo se realiza el proceso de encriptado de la información, además de estudiar los dos modelos más comunes de la criptografía, el modelo de **clave privada** y el modelo de **clave pública**.

Por último, vamos a aprender cómo se puede aplicar la criptografía a los programas que desarrollemos mediante las **firmas digitales** y los **certificados digitales**, mediante funciones HASH.

Concepto de criptografía

La palabra criptografía proviene del griego "**cripto**", que significa **secreto**, y "**grafía**", que significa **escritura**, por lo que la palabra criptografía significa **escritura secreta**.

La criptografía fue creada y está siendo utilizada (entre otros usos) para poder enviar información confidencial o **mensajes privados** a ciertas personas u organizaciones.

Los pasos a seguir para poder aplicar la criptografía a un mensaje son los siguientes:

1. Se escribe el **mensaje 'normal'**, es decir, sin encriptar.

2. Se utilizan **técnicas de encriptado**, más o menos sofisticadas para codifica el mensaje deseado.

3. Se puede **enviar el mensaje** ya encriptado por una línea de **comunicaciones seguras, o no seguras.** Si estamos en el segundo caso, el mensaje enviado puede ser interceptado. Esto no entrañaría ningún peligro inicialmente, ya que estaría encriptado. No obstante, si el método de encriptación llevado a cabo es conocido o descifrado por el ente que lo intercepta, la información quedaría al descubierto.

4. Cuando el **receptor reciba** el mensaje, aplicará la **técnica de desencriptado** para poder ver el mensaje original.

Junto a la criptografía podemos destacar el criptoanálisis, que es una ciencia que se dedica a estudiar la **fortaleza o robustez que tienen los sistemas criptográficos**, pudiendo comprobar cómo de seguros son en realidad.

Mediante el criptoanálisis se están mejorando día a día los sistemas criptográficos.

Existen diferentes tipos de criptografía:

- Criptografía simétrica.
- Criptografía de clave pública o criptografía **asimétrica**.
- Criptografía con umbral.
- Criptografía basada en identidad.

- Criptografía basada en certificados.
- Criptografía sin certificados.
- Criptografía de clave aislada.

Los tipos de criptografía más utilizados en un **entorno profesional**, son los de **criptografía simétrica** y **asimétrica**, que son los que vamos a estudiar más adelante en esta unidad.

Tecnologías de la seguridad de la información

Algunas de las principales tecnologías referentes a la seguridad de la información en informática son:

- **Cortafuegos**,
- Administración de **cuentas de usuarios**,
- **Detección y prevención** de intrusos,
- **Antivirus**,
- Infraestructura de **llave publica**,
- **Capas de Socket Segura** (**SSL**),
- **Conexión única** "Single Sign On - **SSO**",
- **Biométrica**,
- **Cifrado**.

Aplicaciones de la criptografía

Una de las aplicaciones más emergentes de la criptografía, y sobre la que más se puede estar innovando en la actualidad, es el concepto de la **cadena de bloques** blockchain, ya que éste, utiliza **diferentes tipos de criptografía** para garantizar la **seguridad de las transacciones**.

En primer lugar, se utiliza el tipo de criptografía HASH, mediante la cual, se pueden **convertir grandes cantidades de información** en una combinación de **letras y números** única, y muy difícil de imitar. Con esto queremos decir que básicamente se van a resumir enormes cantidades de información, pudiéndose comprobar rápida y fácilmente, que **todos los procesos realizados por los nodos de la blockchain coincidan**. Por otra parte, el usar un código HASH nos va a permitir la creación de las **claves públicas y privadas**, con las que se **reciben y envían criptomonedas**.

Dentro de los datos de la **cadena blockchain**, son utilizadas **diferentes capas** de criptografía, que solamente pueden ser resueltos por **ordenadores** de una **potencia considerable**.

Ya en otro ámbito, concretamente en la cotidianeidad de nuestro día a día, uno de los usos más comunes que tiene la criptografía está en Internet.

Cuando accedemos a un sitio web denominado como HTTPS (fácilmente visible en el apartado de la URL de nuestro navegador favorito), éste utiliza el **protocolo** de seguridad denominado **SSL** (que estudiaremos en la siguiente unidad), por sus siglas en inglés, **Secure Sockets Layer**. Este protocolo se encarga de **cifrar todos los datos** que el usuario pueda enviar al servidor utilizando **diferentes algoritmos criptográficos**.

Un último ejemplo del uso de la criptografía está en el uso de cualquier **sistema financiero** virtual, como puede ser **PayPal**.

Así que recuerda, cada vez que accedas a determinadas páginas webs, realices una compra online..., estás haciendo uso de todo el potencial que nos ofrece la criptografía.

La mejor encriptación

En cuanto a la elección del mejor encriptado, habrá que prestar especial atención a los aspectos de:

- Longitud de la cadena: La cadena encriptada **más larga** será la **más complicada de revertir** en caso de ataque.
- Tipo de caracteres usados: Una encriptación será más segura si usa **diferentes tipos de caracteres**, como pueden ser letras minúsculas y mayúsculas, números y símbolos especiales.
- El tiempo de encriptado: El tiempo que se tarda en encriptar la información también es muy importante, ya que podemos tener una información muy segura, pero **tardar en encriptarse** mucho tiempo, lo cual **no es aconsejable**.

Encriptación de la información

Podemos definir formalmente la encriptación o el cifrado de la información como el proceso por el que la información o los **datos** que se desean proteger son **codificados**, dando lugar a un **texto** que parece ser **aleatorio**, o sin sentido para los humanos.

Igualmente, podemos definir formalmente la desencriptación como la operación inversa a la encriptación, mediante la cual, los datos encriptados se transforman mediante las **técnicas inversas** del algoritmo utilizado, para **encriptarlos en el texto original**.

Es importante conocer los siguientes conceptos básicos para poder hablar de criptografía y encriptación:

- Texto plano:

  Se refiere al texto **original**, sin aplicar ningún algoritmo de encriptación.

- Texto cifrado:

  Se refiere al texto resultado de **aplicar el algoritmo** de encriptación al texto original.

- Algoritmo de cifrado o algoritmo criptográfico:

  Es el algoritmo que utilizaremos para poder **encriptar o cifrar** el texto plano para dar lugar al texto cifrado. **Junto al algoritmo** de cifrado **existe una clave**.

- Clave:

  Cadena de caracteres que serán la **base para el algoritmo** de cifrado, y que permitirán pasar del texto plano al cifrado. Cada clave diferente proporcionará como salida un **texto cifrado diferente**. Esta clave puede ser **simétrica o asimétrica**.

Como resumen podemos decir que el proceso de encriptado de la información se puede representar mediante la siguiente fórmula:

Cifrado → F_K (M) = C

Donde F es el algoritmo que vamos a utilizar para cifrar la información, K será la clave de cifrado, M será el mensaje que queremos cifrar y C será el texto ya cifrado.

Dicho de otra manera:

Para conseguir un cifrado de mensaje: aplicamos un algoritmo de cifrado usando la clave de cifrado sobre el mensaje a cifrar, lo que dará lugar al texto ya cifrado.

Principios de la criptografía

Vamos a ver cuáles son los principios de la criptografía.

Las propiedades que son más deseables en un sistema criptográfico fueron anunciadas por August Kirchhoff en el año 1883. De entre ellas podemos destacar las siguientes:

1. **Si el sistema no es teóricamente irrompible, en la práctica al menos sí debe serlo**.

    Esto se debe a que ningún sistema es 100% seguro y tarde o temprano se terminará rompiendo aunque sea teóricamente irrompible.

2. La **efectividad** que tiene el sistema **no debe depender** de que el **diseño** del mismo sea **secreto**.

3. La clave que vamos a usar tanto para encriptar como para desencriptar los mensajes debe ser **fácilmente memorizable**.

    De esta manera evitamos tener que escribirlas y que puedan ser sustraídas.

4. Los sistemas criptográficos deben dar resultados **alfanuméricos**.

    De esta forma los sistemas serán mucho más seguros y las claves más difíciles de romper.

5. El sistema de criptografía debe ser **operado únicamente por una única** persona para que así sea todo más seguro.

6. El **sistema** debe ser **fácil de utilizar** para que de esta forma la persona que se encarga del sistema no tenga problemas.

Según un test:

El diseño del sistema puede ser público.

Criptografía de clave privada o simétrica

La criptografía de clave simétrica o privada es un método de encriptado que utiliza una **clave** que es secreta, la cual **solo pueden conocer el emisor y el receptor**. Este tipo de criptografía es m**uy apropiada** si queremos garantizar la **confidencialidad**.

A este tipo de criptografía se la denomina simétrica porque la **clave de encriptado y desencriptado** es exactamente la **misma**.

Podemos decir que las principales características de la criptografía simétrica son:

1. La clave es secreta, debiendo conocerla solo las partes involucradas en la comunicación, es decir, el **emisor y el receptor**.
2. Se utiliza la misma clave para cifrar y para descifrar los mensajes de la comunicación.
3. Estos algoritmos de cifrado suelen ser muy rápidos y **no** suelen **aumentar el tamaño** del mensaje, es debido a esto que son muy apropiados para **encriptar grandes cantidades** de texto.

La criptografía de clave privada también tiene una serie de inconvenientes, pudiendo destacar los siguientes:

• Como la clave de cifrado y descifrado es la misma, en el momento del envío de ésta por parte del emisor al receptor, este mensaje puede ser **interceptado** y una persona no deseada puede **hacerse con ella**.

• Como las claves se utilizan en una única comunicación, si se desean comunicar varias personas, deberá haber **una clave para cada combinación** de personas diferentes que vayan a comunicarse, generando así una enorme cantidad de claves.

Una alternativa para **solucionar los problemas de distribución de claves** y todo lo que se deriva de ello, pueden ser la **criptografía asimétrica** y la **criptografía híbrida** que estudiaremos a continuación.

Un ejemplo práctico donde se utilizaba este tipo de criptografía fue la **máquina Enigma**, utilizada por la **Alemania Nazi** en la segunda guerra mundial para **cifrar sus comunicaciones**.

Máquina enigma

Criptografía de clave pública o asimétrica

La criptografía de clave pública surgió para solucionar el problema de distribución de claves que sufría la criptografía de clave privada, permitiendo así tanto al **emisor** como al **receptor** poder **poner en común unas claves mediante un canal** (incluso no seguro) de comunicación.

A este tipo de criptografía se le denomina asimétrica porque las **claves** de encriptado y desencriptado **son diferentes**, a diferencia de la criptografía de clave privada.

Podemos listar las características más importantes de la criptografía asimétrica:

1. Tanto emisor como receptor tienen en su poder un par de claves, una que es **pública**, que es conocida por todo el mundo y que el hecho de conocerla no implica conocer ningún tipo de información sobre la clave privada inversa, y otra que es **privada**, que la **conoce únicamente su poseedor**.
2. Todas las parejas de claves sirven únicamente con ellas mismas, es decir, que son complementarias, y el proceso **no funcionará** si alguna de ellas es **cambiada**.
3. Las claves de encriptado y desencriptado únicamente se pueden generar una vez, de esta forma, es **prácticamente imposible** que dos personas obtengan las mismas claves.
4. Cuando un mensaje es cifrado con la clave pública, únicamente se va a poder descifrar con la clave privada que sea la inversa a esa clave pública.
5. Cuando ciframos un mensaje con la clave privada, estamos demostrando que nosotros hemos sido quienes hemos cifrado dicho mensaje.

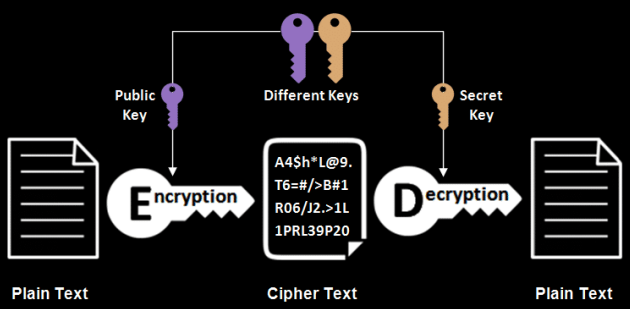
La gran ventaja que ofrece este tipo de criptografía es que **ya no** existe el problema de la **distribución de claves**.

La criptografía de clave pública también tiene algunos inconvenientes:

- Estos algoritmos son algo más lentos.
- Hay que poder garantizar que la clave pública es realmente **de quien dice ser**.

Lo más óptimo sería utilizar una **combinación** de criptografía de clave pública y de clave privada, lo que se conoce como **criptografía híbrida**.

Criptografía asimétrica usada en Bitcoin



Accesos de usuario seguros

Una de las cosas más comunes que deben hacer los programadores de aplicaciones son los accesos de los usuarios a las mismas.

Estos accesos nunca deben implementarse en **texto plano**, ya que con un simple **analizador de tráfico de red** podríamos obtener las **claves de acceso** de una forma extremadamente sencilla.

La idea es que las **claves de acceso** estén **encriptadas en la base de datos** de usuarios de la aplicación, realizando así **dos tareas de forma simultánea**:

1. La primera sería que, al estar esas claves **encriptadas** en la base de datos, si alguien consigue entrar a ella **no podrá obtenerlas**.
2. La segunda es que estamos dotando de una **alta seguridad a los accesos** de los usuarios.

Para realizar estos accesos podríamos seguir los siguientes **pasos**:

- **Obtener la clave** que introduce el usuario.
- **Encriptarla**.
- Una vez encriptada, enviarla por **petición segura HTTPS** para comprobar si el **login en correcto**.

Esquema de acceso seguro

```
// Obtenemos las claves
usuario = obtenerUsuario()
Clave = obtenerClave()

// Encriptamos la clave
Clavesegura = encriptar(clave)

// Comprobamos si el acceso es OK
Si comprobarAcceso(usuario, clavesegura)
        // entramos al sistema
sino
        // mostramos un error
```

Firma digital y certificados digitales

Las firmas digitales son el equivalente a las firmas personales, pero en un entorno tecnológico, es decir, su objetivo es identificar al firmante inequívocamente, pero en lugar de hacerlo en papel, se llevaría a cabo de forma digital.

Las firmas digitales están basadas en:

- criptografía de **clave pública**,
- resumen de **mensajes HASH**.

Un **resumen de mensajes** (**Message-Digest** Algorithm, en inglés) es un algoritmo que para encriptar, toma como entrada un **mensaje con una longitud variable** y lo convierte en un **resumen de una longitud fija**. Algunos algoritmos de este tipo son el **MD5** y el **SHA**.

Otro elemento más que interviene en el proceso de criptografía son los certificados digitales. Éstos se diseñaron para resolver el problema de la confianza que han de depositar las dos partes involucradas en la comunicación, es decir, un certificado digital es un documento electrónico **firmado por un tercero** (entidad certificadora) que da fe de los datos de la firma digital empleada. De forma genérica, podemos decir que vienen a ser como el **notario de la firma digital**.

Según un test:

El certificado digital… Esto garantiza que la persona que ha enviado el mensaje es quien dice ser.

Una entidad certificadora es una organización que **se responsabiliza** de la veracidad de los datos de los firmantes digitales, y, por tanto, de la **emisión y validez** de los certificados oportunos. Creándolos y aportando mecanismos que permitan poder **revocarlos, suspenderlos, y comprobar su validez**.

Extrapolando todos estos conceptos a un entorno de programación, podemos destacar que en **Java** se usa la clase **MessageDigest** y los **algoritmos** que podemos emplear son:

1. **MD2**,
2. **MD5**,
3. **SHA-1**,
4. **SHA-256**,
5. **SHA-384**
6. y **SHA-512**.

Cada vez que vayamos a encriptar un texto deberemos controlar la **excepción** NoSuchAlgorithmException mediante un bloque try-catch.

En el método **getInstance** es donde podremos **indicar cualquiera** de los **algoritmos** listados anteriormente para cifrar el mensaje con una **función HASH**.

Código para encriptar texto con HASH

```
try {
	String password = "esta_es_mi_contraseña_1234";
	MessageDigest md = MessageDigest.getInstance("SHA-256");
	md.update(password.getBytes());
	byte byteData[] = md.digest();
	StringBuilder sb = new StringBuilder();
	for (int i = 0; i < byteData.length; i++) {
		sb.append(Integer.toString((
				byteData[i] & 0xff) + 0x100, 16).substring(1));
	}
	System.out.println("Contraseña -> " + password);
	System.out.println("MD5 -> " + sb.toString());
} catch (NoSuchAlgorithmException error) {
	System.out.println("Error: " + error.toString());
}
```

Ejemplo de funciones HASH

Vamos a ver un ejemplo de codificación con función hash en Java.
En primer lugar, lo primero que vamos a hacer es introducir la contraseña que queremos cifrar y mostrarla.

Ahora vamos a cifrar en primer lugar con el algoritmo hash SHA-256. Para cifrar con cualquier algoritmo hash necesitamos la clase MessageDigest. Con el método getInstance vamos a indicar el tipo de algoritmo con el que queremos cifrar el texto.

Una vez lo hemos indicado, con el método update codificamos el texto y el resultado será con la función digest y vendrá dado en un **array de tipo byte**.
Ahora, mediante un StringBuilder vamos a ir traduciendo el texto cifrado, ya que viene dado en bytes, y de esta forma lo vamos a pasar a string y lo mostramos.
Vamos a repetir el mismo proceso con el algoritmo SHA-512.
Y, por último, haremos el mismo proceso con el algoritmo MD5, mostrando el resultado de los tres algoritmos para comprobar cuál es más seguro. Deberemos realizar todo esto dentro de un **bloque try-catch** para controlar esta excepción.

Ejecutamos el programa para ver el resultado:

- Introducimos una contraseña, por ejemplo, "perro", y aquí tenemos el resultado con el algoritmo SHA-256.
- Con el 512 es más seguro ya que es más largo y con el MD5.
- Si volvemos a ejecutar e introducimos otra contraseña, por ejemplo, "examen", tenemos los resultados de codificar "examen" con los tres algoritmos.

```
Contraseña -> examen
SHA-256 -> 854a557fb0868de7fe6e432766f141f446d3f34f5d9c7e50e0dac94c817d32f9
SHA-512 ->
be30e8557b748fd8d69a11685d36661c5c067ad1331747c4d24c1cd5d621211ace7e2d4f732c58
c9fe57e09768e3cfbf5ca5c9dcc7b3203101f61b433d94ddfb
MD5 -> 32bd3b82800c20c82f979e3cf1b26917
```

Código

```
try {
    Scanner teclado = new Scanner(System.in);
    System.out.println(
            "Introduce la contraseña a cifrar:");
    String password = teclado.nextLine();

    System.out.println("Contraseña → " + password);

    // Ciframos con SHA-256
    MessageDigest md = MessageDigest.getInstance(
            "SHA-256");

    md.update(password.getBytes());
    byte byteData[] = md.digest();

    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < byteData.length; i++) {
        sb.append(Integer.toString((
                byteData[i] & 0xff) +
                0x100, 16).substring(1));
    }

    System.out.println("SHA-256 → " + sb.toString());

    // Ciframos con SHA-256
    MessageDigest md2 = MessageDigest.getInstance(
            "SHA-512");

    md2.update(password.getBytes());
    byte byteData2[] = md2.digest();

    StringBuilder sb2 = new StringBuilder();
    for (int i = 0; i < byteData2.length; i++) {
        sb2.append(Integer.toString((
                byteData2[i] & 0xff) +
                0x100, 16).substring(1));
    }

    System.out.println("SHA-512 → " + sb2.toString());

    // Ciframos con MD5
    MessageDigest md3 = MessageDigest.getInstance(
            "MD5");
```

```
    md3.update(password.getBytes());
    byte byteData3[] = md3.digest();

    StringBuilder sb3 = new StringBuilder();
    for (int i = 0; i < byteData3.length; i++) {
        sb3.append(Integer.toString((
                byteData3[i] & 0xff) +
                0x100, 16).substring(1));
    }

    System.out.println("MD5 -> " + sb3.toString());
} catch (NoSuchAlgorithmException error) {
    System.out.println("Error: " + error.toString());
}
```

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Comunicaciones seguras
Protocolo SSL/TLS
Sockets seguros

Pondremos de manifiesto por qué en una comunicación deberemos encriptar las comunicaciones para hacerlas seguras, utilizando una serie de **protocolos** seguros.

Vamos a estudiar cuáles son esos protocolos seguros, y haremos hincapié en uno de los más importantes, el protocolo **SSL/TLS.**

También seguiremos ahondando sobre el concepto de encriptación, para continuar protegiendo la información de forma segura.

Por último, experimentaremos sobre cómo podremos realizar una comunicación en red de forma segura, utilizando para ello los **sockets seguros**.

Protocolos seguros de comunicaciones

La protección de la información cuando se desarrollan aplicaciones que tienen comunicaciones en red es algo que debemos proporcionar, ya que las comunicaciones mediante una red pueden ser fácilmente interceptadas, y, por lo tanto, pueden ser manipuladas por personas indeseadas.

Cuando combinamos el poder de la **criptografía con** las **comunicaciones** en red, estaríamos creando lo que se conoce como **protocolos seguros de comunicación**, o también llamados protocolos criptográficos, o de encriptación.

Existen multitud de este tipo de protocolos, pero nos centraremos en los dos más comunes:

1. Protocolo SSL:

   "*Secure Sockets Layer*": Este protocolo proporciona la posibilidad de tener una **comunicación segura** en el modelo cliente/servidor, protegiéndolo de **ataques en la red**, como puede ser el problema de "*man in the middle*" u hombre en el medio, que consiste en que un tercero se dedique a **esniffar el tráfico** de la red de comunicaciones, pudiendo **acceder** a información confidencial. (esniffar en red es interceptar y examinar).

2. Protocolo TLS:

   "*Transport Layer Security*" o Seguridad de la Capa de Transporte: Este protocolo surgió como una **evolución del protocolo SSL, proporcionando la posibilidad de utilizar muchos** más algoritmos criptográficos para codificar la información enviada en las comunicaciones.

Los protocolos SSL y TLS son unos protocolos criptográficos que podemos encontrar entre las **capas de aplicación y de transporte del modelo TCP/IP**, gracias a esto, vamos a poder utilizarlos para realizar cifrados de información **en protocolos como Telnet, IMAP, FTP, HTTP**...

Siempre que un protocolo de encriptación como SSL o TLS sea **ejecutado sobre un protocolo** de comunicación, **obtendremos la versión segura del mismo**:

- **FTPS**: Versión segura del protocolo FTP.
- **HTTPS**: Versión segura del protocolo HTTP.
- **SSH**: Versión segura del protocolo **Telnet**.

Según esto, podemos afirmar que, por ejemplo, el protocolo FTPS no es que sea más importante que el protocolo FTP, pero al estar ejecutándose en él un protocolo de criptografía, ya es un protocolo seguro.

Modelo TCP/IP

CAPA	Objetos de transmisión
APLICACIÓN	Mensajes
TRANSPORTE	Paquetes
RED (INTERNET)	Datagramas
ENLACE	Tramas
FÍSICA	Bits

Características SSL/TLS

El protocolo SSL (**Secure Sockets Layer**)

Fue creado por la empresa **Netscape** en un afán de hacer seguras las comunicaciones **entre los navegadores web y los servidores**, aunque se podía, y se puede, utilizar en cualquier aplicación con esquema cliente/servidor.

El protocolo SSL nos va a proporcionar las **propiedades que hacen seguras** a las comunicaciones. Estamos hablando de:

- **Autenticación**.
- **Confidencialidad**.
- **Integridad**.

El funcionamiento del protocolo SSL consiste en que antes de poder tener una comunicación segura entre cliente y servidor, deben de **"*negociarse*" una serie de condiciones o parámetros** para dicha comunicación, esto se conoce como apretón de manos o handshake, conocido como el SSL/TLS Handshake Protocol.

También existe la versión del llamado SSL/TLS Record Protocol, mediante el cual se van a especificar de qué **forma** se van a **encapsular los datos** que serán transmitidos, pudiéndose **incluso negociar los datos de la propia negociación previa**.

Las **fases que se utilizan en el protocolo SSL** son las siguientes:

1. Fase **inicial**: se negocian los **algoritmos** criptográficos que se van a utilizar en la comunicación.

2. Fase de **autenticación**: se intercambiarán las **claves** y se autenticarán las partes mediante certificados de **criptografía asimétrica**. En esta fase es donde se van a **crear las claves** necesarias para realizar la transmisión de información.

3. En la última fase se hará una **verificación** de qué el **canal es seguro** para la comunicación.

4. En este punto ya comenzaría la comunicación segura de información.

Si por cualquier motivo **fallase la negociación**, la **comunicación no** llegaría a establecerse.

Podemos utilizar los siguientes algoritmos criptográficos para ser utilizados con SSL/TLS:

- Algoritmos de clave simétrica: **IDEA, DES**...
- Algoritmos de clave pública: RSA.
- Certificados digitales: RSA.
- Resúmenes: **MD5, SHA**...

Razones del éxito de SLT/TLS

El éxito del protocolo SSL/TLS se debe fundamentalmente a la expansión que ha tenido el **comercio electrónico** en Internet, aunque también es utilizado para poder **crear redes privadas virtuales** o **VPN**.

Consejos para máxima seguridad

La idea de los protocolos es encapsular la información de la mejor forma posible para dar una mayor seguridad a los datos, así que, si éstos siguen esta filosofía, **no sería muy recomendable el uso de multitud protocolos seguros**, ya que **por sí solos suelen ser bastante efectivos**, y lo único que estaríamos consiguiendo es ralentizar la comunicación con todas las operaciones de encriptado extra.

Esto no quiere decir que no se pueda llevar a cabo dicha combinación de protocolos para obtener la máxima seguridad. Pueden existir determinados entornos para los que la información sea **extremadamente sensible**, por lo que, en estos casos, **se podría** llevar a cabo una **combinación de varios protocolos**, aunque sea en **detrimento del tiempo** que se necesitaría para llevar a cabo las comunicaciones de los datos.

Por otra parte, existen **opciones sencillas de combinación de protocolos** que no conllevarían mucha ralentización en las comunicaciones, y aportarían un plus de seguridad a nuestros proyectos. Un ejemplo de esto podría ser el uso de comunicaciones seguras vía **SSL/TLS** y los **datos** que se transmitan en dichas comunicaciones que puedan estar encriptados con **protocolos de seguridad** como puede ser un cifrado con un **certificado digital**.

Encriptación de datos con Cipher

El lenguaje de programación **Java** nos proporciona la clase Cipher, mediante la cual vamos a poder realizar codificaciones de datos.

Uno de los **algoritmos de encriptado** que vamos a poder utilizar es el cifrado AES, no obstante, se admiten muchos modos de operación más.

El algoritmo de cifrado AES, ***Advanced Encryption Standard*** por sus siglas en inglés, es un **cifrado de esquema por bloques** que se comenzó a utilizar en Estados Unidos. AES posee un **tamaño de bloque fijo** de **128 bits** y **tamaños de clave** de **128, 192 o 256 bits**. La mayoría de los **cálculos** del algoritmo AES se hacen en un **campo finito determinado**.

Para poder utilizar AES deberemos descargar e integrar la **biblioteca** de ***commons-codec*** de Apache, la cual podemos descargar desde el siguiente enlace:

http://commons.apache.org/proper/commons-codec/download_codec.cgi

Descargamos los binarios y nos quedamos con el fichero **commons-codec-x.yy.jar**.

Siendo **x e yy** las **versiones actuales** del paquete.

Una vez descargada la **integramos en nuestro proyecto** NetBeans.

Para encriptar un mensaje con una clave mediante el algoritmo AES podemos usar el código:

Cifrado AES

```
public static void main(String[] args)
{
	String key = "92AE31A79FEEB2A3"; // llave
	String iv = "0123456789ABCDEF";
	String mensaje = "Hola mundo";
	try {
		// Proceso de encriptado con AES
		Cipher cipher = Cipher.getInstance("AES/CBC/PKCSSPadding");
		SecretKeySpec skeySpec = new SecretKeySpec(
			key.getBytes(), "AES");
		IvParameterSpec ivParameterSpec = new
		IvParameterSpec(iv.getBytes());
		cipher.init(Cipher.ENCRYPT_MODE, skeySpec, ivParameterSpec);
```

```
            byte[] encrypted = cipher.doFinal(mensaje.getBytes());
            System.out.printin("El mensaje encriptado es: " +
            new String (encodeBaseé4 (encrypted) ));
        }
        catch (InvalidAlgorithmParameterException | InvalidKeyException |
        NoSuchAlgorithmException | BadPaddingException |
        IllegalBlockSizeException | NoSuchPaddingException error)
        {
        System.out.printin("Error");
        }
    }
```

Clases para implementar una firma digital en Java

El lenguaje de programación Java nos proporciona la clase **Signature**, del paquete java.security, que nos va a permitir realizar una implementación de firma digital, además de verificarla.

Vamos a ver los pasos necesarios para poder realizar este proceso.

1. Generamos las claves publicas y privadas mediante la clase **KeyPairGenerator**. Utilizaremos la clase **PrivateKey** para firmar y la clase **PublicKey** para verificar la firma.
2. Realizaremos la firma digital mediante la clase **Signature**, utilizando además un algoritmo de clave asimétrica, como puede ser el **DSA**. Utilizaremos los métodos initSign() para generar la firma, **update()** para crear el resumen del mensaje y **sign()** que terminara de crear la firma digital.
3. Verificaremos la firma mediante la clave publica. Usaremos el método **initVerify()**, al cual le pasaremos la clave publica, luego con el método **update()** actualizaremos el resumen del mensaje para comprobar si coincide con el enviado y con el método **verify()** realizaremos la verificación de la firma.

```
import java.security.*;

public class DigitalSignatureExample {
    public static void main(String[] args) {
        try {
```

```
            // Paso 1: Generar las claves pública y privada
            KeyPairGenerator keyPairGen =
                  KeyPairGenerator.getInstance("DSA");
            keyPairGen.initialize(1024);
            KeyPair keyPair = keyPairGen.generateKeyPair();
            PrivateKey privateKey = keyPair.getPrivate();
            PublicKey publicKey = keyPair.getPublic();

            // Paso 2: Realizar la firma digital
            Signature signature = Signature.getInstance("SHA256withDSA");
            signature.initSign(privateKey);
            String message = "Mensaje a firmar";
            signature.update(message.getBytes());
            byte[] digitalSignature = signature.sign();
            System.out.println("Firma digital generada correctamente.");

            // Paso 3: Verificar la firma
            signature.initVerify(publicKey);
            signature.update(message.getBytes());
            boolean verified = signature.verify(digitalSignature);
            if (verified) {
                System.out.println("La firma digital es válida.");
            } else {
                System.out.println("La firma digital no es válida.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código genera un par de claves pública y privada, firma un mensaje utilizando la clave privada y luego verifica la firma utilizando la clave pública.

Asegúrate de manejar adecuadamente las excepciones en un entorno de producción. Además, ten en cuenta que el algoritmo DSA utilizado aquí es solo un ejemplo.

En aplicaciones reales, es posible que desees utilizar algoritmos más robustos, como RSA o ECDSA.

Certificados para sockets seguros

El lenguaje de programación Java nos permite crear una versión segura de los sockets que utilizamos en unidades anteriores.

Si recordamos, utilizamos los sockets para establecer una comunicación en red entre las dos partes del modelo cliente/servidor, pero los sockets que utilizamos anteriormente no eran seguros, ya que no cifraban la información.

Para poder utilizar la versión segura de los sockets, primeramente, deberemos **crear los certificados** que nos ayudarán a encriptar la información.

Para crear el certificado del servidor necesitamos abrir una ventana de comandos (cmd de windows o bash de linux) y ejecutar el siguiente comando (Chudiang, 2016):

keytool -genkey -keyalg RSA -alias serverKay -keystore serverKey.jks -storepass servpass

Cuando ejecutemos el comando, el sistema nos irá **solicitando una serie de datos**, como nuestro nombre, apellidos, etc. Un certificado va a ir **asociado a alguna persona** u organización, es por este motivo por lo que en este punto nos solicita esos datos.

Cuando tengamos el certificado del servidor, debemos generar el certificado para el cliente. Como el certificado del servidor está **dentro de un almacén**, tenemos que sacarlo de ahí **a un fichero**. El *comando para generar el certificado del cliente* que debemos usar es:

keytool -export -keystore serverKey.jks -alias serverKay -file ServerPublicKey.cer

Repetimos todo el proceso para generar los **ficheros del cliente**. Creamos el certificado del cliente en un **almacén de certificados clientKey.jks**:

keytool -genkey -keyalg RSA -alias clientKay -keystore clientKey.jks -storepass clientpass

Para finalizar metemos la clave pública del cliente en los **certificados de confianza del servidor**:

keytool -import -v -trustcacerts -alias clientKay -file ClientPublicKey.cer -keystore serverTrustedCerts.jks -keypass servpass -storepass servpass

Sockets seguros

Servidor

Vamos a ver cómo podemos implementar un **servidor seguro** en Java. La forma fácil de crear los sockets SSL es usar las factorías de socket SSL por defecto que nos proporciona java. Para el lado del servidor, el código sería (Chudiang, 2016):

```
SSLServerSocketFactory serverFactory =
      (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
ServerSocket serverSocket =
      serverFactory.createServerSocket(puerto);
```

La única pregunta que podemos hacernos en este punto es ¿cómo podemos indicar dónde se encuentran los almacenes de certificados y certificados de confianza que generamos en el paso anterior? Esto lo vamos a poder hacer mediante las **propiedades de System**, o bien con **opciones del parámetro -D** al arrancar nuestra aplicación java.

Las propiedades a fijar son:

- javax.net.ssl.keyStore: Con el que indicamos el **almacén** donde está el certificado que **nos identifica**.
- javax.net.ssl.keyStorePassword: Con el que indicamos la **clave** para **acceder** a dicho **almacén** y para acceder al **certificado** dentro del almacén. ¡¡¡OJO!!
- javax.net.ssl.trustStore: Con el que indicamos el **almacén** donde están los certificados **en los que se confía**.
- javax.net.ssl.trustStorePassword: Con el que indicamos la **clave para acceder** a dicho almacén y a los certificados dentro del almacén (en los que se confía). ¡¡¡OJO!!!

Si decidimos hacerlo mediante System.setProperty(), el código quedaría así:

```
System.setProperty(
      “javax.net.ssl.keyStore”,
      “src/main/certificados/servidor/serverKey.jks”);
System.setProperty(
      “javax.net.ssl.keyStorePassword”,
      ”servpass”);
System.setProperty(
```

```
        “javax.net.ssl.trustStore”,
        “src/main/certificados/servidor/serverTrustedCerts.jks”);
System.setProperty(
        “javax.net.ssl.trustStorePassword”,
        “servpass”);
```

Sockets seguros

Cliente

Vamos a ver cómo podemos implementar un cliente seguro en Java. Al igual que con el servidor, en el cliente la forma fácil de crear los sockets SSL es usar las factorías de socket SSL por defecto que nos proporciona java. El código en este caso sería:

```
SSLSocketFactory clientFactory = (
      SSLSocketFactory) SSLSocketFactory.getDefault();
Socket cliente = clientFactory.createSocket(
      servidorseguro, puerto);
```

Donde mediante la variable **servidorseguro** y **puerto** indicamos cuáles serán la dirección **IP** y el **puerto**, donde está el servidor seguro que hemos creado anteriormente.

Para trabajar con los clientes a partir de este punto, se deberá obtener un socket de la forma habitual.

¿Qué inconveniente podemos encontrar en este mecanismo? Hay que remarcar que todas las variables de configuración que hemos visto anteriormente en la parte de creación del servidor seguro van a afectar a todo el programa Java.

A partir de ese momento, todos los **sockets** que abramos tendrán el **mismo certificado** y confiarán en los mismos certificados. Si quisiésemos poder establecer varios sockets con distintos certificados y distintos certificados de confianza, necesitamos una configuración más compleja.

Una vez que ya tengamos creados nuestro servidor y nuestros clientes seguros, la forma de trabajar va a ser exactamente la misma que la que utilizamos en unidades anteriores en comunicaciones en red. A continuación, veremos un ejemplo de aplicación de sockets seguros:

Ejemplo de aplicación de sockets seguros

Vamos a ver un ejemplo de Socket seguros en Java. Para este ejemplo hemos creado dos clases: 'SSLCliente', que representará un cliente seguro, y 'SSLServidor', que representará un servidor seguro.

Comenzamos por el servidor seguro. En este servidor hemos creado una **variable** del tipo '**SSLServerSocket**' que será el socket seguro para implementar nuestro servidor. En primer lugar, en el constructor de la clase le hemos **pasado el puerto** al que nos conectaremos.

Lo primero que debemos hacer es crear donde están los certificados para nuestro servidor mediante estas instrucciones.

Una vez hecho eso, deberemos obtener las claves de confianza que estarán también en la misma carpeta de los certificados en nuestro caso.

Una vez hecho eso, configuraremos nuestro contexto seguro y nos crearemos un server socket con el puerto que hemos indicado. Una vez hecho esto, nuestro servidor seguro estará creado con las claves de confianza que hemos configurado.

Hemos creado un método 'start' en el que lanzaremos una **hebra** que estará escuchando clientes seguros. Una vez que conecte con un cliente, obtendrá sus buffer de lectura para poder escribir y leer.

SSLservidorSeguro.java

```
public class SSLservidor {

    private final String SERVERPASS = "servpass";
    private final SSLServerSocket serverSocket;
    private final int puerto;

    public SSLservidor(int puerto) throws FileNotFoundException,
            KeyStoreException, IOException, NoSuchAlgorithmException,
            CertificateException, UnrecoverableKeyException,
            KeyManagementException
    {
        this.puerto = puerto;

        // Indico los certificados seguros del servidor
        KeyStore keyStore = KeyStore.getInstance("JKS");
        keyStore.load(new FileInputStream(
                "src" + File.separator
                + "certificados" + File.separator
                + "servidor" + File.separator
                + "serverKey.jks"), SERVERPASS.toCharArray());
```

```
    KeyManagerFactory kmf = KeyManagerFactory.
              getInstance(KeyManagerFactory.
                          getDefaultAlgorithm());
    kmf.init(keyStore, SERVERPASS.toCharArray());

    KeyStore trustedStore = KeyStore.getInstance("JKS");
    trustedStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "servidor" + File.separator
            + "serverTrustedCerts.jks"), SERVERPASS
            .toCharArray());
    TrustManagerFactory tmf =
              TrustManagerFactory.
              getInstance(TrustManagerFactory.
                          getDefaultAlgorithm());

    tmf.init(trustedStore);
    SSLContext sc = SSLContext.getInstance("TLS");
    TrustManager[] trustManagers = tmf.getTrustManagers();
    KeyManager[] keyManagers = kmf.getKeyManagers();
    sc.init(keyManagers, trustManagers, null);

    // Creo el socket seguro del servidor
    SSLServerSocketFactory ssf = sc.getServerSocketFactory();
    serverSocket =
              (SSLServerSocket) ssf.createServerSocket(puerto);
}

public void start() {
    System.out.println("Servidor escuchando en el puerto " +
puerto);

    new Thread() {
        @Override
        public void run() {
            try {
                Socket aClient = serverSocket.accept();
                System.out.println("Cliente aceptado");
                aClient.setSoLinger(true, 1000);
                BufferedReader input = new BufferedReader(
                          new InputStreamReader(
                                      aClient.getInputStream()));
                String recibido = input.readLine();
                System.out.println("Recibido " + recibido);
```

```
                    PrintWriter output = new PrintWriter(
                              aClient.getOutputStream());
                    output.println("Hola, " + recibido);
                    output.flush();
                    aClient.close();
                }
                catch (IOException e) {
                    System.out.println("Error servidor → " +
                e.toString());
                }
            }
        }.start();
    }

}
```

En el cliente tenemos un proceso muy similar.

Aquí hemos creado una variable del tipo '**SSLSocket**' que será nuestro socket seguro.

En el constructor de nuestra clase del cliente seguro tenemos la **dirección del servidor** al que nos conectaremos y el **puerto** al que nos conectaremos.

Ahora, de igual forma que hicimos en el servidor, debemos obtener dónde están los certificados y las claves de confianza. En este caso, para nuestro cliente seguro.

Una vez hecho eso, creamos un cliente mediante el método 'createSocket'.

También hemos creado un método 'start' que será el que **inicie** la funcionalidad del **cliente**. En este método, conectaremos con un servidor seguro, obtendremos sus flujos para poder comunicarnos con él y le enviaremos un mensaje y recibiremos un mensaje.

SSLcliente.java

```
public class SSLcliente {

    private final String CLIENTPASS = "clientpass";
    private final SSLSocket client;

    public SSLcliente(
            String direccionservidor, int puerto) throws
            KeyStoreException, FileNotFoundException, IOException,
            NoSuchAlgorithmException, CertificateException,
            UnrecoverableKeyException, KeyManagementException
```

```
{
    // Indico los certificados seguros del cliente
    KeyStore keyStore = KeyStore.getInstance("JKS");
    keyStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "cliente" + File.separator
            + "clientKey.jks"),
            CLIENTPASS.toCharArray());

    KeyManagerFactory kmf =
              KeyManagerFactory.getInstance(
                          KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(keyStore, CLIENTPASS.toCharArray());
    KeyStore trustedStore = KeyStore.getInstance("JKS");
    trustedStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "cliente" + File.separator
            + "clientTrustedCerts.jks"),
              CLIENTPASS.toCharArray());
    TrustManagerFactory tmf =
              TrustManagerFactory.getInstance(
                          TrustManagerFactory.getDefaultAlgorithm());

    tmf.init(trustedStore);
    SSLContext sc = SSLContext.getInstance("TLS");
    TrustManager[] trustManagers = tmf.getTrustManagers();
    KeyManager[] keyManagers = kmf.getKeyManagers();
    sc.init(keyManagers, trustManagers, null);

    // Creo el socket seguro del cliente
    SSLSocketFactory ssf = sc.getSocketFactory();
    client = (SSLSocket) ssf.createSocket(direccionservidor,
            puerto);
    client.startHandshake();
}

public void start() {
    System.out.println("Inicio cliente");
    new Thread() {
        @Override
        public void run() {
            try {
                Random aleatorio = new Random();
```

```
                    PrintWriter output = new PrintWriter(
                              client.getOutputStream());
                    output.println("Cliente: " +
                              aleatorio.nextInt(100));
                    output.flush();
                    BufferedReader input =
                              new BufferedReader(
                                        new InputStreamReader(

          client.getInputStream()));
                    String received = input.readLine();
                    System.out.println("Recibido: " +
                    received);
                    client.close();
               }
               catch (IOException e) {
                    System.out.println("Error cliente → " +
               e.toString());
               }
          }
        }.start();
    }
}
```

Para poder probarlo, deberemos tener un método 'main' en el que indiquemos el **puerto** en el que nos vamos a conectar, por ejemplo, el 5557.

Nos creamos un servidor y luego ya podremos crear tantos clientes como queramos, siendo esta comunicación una comunicación segura.

clase main

```
public static void main(String[] args) {
      int puerto = 5557;
      try
      {
          // Creo el servidor
          new SSLservidor(puerto).start();
          // Creo un cliente
          new SSLcliente("localhost", puerto).start();
      }
      catch (IOException | KeyManagementException | KeyStoreException
              | NoSuchAlgorithmException | UnrecoverableKeyException
              | CertificateException error)
      {
          System.out.println("Error --> " + error.toString());
      }
  }
```

Forma correcta de realizar una auditoría

Si se ha producido una **filtración de información** en una comunicación, por muy pequeña que sea, puede significar que ha habido un atacante esnifando el tráfico de la aplicación.

Deberíamos empezar a **comprobar** si los datos que utiliza la aplicación **están cifrados** y en caso de que no lo estén deberán implementar un **sistema de cifrado** de los mismos, por ejemplo, mediante un algoritmo de **criptografía asimétrica**, para así poder **decodificar fácilmente la información cuando llegue a su destino**.

Otro punto muy importante en el que deberían detenerse, ya que **hay comunicaciones en red** en la aplicación, será el de **comprobar si los sockets** que se utilizan para implementar dichas comunicaciones **son seguros** o no.

El hecho de usar sockets seguros puede significar la diferencia entre que un **atacante pueda o no obtener la información** que se está transmitiendo.

Como punto final, todas las comunicaciones deberían hacerse usando los sockets seguros, pero enviando la información codificada, para así tener un doble nivel de seguridad.

Ejemplo de acceso seguro a app web

El trabajo consiste en el desarrollo de una **aplicación web**, a la que únicamente se permite el **acceso de forma segura**, es decir, mediante la identificación de los usuarios.

Podemos creer que basta con que no nos aparezca el típico candado amarillo en la barra de direcciones web cuando navegamos, pero tenemos que tener cuidado, ya que podemos estar siendo víctimas de un atacante que oculta esa información, así que, lo más seguro es comprobar que en nuestra barra de direcciones web utilizamos el **protocolo HTTPS**, y con eso ya tendremos una navegación segura.

Reflexión

Hemos comprobado que, en una transmisión de información, deberemos **siempre encriptar las comunicaciones** para así poder hacerlas seguras, utilizando una serie de protocolos seguros.

Hemos visto cuáles son esos protocolos seguros, y sobre todo hemos hecho hincapié en uno de los más importantes, el protocolo SSL/TLS, que es el responsable de que las comunicaciones sean seguras.

También hemos aprendido cómo podemos **encriptar información** de forma segura, utilizando la clase Cipher para ello.

Por último, hemos practicado sobre cómo podemos **realizar una comunicación** en red de forma segura, utilizando para ello los sockets seguros, tanto sockets para crear el servidor, como para crear el cliente, garantizando la seguridad de la comunicación en ambos extremos.