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 18:

“**Ciclo de vida del hilo**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

Contenido

[Objetivo 3](#_Toc201740561)

[Introducción 3](#_Toc201740562)

[Desarrollo 4](#_Toc201740563)

[3.1 Implementación del Servidor Multihilo (ServidorMultithread.java) 4](#_Toc201740564)

[3.2 Implementación del Cliente (Cliente.java) 6](#_Toc201740565)

[Conclusión 9](#_Toc201740566)

[Pregunta 9](#_Toc201740567)

[¿Cómo podrían generar un negocio a través de lo visto en la práctica? 9](#_Toc201740568)

[Bibliografía 9](#_Toc201740569)

## Objetivo

Analizar y demostrar de manera práctica las distintas fases del ciclo de vida de un hilo en Java. Para ello, se desarrollará una aplicación cliente-servidor donde el servidor será capaz de gestionar múltiples clientes de forma concurrente, creando un hilo de ejecución independiente para cada uno, permitiendo así observar cómo cada hilo transita por los estados de Nuevo, Ejecutable, Bloqueado/En Espera y Terminado.

## Introducción

Un hilo (o *thread*) es la unidad de procesamiento más básica que puede ser gestionada por un sistema operativo, permitiendo que un solo proceso ejecute múltiples tareas de forma concurrente. Todos los hilos dentro de un mismo proceso comparten su espacio de memoria, lo que facilita la comunicación entre ellos pero también exige mecanismos de sincronización para evitar conflictos. El uso de hilos es fundamental para mejorar el rendimiento y la capacidad de respuesta de las aplicaciones, especialmente en servidores de red, que necesitan atender a varios clientes simultáneamente sin que una petición bloquee a las demás.

El ciclo de vida de un hilo describe los diferentes estados por los que pasa desde su creación hasta su finalización. Las fases principales son:

* **New (Nuevo):** El hilo ha sido creado como un objeto (por ejemplo, new Thread()), pero el método start() todavía no ha sido invocado. En este estado, el hilo no está vivo y no consume recursos de CPU.
* **Runnable (Ejecutable):** Una vez que se llama al método start(), el hilo pasa al estado ejecutable. Se encuentra en la cola de hilos listos, esperando a que el planificador del sistema operativo le asigne tiempo de procesador. Este estado puede subdividirse en:
  + **Running (En ejecución):** El hilo está activamente utilizando la CPU.
  + **Ready (Listo):** El hilo está listo para ejecutarse pero está esperando su turno.
* **Blocked/Waiting (Bloqueado/En Espera):** El hilo está vivo pero temporalmente inactivo, ya que está esperando que ocurra un evento externo. Esto puede suceder por varias razones:
  + **Blocked (Bloqueado):** Esperando por un recurso, como un bloqueo de un objeto sincronizado.
  + **Waiting (En espera):** Esperando indefinidamente a que otro hilo lo notifique (por ejemplo, mediante object.wait()).
  + **Timed Waiting (Espera temporal):** Esperando por un período de tiempo específico (por ejemplo, con Thread.sleep() o wait(millis)).
* **Terminated (Terminado):** El hilo ha finalizado su ejecución, ya sea porque su método run() ha concluido de manera normal o debido a una excepción no manejada. Una vez en este estado, el hilo no puede volver a ser ejecutado.

## Desarrollo

Para ilustrar el ciclo de vida de un hilo, se desarrolló una aplicación compuesta por un servidor multihilo y un cliente de consola, ambos en Java.

### **3.1 Implementación del Servidor Multihilo (**ServidorMultithread.java**)**

El servidor está diseñado para aceptar múltiples conexiones de clientes y manejar cada una en un hilo separado.

* **Clase Principal (ServidorMultithread):** Su método main inicia un ServerSocket que escucha en el puerto 1234. Entra en un bucle infinito donde espera conexiones con serverSocket.accept(). Por cada cliente que se conecta, crea una instancia de la clase ClientHandler y la inicia en un nuevo hilo.
* **Clase ClientHandler:** Esta clase implementa la interfaz Runnable, conteniendo la lógica de comunicación en su método run().
  1. **New:** Al hacer new Thread(handler), se crea un nuevo hilo en el estado **Nuevo**.
  2. **Runnable:** La llamada a .start() mueve el hilo al estado **Ejecutable**, listo para ser procesado por la JVM.
  3. **Running/Blocked:** Dentro del método run(), el hilo está en estado **En ejecución** mientras procesa código. Sin embargo, cuando llega a la línea inputLine = in.readLine(), el hilo pasa al estado **Bloqueado**, ya que debe esperar a que el cliente envíe datos a través de la red. Una vez que recibe datos, vuelve al estado **Ejecutable/En ejecución** para procesarlos (imprimirlos y enviar una respuesta).
  4. **Terminated:** Cuando el cliente cierra la conexión, in.readLine() devuelve null, el bucle while termina, y el método run() finaliza. En este momento, el hilo de ese cliente específico pasa al estado **Terminado**. El bloque finally asegura que el socket se cierre correctamente.

import java.io.\*;

import java.net.\*;

public class ServidorMultithread {

private static final int PORT = 1234; // Puerto en el que el servidor escucha

public static void main(String[] args) {

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

System.out.println("Servidor en espera de conexiones...");

while (true) {

// Aceptar la conexión del cliente

Socket clienteSocket = serverSocket.accept();

System.out.println("Cliente conectado: " + clienteSocket.getInetAddress());

// Crear un nuevo hilo para manejar al cliente

ClientHandler handler = new ClientHandler(clienteSocket);

new Thread(handler).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

class ClientHandler implements Runnable {

private final Socket clientSocket;

public ClientHandler(Socket socket) {

this.clientSocket = socket;

}

@Override

public void run() {

try (

// Input stream to read from the client

BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

// Output stream to write back to the client

PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

) {

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println("Cliente dice: " + inputLine);

// Respond to the client (echo example)

String response = "Servidor recibió: " + inputLine;

out.println(response); // Send response back

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

clientSocket.close(); // Close the socket when done

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

### **3.2 Implementación del Cliente (**Cliente.java**)**

El cliente es una simple aplicación de consola que se conecta al servidor. El ciclo de vida de su hilo principal también es observable:

1. Se conecta al servidor a través de un Socket.
2. Entra en un bucle while. En la línea mensaje = console.readLine(), el hilo principal del cliente se **bloquea** esperando la entrada del usuario por teclado.
3. Una vez que el usuario escribe un mensaje y presiona Enter, el hilo vuelve a estar **en ejecución**, envía el mensaje al servidor y se vuelve a **bloquear** en input.readLine(), esta vez esperando la respuesta del servidor.
4. Al recibir la respuesta, la imprime y el ciclo se repite.

import java.io.\*;

import java.net.\*;

public class Cliente {

private static final String SERVER\_ADDRESS = "127.0.0.1"; // Cambia por la IP de tu servidor en Ubuntu

private static final int SERVER\_PORT = 1234;

public static void main(String[] args) {

try (

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

BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

BufferedReader console = new BufferedReader(new InputStreamReader(System.in))

) {

System.out.println("Conectado al servidor. Escribe mensajes:");

String mensaje; // Enviar mensajes al servidor

while ((mensaje = console.readLine()) != null) {

output.println(mensaje); // Enviar mensaje al servidor

System.out.println("Servidor responde: " + input.readLine()); // Leer respuesta del servidor

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

A screenshot of a computer

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

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 2. Cliente 1.

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 3. Cliente 2.

## Conclusión

La práctica permitió observar de manera clara y funcional el ciclo de vida de un hilo en el contexto de una aplicación de red. Se demostró que el servidor es capaz de manejar múltiples clientes simultáneamente gracias a la creación de un hilo dedicado para cada uno. Fue posible identificar los momentos clave en que un hilo transita entre sus estados: desde su creación (New), su puesta en marcha (Runnable), su espera por operaciones de E/S (Blocked), y su finalización (Terminated) cuando el cliente se desconecta.

Se concluye que un entendimiento profundo del ciclo de vida del hilo es indispensable para la programación concurrente, ya que permite diseñar aplicaciones más eficientes, predecir su comportamiento y depurar problemas complejos relacionados con bloqueos, inanición (starvation) o condiciones de carrera.

## Pregunta

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

El modelo de servidor multihilo es la arquitectura base de la mayoría de los servicios en línea. Un negocio podría ser un servicio de chat en tiempo real para empresas (similar a Slack), donde cada conexión de usuario es manejada por un hilo. Se podría ofrecer un modelo freemium, con funcionalidades básicas gratuitas y características avanzadas (historial de mensajes, integraciones, videollamadas) bajo una suscripción de pago, generando ingresos recurrentes.

## Bibliografía

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

[2] H. Schildt, Java: The Complete Reference, Twelfth Edition. New York: McGraw-Hill Education, 2021.

[3] The Oracle Corporation, "Defining and Starting a Thread," The Java™ Tutorials. [En línea]. Disponible: [https://docs.oracle.com/javase/tutorial/essential/concurrency/defining.html](https://www.google.com/search?q=https://docs.oracle.com/javase/tutorial/essential/concurrency/defining.html). [Consultado: 25-jun-2025].