T2-PSP.pdf

bloodyraintatii

Programación de servicios y procesos

1º Desarrollo de Aplicaciones Multiplataforma

Estudios España

Accede al documento original

Programación de servicios y procesos

Tema 9: Protocolos nivel de aplicación (teoría)

1. Introducción a los servicios en red

→ Las redes de ordenadores son sistemas interconectados para compartir información y recursos.

→ Sus principales ventajas incluyen:

- Reducción de costes en hardware y software.
- Poder crear grupos de trabajo para organizar a los usuarios.
- Mejora en la administración de equipos y aplicaciones.
- Mayor integridad y seguridad de datos
- Comunicación más eficiente.

→ Los servicios en red se clasifican en:

- Servicios de administración/configuración → facilitan la administración y la gestión de configuración de los equipos. Ej.: DHCP y DNS.
- Servicios de acceso remoto → podremos gestionar conexiones remotas. Ej.: Telnet y SSH.
- Servicios de ficheros → podremos ofrecer grandes cantidades de almacenamiento. Ej.: FTP.
- Servicios de impresión → podremos imprimir documentos remotamente.
- Servicios de información → podremos obtener ficheros en función de su contenido. Ej.: HTTP.
- Servicios de comunicación → los usuarios podrán comunicarse a través de mensajes. Ej.: SMTP.

2. Protocolo a nivel de aplicación

→ Los protocolos más importantes están en el modelo TCP/IP, específicamente en la capa de Aplicación.

→ Los principales protocolos son:

- FTP → transferencia de ficheros (*File Transfer Protocol*).
- Telnet → acceso remoto a otro ordenador.
- SMTP → envío de correos electrónicos.
- HTTP → navegación web.
- SSH → gestión remota segura de otro ordenador.
- NNTP → envío de noticias (*Network News Transport Protocol*).
- IRC → chat por Internet.
- DNS → resolución de direcciones.

3. El protocolo DNS

→ DNS (Domain Name System) → traduce direcciones IP a nombres de dominio y viceversa.

→ Permite que varios nombres compartan una IP, que varias IP compartan nombres, y que un nodo cambie de nombre sin cambiar su IP.

→ Usa una base de datos distribuida para almacenar información de dominios.

4. El protocolo FTP

→ Permite intercambiar ficheros entre ordenadores remotos a alta velocidad.
→ Transmite en texto plano, lo que afecta la seguridad pero mejora la velocidad.
→ SFTP (usado junto a SSH) soluciona el problema de seguridad mediante encriptación.
→ Usa los puertos 20 y 21 (modo pasivo y modo pasivo y activo). La conexión al 20 finaliza cuando se completa la transferencia de datos. El 21 siempre está activo.
→ Características del protocolo FTP:

- Permite conectar usuarios en remoto al servidor FTP.
- Se puede implementar un sistema de privilegios para limitar el acceso a los usuarios.
- Dos modos de conexión → modo activo (con 2 conexiones distintas) y modo pasivo (sin ellas).

5. Los protocolos SMTP, POP3 e IMAP

→ SMTP (*Simple Mail Transfer Protocol*) sigue el modelo Cliente/Servidor.
→ SMTP maneja el envío de correos electrónicos, permitiendo enviar cualquier tipo de documento digitalizado.
→ Usa el puerto 25.
→ Dos formas de acceder a los mensajes:

- POP3 → permite descargar mensajes localmente.
- IMAP → permite consultar mensajes directamente en el servidor.

→ Se usan comandos para la transferencia de correos, como *HELO* (para abrir sesión con el servidor) o *MAIL FROM* (para indicar el emisor del correo).

6. El protocolo HTTP

→ HTTP (*HyperText Transfer Protocol*) → protocolo para navegar por Internet que transfiere páginas HTML.
→ Reglas del protocolo HTML:

- Sigue un modelo petición-respuesta.
- Usa el puerto 80 aunque se puede cambiar.
- Se le llama *agente* (*user agent*) al cliente.
- Se le llama *recursos* a toda la información transmitida, identificada mediante una URL. Ej.: http://www.google.es
- Los recursos también pueden ser: ficheros, consulta a una bbdd, resultado de una operación realizada por un programa, etc.

→ No tiene estado (no recuerda conexiones anteriores).
→ El proceso básico es:

1. Acceso a una URL, pudiendo indicar el puerto en la misma.
2. Descomposición de información, diferenciando todas sus partes (nombre de dominio, IP, puerto…)
3. Conexión con el servidor.
4. Respuesta al cliente con código HTML.
5. Visualización de HTML y cierre de conexión.

Tema 10: Los servicios en red. Sockets I (teoría/práctica)

1. Definición de Sockets

→ Sockets → mecanismos de comunicación entre aplicaciones en red, representando los extremos de una conexión bidireccional. Hay dos tipos:

- Sockets TCP → orientados a conexión:
  - Usan el protocolo TCP.
  - La conexión no comienza hasta que los dos Sockets estén conectados.
  - Transmiten en bytes.
  - Clases en Java → *Socket* (cliente que transmite y recibe datos) y *ServerSocket* (servidor que espera la solicitud de un cliente).
- Sockets UDP → no orientados a conexión:
  - Usan el protocolo TCP.
  - Transmiten en datagramas.
  - Más rápidos pero menos seguros.
  - Clases en Java → usamos *DatagramSocket*.

2. Programación de un servidor TCP

→ El uso de Sockets en Java requiere:

- Crear o abrir los Sockets.
- Crear los flujos de entrada o salida.
- Cerrar los flujos y Sockets.

→ Se debe llevar el control de excepciones con un *try-catch*.

→ Ejemplo: la clase ServerSocket escucha en un puerto específico y acepta conexiones de clientes mediante *accept()*. Crea los flujos de entrada (con *readUTF()*) y salida (con *writeUTF()*). Se cierra el Socket.

3. Programación de un cliente TCP

→ Para crear un cliente debemos:

- Crear un Socket y conectar con el servidor mediante la IP y el puerto.
- Crear flujos de entrada y salida.
- Cerrar los flujos y los Socket.

→ Ejemplo: creamos un Socket conectándose a IP y puerto del servidor, establecemos flujos entrada (con *readUTF()*) y salida (con *writeUTF()*) para la comunicación y cerramos las conexiones al terminar.

4. Clases Datagrampacket y Datagramsocket

→ La clase *DatagramPacket* representa datagramas UDP (arrays de bytes con dirección IP y puerto UDP concreto).

→ Forma de un datagrama:

Array de bytes	Longitud	IP destino	Puerto destino

→ Métodos de *DatagramPacket*:
- *getData()* → devuelve el array de bytes que contiene los datos.
- *getLength()* → 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.

→ Métodos de *DatagramSocket*:
- *send(datagrama)* → enviar de datagramas.
- *receive(datagrama)* → recibir datagramas.
- *close()* → cerrar el Socket.
- *getLocalPort()* → devuelve el puerto donde está conectado del Socket.
- *getPort()* → devuelve el puerto de donde procede el Socket.

5. Programación de un servidor UDP

→ No requiere conexión establecida → la diferenciación entre clientes y servidores será más complicada, considerando como cliente al que inicia la comunicación.
→ Se deben gestionar los errores con *try-catch*.
→ Ejemplo: el servidor *ServidorUDP* espera mensajes y responde usando *DatagramSocket* y *DatagramPacket*.
→ No se debe cerrar ninguna conexión porque no existe.
→ Es más simple que TCP pero menos confiable.

Sincronización Cliente/Servidor TCP o UDP (CASO PRÁCTICO)

→ Objetivo → crear una aplicación cliente/servidor que permita sincronizar el tiempo de conexión entre ambos, registrando cuánto tiempo permanece conectado cada cliente al servidor.

→ Pasos:

1. Configuración Inicial:
    a. Seleccionar el protocolo (TCP o UDP) según necesidades de fiabilidad.
    b. Definir el puerto de comunicación.
    c. Establecer el formato de los mensajes de sincronización.
    d. Determinar el sistema de identificación de clientes.
2. Diseño del Servidor:
    a. Crear el sistema de escucha de conexiones.
    b. Implementar sistema de registro de tiempos.
    c. Establecer estructura para almacenar conexiones activas.
    d. Definir mecanismos para gestión de múltiples clientes.
3. Diseño del Cliente:
    a. Establecer sistema de conexión al servidor.
    b. Crear proceso de solicitud de sincronización.
    c. Implementar sistema de notificación de desconexión.
    d. Desarrollar interfaz de información temporal.
4. Protocolo de Sincronización:
    a. Mensaje inicial de solicitud de conexión.
    b. Respuesta del servidor con marca temporal.
    c. Sistema de confirmación de sincronización.
    d. Protocolo de desconexión ordenada.
5. Sistema de Control:
    a. Registro de tiempos de inicio de conexión.
    b. Monitorización de conexiones activas.
    c. Cálculo de duraciones de conexión.
    d. Almacenamiento de estadísticas.
6. Gestión de Errores:
    a. Sistema de detección de desconexiones inesperadas.
    b. Protocolo de reconexión automática.
    c. Registro de incidencias.
    d. Recuperación de sesiones interrumpidas.
7. Verificación y Pruebas:
    a. Comprobación de sincronización individual.
    b. Pruebas de múltiples conexiones simultáneas.
    c. Validación de registros temporales.
    d. Tests de recuperación ante fallos.

Descansito &Coins

Gemini + 

¿Sabrías identificar en qué te puede ayudar Google Gemini para estudiar?

REGLAS

1. Observa las opciones disponibles
2. Responde como Gemini te ayuda a estudiar.
3. Gana Wuolah coins para descargas sin publi.

Fácil | 10

A. Sintetiza horas de investigación en minutos.

B. Convierte tus apuntes en un esquema visual.

C. Prepara un examen para autoevaluarte.

D. Convierte tus apuntes en podcasts.

E. Sube hasta 1.500 páginas y analiza textos largos.

F. Todas las anteriores.

Gemini

Tema 11: Los servicios en red. Sockets I (teoría/práctica)

1. Programación de un cliente HTTP

→ El protocolo HTTP o HTTPS se basa en un proceso sencillo de solicitudes y respuestas entre cliente y servidor.

1. El cliente establece conexión con un servidor enviando un mensaje de solicitud.
2. El servidor responde con un mensaje similar que contiene el resultado de la operación solicitada.

→ Java proporciona dos clases principales para programar aplicaciones HTTP:

- La clase *URL* → permite representar direcciones de páginas web (como https://www.google.es) para realizar operaciones con ellas.
- La clase *URLConnection* → permite realizar operaciones con la dirección web creada mediante URL, como lanzar operaciones GET y obtener respuestas fácilmente.

→ Al utilizar estas clases se programa un servicio HTTP de alto nivel, lo que significa que las operaciones de comunicación no se visualizan, simplificando la programación como con *sockets*.

2. Programación de un cliente FTP

→ El protocolo FTP (File Transfer Protocol) se utiliza para transferir ficheros entre servidor y cliente o viceversa.

→ Los pasos para crear un cliente FTP son:

1. Realizar conexión al servidor.
2. Comprobar que la conexión se ha realizado con éxito.
3. Validar el usuario FTP conectado. Si es inválido, debemos abortar la conexión y enviar un mensaje de error.
4. Realizar operaciones con el servidor.
5. Desconectar del servidor al terminar.

→ Este proceso puede generar excepciones *SocketException* e *IOException*.

→ Java no dispone de clases específicas para FTP, pero la fundación Apache creó la API *org.apache.commons.net.ftp* con clases como:

- Clase FTP → hereda de *SocketCliente* proporciona funcionalidades básicas para clientes FTP.
- Clase *FTPReplay* → permite operar con valores devueltos por consultas FTP.
- Clase *FTPClient* → hereda de *SocketCliente* y da soporte a funcionalidades del cliente FTP.
- Clase *FTPClientConfig* → permite configurar clientes FTP fácilmente.
- Clase *FTPSClient* → hereda de *FTPClient*, usa el protocolo SSL y permite utilizar el protocolo FTPS (versión segura de FTP).

3. Programación de un cliente Telnet

→ El protocolo Telnet (TELecomunication NETwork) permite acceder a otros equipos conectados a la red para administrarlos remotamente como si estuviéramos frente al equipo. Facilita la administración de equipos sin pantalla ni teclado.

→ Características del protocolo Telnet:
- Utiliza el esquema cliente/servidor.
- Usa el puerto 23.
- Funciona con comandos en modo texto.

→ No ofrece mucha seguridad ya que toda la información se transmite en texto plano (incluyendo contraseñas), lo que limita su uso en grandes empresas.
→ Apache ofrece el paquete *org.apache.commons.net.telnet* con la clase *TelnetClient* para implementar este protocolo. Los métodos más útiles son:
- *SocketClient.connect()* → permite conectar al servidor.
- *TelnetClient.getInputStream()* y *TelnetClient.getOutputStream()* → obtienen flujos de comunicación.
- *TelnetClient.disconnect()* → permite desconectar.

4. Programación de un cliente SMTP
→ El protocolo SMTP se utiliza para enviar y recibir correos electrónicos.
→ Java proporciona la API *javax.mail* con clases que facilitan la gestión de correos:
- Clase *Session* → representa una sesión de usuario para correo electrónico, agrupando toda la configuración por defecto. El método *getDefaultInstance()* obtiene una sesión predeterminada.
- Clase *Message* → representa un mensaje de correo electrónico. Permite configurar remitente (*setFrom()*), asunto (*setSubject()*) y texto (*setText()*).
- Clase *Transport* → representa el envío de correos electrónicos. Hereda de la clase *Service*, que proporciona funcionalidades comunes a todos los servicios de mensajería.

→ Con estas clases se puede crear fácilmente una aplicación para enviar correos electrónicos, por ejemplo, mediante una cuenta de Gmail.

5. Socket e hilos
→ Los servidores implementados con sockets TCP y UDP tienen la limitación de atender solo a un cliente a la vez, formando una cola de espera para los demás clientes. Esto es contraproducente para cualquier servicio, siendo ideal que cada servidor atienda a muchos clientes simultáneamente.
→ La concurrencia se logra usando hebras o hilos, lanzando una hebra por cada tarea concurrente, dando la sensación de ejecución simultánea. Las hebras se ejecutan en la parte del servidor, de forma que el cliente no percibe si lo atiende el servidor o una hebra.
→ Para crear un programa cliente/servidor concurrente con TCP mediante hilos, se necesitan las siguientes clases:
- Clase Servidor → representa el servidor de la aplicación.
- Clase ServidorHebra → representa una hebra lanzada en el servidor para atender al cliente conectado.
- Clase Cliente → representa un cliente de la aplicación.

→ En el *main()* de la clase servidor creamos un *ServerSocket* para esperar a un cliente. Cuando el servidor detecta una petición de cliente, la acepta, procesa y crea una hebra para atenderla. Cada cliente es atendido por una hebra diferente, permitiendo al servidor volver a escuchar nuevas peticiones inmediatamente.

Realizar un Cliente/Servidor Telnet (CASO PRÁCTICO)

1. Establecimiento de conexión:
    a. El cliente inicia una conexión TCP hacia el puerto 23 del servidor.
    b. El servidor acepta la conexión e inicia un proceso de negociación.
2. Negociación de opciones:
    a. Ambas partes intercambian información sobre capacidades (modo eco, tipo de terminal, etc.).
    b. Se establecen los parámetros de la sesión.
3. Autenticación:
    a. El servidor solicita credenciales al cliente.
    b. El cliente envía nombre de usuario y contraseña (en texto plano, lo que representa una debilidad de seguridad).
4. Sesión interactiva:
    a. El servidor proporciona un intérprete de comandos.
    b. El cliente envía comandos que se ejecutan remotamente.
    c. El servidor devuelve los resultados al cliente.
5. Cierre de conexión:
    a. Cualquier parte puede finalizar la conexión TCP.
    b. Se liberan los recursos asignados a la sesión.

Realizar un Cliente/Servidor SMTP (CASO PRÁCTICO)

1. Establecimiento de conexión:
    a. El cliente inicia una conexión TCP hacia el puerto 25 del servidor
    b. El servidor responde con un mensaje de bienvenida (código 220)
2. Saludo e identificación:
    a. El cliente envía el comando HELO/EHLO seguido de su nombre de dominio
    b. El servidor responde con sus capacidades (código 250)
3. Transacción de correo:
    a. El cliente envía MAIL FROM: indicando el remitente
    b. El servidor acepta (código 250)
    c. El cliente envía RCPT TO: para cada destinatario
    d. El servidor acepta cada destinatario válido (código 250)
4. Transferencia de datos:
    a. El cliente envía el comando DATA
    b. El servidor indica que está listo (código 354)
    c. El cliente envía el contenido del mensaje (encabezados y cuerpo)
    d. Finaliza con una línea que contiene solo un punto "."
    e. El servidor confirma la recepción (código 250)
5. Finalización:
    a. El cliente puede iniciar otra transacción o terminar con QUIT
    b. El servidor confirma (código 221) y cierra la conexión

Tema 12: El servicio web (teoría/práctica)

1. ¿Qué es un servicio web?

→ Un servicio web (web service) es un programa que proporciona comunicación entre aplicaciones software ejecutándose en distintas plataformas. Lo forman componentes de aplicaciones distribuidas disponibles externamente, accesibles mediante protocolos web estándar, usando lenguajes de programación independientes de la plataforma para el intercambio de mensajes.

→ Características de los servicios web:

- Accesibles a través de la web mediante el protocolo HTTP y lenguaje XML.
- Utilizan XML como formato de intercambio de datos independiente de plataformas, permitiendo la comunicación entre ambos.
- Realizan funciones bien definidas y contienen una descripción de sí mismos.
- Son localizables mediante mecanismos que permiten encontrarlos.
- Son componentes independientes integrables en sistemas distribuidos complejos.
- Ofrecen interoperabilidad mediante HTTP y XML, permitiendo que distintas aplicaciones los utilicen.

2. Arquitectura orientada a servicios: SOA

→ La arquitectura SOA (*Service Oriented Architecture*) es una tecnología para diseñar aplicaciones basadas en peticiones a un servicio. Permite crear elementos software reutilizables e independientes del lenguaje con que fueron creados.

→ Ha dado lugar al Software as a Service (SaaS), donde las aplicaciones no se instalan en el ordenador del cliente sino en un servidor que recibe peticiones de los clientes, disponible desde cualquier punto con acceso a Internet.

→ Los servicios web son el punto fuerte de SOA, que trabaja con características que permiten ejecutarlos perfectamente. Existen dos tipos:

1. Arquitectura Orientada a Servicios Tradicional (SOA tradicional):
    - Utiliza principios y tecnologías básicas de servicios web.
    - Puede usar SOAP, WSDL y UDDI.
    - No incluye características como seguridad, transaccionalidad, garantía de entrega, direccionamiento.
2. Arquitectura Orientada a Servicios de segunda generación:
    - Amplía la funcionalidad añadiendo características que mejoran la calidad del servicio.
    - Mejora política de seguridad, transacción, gestión, etc.
    - Recibe actualizaciones frecuentes por la aparición de nuevos fallos de seguridad y formas de realizar transacciones.

→ SOA es el modelo de la arquitectura, mientras que SOAP (Simple Object Access Protocol) es una forma de comunicación (protocolo) permitida en SOA.

Tema 13: La programación segura (teoría)

1. Introducción a la seguridad

→ La seguridad absoluta es imposible en informática. Un sistema informático puede considerarse seguro cuando cumple las siguientes características:

- Confidencialidad → solo personas autorizadas acceden al sistema.
- Integridad → solo personas autorizadas pueden modificar la información.
- No repudio → un usuario no puede negar haber enviado un mensaje.
- Disponibilidad → todos los recursos están disponibles para usuarios autorizados.

→ Las empresas deben tener políticas de seguridad firmemente establecidas.

→ Organismos oficiales a nivel mundial, como INCIBE (España) y el Computer Emergency Response Team Coordination Center, supervisan la seguridad informática.

2. Amenazas de seguridad

→ Amenaza de seguridad → condición del entorno que puede violar la seguridad del sistema.

→ Tipos de amenazas:

- Tipos por origen → conectar un sistema a cualquier entorno.
- Tipos por efecto → robos de información, anulación de sistemas, suplantación de identidad, etc.
- Tipos por medio utilizado → virus, ingeniería social, ataques de denegación de servicio, phishing, etc.

→ Cuatro principales grupos de amenazas:

- Interrupción → destruyen recursos del sistema.
- Intercepción → acceden a recursos ajenos.
- Modificación → acceden y alteran recursos ajenos.
- Fabricación → insertan datos falsos en los originales.

3. Ataques a un sistema informático

→ Los ciberataques se clasifican principalmente en:

1. Ataques pasivos → el atacante escucha o monitoriza el tráfico sin alterar la comunicación. Obtiene credenciales, páginas visitadas, etc.
2. Ataques activos → el atacante modifica los datos transmitidos o crea flujos falsos. Objetivos: suplantación de identidad, reactuación (reenvío de mensajes), denegación de servicio.

4. Vulnerabilidades en el software

→ Vulnerabilidades en el software → fallos o huecos en la seguridad que permiten accesos no autorizados.

→ Los problemas principales:

- Dificultad para detectarlas.
- Imposibilidad de encontrar todos los fallos antes de lanzar la aplicación.
- Necesidad de monitoreo continuo y actualizaciones.

En Java, la seguridad se basa en dos pilares:

1. Seguridad interna → criterios de tratamiento de errores (try-catch-finally).
2. Políticas de acceso → permisos de la aplicación para usar recursos del sistema.

5. Mecanismos de control de acceso

→ Políticas de seguridad → documentos de alto nivel fundamentales para la ciberseguridad.
→ Contienen:

- Definición de seguridad de la información.
- Medidas a adoptar por empresa, trabajadores y departamento de sistemas.

→ Deben complementarse con objetivos de seguridad y procesos adicionales y ser accesibles para todo el personal.

6. Validación de entradas

→ Las entradas de usuarios pueden causar fallos de seguridad como *buffer overflow* (se desborda el tamaño de una determinada variable).
→ La validación mediante expresiones regulares permite:

1. Mantener la consistencia de los datos.
2. Evitar desbordamientos de memoria.

→ Java usa la biblioteca *java.util.regex* con la clase *Pattern* para definir expresiones regulares que validan entradas.

Elemento	Comentario	Operador	Comentario
x	El carácter x.	[a-z]{2}	Dos letras minúsculas.
[abc]	Los caracteres a, b o c.	[a-z]{2,5}	Entre dos y cinco letras minúsculas.
[a-z]	Una letra minúscula.	[a-z]{2,}	Más de 2 letras minúsculas.
[A-Z]	Una letra mayúscula.	hola|adios	Solo se pueden introducir las palabras hola o adiós. | funciona como la operación OR.
[a-Za-z]	Una letra minúscula o mayúscula.	XY	Solo se pueden introducir dos expresiones. Esta es la operación AND.
[0-9]	Un número entre 0 y 9.	e(n|l)campo	Los delimitadores ( ) permiten hacer expresiones más complejas. En el ejemplo se debe introducir el texto "en campo" o "el campo".
[A-Za-z0-9]	Una letra minúscula, mayúscula o un número.		

Tema 14: La criptografía (teoría/práctica)

1. Concepto de criptografía

→ Criptografía → proviene del griego "cripto" (secreto) y "grafía" (escritura), significando "escritura secreta".

→ Fue creada para enviar información confidencial o mensajes privados a personas u organizaciones específicas.

→ Pasos para aplicar la criptografía:

- Escribir el mensaje normal.
- Usar técnicas de encriptado para codificarlo.
- Enviar el mensaje encriptado por una línea de comunicaciones seguras o no seguras. Si el mensaje es interceptado, no correría peligro si se desconoce el método de encriptación.
- El receptor aplica técnicas de desencriptado para ver el mensaje original.

→ Criptoanálisis → ciencia dedicada a estudiar la fortaleza de los sistemas criptográficos y mejorarlos continuamente.

→ Tipos de criptografía → simétrica, asimétrica (de clave pública), con umbral, basada en identidad, basada en certificados, sin certificados y de clave aislada.

→ Los más utilizados profesionalmente son la criptografía simétrica y asimétrica.

2. Aplicaciones de la criptografía

→ Una aplicación emergente es la cadena de bloques (blockchain) → utiliza diferentes tipos de criptografía para garantizar la seguridad de las transacciones.

→ Emplea criptografía HASH para convertir grandes cantidades de información en combinaciones únicas de letras y números difíciles de imitar, permitiendo resumir información y crear claves públicas y privadas para el manejo de criptomonedas.

→ En el uso cotidiano, la criptografía se aplica en Internet cuando accedemos a sitios web HTTPS, que utilizan el protocolo de seguridad SSL (Secure Sockets Layer).

→ SSL → protocolo que cifra los datos enviados al servidor mediante algoritmos criptográficos.

→ Otro ejemplo práctico es su uso en sistemas financieros virtuales como PayPal, donde se protegen las transacciones y datos personales.

3. Encriptación de la información

→ Encriptación o cifrado de información → proceso mediante el cual los datos son codificados para producir un texto aparentemente aleatorio o sin sentido.

→ Desencriptación → operación inversa que transforma los datos encriptados en el texto original mediante las técnicas inversas del algoritmo utilizado.

→ Conceptos básicos

- Texto plano → original sin encriptar.
- Texto cifrado → resultado de aplicar encriptación.
- Algoritmo de cifrado/algoritmo criptográfico → método para encriptar el texto plano.
- Clave → cadena de caracteres base para el algoritmo de cifrado.

→ Fórmula del proceso de encriptado → Cifrado→$F_K(M)=C$, donde F es el algoritmo, K la clave de cifrado, M el mensaje original y C el texto cifrado.

4. Criptografía de clave privada o simétrica

→ Criptografía de clave simétrica o privada → método que usa una única clave secreta conocida solo por el emisor y receptor para garantizar la confidencialidad. Se denomina simétrica porque la misma clave se usa tanto para encriptar como para desencriptar.

→ Principales características:

- La clave es secreta y conocida solo por las partes comunicantes.
- Se utiliza la misma clave para cifrar y descifrar.
- Los algoritmos son rápidos y no aumentan significativamente el tamaño del mensaje, haciéndolos ideales para grandes cantidades de texto.

→ Inconvenientes:

- Posibilidad de que el mensaje sea interceptado por alguien no deseado.
- Necesidad de una clave diferente para cada combinación de comunicantes, generando una gran cantidad de claves.

→ Soluciones → criptografía asimétrica y criptografía híbrida.

5. Criptografía de clave pública o simétrica

→ Criptografía de clave pública o asimétrica → surgió para solucionar el problema de distribución de claves de la criptografía simétrica, permitiendo a emisor y receptor compartir claves incluso por canales no seguros. Se denomina asimétrica porque utiliza claves diferentes para encriptar y desencriptar.

→ Características principales:

- Cada participante tiene un par de claves (una pública conocida por todos y una privada conocida solo por su poseedor).
- Las parejas de claves son complementarias y funcionan únicamente entre sí.
- Las claves solo pueden generarse una vez.
- Los mensajes cifrados con clave pública solo pueden descifrarse con la clave privada correspondiente.
- Cifrar con clave privada demuestra la autoría del mensaje.

→ Ventaja → eliminar el problema de distribución de claves.

→ Inconvenientes:

- Los algoritmos son más lentos.
- Se requiere garantizar que la clave pública pertenece realmente a quien dice ser.

→ Lo óptimo es utilizar una combinación de ambas criptografías → criptografía híbrida.

6. Firma digital y certificados digitales

→ Las firmas digitales son el equivalente tecnológico de las firmas personales, identificando inequívocamente al firmante en entornos digitales.

→ Están basadas en criptografía de clave pública y resumen de mensajes HASH.

→ Resumen de mensajes HASH → algoritmos que convierten mensajes de longitud variable en resúmenes de longitud fija. Ejemplos: MD5 y SHA.

→ Certificados digitales → documentos electrónicos firmados por una entidad certificadora que valida los datos de la firma digital, funcionando como un notario digital.

→ La entidad certificadora es una organización responsable de verificar la veracidad de los datos de los firmantes, emitir certificados, y proporcionar mecanismos para revocarlos, suspenderlos o comprobar su validez.

→ En Java se usa la clase *MessageDigest* con algoritmos como MD2, MD5, SHA-1, SHA-256, SHA-384 y SHA-512 para implementar estas funcionalidades, controlando excepciones como *NoSuchAlgorithmException* mediante bloques try-catch.

Tema 15: Comunicaciones seguras (teoría/práctica)

1. Protocolos seguros de comunicaciones

→ La protección de la información en aplicaciones con comunicaciones en red es fundamental, ya que estas pueden ser interceptadas y manipuladas.

→ Los protocolos seguros de comunicación (=protocolos criptográficos o de encriptación) combinan la criptografía con las comunicaciones en red para asegurar la información.

→ Los dos protocolos más comunes son:

1. Protocolo SSL → proporciona comunicación segura en el modelo cliente/servidor, protegiendo contra ataques como "*man in the middle*", donde un tercero espía el tráfico para acceder a información confidencial.
2. Protocolo TLS → evolución del SSL que permite utilizar más algoritmos criptográficos para codificar la información enviada.

→ Estos protocolos se encuentran entre las capas de aplicación y de transporte del modelo TCP/IP, permitiendo cifrar información en protocolos como Telnet, IMAP, FTP o HTTP.

→ Cuando se aplican estos protocolos de encriptación sobre protocolos de comunicación, se obtienen versiones seguras: FTPS (de FTP), HTTPS (de HTTP) y SSH (de Telnet).

2. Características SSL/TLS

→ Protocolo SSL (Secure Sockets Layer) → creado por Netscape para hacer seguras las comunicaciones entre navegadores web y servidores, aunque puede utilizarse en cualquier aplicación cliente/servidor.

→ Propiedades fundamentales:

- Autenticación.
- Confidencialidad.
- Integridad.

→ El funcionamiento de SSL requiere una negociación previa ("*handshake*") de parámetros para establecer la comunicación segura.

→ Existe también el SSL/TLS Record Protocol que especifica cómo se encapsulan los datos a transmitir.

→ Fases del protocolo SSL:

1. Fase inicial → negociación de algoritmos criptográficos para la comunicación.
2. Fase de autenticación → intercambio de claves y autenticación mediante certificados de criptografía asimétrica.
3. Fase de verificación → comprobación de que el canal es seguro.
4. Inicio de la comunicación segura.

→ Si falla la negociación, la comunicación no se establece.

→ Algoritmos criptográficos usados con SSL/TLS:

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

3. Encriptación de datos con Cipher

→ Clase *Cipher* de Java → para realizar codificaciones de datos.

→ Uno de los algoritmos disponibles es el cifrado AES (Advanced Encryption Standard), que es un cifrado por bloques utilizado inicialmente en Estados Unidos, aunque se admiten muchos modos de operación más.

→ 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 de este algoritmo se realizan en un campo finito determinado.

→ Para utilizar AES en Java, es necesario descargar e integrar la biblioteca *commons-codec* de Apache. Una vez descargada e integrada en el proyecto, se puede implementar la encriptación de mensajes mediante este algoritmo utilizando la clase Cipher.

4. Certificados para sockets seguros

→ Java permite crear versiones seguras de sockets para establecer comunicaciones cifradas en el modelo cliente/servidor.

→ Para utilizar sockets seguros, primero es necesario crear los certificados que ayudarán a encriptar la información.

El proceso para crear certificados seguros incluye:

1. Crear el certificado del servidor usando la herramienta keytool (parte del JDK) en la ventana de comandos.
2. Exportar el certificado del servidor a un fichero.
3. Crear el certificado del cliente.
4. Introducir la clave pública del cliente en los certificados de confianza del servidor.

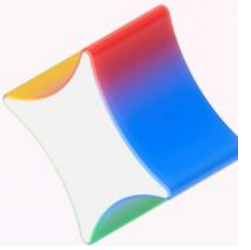
→ Estos certificados contendrán información personal o de organización y establecerán una base de confianza entre cliente y servidor para la comunicación segura.

Práctica de Criptografía (CASO PRÁCTICO)

→ Objetivo: establecer un canal seguro para transmitir información financiera entre dos aplicaciones bancarias.

→ Pasos:

1. Preparación → importar las bibliotecas necesarias (javax.crypto) y definir las claves a utilizar en el cifrado.
2. Generación de claves → crear un generador de claves seguras, generar un par de claves (pública y privada) para RSA y guardar las claves en archivos.
3. Cifrado de datos → obtener los datos a cifrar (texto o archivo), inicializar objeto Cipher en modo cifrado con la clave pública, aplicar el cifrado a los datos y guardar o transmitir los datos cifrados.
4. Transferencia segura → configurar un socket seguro con SSL/TLS, establecer la conexión usando los certificados generados y enviar los datos cifrados a través del socket.
5. Recepción → recibir los datos cifrados en el otro extremo, inicializar objeto Cipher en modo descifrado con la clave privada y descifrar los datos recibidos.
6. Verificación → calcular un hash de los datos originales, comparar con el hash recibido para verificar integridad y confirmar que la comunicación se completó correctamente.



CRITERIOS

Tema 9: Protocolos nivel de aplicación (teoría)
1. Introducción a los servicios en red
2. Protocolo a nivel de aplicación
3. El protocolo DNS
4. El protocolo FTP
5. Los protocolos SMTP, POP3 e IMAP
6. El protocolo HTTP

Tema 10: Los servicios en red. Sockets I (teoría/práctica)
1. Definición de Sockets
2. Programación de un servidor TCP
3. Programación de un cliente TCP
4. Clases Datagrampacket y Datagramsocket
5. Programación de un servidor UDP

Tema 11: Los servicios en red. Sockets I (teoría/práctica)
1. Programación de un cliente HTTP
2. Programación de un cliente FTP
3. Programación de un cliente Telnet
4. Programación de un cliente SMTP
5. Socket e hilos

Tema 12: El servicio web (teoría/práctica)
1. ¿Qué es un servicio web?
2. Arquitectura orientada a servicios: SOA

Tema 13: La programación segura (teoría)
1. Introducción a la seguridad
2. Amenazas de seguridad
3. Ataques a un sistema informático
4. Vulnerabilidades en el software
5. Mecanismos de control de acceso
6. Validación de entradas

Tema 14: La criptografía (teoría/práctica)
1. Concepto de criptografía
2. Aplicaciones de la criptografía
3. Encriptación de la información
4. Criptografía de clave privada o simétrica
5. Criptografía de clave pública o simétrica
6. Firma digital y certificados digitales

Tema 15: Comunicaciones seguras (teoría/práctica)
1. Protocolos seguros de comunicaciones
2. Características SSL/TLS
3. Encriptación de datos con Cipher
4. Certificados para sockets seguros

Posibles casos prácticos:
Sincronización Cliente/Servidor TCP o UDP
Realizar un Cliente/Servidor Telnet
Realizar un Cliente/Servidor SMTP
Práctica de Criptografía