
INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO


# APLICACIONES PARA COMUNICACIONES EN RED

UNIDAD III: Arquitectura Cliente-Servidor

# PRÁCTICA 4 – SERVIDOR HTTPS C/ POOL DIRECCIONES

Grupo 6CM2 - Ciclo 2026A

Alumnas:

Cruz Rodríguez Arely Amairani

Ortiz Villaseñor Alexandra

Fecha de entrega: Martes 06 de enero de 2026

# INTRODUCCIÓN

El desarrollo de aplicaciones en red modernas se sustenta primordialmente en la Arquitectura Cliente-Servidor, un modelo de diseño de software donde las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. En este contexto, el protocolo HTTP (Hypertext Transfer Protocol) surge como el estándar fundamental para la interoperabilidad en la World Wide Web. La presente práctica tiene como objetivo primordial la programación de un servidor HTTP funcional, capaz de gestionar peticiones síncronas mediante sockets de flujo y procesamiento multihilo, ajustándose a las especificaciones técnicas del RFC 2616.

Para garantizar que aplicaciones en red independientes de la arquitectura o el sistema operativo puedan comunicarse, el IETF (Internet Engineering Task Force) establece normativas rigurosas que definen el comportamiento de estos protocolos. En esta implementación, se ha diseñado un sistema distribuido que emplea sockets de flujo bloqueantes, los cuales permiten establecer conexiones fiables extremo a extremo sobre la capa de transporte. La relevancia de este proyecto radica en la comprensión profunda del intercambio de mensajes (petición y respuesta) y en la manipulación de diversos tipos MIME, lo cual es esencial para el manejo de recursos multimedia en la red.

Un aspecto crítico abordado en este proyecto es la escalabilidad y la gestión de recursos mediante un pool de conexiones. En una arquitectura cliente-servidor real, la saturación del servidor es un riesgo latente; por ello, se ha implementado una lógica de balanceo de carga rudimentaria basada en la redirección (código de estado 307). Esto permite que, al superar el 50% de la capacidad operativa del pool de hilos, el servidor principal delegue el tráfico a un servidor de respaldo. Con este enfoque, el estudiante no solo aplica conocimientos de programación concurrente en Java, sino que también analiza críticamente los servicios definidos en la capa de transporte para construir soluciones de red robustas y eficientes que cumplen con los estándares internacionales de comunicación.

# DESARROLLO

## Clase Backupserver.java

La clase BackupServer.java funciona como una unidad de soporte crítica dentro de la arquitectura de red distribuida del proyecto, actuando bajo el principio de **redundancia y alta disponibilidad**. Su propósito principal es operar en un puerto alterno (específicamente el puerto 8081) para recibir y gestionar de forma autónoma el tráfico que el servidor principal no puede procesar debido a la saturación del pool de conexiones.

Al ser activada mediante una redirección HTTP 307 desde el servidor principal, esta clase establece su propio ServerSocket para escuchar peticiones entrantes y responde al cliente con un mensaje informativo indicando que ha sido redirigido temporalmente. Esta implementación garantiza la continuidad del servicio y evita la pérdida de peticiones ante picos de demanda, demostrando un manejo eficiente de la carga de trabajo y el cumplimiento de los estándares de interoperabilidad del protocolo HTTP.

```
package practica4.servidorhttp;

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

public class BackupServer {
    public static void main(String[] args) {
        int port = 8081;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println(">>> SERVIDOR DE RESPALDO ACTIVO EN PUERTO " + port + " <<<");

            while (true) {
                try (Socket client = serverSocket.accept();
                     PrintWriter out = new PrintWriter(client.getOutputStream())) {

                    System.out.println("Atendiendo petición redirigida...");

                    out.println("HTTP/1.1 200 OK");
                    out.println("Content-Type: text/html; charset=UTF-8");
                    out.println();
                    out.println("<!DOCTYPE html><html><head><title>Respaldo</title></head>");
                    out.println("<body style='font-family:sans-serif; text-align:center;
padding-top:50px;'>");
                    out.println("<h1 style='color:#741E41;'>Servidor de Respaldo</h1>");
                    out.println("<p>Has sido redirigido porque el servidor principal alcanzó el
50% de su capacidad.</p>");
                    out.println("<a href='http://localhost:8080'>Intentar volver al
principal</a>");
                    out.println("</body></html>");
                    out.flush();
                }
            }
        } catch (IOException e) {
            System.err.println("Error en Backup Server: " + e.getMessage());
        }
    }
}
```

# Clase Server.java

Esta clase representa el **núcleo central** y el cerebro de la arquitectura. Su función principal es actuar como un **orquestador de tráfico**, gestionando la entrada de usuarios y asegurando que el sistema nunca colapse por exceso de demanda.
Las responsabilidades principales de la clase son:

**Gestión de Recursos y Concurrencia:**

- **Pool de Hilos Dinámico:** Utiliza un **ExecutorService** para crear un "ejército" de hilos de trabajo. El tamaño de este pool es definido por el usuario al inicio, permitiendo adaptar la potencia del servidor según el hardware disponible.
- **Contador Atómico:** Emplea un **AtomicInteger** llamado **activeConnections**. Este es un monitor de alta precisión que rastrea en tiempo real cuántas personas están conectadas, evitando errores de cálculo cuando múltiples usuarios entran al mismo tiempo.

**Balanceo de Carga e Inteligencia de Red:**

- **Umbral de Saturación:** Implementa una regla de negocio crítica: si el servidor alcanza el **50% de su capacidad**, activa automáticamente un protocolo de protección.
- **Redirección Inteligente (HTTP 307):** En lugar de rechazar la conexión y mostrar un error, el servidor utiliza el método **sendTemporaryRedirect**. Este envía una instrucción al navegador del cliente para que viaje automáticamente hacia el **Servidor de Respaldo** (localhost:8081).

**Ciclo de Vida del Servicio:**

- **Socket de Escucha:** Abre un **ServerSocket** en el **puerto 8080**, manteniéndose en un estado de "escucha activa" mediante un bucle infinito (while(true)).
- **Delegación de Tareas:** Cuando hay espacio disponible en el pool, el servidor acepta el Socket del cliente y le entrega la responsabilidad a **ClientHandler**. Esto permite que el servidor principal quede libre de inmediato para recibir a la siguiente persona mientras los hilos procesan las peticiones en segundo plano.

```
package practica4.servidorhttp;

import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.Scanner;

public class Server {
    // Puerto principal
    private static final int PORT = 8080;

    // Tamaño definido por el usuario
    public static int MAX_POOL_SIZE;

    // Contador atómico para rastrear conexiones en tiempo real
    public static AtomicInteger activeConnections = new AtomicInteger(0);

    private static ExecutorService pool;

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

```
        Scanner sc = new Scanner(System.in);

        // 1. El usuario define el tamaño del pool
        System.out.print("Ingrese el tamaño máximo del pool de conexiones: ");
        MAX_POOL_SIZE = sc.nextInt();

        // Inicializamos el pool con el tamaño elegido
        pool = Executors.newFixedThreadPool(MAX_POOL_SIZE);

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("--- SERVIDOR PRINCIPAL INICIADO ---");
            System.out.println("Puerto: " + PORT);
            System.out.println("Capacidad máxima: " + MAX_POOL_SIZE);
            System.out.println("Umbral de redirección (50%): " + (MAX_POOL_SIZE / 2));
            System.out.println("Esperando conexiones...\n");

            while (true) {
                Socket clientSocket = serverSocket.accept();

                // Lógica de Redirección:
                // Si las conexiones actuales superan la mitad del tamaño definido
                if (activeConnections.get() >= (MAX_POOL_SIZE / 2)) {
                    System.out.println("[ALERTA] Pool al 50% o más (" + activeConnections.get()
+ "). Redireccionando...");
                    sendTemporaryRedirect(clientSocket, "http://localhost:8081");
                } else {
                    // Si hay espacio, incrementamos el contador y procesamos
                    activeConnections.incrementAndGet();
                    System.out.println("[INFO] Cliente aceptado. Conexiones activas: " +
activeConnections.get());
                    pool.execute(new ClientHandler(clientSocket));
                }
            }
        } catch (IOException e) {
            System.err.println("Error crítico en el servidor: " + e.getMessage());
        }
    }

    /**
     * Envía una respuesta HTTP 307 para redirigir al navegador al segundo servidor
     */
    private static void sendTemporaryRedirect(Socket socket, String newUrl) {
        // Al usar Try-with-resources, 'out' y 'socket.getOutputStream()'
        // se cerrarán automáticamente, lo que cierra el socket.
        try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
            out.println("HTTP/1.1 307 Temporary Redirect");
            out.println("Location: " + newUrl);
            out.println("Content-Length: 0");
            out.println("Connection: close");
            out.println();
            // out.flush(); // No es estrictamente necesario si usas 'true' en el constructor
        } catch (IOException e) {
            System.err.println("Error al intentar redireccionar: " + e.getMessage());
        }
    }
}
```

# Clase ClientHandler.java

Esta clase representa la **inteligencia operativa** del servidor; es el componente encargado de interpretar los deseos del cliente y ejecutarlos sobre el sistema de archivos. Mientras que la clase Server actúa como recepcionista, ClientHandler es el especialista que procesa cada solicitud de forma individual y detallada.

**Procesamiento y Análisis de Protocolo**

- **Análisis de Sintaxis (Parsing):** Utiliza **StringTokenizer** para fragmentar la línea de solicitud HTTP, abstrayendo el Método (GET, POST, etc.) y el Recurso solicitado (archivo) para determinar la ruta de ejecución.
- **Gestión Multihilo:** Al implementar la **interfaz Runnable**, permite que cada cliente sea atendido en su propio hilo de ejecución, evitando que una petición lenta bloquee el resto del sistema.

**Ejecución de Métodos HTTP (Verbos)**

1. **GET (Lectura):** Localiza recursos en la carpeta raíz (www), determina su tipo MIME (HTML, imágenes, fuentes TTF, etc.) y transfiere los datos binarios mediante un **BufferedOutputStream**.
2. **POST/PUT (Escritura):** Implementa lógica de recepción de datos. POST simula la creación de registros, mientras que PUT permite la carga dinámica de archivos en el servidor conservando el nombre original proporcionado en la URL.
3. **DELETE (Borrado):** Gestiona la eliminación segura de archivos específicos en el servidor, verificando primero la existencia del recurso y validando que no se trate de un directorio.

**Robustez y Control de Errores**

- **Manejo de Errores Estándar:** Mediante el método **sendError**, el manejador genera dinámicamente páginas HTML de respuesta para códigos de estado críticos como 404 Not Found, 400 Bad Request o 501 Not Implemented.
- **Finalización Segura de Recursos:** En su bloque finally, garantiza que el contador de conexiones activas en la clase Server disminuya siempre y que todos los flujos de datos (streams) y el socket se cierren correctamente, previniendo fugas de memoria o bloqueos en el pool.

```
package practica4.servidorhttp;

import java.io.*;
import java.net.Socket;
import java.util.StringTokenizer;

public class ClientHandler implements Runnable {
    private final Socket connect;
    private static final String WEB_ROOT = "www"; // Carpeta donde estarán tus archivos

    public ClientHandler(Socket c) {
        this.connect = c;
    }
    @Override
    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
```

```
        BufferedOutputStream dataOut = null;
        try {
            in = new BufferedReader(new InputStreamReader(connect.getInputStream()));
            out = new PrintWriter(connect.getOutputStream());
            dataOut = new BufferedOutputStream(connect.getOutputStream());

            String input = in.readLine();
            if (input == null || input.isEmpty()) return;

            StringTokenizer parse = new StringTokenizer(input);
            if (!parse.hasMoreTokens()) return;

            String method = parse.nextToken().toUpperCase();
            String fileRequested = parse.nextToken();

            // SINTAXIS ALTERNATIVA CON IF-ELSE (Elimina errores de Switch)
            switch (method) {
                case "GET" -> processGet(fileRequested, out, dataOut);
                case "POST" -> processPost(in, out, dataOut);
                case "PUT" -> processPut(fileRequested, in, out, dataOut);
                case "DELETE" -> processDelete(fileRequested, out, dataOut);
                default -> sendError(out, dataOut, "501 Not Implemented");
            }
        } catch (IOException ioe) {
            System.err.println("Error en el servidor: " + ioe.getMessage());
        } finally {
            // 1. Lo primero es liberar el espacio en el pool para otros clientes
            try {
                if (Server.activeConnections != null) {
                    Server.activeConnections.decrementAndGet();
                }
            } catch (Exception e) {
                System.err.println("Error al decrementar contador: " + e.getMessage());
            }
            // 2. Cerramos los recursos
            try {
                if (in != null) in.close();
                if (out != null) out.close();
                if (dataOut != null) dataOut.close();
                if (connect != null) connect.close();
                System.out.println("[Pool] Conexión liberada. Activas: " +
Server.activeConnections.get());
            } catch (Exception e) {
                System.err.println("Error al cerrar streams: " + e.getMessage());
            }
        }
    }
    // Aquí irán los métodos processGet, processPost, etc.
    private String getContentType(String fileRequested) {
        if (fileRequested.endsWith(".html")) return "text/html";
        if (fileRequested.endsWith(".css")) return "text/css";
        if (fileRequested.endsWith(".json")) return "application/json";
        if (fileRequested.endsWith(".jpg") || fileRequested.endsWith(".jpeg")) return
"image/jpeg";
        if (fileRequested.endsWith(".png")) return "image/png";
        if (fileRequested.endsWith(".ttf")) return "font/ttf";
        if (fileRequested.endsWith(".svg")) return "image/svg+xml";
        return "text/plain";
    }
    private byte[] readFileData(File file, int fileLength) throws IOException {
        FileInputStream fileIn = null;
```

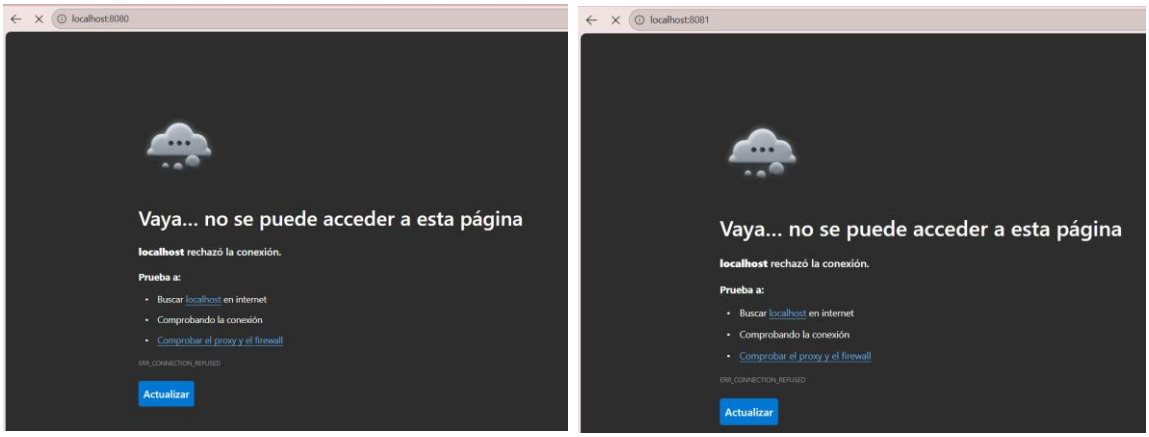
```
        byte[] fileData = new byte[fileLength];
        try {
            fileIn = new FileInputStream(file);
            fileIn.read(fileData);
        } finally {
            if (fileIn != null) fileIn.close();
        }
        return fileData;
    }
    private void sendError(PrintWriter out, BufferedOutputStream dataOut, String errorCode)
throws IOException {
    // Definimos un cuerpo HTML sencillo para el error
        String errorHtml = "<html><head><title>Error</title></head><body>" +
                           "<h1>Error: " + errorCode + "</h1>" +
                           "<p>El servidor no pudo procesar la solicitud.</p>" +
                           "</body></html>";
        byte[] errorData = errorHtml.getBytes();

        // Enviamos las cabeceras de error
        out.println("HTTP/1.1 " + errorCode);
        out.println("Server: Java HTTP Server : 1.0");
        out.println("Content-type: text/html");
        out.println("Content-length: " + errorData.length);
        out.println(); // Línea en blanco obligatoria
        out.flush();

        // Enviamos el cuerpo del error
        dataOut.write(errorData, 0, errorData.length);
        dataOut.flush();
    }
    private void processGet(String fileRequested, PrintWriter out, BufferedOutputStream dataOut)
throws IOException {
        if (fileRequested.endsWith("/")) fileRequested += "index.html";
        File file = new File(WEB_ROOT, fileRequested);
        int fileLength = (int) file.length();
        String content = getContentType(fileRequested);
        if (file.exists()) {
            byte[] fileData = readFileData(file, fileLength);

            // Cabeceras HTTP
            out.println("HTTP/1.1 200 OK");
            out.println("Server: Java HTTP Server : 1.0");
            out.println("Content-type: " + content);
            out.println("Content-length: " + fileLength);
            out.println(); // Línea en blanco obligatoria entre headers y body
            out.flush();

            dataOut.write(fileData, 0, fileLength);
            dataOut.flush();
        } else {
            sendError(out, dataOut, "404 Not Found");
        }
    }
    private void processPost(BufferedReader in, PrintWriter out, BufferedOutputStream dataOut)
throws IOException {
        String headerLine;
        int contentLength = -1;

        // 1. Validar Headers
        while ((headerLine = in.readLine()) != null && !headerLine.isEmpty()) {
            if (headerLine.startsWith("Content-Length:")) {
```

```
                contentLength = Integer.parseInt(headerLine.substring(15).trim());
            }
        }
        // Error si no hay Content-Length (Requisito para POST estándar)
        if (contentLength <= 0) {
            sendError(out, dataOut, "411 Length Required");
            return;
        }
        // 2. Leer Body
        char[] body = new char[contentLength];
        int bytesRead = in.read(body, 0, contentLength);
        if (bytesRead != contentLength) {
            sendError(out, dataOut, "400 Bad Request (Incomplete Body)");
            return;
        }
        String response = "Recurso creado exitosamente vía POST";
        byte[] responseBytes = response.getBytes();
        // 3. Responder usando dataOut para ser consistentes
        out.println("HTTP/1.1 201 Created");
        out.println("Content-Type: text/plain");
        out.println("Content-Length: " + responseBytes.length);
        out.println();
        out.flush();
        dataOut.write(responseBytes);
        dataOut.flush();
    }
    private void processPut(String fileRequested, BufferedReader in, PrintWriter out,
BufferedOutputStream dataOut) throws IOException {
    String headerLine;
    int contentLength = -1;
    // 1. Leer Headers para obtener el tamaño
    while ((headerLine = in.readLine()) != null && !headerLine.isEmpty()) {
        if (headerLine.startsWith("Content-Length:")) {
            contentLength = Integer.parseInt(headerLine.substring(15).trim());
        }
    }
    if (contentLength < 0) {
        sendError(out, dataOut, "411 Length Required");
        return;
    }
    // 2. Limpiar el nombre del archivo solicitado
    String fileName = fileRequested;
    if (fileName.startsWith("/")) {
        fileName = fileName.substring(1); // Quitar la barra inicial
    }
    // Si la ruta está vacía (solo enviaron /), asignamos un nombre por defecto
    if (fileName.isEmpty()) {
        fileName = "archivo_sin_nombre.txt";
    }
    // 3. Leer el cuerpo del mensaje
    char[] body = new char[contentLength];
    int read = 0;
    while (read < contentLength) {
        int result = in.read(body, read, contentLength - read);
        if (result == -1) break;
        read += result;
    }
    // 4. Guardar con el nombre original en WEB_ROOT
    File fileToSave = new File(WEB_ROOT, fileName);
    // Usamos FileWriter para texto o FileOutputStream si quisieras soportar binarios (imágenes)
    try (FileWriter writer = new FileWriter(fileToSave)) {
```

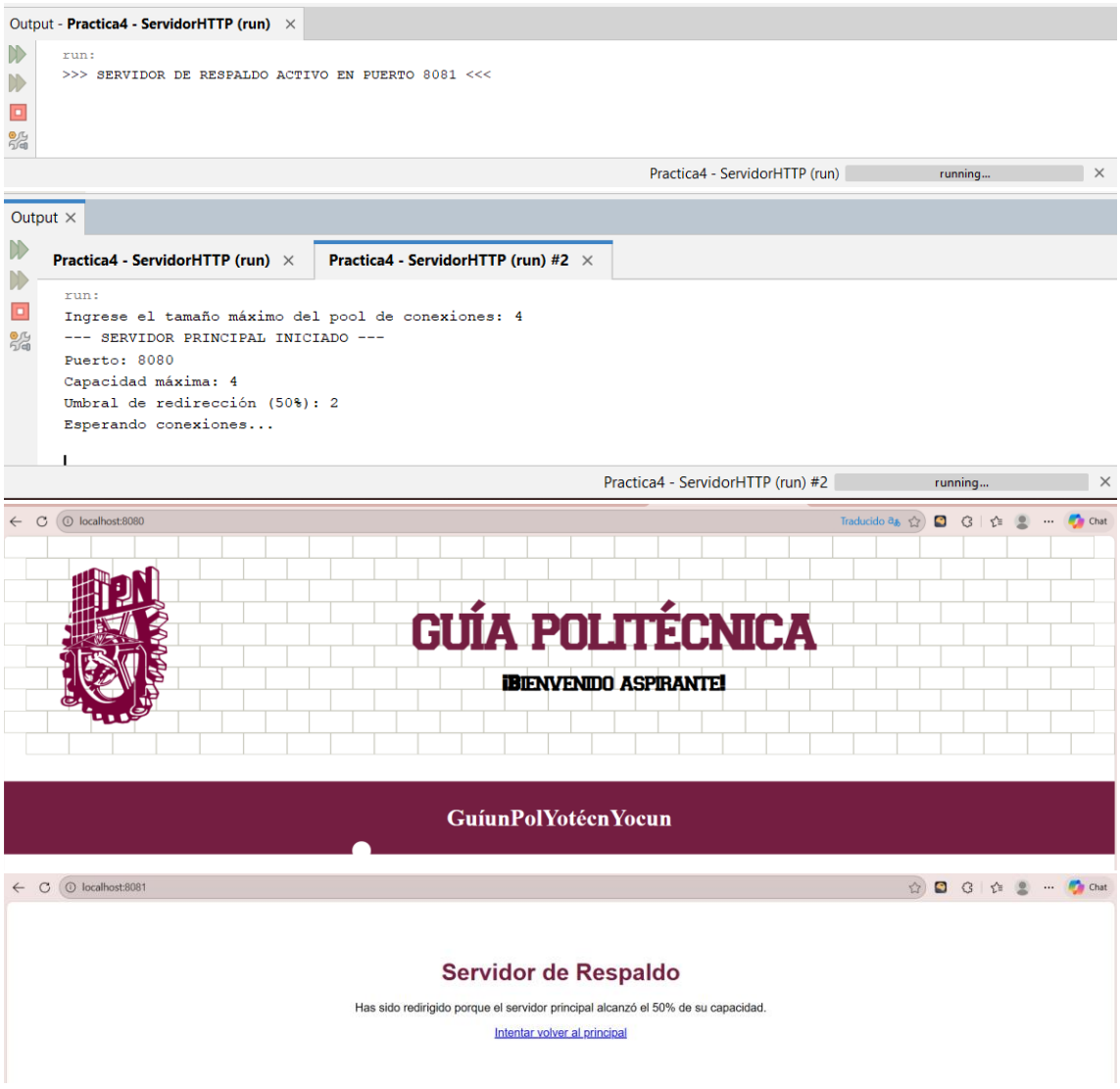
```
        writer.write(body);

        String msg = "Archivo '" + fileName + "' guardado/actualizado correctamente";
        out.println("HTTP/1.1 200 OK");
        out.println("Content-Type: text/plain");
        out.println("Content-Length: " + msg.length());
        out.println();
        out.flush();

        dataOut.write(msg.getBytes());
        dataOut.flush();

        System.out.println("[PUT] Archivo creado: " + fileToSave.getAbsolutePath());
    } catch (IOException e) {
        sendError(out, dataOut, "500 Internal Server Error (Error de escritura)");
    }
}

    private void processDelete(String fileRequested, PrintWriter out, BufferedOutputStream
dataOut) throws IOException {
    // 1. Limpiar la ruta: si empieza con '/', quitarlo para que sea relativo a WEB_ROOT
    String path = fileRequested;
    if (path.startsWith("/")) {
        path = path.substring(1);
    }
    File file = new File(WEB_ROOT, path);
    // Debug para que veas en la consola de NetBeans qué está intentando borrar
    System.out.println("[DELETE] Intentando borrar: " + file.getAbsolutePath());

    if (file.exists() && !file.isDirectory()) {
        if (file.delete()) {
            out.println("HTTP/1.1 200 OK");
            out.println("Content-Type: text/plain"); // Es bueno agregar el tipo
            out.println();
            out.println("Archivo eliminado exitosamente");
        } else {
            sendError(out, dataOut, "500 Internal Server Error");
        }
    } else {
        // Si entra aquí, es que file.exists() es falso
        sendError(out, dataOut, "404 Not Found");
    }
    out.flush();
}
}
```

# PRUEBAS

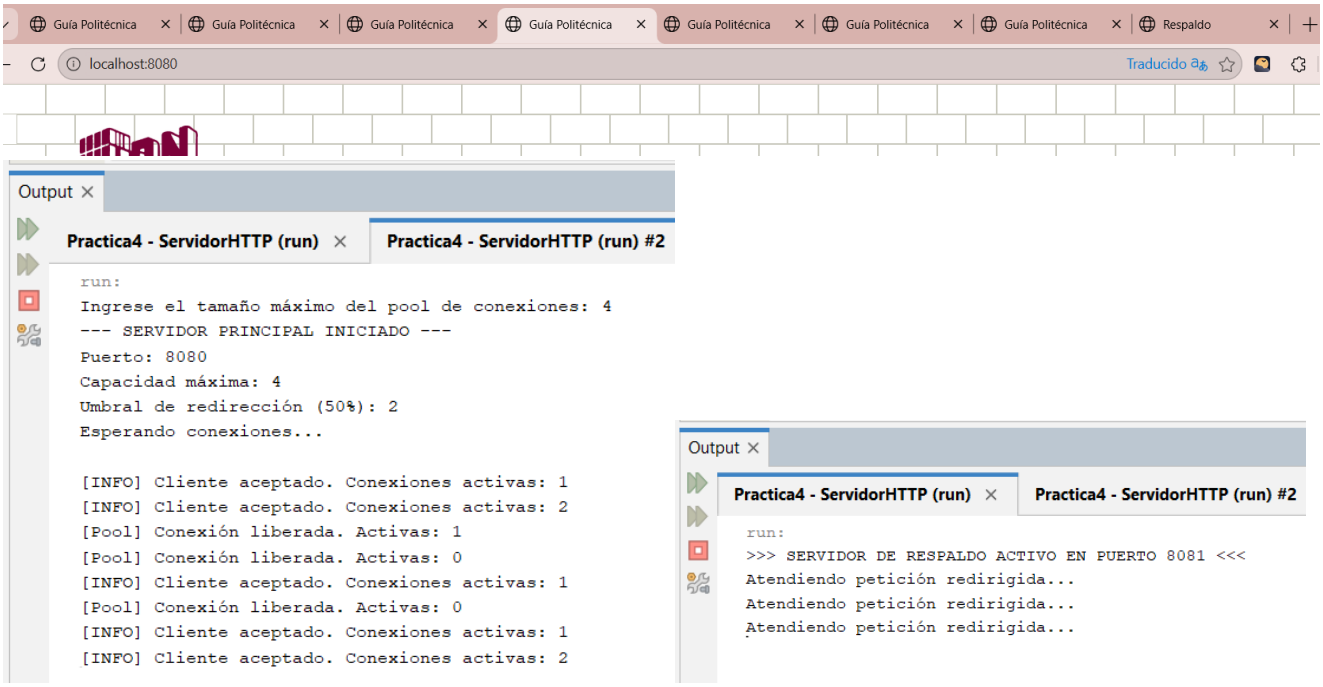
1. Antes de correr cualquiera de los servidores no se muestra ninguna página ni en el puerto 8080 ni el 8081:



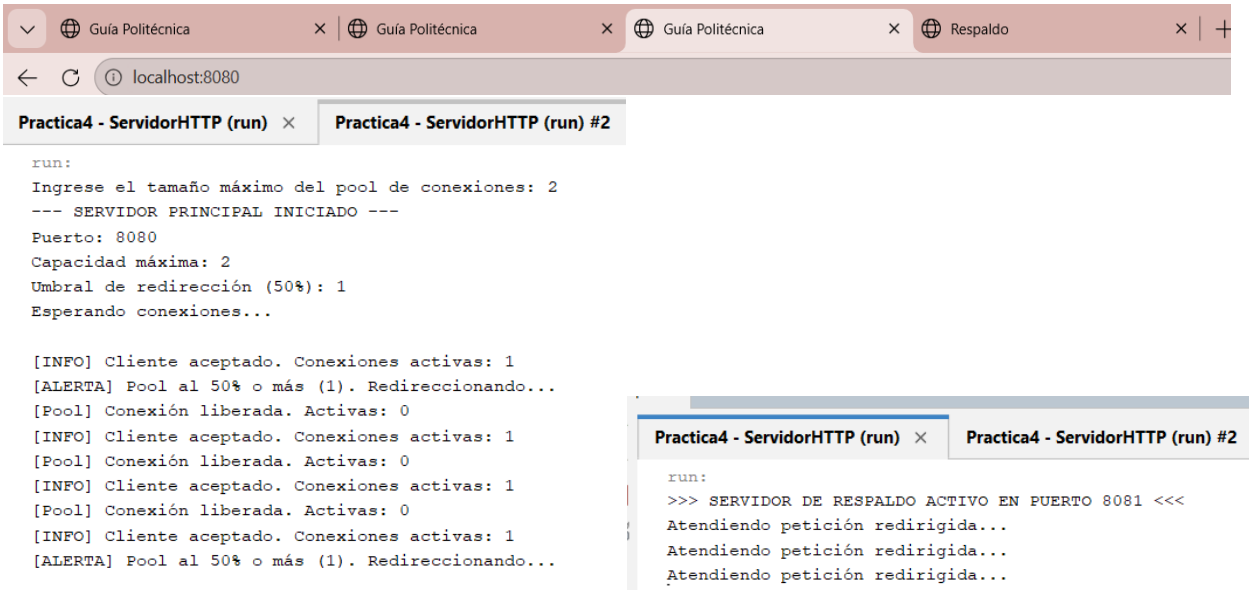

2. Una vez que ejecutamos ambos servidores:



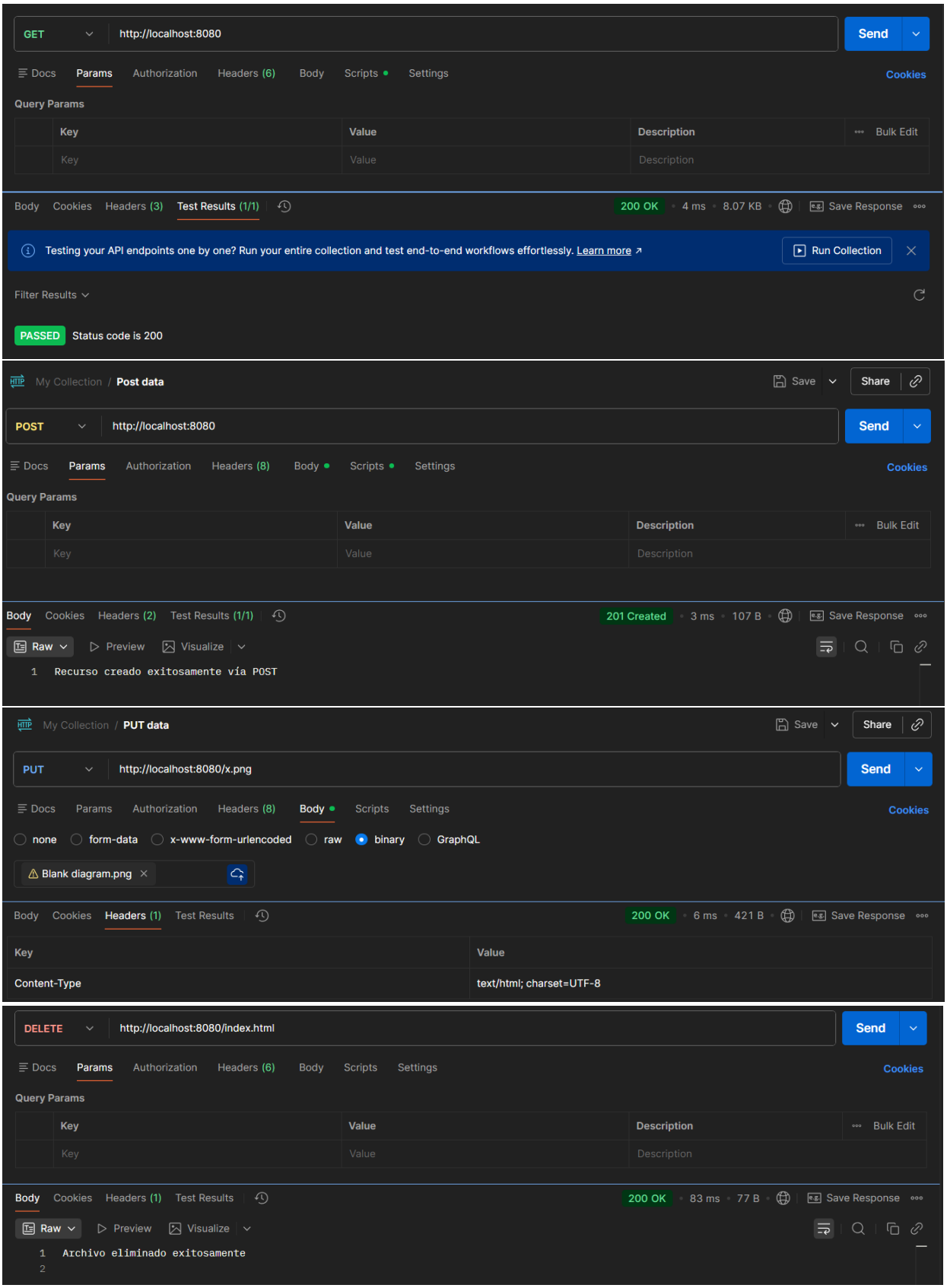
3. Con un pool de tamaño 4 el servidor principal difícilmente llega a a más del 50% de su capacidad, pues la complejidad de la página no es muy alta, entonces el servidor de respaldo no se manda a llamar:




Podemos comprar el numero de redireccionamientos que ocurren con un pool de tamaño 2:



4. Además, se probó también en postman:



# CONCLUSIONES

**Cruz Rodríguez Arely Amairani**

Esta práctica me permitió comprender que el protocolo HTTP es mucho más que un simple intercambio de texto; es un sistema de reglas definidas por organismos como el IETF que garantizan que la web sea universal. Al programar el ClientHandler, aprendí la importancia de interpretar correctamente los encabezados (headers), especialmente el Content-Length, ya que de ello depende que el servidor no se bloquee al recibir datos vía POST o PUT.

Lo más revelador fue implementar los diferentes tipos MIME; ver cómo el mismo flujo de bytes puede ser interpretado por el navegador como una imagen, una tipografía o un documento HTML me dio una perspectiva clara sobre la interoperabilidad. Definitivamente, construir el servidor desde cero con sockets de flujo me ayudó a desmitificar el funcionamiento de la capa de aplicación y a valorar la precisión técnica que requiere el RFC 2616.

**Ortiz Villaseñor Alexandra**

Desde mi punto de vista, lo más valioso de este proyecto fue enfrentar los retos de la concurrencia y la alta disponibilidad. Antes de esta práctica, veía a los servidores como entidades estáticas, pero implementar un **Pool de Hilos** dinámico y un contador atómico en la clase Server me enseñó cómo se gestionan realmente múltiples usuarios en un entorno real.

La lógica de redirección con el código de estado 307 fue el mayor aprendizaje: entender que un servidor no debe simplemente 'caerse' cuando está saturado, sino que debe ser capaz de delegar carga a un BackupServer. Esta experiencia me dejó claro que en la arquitectura Cliente-Servidor, la gestión eficiente de los sockets bloqueantes y la capacidad de respuesta ante el exceso de tráfico son lo que diferencia a una aplicación robusta de una frágil.

# BIBLIOGRAFÍA


- **Estándares y Protocolos Oficiales (IETF)** [1] R. Fielding *et al.*, "Hypertext Transfer Protocol -- HTTP/1.1," IETF, RFC 2616, Jun. 1999. [En línea]. Disponible en: https://datatracker.ietf.org/doc/html/rfc2616
- **Documentación de Lenguaje y API (Programación)** [2] Oracle, "Class ServerSocket," Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification, 2021. [En línea]. Disponible en: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/ServerSocket.html
- [3] Oracle, "Class Socket," Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification, 2021. [En línea]. Disponible en: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/Socket.html
- [4] Oracle, "Package java.util.concurrent," Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification, 2021. [En línea]. Disponible en: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/package-summary.html
- **Libros de Texto y Teoría de Redes** [5] J. F. Kurose y K. W. Ross, *Redes de computadoras: Un enfoque descendente*, 7ma ed. Madrid, España: Pearson Education, 2017.
- [6] A. S. Tanenbaum y D. J. Wetherall, *Redes de computadoras*, 5ta ed. México: Pearson Educación, 2012.
- **Tipos MIME y Medios** [7] IANA, "Media Types," Internet Assigned Numbers Authority, 2024. [En línea]. Disponible en: https://www.iana.org/assignments/media-types/media-types.xhtml
- [8] Google, "Gemini (Versión 1.5 Flash)," [Software de IA generativa]. 2024. Disponible en: https://gemini.google.com. [Asistencia en la arquitectura de software, depuración de código Java y redacción de documentación técnica].