**RESUMEN DE TEMARIO PSP**

PSP 4

**1.- PARADIGMA CLIENTE/SERVIDOR.**

El modelo Cliente/Servidor en comunicaciones entre equipos:

* Paradigma principal en comunicaciones.
* Cliente solicita servicios al servidor.
* Arquitectura distribuida para acceso a recursos.
* **Cliente**: Interactúa, envía solicitudes, muestra resultados.
* **Servidor:** Procesa solicitudes, gestiona lógica, asegura datos.
* **Cliente: Interfaz usuario. Servidor:** Administración datos y seguridad.
* **División modular:** Facilita desarrollo y mantenimiento.

# 1.1.- Características básicas.

Características básicas de la arquitectura Cliente/Servidor:

* **Cliente interactúa con usuario, servidor con recursos compartidos**.
* **Procesamiento:** Servidor realiza trabajo, cliente interactúa.
* **Relación entre procesos en uno o varios equipos.**
* **Distinción de funciones basada en "servicio".**
* **Relación muchos a uno:** Servidor atiende múltiples clientes.
* **Clientes activos:** Solicitan servicios. Servidores pasivos: Esperan peticiones.
* **Comunicación mediante intercambio de mensajes.**
* **Soporte para sistemas heterogéneos**: Conexión independiente de plataformas.

# 1.2.- Ventajas y desventajas.

Ventajas del esquema Cliente/Servidor:

* **Utilización de clientes ligeros:** El servidor realiza procesamiento.
* **Integración entre sistemas y interfaces amigables.**
* **Uso de interfaces gráficas interactivas.**
* **Mantenimiento y desarrollo rápido de aplicaciones.**
* **Estructura modular para integración de nuevas tecnologías.**
* **Soluciones locales con integración de información global.**
* **Acceso centralizado a recursos.**
* **Acceso simultáneo a datos entre clientes.**

Desventajas del esquema Cliente/Servidor:

* **Mantenimiento complejo**: Interacción entre partes de hardware y software.
* **Manejo de errores y seguridad del sistema.**
* **Garantizar consistencia de la información.**
* **Necesidad de sincronización para evitar conflictos de datos entre clientes.**

# 1.3.- Modelos.

Clasificación de modelos Cliente/Servidor por número de capas:

1. **1 capa (1-tier):** Cliente/servidor en mismo equipo, sin comunicaciones por red.
2. **2 capas (2-tiers):** Modelo tradicional con servidor y clientes distintos. Puede tener problemas de escalabilidad.
3. **3 capas (3-tiers):** Se agrega servidor de aplicación entre clientes y servidor de datos para mejorar rendimiento. Permite añadir servidores de datos adicionales.
4. **n capas (n-tiers):** Se pueden añadir capas adicionales de servidores para separar funcionalidades y mejorar rendimiento del sistema.

# 1.4.- Programación.

Pasos internos del servidor para la comunicación:

1. **Publicar puerto:** Se anuncia el puerto para recibir conexiones.
2. **Esperar peticiones:** El servidor aguarda conexiones entrantes.
3. **Envío y recepción de datos:** Se establecen flujos de entrada y salida para la comunicación. El servidor procesa las solicitudes y envía respuestas.
4. **Cerrar socket del cliente:** Al terminar la comunicación, se cierra el socket del cliente.

Pasos del cliente para la comunicación:

1. **Conectarse con el servidor:** El cliente establece conexión con un servidor específico en un puerto determinado, creando un socket.
2. **Envío y recepción de datos:** Se configuran flujos de entrada y salida para la comunicación de datos.
3. **Cerrar socket:** Al concluir la comunicación, se cierra el socket.

# 1.5.- Ejemplo I.

**servidor.java**

**import java.io.\* ;**

**import java.net.\* ;**

**class Servidor {**

**static final int Puerto=2000;**

**public Servidor( ) {**

**try {**

**ServerSocket sServidor = new ServerSocket(Puerto);**

**System.out.println("Escucho el puerto " + Puerto );**

**for ( int nCli = 0; nCli < 3; nCli++) {**

**Socket sCliente = sServidor.accept();**

**System.out.println("Sirvo al cliente " + nCli);**

**DataOutputStream flujo\_salida= new DataOutputStream(sCliente.getOutputStream());**

**flujo\_salida.writeUTF( "Hola cliente " + nCli );**

**sCliente.close();**

**}**

**System.out.println("Se han atendido los clientes");**

**} catch( Exception e ) {**

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

**}**

**public static void main( String[] arg ) {**

**new Servidor();}**

**}**

# 2.- Optimización de sockets.

Al utilizar sockets, es esencial optimizar su rendimiento y garantizar la seguridad del sistema. Para una aplicación cliente/servidor eficiente, se deben tener en cuenta las siguientes características:

1. **Atender múltiples peticiones simultáneamente:** El servidor debe ser capaz de gestionar varias solicitudes de clientes al mismo tiempo para acceder a los recursos o servicios ofrecidos.
2. **Seguridad:** Para proteger el sistema, el servidor debe prevenir la pérdida de información, filtrar las peticiones de los clientes para asegurar su validez, y llevar un registro de las transacciones de los clientes.
3. **Monitorización de tiempos de respuesta:** Es necesario implementar mecanismos para supervisar los tiempos de respuesta de los clientes y analizar el comportamiento del sistema.Principio del formulario

Final del formulario

# 2.1.- Atender múltiples peticiones simultáneas.

La implementación de un servidor concurrente en el modelo cliente/servidor implica la creación de múltiples hilos de ejecución para atender a varios clientes simultáneamente. Esto se logra mediante un bucle que espera nuevas conexiones y crea un hilo para cada cliente conectado. El esquema de funcionamiento interno se representa gráficamente, mostrando los módulos y su interconexión. El código básico para lograr esto es:

**while(true) {**

**// Esperar conexión del cliente**

**Socket skCliente = skServidor.accept();**

**System.out.println("Cliente conectado");**

**// Atender al cliente mediante un hilo (thread)**

**new Servidor(skCliente).start();**

**}**

Este enfoque permite gestionar múltiples clientes de forma concurrente y mantener la independencia de la atención a cada uno de ellos.

# 2.2.- Threads.

Para crear una hay que definir la clase que extienda de Threads:

**class Servidor extends Thread{**

**public Servidor() {**

**// Inicialización de la hebra }**

**public static void main( String[] arg ) {**

**new Servidor().start();**

**}**

**public void run(){**

**//tareas que realiza la hebra }**

**}**

donde:

La función public Servidor permite inicializar los valores iniciales que recibe la hebra.

* La función run() es la encargada de realizar las tareas de la hebra.
* Para iniciar la hebra se crea el objeto Servidor y se inicia: new Servidor().start ();

# 2.3.- Ejemplo II.

**Servidor.java**

**import java.io.\* ;**

**import java.net.\* ;**

**class Servidor extends Thread{**

**Socket skCliente;**

**static final int Puerto=2000;**

**public Servidor(Socket sCliente) {**

**skCliente=sCliente;**

**}**

**public static void main( String[] arg ) {**

**try{**

**// Inicio el servidor en el puerto**

**ServerSocket skServidor = new ServerSocket(Puerto);**

**System.out.println("Escucho el puerto " + Puerto );**

**while(true){**

**// Se conecta un cliente**

**Socket skCliente = skServidor.accept();**

**System.out.println("Cliente conectado");**

**// Atiendo al cliente mediante un thread**

**new Servidor(skCliente).start();**

**}**

**} catch (Exception e) {;}**

**}**

**public void run(){**

**try {**

**// Creo los flujos de entrada y salida**

**DataInputStream flujo\_entrada = new DataInputStream(skCliente.getInputStream());**

**DataOutputStream flujo\_salida= new DataOutputStream(skCliente.getOutputStream());**

**// ATENDER PETICIÓN DEL CLIENTE**

**flujo\_salida.writeUTF(“Se ha conectado el cliente de forma correcta”);**

**// Se cierra la conexión**

**skCliente.close();**

**System.out.println("Cliente desconectado");**

**} catch( Exception e ) {**

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

**}**

**}**

**}**

# 2.3.1.- Ejemplo II (II).

**Cliente.java**

**import java.io.\*;**

**import java.net.\*;**

**class Cliente {**

**static final String HOST = "localhost";**

**static final int Puerto=2000;**

**public Cliente( ) {**

**try{**

**Socket sCliente = new Socket( HOST , Puerto );**

**// Creo los flujos de entrada y salida**

**DataInputStream flujo\_entrada = new DataInputStream(skCliente.getInputStream());**

**DataOutputStream flujo\_salida= new DataOutputStream(skCliente.getOutputStream());**

**// TAREAS QUE REALIZA EL CLIENTE**

**String datos=flujo\_entrada.readUTF();**

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

**sCliente.close();**

**} catch( Exception e ) {**

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

**}**

**}**

**public static void main( String[] arg ) {**

**new Cliente();**

**}**

**}**

# 2.4.- Pérdida de información.

La pérdida de paquetes en las comunicaciones de red puede afectar la integridad de los datos transmitidos, como en el caso de enviar un archivo donde la pérdida de un solo paquete puede resultar en una transmisión incorrecta.

Para evitar la pérdida de paquetes, se emplea un método de confirmación mediante paquetes ACK (acknowledgement) enviados por el receptor al emisor. Si un paquete no llega correctamente al receptor, este no envía el ACK, y el emisor reenvía el paquete después de un tiempo de espera.

Para mejorar la eficiencia de este método, se permite al emisor enviar múltiples paquetes antes de esperar los ACK correspondientes. Esto reduce el tiempo de espera y mejora la velocidad de transmisión.

Para gestionar la confirmación de paquetes, se utiliza un vector que indica qué paquetes han sido enviados correctamente. Este vector se desplaza a la izquierda hasta llegar al primer paquete no confirmado, permitiendo al emisor enviar más paquetes mientras espera la confirmación de los anteriores.

El tamaño del vector de ACK influye en el rendimiento del sistema, ya que determina cuántos paquetes pueden enviarse simultáneamente. Sin embargo, existe una limitación en la cantidad de memoria RAM que puede ocupar el vector.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vector de ACK (estado inicial) | | | | | | | | | | |
| **Mensaje** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **ACK** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vector de ACK (desplazado) | | | | | | | | | | |
| **Mensaje** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **ACK** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

# 2.5.- Transacciones.

Principales fallos de seguridad en programas cliente/servidor:

1. **Operaciones no autorizadas:** Cliente intenta realizar acciones a las que no tiene acceso.
2. **Mensajes mal formados:** Cliente envía datos incorrectos que pueden causar errores en el servidor.

Para evitar estos problemas, es crucial modelar el flujo de información y comportamiento del servidor con un diagrama de estados o autómata. Ejemplo:

* **Estado 0:** Inicio. Se solicita al cliente introducir un comando.
* **Estado 1:** Espera de comando. El cliente puede enviar "ls", "get" o "exit".
* **Estado 2:** Comando "ls". Muestra contenido del directorio.
* **Estado 3:** Comando "get". Solicita nombre del archivo.
* **Estado 4:** Muestra contenido del archivo.
* **Estado -1:** Finaliza la conexión del cliente al recibir comando "exit".

Se utilizan variables de estado y comando para seguir el comportamiento del autómata. En casos simples se usan estructuras de control, y en casos complejos, tablas de transición de estados.

# 2.6.- Ejemplo III.

A continuación, a modo de ejemplo se muestra la estructura general para implementar el diagrama de transiciones del ejemplo anterior.

**int estado=1**

**do{**

**switch(estado){**

**case 1:**

**flujo\_salida.writeUTF("Introduce comando (ls/get/exit)");**

**comando=flujo\_entrada.readUTF();**

**if(comando.equals("ls")){**

**System.out.println("\tEl cliente quiere ver el contenido del directorio");**

**// Muestro el directorio**

**estado=1;**

**break;**

**}else**

**if(comando.equals("get")){**

**// Voy al estado 3 para mostrar el fichero**

**estado=3;**

**break;**

**}else**

**estado=1;**

**break;**

**case 3://voy a mostrar el archivo**

**flujo\_salida.writeUTF("Introduce el nombre del archivo");**

**String fichero =flujo\_entrada.readUTF();**

**// Muestor el fichero**

**estado=1;**

**break;**

**}**

**if(comando.equals("exit")) estado=-1;**

**}while(estado!=-1);**

# 2.7.- Monitorizar tiempos de respuesta.

Para medir el tiempo de procesamiento de una solicitud del cliente en milisegundos, puedes utilizar el siguiente código en Java:

**import java.util.Date;**

**long tiempoInicio = (new Date()).getTime();**

**// Procesar la petición del cliente**

**long tiempoFin = (new Date()).getTime();**

**System.out.println("Tiempo de procesamiento: " + (tiempoFin - tiempoInicio) + " ms");**

Este código mide el tiempo que transcurre desde el inicio hasta la finalización del procesamiento de la solicitud del cliente.

Para medir el tiempo de transmisión, necesitas sincronizar los relojes del sistema en los equipos cliente y servidor utilizando el servicio NTP (Network Time Protocol). En Windows, la sincronización ocurre automáticamente, mientras que en GNU/Linux puedes usar el siguiente comando:

**/usr/sbin/ntpdate -u 0.centos.pool.ntp.org**

Otra forma de calcular el tiempo de transmisión es utilizar el comando ping. Por ejemplo, el tiempo promedio para que un cliente acceda al servidor [www.google.es](http://www.google.es) puede ser alrededor de 115 milisegundos, como se muestra en la figura del resultado de un ping.

# 2.8.- Ejemplo IV.

Ejemplo en el que se calcula el tiempo de trasmisión de datos entre una aplicación Cliente y Servidor. Para ello, el servidor le va a enviar al cliente un mensaje con el tiempo del sistema en milisegundos y el cliente cuando reciba el mensaje calculará la diferencia entre el tiempo de su sistema y el del mensaje.  
**Servidor.java**

**import java.io.\* ;**

**import java.net.\* ;**

**import java.util.Date;**

**class Servidor {**

**static final int Puerto=2000;**

**public Servidor( ) {**

**try {**

**// Inicio el servidor en el puerto**

**ServerSocket sServidor = new ServerSocket(Puerto);**

**System.out.println("Escucho el puerto " + Puerto );**

**// Se conecta un cliente**

**Socket sCliente = sServidor.accept(); // Crea objeto**

**System.out.println("Cliente conectado");**

**// Creo los flujos de entrada y salida**

**DataInputStream flujo\_entrada = new DataInputStream( sCliente.getInputStream());**

**DataOutputStream flujo\_salida= new DataOutputStream(sCliente.getOutputStream());**

**// CUERPO DEL ALGORITMO**

**long tiempo1=(new Date()).getTime();**

**flujo\_salida.writeUTF(Long.toString(tiempo1));**

**// Se cierra la conexión**

**sCliente.close();**

**System.out.println("Cliente desconectado");**

**} catch( Exception e ) {**

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

**}**

**}**

**public static void main( String[] arg ) {**

**new Servidor();}**

**}**

**Cliente.java**

**import java.io.\*;**

**import java.net.\*;**

**import java.util.Date;**

**class Cliente {**

**static final String HOST = "localhost";**

**static final int Puerto=2000;**

**public Cliente( ) {**

**String datos=new String();**

**String num\_cliente=new String();**

**// para leer del teclado**

**BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));**

**try{**

**// Me conecto al puerto**

**Socket sCliente = new Socket( HOST , Puerto );**

**// Creo los flujos de entrada y salida**

**DataInputStream flujo\_entrada = new DataInputStream(sCliente.getInputStream());**

**DataOutputStream flujo\_salida= new DataOutputStream(sCliente.getOutputStream());**

**// CUERPO DEL ALGORITMO**

**datos=flujo\_entrada.readUTF();**

**long tiempo1=Long.valueOf(datos);**

**long tiempo2=(new Date()).getTime();**

**System.out.println("\n El tiempo es:"+(tiempo2-tiempo1));**

**// Se cierra la conexión**

**sCliente.close();**

**} catch( Exception e ) {**

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

**}**

**}**

**public static void main( String[] arg ) {**

**new Cliente();**

**}**

**}**

PSP5

# Generación de servicios en red.

# 1.- Introducción.

Una red informática conecta equipos para compartir información y recursos, mejorando la eficiencia del sistema. Los servicios en red son programas que proporcionan funciones específicas, clasificados según su propósito:

1. Administración/Configuración: DHCP, DNS.
2. Acceso y control remoto: Telnet, SSH.
3. Ficheros: FTP.
4. Impresión: Servicios de impresión remota.
5. Información: HTTP, servidores de bases de datos.
6. Comunicación: Email (SMTP).

Los servicios suelen llamarse por el nombre del protocolo en que se basan

# 2.- Protocolos de comunicaciones del nivel de aplicación.

El modelo TCP/IP es fundamental para la comunicación en redes, tanto en Internet como en redes locales y corporativas. La capa de Aplicación, la más alta en este modelo, incluye protocolos de alto nivel para servicios en red. Algunos protocolos estándar de esta capa son:

1. **FTP**: Transferencia de ficheros.
2. **Telnet**: Acceso remoto a máquinas.
3. **SMTP**: Transferencia de correo electrónico.
4. **HTTP**: Transferencia de hipertexto.
5. **SSH**: Gestión remota segura.
6. **NNTP**: Transferencia de noticias.
7. **IRC**: Chat basado en Internet.
8. **DNS:** Traducción de direcciones de red.

Estos protocolos son ampliamente utilizados y esenciales para diversas funciones de comunicación en red.

# 2.1.- Comunicación entre aplicaciones.

El modelo TCP/IP opera sobre el principio del modelo cliente/servidor, donde:

* El Cliente es el programa que el usuario ejecuta para solicitar un servicio al servidor. Inicia la comunicación.
* El Servidor es el programa que se ejecuta en una máquina de red y ofrece servicios a uno o varios clientes. Permanece a la espera de peticiones y responde proporcionando el servicio solicitado.

En la capa de aplicación, los programas se comunican a través de la red. Los datos de la aplicación se codifican según un protocolo estándar y se pasan al nivel siguiente de la pila TCP/IP.

En el nivel de transporte, las aplicaciones suelen usar TCP o UDP, asociados a un número de puerto (HTTP - puerto 80, FTP - puerto 21, etc.).

Por ejemplo, la World Wide Web utiliza HTTP:

* Los navegadores (clientes) solicitan páginas web y los servidores las sirven.
* El cliente solicita una página mediante HTTP.
* El servidor envía la página al cliente que la solicitó.

Este diálogo entre cliente y servidor sigue un protocolo de aplicación, como HTTP.

**2.2.- Conexión, transmisión y desconexión.**

En Java, los protocolos de aplicación se comunican con el nivel de transporte a través de una API llamada Socket API, implementada mediante las clases del paquete java.net, como Socket y ServerSocket.

Un socket en Java representa una conexión para la transmisión de información entre dos ordenadores o entre un ordenador y sí mismo. Esta abstracción permite a los programadores desentenderse de las capas inferiores.

En una red, un socket se caracteriza por cinco parámetros: el protocolo utilizado, dos direcciones IP (local y remota) y dos puertos (local y remoto).

Los pasos para establecer, mantener y cerrar una conexión TCP/IP son:

1. Creación de sockets en el cliente y el servidor.
2. El servidor establece el puerto de servicio.
3. El servidor espera las peticiones de los clientes.
4. Un cliente se conecta al servidor.
5. El servidor acepta la conexión.
6. Se intercambian los datos.
7. El cliente, el servidor o ambos cierran la conexión.

# 2.3.- DNS y resolución de nombres.

Todas las computadoras y dispositivos en una red TCP/IP se identifican mediante direcciones IP, como por ejemplo 117.142.234.125.

En IPv4, una dirección IP consta de cuatro bytes sin signo (de 0 a 254) separados por puntos para facilitar su lectura. Sin embargo, recordar estas direcciones puede ser difícil para los seres humanos.

El Sistema de Nombres de Dominio (DNS) es un mecanismo que asigna nombres de dominio, como [www.todofp.es](http://www.todofp.es), a direcciones IP. DNS es un sistema jerárquico que traduce nombres a direcciones IP, permitiendo a los usuarios acceder a los servicios de red utilizando nombres fáciles de recordar en lugar de direcciones IP.

Además de la traducción de nombres a direcciones IP, el DNS ofrece otras ventajas:

* Permite que una misma dirección IP sea compartida por varios dominios.
* Permite que un mismo dominio tenga varias direcciones IP.
* Facilita el cambio de dirección IP de un servicio sin cambiar el nombre de dominio.

**2.4.- El protocolo FTP.**

El protocolo FTP (Protocolo de Transferencia de Archivos) facilita la transferencia de archivos entre sistemas a través de redes TCP/IP. Sus principales características son:

* Permite el intercambio de archivos entre máquinas remotas mediante la red.
* Ofrece conexiones y transferencias de datos rápidas.

Sin embargo, presenta una deficiencia importante en seguridad:

* Las contraseñas e información se transmiten en texto plano, lo que puede ser interceptado por posibles atacantes.

Para abordar esta vulnerabilidad, se puede utilizar la encriptación del tráfico de información a través de protocolos no estándar como SFTP con SSH o FTPS con SSL.

El servicio FTP funciona en una arquitectura cliente/servidor:

* El servidor utiliza dos puertos: 20 para transferencia de datos y 21 para órdenes de control como conexión y desconexión.
* El cliente se conecta al servidor utilizando un puerto local mayor de 1024 y el puerto 21 del servidor como destino.

Las características principales del servicio FTP incluyen:

* La conexión de usuarios remotos al servidor puede ser mediante usuario del sistema, usuario genérico (anonymous) o usuario virtual.
* El acceso al sistema de archivos varía según el tipo de usuario y sus privilegios.
* Soporta dos modos de conexión: activo y pasivo. En modo activo, el servidor inicia la conexión, mientras que en modo pasivo, el cliente inicia la conexión, lo que puede requerir configuración adicional en el servidor para sortear cortafuegos.

# 2.5.- Los protocolos SMTP y POP3.

El correo electrónico es un servicio que permite a los usuarios enviar y recibir mensajes y archivos rápidamente a través de la red. Se basa principalmente en el protocolo SMTP (Protocolo Simple de Transferencia de Correo).

Algunas consideraciones importantes sobre el servicio de correo a través de SMTP son:

* El servidor mantiene las cuentas de los usuarios y los buzones correspondientes.
* Los clientes de correo gestionan la descarga de mensajes y su elaboración.
* El servicio SMTP utiliza el puerto 25 para la transmisión de correo saliente.
* El protocolo SMTP se encarga del transporte del correo desde la máquina del usuario remitente hasta el servidor del destinatario.

El proceso de intercambio de mensajes entre cliente y servidor SMTP es el siguiente:

1. El usuario remitente redacta su mensaje y lo envía hacia su servidor de correo.
2. El servidor SMTP del remitente reenvía el mensaje al servidor del destinatario.
3. El servidor del destinatario almacena el mensaje en el buzón del destinatario.
4. El destinatario puede descargar el mensaje de su buzón utilizando protocolos como POP3 o IMAP, o consultarlos vía web.

# 2.6.- El protocolo HTTP.

El protocolo HTTP, o Protocolo de Transferencia de Hipertexto, es un conjunto de normas que facilita la comunicación entre un servidor y un cliente, permitiendo la transmisión de páginas HTML o páginas web. Algunas consideraciones importantes sobre HTTP son las siguientes:

* Sigue un esquema petición-respuesta entre cliente y servidor.
* Utiliza por defecto el puerto 80.
* El cliente que realiza la petición, como un navegador web, se conoce como agente del usuario (user agent).
* Los recursos se identifican mediante URL, como por ejemplo <http://www.iesalandalus.org/organizacion.htm>.

El funcionamiento esquemático del protocolo HTTP es el siguiente:

1. El usuario especifica la dirección del recurso en el cliente web, como <http://www.iesalandalus.org/organizacion.htm>.
2. El cliente web descompone la URL y establece una conexión al servidor.
3. El cliente solicita el recurso web mediante un mensaje al servidor, como GET /organizacion.htm HTTP/1.1.
4. El servidor responde con un mensaje indicando si existe la página, como HTTP/1.1 200 OK, y envía la página web o un código de error.
5. El cliente interpreta el código HTML recibido.
6. Se cierra la conexión.

HTTP es un protocolo sin estado, lo que significa que no recuerda conexiones anteriores. Para mantener el estado, algunas aplicaciones utilizan cookies.

Principio del formulario

# 3.- Bibliotecas de clases y componentes Java.

Java ofrece amplias capacidades para la interconexión TCP/IP y el desarrollo de aplicaciones cliente/servidor, así como la creación de servicios en red. Algunas características incluyen:

* **API de java.net**: Este paquete proporciona clases para crear una variedad de servicios de red, tanto servidores como clientes.

Otros paquetes relevantes para comunicaciones y servicios en red son:

* **java.rmi**: Permite la implementación de una interfaz de objeto remoto (RMI).
* **javax.mail**: Facilita la implementación de sistemas de correo electrónico.

Para ciertos servicios estándar, Java no proporciona objetos predefinidos, por lo que es necesario recurrir a bibliotecas externas. Por ejemplo, el API proporcionado por Apache Commons Net (paquete org.apache.commons.net) es una opción popular. Esta API permite la implementación de clientes para diversos protocolos estándar como Telnet, FTP o FTP sobre HTTP, entre otros.

# 3.1.- Objetos predefinidos.

El paquete java.net proporciona una API que se divide en dos niveles:

### API de bajo nivel:

* **Direcciones**: Representadas por la clase InetAddress, implementa una dirección IP.
* **Sockets**: Mecanismos básicos de comunicación bidireccional de datos.
  + Socket: Implementa un extremo de una conexión bidireccional.
  + ServerSocket: Implementa un socket que los servidores utilizan para escuchar y aceptar peticiones de clientes.
  + DatagramSocket: Implementa un socket para el envío y recepción de datagramas.
  + MulticastSocket: Representa un socket datagrama para enviar paquetes de multidifusión.
* **Interfaces**: Describen las interfaces de red.
  + NetworkInterface: Representa una interfaz de red con un nombre y una lista de direcciones IP asignadas.

### API de alto nivel:

* **URI**: Representa identificadores de recursos universales.
  + URI: Clase que implementa una URI.
* **URL**: Representa localizadores de recursos universales.
  + URL: Clase que representa una dirección URL.
* **Conexiones**: Representan conexiones con el recurso apuntado por URL.
  + URLConnection: Superclase de todas las clases que representan un enlace de comunicaciones entre la aplicación y una URL.
  + HttpURLConnection: Representa una URLConnection con soporte para HTTP y características especiales.

Estas clases de alto nivel proporcionan una abstracción más fácil de usar para acceder a recursos de red.

# 3.2.- Métodos y ejemplos de uso de InetAddress.

La clase **InetAddress** proporciona métodos para manipular direcciones IP y nombres de dominio, así como para resolver nombres de host a direcciones IP y viceversa. Aquí están los principales métodos de la clase **InetAddress:**

* **getLocalHost():** Devuelve un objeto InetAddress con los datos de direccionamiento del equipo local en la red local.
* **getByName(String host):** Devuelve un objeto InetAddress con los datos de direccionamiento del host especificado.
* **getAllByName(String host):** Devuelve un array de objetos InetAddress con los datos de direccionamiento del host especificado.
* **getHostAddress():** Devuelve la IP como una cadena de texto.
* **getAddress():** Devuelve un array de bytes de la IP.
* **getHostName():** Devuelve el nombre del host como una cadena de texto.
* **isReachable(int tiempo):** Devuelve TRUE o FALSE si la dirección es alcanzable en el tiempo indicado.

Estos métodos pueden generar una excepción UnknownHostException si no pueden resolver el nombre pasado como parámetro. Además, la clase no tiene constructores, pero se pueden obtener instancias de InetAddress mediante los métodos estáticos mencionados.

# 3.3.- Programación con URL.

Una URL, o Localizador Uniforme de Recursos, representa una dirección a un recurso en la World Wide Web. Puede referirse a sitios web, archivos, sitios FTP, direcciones de correo electrónico, entre otros. La estructura de una URL se puede dividir en las siguientes partes:

* **Protocolo**: El protocolo utilizado para la comunicación, como HTTP, HTTPS, FTP, etc.
* **Nombre del host**: El nombre del servidor que proporciona el recurso.
* **Puerto**: El puerto de red en el servidor para conectarse. Si no se especifica, se utiliza el puerto por defecto para el protocolo.
* **Ruta**: La ruta o ubicación del recurso en el servidor.
* **Referencia**: Un fragmento opcional que indica una parte específica dentro del recurso.

Por ejemplo:

* http://www.iesalandalus.org/organizacion.htm: Utiliza el protocolo HTTP, el nombre del host es [www.iesalandalus.org](http://www.iesalandalus.org) y la ruta es organizacion.htm. Si no se especifica, se utiliza el puerto por defecto para HTTP, que es el 80.
* http://www.iesalandalus.org:85/organizacion.htm: Aquí se especifica el puerto 85.
* http://www.dominio.es/public/pag.html#apartado1: Indica la ruta /public/pag.html y la referencia #apartado1 dentro de la página HTML.

La clase URL en Java permite representar y trabajar con URLs. Con ella, puedes establecer conexiones con recursos en Internet o en una red local. Puedes invocar métodos sobre el objeto URL para obtener el contenido del recurso en el cliente. También facilita la conexión entre computadoras y el intercambio de datos utilizando streams, ofreciendo una funcionalidad similar a la programación de sockets pero de manera más integrada y orientada a URLs.

# 3.4.- Crear y analizar objetos URL.

Para crear objetos URL en Java, la clase URL ofrece varios constructores que permiten especificar diferentes partes de la URL:

1. Se pueden pasar todos los elementos de la URL por separado, como el protocolo, el host, el puerto y el archivo:

**URL url = new URL("http", "www.iesalandalus.org", 80, "index.htm");**

Esto crearía la URL: http://www.iesalandalus.org:80/index.htm.

1. También se puede pasar simplemente la cadena completa de la URL y dejar que el sistema utilice los valores por defecto:

**URL url = new URL("http://www.iesalandalus.org");**

Esto crearía la URL: http://www.iesalandalus.org.

1. Otra opción es pasar una ruta relativa como primer parámetro.
2. Además, se pueden proporcionar especificaciones personalizadas junto con un manejador del protocolo.

Cada constructor puede lanzar una excepción MalformedURLException si los argumentos son nulos o si el protocolo es desconocido. Es importante manejar esta excepción usando un bloque try-catch.

Una vez creada, una URL es de solo lectura y no se pueden cambiar sus atributos. Sin embargo, se pueden analizar y descomponer utilizando métodos como getProtocol(), getHost(), getPort(), getDefaultPort(), getFile(), y getRef() para obtener información sobre el protocolo, host, puerto, archivo y referencia de la URL respectivamente.

**3.5.- Leer y escribir a través de una URLConnection.**

Final del formulario

Un objeto URLConnection en Java se puede utilizar para leer desde y escribir hacia el recurso al que hace referencia una URL.

Dos métodos principales que se utilizan para trabajar con conexiones URL son:

1. **URL.openConnection():** Este método devuelve un objeto URLConnection que representa una nueva conexión con el recurso remoto al que se refiere la URL.
2. **URL.openStream():** Este método abre una conexión a la dirección URL y devuelve un InputStream para la lectura de esa conexión. Es una abreviatura de openConnection().getInputStream().

A continuación, se muestra un ejemplo básico que lee una URL utilizando estos métodos:

**import java.net.\*;**

**public class Main {**

**public static void main(String[] args) {**

**try {**

**URL url = new URL("https://www.example.com");**

**URLConnection connection = url.openConnection();**

**InputStream inputStream = url.openStream();**

**// Manejar los flujos necesarios para realizar la lectura**

**// Por ejemplo, leer el contenido de la URL línea por línea**

**BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));**

**String line;**

**while ((line = reader.readLine()) != null) {**

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

**}**

**reader.close();**

**} catch (IOException e) {**

**e.printStackTrace(); }**

**} }**

Este código abre una conexión a la URL "https://www.example.com", obtiene un InputStream para leer el contenido de esa conexión, y luego lee el contenido línea por línea e imprime cada línea en la consola.

Además de leer archivos de texto o analizar el código fuente de una página web, los objetos URLConnection se pueden utilizar para diversos propósitos, como descargar archivos, validar el contenido de una URL, etc.

# 3.6.- Trabajar con el contenido de una URL.

Para obtener el contenido de un objeto URL, se utiliza el método:

* **URL.getContent():** Este método devuelve el contenido de la URL. Internamente, determina el tipo de contenido del objeto URL llamando al método getContentType(). Si es la primera vez que la aplicación ha visto ese tipo de contenido específico, se crea un manejador para dicho tipo de contenido.

Crear una conexión URL implica los siguientes pasos:

1. Crear un objeto URLConnection, lo cual se hace llamando al método openConnection() de un objeto URL.
2. Establecer los parámetros y propiedades de la petición.
3. Establecer la conexión utilizando el método connect().
4. Se puede obtener información sobre la cabecera y/o el recurso remoto.

Algunos métodos útiles de la clase URLConnection son:

* **connect():** Establece una conexión entre la aplicación (el cliente) y el recurso (el servidor), permitiendo interactuar con el recurso y consultar los tipos de cabeceras y contenidos para determinar el tipo de recurso.
* **getContentType():** Devuelve el valor del campo de cabecera content-type, o null si no está definido.
* **getContentLength():** Devuelve el valor del campo de cabecera content-length, o -1 si no está definido.
* **getLastModified():** Devuelve la fecha de la última modificación del recurso.

En resumen, para acceder a recursos de Internet a través de objetos URL y URLConnection, se siguen estos pasos y se utilizan estos métodos para establecer la conexión y obtener información sobre el recurso remoto.

# 4.- Programación de aplicaciones cliente.

Es esencial entender cómo funcionan los protocolos (HTTP, FTP, SMTP, TELNET) al programar aplicaciones cliente para ellos. Se utilizan bibliotecas especiales para facilitar la programación, pero comprender el intercambio de mensajes entre cliente y servidor es crucial, especialmente si se usa programación a niveles bajos como sockets.

**4.1.- Programación de un cliente HTTP.**

El ejemplo visto sobre el acceso a recursos de la red mediante URL utilizando las clases URL y URLConnection se puede considerar como un cliente HTTP básico. Este tipo de cliente HTTP es más sencillo y no realiza todas las operaciones complejas que un navegador web completo haría, como la interpretación y renderización de páginas HTML. Por otro lado, al programar aplicaciones con las clases URL y URLConnection, trabajamos en un nivel más alto de abstracción, donde se oculta la complejidad de la comunicación subyacente que era evidente al programar un cliente HTTP con sockets.

**4.2.- Bibliotecas para programar un cliente FTP.**

El paquete org.apache.commons.net.ftp de la Apache Software Foundation proporciona las clases necesarias para implementar clientes FTP en aplicaciones Java. Algunas de estas clases incluyen:

* **FTP:** Proporciona funcionalidades básicas para un cliente FTP. Esta clase hereda de SocketClient.
* **FTPReply:** Permite almacenar las respuestas devueltas por el servidor como códigos de respuesta a las solicitudes del cliente.
* **FTPClient:** Encapsula toda la funcionalidad para almacenar y recuperar archivos de un servidor FTP, manejando los detalles de bajo nivel para la interacción con el servidor. Hereda de SocketClient.
* **FTPClientConfig:** Ofrece una forma alternativa de configurar objetos FTPClient.
* **FTPSClient:** Proporciona soporte para FTP seguro sobre SSL. Hereda de FTPClient.

**4.3.- Programación de un cliente FTP.**

Para crear un cliente FTP utilizando la clase FTPClient:

1. Realizar la conexión del cliente con el servidor utilizando el método connect(InetAddress host), que establece una conexión con el servidor especificado.
2. Comprueba la conexión utilizando el método getReplyCode() para obtener un código de respuesta del servidor que indique el éxito o fracaso de la conexión.
3. Valida el usuario utilizando el método login(String usuario, String password) para autenticarse en el servidor.
4. Realiza operaciones contra el servidor, como listar archivos disponibles en una carpeta remota con listNames() o descargar el contenido de un archivo remoto utilizando retrieveFile(String rutaRemota, OutputStream ficheroLocal).
5. Finalmente, desconéctate del servidor utilizando disconnect() o logout().

Recuerda que durante este proceso pueden surgir excepciones como SocketException si se supera el tiempo de espera para la conexión, o IOException si no se puede acceder al archivo especificado.

**4.4.- Programación de un cliente Telnet.**

El protocolo Telnet, o Telecommunication Network, permite acceder y administrar equipos de forma remota a través de la red, utilizando un modelo cliente/servidor. Funciona en modo texto y el servidor escucha las solicitudes en el puerto 23.

Aunque Telnet puede ser útil para administrar dispositivos remotos, no se utiliza ampliamente en la actualidad debido a sus deficiencias de seguridad. Toda la información, incluidos los datos de usuario y contraseña, se transmiten en texto plano, lo que lo hace vulnerable a ataques. Por esta razón, se prefieren soluciones más seguras como SSH (Secure Shell).

La API de Apache proporciona una biblioteca para programar clientes Telnet, llamada org.apache.commons.net.telnet. Esta biblioteca incluye la clase TelnetClient, que permite implementar un terminal virtual para el protocolo Telnet. Algunos métodos importantes de esta clase son connect() para establecer la conexión con el servidor, getInputStream() y getOutputStream() para enviar y recibir datos, y disconnect() para cerrar la conexión.

**4.5.- Programación de un cliente SMTP.**

En el primer caso, un cliente SMTP basado en sockets permite enviar correos electrónicos directamente a través de la comunicación directa con el servidor SMTP. En el segundo caso, utilizando el API javax.mail, se facilita la implementación de un cliente SMTP con clases como Session, Message y Transport, que proporcionan métodos para configurar y enviar mensajes de correo electrónico de manera más eficiente y simple. La clase Session permite configurar propiedades como el protocolo, servidor SMTP, y autenticación. La clase Message modela un mensaje de correo electrónico y proporciona métodos para establecer el remitente, destinatarios, asunto y cuerpo del mensaje. La clase Transport se encarga del transporte del mensaje y proporciona el método send() para enviar el mensaje a las direcciones especificadas.

**5.- Programación de servidores.**

Cuando se diseña o programa un servidor o servicio en red, es crucial considerar los siguientes aspectos:

1. Capacidad para manejar múltiples peticiones concurrentes: El servidor debe ser capaz de atender varias solicitudes simultáneamente. Esto se logra mediante la programación del servidor utilizando hilos o Threads, lo que permite gestionar las conexiones de forma concurrente.
2. Optimización del tiempo de respuesta: Es fundamental controlar y optimizar el tiempo de respuesta del servidor. Esto se logra mediante la monitorización de los tiempos de proceso y transmisión del servidor, identificando y mejorando los cuellos de botella en el rendimiento.

En Java, la clase ServerSocket se utiliza para crear servidores. Para programar servidores basados en protocolos de aplicación como HTTP, es necesario comprender el comportamiento y funcionamiento del protocolo en cuestión, así como los mensajes intercambiados entre el servidor y el cliente durante las solicitudes de datos.

**5.1.- Programación de un servidor HTTP.**

Para comprender cómo funciona el servidor web y establecer las hipótesis de trabajo, consideraremos lo siguiente:

1. **Protocolo HTTP y sus versiones:** El servidor estará basado en la versión 1.1 del protocolo HTTP y solo implementará una parte de este protocolo.
2. **Tipos de mensajes:** El servidor manejará dos tipos de mensajes: peticiones de clientes a servidores y respuestas de servidores a clientes. En este caso, el servidor solo implementará peticiones GET.

Para crear el servidor HTTP, seguiremos estos pasos básicos:

1. **Creación de un socket servidor:** Se creará un socketServer asociado al puerto 80, que es el puerto por defecto para el protocolo HTTP.
2. **Espera de peticiones del cliente:** El servidor estará en un bucle esperando las peticiones entrantes de los clientes.
3. **Aceptar la petición del cliente:** Cuando se reciba una petición del cliente, el servidor la aceptará y la procesará.
4. **Procesar la petición:** Se llevará a cabo el intercambio de mensajes según el protocolo HTTP, y se transmitirá la información solicitada por el cliente.
5. **Cerrar el socket del cliente:** Una vez que se haya respondido a la petición del cliente, se cerrará el socket correspondiente.

# 5.2.- Implementar comunicaciones simultáneas.

Para habilitar la capacidad de nuestro servidor para atender comunicaciones simultáneas, necesitamos utilizar hilos o threads. El enfoque típico para lograr esto implica modificar el código para que pueda ejecutar múltiples hilos de manera concurrente ,de la siguiente manera:

1. El hilo principal (o el que inicia la aplicación) creará el socket servidor que estará esperando la llegada de peticiones.
2. Cuando llegue una petición, el hilo principal la aceptará y asignará un socket cliente para enviar la respuesta. En lugar de manejarla directamente, el hilo principal creará un nuevo hilo para manejarla y enviarla a través del socket cliente asignado. Esto le permitirá al hilo principal seguir esperando nuevas peticiones.

Esquemáticamente, el código del hilo principal tendría el siguiente aspecto:

**try {**

**ServerSocket serverSocket = new ServerSocket(port);**

**while (true) {**

**// Acepta una petición y asigna un socket cliente para la respuesta**

**Socket clientSocket = serverSocket.accept();**

**// Crea un nuevo hilo para manejar la petición y enviar la respuesta**

**Thread thread = new RequestHandlerThread(clientSocket);**

**thread.start();**

**}**

**} catch (IOException ex) {**

**// Manejar la excepción**

**}**

La clase RequestHandlerThread sería una extensión de la clase Thread de Java, cuyo constructor almacenaría el Socket cliente que recibe, para que su método run() lo utilice posteriormente para manejar la respuesta:

**class RequestHandlerThread extends Thread {**

**private Socket clientSocket;**

**public RequestHandlerThread(Socket clientSocket) {**

**this.clientSocket = clientSocket;**

**}**

**public void run() {**

**try {**

**// Manejar la petición utilizando el socket cliente**

**} catch (IOException ex) {**

**// Manejar la excepción**

**}**

**}**

**}**

**5.3.- Monitorización de tiempos de respuesta.**

Para medir los tiempos de respuesta del servidor, podemos considerar dos aspectos:

1. **Tiempo de procesamiento:** Este es el tiempo que el servidor necesita para procesar la solicitud del cliente y enviar los datos de respuesta. Podemos medir este tiempo en el servidor mismo usando el siguiente código para medir el tiempo de procesamiento en milisegundos:

**long startTime = System.currentTimeMillis();**

**// Procesamiento de la solicitud del cliente**

**long endTime = System.currentTimeMillis();**

**long processingTime = endTime - startTime;**

1. **Tiempo de transmisión:** Este es el tiempo que tardan los mensajes en viajar desde el servidor al cliente a través de la red. Para medir este tiempo, el servidor puede enviar un mensaje al cliente con la marca de tiempo del sistema en el momento en que envía la respuesta. El cliente, al recibir este mensaje, puede calcular su propio tiempo del sistema y compararlo con el tiempo recibido en el mensaje. Es importante que los relojes de sistema de ambos equipos estén sincronizados para obtener mediciones precisas. En Windows, por ejemplo, la sincronización de los relojes se realiza automáticamente a través del servicio de tiempo (NTP).

PSP6

**Técnicas de programación segura.**

**1.- Introducción a la seguridad Informática.**

La seguridad en sistemas informáticos se define por su capacidad para garantizar:

1. **Confidencialidad:** Solo las entidades autorizadas pueden acceder a la información.
2. **Integridad:** La información solo puede ser modificada por entidades autorizadas.
3. **No repudio:** Impide que las partes nieguen haber realizado ciertas comunicaciones.
4. **Disponibilidad:** Los recursos del sistema están disponibles para las entidades autorizadas cuando se necesitan.

La gestión de incidentes de seguridad requiere una política de seguridad que defina responsabilidades y reglas para evitar y minimizar amenazas. Se deben proteger tres elementos principales: hardware, software y datos, siendo estos últimos los más sensibles y vitales para la organización. La seguridad debe aplicarse a todos los niveles y elementos del sistema.

**1.1.- Amenazas de seguridad.**

Las amenazas de seguridad se refieren a condiciones en el entorno del sistema que, si se aprovechan, podrían resultar en una violación de la seguridad, afectando la confidencialidad, integridad, disponibilidad o el uso legítimo de los recursos.

Estas amenazas pueden ser clasificadas en cuatro categorías generales:

1. **Interrupción:** Se produce cuando un recurso del sistema es destruido o deja de estar disponible, atacando la disponibilidad.
2. **Intercepción:** Sucede cuando una entidad no autorizada obtiene acceso a un recurso, afectando la confidencialidad.
3. **Modificación:** Ocurre cuando una entidad no autorizada accede y manipula un recurso, comprometiendo la integridad.
4. **Fabricación:** Se presenta cuando una entidad no autorizada inserta elementos falsificados en el sistema, atacando la autenticidad.

**1.2.- Ataques.**

Los ataques se pueden clasificar en ataques pasivos y ataques activos:

1. **Ataques pasivos:** En estos ataques, el atacante simplemente escucha o monitoriza la comunicación sin alterarla. El objetivo es obtener información confidencial, como contraseñas o datos sensibles. Son difíciles de detectar ya que no alteran los datos, pero se pueden prevenir mediante el cifrado de la información.
2. **Ataques activos:** Estos ataques implican la modificación o creación de datos en la comunicación. Pueden incluir suplantación de identidad, reenvío de mensajes, modificación de datos o denegación de servicio, entre otros. Son más fáciles de detectar que los ataques pasivos, pero pueden causar daños significativos si no se previenen adecuadamente.

**1.3.- Vulnerabilidades en el software.**

Para garantizar la seguridad de una aplicación en Java, nos centraremos en dos pilares fundamentales:

1. **Seguridad interna de la aplicación:** Se refiere a programar la aplicación de forma robusta para que funcione como se espera y esté protegida contra posibles fallos. Algunas técnicas para lograr esto incluyen la gestión de excepciones, validaciones de entradas de datos y el uso de registros de actividad para detectar problemas.
2. **Políticas de acceso:** Después de asegurar la aplicación, es importante definir políticas de acceso para determinar las acciones que la aplicación puede realizar en el sistema. Por ejemplo, se pueden establecer reglas que limiten el acceso a ciertos archivos o recursos del sistema. Esto ayuda a mitigar el impacto de posibles ataques, ya que incluso si un usuario malicioso utiliza la aplicación de manera incorrecta, sus acciones estarán restringidas por estas políticas de acceso.

**2.- Programación segura**

**2.1.- Excepciones (I).**

Las excepciones son eventos que interrumpen el flujo normal de ejecución de un programa. Pueden ocurrir por diversas razones, como dispositivos no disponibles, falta de permisos, archivos inexistentes o problemas de red.

En Java, los tipos fundamentales de excepciones son:

1. **Error:** Indican problemas graves y no recuperables, que generalmente no deben ser capturados.
2. **Exception:** Excepciones no definitivas que se detectan durante el tiempo de ejecución.
3. **RuntimeException:** Excepciones que ocurren durante la ejecución del programa.

Es importante manejar adecuadamente las excepciones en un programa para evitar fallos en su ejecución.

**2.1.1.- Excepciones (II).**

El manejo de excepciones en un programa generalmente implica los siguientes pasos:

1. **Bloque Try:** Colocar el código que puede generar excepciones dentro de un bloque try.
2. **Captura de Excepciones:** Identificar y especificar qué hacer en caso de que ocurran excepciones. Esto se hace utilizando bloques catch para cada tipo de excepción que se desea manejar.
3. **Manejo de Excepciones:** Dentro de cada bloque catch, se incluye el código que se ejecutará para manejar la excepción específica.
4. **Bloque Finally (opcional):** Utilizar un bloque finally para especificar código que siempre se ejecutará, independientemente de si ocurre una excepción o no. Esto es útil para liberar recursos u otras tareas de limpieza.

Por ejemplo:

**try {**

**// Código posiblemente problemático**

**} catch (TipoDeExcepcion1 e) {**

**// Código para manejar la excepción de TipoDeExcepcion1**

**} catch (TipoDeExcepcion2 e) {**

**// Código para manejar la excepción de TipoDeExcepcion2**

**} finally {**

**// Código que se ejecutará siempre, independientemente de si se lanzó una excepción o no**

**}**

Es importante tener en cuenta que las excepciones deben ser manejadas de manera adecuada para garantizar la estabilidad y la robustez del programa.

**2.2.- Ejemplo I.**

**import java.io.\*;**

**import java.util.Vector;**

**import java.util.Random;**

**class EscribirNumeros {**

**private Vector<Integer> numeros;**

**private static final int SIZE = 100;**

**public EscribirNumeros (){**

**// Generamos el vector con números aleatorios**

**numeros = new Vector<Integer>(SIZE);**

**Random randomGenerator = new Random();**

**for (int i = 0; i < SIZE; i++){**

**Integer num=randomGenerator.nextInt(100);**

**numeros.addElement(new Integer(num)); }**

**// Guardamos el vector en un fichero**

**PrintWriter out = null;**

**try {**

**System.out.println("Entrando a la zona Try");**

**out = new PrintWriter(new FileWriter("Salida.txt"));**

**for (int i = 0; i < SIZE; i++)**

**out.println("Valor de: " + i + " = " + numeros.elementAt(i));**

**} catch (ArrayIndexOutOfBoundsException e) {**

**// Manejo de la excepción ArrayIndexOutOfBoundsException**

**System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());**

**} catch (IOException e) {**

**// Manejo de la excepción IOException**

**System.err.println("Caught IOException: " + e.getMessage());**

**} finally {**

**// Cierre del fichero en el bloque finally para asegurar que se ejecuta**

**if (out != null) {**

**System.out.println("Cerrando el fichero");**

**out.close();**

**} else {**

**System.out.println("NO se ha abierto el fichero");**

**}**

**}**

**}**

**public static void main( String[] arg ) {**

**new EscribirNumeros ();**

**}**

**}**

En este código, se utilizan bloques try, catch y finally para manejar posibles excepciones durante la ejecución del programa. Esto garantiza que el programa pueda continuar ejecutándose de manera controlada incluso si ocurren errores.

**2.3.- Validación de entradas.**

La validación de datos es crucial para mantener la integridad y la seguridad de una aplicación, especialmente cuando se trata de datos introducidos por los usuarios. Una técnica comúnmente utilizada para realizar esta validación es el uso de expresiones regulares.

En Java, la clase Pattern del paquete java.util.regex proporciona una forma poderosa de definir y aplicar expresiones regulares para validar el formato de los datos de entrada. Aquí tienes un resumen de cómo utilizar expresiones regulares para la validación de datos en Java:

1. **Importar la librería:**

 **import java.util.regex.\*;**

1. **Definir Pattern y Matcher:**

 **Pattern pat = null;**

**Matcher mat = null;**

1. **Compilar el patrón a utilizar:**

 **pat = Pattern.compile(patron);**

Donde patron es la expresión regular que define el formato deseado para la entrada de datos.

 **Elementos y operadores para la validación de entradas:**

Aquí hay algunos ejemplos de elementos y operadores que se pueden usar en expresiones regulares para validar entradas:

* **[a-z]:** Una letra en minúscula.
* **[A-Z]:** Una letra en mayúscula.
* **[0-9]:** Un número comprendido entre 0 y 9.
* **{2}:** Indica que se deben introducir exactamente 2 caracteres.
* **{2,5}:** Indica que se deben introducir de 2 a 5 caracteres.
* **|:** Operación OR lógica para indicar múltiples opciones.

 **Pasos para la validación:**

* 1. Pasar el texto a comprobar al evaluador de expresiones:

**mat = pat.matcher(texto\_a\_comprobar);**

* 1. Comprobar si hay alguna coincidencia:

**if(mat.find()) {**

**// Coincide con el patrón**

**} else {**

**// NO coincide con el patrón**

**}**

Estos pasos te permitirán validar datos de entrada en Java utilizando expresiones regulares, lo que ayuda a mantener la consistencia y seguridad de tu aplicación.Principio del formulario

**2.4.- Ejemplo II.**

**import java.io.\*;**

**import java.util.regex.\*;**

**class ValidarEntrada {**

**public ValidarEntrada(){**

**String dni\_cliente=new String();**

**Pattern pat=null;**

**Matcher mat=null;**

**// para leer del teclado**

**BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));**

**try{**

**System.out.println("Introduce tu DNI (Formato 00000000-A):");**

**dni\_cliente=reader.readLine();**

**pat=Pattern.compile("[0-9]{8}-[a-zA-Z]");**

**mat=pat.matcher(dni\_cliente);**

**if(mat.find()){**

**System.out.println("Correcto!! "+dni\_cliente);**

**}else{**

**System.out.println("El DNI esta mal "+dni\_cliente);**

**}**

**} catch( Exception e ) {**

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

**}**

**}**

**public static void main( String[] arg ) {**

**new ValidarEntrada(); }**

**}**

**2.5.- Ficheros de registro.**

**Importar la librería:**

**import java.util.logging.\*;**

**Definir un logger:**

**Logger logger = Logger.getLogger("MyLog");**

**Asociar el logger a un fichero de log:**

**FileHandler fh = new FileHandler("c:\\MyLogFile.log", true);**

Donde "c:\\MyLogFile.log" es la ruta y el nombre del fichero de registro. El segundo parámetro true indica que se añadirán registros al fichero existente.

**Establecer si queremos visualizar los mensajes de log por pantalla:**

**logger.setUseParentHandlers(false);**

Esto evita que los mensajes de registro se muestren por pantalla.

**Establecer el formato del fichero:**

**SimpleFormatter formatter = new SimpleFormatter();**

**fh.setFormatter(formatter);**

Puedes elegir entre un formato de texto simple (SimpleFormatter) o un formato XML (XMLFormatter).

**Establecer el nivel de seguridad de los registros:**

**logger.setLevel(Level.SEVERE);**

Puedes establecer diferentes niveles de seguridad para los registros, como SEVERE, WARNING, INFO, etc.

**Agregar registros a la aplicación:**

**logger.log(Level.WARNING, "Mi primer log");**

Utiliza el método log() del logger para registrar eventos con un determinado nivel de importancia.

Siguiendo estos pasos, puedes configurar y utilizar registros en tu aplicación Java para realizar un seguimiento detallado de las acciones y eventos que ocurren durante su ejecución. Esto es útil para depurar y diagnosticar problemas, así como para mantener un registro de la actividad del sistema.

**2.6.- Ejemplo III.**

**MyLogger.java**

**import java.io.\*;**

**import java.util.logging.\*;**

**public class MyLogger {**

**public static void main(String[] args) {**

**Logger logger = Logger.getLogger(”MyLog”);**

**FileHandler fh;**

**try {**

**// Configuro el logger y establezco el formato**

**fh = new FileHandler(”c:\\MyLogFile.log”, true);**

**logger.addHandler(fh);**

**logger.setLevel(Level.ALL);**

**SimpleFormatter formatter = new SimpleFormatter();**

**fh.setFormatter(formatter);**

**// Añado un mensaje al log**

**logger.log(Level.WARNING,”Mi primer log”);**

**} catch (SecurityException e) {**

**e.printStackTrace() ;**

**} catch (IOException e) {**

**e.printStackTrace() ; }**

**} }**

**3.- Políticas de seguridad.**

**3.1.- Modelo de seguridad de java.**

### Modelos de seguridad en Java:

1. **JDK 1.0:**
   * Modelo básico donde las aplicaciones locales tenían total acceso a todos los recursos del sistema y las aplicaciones remotas tenían restricciones.
2. **JDK 1.1:**
   * Se permitió que las aplicaciones firmadas digitalmente accedieran a todos los recursos del sistema, aumentando así la flexibilidad y el control sobre el acceso.
3. **JDK 1.2:**
   * Se mejoró la seguridad permitiendo establecer políticas de acceso tanto para aplicaciones locales como remotas.
   * Las solicitudes de acceso se envían al Security Manager para su autorización después de verificar la política de seguridad.

### Elementos para establecer políticas de seguridad:

1. **Origen:** Puede ser el usuario o la ruta de acceso de la aplicación, determinando quién o qué tiene acceso.
2. **Permisos:** Especifica los privilegios de acceso, como lectura/escritura sobre un fichero o envío de información.
3. **Destino:** Indica el objetivo al que se aplica el permiso, como la ubicación de un fichero en el disco.
4. **Acción:** Describe las acciones permitidas sobre el recurso, como leer o escribir en un fichero.

### Permisos más utilizados en Java:

1. **PropertyPermission:** Permite establecer permisos de acceso sobre información del sistema, como el sistema operativo, la versión de Java o el directorio home del usuario.
2. **SocketPermission:** Controla los permisos sobre las comunicaciones de red, pudiendo establecer restricciones sobre direcciones IP, URL y puertos de comunicación.
3. **FilePermission:** Regula los permisos de acceso al disco para una aplicación, permitiendo establecer permisos de lectura o escritura sobre ficheros o carpetas específicos.

Estos elementos y permisos proporcionan un marco sólido para establecer políticas de seguridad en las aplicaciones Java, asegurando un control adecuado sobre el acceso a los recursos del sistema.

**3.2.- Asegurando las aplicaciones (I).**

* Las aplicaciones locales se ejecutan sin Security Manager por defecto, lo que les permite acceder a los recursos libremente.
* Para aplicar políticas de seguridad, se activa el Security Manager con la opción "-Djava.security.manager".
* Con el Security Manager activado, se deniega el acceso a datos sensibles como el directorio del usuario, pero se permite el acceso a datos no sensibles como la versión del sistema operativo.
* Es necesario establecer políticas de acceso adecuadas para gestionar el acceso a datos sensibles.

### Código: GetProps.java

**import java.lang.\*;**

**import java.security.\*;**

**class GetProps {**

**public static void main(String[] args) {**

**String s;**

**try {**

**System.out.println("About to get os.name property value");**

**s = System.getProperty("os.name", "not specified");**

**System.out.println(" The name of your operating system is: " + s);**

**System.out.println("About to get java.version property value");**

**s = System.getProperty("java.version", "not specified");**

**System.out.println(" The version of the JVM you are running is: " + s);**

**System.out.println("About to get user.home property value");**

**s = System.getProperty("user.home", "not specified");**

**System.out.println(" Your user home directory is: " + s);**

**System.out.println("About to get java.home property value");**

**s = System.getProperty("java.home", "not specified");**

**System.out.println(" Your JRE installation directory is: " + s);**

**} catch (Exception e) {**

**System.err.println("Caught exception " + e.toString()); }**

**} }**

### Código: Ejecución con Security Manager

**java -Djava.security.manager GetProps**

Esta línea de comando activa el Security Manager para la aplicación GetProps. Al ejecutarla, se deniega el acceso a datos sensibles como el directorio del usuario, pero se permite el acceso a datos no sensibles como la versión del sistema operativo.

**3.3.- Firmando ficheros Jar.**

**Comandos:**

1. Compilación: **javac GetProps.java**
2. Creación del archivo JAR: **jar cvf GetProps.jar GetProps.class**
3. Generación de claves: **keytool -genkey -alias firmar -keypass hola00 -keystore DAM -storepass distancia**
4. Firma del archivo JAR: **jarsigner -keystore DAM -signedjar sGetProps.jar GetProps.jar firmar**
5. Exportación de la llave pública: **keytool -export -keystore DAM -alias firmar -file Javier.cert**

**3.4.- Utilizando ficheros Jar firmados.(no hacer caso)**

1. Ejecutar la aplicación con el Security Manager activado:

**java -cp sGetProps.jar GetProps**

**java -Djava.security.manager -cp sGetProps.jar GetProps**

1. Importar el certificado:

**keytool -import -alias Javier -file Javier.cert -keystore DAM**

1. Configurar las políticas de acceso:
   * Ejecuta policytool.
   * Abre el archivo java.policy.
   * Edita el almacén de claves y establece la ubicación de las claves.
   * Agrega una entrada para permitir el acceso de tu código, especificando el usuario Javier.
   * Añade permisos para user.home y java.home con acciones de lectura.
   * Guarda los cambios.
2. Vuelve a ejecutar la aplicación:

**java -Djava.security.manager -cp sGetProps.jar GetProps**

**3.5.- Herramientas de seguridad.**

Las herramientas proporcionadas por el JDK para implementar las normas de seguridad de Java son:

1. **policytool**: Herramienta gráfica para administrar las normas de seguridad. Se ejecuta con el comando policytool. Permite abrir y modificar archivos de normas, con la ruta predeterminada en c:\Archivos de programa\Java\jre7\lib\security\java.policy.
2. **keytool**: Permite gestionar la base de datos de claves y certificados del sistema. Se utiliza con el comando keytool [comandos]. Se puede obtener la lista de comandos y opciones con keytool -help.
3. **jarsigner**: Permite firmar y verificar la firma de los archivos JAR. Su sintaxis es jarsigner [opciones] Fichero.jar alias, donde Fichero.jar es el archivo a firmar o verificar, y alias es el alias de entrada al almacén de claves que contiene la clave privada utilizada para generar la firma. Las opciones se pueden consultar ejecutando simplemente jarsigner.

**Anexo I.- Ataques.**

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Descripción** | |
| **Sistemas** | |  |
| **Explotar bugs del software** | Utilizar fallos de seguridad en el software para atacar un sistema. |  |
| **Romper contraseñas** | Fuerza bruta o ataques basados en diccionarios que permiten obtener las contraseñas del sistema o de un determinado servicio. |  |
| **Red** | |  |
| **Barridos de ping** | Utilización del protocolo ICMP para determinar los equipos activos de una red. |  |
| **Confianza transitiva** | Aprovechar la confianza entre usuarios o equipos para tomar sus privilegios. |  |
| **DNS spoofing** | Falsificación de una entrada DNS que apunta a un servidor no autorizad |  |
| **DoS** | Ataque de denegación de servicio que consiste en saturar un sistema para impedir la utilización correcta del sistema. |  |
| **Hijacking** | Permite a un usuario robar una conexión de un usuario que ha sido autentificado en el sistema. |  |
| **Man in the middle** | A través del ataque ARPspoofing el atacante se sitúa en medio de la comunicación entre varios equipos para realizar otros ataques, como sniffer, spoofing, phising etc. |  |
| **Mensajes de control de red o enrutamiento fuente** | Se envían paquetes ICMP para hacer pasar los paquetes por un router comprometido. |  |
| **Navegación anónima** | No se considera directamente un ataque, pero la suelen utilizar los atacantes para realizar sus fechorías. Se denomina “navegación anónima” cuando un usuario utilizar diferentes servidores proxy para ocultar su dirección IP. |  |
| **Phising** | Ataque informático que consiste en falsificar un sitio web para poder obtener las contraseñas de sus usuarios. |  |
| **Reenvío de paquetes** | Retransmisión de paquetes para engañar o duplicar un mensaje (p.ej. una transferencia). |  |
| **Sniffer** | Programa o equipo que registra todo el tráfico de una red. Se utiliza especialmente para obtener las contraseñas de los sistemas. |  |
| **Spoofing** | El atacante envía paquetes con una dirección fuente incorrecta. Las respuestas se envían a la dirección falsa. Pueden usarse para: Acceder a recursos confiados sin privilegios Para DoS directo como indirecto o recursivo. |  |
| **Servidores web** | |  |
| **Inyección SQL** | Ataque que consiste en modificar las consultas SQL de un servidor web para poder realizar consultas SQL maliciosas. |  |
| **LFI** | Ataque informático que consiste en hacer que un servidor ejecute un script que está alojado en el mismo servidor. |  |
| **RFI** | Ataque informático que consiste en hacer que un servidor ejecute un script que está alojado en una máquina remota. |  |
| **XSS** | Consiste en engañar al servidor web para que ejecute un script malicioso en el navegador del cliente que visita una determinada página. |  |
| **Aplicaciones** | |  |
| **Crack** | Software que permite romper la protección de una aplicación comercial. |  |
| **Keylogger** | Software o hardware que registra todas las pulsaciones de teclado que se realizan en el sistema. |  |
| **Rootkit** | Software que se instala en un sistema y oculta toda la actividad de un usuario (el atacante). |  |
| **Troyano** | Software que se instala en el ordenador atacado que permite al atacante hacerse con el control de la máquina. |  |
| **Virus** | Software que se copia automáticamente y que tiene por objeto alterar el normal funcionamiento del equipo sin el permiso o el conocimiento del usuario. |  |
| **Varios** | |  |
| **Ingeniería social** | Atacante que consiste en convencer a un usuario legítimo para que facilite información (contraseñas, configuraciones, etc.). |  |
| **Rubber-hosse** | Utilizar soborno o tortura para obtener una determinada información. |  |

PSP7

# Aplicaciones con comunicaciones seguras.

**1.- Introducción.**

La seguridad en aplicaciones con comunicaciones se centra en garantizar la confidencialidad, integridad, autenticación, no repudio y autorización de los datos transmitidos. La criptografía es fundamental para lograr estos objetivos, proporcionando métodos para cifrar y firmar digitalmente la información.

**2.- Criptografía.**

La criptografía es el arte y la ciencia de codificar información de manera que solo las personas autorizadas puedan acceder a ella. Su objetivo es asegurar la confidencialidad, integridad y autenticidad de los datos. Se utiliza para enmascarar mensajes de manera que sean ininteligibles para personas no autorizadas, protegiendo así la información sensible durante su transmisión. El criptoanálisis es el estudio de la seguridad de los sistemas criptográficos, buscando vulnerabilidades y formas de romper la seguridad. La criptología es el campo que engloba tanto la criptografía como el criptoanálisis.

**2.1.- Encriptación de la información.**

La encriptación es el proceso de convertir información legible en datos aparentemente aleatorios y sin sentido, mientras que la desencriptación es el proceso inverso que convierte los datos encriptados de vuelta a su forma original. Los términos asociados con la encriptación incluyen el texto llano o claro (información original), el criptograma o texto cifrado (resultado de la encriptación), y el algoritmo criptográfico o de cifrado (conjunto de pasos para encriptar y desencriptar). Este algoritmo se controla mediante una clave, que puede ser simétrica (misma clave para cifrar y descifrar) o asimétrica (claves diferentes).

**2.2.- Principios criptográficos.**

Los principios deseables de un sistema criptográfico, enunciados por Auguste Kerckhoffs, establecen que un sistema debe ser prácticamente irrompible, su efectividad no debe depender del secreto del diseño, la clave debe ser fácilmente memorizable, los criptogramas deben ser alfanuméricos, y el sistema debe ser operable y fácil de usar por una sola persona. El principio de Kerckhoffs, reformulado por Claude Shannon, señala que el adversario conoce el sistema, lo que implica que la seguridad se basa en el secreto de la clave y no del diseño.

La seguridad de un sistema criptográfico depende del diseño del algoritmo y la longitud de la clave utilizada. Cuanto más robusto sea el algoritmo y más larga sea la clave, mayor será la seguridad del sistema. En la actualidad, se emplean principalmente dos métodos criptográficos: el cifrado simétrico (clave privada) y el cifrado asimétrico (clave pública).

**2.3.- Criptografía de clave privada o simétrica.**

La criptografía de clave privada, conocida como simétrica, utiliza una clave secreta compartida entre el emisor y el receptor para cifrar y descifrar la información. Esta clave única se utiliza tanto para el cifrado como para el descifrado. Es eficaz para garantizar la confidencialidad de la información.

Principales características:

* **La clave es privada y conocida solo por las partes involucradas.**
* **Se emplea la misma clave para cifrar y descifrar los mensajes.**

Ventajas:

* **Los algoritmos de clave privada son rápidos y eficientes, ideales para cifrar grandes volúmenes de datos.**

Inconvenientes:

* **Distribución de claves: El receptor debe conocer la clave utilizada, lo que puede ser complicado ya que el emisor debe enviarla de manera segura.**
* **Gestión de claves: Si el grupo de comunicación es grande, se necesitaría una gran cantidad de claves diferentes para permitir la comunicación privada entre todos los miembros.**

Ejemplos de algoritmos de este tipo incluyen AES (Rijndael), DES, 3-DES, IDEA y RC5.

**2.4.- Criptografía de clave pública o asimétrica.**

La criptografía de clave pública, conocida como asimétrica, resuelve el problema de distribución de claves de la criptografía de clave privada al permitir que emisor y receptor acuerden una clave en canales inseguros. Aquí están los aspectos más importantes de este tipo de criptografía:

Características:

* Cada entidad posee un par de claves: una pública, conocida por todos, y una privada, conocida solo por su dueño.
* Las claves pública y privada son complementarias: lo que se cifra con una solo puede descifrarse con la otra.
* Las claves se generan en pares y son únicas, lo que garantiza su seguridad.
* Cifrar con la clave pública solo puede ser descifrado por la clave privada correspondiente.
* La clave pública no revela información sobre la clave privada ni permite descifrar mensajes cifrados con ella.
* Cifrar un mensaje con la clave privada sirve como prueba de autenticidad del remitente.

Ventajas:

* Elimina el problema de la distribución de claves, ya que cada entidad tiene su propio par de claves.

Inconvenientes:

* Son más lentos que los algoritmos simétricos.
* Requiere garantizar la autenticidad de la clave pública, lo que puede ser vulnerable al ataque del hombre en el medio.

Algunos ejemplos de algoritmos de clave pública son DSA, RSA (el estándar de facto), y el algoritmo de Diffie-Hellman.

Por lo general, se utiliza una combinación de ambos tipos de criptografía: asimétrica para establecer una clave de sesión privada, que luego se utiliza para cifrar los datos de forma simétrica.

**2.5.- Resumen de mensajes, firma digital y certificados digitales.**

Otras técnicas criptográficas incluyen:

1. Resumen de mensajes mediante funciones HASH: Transforman mensajes de longitud variable en resúmenes de longitud fija, asegurando que siempre producen la misma salida y son unidireccionales.
   * Utilizados principalmente para garantizar la integridad de los datos.
   * Cuando se combinan con técnicas de clave pública, proporcionan una forma eficiente de identificación.
   * Ejemplos de algoritmos **HASH: MD5 y SHA**.
2. Firmas digitales: Equivalentes digitales de firmas personales, basadas en criptografía de clave pública y funciones HASH.
   * Usadas para verificar la integridad y autenticidad de un mensaje.
   * El emisor codifica un mensaje con una función HASH, cifra el resultado con su clave privada y envía la firma digital junto con el mensaje.
   * El receptor utiliza la clave pública del emisor para verificar la firma y compararla con el mensaje decodificado.
3. Certificados digitales: Mensajes firmados por una entidad confiable para autenticar a las partes en una comunicación.
   * Resuelven problemas de confianza entre las partes mediante la delegación a una tercera entidad.
   * Deben contener información como número de serie, periodo de validez y firma digital del emisor, entre otros.
   * Introducen el concepto de Entidad Certificadora, responsable de validar los certificados y proporcionar mecanismos para su gestión, incluyendo revocación y suspensión.

**2.6.- Principales aplicaciones de la criptografía.**

Las principales aplicaciones de la criptografía son:

1. **Seguridad de las comunicaciones**: Permite establecer canales seguros en redes informáticas, incluso sobre redes no seguras. El uso de algoritmos de cifrado simétrico combinado con clave pública garantiza la privacidad sin comprometer la velocidad de transferencia.
2. **Identificación y autenticación**: Utiliza firmas digitales y otras técnicas criptográficas para identificar individuos o validar el acceso a recursos en entornos de red de manera más segura que los sistemas tradicionales de usuario y clave.
3. **Certificación**: Agentes fiables, como entidades certificadoras, validan la identidad de agentes desconocidos, como usuarios reales, mediante un sistema de certificación que utiliza la criptografía para identificar y autenticar a gran escala.
4. **Comercio electrónico**: Facilita el comercio electrónico al proporcionar canales seguros y mecanismos de identificación, reduciendo el riesgo de espionaje, fraudes y robos tanto para empresas como para usuarios.

**3.- Protocolos seguros de comunicaciones.**

Los protocolos criptográficos son herramientas clave para garantizar la seguridad en las comunicaciones en red. Algunos de los más destacados son:

1. **SSL (Secure Sockets Layer)**: Proporciona una comunicación segura entre cliente y servidor, protegiendo contra posibles ataques en la red, como el "hombre en el medio" (man-in-the-middle).
2. **TLS (Transport Layer Security)**: Es una evolución de SSL que amplía los algoritmos criptográficos utilizados para garantizar una comunicación segura.

Estos protocolos operan en una capa intermedia entre los protocolos de aplicación y los de transporte, como TCP o UDP, permitiendo el cifrado de protocolos de aplicación como Telnet, FTP, SMTP, IMAP y HTTP. Cuando un protocolo de aplicación se ejecuta sobre SSL o TLS, se convierte en su versión segura. Por ejemplo:

* **SSH (Secure Shell)**: Se utiliza en lugar de Telnet para comunicaciones seguras.
* **HTTPS (Hypertext Transfer Protocol Secure)**: Utilizado para comunicaciones seguras de hipertexto, es esencialmente HTTP sobre SSL o TLS, lo que lo hace un protocolo seguro para transferencia de datos en la web.

**3.1.- Protocolo criptográfico SSL/TLS.**

El protocolo SSL (Secure Sockets Layer) fue desarrollado por Netscape para asegurar las comunicaciones entre un navegador y un servidor web, aunque puede ser usado en otros tipos de conexiones cliente/servidor. Proporciona autenticación, confidencialidad e integridad de los datos transmitidos.

El funcionamiento del protocolo SSL implica varias fases:

1. **Negociación de algoritmos**: En esta fase, se acuerdan los algoritmos criptográficos a utilizar.
2. **Autenticación y clave de sesión**: Se intercambian claves y se autentica mediante certificados, utilizando criptografía asimétrica. Se genera una clave de sesión para cifrar los datos transmitidos de manera eficiente.
3. **Validación del canal seguro**: Se verifica que el canal de comunicación esté seguro.

Una vez completadas estas fases, se establece un canal seguro para la comunicación.

El protocolo SSL utiliza una combinación de diferentes algoritmos, incluyendo:

* Algoritmos de cifrado simétrico como DES, 3DES, RC2, RC4, e IDEA.
* Algoritmos de clave pública como RSA.
* Algoritmos de resumen como MD5 y SHA.
* Certificados como DSS y RSA.
* Clave de sesión diferente en cada transacción.

SSL emplea criptografía asimétrica en las fases de negociación y criptografía simétrica para la transmisión de datos. Es ampliamente utilizado en el comercio electrónico en Internet y en la creación de redes privadas virtuales (VPN).

**3.2.- Otros protocolos seguros.**

HTTPS es la versión segura de HTTP, utilizando SSL/TLS para cifrar y proteger la comunicación en línea. Se utiliza para garantizar seguridad en sitios web que manejan información sensible.

El protocolo SSH proporciona seguridad en el acceso remoto a computadoras, utilizando cifrado y operando en el puerto 22. Permite una comunicación segura entre el cliente y el servidor.

**4.- Criptografía en Java.**

Java proporciona APIs para la creación de aplicaciones con comunicaciones seguras, que incluyen:

* Encriptación de información con clave pública y privada.
* Firma digital y verificación.
* Resúmenes de mensajes (HASH).
* Certificados digitales y validación de certificados.
* Comunicaciones de red seguras con SSL.

Estas APIs permiten usar diversos algoritmos criptográficos (como DES, RSA, MD5) proporcionados por proveedores de servicios criptográficos (PSC) en Java.

La arquitectura criptográfica de Java incluye la Arquitectura Criptográfica de Java (JCA) y la Extensión Criptográfica de Java (JCE). A partir de JDK 1.4, JCE está integrado en el JDK y se considera parte de JCA. Los paquetes que conforman esta arquitectura son java.security y javax.crypto.

Además, otras bibliotecas disponibles en el JDK que utilizan la arquitectura proveedora de JCA son:

* Java Sockets Seguros (JSSE) para comunicaciones seguras.
* Servicio General de Seguridad a través de Kerberos (JGSS).
* Capa de autenticación y seguridad Java (SASL).
* Servicio de Autenticación y Autorización (JAAS) para autenticación y control de acceso.

**4.1.- Arquitectura criptográfica de Java.**

La arquitectura criptográfica de Java, conocida como JCA, se basa en los siguientes principios:

1. **Independencia de la aplicación:** Los programas no necesitan implementar algoritmos criptográficos directamente, sino que los solicitan a través de la plataforma Java a proveedores de servicios criptográficos (PSC). Varios proveedores pueden ser utilizados en la misma aplicación.
2. **Interoperabilidad:** Los diferentes proveedores son compatibles con todas las aplicaciones, lo que significa que una aplicación no está ligada a un proveedor específico y viceversa.
3. **Extensibilidad:** Se incluyen proveedores integrados que implementan un conjunto básico de servicios de seguridad, pero también se pueden instalar proveedores personalizados que ofrecen nuevos servicios.

La arquitectura JCA consta de dos componentes principales:

* Bibliotecas de clases e interfaces en el paquete java.security.
* Proveedores de servicios criptográficos que implementan algoritmos criptográficos, como Sun.

La extensión criptográfica de Java (JCE), que incluye el proveedor SunJCE, complementa la funcionalidad de JCA. Proporciona servicios adicionales a través de los paquetes javax.crypto, javax.crypto.spec y javax.crypto.interfaces.

A partir de JDK 1.4, la distinción entre JCA y JCE es menos clara ya que JCE está integrado en el JDK. La arquitectura criptográfica de Java incluye:

* Bibliotecas de clases e interfaces en java.security y javax.crypto.
* Proveedores de servicios criptográficos como Sun, SunRsaSign, SunJCE, que implementan algoritmos criptográficos.

Este enfoque de arquitectura de proveedores permite la coexistencia de múltiples implementaciones de algoritmos criptográficos en Java.

**4.2.- Proveedores y motores criptográficos.**

Un proveedor de servicios criptográficos es una entidad de seguridad que desarrolla sus propios servicios de seguridad en Java, implementando las clases e interfaces definidas en JCA/JCE y siguiendo el estándar definido en JCA.

Un motor criptográfico es el conjunto de clases e interfaces que debe implementar un proveedor de servicios criptográficos, que incluye clases como MessageDigest para resúmenes de mensajes, Signature para firmas digitales, KeyFactory para manejo de claves, KeyPairGenerator para generar claves públicas y privadas, y Cipher para encriptación y desencriptación de información.

El proveedor por defecto en JDK es el proveedor SUN, que ofrece implementaciones de algoritmos como DSA, MD5 y SHA-1, generación de claves públicas y privadas para DSA, factoría de claves para conversión de claves públicas a privadas, y construcción de certificados X.509.

Para utilizar más de un proveedor criptográfico en una aplicación, se puede modificar el archivo java.security ubicado en el directorio lib/security, donde se define cada proveedor con el formato security.provider.n=nombre\_de\_la\_clase. Esto permite seleccionar proveedores específicos al solicitar algoritmos criptográficos mediante el método getInstance(), ya sea del proveedor por defecto o de uno específico.

**4.3.- Gestión de claves con el paquete java.security.**

En la generación de claves, se utilizan números aleatorios seguros que se generan en base a una semilla. Esto permite crear algoritmos seguros, ya que sería muy difícil determinar los valores generados sin conocer la semilla.

El paquete java.security proporciona varias clases para la gestión de claves:

* La interfaz Key permite la representación, almacenamiento y envío de claves de forma serializada dentro de un sistema. Proporciona métodos como getAlgorithm(), getEncoded() y getFormat().
* La clase KeyPairGenerator permite la generación de claves públicas y privadas (asimétricas), produciendo objetos KeyPair que contienen una PublicKey y una PrivateKey.
* La clase KeyGenerator permite la generación de claves privadas (simétricas), produciendo objetos SecretKey.
* La clase SecureRandom permite generar números aleatorios seguros, con métodos como setSeed() y nextBytes().

Es importante tener en cuenta que el tamaño de las claves afecta al tiempo de cálculo y a la agilidad de la administración de claves en una aplicación.

Se pueden descargar proyectos Java que demuestran la generación de claves y muestran los valores obtenidos de la PrivateKey y la PublicKey.

**4.4.- Resúmenes de mensajes con la clase MessageDigest.**

Ejemplo de cómo crear un resumen de mensaje utilizando el algoritmo SHA-1:

**import java.security.MessageDigest;**

**import java.security.NoSuchAlgorithmException;**

**public class MessageDigestExample {**

**public static void main(String[] args) {**

**String message = "Este es un mensaje de prueba.";**

**try {**

**// Obtener una instancia del algoritmo de resumen SHA-1**

**MessageDigest digest = MessageDigest.getInstance("SHA-1");**

**// Actualizar el resumen con el mensaje**

**digest.update(message.getBytes());**

**// Completar la obtención del resumen**

**byte[] hashedMessage = digest.digest();**

**// Imprimir el resumen en formato hexadecimal**

**System.out.println("Resumen del mensaje (SHA-1): " + bytesToHex(hashedMessage));**

**} catch (NoSuchAlgorithmException e) {**

**System.err.println("El algoritmo de resumen SHA-1 no está disponible.");**

**e.printStackTrace();**

**}**

**}**

**// Método para convertir bytes a hexadecimal**

**private static String bytesToHex(byte[] bytes) {**

**StringBuilder result = new StringBuilder();**

**for (byte b : bytes) {**

**result.append(String.format("%02X", b));**

**}**

**return result.toString();**

**}**

**}**

Este código calculará el resumen del mensaje "Este es un mensaje de prueba." utilizando el algoritmo SHA-1 y mostrará el resumen resultante en formato hexadecimal.

* getInstance(): obtiene el algoritmo de resumen.
* update(): obtiene el resumen.
* digest(): completa la obtención del resumen.
* MD5. Genera una salida de 128 bits de longitud fija.
* SHA-1. Genera una salida de 160 bits.

**4.5.- Firma digital con la clase Signature de java.security.**

La clase Signature del paquete java.security en Java permite realizar firmas digitales y su verificación. Pasos necesarios para realizar la firma de un mensaje y luego verificarla:

1. **Generar las claves públicas y privadas utilizando la clase KeyPairGenerator:**
   * La PrivateKey se utiliza para firmar.
   * La PublicKey se utiliza para verificar la firma.
2. **Realizar la firma digital utilizando la clase Signature y un algoritmo asimétrico, como DSA:**
   * Crear un objeto Signature.
   * Llamar al método initSign() con la clave privada.
   * Utilizar el método update() para crear el resumen del mensaje.
   * Llamar al método sign() para obtener la firma digital.
3. **Verificar la firma creada utilizando la clave pública generada:**
   * Llamar al método initVerify() con la clave pública.
   * Utilizar update() para actualizar el resumen del mensaje y comprobar si coincide con el original.
   * Llamar al método verify() para realizar la verificación de la firma.

El algoritmo de firma digital DSA está implementado en el JDK de SUN y es parte del estándar de firmas digitales DSS. Con este algoritmo, se pueden utilizar los algoritmos de resumen MD5 y SHA-1.

**4.6.- Encriptación con la clase Cipher del paquete javax.crypto**

**Ver en el libro los ejemplos.**

**5.- Sockets seguros en Java (JSSE).**

JSSE (Extensión Java Sockets Seguros) es una biblioteca integrada en Java desde el JDK 1.4 que proporciona comunicaciones seguras utilizando el protocolo SSL (Secure Sockets Layer) a través del paquete javax.net.ssl. Utiliza la arquitectura de proveedores de JCA (Arquitectura Criptográfica de Java) para garantizar la seguridad en las comunicaciones.

Los sockets seguros, basados en el protocolo SSL, aseguran la autenticación, integridad y confidencialidad de los datos transmitidos en una comunicación. En Java, los usos más comunes de SSL se realizan utilizando las clases SSLSocket para programar sockets seguros del cliente y SSLServerSocket para programar sockets seguros del servidor. Aunque los sockets seguros funcionan de manera similar a los sockets normales, requieren cambios específicos en su proceso de creación e inicialización para garantizar la seguridad de la comunicación.

**5.1.- Programar un socket seguro de servidor.**

Para crear un socket seguro de servidor en Java, se utiliza el patrón de diseño Factory. Los pasos para programar un socket seguro de servidor son los siguientes:

1. Obtener un objeto SSLServerSocketFactory.
2. Crear un objeto SSLServerSocket especificando el puerto de escucha del servidor.
3. Esperar y aceptar conexiones entrantes de clientes.
4. Crear un canal seguro sobre el socket aceptado.

A continuación, se muestra un ejemplo de código que sigue estos pasos para establecer un socket seguro del servidor que escucha en el puerto 5000:

**import javax.net.ssl.\*;**

**public class SecureServerSocketExample {**

**public static void main(String[] args) {**

**try {**

**// Paso 1: Obtener un objeto SSLServerSocketFactory**

**SSLServerSocketFactory serverSocketFactory =**

**(SSLServerSocketFactory) SSLServerSocketFactory.getDefault();**

**// Paso 2: Crear un objeto SSLServerSocket especificando el puerto de escucha**

**SSLServerSocket serverSocket =**

**(SSLServerSocket) serverSocketFactory.createServerSocket(5000);**

**// Paso 3: Esperar y aceptar conexiones entrantes de clientes**

**System.out.println("Servidor seguro esperando por conexiones...");**

**SSLSocket clientSocket = (SSLSocket) serverSocket.accept();**

**// Paso 4: Crear un canal seguro sobre el socket aceptado**

**// Aquí se realizarían operaciones de lectura y escritura seguras**

**// Cerrar el socket seguro del servidor**

**serverSocket.close();**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

**}**

**}**

Este código establece un socket seguro del servidor que escucha en el puerto 5000 y espera conexiones entrantes de clientes. Una vez que se acepta una conexión, se crea un canal seguro sobre el socket aceptado para realizar operaciones de lectura y escritura seguras.

**5.2.- Programar un socket seguro cliente.**

Crear un socket seguro cliente en Java:

1. Obtener un objeto SSLSocketFactory.
2. Crear un objeto SSLSocket especificando el nombre del servidor y el puerto de escucha.
3. Establecer un canal seguro de comunicación con el servidor.

A continuación, se muestra un ejemplo de código que sigue estos pasos para crear un socket seguro cliente que se conectará al servidor en localhost y puerto 5000:

**import javax.net.ssl.\*;**

**public class SecureClientSocketExample {**

**public static void main(String[] args) {**

**try {**

**// Paso 1: Obtener un objeto SSLSocketFactory**

**SSLSocketFactory socketFactory =**

**(SSLSocketFactory) SSLSocketFactory.getDefault();**

**// Paso 2: Crear un objeto SSLSocket especificando el nombre del servidor y el puerto**

**SSLSocket clientSocket =**

**(SSLSocket) socketFactory.createSocket("localhost", 5000);**

**// Paso 3: Establecer un canal seguro de comunicación con el servidor**

**// Aquí se realizarían operaciones de lectura y escritura seguras**

**// Cerrar el socket seguro del cliente**

**clientSocket.close();**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

**}**

**}**

Este código establece un socket seguro cliente que se conectará al servidor en localhost y puerto 5000. Una vez establecida la conexión, se puede utilizar el canal seguro para realizar operaciones de lectura y escritura seguras.

**5.3.- Ejemplos de aplicaciones con comunicaciones seguras.**

**Ver temario.**