Paso 1: La Estructura Básica del Servidor y la Petición de Parámetros

Objetivo de este paso:

Nuestro primer objetivo es crear el "esqueleto" de la clase `ServidorUDP`. Este esqueleto no hará nada de red todavía, pero se encargará de una tarea fundamental descrita en el documento del proyecto: **preguntar al usuario los parámetros de la simulación (N, V, S) y validar que son correctos**.

El Código:

Vamos a crear el archivo `ServidorUDP.java`. Dentro, escribiremos el método `main`, que es el punto de entrada de cualquier programa en Java, y usaremos la clase `Scanner` para leer lo que el usuario escriba por la consola.
`

```
// Importamos las clases que vamos a necesitar. Scanner es para leer desde la consola.
import java.util.Scanner;

public class ServidorUDP {

    // Parámetros de la simulación que serán leídos del usuario.
    // Los declaramos como 'static' para que sean accesibles desde el método 'main'.
    private static int N; // Número total de clientes
    private static int V; // Número de vecinos por grupo
    private static int S; // Número de iteraciones

    public static void main(String args[]) {
        // 1. CREAR UN OBJETO SCANNER
        // System.in es el flujo de entrada estándar (el teclado).
        // Scanner "envuelve" ese flujo para que podamos leer datos de forma sencilla.
        Scanner escaner = new Scanner(System.in);

        System.out.println("--- CONFIGURACIÓN DEL SERVIDOR ---");

        // 2. PEDIR LOS PARÁMETROS
        System.out.print("Introduce el número total de clientes (N): ");
        N = escaner.nextInt();

        System.out.print("Introduce el número de $vecinos$ por grupo (V): ");
        V = escaner.nextInt();

        System.out.print("Introduce el número de iteraciones (S): ");
        S = escaner.nextInt();

        // 3. VALIDAR LOS PARÁMETROS
        // El documento especifica que N debe ser múltiplo de V[cite: 55].
```

```
        // El operador '%' (módulo) nos da el resto de una división.
        // Si el resto de N/V es 0, significa que es una división exacta.
        if (N % V != 0) {
            System.out.println("Error: N debe ser múltiplo de V.");
            escaner.close(); // Cerramos el escaner para liberar recursos
            return; // 'return' termina la ejecución del método main, y por
tanto del programa.
        }

        System.out.println("\nParámetros correctos. El servidor puede
continuar...");
        // Aquí, en los próximos pasos, empezará la lógica de red.

        // 4. CERRAR EL SCANNER
        // Es una buena práctica cerrar siempre los recursos que abrimos.
        escaner.close();
    }
}
```

Explicando los Conceptos (API de Java):

- **`Scanner`** : Es una clase del paquete `java.util` que nos facilita enormemente la lectura de datos desde diferentes fuentes. En este caso, `new Scanner(System.in)` le dice que lea desde la entrada estándar del sistema, que por defecto es el teclado.
- **`escaner.nextInt()`** : Este método específico de `Scanner` pausa el programa, espera a que el usuario escriba una secuencia de dígitos y pulse `Enter` , y entonces convierte esos dígitos en un número de tipo `int` .

Resumen de lo que hemos logrado:

Al final de este paso, tendrás un programa `ServidorUDP.java` que puedes compilar y ejecutar. El programa se iniciará, te pedirá los datos correctamente, comprobará que `N` sea múltiplo de `V` y luego terminará. Es la base sólida sobre la que construiremos toda la lógica de red.

Paso 2 (Revisado): Almacenar Información del Cliente y Abrir el Socket de Red

Objetivo de este paso:

Ahora que el servidor sabe los parámetros de la simulación, necesita dos cosas más antes de poder aceptar conexiones:

1. Una **estructura para "recordar"** la información de cada cliente que se conecte. Usaremos un `record` para esto, ya que es la forma más moderna y concisa.
2. Una forma de **"abrir la puerta" a la red** para poder escuchar los mensajes que lleguen. Usaremos un `DatagramSocket` para ello.

El Código:

Vamos a modificar nuestro archivo `ServidorUDP.java` para añadirle la estructura `InfoCliente` , el mapa `clientesConectados` y el `DatagramSocket` que se quedará escuchando en un puerto.

```java
// Importamos las clases de red (net) y de utilidades (util) que vamos a usar
import java.io.IOException;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap; // Usaremos esta implementación de Map

public class ServidorUDP {

    // --- AÑADIMOS ESTAS DOS PARTES NUEVAS ---

    /**
     * Estructura para almacenar la información de cada cliente conectado.
     * Usamos un 'record' por su concisión. Automáticamente crea una clase inmutable
     * con constructor, getters (ej. .idCliente()), y otros métodos útiles.
     * @param idCliente ID único que le asignaremos.
     * @param idGrupo ID del grupo de trabajo al que pertenecerá.
     * @param direccion La dirección IP del cliente (objeto InetAddress).
     * @param puerto El puerto desde el que nos habla el cliente.
     */
    private record InfoCliente(int idCliente, int idGrupo, InetAddress direccion, int puerto) {}

    /**
     * Mapa para almacenar a los clientes. La clave es el ID del cliente (Integer)
     * y el valor es el objeto InfoCliente con todos sus datos.
     * Es nuestro "archivador" o "memoria" de clientes.
     */
    private static Map<Integer, InfoCliente> clientesConectados = new ConcurrentHashMap<>();

    // --- EL RESTO DEL CÓDIGO QUE YA TENÍAMOS ---

    private static int N, V, S;

    public static void main(String args[]) {
        Scanner escaner = new Scanner(System.in);
        // ... (el código para pedir N, V, S y la validación se queda igual) ...

        System.out.println("\nParámetros correctos. El servidor puede continuar...");

        // --- AÑADIMOS LA LÓGICA DE RED ---

        System.out.print("Inicializando servidor UDP... ");

        // Usamos un bloque "try-with-resources". Esto asegura que el socket
        // se cierre automáticamente al final, incluso si hay un error.
        try (DatagramSocket socketUDP = new DatagramSocket(10578)) {

            System.out.println("\t[OK]");
```

```
            System.out.println("Servidor escuchando en puerto " +
socketUDP.getLocalPort());

            // En los siguientes pasos, aquí dentro irá el bucle para aceptar
clientes.
            System.out.println("Servidor listo. Esperando conexiones de
clientes...");

        } catch (IOException e) {
            // Esta excepción puede saltar si hay un problema al abrir el
socket
            // (por ejemplo, si el puerto ya está en uso por otro programa).
            System.err.println("Error al iniciar el servidor: " +
e.getMessage());
            e.printStackTrace();
        }

        escaner.close(); // Movemos el cierre del escaner al final de todo
    }
}
```

Explicando los Conceptos (API de Java):

- **`record InfoCliente`** : Como decidimos, usamos `record` . Es una línea de código que equivale a una clase entera con constructor y métodos para acceder a los datos. Es perfecto para guardar la "ficha" de cada cliente en el servidor.
- **`Map<Integer, InfoCliente>`** : Sigue siendo nuestro "archivador". Guarda pares `(ID_Cliente, Ficha_del_Cliente)` . Esto nos permitirá, más adelante, buscar la ficha de un cliente de forma súper rápida usando su ID.
- **`DatagramSocket`** : Esta es la clase clave para la comunicación UDP en Java. Piensa en ella como un **portal o una puerta de enlace**. La línea `new DatagramSocket(10578)` hace dos cosas:
  1. Crea el objeto `socketUDP` .
  2. Le dice al sistema operativo: "Oye, a partir de ahora, cualquier paquete de datos UDP que llegue a esta máquina y que vaya dirigido al puerto 10578, entrégamelo a mí, a este programa de Java". Este proceso se llama "enlazar" (binding) el socket a un puerto.
- **`try-with-resources`** : La sintaxis `try (DatagramSocket socketUDP = ...)` es una característica de Java que nos ahorra trabajo. Gestiona automáticamente los recursos que se abren, como las conexiones de red. Cuando el código dentro del bloque `try` termina (ya sea normalmente o por un error), Java se encarga de llamar a `socketUDP.close()` por nosotros. Esto es fundamental para no dejar puertos abiertos innecesariamente.
- **`IOException`** : Java nos obliga a prepararnos para lo peor. Una `IOException` (Excepción de Entrada/Salida) es la forma que tiene Java de decir: "puedo fallar por algo que no es tu culpa". Por ejemplo, si intentas usar el puerto 10578 pero otro programa ya lo está usando, el sistema operativo te dará un error. Al poner el código en un bloque `try-catch` , estamos manejando ese posible error de forma controlada en lugar de dejar que el programa se cierre bruscamente.

Resumen de lo que hemos logrado:

Ahora nuestro servidor es más robusto. Tiene una "memoria" (`clientesConectados`) lista para ser llenada y una "oreja" (`socketUDP`) que ya está escuchando activamente en el puerto 10578 de la red. Todavía no estamos procesando los mensajes que llegan, pero la infraestructura principal ya está montada y lista.

Paso 3: La Fase de Registro - Aceptando Conexiones de Clientes

Objetivo de este paso:

Según el documento del proyecto, el servidor debe primero esperar a que los `N` clientes se conecten antes de que la simulación pueda empezar. En este paso, vamos a implementar exactamente ese bucle de registro. Haremos que el servidor:

1. Espere a que llegue un paquete.
2. Extraiga la dirección y puerto del cliente que lo envió.
3. Le asigne un `idCliente` y un `idGrupo` .
4. Guarde su "ficha" (`InfoCliente`) en el mapa `clientesConectados` .
5. Repita esto `N` veces.

El Código:

Añadiremos un bucle `for` dentro de nuestro bloque `try-with-resources` . Dentro de este bucle, usaremos dos clases nuevas: `DatagramPacket` para manejar los datos y el método `socket.receive()` para esperar.

```
// Las importaciones y la primera parte del código se mantienen igual.

public class ServidorUDP {

    // ... (record InfoCliente, mapa clientesConectados, variables N, V, S)
...

    public static void main(String args[]) {
        // ... (petición de parámetros y validación) ...

        System.out.println("\nParámetros correctos. El servidor puede
continuar...");
        System.out.print("Inicializando servidor UDP... ");

        try (DatagramSocket socketUDP = new DatagramSocket(10578)) {
            System.out.println("\t[OK]");
            System.out.println("Servidor escuchando en puerto " +
socketUDP.getLocalPort());

            // --- AÑADIMOS EL BUCLE DE REGISTRO ---

            //
============================================================
            // FASE 1: INICIO Y REGISTRO DE CLIENTES
            //
============================================================
            System.out.println("\nFASE DE INICIO: Esperando " + N + "
clientes...");
```

```
            for (int i = 0; i < N; i++) {
                // 1. PREPARAR UN PAQUETE VACÍO PARA RECIBIR DATOS
                // Creamos un buffer (un espacio de memoria) de 1024 bytes.
                byte[] bufer = new byte[1024];
                // Creamos un DatagramPacket que usa este buffer para guardar
lo que reciba.
                DatagramPacket peticion = new DatagramPacket(bufer,
bufer.length);

                // 2. ESPERAR A RECIBIR UN PAQUETE (LLAMADA BLOQUEANTE)
                // El programa se detendrá aquí hasta que un cliente envíe un
paquete a este puerto.
                socketUDP.receive(peticion);
                System.out.println(" -> Conexión entrante recibida...");

                // 3. EXTRAER LA INFORMACIÓN DEL REMITENTE
                InetAddress direccionCliente = peticion.getAddress();
                int puertoCliente = peticion.getPort();

                // 4. ASIGNAR IDS Y CREAR LA FICHA DEL CLIENTE
                // El documento dice que los vecinos se asignan por orden de
conexión.
                int idCliente = i;
                int idGrupo = i / V; // División entera. Para V=10, clientes
0-9 -> grupo 0, 10-19 -> grupo 1, etc.

                // Creamos el objeto InfoCliente con los datos obtenidos y
asignados.
                InfoCliente nuevoCliente = new InfoCliente(idCliente, idGrupo,
direccionCliente, puertoCliente);

                // 5. GUARDAR AL CLIENTE EN EL MAPA
                clientesConectados.put(idCliente, nuevoCliente);
                System.out.println("   Cliente " + idCliente + " (Grupo " +
idGrupo + ") registrado.");
            }

            System.out.println("\nRegistro completo. Todos los " + N + "
clientes están conectados.");

        } catch (IOException e) {
            System.err.println("Error en la comunicación de red: " +
e.getMessage());
            e.printStackTrace();
        }
        escaner.close();
    }
}
```

Explicando los Conceptos (API de Java):

- **`DatagramPacket`** : Piensa en esta clase como el **sobre** para tus cartas (datos). Para recibir, creas un "sobre vacío" (`new DatagramPacket(bufer, bufer.length)`) donde el cartero

(`socket`) pueda meter la carta. El `bufer` es un array de `byte` , que es la forma en que Java maneja datos binarios para la red.

- **`socketUDP.receive(peticion)`** : Este es uno de los métodos más importantes. Es una **llamada bloqueante**. Esto significa que la ejecución de tu programa se **congela** en esta línea y no continuará hasta que un paquete UDP llegue al puerto 10578. Cuando llega un paquete, el método `receive` copia los datos del paquete dentro del `bufer` de tu `DatagramPacket` y también rellena la información del remitente (su IP y puerto).
- **`peticion.getAddress()` y `peticion.getPort()`** : Una vez que `receive()` ha terminado, nuestro objeto `peticion` está lleno. Con estos métodos, podemos preguntarle: "¿Quién te ha enviado?". `getAddress()` nos devuelve un objeto `InetAddress` (la IP) y `getPort()` nos devuelve un `int` (el puerto). Esta es la única forma que tiene el servidor de saber a quién responder.
- **`clientesConectados.put(idCliente, nuevoCliente)`** : Este es el método de los `Map` para añadir una nueva entrada. Le decimos: "guarda este objeto `nuevoCliente` y asócialo a la clave `idCliente` ".

Resumen de lo que hemos logrado:

Hemos construido el "recepcionista" del servidor. Ahora es capaz de esperar a que lleguen los `N` clientes, uno por uno, tomarles los datos (IP y puerto), crearles una ficha en el sistema (`InfoCliente`) y guardarla en el "archivador" (`clientesConectados`). Al final de este bucle, el servidor conoce a todos los participantes de la simulación y dónde encontrarlos.

Paso 4: El Pistoletazo de Salida - Iniciando la Simulación

Objetivo de este paso:

El documento del proyecto es muy claro en este punto: "Cuando ya se hayan creado los N sockets (...), entonces el servidor enviará un mensaje a todos indicando que comience la simulación".

Nuestro objetivo es exactamente ese: recorrer la lista de clientes que acabamos de registrar y enviarles a **todos** un mensaje para que sepan que pueden empezar a trabajar. Esto actúa como una barrera de sincronización.

El Código:

Justo después del bucle de registro, añadiremos un nuevo bucle. Esta vez, en lugar de recibir, vamos a **enviar** paquetes. Fíjate en cómo creamos el `DatagramPacket` ahora, ya que es diferente.

```
// ... (el código anterior se mantiene igual hasta el final del bucle de registro) ...

            System.out.println("\nRegistro completo. Todos los " + N + " clientes están conectados.");

            // --- AÑADIMOS ESTA NUEVA SECCIÓN ---

            //
============================================================
            // FASE 2: SIMULACIÓN - SEÑAL DE INICIO
```

```
            //
================================================================
            System.out.println("\nFASE DE SIMULACIÓN: Enviando señal de inicio
a todos los clientes...");

            String senalInicio = "INICIAR_SIMULACION";
            byte[] datosInicio = senalInicio.getBytes(); // Convertimos el
mensaje a bytes

            // Recorremos todos los clientes que hemos guardado en nuestro
mapa.
            // .values() nos da una colección de todos los objetos
InfoCliente.
            for (InfoCliente cliente : clientesConectados.values()) {

                // Creamos un paquete PARA ENVIAR.
                // Le damos los datos, su longitud, y la dirección y puerto
del destinatario.
                DatagramPacket paqueteInicio = new DatagramPacket(
                    datosInicio,
                    datosInicio.length,
                    cliente.direccion(), // IP del cliente que guardamos en el
Paso 3
                    cliente.puerto()     // Puerto del cliente que guardamos
en el Paso 3
                );

                // Enviamos el paquete a través de nuestro socket.
                socketUDP.send(paqueteInicio);
            }

            System.out.println("Señal de inicio enviada. La simulación
comienza ahora.");
            // Aquí empezará el bucle principal de la simulación.

        } catch (IOException e) {
            // ...
        }
        // ...
    }
}
```

Explicando los Conceptos (API de Java):

- **`mensaje.getBytes()`** : La comunicación por red no envía texto, envía bytes. Este método de la clase `String` convierte nuestra cadena de texto (ej: "INICIAR_SIMULACION") en un array de bytes (`byte[]`) para que pueda ser enviado por la red.
- **Constructor de `DatagramPacket` para Envío**: ¡Esto es clave! Ahora usamos un constructor diferente: `new DatagramPacket(byte[] bufer, int longitud, InetAddress direccion, int puerto)`
  - **`bufer`** : Los datos que queremos enviar.
  - **`longitud`** : Cuántos bytes de ese `bufer` queremos enviar.
  - **`direccion`** : El objeto `InetAddress` del destinatario.

    - `puerto` : El número de puerto del **destinatario**. Como ves, al enviar, debemos especificar explícitamente a dónde va el "sobre". Al recibir, el "sobre" ya venía con la dirección del remitente.
- `socketUDP.send(paquete)` : Este es el método opuesto a `receive()` . Toma un `DatagramPacket` que ya tiene datos y un destino, y lo entrega al sistema operativo para que lo envíe por la red. A diferencia de `receive()` , `send()` no suele ser bloqueante; envía los datos y el programa continúa inmediatamente.
- `clientesConectados.values()` : Este método de los `Map` es muy útil. Nos devuelve una colección con todos los valores (en nuestro caso, todos los objetos `InfoCliente`), permitiéndonos recorrerlos fácilmente con un bucle `for-each` sin preocuparnos por sus claves.

Paso 5: El Bucle Principal - El Corazón del Servidor

Objetivo de este paso:

El servidor necesita un bucle que se mantenga activo durante toda la simulación, recibiendo y procesando todos los mensajes que los clientes envíen (coordenadas, ACKs, etc.). Este bucle no se repetirá un número fijo de veces, sino que debe continuar **hasta que todos los N clientes hayan terminado**.

En este paso, construiremos la estructura de este bucle `while` y haremos que, por ahora, simplemente reciba cualquier mensaje, identifique quién lo envió y lo muestre en la consola.

El Código:

Después de enviar la señal de inicio, añadiremos una variable para contar cuántos clientes han finalizado y el bucle `while` que contendrá toda la lógica de retransmisión. También crearemos una función de ayuda, `buscarClientePorDireccion` , para identificar a los remitentes.

```
// ... (código anterior hasta el envío de la señal de inicio) ...

            System.out.println("Señal de inicio enviada. La simulación
comienza ahora.");

            // --- AÑADIMOS EL BUCLE PRINCIPAL DE LA SIMULACIÓN ---

            int clientesFinalizados = 0;

            // Este bucle se ejecutará mientras el número de clientes que han
terminado
            // sea menor que el número total de clientes.
            while (clientesFinalizados < N) {

                // 1. PREPARAMOS TODO PARA RECIBIR UN PAQUETE (igual que en el
registro)
                byte[] bufer = new byte[1024];
                DatagramPacket peticion = new DatagramPacket(bufer,
bufer.length);

                // 2. ESPERAMOS UN MENSAJE (LLAMADA BLOQUEANTE)
                socketUDP.receive(peticion);
```

```
                // 3. IDENTIFICAMOS AL REMITENTE
                // Usaremos una función auxiliar para mantener el código
limpio
                InfoCliente clienteRemitente =
buscarClientePorDireccion(peticion.getAddress(), peticion.getPort());

                // Si el mensaje viene de una IP/puerto que no reconocemos, lo
ignoramos.
                if (clienteRemitente == null) {
                    System.err.println("Mensaje recibido de un cliente
desconocido. Se ignora.");
                    continue; // 'continue' salta al inicio de la siguiente
iteración del bucle.
                }

                // 4. CONVERTIMOS EL MENSAJE A TEXTO Y LO MOSTRAMOS
                // Es crucial usar getLength() para no leer "basura" del resto
del buffer.
                String mensajeRecibido = new String(peticion.getData(), 0,
peticion.getLength());

                System.out.println(
                    "Mensaje recibido del Cliente " +
clienteRemitente.idCliente() + ": \"" + mensajeRecibido + "\""
                );

                // Por ahora, no hacemos nada con el mensaje.
                // En el siguiente paso, aquí irá la lógica para decidir qué
hacer.
            }

        } catch (IOException e) {
            // ...
        }
        // ...
    }

    // --- AÑADIMOS NUESTRA PRIMERA FUNCIÓN AUXILIAR ---

    /**
     * Busca en nuestro mapa y devuelve la InfoCliente correspondiente a una
IP y puerto.
     * @param direccion La dirección IP del cliente a buscar.
     * @param puerto El puerto del cliente a buscar.
     * @return El objeto InfoCliente si se encuentra, o null si no existe.
     */
    private static InfoCliente buscarClientePorDireccion(InetAddress
direccion, int puerto) {
        // Recorremos todos los clientes guardados en nuestro mapa
        for (InfoCliente cliente : clientesConectados.values()) {
            // Comparamos la dirección y el puerto del cliente guardado
            // con la dirección y puerto del paquete que acabamos de recibir.
            if (cliente.direccion().equals(direccion) && cliente.puerto() ==
```

```
puerto) {
                return cliente; // Si coinciden, lo hemos encontrado.
            }
        }
        return null; // Si el bucle termina y no lo hemos encontrado,
devolvemos null.
    }
}
```

Explicando los Conceptos (API de Java y Lógica):

- **Bucle `while`** : A diferencia del `for` que usamos para el registro (que se repetía un número conocido de veces `N`), el `while` es perfecto para cuando la condición de fin es más abstracta. `while (clientesFinalizados < N)` se traduce como: "Mientras no hayan terminado todos los clientes, sigue ejecutando este bucle". Esto hace que nuestro servidor sea flexible y espere lo que haga falta.
- **`new String(bytes, offset, longitud)`** : Esta es la forma correcta de convertir un `byte[]` que hemos recibido de la red a un `String` .
  - `peticion.getData()` : Nos da el `bufer` completo (1024 bytes).
  - `0` : Le decimos que empiece a convertir desde el principio del `bufer` .
  - `peticion.getLength()` : **¡Crucial!** El cliente puede que solo haya enviado un mensaje de 20 bytes. Este método nos dice la longitud real de los datos recibidos. Si no lo usáramos, intentaríamos convertir los 1024 bytes completos, incluyendo 1004 bytes de "basura", lo que daría lugar a errores.
- **Funciones Auxiliares**: El método `buscarClientePorDireccion` es nuestro primer ejemplo de cómo mantener el código limpio. En lugar de poner ese bucle `for` dentro del `while` , lo extraemos a una función con un nombre descriptivo. Esto hace que el `while` principal sea mucho más fácil de leer.

Resumen de lo que hemos logrado:

Hemos construido la sala de control principal del servidor. Ahora tiene un bucle que se ejecuta durante toda la simulación, capaz de recibir cualquier mensaje que llegue. Todavía no sabe **qué hacer** con los mensajes, pero ya sabe **cómo recibirlos** e **identificar al remitente**. Es como un operador de radio que escucha el canal, anota quién llama y cuál es el mensaje, preparándose para actuar.

Paso 6: Definiendo el Protocolo de Comunicación (Nuestro 'Idioma')

Objetivo de este paso:

El objetivo es diseñar un **protocolo simple** para que el cliente y el servidor se entiendan. Un protocolo no es más que un conjunto de reglas sobre el formato de los mensajes. Piénsalo como definir la gramática de un idioma. Decidiremos qué palabras clave usar y cómo separar la información dentro de un mismo mensaje.

Esto nos permitirá, en el siguiente paso, crear la lógica del servidor que lea el "comando" de cada mensaje y actúe en consecuencia.

El Diseño del Protocolo:

Para que sea fácil de programar y depurar, usaremos un protocolo basado en texto simple. La regla será: `COMANDO;DATO1;DATO2;...`

Usaremos el punto y coma (`;`) como separador, ya que es un carácter que no usaremos en nuestros datos (que serán principalmente números).

Aquí está la lista de todos los tipos de mensajes que nuestros programas intercambiarán:

1. **Registro (Cliente → Servidor)**
   - **Mensaje:** `REGISTER`
   - **Propósito:** Un cliente nuevo le dice al servidor "¡Hola, estoy aquí!". Es el primer mensaje que envía un cliente.
2. **Confirmación de Registro (Servidor → Cliente)**
   - **Mensaje:** `REGISTRADO;{idCliente};{idGrupo};{V}`
   - **Ejemplo:** `REGISTRADO;7;0;10`
   - **Propósito:** El servidor le responde al cliente con la "ficha" que le ha creado: su ID único, el ID de su grupo de trabajo y el número de vecinos que tiene.
3. **Inicio de Simulación (Servidor → Todos los Clientes)**
   - **Mensaje:** `INICIAR_SIMULACION`
   - **Propósito:** El mensaje que el servidor envía a todos a la vez para dar el pistoletazo de salida.
4. **Envío de Coordenadas (Cliente → Servidor)**
   - **Mensaje:** `COORDS;{idCliente};{valorCoordenada}`
   - **Ejemplo:** `COORDS;7;85`
   - **Propósito:** El mensaje principal de la simulación. El cliente envía sus coordenadas (para simplificar, enviaremos solo un valor numérico) e incluye su propio ID para que el servidor sepa quién lo envía.
5. **Reconocimiento (Vecino → Servidor)**
   - **Mensaje:** `ACK;{idDelClienteOriginal}`
   - **Ejemplo:** `ACK;7`
   - **Propósito:** Cuando a un vecino le llega un mensaje de `COORDS` reenviado por el servidor, debe responder. Su respuesta (`ACK`) debe incluir el ID del cliente que envió las coordenadas originalmente, para que el servidor sepa a quién pertenece este reconocimiento.
6. **Reconocimiento Reenviado (Servidor → Cliente Original)**
   - **Mensaje:** `ACK_DE;{idDelVecinoQueRespondio}`
   - **Ejemplo:** `ACK_DE;12`
   - **Propósito:** El servidor reenvía el ACK al cliente original, informándole de qué vecino (`idDelVecinoQueRespondio`) ya le ha respondido.
7. **Finalización (Cliente → Servidor)**
   - **Mensaje:** `DONE;{tiempoPromedio}`
   - **Ejemplo:** `DONE;345.82`
   - **Propósito:** Cuando un cliente termina sus `S` iteraciones, envía este mensaje al servidor con su tiempo medio de respuesta para que el servidor pueda hacer los cálculos finales.

¿Cómo lo implementamos en el código?

Todavía no escribiremos la lógica completa, pero sí la parte que "desmonta" el mensaje. Lo haremos con el método `.split()` de los `String`.

```
// Dentro del bucle while del Paso 5, cambiaremos esta parte:
// ANTES:
// String mensajeRecibido = new String(peticion.getData(), 0,
peticion.getLength());
// System.out.println("Mensaje recibido...");

// AHORA:
String mensajeRecibido = new String(peticion.getData(), 0,
peticion.getLength());
String[] partes = mensajeRecibido.split(";"); // "Desmontamos" el mensaje
usando ';' como separador
String comando = partes[0]; // La primera parte es siempre el comando

System.out.println(
    "Comando '" + comando + "' recibido del Cliente " +
clienteRemitente.idCliente()
);

// En el siguiente paso, aquí irá un 'switch' para actuar según el 'comando'.
```

Explicando los Conceptos (API de Java):

- **`String.split(String separador)`**: Este es un método increíblemente útil de la clase `String`. Toma una cadena de texto y la "corta" en trozos usando el separador que le indiques. Devuelve un **array de Strings** (`String[]`) con los trozos.
  - Si `mensajeRecibido` es `"COORDS;7;85"`, entonces `mensajeRecibido.split(";")` devuelve un array: `["COORDS", "7", "85"]`.
  - Así, `partes[0]` será `"COORDS"`, `partes[1]` será `"7"`, y `partes[2]` será `"85"`.

Resumen de lo que hemos logrado:

Hemos diseñado el lenguaje que usarán nuestros programas para comunicarse. Ya no son solo mensajes al azar, ahora tienen una estructura y un significado. Esto nos permite crear una lógica clara en el servidor para tomar decisiones basadas en el tipo de mensaje recibido. Hemos sentado las las bases para la "inteligencia" de nuestro servidor.

Paso 7: Implementando la Lógica del Servidor (El Cerebro del Router)

Objetivo de este paso:

Este es el paso donde implementamos la funcionalidad principal del servidor descrita en el documento del proyecto. Usando el protocolo que definimos, vamos a leer el `comando` de cada mensaje y ejecutar una acción diferente para cada uno:

- Si es `COORDS`, lo reenviamos a todos los vecinos del remitente.
- Si es `ACK`, lo reenviamos al cliente que envió las coordenadas originales.
- Si es `DONE`, actualizamos nuestro contador para saber que un cliente ha terminado.

El Código:

Vamos a trabajar dentro del bucle `while`. Usaremos una estructura `switch` para organizar el código de forma limpia. También implementaremos la lógica dentro de las funciones auxiliares que ya habíamos esbozado.

```
// Dentro del main, en el bucle while, reemplazamos la parte final

            // ... bucle while (clientesFinalizados < N) ...
            while (clientesFinalizados < N) {
                // ... recibir el paquete y buscar al cliente remitente ...

                String mensajeRecibido = new String(peticion.getData(), 0,
peticion.getLength());
                String[] partes = mensajeRecibido.split(";");
                String comando = partes[0];

                // --- AÑADIMOS EL SWITCH PARA PROCESAR EL COMANDO ---
                switch (comando) {
                    case "COORDS":
                        // Si el comando es COORDS, llamamos a la función que
lo gestiona.
                        gestionarCoordenadas(socketUDP, peticion.getData(),
peticion.getLength(), clienteRemitente);
                        break; // 'break' es importante para salir del switch.

                    case "ACK":
                        // Si es un ACK, llamamos a su respectiva función.
                        gestionarAck(socketUDP, partes, clienteRemitente);
                        break;

                    case "DONE":
                        // La lógica de DONE es corta, la podemos poner aquí
mismo.
                        clientesFinalizados++;
                        System.out.println(
                            " -> Cliente " + clienteRemitente.idCliente() + "
ha terminado. (" + clientesFinalizados + "/" + N + ")"
                        );
                        break;

                    default:
                        // Si llega un comando que no reconocemos.
                        System.err.println("Comando desconocido '" + comando +
"' recibido. Se ignora.");
                }
            }
            // Cuando el bucle while termine, aquí irá la fase de cálculo de
resultados.

// ... (El resto del main y el método buscarClientePorDireccion se mantienen)
...

// --- AHORA IMPLEMENTAMOS LA LÓGICA DE LAS FUNCIONES AUXILIARES ---
```

```java
    /**
     * Gestiona la recepción de un mensaje de coordenadas de un cliente.
     * Su única función es retransmitir este mensaje a todos los demás
clientes
     * que pertenecen al mismo grupo que el remitente (sus vecinos).
     */
    private static void gestionarCoordenadas(DatagramSocket socket, byte[]
datosOriginales, int longitud, InfoCliente remitente) throws IOException {
        System.out.println("Recibido COORDS del Cliente " +
remitente.idCliente() + ". Retransmitiendo al grupo " + remitente.idGrupo() +
"...");

        // Recorremos todos los clientes conectados para encontrar a los
vecinos.
        for (InfoCliente vecino : clientesConectados.values()) {

            // La condición para ser un vecino es:
            // 1. Pertenecer al mismo grupo que el remitente.
            // 2. NO ser el propio remitente.
            if (vecino.idGrupo() == remitente.idGrupo() && vecino.idCliente()
!= remitente.idCliente()) {

                // Creamos un nuevo paquete para enviar los datos originales
al vecino.
                DatagramPacket paqueteReenviado = new
DatagramPacket(datosOriginales, longitud, vecino.direccion(),
vecino.puerto());
                socket.send(paqueteReenviado);
            }
        }
    }

    /**
     * Gestiona la recepción de un mensaje de reconocimiento (ACK) de un
cliente.
     * Reenvía el ACK al cliente que envió originalmente las coordenadas.
     */
    private static void gestionarAck(DatagramSocket socket, String[] partes,
InfoCliente remitenteAck) throws IOException {
        // Según nuestro protocolo, el ACK tiene el formato "ACK;
{idDelClienteOriginal}"
        // Así que partes[1] contiene el ID del destinatario final de este
ACK.
        int idRemitenteOriginal = Integer.parseInt(partes[1]);

        // Buscamos en el mapa la "ficha" del cliente original usando su ID.
        // ¡Esta es la gran ventaja de usar un Map! El acceso es casi
instantáneo.
        InfoCliente remitenteOriginal =
clientesConectados.get(idRemitenteOriginal);

        if (remitenteOriginal != null) {
            // Si encontramos al cliente, le preparamos y enviamos el ACK.
```

```
            // Protocolo: "ACK_DE;{idDelVecinoQueRespondio}"
            String mensajeAckReenviado = "ACK_DE;" + remitenteAck.idCliente();
            byte[] datosAck = mensajeAckReenviado.getBytes();
            DatagramPacket paqueteAck = new DatagramPacket(datosAck,
datosAck.length, remitenteOriginal.direccion(), remitenteOriginal.puerto());

            socket.send(paqueteAck);
            System.out.println("   -> Reenviado ACK del Cliente " +
remitenteAck.idCliente() + " al Cliente " + idRemitenteOriginal);
        }
    }
```

Explicando los Conceptos (API de Java y Lógica):

- **`switch (variable)`** : Es una estructura de control que permite ejecutar diferentes bloques de código según el valor de una variable. Es una alternativa más limpia a usar múltiples `if-else if` . Compara el valor de `comando` con cada `case` y ejecuta el código correspondiente.
- **Lógica de Vecinos**: La condición `vecino.idGrupo() == remitente.idGrupo() && vecino.idCliente() != remitente.idCliente()` es la implementación directa de la regla del proyecto. Así nos aseguramos de que el mensaje se reenvía solo a los compañeros de grupo y de que el cliente no se reenvía el mensaje a sí mismo.
- **`Integer.parseInt(String)`** : Los datos que recibimos en el mensaje son siempre texto (`String`). Cuando recibimos el ID en `"ACK;7"` , `partes[1]` es el `String` `"7"` . Para poder usarlo como número (por ejemplo, para buscar en el mapa), necesitamos convertirlo a `int` . Esta función hace exactamente eso.
- **`mapa.get(clave)`** : Este es el método más importante de un `Map` . Le das una clave (el `idRemitenteOriginal`) y te devuelve el valor asociado (el objeto `InfoCliente` completo) de forma muy eficiente, sin necesidad de recorrer toda la lista como hacíamos en `buscarClientePorDireccion` .

Resumen de lo que hemos logrado:

¡El cerebro de nuestro servidor ya está completo! Ya no solo escucha y entiende, sino que ahora actúa. Sabe exactamente qué hacer con cada tipo de mensaje: reenvía coordenadas a los vecinos, devuelve los ACKs a quien corresponde y cuenta a los clientes que van terminando. Hemos implementado la lógica de enrutamiento y gestión que define el proyecto.

Paso 8: La Fase Final - Cálculo y Presentación de Resultados

Objetivo de este paso:

Según el documento del proyecto, una vez que todos los clientes han terminado, el servidor debe realizar una última tarea: "calcula el tiempo medio de respuesta de cada grupo y el tiempo medio de respuesta de todo el sistema para esa simulación".

Nuestro objetivo es implementar esta fase de cálculo. Para ello, necesitamos modificar un poco el paso anterior para que no solo contemos a los clientes que terminan, sino que también **almacenemos los tiempos medios** que nos envían.

El Código:

Primero, declararemos un par de variables antes del bucle `while` para ir sumando los tiempos. Luego, ampliaremos el `case "DONE"` y finalmente, después de que el bucle `while` termine, añadiremos el código que hace los cálculos y los muestra.

```
// ... (código hasta justo antes del bucle `while`) ...

            System.out.println("Señal de inicio enviada. La simulación
comienza ahora.");

            // --- AÑADIMOS ESTAS VARIABLES PARA LOS CÁLCULOS ---
            int clientesFinalizados = 0;
            Map<Integer, Double> tiemposPorGrupo = new HashMap<>(); // Para
sumar los tiempos de cada grupo
            double tiempoTotalSistema = 0.0; // Para sumar todos los tiempos

            // Bucle principal para procesar mensajes de la simulación
            while (clientesFinalizados < N) {
                // ... (recepción del paquete y búsqueda del remitente) ...

                // ... (split del mensaje y obtención del comando) ...

                switch (comando) {
                    case "COORDS":
                        gestionarCoordenadas(socketUDP, peticion.getData(),
peticion.getLength(), clienteRemitente);
                        break;
                    case "ACK":
                        gestionarAck(socketUDP, partes, clienteRemitente);
                        break;
                    case "DONE":
                        // --- AMPLIAMOS LA LÓGICA DEL CASE "DONE" ---
                        clientesFinalizados++;

                        // Según nuestro protocolo "DONE;{tiempoPromedio}",
partes[1] es el tiempo.
                        // Lo convertimos de String a un número Double.
                        double tiempoMedioCliente =
Double.parseDouble(partes[1]);

                        // Lo sumamos al total del sistema.
                        tiempoTotalSistema += tiempoMedioCliente;

                        // Lo sumamos al total de su grupo correspondiente en
el mapa.
                        // .merge() es un método útil: si no hay entrada para
el grupo, la crea;
                        // si ya existe, suma el valor nuevo al existente.
                        tiemposPorGrupo.merge(clienteRemitente.idGrupo(),
tiempoMedioCliente, Double::sum);

                        System.out.println(
                            " -> Cliente " + clienteRemitente.idCliente() + "
ha terminado. (" + clientesFinalizados + "/" + N + ")"
                        );
```

```
                        break;
                    default:
                        System.err.println("Comando desconocido '" + comando +
"' recibido. Se ignora.");
                }
            }

            // --- AÑADIMOS LA FASE FINAL DE CÁLCULO ---
            // Este código se ejecuta solo cuando el bucle 'while' ha
terminado.

            //
============================================================
            // FASE 3: CÁLCULO DE RESULTADOS
            //
============================================================
            System.out.println("\nFASE DE CÁLCULO: Todos los clientes han
terminado.");
            System.out.println("--- RESULTADOS FINALES ---");

            // Usamos printf para formatear la salida y que se vea bonita.
            // %.4f significa "formatea este número decimal con 4 cifras
decimales". \n es un salto de línea.
            System.out.printf("Tiempo medio de respuesta de todo el sistema:
%.4f ms\n", tiempoTotalSistema / N);

            // Recorremos el mapa de tiempos por grupo para mostrar cada
resultado.
            // .entrySet() nos da cada par (idGrupo, sumaDeTiempos) del mapa.
            for (Map.Entry<Integer, Double> entrada :
tiemposPorGrupo.entrySet()) {
                System.out.printf(
                    "  - Tiempo medio del Grupo %d: %.4f ms\n",
                    entrada.getKey(),           // La clave (el id del grupo)
                    entrada.getValue() / V      // El valor (la suma de
tiempos) dividido por el número de vecinos
                );
            }

        } catch (IOException e) {
            // ...
        }
        // ...
    }

// ... (resto de funciones auxiliares sin cambios) ...
```

Explicando los Conceptos (API de Java y Lógica):

- **`Double.parseDouble(String)`** : Al igual que `Integer.parseInt` , este método convierte un `String` que representa un número decimal (ej: `"345.82"`) en un tipo de dato numérico `double` con el que podemos hacer cálculos.
- **`mapa.merge(clave, valor, funcion)`** : Este es un método muy elegante de los `Map` . Intenta hacer esto:

1. Busca la `clave` (nuestro `idGrupo`).
2. Si la clave no existe, la crea y le asigna el `valor` (`tiempoMedioCliente`).
3. Si la clave ya existe, no la sobreescribe, sino que ejecuta la `funcion` que le pasamos. La función `Double::sum` es una forma corta de decir `(valorAntiguo, valorNuevo) -> valorAntiguo + valorNuevo`. Es decir, suma el tiempo del cliente actual al total que ya teníamos para ese grupo.

- **`mapa.entrySet()`**: Cuando queremos recorrer un mapa y necesitamos tanto la clave como el valor de cada entrada, usamos `.entrySet()`. Nos devuelve un conjunto de objetos `Map.Entry`, y cada `Entry` tiene los métodos `.getKey()` y `.getValue()` para acceder a sus partes.
- **`System.out.printf()`**: A diferencia de `println`, que simplemente imprime texto, `printf` (print formatted) nos permite crear una "plantilla" y rellenar los huecos con variables, dándoles un formato específico. Es ideal para mostrar resultados numéricos de forma ordenada.

Resumen de lo que hemos logrado:

¡El servidor está completo! Hemos finalizado su última tarea. Ahora, después de que todos los clientes terminan, el servidor es capaz de procesar los resultados que le han enviado, calcular las estadísticas finales tal y como exige el proyecto, y presentarlas de forma clara al usuario. Hemos completado el ciclo de vida entero del servidor, desde el inicio hasta el reporte final.

Construyendo el Cliente - Paso 9: El Esqueleto de `PersonaUDP`

Objetivo de este paso:

El objetivo es crear la estructura fundamental de la clase que representará a un único cliente. Según el documento del proyecto, los clientes se ejecutan de forma **asíncrona**. La forma más natural de lograr esto en Java es haciendo que cada cliente sea un **Hilo (`Thread`)** de ejecución independiente.

En este paso, crearemos la clase `PersonaUDP` que hereda de `Thread`, le daremos las variables que necesitará para vivir y su constructor para inicializarlas.

El Código:

Crea un nuevo archivo llamado `PersonaUDP.java`.

```
// Importamos las clases de red que sabemos que vamos a necesitar.
import java.net.DatagramSocket;
import java.net.InetAddress;

// Hacemos que nuestra clase "herede" de la clase Thread.
// Esto significa que un objeto PersonaUDP ES un Hilo y puede ejecutarse en paralelo.
public class PersonaUDP extends Thread {

    // --- VARIABLES (ATRIBUTOS) DEL CLIENTE ---

    // Información del servidor y la simulación (se recibe en el constructor)
    private String ipServidor;
    private int puertoServidor;
```

```
    private int V_vecinos;
    private int S_iteraciones;

    // Información propia del cliente (se la dará el servidor durante el
registro)
    private int idCliente;
    private int idGrupo;

    // Herramienta de red del cliente
    private DatagramSocket socket;

    /**
     * Constructor. Se llama cuando se crea un nuevo objeto PersonaUDP.
     * Su única función es recibir los parámetros y guardarlos en las
variables de la clase.
     */
    public PersonaUDP(String ip, int puerto, int v, int s) {
        this.ipServidor = ip;
        this.puertoServidor = puerto;
        this.V_vecinos = v;
        this.S_iteraciones = s;
        // Las variables idCliente, idGrupo y socket aún no tienen valor.
        // Se inicializarán cuando el hilo se ejecute.
    }

    /**
     * Este es el corazón del hilo. Cuando se llame a .start() sobre un objeto
PersonaUDP,
     * el código que pongamos aquí dentro es el que se ejecutará en paralelo.
     */
    @Override
    public void run() {
        System.out.println("Soy un nuevo hilo cliente, ¡listo para empezar!");
        // En los siguientes pasos, aquí irá toda la lógica:
        // 1. Crear el socket.
        // 2. Registrarse en el servidor.
        // 3. Esperar la señal de inicio.
        // 4. Bucle de S iteraciones.
        // 5. Enviar resultados.
        // 6. Cerrar socket.
    }
}
```

Explicando los Conceptos (API de Java y Lógica):

- **`extends Thread`** : Esta es una de las palabras clave más importantes de la programación concurrente en Java. Le estamos diciendo: "Mi clase `PersonaUDP` no es una clase cualquiera, es una especialización de la clase `Thread` de Java". Al hacer esto, nuestra clase hereda todo el comportamiento de un hilo.
- **`@Override public void run()`** : Este método es la consecuencia directa de heredar de `Thread` . La clase `Thread` tiene un método `run()` que por defecto no hace nada. Nosotros lo **sobrescribimos** (`@Override` es una anotación que le dice al compilador "estoy sobrescribiendo un método, avísame si me equivoco en el nombre") para meter ahí toda la

lógica que queremos que nuestro cliente ejecute. Este método es el punto de entrada del hilo.

- **Constructor (`public PersonaUDP(...)`)**: Es un método especial que se ejecuta cuando creamos un nuevo objeto (`new PersonaUDP(...)`). Lo usamos para pasarle al objeto la información que necesita para vivir (como la dirección del servidor) y guardarla en sus variables internas (atributos). La palabra clave `this` se usa para diferenciar entre el parámetro del constructor (`ip`) y la variable de la clase (`this.ipServidor`).
- **Variables de Instancia (Atributos)**: Las variables como `ipServidor` , `idCliente` , etc., se declaran al principio de la clase. A diferencia de las variables dentro de un método, estas "pertenecen" al objeto y guardan su estado durante toda su vida.

Resumen de lo que hemos logrado:

Hemos creado el plano de nuestro cliente. La clase `PersonaUDP` ya existe, sabe heredar de `Thread` para poder ejecutarse en paralelo, y tiene un constructor para recibir los parámetros de la simulación. Su método `run()` está vacío, pero es el escenario donde, en los próximos pasos, representaremos toda la obra: registro, simulación y finalización.

Paso 10: Registro del Cliente en el Servidor

Objetivo de este paso:

Lo primero que debe hacer un cliente al ejecutarse es "presentarse" al servidor. No puede empezar a enviar coordenadas sin más, porque el servidor no sabría quién es ni a qué grupo pertenece.

Por lo tanto, nuestro objetivo es implementar el **saludo inicial (handshake)**:

1. El cliente enviará el mensaje `REGISTER` al servidor.
2. El cliente se quedará esperando la respuesta.
3. Cuando reciba la respuesta del servidor (ej: `REGISTRADO;7;0;10`), guardará los IDs que el servidor le ha asignado.

El Código:

Trabajaremos dentro del método `run()` de la clase `PersonaUDP` . Añadiremos la creación del `DatagramSocket` , la lógica para enviar el mensaje de registro y la lógica para recibir y procesar la respuesta.

```
// Importamos más clases que vamos a necesitar
import java.io.IOException;
import java.net.*; // Contiene DatagramSocket, InetAddress, etc.

public class PersonaUDP extends Thread {

    // ... (variables y constructor sin cambios) ...

    @Override
    public void run() {
        // Usaremos un try-catch para manejar los posibles errores de red
(IOException)
        try {
```

```
            // 1. CREAR EL SOCKET DEL CLIENTE
            // Al crearlo sin número de puerto, el sistema operativo le asigna
uno libre.
            // Cada cliente tendrá su propio socket en un puerto diferente.
            socket = new DatagramSocket();

            // Obtenemos el objeto InetAddress del servidor a partir del
String con su IP.
            InetAddress direccionServidor = InetAddress.getByName(ipServidor);

            System.out.println("Cliente iniciado, intentando registrarse en el
servidor...");

            // --- LÓGICA DE REGISTRO ---

            // 2. ENVIAR EL MENSAJE DE REGISTRO
            String mensajeRegistro = "REGISTER";
            byte[] datosEnvio = mensajeRegistro.getBytes();
            DatagramPacket paqueteRegistro = new DatagramPacket(
                datosEnvio,
                datosEnvio.length,
                direccionServidor,
                puertoServidor
            );
            socket.send(paqueteRegistro);

            // 3. ESPERAR LA RESPUESTA DEL SERVIDOR
            byte[] buferRecepcion = new byte[1024];
            DatagramPacket paqueteRespuesta = new
DatagramPacket(buferRecepcion, buferRecepcion.length);

            socket.receive(paqueteRespuesta); // Se queda bloqueado aquí hasta
recibir respuesta

            // 4. PROCESAR LA RESPUESTA
            String respuesta = new String(paqueteRespuesta.getData(), 0,
paqueteRespuesta.getLength());

            // Usamos el protocolo que definimos:
"REGISTRADO;idCliente;idGrupo;V"
            String[] partes = respuesta.split(";");

            if (partes[0].equals("REGISTRADO")) {
                // Convertimos las partes de texto a números y las guardamos
                this.idCliente = Integer.parseInt(partes[1]);
                this.idGrupo = Integer.parseInt(partes[2]);
                this.V_vecinos = Integer.parseInt(partes[3]);

                System.out.println(
                    "[Cliente " + this.idCliente + "] ¡Registrado con éxito!
Soy del Grupo " + this.idGrupo
                );
            } else {
                System.err.println("Error: Respuesta inesperada del servidor:
```

```
" + respuesta);
                return; // Si la respuesta no es la esperada, terminamos la
ejecución.
            }

            // En el siguiente paso, aquí esperaremos la señal de inicio.

        } catch (IOException e) {
            System.err.println("Error de red en el cliente: " +
e.getMessage());
            e.printStackTrace();
        } finally {
            // El bloque 'finally' se ejecuta siempre, haya error o no.
            // Es el lugar ideal para asegurarse de cerrar el socket.
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }
        }
    }
}
```

Explicando los Conceptos (API de Java y Lógica):

- **`new DatagramSocket()`** : En el servidor, le dimos un puerto fijo para que los clientes supieran dónde encontrarlo. En el cliente, usamos el constructor sin argumentos. Esto le pide al sistema operativo: "dame cualquier puerto que esté libre". Esto es fundamental, porque si ejecutamos 50 clientes en la misma máquina, cada uno necesita su propio puerto de salida.
- **`InetAddress.getByName(ipServidor)`** : De nuevo, usamos este método para convertir el `String` con la IP del servidor en un objeto `InetAddress` que `DatagramPacket` puede usar.
- **Secuencia Enviar-Recibir (Handshake)**: Este patrón es muy común. El cliente envía una petición (`REGISTER`) y se queda inmediatamente a la espera (`socket.receive()`) de una respuesta. No continúa hasta que el servidor le contesta. Esto asegura que el cliente no empiece la simulación sin tener su identidad (`idCliente` , `idGrupo`).
- **`partes[0].equals("REGISTRADO")`** : Después de recibir y "desmontar" el mensaje, comprobamos que el comando sea el que esperábamos. Es una buena práctica para asegurarse de que la comunicación va por buen camino.
- **Bloque `finally`** : El código dentro de `finally` se ejecuta siempre al salir del bloque `try` , sin importar si la ejecución fue normal o si saltó una excepción. Por eso es el lugar perfecto para poner el código de limpieza, como `socket.close()` , para garantizar que nunca dejamos conexiones de red abiertas.

Resumen de lo que hemos logrado:

Nuestro cliente ha cobrado vida. Ya no es solo un esqueleto; ahora es capaz de realizar su primera tarea fundamental: presentarse al servidor y recibir su identidad. Ha establecido comunicación, ha seguido el protocolo de registro y ahora tiene la información (`idCliente` , `idGrupo`) que necesita para participar en la simulación. Sabe "quién es" en el sistema.

Paso 11: Esperando la Señal de Salida

Objetivo de este paso:

Después de registrarse, el cliente no puede empezar a enviar coordenadas por su cuenta. Tiene que esperar a que el servidor le dé la orden. Esto, como vimos en el Paso 4 del servidor, ocurre cuando el último de los `N` clientes se ha registrado.

Nuestro objetivo es añadir la lógica para que, justo después de registrarse, el cliente se quede esperando a recibir el mensaje `INICIAR_SIMULACION`.

El Código:

Añadiremos este nuevo bloque de código dentro del método `run()`, justo después de que la parte del registro haya terminado con éxito. Como verás, el código para recibir un mensaje es casi idéntico al que usamos en el paso anterior.

```
// Dentro del método run() de PersonaUDP

    @Override
    public void run() {
        try {
            // ... (Creación del socket y la dirección del servidor) ...

            // --- LÓGICA DE REGISTRO (del Paso 10, sin cambios) ---
            // ... (enviar "REGISTER" y recibir "REGISTRADO;...") ...

            if (partes[0].equals("REGISTRADO")) {
                // ... (guardar idCliente, idGrupo, etc.) ...
                System.out.println(
                    "[Cliente " + this.idCliente + "] ¡Registrado con éxito!
Soy del Grupo " + this.idGrupo
                );
            } else {
                // ... (manejo de error) ...
            }

            // --- AÑADIMOS LA LÓGICA DE ESPERA DE INICIO ---

            System.out.println("[Cliente " + idCliente + "] Esperando señal de
inicio del servidor...");

            // 1. PREPARAMOS UN PAQUETE VACÍO PARA RECIBIR (igual que antes)
            byte[] buferInicio = new byte[1024];
            DatagramPacket paqueteInicio = new DatagramPacket(buferInicio,
buferInicio.length);

            // 2. ESPERAMOS EL MENSAJE DE INICIO (LLAMADA BLOQUEANTE)
            socket.receive(paqueteInicio);

            // 3. PROCESAMOS LA SEÑAL
            String senal = new String(paqueteInicio.getData(), 0,
paqueteInicio.getLength());

            if (senal.equals("INICIAR_SIMULACION")) {
                System.out.println("[Cliente " + idCliente + "] ¡Señal
```

```
recibida! Comenzando simulación...");
            } else {
                System.err.println("[Cliente " + idCliente + "] Error: Se
esperaba señal de inicio, pero se recibió: " + senal);
                return; // Terminamos si la señal no es la correcta.
            }

            // Aquí, en los próximos pasos, comenzará el bucle de S
iteraciones.

        } catch (IOException e) {
            // ...
        } finally {
            // ...
        }
    }
```

Explicando los Conceptos (API de Java y Lógica):

- **Sincronización**: Este es el concepto clave de este paso. Aunque todos nuestros clientes son hilos independientes y asíncronos, este punto del código actúa como una **barrera de sincronización**. Todos los `N` clientes llegarán a su propia línea `socket.receive(paqueteInicio)` en momentos diferentes, pero ninguno podrá continuar más allá de esa línea hasta que el servidor, que ha esperado a que todos se registren, les envíe el mensaje de `INICIAR_SIMULACION`. Esto garantiza que la simulación comience de forma justa para todos.
- **Reutilización de Patrones**: Observa que el patrón de código para recibir este mensaje (`crear buffer -> crear DatagramPacket -> llamar a socket.receive() -> convertir a String`) es exactamente el mismo que usamos para recibir la confirmación de registro. Este es el patrón estándar para recibir cualquier mensaje UDP. Repetirlo ayuda a consolidar el aprendizaje.

Resumen de lo que hemos logrado:

Nuestro cliente ya es "obediente". No solo se presenta al servidor, sino que ahora sabe esperar pacientemente la orden para empezar. Ha superado con éxito las dos primeras fases de comunicación con el servidor: el registro individual y la espera sincronizada. Está en la línea de salida, con el motor en marcha, listo para correr.

Paso 12: El Reto Asíncrono y el Hilo de Escucha

Objetivo de este paso:

El documento del proyecto especifica que los clientes se ejecutan de forma **ASÍNCRONA**. Esto crea un problema complejo:

Imagina que eres el **Cliente A**. Acabas de enviar tus coordenadas y estás esperando a que te lleguen los `ACKs` de tus vecinos. Tu programa está bloqueado en una línea `socket.receive()`, esperando.

Pero, ¿qué pasa si mientras esperas, tu vecino, el **Cliente B**, envía *sus* coordenadas? El servidor las recibirá y te las reenviará a ti (Cliente A). Pero si tu programa está bloqueado esperando un

`ACK`, **no puede** recibir al mismo tiempo las coordenadas del Cliente B para responderle. Un único hilo no puede hacer dos cosas a la vez.

La solución es dividir el trabajo. Nuestro cliente `PersonaUDP` tendrá **dos hilos internos**:

1. **Hilo Principal:** El que ejecuta el método `run()`. Será el "Jefe de Tareas". Su trabajo será llevar la cuenta de las `S` iteraciones, enviar las coordenadas y esperar a que se complete su propia tarea (recibir todos los ACKs).
2. **Hilo de Escucha:** Un nuevo hilo que crearemos. Será el "Recepcionista". Su **único trabajo**, 24/7, será estar sentado en `socket.receive()` esperando a que llegue CUALQUIER paquete. No le importa qué es, solo lo recibe.

¿Y cómo se comunican? Usaremos una "bandeja de entrada" compartida y segura: una `ConcurrentLinkedQueue`. El Recepcionista (Hilo de Escucha) recoge todos los mensajes y los que son ACKs para el Jefe de Tareas, los deja en esta bandeja. El Jefe de Tareas solo tendrá que mirar la bandeja para ver si sus ACKs han llegado.

El Código:

Vamos a modificar `PersonaUDP.java` para añadir la bandeja de entrada (`ConcurrentLinkedQueue`), un nuevo método para el hilo de escucha (`escucharAlServidor`), y la línea en `run()` que pone en marcha a este nuevo hilo.

```
// Añadimos las importaciones necesarias
import java.io.IOException;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentLinkedQueue; // ¡Importante!

public class PersonaUDP extends Thread {
    // ... (variables existentes sin cambios) ...

    // --- AÑADIMOS LA "BANDEJA DE ENTRADA" SEGURA PARA HILOS ---
    /**
     * Cola segura para hilos para comunicar los ACKs recibidos por el hilo de escucha
     * al hilo principal de la simulación.
     */
    private final ConcurrentLinkedQueue<String> acksRecibidos = new ConcurrentLinkedQueue<>();

    // ... (constructor sin cambios) ...

    @Override
    public void run() {
        try {
            // ... (Creación del socket, registro y espera de la señal de inicio, sin cambios) ...

            System.out.println("[Cliente " + idCliente + "] ¡Señal recibida! Comenzando simulación...");

            // --- AÑADIMOS LA CREACIÓN Y ARRANQUE DEL HILO DE ESCUCHA ---
```

```
            // Creamos un nuevo hilo y le decimos que su tarea es ejecutar
            // el código que está en nuestro método escucharAlServidor.
            Thread hiloEscucha = new Thread(this::escucharAlServidor);
            hiloEscucha.start(); // ¡Ponemos al recepcionista a trabajar!

            // Aquí, en el siguiente paso, irá el bucle 'for' de S iteraciones
del "Jefe de Tareas".

            // Al final de la simulación, tendremos que detener al hilo de
escucha.
            // Lo interrumpimos para que su bucle 'while' termine.
            hiloEscucha.interrupt();


        } catch (IOException e) {
            // ...
        } finally {
            // ...
        }
    }

    // --- AÑADIMOS EL NUEVO MÉTODO PARA EL HILO DE ESCUCHA ---

    /**
     * Este método se ejecuta en un hilo secundario. Su única función es
     * escuchar continuamente los paquetes entrantes del servidor y
procesarlos.
     */
    private void escucharAlServidor() {
        // Este bucle se ejecuta "para siempre" hasta que el hilo principal lo
interrumpa.
        while (!Thread.currentThread().isInterrupted()) {
            try {
                byte[] bufer = new byte[1024];
                DatagramPacket paqueteRecibido = new DatagramPacket(bufer,
bufer.length);

                socket.receive(paqueteRecibido); // Se queda bloqueado aquí
hasta recibir ALGO

                String mensaje = new String(paqueteRecibido.getData(), 0,
paqueteRecibido.getLength());

                System.out.println("[Cliente " + idCliente + " - Hilo Escucha]
Mensaje recibido: " + mensaje);

                // En el siguiente paso, aquí pondremos la lógica para decidir
                // si el mensaje es un COORDS (y responder ACK) o un ACK_DE (y
meterlo en la cola).

            } catch (IOException e) {
                // Si el socket se cierra o hay un error, el hilo debe parar.
                if (socket.isClosed() ||
Thread.currentThread().isInterrupted()) {
```

```
                    break; // Sale del bucle while.
                }
                System.err.println("Error de red en el hilo de escucha: " +
e.getMessage());
            }
        }
        System.out.println("[Cliente " + idCliente + " - Hilo Escucha] Hilo de
escucha terminado.");
    }

    // ... (el resto de métodos que creemos irán aquí) ...
}
```

Explicando los Conceptos (API de Java y Lógica):

- **Programación Concurrente**: Es la capacidad de un programa para ejecutar múltiples tareas "a la vez". Java lo logra con `Thread` s. Nuestro cliente ahora es un programa concurrente.
- **`new Thread(this::escucharAlServidor)`** : Esta es la forma moderna en Java de crear un hilo. Le estamos diciendo: "Crea un nuevo objeto `Thread` cuya tarea (`Runnable`) será ejecutar el método `escucharAlServidor` de esta misma instancia (`this`)".
- **`hiloEscucha.start()`** : Este método es el que realmente pone en marcha el nuevo hilo. A partir de esta línea, tenemos dos flujos de ejecución paralelos: el método `run()` continúa, y al mismo tiempo, el método `escucharAlServidor()` empieza a ejecutarse.
- **`ConcurrentLinkedQueue`** : Es una clase del paquete de concurrencia de Java. Es una cola (el primero que entra es el primero que sale) diseñada para ser usada por múltiples hilos a la vez sin que se corrompan los datos. Es la herramienta perfecta para nuestra "bandeja de entrada" compartida.
- **`!Thread.currentThread().isInterrupted()`** : Esta es la forma correcta y segura de mantener un hilo vivo. El bucle se repite mientras el hilo "no haya sido interrumpido". Más tarde, desde el hilo principal, llamaremos a `hiloEscucha.interrupt()` para poner una "bandera" de interrupción a `true` , y en la siguiente comprobación, el bucle `while` terminará de forma ordenada.

Resumen de lo que hemos logrado:

Hemos resuelto el problema más difícil del diseño del cliente. Hemos dividido el trabajo en dos hilos: uno que actúa (el principal) y otro que escucha (el secundario). Hemos introducido una "bandeja de entrada" segura (`ConcurrentLinkedQueue`) para que se comuniquen. Aunque el hilo de escucha todavía no procesa los mensajes de forma inteligente, la estructura asíncrona fundamental ya está en su sitio.

Paso 13: Dando Inteligencia al Hilo de Escucha

Objetivo de este paso:

El objetivo es implementar la lógica dentro del método `escucharAlServidor()` . Este hilo debe ser capaz de diferenciar entre los dos tipos de mensajes que puede recibir durante la simulación:

1. Un mensaje de **`COORDS`** (reenviado por el servidor desde un vecino): Según el documento del proyecto, cuando un cliente recibe esto, debe contestar con un mensaje de

reconocimiento (`ACK`) al servidor.

2. Un mensaje de **`ACK_DE`** (un reconocimiento de un vecino para un mensaje que *nosotros* enviamos): Este es el mensaje que nuestro hilo principal está esperando. El hilo de escucha debe tomarlo y depositarlo en la "bandeja de entrada" (`acksRecibidos`) para que el hilo principal lo vea.

El Código:

Nos centraremos exclusivamente en el método `escucharAlServidor()` . Reemplazaremos el `System.out.println` genérico por un `if-else if` que implemente la lógica descrita.

```
// Dentro de la clase PersonaUDP

    /**
     * Este método se ejecuta en un hilo secundario. Su única función es
     * escuchar continuamente los paquetes entrantes del servidor y
procesarlos.
     */
    private void escucharAlServidor() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                byte[] bufer = new byte[1024];
                DatagramPacket paqueteRecibido = new DatagramPacket(bufer,
bufer.length);

                socket.receive(paqueteRecibido); // Bloqueante: espera un
paquete

                String mensaje = new String(paqueteRecibido.getData(), 0,
paqueteRecibido.getLength());
                String[] partes = mensaje.split(";");
                String comando = partes[0];

                // --- AÑADIMOS LA LÓGICA DE DECISIÓN ---

                if (comando.equals("COORDS")) {
                    // Si es un mensaje de coordenadas de un vecino...

                    // 1. EXTRAEMOS EL ID DEL REMITENTE ORIGINAL
                    // Protocolo: "COORDS;{idCliente};{valor...}" -> partes[1]
es el ID
                    int idRemitenteOriginal = Integer.parseInt(partes[1]);

                    // 2. PREPARAMOS EL MENSAJE DE ACK
                    // Protocolo: "ACK;{idDelClienteOriginal}"
                    String mensajeAck = "ACK;" + idRemitenteOriginal;
                    byte[] datosAck = mensajeAck.getBytes();

                    // 3. ENVIAMOS EL ACK DE VUELTA AL SERVIDOR
                    // El paquete se envía a la dirección de donde vino el
mensaje COORDS (el servidor).
                    DatagramPacket paqueteAck = new DatagramPacket(
                        datosAck,
                        datosAck.length,
```

```
                        paqueteRecibido.getAddress(), // IP del remitente
(servidor)
                        paqueteRecibido.getPort()      // Puerto del remitente
(servidor)
                    );
                    socket.send(paqueteAck);

                } else if (comando.equals("ACK_DE")) {
                    // Si es un reconocimiento para un mensaje que nosotros
enviamos...

                    // Simplemente lo añadimos a la cola compartida.
                    // El hilo principal se encargará de leerlo de ahí.
                    acksRecibidos.add(mensaje);
                }

            } catch (IOException e) {
                if (socket.isClosed() ||
Thread.currentThread().isInterrupted()) {
                    break;
                }
                System.err.println("Error de red en el hilo de escucha: " +
e.getMessage());
            }
        }
        System.out.println("[Cliente " + idCliente + " - Hilo Escucha] Hilo de
escucha terminado.");
    }
```

Explicando los Conceptos (API de Java y Lógica):

- **Clasificación de Mensajes**: El `if-else if` es el cerebro del "Recepcionista". Le permite tomar una acción diferente para cada tipo de mensaje, basándose en el `comando` que definimos en nuestro protocolo.
- **Respuesta Inmediata**: La responsabilidad del hilo de escucha es reaccionar rápido. Cuando recibe coordenadas de un vecino (`COORDS`), no espera ni analiza nada más; inmediatamente construye y envía el `ACK` . Esto cumple con el requisito de que los clientes se contesten entre sí (a través del servidor).
- **`paqueteRecibido.getAddress()` y `.getPort()`** : Este es un punto muy importante. Para responder a un mensaje, no necesitamos tener guardada la dirección del servidor de antemano. El propio `DatagramPacket` que recibimos ya contiene la dirección de retorno del remitente. Al usar `paqueteRecibido.getAddress()` y `paqueteRecibido.getPort()` como destino de nuestro paquete de respuesta, nos aseguramos de que la respuesta vaya exactamente a quien nos envió el mensaje (en este caso, el servidor).
- **`acksRecibidos.add(mensaje)`** : Este es el momento en que los dos hilos se comunican. El hilo de escucha (`escucharAlServidor`) añade un `ACK` a la cola. El hilo principal (`run`), que está en otra parte del programa, podrá ver que la cola ya no está vacía y procesar este nuevo elemento.

Resumen de lo que hemos logrado:

Nuestro "Recepcionista" ha sido entrenado. Ya no solo recoge los mensajes, sino que los lee, los clasifica y actúa. Sabe responder a los mensajes de los vecinos de forma autónoma y sabe cómo dejar los ACKs importantes en la "bandeja de entrada" (`acksRecibidos`) para que el "Jefe de Tareas" (el hilo principal) los vea. La comunicación asíncrona ahora es completamente funcional.

Paso 14: El Bucle Principal de Simulación del Cliente

Objetivo de este paso:

El documento del proyecto establece que cada cliente debe realizar **S ciclos de ejecución**. Nuestro objetivo ahora es construir el bucle `for` que se repetirá `S` veces. Dentro de cada una de estas repeticiones (ciclos), el cliente realizará la secuencia completa:

1. Enviar sus coordenadas.
2. Poner en marcha un cronómetro.
3. Esperar los reconocimientos (`ACKs`). (Esto lo haremos en el siguiente paso).
4. Detener el cronómetro y guardar el tiempo.

En este paso nos centraremos en el esqueleto del bucle, el envío de coordenadas y el inicio del cronómetro.

El Código:

Añadiremos el bucle `for` en el método `run()` , justo después de iniciar el `hiloEscucha` . También declararemos una `List` para ir guardando los tiempos de cada ciclo.

```
// Importamos ArrayList y ThreadLocalRandom
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

// Dentro de la clase PersonaUDP

    @Override
    public void run() {
        try {
            // ... (creación de socket, registro, espera de señal...)

            Thread hiloEscucha = new Thread(this::escucharAlServidor);
            hiloEscucha.start();

            // --- AÑADIMOS EL BUCLE PRINCIPAL DE LA SIMULACIÓN ---

            // Creamos una lista para guardar el tiempo que tarda cada ciclo.
            List<Long> tiemposDeRespuesta = new ArrayList<>();

            // Este bucle se repetirá S_iteraciones veces.
            for (int i = 0; i < S_iteraciones; i++) {

                System.out.println(
                    "[Cliente " + idCliente + " - Hilo Principal] Iniciando
ciclo " + (i + 1) + "/" + S_iteraciones
                );
```

```
                // 1. LIMPIAR LA BANDEJA DE ENTRADA
                // Es crucial vaciar la cola de ACKs antes de empezar un nuevo
ciclo.
                acksRecibidos.clear();

                // 2. PREPARAR Y ENVIAR LAS COORDENADAS
                // Usamos ThreadLocalRandom para generar un número aleatorio
como coordenada.
                String coordenadas = "COORDS;" + this.idCliente + ";" +
ThreadLocalRandom.current().nextInt(0, 101);
                byte[] datosEnvio = coordenadas.getBytes();

                // Obtenemos la dirección del servidor (podríamos haberla
guardado antes).
                InetAddress direccionServidor =
InetAddress.getByName(ipServidor);
                DatagramPacket paqueteEnvio = new DatagramPacket(datosEnvio,
datosEnvio.length, direccionServidor, puertoServidor);

                // 3. INICIAR EL CRONÓMETRO
                long tiempoInicio = System.nanoTime();

                // 4. ENVIAR EL PAQUETE
                socket.send(paqueteEnvio);

                // 5. ESPERAR LOS ACKS (lo implementaremos en el siguiente
paso)
                // esperarAcks();

                // 6. DETENER CRONÓMETRO Y GUARDAR TIEMPO (lo haremos en el
siguiente paso)
                // long tiempoFin = System.nanoTime();
                // long duracion = (tiempoFin - tiempoInicio) / 1_000_000; //
en milisegundos
                // tiemposDeRespuesta.add(duracion);
            }

            // ... (el resto del código, como interrumpir el hilo, vendrá
después)

        } catch (IOException e) {
            // ...
        } finally {
            // ...
        }
    }

// ... (método escucharAlServidor sin cambios) ...
```

Explicando los Conceptos (API de Java y Lógica):

- **`for (int i = 0; i < S_iteraciones; i++)`** : Volvemos a usar el `for` clásico porque sabemos exactamente cuántas veces queremos repetir la tarea: `S` veces, que es el número

de iteraciones que nos dio el usuario.

- **`acksRecibidos.clear()`** : Este paso es **fundamental**. `clear()` es un método que vacía la cola. ¿Por qué es tan importante? Imagina que en el ciclo 1 un ACK llega tarde. Si no limpiáramos la cola, ese ACK del ciclo 1 seguiría ahí al empezar el ciclo 2. Nuestro programa pensaría erróneamente que ya ha recibido un ACK del ciclo 2 y podría terminar el ciclo antes de tiempo. Limpiar la cola nos asegura que cada ciclo empieza desde cero.
- **`ThreadLocalRandom`** : En entornos con muchos hilos, esta es la clase recomendada para generar números aleatorios. `ThreadLocalRandom.current().nextInt(0, 101)` nos da un número entero aleatorio entre 0 (incluido) y 101 (excluido), es decir, de 0 a 100.
- **`System.nanoTime()`** : Para medir el tiempo de respuesta de una operación de red, necesitamos mucha precisión. `System.currentTimeMillis()` (que da milisegundos) a veces no es suficiente. `System.nanoTime()` nos da el tiempo del reloj de alta resolución del sistema en **nanosegundos**. Es el método ideal para medir duraciones cortas de forma precisa. No sirve para saber la hora del día, solo para medir "cuánto tiempo ha pasado entre el punto A y el punto B".

Resumen de lo que hemos logrado:

Hemos construido el motor de nuestro cliente. El hilo principal ahora tiene un bucle `for` que ejecuta el ciclo de la simulación `S` veces. En cada ciclo, sabe cómo limpiar su estado, preparar y enviar su mensaje de coordenadas, y cómo iniciar el cronómetro. Aún no sabe cómo esperar las respuestas, pero la parte "activa" de su trabajo ya está implementada.

Paso 15: La Espera - Recibiendo los Reconocimientos (ACKs)

Objetivo de este paso:

El documento del proyecto dice que un ciclo de ejecución termina "en el instante en el que ha recibido el último de todos los reconocimientos". Además, la Figura 1 muestra una espera con un timeout de 20 segundos.

Nuestro objetivo es implementar esta lógica de espera en el hilo principal (`run`). Este hilo debe:

1. Pausarse después de enviar las coordenadas.
2. Vigilar la "bandeja de entrada" (`acksRecibidos`) hasta que contenga `V-1` mensajes (un ACK por cada vecino).
3. Implementar un timeout de 20 segundos para no quedarse esperando indefinidamente.
4. Una vez que se cumplen las condiciones, detener el cronómetro y guardar la duración del ciclo.

El Código:

Crearemos una nueva función auxiliar, `esperarAcks()` , para mantener el código limpio. Luego la llamaremos desde nuestro bucle `for` en `run()` y completaremos la lógica del cronómetro.

```
// Dentro de la clase PersonaUDP

    @Override
    public void run() {
        try {
            // ... (código hasta el inicio del bucle for) ...
```

```
            List<Long> tiemposDeRespuesta = new ArrayList<>();
            for (int i = 0; i < S_iteraciones; i++) {
                // ... (código para limpiar la cola y enviar el paquete de
COORDS) ...

                long tiempoInicio = System.nanoTime();
                socket.send(paqueteEnvio);

                // --- AÑADIMOS LA LÓGICA DE ESPERA Y MEDICIÓN ---

                // 5. ESPERAR LOS ACKS
                // Llamamos a nuestra nueva función que contiene la lógica de
espera.
                esperarAcks();

                // 6. DETENER EL CRONÓMETRO Y GUARDAR EL TIEMPO
                long tiempoFin = System.nanoTime();
                long duracion = (tiempoFin - tiempoInicio) / 1_000_000; //
Convertimos nanosegundos a milisegundos
                tiemposDeRespuesta.add(duracion);

                System.out.println(
                    "[Cliente " + idCliente + " - Hilo Principal] Ciclo " + (i
+ 1) + " completado en " + duracion + "ms."
                );
            }

            // ... (aquí irá la lógica final después del bucle) ...

        } catch (IOException | InterruptedException e) { // Añadimos
InterruptedException
            System.err.println("[Cliente " + idCliente + "] Error: " +
e.getMessage());
        } finally {
            // ...
        }
    }

    // --- AÑADIMOS LA NUEVA FUNCIÓN DE ESPERA ---

    /**
     * Pone al hilo principal en espera hasta que se hayan recibido todos los
     * reconocimientos (V-1) o hasta que se supere un timeout de 20 segundos.
     * @throws InterruptedException si el hilo es interrumpido mientras
duerme.
     */
    private void esperarAcks() throws InterruptedException {
        long inicioEspera = System.currentTimeMillis(); // Tiempo de inicio
para el timeout

        // El bucle se repite mientras el número de ACKs en la cola sea menor
        // que el número de vecinos que deben responder (V - 1).
        while (acksRecibidos.size() < (V_vecinos - 1)) {
```

```
            // Comprobación del Timeout de 20 segundos
            if (System.currentTimeMillis() - inicioEspera > 20000) {
                System.err.println("[Cliente " + idCliente + "] TIMEOUT: No se
recibieron todos los ACKs a tiempo.");
                break; // 'break' rompe el bucle while y continúa.
            }

            // Pausa para no consumir 100% de CPU.
            // Esto evita que el bucle se ejecute millones de veces por
segundo.
            Thread.sleep(50); // Pausa de 50 milisegundos.
        }
    }

// ... (método escucharAlServidor sin cambios) ...
```

Explicando los Conceptos (API de Java y Lógica):

- **`acksRecibidos.size() < (V_vecinos - 1)`** : Esta es la condición principal de la espera. El hilo principal "pregunta" constantemente a la cola `acksRecibidos` cuántos elementos tiene (`.size()`). La simulación de un ciclo no termina hasta que el tamaño de la cola sea igual al número de vecinos (`V_vecinos - 1` , porque el grupo total es de `V` , pero uno mismo no se envía ACKs).
- **Lógica de Timeout**: El patrón `long inicioEspera = System.currentTimeMillis();` seguido de la comprobación `if (System.currentTimeMillis() - inicioEspera > 20000)` es la forma estándar de implementar un timeout en software. `System.currentTimeMillis()` da la hora actual en milisegundos, por lo que la resta nos da el tiempo transcurrido.
- **`Thread.sleep(50)` y "Busy-Waiting"**: Si quitáramos esta línea, el bucle `while` se ejecutaría a la máxima velocidad posible, comprobando la condición millones de veces por segundo. Esto consumiría el 100% de un núcleo de la CPU sin hacer ningún trabajo útil. A esto se le llama **"espera activa" o "busy-waiting"**. Al poner `Thread.sleep(50)` , le decimos al hilo principal: "duérmete durante 50 milisegundos y deja que la CPU haga otras cosas". Esto hace que nuestro programa sea muchísimo más eficiente.
- **`InterruptedException`** : El método `Thread.sleep()` puede lanzar esta excepción si otro hilo "despierta" a nuestro hilo mientras está durmiendo. Java nos obliga a capturarla. La hemos añadido al `catch` del método `run` .

Resumen de lo que hemos logrado:

Hemos completado el ciclo de comunicación del cliente. El hilo principal ahora sabe cómo esperar de manera eficiente y segura a que el hilo de escucha recolecte todas las respuestas necesarias. También hemos implementado la regla del timeout de 20 segundos, haciendo nuestro cliente más robusto. El "Jefe de Tareas" ya puede delegar la escucha, enviar su trabajo y esperar a que le confirmen que ha sido recibido antes de registrar el tiempo final.

Paso 16: Finalizando la Simulación y Enviando Resultados

Objetivo de este paso:

Una vez que el bucle `for` de `S` iteraciones ha terminado, el cliente ha completado su trabajo de simulación. Según el documento del proyecto, su última tarea es:

1. Calcular el **tiempo medio** de todos los ciclos que ha realizado.
2. Enviar este resultado al servidor con el mensaje `DONE`.
3. Cerrar sus recursos de forma ordenada y terminar su ejecución.

El Código:

Añadiremos el código final dentro del método `run()`, justo después de que termine el bucle `for`. También completaremos la parte de `hiloEscucha.interrupt()` para asegurarnos de que el hilo de escucha se apaga correctamente.

```
// Dentro de la clase PersonaUDP

    @Override
    public void run() {
        // Inicializamos la lista aquí para que sea accesible después del
bucle
        List<Long> tiemposDeRespuesta = new ArrayList<>();
        Thread hiloEscucha = null; // Lo declaramos fuera para poder acceder a
él después del try

        try {
            // ... (socket, registro, espera de señal...)

            hiloEscucha = new Thread(this::escucharAlServidor);
            hiloEscucha.start();

            // Bucle for de S iteraciones (del Paso 14 y 15, sin cambios)
            for (int i = 0; i < S_iteraciones; i++) {
                // ... (código del ciclo: clear, send, esperarAcks, guardar
tiempo...)
            }

            // --- AÑADIMOS LA LÓGICA DE FINALIZACIÓN ---

            // FASE 3: CÁLCULO Y FINALIZACIÓN

            // 1. CALCULAR EL TIEMPO PROMEDIO
            // Usamos Streams de Java para una forma muy concisa de calcular
la media.
            double tiempoPromedio = tiemposDeRespuesta.stream() // Convierte
la lista en un flujo de datos
                                                    .mapToLong(val -> val) //
Convierte cada elemento a un tipo long
                                                    .average() // Calcula la
media
                                                    .orElse(0.0); // Si la
lista estuviera vacía, devuelve 0.0

            System.out.printf(
                "[Cliente %d - Hilo Principal] Simulación terminada. Tiempo
medio: %.4f ms\n",
```

```
                idCliente, tiempoPromedio
            );

            // 2. ENVIAR EL RESULTADO AL SERVIDOR
            String mensajeFinalizado = "DONE;" + tiempoPromedio;
            byte[] datosEnvio = mensajeFinalizado.getBytes();

            InetAddress direccionServidor = InetAddress.getByName(ipServidor);
            DatagramPacket paqueteFinalizado = new DatagramPacket(datosEnvio,
datosEnvio.length, direccionServidor, puertoServidor);
            socket.send(paqueteFinalizado);

        } catch (IOException | InterruptedException e) {
            System.err.println("[Cliente " + idCliente + "] Error: " +
e.getMessage());
        } finally {
            // --- LÓGICA DE CIERRE ORDENADO ---

            // 3. DETENER EL HILO DE ESCUCHA DE FORMA SEGURA
            if (hiloEscucha != null) {
                hiloEscucha.interrupt(); // Le pedimos amablemente que se
detenga
                try {
                    hiloEscucha.join(1000); // Esperamos hasta 1 segundo a que
termine
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // 4. CERRAR EL SOCKET
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }
        }
    }

// ... (método esperarAcks y escucharAlServidor sin cambios) ...
```

Explicando los Conceptos (API de Java y Lógica):

- **Streams de Java (`.stream()` ...)**: A partir de Java 8, los Streams ofrecen una forma muy potente y declarativa de procesar colecciones de datos. En lugar de escribir un bucle `for` para sumar todos los tiempos y luego dividir, simplemente describimos lo que queremos:
  1. `.stream()` : Convierte nuestra `List<Long>` en un flujo.
  2. `.mapToLong(val -> val)` : Lo convierte en un flujo de números primitivos `long` (más eficiente).
  3. `.average()` : Calcula la media y devuelve un objeto especial llamado `OptionalDouble` (porque la media podría no existir si la lista está vacía).
  4. `.orElse(0.0)` : Extrae el valor del `OptionalDouble` , o devuelve `0.0` si estaba vacío.
- **`hiloEscucha.interrupt()`** : Este método no "mata" al hilo. Simplemente le pone una "bandera" interna que dice "han solicitado que te interrumpas". Nuestro bucle `while (!Thread.currentThread().isInterrupted())` en `escucharAlServidor` está diseñado

para comprobar esta bandera en cada iteración y terminarse de forma ordenada cuando la ve.

- **`hiloEscucha.join(1000)`** : Este método le dice al hilo principal: "Pausa tu propia ejecución y espera aquí hasta que el `hiloEscucha` haya terminado por completo". Le ponemos un timeout de 1000 milisegundos (1 segundo) por seguridad. Esto es **muy importante** para evitar una "condición de carrera": no queremos que el bloque `finally` se ejecute y cierre el `socket` mientras el `hiloEscucha` podría estar usándolo en su última operación. Con `join` , nos aseguramos de que el hilo de escucha termina primero, y solo después cerramos el socket.

Paso 17: El Lanzador - Poniendo a Trabajar a Todos los Clientes

Objetivo de este paso:

El objetivo es crear la clase `ClienteUDP` con su método `main` . La única responsabilidad de esta clase es:

1. Preguntar al usuario los parámetros de la simulación (la IP/puerto del servidor y N, V, S).
2. Crear `N` instancias (objetos) de nuestra clase `PersonaUDP` .
3. Iniciar cada una de esas instancias como un hilo independiente.
4. Esperar a que todos los hilos terminen su trabajo.

Esta clase no envía ni recibe paquetes. Es simplemente el punto de entrada de la aplicación cliente.

El Código:

Crea el último archivo que nos falta: `ClienteUDP.java` .

```
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Clase principal encargada de lanzar la simulación del lado del cliente.
 * Su única responsabilidad es pedir al usuario los parámetros de la
simulación
 * y crear e iniciar N hilos, donde cada hilo es una instancia de PersonaUDP
 * que actuará como un cliente independiente.
 */
public class ClienteUDP {

    public static void main(String[] args) {
        Scanner escaner = new Scanner(System.in);
        System.out.println("--- CONFIGURACIÓN DEL CLIENTE ---");
        System.out.print("Introduce la IP del servidor (ej. 127.0.0.1): ");
        String ipServidor = escaner.nextLine();
        System.out.print("Introduce el puerto del servidor (ej. 10578): ");
        int puertoServidor = escaner.nextInt();

        // Estos parámetros deben coincidir con los que se usaron para iniciar
el servidor.
        System.out.print("Introduce el número total de clientes (N): ");
        int numeroClientes = escaner.nextInt();
```

```
        System.out.print("Introduce el número de vecinos por grupo (V): ");
        int numeroVecinos = escaner.nextInt();
        System.out.print("Introduce el número de iteraciones por cliente (S):
");
        int numeroIteraciones = escaner.nextInt();
        escaner.close();

        System.out.println("\n=== Iniciando " + numeroClientes + " Clientes
UDP ===");
        System.out.println("Conectando a: " + ipServidor + ":" +
puertoServidor);

        // 1. CREAR UNA LISTA PARA GUARDAR LOS HILOS
        ArrayList<Thread> listaHilosClientes = new ArrayList<>();

        // 2. CREAR LAS N INSTANCIAS DE PersonaUDP
        for (int i = 0; i < numeroClientes; i++) {
            // Creamos un nuevo objeto trabajador y lo añadimos a nuestra
lista.
            listaHilosClientes.add(new PersonaUDP(ipServidor, puertoServidor,
numeroVecinos, numeroIteraciones));
        }

        // 3. INICIAR TODOS LOS HILOS
        System.out.println("Lanzando todos los hilos cliente...");
        for (Thread hiloCliente : listaHilosClientes) {
            hiloCliente.start(); // ¡Esto inicia la ejecución del método run()
en un nuevo hilo!
            try {
                // Pequeña pausa para no iniciar los N clientes en el mismo
microsegundo.
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 4. ESPERAR A QUE TODOS LOS HILOS TERMINEN
        System.out.println("Todos los clientes han sido lanzados. El programa
principal esperará a que terminen.");
        for (Thread hiloCliente : listaHilosClientes) {
            try {
                // .join() hace que el hilo actual (el 'main') se pause
                // hasta que el 'hiloCliente' haya terminado su ejecución.
                hiloCliente.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("\n=== Todos los clientes han terminado su
ejecución. Programa finalizado. ===");
    }
}
```

Explicando los Conceptos (API de Java y Lógica):

- **`ArrayList<Thread>`** : Usamos una lista para "fichar" a cada trabajador (`PersonaUDP`) que creamos. Esto es necesario porque si no los guardamos en una lista, después de iniciarlos perderíamos la referencia a ellos y no podríamos esperar a que terminen.
- **`hiloCliente.start()` vs `hiloCliente.run()`** : ¡Esta es la diferencia más importante en la programación con hilos!
  - **`.start()`** : Este es el método correcto. Le dice a la Máquina Virtual de Java (JVM): "Crea un nuevo hilo de ejecución completamente nuevo, y dentro de ese nuevo hilo, ejecuta el contenido del método `run()` ". El hilo `main` (el que está ejecutando el `for`) no espera, sino que continúa inmediatamente para iniciar el siguiente hilo. Así es como conseguimos que los `N` clientes se ejecuten en paralelo.
  - **`.run()`** : Si por error llamáramos a `.run()` directamente, **no se crearía ningún hilo nuevo**. El código del método `run()` se ejecutaría dentro del hilo `main` . El resultado sería que el cliente 1 se ejecutaría por completo, y solo cuando terminara, empezaría el cliente 2, y así sucesivamente. Sería una ejecución **secuencial**, no paralela.
- **`hiloCliente.join()`** : Este es el método que nos permite esperar. Cuando el hilo `main` llega a esta línea, se detiene y espera a que el `hiloCliente` sobre el que ha llamado a `.join()` complete su método `run()` y muera. Al poner esto en un bucle, el hilo `main` esperará por el primer cliente de la lista, luego por el segundo, y así hasta que el último haya terminado. Esto garantiza que el mensaje final del programa solo se muestre cuando todo el trabajo ha concluido.

Resumen de lo que hemos logrado:

¡Lo hemos conseguido! Hemos creado la pieza final que une todo. El `ClienteUDP` actúa como el director de orquesta que, siguiendo las instrucciones del usuario, contrata (`new PersonaUDP`), organiza (`ArrayList`) y da la orden de empezar (`.start()`) a cada uno de los `N` músicos. Luego, espera pacientemente (`.join()`) a que la sinfonía termine antes de dar por concluido el concierto. Ahora tienes un proyecto completo y funcional, construido desde cero.