Generación de servicios en red

1. SERVICIOS

En **capítulos anteriores** se ha estudiado la computación distribuida y los mecanismos básicos de comunicación en red, con especial atención al **modelo cliente/servidor**. Este modelo, ampliamente utilizado, se basa en la **interacción entre clientes que solicitan servicios y servidores que los proporcionan**. A lo largo de este capítulo se abordará qué es un servicio, sus características principales, y cómo diseñar aplicaciones capaces de ofrecerlos.

1. SERVICIOS

Estructura y Función de un Sistema

Todo sistema tecnológico, ya sea en informática u otro campo, se divide en dos aspectos fundamentales: estructura y función. La **estructura** se refiere a los **componentes físicos o software que conforman el sistema**, mientras que la **función** describe su **propósito o utilidad**.

Por ejemplo, una lavadora tiene como estructura sus partes mecánicas y electrónicas, como el tambor o el motor, mientras que su función es lavar ropa.

En un sistema distribuido como WhatsApp, la estructura está formada por las aplicaciones cliente y servidor, y la función es facilitar la comunicación entre usuarios mediante mensajes y archivos multimedia. Este enfoque ayuda a entender cualquier sistema al identificar "de qué está hecho" y "para qué sirve".

1. SERVICIOS

Concepto de Servicio

Un servicio es el conjunto de **mecanismos** que un sistema proporciona para que los **usuarios accedan** a su **función**.

Mientras que la función define qué hace el sistema, el servicio concreta cómo se hace accesible esa funcionalidad. Por ejemplo, una aplicación de mensajería puede tener como función permitir la comunicación, pero su servicio se define por las herramientas concretas que ofrece, como el envío de texto o archivos. Un mismo sistema puede proporcionar múltiples servicios, dependiendo de su finalidad.

1. SERVICIOS

Interfaz del Servicio

La interfaz del servicio es el **punto de contacto entre el usuario y el sistema**. Especifica los procedimientos y restricciones necesarios para utilizar el servicio de forma controlada y eficiente. Por ejemplo, en una aplicación como WhatsApp, la interfaz incluye elementos como la aplicación móvil, los menús y las pantallas de usuario, que guían el acceso a las funcionalidades del sistema. Esta interfaz asegura que el servicio se utilice de manera adecuada y bajo las condiciones esperadas.

1. SERVICIOS

Servicios en Comunicaciones de Aplicaciones Distribuidas

Durante las comunicaciones en aplicaciones distribuidas, intervienen múltiples servicios que trabajan en conjunto para garantizar el paso de mensajes entre emisores y receptores. Estos servicios se organizan en la **pila de protocolos IP**

La **pila de protocolos IP** organiza los servicios en diferentes niveles jerárquicos que trabajan de forma cooperativa para garantizar el paso de mensajes entre emisores y receptores.

Los niveles principales de la pila son:

- **Red**: Tecnologías y servicios de conexión física.
- **Internet**: Mecanismos de encaminamiento y direccionamiento.
- **Transporte**: Gestión de conexiones y transmisión fiable de datos.
- **Aplicación**: Servicios que interactúan directamente con el usuario final.

1. SERVICIOS

Servicios en Cada Nivel

Cada nivel proporciona un servicio específico al nivel superior mediante una **interfaz de servicio**, que regula cómo interactúan entre ellos.

Ejemplos:

- Nivel de red: Servicios de comunicación física (p. ej., Ethernet).
- Nivel de Internet: Encaminamiento de paquetes (p. ej., IP).
- Nivel de transporte: Gestión de puertos y transmisión fiable (p. ej., TCP, UDP).
- Nivel de aplicación: Protocolos de usuario (p. ej., HTTP, FTP).

Protocolos de Comunicación

Cada nivel opera con su propio protocolo, que define cómo interactúa con otros elementos del sistema.

Estos protocolos garantizan que los servicios se ejecuten de manera eficiente y ordenada.

2. PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

A la hora de programar una aplicación siguiendo el modelo cliente/servidor, se deben definir de forma precisa los siguientes aspectos: **funciones del servidor, tecnología de comunicaciones y protocolo de nivel de aplicación**.

Funciones del servidor

A la hora de programar una aplicación distribuida siguiendo el modelo cliente/servidor, el primer paso debe ser siempre definir de forma clara las funciones del servidor. Algunas de las preguntas clave que debemos hacernos a la hora de definir nuestro servidor son:

¿Cuál es la función básica de nuestro servidor?

- El servicio que proporciona nuestro servidor, ¿es rápido o lento?
- El servicio que proporciona nuestro servidor, ¿puede resolverse con una simple petición y respuesta o requiere del intercambio de múltiples mensajes entre este y el cliente?
- ¿Debe ser capaz nuestro servidor de atender a varios clientes simultáneamente?

El resultado de contestar estas y otras preguntas similares debe ser el hacernos una idea clara de cómo va a ser nuestro servidor, qué va a hacer y cómo va a interactuar con los clientes.

2. PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Elección de la Tecnología de Comunicación

El siguiente paso, después de definir las características del servidor, es seleccionar la tecnología de comunicaciones adecuada. Los dos mecanismos básicos disponibles son los **sockets stream** y los **sockets datagram**. Cada tipo presenta ventajas e inconvenientes, por lo que su elección debe basarse en las características del servicio que proporcionará el servidor.

Sockets Stream
Características:

- Orientados a conexión.
- Garantizan la entrega ordenada y fiable de los mensajes.
- Adecuados para aplicaciones complejas que requieren un intercambio intensivo de mensajes entre cliente y servidor.

Uso recomendado: Cuando la fiabilidad y el mantenimiento de una conexión estable son prioritarios.

Sockets Datagram
Características:

- No orientados a conexión.
- Menos fiables: los mensajes pueden perderse o llegar desordenados.
- Mayor eficiencia, con menor sobrecarga en la comunicación.

Uso recomendado: En aplicaciones sencillas donde no sea crítico perder mensajes.

2. PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Definición del Protocolo de Nivel de Aplicación

¿Qué es un Protocolo de Nivel de Aplicación?

Un protocolo de nivel de aplicación es un conjunto de reglas que regula la interacción entre los elementos de una aplicación distribuida. En el modelo cliente/servidor, este protocolo define cómo se comunican el cliente y el servidor, estableciendo las bases para el intercambio de mensajes.

Elementos Clave del Protocolo

1. **Formato de los Mensajes**:
   - Especifica cómo deben estructurarse los mensajes intercambiados entre cliente y servidor.
   - Incluye detalles como el contenido, la organización y las etiquetas necesarias para interpretar los datos.
2. **Secuencia de Mensajes**:
   - Define las posibles secuencias en las que los mensajes pueden ser enviados y recibidos.
   - Determina el orden permitido de las peticiones y respuestas, asegurando que la comunicación sea coherente.
3. **Tipos de Mensajes**:
   - Incluye todos los tipos de peticiones y respuestas que pueden ser enviados entre cliente y servidor.
   - Cada tipo de mensaje debe estar claramente definido, indicando cuándo puede enviarse y en qué contexto.

2. PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Servidores Multihilo: Gestión de Múltiples Clientes

Los servidores modernos están diseñados para atender a múltiples clientes de manera simultánea, cumpliendo con dos condiciones clave:

1. **Aislamiento entre clientes**: Cada cliente debe percibir que está operando con el servidor de manera exclusiva, sin interferencias de otros clientes.
2. **Atención rápida y simultánea**: Nuevas peticiones deben ser atendidas tan pronto como sea posible, incluso mientras se procesan otras peticiones en curso.

Para cumplir con estas condiciones, se utiliza un enfoque **multihilo**, donde cada cliente es atendido por un hilo de ejecución independiente. Esto garantiza que el servidor pueda gestionar múltiples solicitudes de manera eficiente y sin bloqueos.

2. PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Funcionamiento de un Servidor Multihilo

1. **Hilos independientes para cada cliente:**
    - Cuando un cliente envía una nueva petición, el servidor crea un nuevo hilo que se encarga exclusivamente de interactuar con ese cliente.
    - Este diseño permite que el hilo principal del servidor quede libre para gestionar nuevas conexiones.
2. **Sockets y operación accept:**
    - Los sockets stream, orientados a conexión, son ideales para servidores multihilo.El servidor utiliza un socket servidor que espera conexiones entrantes.
    - Cuando llega una nueva conexión, la operación accept crea un nuevo socket para comunicarse con ese cliente en particular.
3. **Asignación de hilos:**
    - Tras aceptar una conexión, el servidor arranca un nuevo hilo y le asigna el socket del cliente.
    - Este nuevo hilo gestiona todas las interacciones con el cliente, utilizando el socket ya conectado.
4. **Eficiencia del hilo principal:**
    - Mientras los hilos secundarios interactúan con los clientes, el hilo principal del servidor permanece libre, esperando nuevas conexiones y ejecutando accept.

Ventajas del Enfoque Multihilo

•**Atención simultánea**: Permite gestionar múltiples clientes al mismo tiempo, sin hacerlos esperar innecesariamente.

•**Escalabilidad**: Adecuado para sistemas que deben atender a miles de usuarios, como servidores de correo o páginas web populares.

•**Aislamiento**: Cada cliente tiene un canal exclusivo para comunicarse con el servidor, garantizando independencia en la atención.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Las **aplicaciones distribuidas**, como páginas web y correo electrónico, siguen el **modelo cliente/servidor** para comunicarse. Cliente y servidor suelen ser **independientes**, desarrollados por distintas personas o empresas y en lenguajes diferentes. Para garantizar su comunicación efectiva, es esencial **definir protocolos de nivel de aplicación**. A lo largo del tiempo, se han establecido protocolos estándar que facilitan el desarrollo de estos sistemas.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Telnet: Comunicación Remota en Texto Plano

- Protocolo de nivel de aplicación para **comunicación bidireccional en texto plano (ASCII)**.
- Simula una **conexión virtual a una terminal de texto**, permitiendo acceso remoto a máquinas.
- Utiliza **TCP (sockets stream)** y el puerto **23** por defecto.
- Funciona como una **sesión de línea de comandos** (Shell en UNIX, Símbolo del sistema en Windows).
- **Problema de seguridad**: envía datos sin cifrar, lo que facilita la intercepción de información sensible.
- **Uso desaconsejado** en redes abiertas; se recomienda solo en entornos controlados.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

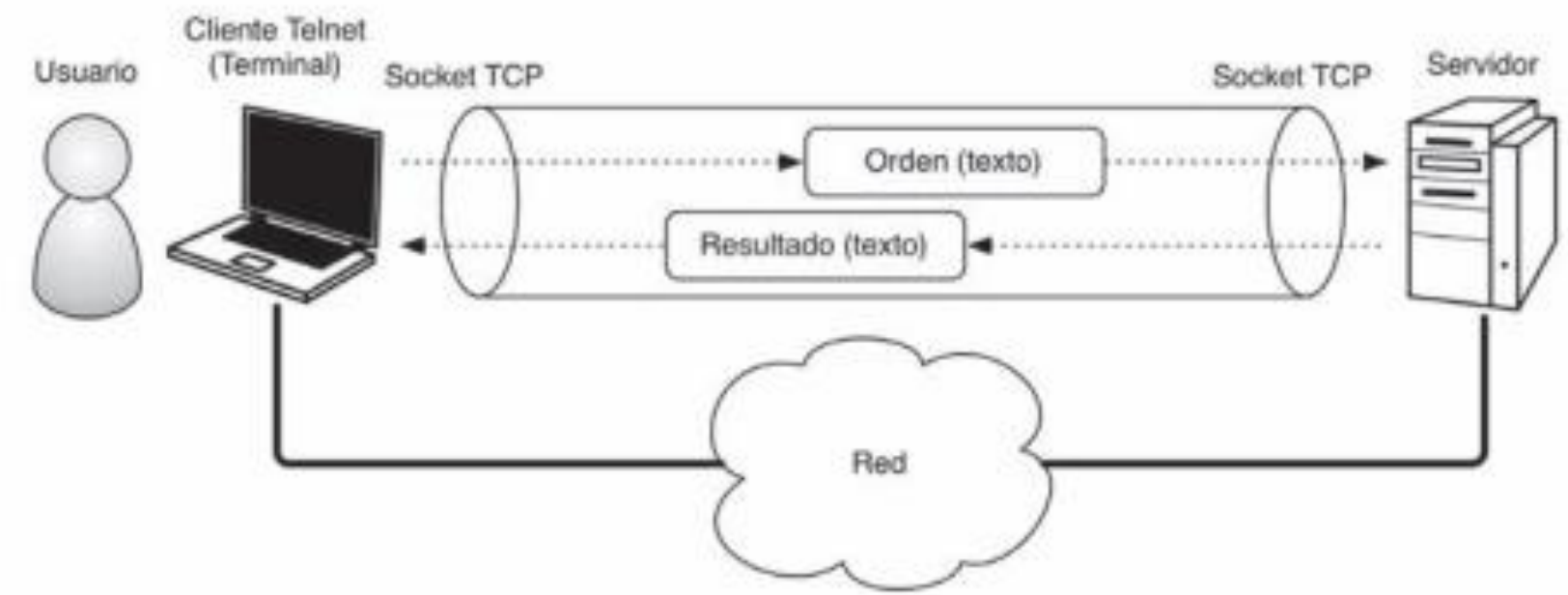

Figura 4.3. *Comunicación usando el protocolo de nivel de aplicación Telnet*

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

SSH (Secure Shell): Comunicación Segura

- **Protocolo de nivel de aplicación** similar a Telnet, pero más moderno (**1995**).
- Diseñado para **solucionar las vulnerabilidades de Telnet**.
- **Cifra la información** transferida entre cliente y servidor, garantizando seguridad.
- **Recomendado** para sesiones remotas en redes inseguras como Internet.
- **Protocolo con estado (stateful)**: **mantiene información de la sesión** durante toda la comunicación, permitiendo autenticación y seguimiento de comandos.
- Usa el **puerto 22** por defecto.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

FTP (File Transfer Protocol): Transferencia de Archivos en Red

- **Protocolo de nivel de aplicación** para la transferencia de archivos en red.
- Utiliza **TCP (sockets stream)** para la comunicación.
- Establece **dos conexiones simultáneas**:
  - **Conexión de control (puerto 21):** envía órdenes y recibe información.
  - **Conexión de datos:** transfiere archivos.
- Permite **subida y descarga de archivos**, además de gestionar directorios y permisos.
- **Protocolo con estado (stateful):** mantiene la sesión activa durante la transferencia.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

FTP (File Transfer Protocol): Transferencia de Archivos en Red

- **Protocolo de nivel de aplicación** para la transferencia de archivos en red.
- Utiliza **TCP (sockets stream)** para la comunicación.
- Establece **dos conexiones simultáneas**:
  - **Conexión de control (puerto 21):** envía órdenes y recibe información.
  - **Conexión de datos:** transfiere archivos.
- Permite **subida y descarga de archivos**, además de gestionar directorios y permisos.
- **Protocolo con estado (stateful):** mantiene la sesión activa durante la transferencia.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

HTTP (Hypertext Transfer Protocol): Base de la Web

- **Protocolo de nivel de aplicación** esencial para la World Wide Web.
- Controla la **transferencia de documentos de hipertexto** entre navegadores y servidores web.
- **Documentos de hipertexto**: incluyen **hiperenlaces**, imágenes y formatos avanzados.
- **Modelo cliente/servidor**, basado en **petición-respuesta**.
- Funciona principalmente sobre **TCP (puerto 80)**, pero puede usar otros protocolos de transporte.
- **Protocolo sin estado (stateless)**: no almacena información de la sesión.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Cookies y almacenamiento de estado en HTTP

- **Las cookies** son pequeños fragmentos de información almacenados en el cliente (navegador).
- Permiten **mantener datos entre peticiones**, como sesiones de usuario, preferencias y carritos de compra.
- Enviadas por el servidor y guardadas por el navegador para su uso en futuras peticiones.
- Mejoran la experiencia del usuario, pero pueden tener implicaciones de **seguridad y privacidad**.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Cookies y almacenamiento de estado en HTTP

- **Las cookies** son pequeños fragmentos de información almacenados en el cliente (navegador).
- Permiten **mantener datos entre peticiones**, como sesiones de usuario, preferencias y carritos de compra.
- Enviadas por el servidor y guardadas por el navegador para su uso en futuras peticiones.
- Mejoran la experiencia del usuario, pero pueden tener implicaciones de **seguridad y privacidad**.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Sesiones HTTP y Comunicación Cliente-Servidor

- **Sesión HTTP**: secuencia de intercambios **petición-respuesta** entre cliente y servidor.
- Cliente inicia la sesión estableciendo una **conexión TCP (puerto 80)**.
- Servidor espera peticiones HTTP y responde con:
  - **Estado de la petición** (éxito o error).
  - **Cuerpo del mensaje** con contenido solicitado (páginas web, archivos, etc.).

Recursos y URLs en HTTP

- **Recurso:** cualquier documento almacenado en el servidor.
- Identificado mediante una **URL (Uniform Resource Locator)**.
- La URL se usa en hiperenlaces para acceder a recursos en la Web.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Métodos HTTP Más Utilizados

•**GET**: Solicita un recurso (página web, imagen, etc.).

- Es la primera petición que hace un navegador al cargar una web.

•**HEAD**: Igual que GET, pero solo devuelve metadatos (sin cuerpo del mensaje).

- Útil para verificar la existencia de un recurso sin descargarlo.

•**POST**: Envía datos al servidor (formularios, comentarios, etc.).

- Se usa en foros, registros y envío de información.

•**PUT**: Sube o actualiza un recurso en el servidor.

- Si el recurso ya existe, lo reemplaza.

•**DELETE**: Elimina un recurso del servidor.

- No siempre está permitido por razones de seguridad.

•**OPTIONS**: Informa qué métodos admite el servidor en una URL.

•**TRACE**: Devuelve la petición original a modo de eco (útil para depuración de modificaciones por elementos intermedios).

•**CONNECT**: Convierte la conexión en un túnel TCP/IP.

- Se usa en envío de datos cifradas (HTTPS).

•**PATCH**: Modifica parcialmente un recurso en lugar de reemplazarlo por completo.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Compatibilidad y Seguridad en HTTP

•**No todos los servidores aceptan todos los métodos**.

- **GET y HEAD** son los mínimos necesarios.
- **OPTIONS** suele ser soportado para consulta.

•**Restricciones por seguridad**:

- **DELETE y PUT** pueden estar bloqueados.
- **TRACE** puede deshabilitarse para evitar ataques de inyección de información.

•**HTTPS (HTTP Seguro)**:

- Usa cifrado TLS/SSL para proteger las comunicaciones.
- Evita ataques de interceptación y robo de datos.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Códigos de Estado HTTP

- Todas las respuestas HTTP incluyen un **código de estado** en la línea de estado.
- Son **números de tres dígitos**, indicando el resultado de la petición.

División en 5 categorías:

1xx - Información

- Indican que la petición sigue en proceso.
- **100 (Continue):** el cliente debe seguir enviando datos.

2xx - Éxito

- La petición fue recibida y procesada correctamente.
- **200 (OK):** éxito estándar.
- **202 (Accepted):** la petición fue aceptada, pero aún no procesada.

3xx - Redirección

- El cliente debe hacer otra acción para completar la petición.
- **303 (See Other):** redirige a otra URL.

4xx - Error del Cliente

- Indican que el problema proviene del cliente.
- **400 (Bad Request):** petición incorrecta.
- **403 (Forbidden):** acceso denegado.
- **404 (Not Found):** recurso no encontrado.

5xx - Error del Servidor

- Indican fallos internos del servidor.
- **500 (Internal Server Error):** error inesperado.
- **503 (Service Unavailable):** servicio no disponible.

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Protocolos de Correo Electrónico

POP3 (Post Office Protocol, versión 3)

- Protocolo de nivel de aplicación para acceder a correos electrónicos almacenados en servidores.
- Utilizado por clientes como Thunderbird y Outlook.
- Funciona sobre TCP (puerto 110).
- Permite descargar y borrar mensajes, pero no sincroniza cambios con el servidor.
- Protocolo sin estado (stateless): cada sesión es independiente.

SMTP (Simple Mail Transfer Protocol)

- Protocolo estándar para el envío de correos electrónicos en Internet.
- Usado por clientes de correo para enviar mensajes a los servidores.
- Funciona sobre TCP (puerto 587).
- No incorpora cifrado de seguridad de forma nativa, aunque puede usar extensiones como STARTTLS.
- Protocolo sin estado (stateless).

3. PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

Otros Protocolos de Nivel de Aplicación

DHCP (Dynamic Host Configuration Protocol):

- Asigna dinámicamente direcciones IP en una red.

DNS (Domain Name System):

- Traduce nombres de dominio en direcciones IP.

NTP (Network Time Protocol):

- Sincroniza la hora entre dispositivos en una red.

TLS (Transport Layer Security):

- Añade cifrado y seguridad a protocolos como HTTPS y FTPS.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

Abstracción en la Programación de Aplicaciones Distribuidas

- **Programar con sockets directamente es poco práctico** en aplicaciones de alto nivel.
- Existen **tecnologías avanzadas** que simplifican la comunicación y ofrecen funcionalidades mejoradas.
- Estas tecnologías actúan como una **capa intermedia** entre el nivel de transporte y el nivel de aplicación.

Invocación de Métodos Remotos (RMI)

- Basada en la idea de que la **invocación de métodos es una forma de comunicación** entre objetos.
- Un **objeto A** puede llamar un método en un **objeto B** que está en otra máquina de la red.
- Se intercambian **mensajes a través de la red** para realizar la ejecución del método.
- En **Java**, esta técnica se conoce como **Remote Method Invocation (RMI)**.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

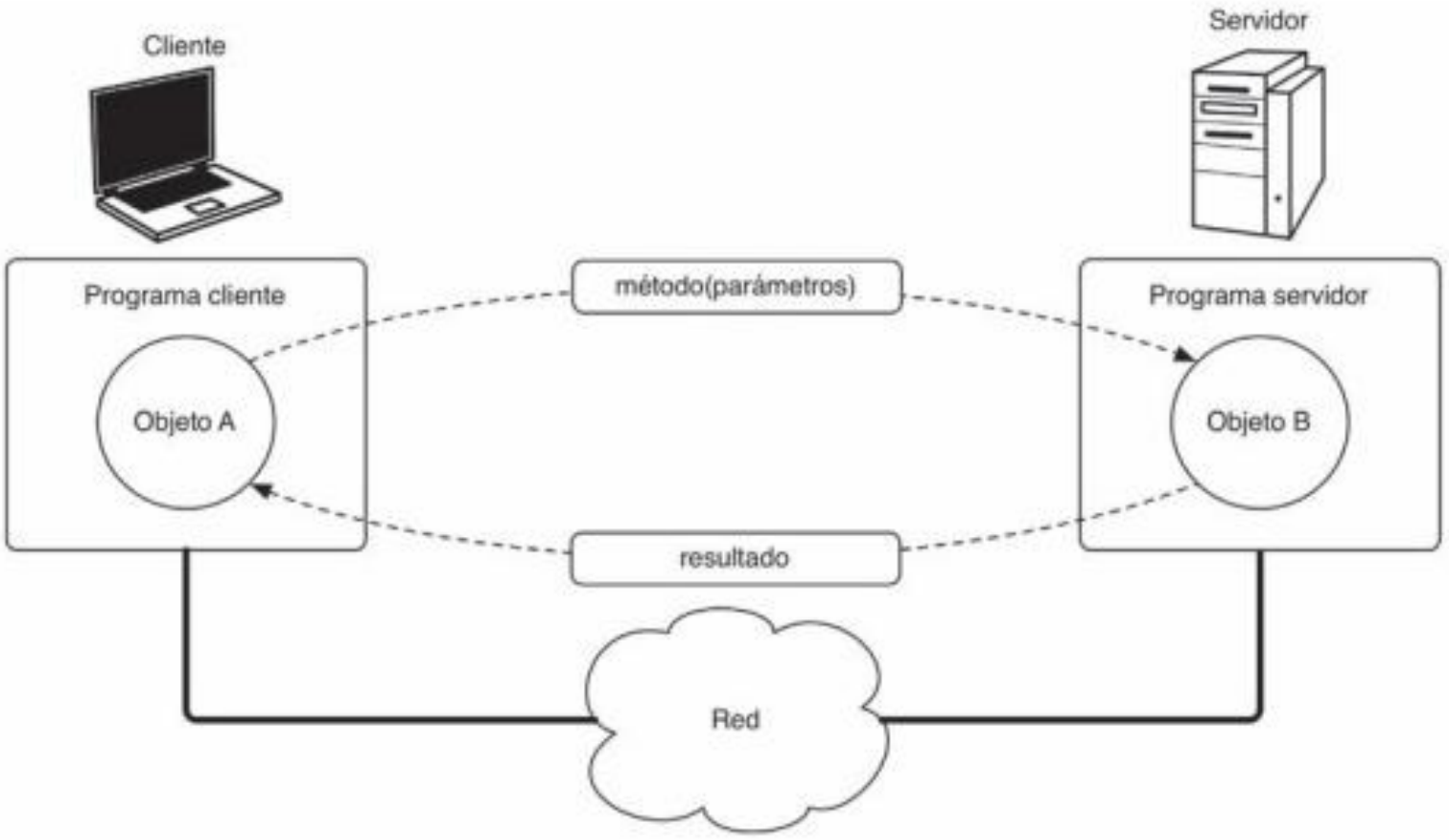

Figura 4.4. *Ejemplo de invocación de un método remoto*

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

Infraestructura de comunicaciones para la invocación de métodos remotos

Para poder llevar a cabo la invocación de métodos remotos se necesita una capa de software adicional entre la aplicación de alto nivel y el nivel de transporte de la pila IP.

Funciones de la Capa de Software Adicional

- Traduce llamadas a métodos y valores de retorno en **mensajes** enviados por la red.
- Convierte los **mensajes recibidos** en valores de retorno y llamadas a métodos.
- Permite **crear y localizar objetos remotos** en la red.
- Facilita la programación de aplicaciones distribuidas, **ocultando la complejidad** de la comunicación en red.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

Componentes Principales

1. Stubs (Intermediarios del Cliente)

- Actúan como **representantes del objeto remoto** dentro del programa cliente.
- El cliente **invoca métodos en el stub**, que:
  - **Crea un mensaje** con la petición y lo envía al objeto remoto.
  - **Espera la respuesta** del servidor.
  - **Devuelve el resultado** como si la ejecución fuera local.
- Permiten que el cliente **no gestione la comunicación en red directamente**.

2. Registro de Objetos Remotos

- Servicio encargado de **gestionar y localizar los objetos remotos que existen en el sistema**.
- Cuando un objeto remoto **se crea, se registra** en este sistema.
- Un cliente que necesita usar un objeto remoto **consulta el registro** para obtener su información.
- Facilita la conexión entre clientes y servidores de forma **eficiente y organizada**.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

Proceso Detallado de la Invocación de Métodos Remotos (RMI)

Para realizar una **invocación de métodos remotos**, es necesario seguir una secuencia de pasos en **servidor y cliente**.

1. Arranque del Registro de Objetos Remotos (Servidor)

- Se inicia el **registro de objetos remotos**.
- Es un servicio que permite a los clientes localizar objetos remotos.
- **Este debe ser el primer paso**, si no está en ejecución previamente.

2. Creación del Objeto Servidor (Servidor)

- Se **crea el objeto servidor**, como cualquier otro objeto en el programa.
- Este objeto contendrá los **métodos que serán invocados remotamente**.

3. Inscripción del Objeto en el Registro (Servidor)

- El objeto servidor **se registra** en el sistema para que pueda ser encontrado.
- Se asigna un **nombre identificador** que lo hace accesible para los clientes.
- Una vez registrado, se convierte en un **objeto remoto**.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

4. Localización del Objeto Remoto (Cliente)

- El cliente busca el objeto remoto en el **registro**.
- Se conecta al registro utilizando su **dirección y puerto**.
- Obtiene la información necesaria para establecer la comunicación.
- **El proceso suele incluir la creación del objeto stub**, que actuará como intermediario.

5. Invocación del Método en el Objeto Stub (Cliente)

- El cliente llama a un **método del stub**, como si fuera una llamada local.
- El stub **construye un mensaje** con la petición y lo envía al servidor.

6. Intercambio de Mensajes entre Stub y Objeto Servidor (Red)

- **El stub envía la solicitud** con el método y los parámetros al servidor.
- El **objeto remoto** recibe la petición y ejecuta el método.
- **Se genera un mensaje con el resultado** y se envía de vuelta al cliente.

7. Obtención del Valor de Retorno (Cliente)

- El stub recibe la respuesta del servidor.
- **Convierte el mensaje en un valor de retorno** y lo entrega al cliente.
- Desde el punto de vista del cliente, parece una llamada de método local.

4. Técnicas Avanzadas de Programación de Aplicaciones Distribuidas

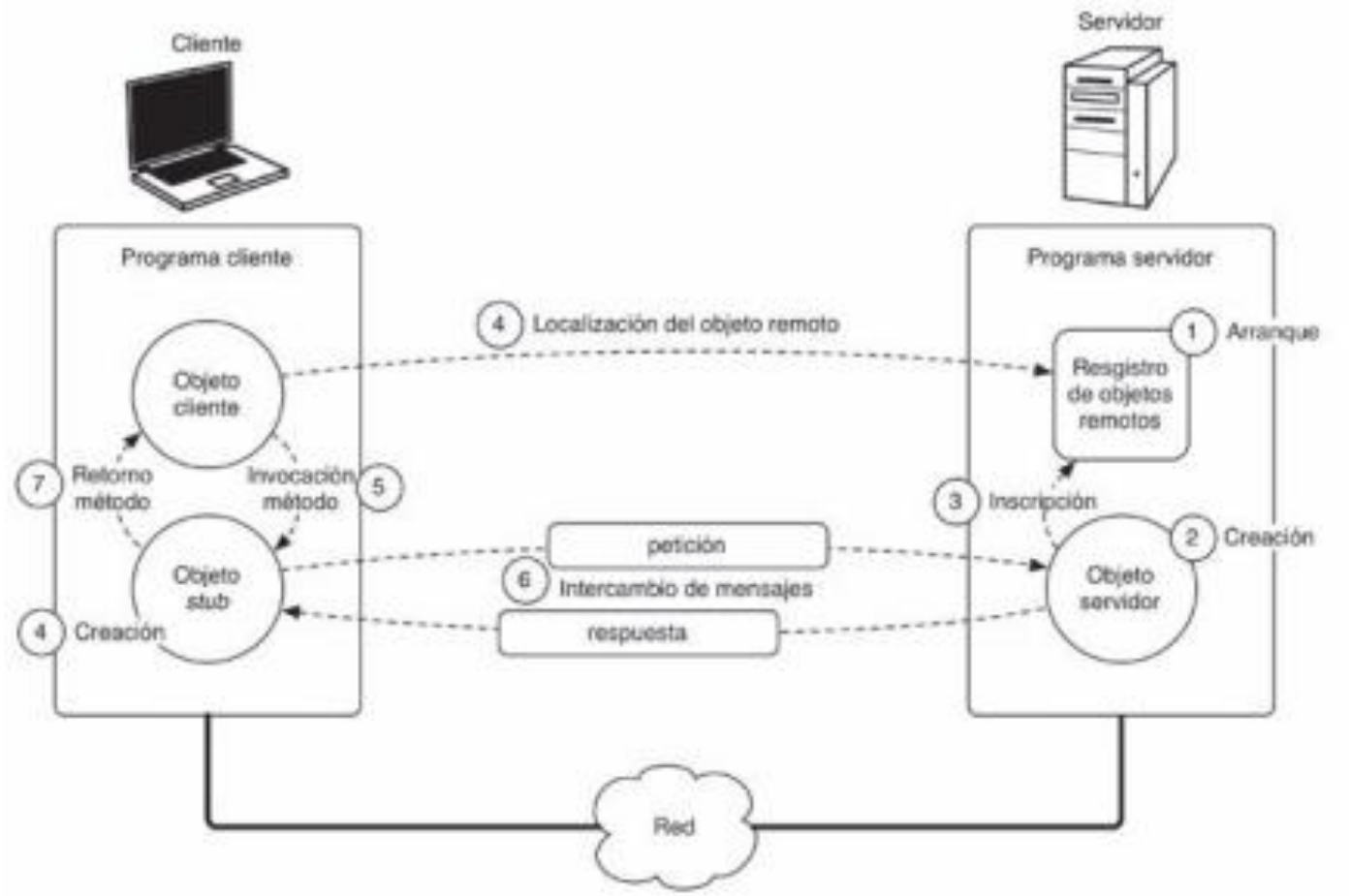

Figura 4.5. *Proceso detallado de invocación de un método remoto*

Muchas gracias

Explicación Detallada del Código

Estos ejemplos muestran cómo implementar una comunicación básica cliente-servidor utilizando **sockets datagram** en Java. Los sockets datagram son parte de la comunicación orientada a paquetes y se implementan con la clase `DatagramSocket`. Este tipo de comunicación se basa en el protocolo UDP (User Datagram Protocol), que es más eficiente pero menos fiable que el protocolo TCP.

A continuación, se explica en detalle cómo funcionan cada una de las clases y los objetos utilizados:

Clase `HoraClient` (Cliente)

La clase `HoraClient` representa la parte cliente de la comunicación. Su objetivo es enviar un mensaje al servidor solicitando la hora actual y recibir la respuesta.

1. **Creación del Socket Datagram**:
   - Se utiliza la clase `DatagramSocket` para crear un socket que permita enviar y recibir paquetes de datos.
   - En este caso, el cliente crea un socket sin especificar puerto, permitiendo que el sistema operativo asigne uno automáticamente.

     ```
     DatagramSocket datagramSocket = new DatagramSocket();
     ```

   - Este socket actúa como el punto de conexión del cliente para la comunicación.
2. **Preparación del Mensaje**:
   - El mensaje `"hora"` se crea como un objeto de tipo `String` y se convierte en bytes usando el método `getBytes()`.
   - La conversión a bytes es necesaria porque los sockets trabajan directamente con datos binarios.

     ```
     String mensaje = "hora";
     mensaje.getBytes();
     ```

3. **Definición de la Dirección y Puerto del Servidor**:
   - La clase `InetAddress` se usa para obtener la dirección IP del servidor. Aquí se usa `"localhost"`, que apunta a la máquina local.
   - El puerto utilizado es `5555`, que debe coincidir con el puerto en el que el servidor está escuchando.

     ```
     InetAddress serverAddr =
     InetAddress.getByName("localhost");
     ```

4. **Creación y Envío del Paquete**:
   - Se utiliza la clase `DatagramPacket` para crear un paquete de datos que contiene el mensaje, su longitud, la dirección del servidor y el puerto.
   - El paquete se envía usando el método `send()` del socket.

```
DatagramPacket datagrama1 = new
DatagramPacket(mensaje.getBytes(),
mensaje.getBytes().length, serverAddr, 5555);
datagramSocket.send(datagrama1);
```

5. **Recepción de la Respuesta**:
   - El cliente prepara un buffer de tamaño suficiente para recibir la respuesta. Este buffer se incluye en un objeto `DatagramPacket`.
   - El método `receive()` bloquea la ejecución hasta que se recibe un paquete, que luego se almacena en el buffer.

```
byte[] respuesta = new byte[100];
DatagramPacket datagrama2 = new DatagramPacket(respuesta,
respuesta.length);
datagramSocket.receive(datagrama2);
```

6. **Procesamiento y Cierre**:
   - La respuesta se convierte de bytes a texto usando el constructor de `String`.
   - Finalmente, el socket se cierra para liberar recursos.

```
System.out.println("Mensaje recibido: " + new
String(respuesta));
datagramSocket.close();
```

Clase `HoraServer` (Servidor)

La clase `HoraServer` actúa como un servidor que escucha las solicitudes de los clientes y responde con la hora actual.

1. **Creación del Socket y Dirección**:
   - Se utiliza la clase `DatagramSocket` para crear un socket que escuche en la dirección `localhost` y el puerto `5555`.
   - La clase `InetSocketAddress` se usa para asociar la dirección y el puerto al socket.

```
InetSocketAddress addr = new
InetSocketAddress("localhost", 5555);
DatagramSocket datagramSocket = new DatagramSocket(addr);
```

2. **Bucle de Recepción**:
   - El servidor entra en un bucle infinito para recibir y procesar mensajes continuamente. Este bucle garantiza que el servidor permanezca activo mientras espera solicitudes.

```
while (datagramSocket != null) {
    // Procesamiento
}
```

3. **Recepción de Mensajes**:
    - Se prepara un buffer de tamaño suficiente para recibir un mensaje.
    - Se utiliza un objeto `DatagramPacket` para recibir los datos del cliente. El método `receive()` bloquea la ejecución hasta que llega un paquete.

```
byte[] buffer = new byte[4];
DatagramPacket datagrama1 = new DatagramPacket(buffer,
buffer.length);
datagramSocket.receive(datagrama1);
```

4. **Procesamiento del Mensaje**:
    - El contenido del mensaje se extrae del paquete y se convierte a un `String`.
    - Se obtiene la dirección IP y el puerto del cliente a partir del paquete recibido.
    - Si el mensaje es `"hora"`, el servidor obtiene la hora actual del sistema y la convierte en un array de bytes para enviarla como respuesta.

```
Date d = new Date(System.currentTimeMillis());
byte[] respuesta = d.toString().getBytes();
DatagramPacket datagrama2 = new DatagramPacket(respuesta,
respuesta.length, clientAddr, clientPort);
datagramSocket.send(datagrama2);
```

5. **Gestión de Errores y Mensajes No Reconocidos**:
    - Si el mensaje no es reconocido, el servidor imprime un mensaje indicando que no puede procesarlo.
    - Los errores de entrada/salida se gestionan mediante bloques `try-catch`.

Relación entre Cliente y Servidor

- El cliente envía un paquete con el mensaje `"hora"` al servidor.
- El servidor recibe este mensaje, verifica su contenido y responde con la hora actual si el mensaje es válido.
- La comunicación se realiza utilizando **paquetes UDP**, lo que significa que los mensajes no están garantizados para llegar ni en orden ni sin pérdida, pero son rápidos y eficientes.

Estos ejemplos muestran cómo se implementa una comunicación básica cliente-servidor utilizando sockets datagram en Java, y son una introducción práctica a los conceptos de programación de redes en sistemas distribuidos.

Introducción

Este proyecto implementa un sistema cliente-servidor utilizando sockets en Java. El servidor realiza operaciones aritméticas (suma, resta, multiplicación y división) solicitadas por el cliente. Este documento desglosa el código, explicando cada elemento y concepto involucrado, con el objetivo de que tus alumnos comprendan cómo funcionan estos programas y puedan adaptarlos o mejorarlos.

1. Clase `CalcServer` (Servidor)

La clase `CalcServer` tiene dos partes principales:

- Un **servidor principal**, que escucha conexiones entrantes.
- Un **hilo de ejecución**, que procesa las solicitudes de cada cliente de manera independiente.

1.1 Funcionamiento del Servidor Principal

El servidor utiliza un objeto de tipo `ServerSocket` para gestionar conexiones. Estas son las operaciones clave:

1. **Creación del Socket Servidor**:

   ```
   ServerSocket serverSocket = new ServerSocket();
   ```

   - El `ServerSocket` permite al servidor escuchar conexiones entrantes en un puerto específico.
2. **Vinculación a una Dirección y Puerto**:

   ```
   InetSocketAddress addr = new InetSocketAddress("localhost",
   5555);
   serverSocket.bind(addr);
   ```

   - El método `bind` asocia el socket a la dirección `localhost` y al puerto `5555`, especificando dónde estará disponible el servidor.
3. **Aceptación de Conexiones**:

   ```
   Socket newSocket = serverSocket.accept();
   ```

   - El método `accept()` bloquea la ejecución hasta que un cliente intenta conectarse. Una vez establecida la conexión, devuelve un nuevo `Socket` para comunicar con ese cliente.

4. **Inicio de un Hilo para el Cliente**:

```
CalcServer hilo = new CalcServer(newSocket);
hilo.start();
```

   - Cada cliente se gestiona en un hilo independiente creado a partir de la clase `CalcServer`. Esto permite atender a varios clientes simultáneamente.

1.2 Clase `CalcServer` como Hilo

La clase `CalcServer` extiende `Thread` para manejar las solicitudes de un cliente en paralelo. Estas son las operaciones clave:

1. **Recepción de Datos**:
   - Los datos se leen del `InputStream` asociado al socket del cliente:

     ```
     InputStream is = clientSocket.getInputStream();
     ```

   - Se recibe:
     - La operación (como un carácter: `+`, `-`, `*`, `/`).
     - Los dos operandos como enteros.
2. **Validación de la Operación**:
   - Se verifica que el operador sea válido:

     ```
     if (operacion.equals("+") || operacion.equals("-") ||
     operacion.equals("*") || operacion.equals("/")) { ... }
     ```

3. **Cálculo del Resultado**:
   - Dependiendo de la operación recibida, se realiza el cálculo:

     ```
     if (operacion.equals("+")) {
         result = op1 + op2;
     } else if (operacion.equals("-")) { ... }
     ```

   - Si el operador es inválido, no se realiza el cálculo.
4. **Envío del Resultado**:
   - El resultado se envía al cliente utilizando el `OutputStream`:

     ```
     os.write(result);
     ```

5. **Manejo de Errores**:
   - Se capturan y gestionan posibles excepciones, como problemas en la conexión.

2. Clase `CalcClient` (Cliente)

La clase `CalcClient` es un programa que se conecta al servidor, envía una solicitud de cálculo y recibe el resultado.

2.1 Creación del Socket Cliente

1. **Creación del Socket**:

   ```
   Socket clientSocket = new Socket();
   ```

   - Se crea un socket que actuará como punto de comunicación del cliente.
2. **Conexión al Servidor**:

   ```
   InetSocketAddress addr = new InetSocketAddress("localhost",
   5555);
   clientSocket.connect(addr);
   ```

   - El cliente se conecta a la dirección y puerto del servidor. Si el servidor no está disponible, se genera una excepción.

2.2 Interacción Cliente-Servidor

1. **Envío de Datos**:
   - El cliente utiliza el `OutputStream` del socket para enviar:
     - El operador (`+`, `-`, `*`, `/`):

       ```
       os.write("+".getBytes());
       ```

     - Los dos operandos como enteros:

       ```
       os.write(59);  // Primer operando
       os.write(130); // Segundo operando
       ```

2. **Recepción del Resultado**:
   - El cliente lee el resultado del cálculo desde el `InputStream`:

     ```
     int result = is.read();
     ```

   - El resultado se imprime en la consola.
3. **Cierre del Socket**:
   - Una vez completada la comunicación, el cliente cierra el socket:

     ```
     clientSocket.close();
     ```