Imagen que contiene Icono

El contenido generado por IA puede ser incorrecto.Dibujo en blanco y negro

El contenido generado por IA puede ser incorrecto.

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales

Aplicaciones para comunicaciones en red

Academia “Sistemas Distribuidos”

Plan 2020

Práctica 20:

“**Pool de hilos**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

Contenido

[Objetivo 3](#_Toc201738331)

[Introducción 3](#_Toc201738332)

[Desarrollo 4](#_Toc201738333)

[3.1 Implementación del Servidor RPC (server.py) 4](#_Toc201738334)

[3.2 Implementación del Cliente RPC (client.py) 5](#_Toc201738335)

[Conclusión 10](#_Toc201738336)

[Pregunta 10](#_Toc201738337)

[¿Cómo podrían generar un negocio a través de lo visto en la práctica? 10](#_Toc201738338)

[Bibliografía 10](#_Toc201738339)

## Objetivo

Desarrollar un servidor de transferencia de archivos (FTP) en Java que utilice un pool de hilos para gestionar de manera eficiente múltiples conexiones de clientes de forma concurrente. El objetivo es comprender y aplicar el patrón de diseño de pool de hilos para mejorar el rendimiento y el control de recursos de una aplicación de red, implementando también un cliente con interfaz gráfica para interactuar con dicho servidor.

## Introducción

Un pool de hilos (thread pool) es un patrón de diseño utilizado en programación concurrente que consiste en gestionar una colección de hilos de trabajo reutilizables. En lugar de crear un nuevo hilo para cada tarea que se debe ejecutar y destruirlo al finalizar, el pool mantiene un número de hilos inactivos, listos para ser asignados a nuevas tareas. Este enfoque mejora significativamente el rendimiento al reducir la sobrecarga asociada a la creación y destrucción de hilos.

Los principales beneficios de utilizar un pool de hilos son:

* **Eficiencia:** Se reutilizan los hilos existentes, evitando el costo computacional de su creación.
* **Control de recursos:** Se limita el número máximo de hilos que pueden estar activos simultáneamente, lo que previene la saturación de recursos del sistema (como CPU y memoria).
* **Gestión simplificada:** Facilita el manejo de la concurrencia, ya que el pool se encarga del ciclo de vida de los hilos.

Los componentes fundamentales de un pool de hilos son:

* **Cola de tareas (Task Queue):** Una estructura de datos que almacena las tareas pendientes. Cuando todos los hilos del pool están ocupados, las nuevas tareas se encolan hasta que un hilo quede libre.
* **Hilos (Threads):** Las unidades de ejecución que toman tareas de la cola y las procesan.
* **Administrador de hilos:** El componente que gestiona la creación, asignación y el ciclo de vida de los hilos en el pool.
* **Tareas (Tasks):** Las unidades de trabajo, que en Java suelen ser instancias de clases que implementan las interfaces Runnable o Callable.

## Desarrollo

Para la práctica, se implementó un sistema cliente-servidor para la transferencia de archivos. El servidor utiliza un pool de hilos para atender a múltiples clientes, y el cliente cuenta con una interfaz gráfica de usuario (GUI) creada con Java Swing.

### **3.1 Implementación del Servidor FTP (**FTPServer.java**)**

El servidor está diseñado para escuchar conexiones entrantes en el puerto 21 y gestionar cada una de ellas de forma concurrente.

* **Inicialización:** En el método main, se crea un ServerSocket para aceptar conexiones y un ExecutorService utilizando Executors.newFixedThreadPool(MAX\_THREADS). Esto crea un pool con un número fijo de 10 hilos, que limitará el número de clientes atendidos simultáneamente.
* **Gestión de Conexiones:** El servidor entra en un bucle infinito donde espera conexiones de clientes con serverSocket.accept(). Por cada cliente conectado, en lugar de crear un nuevo hilo manualmente, se delega la tarea al pool de hilos mediante threadPool.execute(), pasándole una nueva instancia de FTPHandler.
* **Manejador de Clientes (FTPHandler):** Esta clase implementa la interfaz Runnable, lo que permite que sea ejecutada por un hilo del pool. Su método run gestiona la comunicación con un único cliente. Lee el comando enviado por el cliente; si el comando es PUT, extrae el nombre del archivo, envía una confirmación ("READY") y procede a recibir el archivo llamando al método receiveFile.
* **Recepción de Archivos:** El método receiveFile lee los datos del archivo desde el InputStream del cliente en bloques de 4096 bytes y los escribe en un nuevo archivo en el servidor. La transferencia termina cuando se detecta el comando END enviado por el cliente, asegurando que la lectura no se bloquee indefinidamente.

import java.io.\*;

import java.net.\*;

import java.util.concurrent.\*;

public class FTPServer {

private static final int PORT = 21; // Puerto FTP estándar

private static final int MAX\_THREADS = 10;

public static void main(String[] args) throws IOException {

ServerSocket serverSocket = new ServerSocket(PORT);

ExecutorService threadPool = Executors.newFixedThreadPool(MAX\_THREADS);

System.out.println("Servidor FTP iniciado en el puerto " + PORT);

while (true) {

Socket clientSocket = serverSocket.accept();

threadPool.execute(new FTPHandler(clientSocket));

}

}

}

class FTPHandler implements Runnable {

private Socket clientSocket;

public FTPHandler(Socket socket) {

this.clientSocket = socket;

}

@Override

public void run() {

try {

InputStream input = clientSocket.getInputStream();

OutputStream output = clientSocket.getOutputStream();

BufferedReader reader = new BufferedReader(new InputStreamReader(input));

PrintWriter writer = new PrintWriter(output, true);

String command = reader.readLine();

System.out.println("Comando recibido: " + command);

if (command.startsWith("PUT")) {

String fileName = command.split(" ")[1];

// Confirmación de que el servidor está listo para recibir el archivo

writer.println("READY");

// Recibir el archivo

receiveFile(fileName, input, reader, writer);

}

clientSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

private void receiveFile(String fileName, InputStream input, BufferedReader reader, PrintWriter writer) {

try {

File file = new File("servidor\_" + fileName);

FileOutputStream fileOutput = new FileOutputStream(file);

byte[] buffer = new byte[4096];

int bytesRead;

// Recibir archivo hasta que se detecte el comando "END"

while ((bytesRead = input.read(buffer)) != -1) {

fileOutput.write(buffer, 0, bytesRead);

// Verificar si el cliente envió "END"

if (reader.ready() && "END".equals(reader.readLine())) {

break;

}

}

fileOutput.close();

System.out.println("Archivo " + fileName + " recibido con éxito.");

// Enviar respuesta al cliente

writer.println("Archivo " + fileName + " subido correctamente.");

} catch (IOException e) {

e.printStackTrace();

writer.println("Error al recibir el archivo " + fileName);

}

}

}

### **3.2 Implementación del Cliente FTP (**FTPCliente.java **y** FTPClienteGUI.java**)**

El cliente es una aplicación de escritorio con una interfaz gráfica simple para que el usuario pueda seleccionar y subir un archivo al servidor.

* **GUI (FTPCliente.java):** Se crea una ventana (JFrame) con un campo de texto para mostrar la ruta del archivo y un botón para iniciar el proceso de subida. Un ActionListener asociado al botón invoca al método selectFile cuando el usuario hace clic.
* **Selección de Archivo:** El método selectFile utiliza un JFileChooser para permitir al usuario navegar por su sistema de archivos y elegir un fichero. Una vez seleccionado, se llama a uploadFile para comenzar la transferencia.
* **Transferencia de Archivo:** El método uploadFile se conecta al servidor mediante un Socket. Envía el comando PUT seguido del nombre del archivo y espera la respuesta "READY" del servidor. Si la confirmación es recibida, llama a sendFile para transmitir el contenido del archivo. Finalmente, envía un comando END para notificar al servidor que la transferencia ha concluido y espera una respuesta final.
* **Clase Principal (FTPClienteGUI.java):** Esta clase contiene únicamente el método main, cuya responsabilidad es instanciar y hacer visible la ventana del cliente FTP.

import java.awt.\*;

import java.awt.event.\*;

import java.io.\*;

import java.net.\*;

import javax.swing.\*;

public class FTPCliente extends JFrame {

private static final String SERVER\_ADDRESS = "127.0.0.1"; // IP del servidor FTP

private static final int PORT = 21;

private JTextField filePathField;

private JButton uploadButton;

public FTPCliente() {

setTitle("Cliente FTP");

setSize(400, 200);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLayout(new FlowLayout());

filePathField = new JTextField(20);

uploadButton = new JButton("Seleccionar y Subir Archivo");

uploadButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

selectFile();

}

});

add(filePathField);

add(uploadButton);

}

private void selectFile() {

JFileChooser fileChooser = new JFileChooser();

int returnValue = fileChooser.showOpenDialog(this);

if (returnValue == JFileChooser.APPROVE\_OPTION) {

File selectedFile = fileChooser.getSelectedFile();

filePathField.setText(selectedFile.getAbsolutePath());

uploadFile(selectedFile.getAbsolutePath());

}

}

public void uploadFile(String filePath) {

try (

Socket socket = new Socket(SERVER\_ADDRESS, PORT);

OutputStream output = socket.getOutputStream();

InputStream input = socket.getInputStream();

PrintWriter writer = new PrintWriter(output, true);

BufferedReader reader = new BufferedReader(new InputStreamReader(input))

) {

// Enviar comando PUT con el nombre del archivo

File file = new File(filePath);

writer.println("PUT " + file.getName());

// Confirmar que el servidor está listo

String serverResponse = reader.readLine();

if ("READY".equals(serverResponse)) {

System.out.println("Servidor listo para recibir el archivo.");

// Enviar el archivo

sendFile(file, output);

// Confirmar finalización

writer.println("END");

// Respuesta final del servidor

String finalResponse = reader.readLine();

System.out.println("Respuesta final del servidor: " + finalResponse);

} else {

System.out.println("Error: El servidor no está listo. Respuesta: " + serverResponse);

}

} catch (IOException e) {

e.printStackTrace();

}

}

private void sendFile(File file, OutputStream output) throws IOException {

try (FileInputStream fileInput = new FileInputStream(file)) {

byte[] buffer = new byte[4096];

int bytesRead;

System.out.println("Enviando archivo: " + file.getName());

while ((bytesRead = fileInput.read(buffer)) != -1) {

output.write(buffer, 0, bytesRead);

}

output.flush();

System.out.println("Archivo " + file.getName() + " enviado con éxito.");

}

}

}

public class FTPClienteGUI {

public static void main(String[] args) {

FTPCliente cliente = new FTPCliente();

cliente.setVisible(true);

}

}

A screenshot of a computer screen

AI-generated content may be incorrect.  
Figura 1. Servidor en ejecución.

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 2. Envío de archivo a través de la interfaz.

## Conclusión

La práctica se completó de manera exitosa, logrando implementar un servidor FTP concurrente que gestiona múltiples clientes de manera eficiente mediante un pool de hilos. El uso de ExecutorService en el servidor simplificó notablemente la gestión de la concurrencia, eliminando la necesidad de crear y destruir hilos manualmente, lo que se traduce en una mejor utilización de los recursos y un mayor rendimiento.

Se concluye que el patrón de pool de hilos es una herramienta fundamental y poderosa para el desarrollo de aplicaciones de red robustas y escalables. Permite que el servidor mantenga su capacidad de respuesta bajo altas cargas de trabajo, al tiempo que previene el agotamiento de recursos del sistema. La práctica ha sido clave para comprender y aplicar de manera tangible estos conceptos avanzados de programación concurrente.

## Pregunta

### ¿Cómo podrían generar un negocio a través de lo visto en la práctica?

A partir de este modelo, se podría desarrollar y comercializar un servicio de "Managed File Transfer" (MFT). Este servicio ofrecería a las empresas una plataforma segura, escalable y de alto rendimiento para la transferencia de archivos grandes o sensibles. El modelo de negocio sería por suscripción, con diferentes niveles de servicio según el volumen de datos, el número de usuarios y las características de seguridad adicionales (como encriptación avanzada, auditoría y reportes de transferencia).

## Bibliografía

[1] B. Goetz et al., Java Concurrency in Practice. Upper Saddle River, NJ: Addison-Wesley, 2006.

[2] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach, 8th ed. Boston: Pearson, 2021.

[3] The Oracle Corporation, "ExecutorService (Java SE 11 & JDK 11)," Java® Platform, Standard Edition & Java Development Kit Version 11 API Specification. [En línea]. Disponible: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html>. [Consultado: 25-jun-2025].