Guía de Estudio: Procesos, Hilos y Comunicación en Red

1. Fundamentos de Procesos: La Base de la Ejecución

1.1. ¿Qué es un Proceso? De Programa a Ejecución

En el mundo de los sistemas operativos, es crucial distinguir entre un **programa** y un **proceso**.

- Un **programa** es un conjunto de instrucciones estáticas, un archivo ejecutable que reside en el disco duro. Es una entidad pasiva.
- Un **proceso** es una instancia de un programa en ejecución. Es una entidad activa que el sistema operativo gestiona y a la que asigna recursos.

Cuando ejecutas un programa, el sistema operativo crea un proceso. Este proceso tiene características clave que lo definen como una unidad de trabajo independiente:

- **Espacio de Memoria Propio:** Cada proceso dispone de su propia área de memoria virtual. Esto garantiza que un proceso no pueda interferir directamente con la memoria de otro, proporcionando aislamiento y seguridad.
- **Acceso a Recursos del Sistema:** El sistema operativo asigna recursos al proceso, como tiempo de CPU, acceso al disco duro, memoria RAM y periféricos.
- **Estado de Ejecución:** Un proceso siempre se encuentra en un estado específico (listo, en ejecución, bloqueado, etc.) que define su actividad actual.
- **Capacidad de Creación:** Un proceso puede crear nuevos procesos durante su ejecución, los cuales a su vez pueden crear otros. De esta forma, a partir de un proceso inicial se puede crear una **jerarquía de procesos**.

1.2. El Ciclo de Vida de un Proceso: Los Estados

Un proceso atraviesa diferentes estados a lo largo de su ciclo de vida. El sistema operativo gestiona las transiciones entre estos estados basándose en la disponibilidad de recursos y las acciones del propio proceso.

- `Listo (Ready)`
    - **Descripción:** El proceso está cargado en la memoria principal y preparado para ejecutarse, pero está a la espera de que el planificador del sistema operativo le asigne la CPU.
    - **Causa de Transición:** Un proceso pasa a este estado al ser creado o cuando es desalojado de la CPU (por ejemplo, al agotarse su *quantum* de tiempo).
- `Ejecución (Running)`
    - **Descripción:** El proceso está utilizando activamente la CPU para ejecutar sus instrucciones.
    - **Causa de Transición:** El planificador del sistema operativo selecciona un proceso del estado `Listo` y le asigna la CPU.
- `Bloqueado (Blocked)`

    - **Descripción:** El proceso no puede continuar su ejecución porque está esperando que ocurra un evento externo, como una operación de entrada/salida (leer un archivo del disco), la disponibilidad de un recurso o una señal de otro proceso.
    - **Causa de Transición:** El proceso solicita una operación que no puede completarse de inmediato. Mientras está bloqueado, no compite por la CPU.
- `Terminado (Terminated)`
    - **Descripción:** El proceso ha finalizado su ejecución. El sistema operativo libera todos los recursos que le habían sido asignados.
    - **Causa de Transición:** El proceso completa su tarea o es finalizado por el sistema operativo o el usuario.

Nota: Existen estados adicionales que ofrecen mayor granularidad, como `Listo suspendido` y `Bloqueado suspendido`, que representan procesos que han sido temporalmente movidos de la memoria principal al almacenamiento secundario para liberar RAM.

1.3. Creación de Procesos Externos en Java

Java, como lenguaje de alto nivel, ofrece mecanismos para interactuar con el sistema operativo subyacente, permitiendo lanzar y gestionar procesos externos. Esto es útil para ejecutar comandos del sistema, scripts u otros programas desde una aplicación Java.

1.3.1. Usando `Runtime`

La clase `Runtime` proporciona una forma sencilla de ejecutar comandos del sistema. Es un enfoque directo, aunque menos flexible que `ProcessBuilder`.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EjecutorRuntime {
    public static void main(String[] args) {
        try {
            // 1. Obtener la instancia singleton de Runtime para interactuar con el SO.
            Runtime runtime = Runtime.getRuntime();

                // 2. Definir el comando para listar archivos detalladamente, adaptado al SO.
            String comando = System.getProperty("os.name").startsWith("Windows") ? "dir" : "ls -l";
            Process proceso = runtime.exec(comando);

            // 3. Capturar la salida estándar del proceso para mostrarla.
            System.out.println("Salida del comando '" + comando + "':");
                            BufferedReader reader = new BufferedReader(new InputStreamReader(proceso.getInputStream()));
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }

               // 4. Esperar a que el proceso termine y obtener su código de salida.
            int codigoSalida = proceso.waitFor();
            if (codigoSalida == 0) {
                System.out.println("\nComando ejecutado correctamente.");
```

```
            } else {
                 System.out.println("\nError en la ejecución del comando. Código
de salida: " + codigoSalida);
            }

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

El propósito de la clase `Runtime` es actuar como un puente entre la aplicación Java y su entorno de ejecución. Los pasos clave son: **(1)** obtener la instancia `Runtime`, **(2)** invocar `exec()` con el comando a ejecutar, **(3)** leer el `InputStream` del objeto `Process` resultante para capturar su salida, y **(4)** llamar a `waitFor()` para bloquear el hilo actual hasta que el subproceso finalice.

1.3.2. Usando `ProcessBuilder`

La clase `ProcessBuilder` es una alternativa más moderna y potente a `Runtime`. Ofrece mayor control sobre la configuración del proceso, como la separación clara entre el comando y sus argumentos, la redirección de la entrada/salida y el establecimiento del directorio de trabajo.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EjecutorProcessBuilder {
    public static void main(String[] args) {
        try {
            // 1. Crear una instancia de ProcessBuilder, separando el comando de
sus argumentos.
            ProcessBuilder builder = new ProcessBuilder("ping", "8.8.8.8");

            // 2. Iniciar el proceso. Esto devuelve un objeto Process.
            System.out.println("Ejecutando ping a 8.8.8.8...");
            Process proceso = builder.start();

              // 3. Capturar y mostrar la salida estándar del proceso en tiempo
real.
                            BufferedReader reader = new BufferedReader(new
InputStreamReader(proceso.getInputStream()));
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }

               // 4. Esperar a que el proceso termine y obtener el código de
salida.
            int codigoSalida = proceso.waitFor();
            System.out.println("\nEl proceso finalizó con el código de salida: "
+ codigoSalida);

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

`ProcessBuilder` es la opción preferida porque su API permite una configuración más detallada y segura. El flujo es similar a `Runtime`: **(1)** se crea una instancia, **(2)** se llama a `start()`, **(3)** se

gestiona su salida, y **(4)** se espera su finalización con `waitFor()`. Separar el comando de sus argumentos **(1)** no es solo una cuestión de flexibilidad; es una práctica de seguridad crítica que previene vulnerabilidades de inyección de comandos, un vector de ataque común cuando los comandos se construyen como una sola cadena.

Mientras que los procesos proporcionan un robusto aislamiento a un alto coste, las aplicaciones modernas demandan una concurrencia de grano más fino dentro del espacio de memoria de un único programa. Este requisito de un paralelismo ligero y eficiente, donde las tareas pueden compartir datos directamente, nos conduce al concepto de hilos.

2. El Poder de la Programación Multihilo (Threads)

2.1. Hilos vs. Procesos: La Ventaja de la Eficiencia

Un **hilo** (o *thread*) es la unidad de ejecución más pequeña que un sistema operativo puede gestionar. A diferencia de los procesos, varios hilos pueden coexistir dentro de un mismo proceso. La diferencia fundamental radica en la gestión de la memoria:

- **Procesos:** Tienen espacios de memoria separados y aislados.
- **Hilos:** Comparten el mismo espacio de memoria y recursos (como archivos abiertos) del proceso que los contiene.

Esta memoria compartida hace que la comunicación entre hilos sea mucho más rápida y eficiente que la comunicación entre procesos. La programación multihilo ofrece beneficios clave:

- **Mejora del Rendimiento:** En sistemas con múltiples núcleos de CPU, los hilos pueden ejecutarse en paralelo, acelerando significativamente tareas computacionalmente intensivas como la renderización de gráficos o cálculos científicos.
- **Interfaces de Usuario Receptivas:** Permite que las tareas de larga duración (como descargar un archivo) se ejecuten en un hilo de fondo, mientras que el hilo principal mantiene la interfaz de usuario fluida y receptiva a las interacciones del usuario.
- **Procesamiento Concurrente:** Es ideal para aplicaciones que deben manejar múltiples operaciones simultáneamente, como un servidor web que atiende a varios clientes a la vez.

2.2. Conceptos Clave de la Concurrencia

La programación multihilo introduce desafíos únicos que deben ser gestionados para evitar errores.

- **Condición de Carrera (Race Condition):**
    - **¿Qué es?** Ocurre cuando dos o más hilos acceden y modifican un recurso compartido (como una variable o un archivo) al mismo tiempo, y el resultado final depende del orden impredecible en que se ejecutan.
    - **El Problema:** Puede llevar a resultados incorrectos, inconsistentes y difíciles de depurar. Por ejemplo, si dos hilos incrementan un contador simultáneamente, es posible que uno de los incrementos se pierda.
- **Sincronización:**
    - **¿Qué es?** Es el conjunto de mecanismos utilizados para coordinar la ejecución de múltiples hilos y garantizar un acceso seguro a los recursos compartidos.

- **La Solución:** El concepto central de la sincronización es la **exclusión mutua (mutex)**, que asegura que solo un hilo pueda acceder a una sección crítica del código (la que modifica el recurso compartido) en un momento dado.

2.3. Creación y Gestión de Hilos en Java

Java proporciona dos enfoques principales para crear y ejecutar hilos, permitiendo a los desarrolladores elegir el más adecuado según la estructura de su aplicación.

2.3.1. Implementando la Interfaz `Runnable`

Este es el enfoque más flexible y recomendado, ya que separa la tarea a realizar (`Runnable`) de la mecánica del hilo (`Thread`).

```
// 1. Crear una clase que implemente la interfaz Runnable.
class MiTarea implements Runnable {
    @Override
    public void run() {
        // El código dentro de run() es lo que ejecutará el hilo.
        for (int i = 1; i <= 5; i++) {
               System.out.println("Hilo " + Thread.currentThread().getId() + ": 
Contador " + i);
            try {
                Thread.sleep(1000); // Simula una tarea que toma tiempo.
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Buena práctica
            }
        }
    }
}

public class EjemploRunnable {
    public static void main(String[] args) throws InterruptedException {
        int numHilos = 8;
        Thread[] hilos = new Thread[numHilos];
        System.out.println("Iniciando " + numHilos + " hilos...");

            // 2. Crear instancias de Thread, pasando la tarea Runnable, y 
iniciarlos.
        for (int i = 0; i < numHilos; i++) {
            MiTarea tarea = new MiTarea();
            hilos[i] = new Thread(tarea);
            hilos[i].start(); // Invoca al método run() de la tarea.
        }

        // El hilo principal también puede hacer trabajo.
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hilo principal: Contador " + i);
            Thread.sleep(1000);
        }

        // 3. Esperar a que todos los hilos terminen usando join().
        for (int i = 0; i < numHilos; i++) {
            hilos[i].join();
        }

               System.out.println("Todos los hilos han finalizado. Programa 
terminado.");
    }
}
```

2.3.2. Extendiendo la Clase `Thread`

Este enfoque acopla la tarea directamente a la clase `Thread`. Es más simple para casos sencillos, pero menos flexible, ya que Java no permite la herencia múltiple.

```
// 1. Crear una clase que herede de Thread y sobreescriba el método run().
class MiHilo extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Ejecutando MiHilo " + this.getId() + ": " + i);
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class EjemploThread {
    public static void main(String[] args) throws InterruptedException {
        int numHilos = 8;
        MiHilo[] hilos = new MiHilo[numHilos];
              System.out.println("Iniciando " + numHilos + " instancias de
MiHilo...");

        // 2. Crear instancias de la clase e iniciarlas.
        for (int i = 0; i < numHilos; i++) {
            hilos[i] = new MiHilo();
            hilos[i].start();
        }

        // 3. Esperar a que todos los hilos terminen.
        for (int i = 0; i < numHilos; i++) {
            hilos[i].join();
        }

               System.out.println("Todos los hilos han finalizado. Programa
terminado.");
    }
}
```

2.4. Sincronización para Evitar el Caos: La Palabra Clave `synchronized`

Para implementar la exclusión mutua y prevenir condiciones de carrera, Java proporciona la palabra clave `synchronized`. Cuando un método o bloque de código es declarado `synchronized`, solo un hilo puede ejecutarlo a la vez sobre la misma instancia de objeto.

El siguiente ejemplo simula una tienda con un stock limitado y varios clientes intentando comprar productos concurrentemente.

```
class Tienda {
    private int stock = 10;

    // El método está sincronizado para evitar que múltiples clientes
    // modifiquen el stock simultáneamente.
    public synchronized void comprarProducto(String cliente, int cantidad) {
            System.out.println(cliente + " intenta comprar " + cantidad + "
productos.");
        if (stock >= cantidad) {
```

```
            // Simula el tiempo que toma procesar la compra
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            stock -= cantidad;
              System.out.println(" -> " + cliente + " compró " + cantidad + "
productos. Stock restante: " + stock);
        } else {
             System.out.println(" -> No hay suficiente stock para " + cliente +
".");
        }
    }
}

class Cliente implements Runnable {
    private Tienda tienda;
    private String nombre;
    private int cantidad;

    public Cliente(Tienda tienda, String nombre, int cantidad) {
        this.tienda = tienda;
        this.nombre = nombre;
        this.cantidad = cantidad;
    }

    @Override
    public void run() {
        tienda.comprarProducto(nombre, cantidad);
    }
}

public class PrincipalTienda {
    public static void main(String[] args) {
        Tienda miTienda = new Tienda();
        // Creamos varios clientes (hilos) que intentan comprar
        new Thread(new Cliente(miTienda, "Cliente 1", 5)).start();
        new Thread(new Cliente(miTienda, "Cliente 2", 3)).start();
        new Thread(new Cliente(miTienda, "Cliente 3", 4)).start();
    }
}
```

Sin `synchronized`, dos clientes podrían leer el stock al mismo tiempo, ambos ver que hay suficiente, y ambos realizar la compra, resultando en un stock negativo y datos inconsistentes. La sincronización garantiza que la verificación y actualización del stock sea una operación atómica. Esta palabra clave es el mecanismo principal de Java para implementar el concepto de **Mutex (Exclusión Mutua)** que definimos anteriormente.

Nota del experto: Aunque es potente, usar `synchronized` en métodos extensos puede crear cuellos de botella en el rendimiento. Para un control más granular, especialmente en escenarios de alta contención, los desarrolladores avanzados suelen recurrir a mecanismos más flexibles del paquete `java.util.concurrent.locks`.

2.5. Prioridades de los Hilos

Java permite asignar una prioridad a los hilos para influir en el planificador del sistema operativo. Se pueden usar las constantes de la clase `Thread`:

- `Thread.MIN_PRIORITY` (valor 1)
- `Thread.NORM_PRIORITY` (valor 5)
- `Thread.MAX_PRIORITY` (valor 10)

Un hilo con mayor prioridad tiene, teóricamente, más posibilidades de recibir tiempo de CPU.

Advertencia Crucial: La prioridad de un hilo es solo una **sugerencia** para el planificador del sistema operativo, **no una garantía**. El comportamiento real puede variar significativamente entre diferentes sistemas operativos e implementaciones de la Máquina Virtual de Java (JVM). Confiar en las prioridades para la corrección de un programa es un defecto de diseño. Para un ordenamiento determinista, siempre se deben usar constructos de sincronización adecuados como `synchronized`, `Locks` o `Semaphores`.

Hemos establecido cómo los hilos se comunican eficientemente a través de la memoria compartida, un paradigma potente pero arriesgado que se gestiona mediante la sincronización. Ahora, escalaremos este concepto hacia afuera: ¿cómo logran una comunicación fiable los procesos completamente aislados, a menudo en máquinas totalmente diferentes? Esto requiere pasar de la memoria compartida a una abstracción basada en la red: el socket.

3. Comunicación a Través de la Red

3.1. El Fundamento de la Comunicación: Los Sockets

Un **socket de red** es un punto final de comunicación bidireccional entre dos programas que se ejecutan en una red. Es una abstracción que permite a una aplicación enviar y recibir datos como si estuviera escribiendo o leyendo de un archivo local.

Un socket se define por la combinación de tres componentes:

- **Dirección IP:** Identifica de forma única a un dispositivo en la red (p. ej., `192.168.1.10`).
- **Número de Puerto:** Identifica la aplicación o servicio específico en ese dispositivo (p. ej., `80` para HTTP).
- **Protocolo de Transporte:** Define las reglas de la comunicación (p. ej., `TCP` o `UDP`).

Por ejemplo, un socket para un servidor web podría ser `192.168.1.10:80 TCP`.

3.2. Modelos de Comunicación en Red

Existen diferentes arquitecturas o modelos para estructurar la comunicación entre aplicaciones en red.

Modelo	Descripción Clave	Caso de Uso Típico
Cliente-Servidor	Un **servidor** centralizado provee servicios y espera peticiones. Múltiples **clientes** inician la comunicación solicitando esos servicios.	Aplicaciones web (navegador como cliente, servidor web como servidor).
Peer-to-Peer (P2P)	No hay un servidor central. Cada nodo (**par**) actúa tanto como cliente como servidor, comunicándose directamente con otros pares.	Sistemas de intercambio de archivos (BitTorrent), algunas aplicaciones de mensajería.
Publicación/	Un intermediario (**broker**) gestiona la comunicación. Los **publicadores** envían	Redes sociales (recibir notificaciones de un hashtag),

Suscripción	mensajes a temas, y los **suscriptores** reciben los mensajes de los temas a los que están suscritos.	sistemas de mensajería en tiempo real.

3.3. Protocolos de Transporte: TCP vs. UDP

TCP y UDP son los dos protocolos de transporte más comunes en Internet, y ofrecen diferentes garantías de servicio.

Característica	TCP (Protocolo de Control de Transmisión)	UDP (Protocolo de Datagramas de Usuario)
Fiabilidad	**Alta.** Garantiza que los datos lleguen sin errores y en el orden correcto.	**Baja.** No garantiza la entrega ni el orden de los paquetes.
Tipo de Conexión	**Orientado a conexión.** Establece una conexión fiable mediante un **"saludo de tres vías" (three-way handshake)** antes de transferir datos.	**Sin conexión.** Envía paquetes (datagramas) sin establecer una conexión previa.
Velocidad	**Más lento,** debido a la sobrecarga de la gestión de la conexión y la verificación de errores.	**Más rápido,** ya que no tiene la sobrecarga de establecer conexiones ni de confirmar entregas.
Garantía de Entrega	**Garantizada.** Si un paquete se pierde, se retransmite.	**No garantizada.** Los paquetes pueden perderse o llegar desordenados.
Caso de Uso	Navegación web (HTTP), correo electrónico (SMTP), transferencia de archivos (FTP).	Transmisión de video en vivo (streaming), juegos en línea, VoIP (donde la velocidad es más crítica que la pérdida ocasional de datos).

3.4. Programación de Sockets en Java

Java proporciona un robusto conjunto de clases en el paquete `java.net` para la programación de comunicaciones en red, facilitando la creación de aplicaciones cliente y servidor.

3.4.1. Ejemplo TCP: Cliente y Servidor

La comunicación TCP se basa en la clase `ServerSocket` para el servidor (que espera conexiones) y `Socket` para el cliente (que inicia la conexión).

Código del Servidor TCP

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

public class ServidorTCP {
    public static void main(String[] args) {
        try {
            // 1. ServerSocket espera conexiones en el puerto 12345.
            ServerSocket serverSocket = new ServerSocket(12345);
            System.out.println("Servidor esperando conexiones...");
```

```
            // 2. accept() bloquea hasta que un cliente se conecta y devuelve un
Socket para la comunicación.
            Socket clientSocket = serverSocket.accept();
                                System.out.println("Cliente conectado desde: " +
clientSocket.getInetAddress());

            // 3. Obtener flujos para leer del cliente y escribir al cliente.
                                BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
              PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);

            String mensajeCliente = in.readLine();
            System.out.println("Mensaje recibido: " + mensajeCliente);
                   out.println("Hola, cliente. Mensaje recibido."); // Enviar
respuesta.

            // 4. Cerrar conexiones.
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En el servidor, el flujo es: **(1)** crear un `ServerSocket` en un puerto, **(2)** llamar a `accept()` para esperar a un cliente, **(3)** usar los flujos del `Socket` devuelto para comunicarse, y **(4)** cerrar los recursos.

Código del Cliente TCP

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

public class ClienteTCP {
    public static void main(String[] args) {
        try {
               // 1. Crear un Socket para conectarse al servidor en localhost,
puerto 12345.
            Socket socket = new Socket("localhost", 12345);
            System.out.println("Conectado al servidor.");

            // 2. Obtener flujos para escribir al servidor y leer del servidor.
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                                BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            // 3. Enviar un mensaje y recibir la respuesta.
            out.println("Hola, servidor!");
            String respuestaServidor = in.readLine();
            System.out.println("Respuesta del servidor: " + respuestaServidor);

            // 4. Cerrar la conexión.
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

El cliente es más directo: **(1)** instancia un `Socket` con la IP y puerto del servidor, **(2)** obtiene los flujos, **(3)** se comunica, y **(4)** cierra el socket.

3.4.2. Ejemplo UDP: Emisor y Receptor

La comunicación UDP es "sin conexión" y se basa en el envío de paquetes de datos (datagramas) utilizando las clases `DatagramSocket` y `DatagramPacket`.

Código del Receptor UDP (Servidor)

```
import java.net.*;

public class ReceptorUDP {
    public static void main(String[] args) {
        try {
            // 1. Crear un DatagramSocket para escuchar en el puerto 12345.
            DatagramSocket socket = new DatagramSocket(12345);
            byte[] buffer = new byte[1024];
            System.out.println("Receptor UDP esperando paquetes...");

            // 2. Crear un DatagramPacket para recibir los datos.
                   DatagramPacket paqueteRecibido = new DatagramPacket(buffer,
buffer.length);

            // 3. receive() bloquea hasta que llega un paquete.
            socket.receive(paqueteRecibido);

            // 4. Extraer la información del paquete y mostrarla.
                    String mensaje = new String(paqueteRecibido.getData(), 0,
paqueteRecibido.getLength());
            System.out.println("Paquete recibido: " + mensaje);

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

El receptor UDP: **(1)** abre un `DatagramSocket` en un puerto, **(2)** prepara un `DatagramPacket` como buffer, **(3)** bloquea en `receive()` esperando datos, y **(4)** procesa el paquete recibido.

Código del Emisor UDP (Cliente)

```
import java.net.*;

public class EmisorUDP {
    public static void main(String[] args) {
        try {
            // 1. Crear un DatagramSocket para enviar paquetes.
            DatagramSocket socket = new DatagramSocket();
            InetAddress direccionServidor = InetAddress.getByName("localhost");
            String mensaje = "Hola desde el emisor UDP!";
            byte[] buffer = mensaje.getBytes();

               // 2. Crear un DatagramPacket con el mensaje, la dirección y el
puerto de destino.
                    DatagramPacket paqueteEnviado = new DatagramPacket(buffer,
buffer.length, direccionServidor, 12345);
```

```
            // 3. Enviar el paquete.
            socket.send(paqueteEnviado);
            System.out.println("Paquete enviado.");

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

El emisor UDP: **(1)** abre un `DatagramSocket` (sin puerto fijo), **(2)** crea un `DatagramPacket` que contiene los datos, la IP y el puerto del destinatario, y **(3)** lo envía con `send()`.

3.4.3. Servidor Multicliente: Combinando Hilos y Sockets

Un servidor real debe poder atender a múltiples clientes simultáneamente. Bloquearse esperando a un solo cliente no es escalable. La solución ideal es combinar sockets con hilos, una aplicación práctica del **Procesamiento Concurrente** que vimos en la sección 2.1. El servidor principal acepta conexiones y, por cada cliente, lanza un nuevo hilo que se encarga de toda la comunicación con ese cliente específico.

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

// 1. Hilo que gestiona la comunicación con un solo cliente.
class ClienteHandler implements Runnable {
    private Socket clientSocket;
    public ClienteHandler(Socket socket) { this.clientSocket = socket; }

    @Override
    public void run() {
        try (
              PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);
                                BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()))
        ) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                                          System.out.println("Recibido de " +
clientSocket.getInetAddress() + ": " + inputLine);
                out.println("Eco: " + inputLine); // Responder al cliente
            }
        } catch (IOException e) {
            System.out.println("Error con el cliente: " + e.getMessage());
        } finally {
                        try { clientSocket.close(); } catch (IOException e)
{ e.printStackTrace(); }
        }
    }
}

// 2. Servidor principal que acepta conexiones y delega a los hilos.
public class ServidorMulticliente {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(12345);
             System.out.println("Servidor multicliente iniciado en el puerto
12345...");

        while (true) {
```

```
            // 3. Aceptar una nueva conexión de cliente (bloqueante).
            Socket clientSocket = serverSocket.accept();
                        System.out.println("Nuevo cliente conectado: " +
clientSocket.getInetAddress());

                // 4. Crear y lanzar un nuevo hilo para gestionar al cliente,
liberando al hilo principal.
            Thread clientThread = new Thread(new ClienteHandler(clientSocket));
            clientThread.start();
        }
    }
}
```

La arquitectura es elegante: **(1)** se define una clase `Runnable` (`ClienteHandler`) para encapsular la lógica de comunicación. **(2)** El servidor principal entra en un bucle infinito donde **(3)** espera conexiones con `accept()`. Por cada conexión aceptada, **(4)** crea un nuevo hilo `ClienteHandler` y lo inicia, volviendo inmediatamente a esperar la siguiente conexión.

Hemos visto cómo los procesos forman la base de la ejecución, los hilos permiten la concurrencia eficiente dentro de un proceso, y las redes posibilitan la comunicación entre procesos distribuidos. Para dominar estos temas, es esencial tener un vocabulario claro y preciso.

4. Glosario de Términos Clave

- **Sincronización:** Mecanismo utilizado para garantizar que múltiples hilos puedan acceder y modificar variables compartidas de manera segura y predecible, evitando condiciones de carrera.
- **Mutex (Mutual Exclusion):** Un mecanismo de bloqueo que permite que solo un proceso o hilo a la vez acceda a un recurso compartido o sección crítica. Otros deben esperar hasta que el recurso sea liberado.
- **Deadlock (Interbloqueo):** Una situación en la que dos o más hilos o procesos se bloquean mutuamente, cada uno esperando un recurso que el otro posee, lo que resulta en un punto muerto donde ninguno puede continuar. Por ejemplo, en el caso de la tienda, si un cliente intentara comprar en la `TiendaA` y luego en la `TiendaB` (bloqueando A y luego B), mientras otro cliente lo hace en orden inverso (bloqueando B y luego A), podrían quedarse esperando indefinidamente el uno al otro.
- **Condición de Carrera (Race Condition):** Un error que ocurre cuando el resultado de una operación depende del orden impredecible en que dos o más hilos acceden y manipulan un recurso compartido.
- **Socket:** Un punto final de una conexión de red, identificado por una dirección IP y un número de puerto, que permite a las aplicaciones enviar y recibir datos a través de una red.
- **TCP (Protocolo de Control de Transmisión):** Un protocolo de transporte orientado a conexión que garantiza la entrega fiable, ordenada y sin errores de los datos.
- **UDP (Protocolo de Datagramas de Usuario):** Un protocolo de transporte sin conexión que es más rápido pero menos fiable que TCP, ya que no establece conexión previa ni garantiza la entrega o el orden de los paquetes.