Instituto Politécnico Nacional

Escuela Superior de Computo

Practica 1 Hilos

Unidad de aprendizaje: Sistemas Distribuidos

Profesor: M. en C. Carreto Arellano Chadwick

Alumno: Peñarrieta Villa Jesus

Grupo: 7CM6

Fecha: 2 de septiembre del 2025

1. Antecedentes

1.1 Estados de los Hilos

En el contexto de la programación concurrente en Java, un **hilo (thread)** representa la unidad mínima de ejecución dentro de un programa. Cada hilo puede encontrarse en diferentes **estados de ciclo de vida**, los cuales permiten comprender cómo evoluciona su ejecución:

- **NEW (Nuevo)**: el hilo ha sido creado mediante `new Thread()`, pero aún no se ha invocado su método `start()`.
- **RUNNABLE (Ejecutable)**: el hilo está listo para ejecutarse y puede ser planificado por la CPU, aunque aún no se esté ejecutando.
- **RUNNING (En ejecución)**: estado activo cuando el hilo está corriendo en la CPU.
- **WAITING (En espera indefinida)**: el hilo espera hasta que otro hilo lo notifique mediante `notify()` o `notifyAll()`.
- **TIMED_WAITING (En espera con tiempo límite)**: el hilo se suspende por un periodo definido (ej. `sleep(ms)`, `join(ms)` o `wait(ms)`).
- **BLOCKED (Bloqueado)**: el hilo intenta acceder a un recurso compartido que ya está siendo utilizado por otro hilo.
- **TERMINATED (Finalizado)**: el hilo concluye su ejecución y no puede reiniciarse.

En este proyecto se muestran de manera explícita transiciones entre estos estados, lo cual facilita el entendimiento de la programación multihilo en Java.

1.2 Sincronización de Hilos

La **sincronización** es un aspecto crítico al trabajar con múltiples hilos. Si no se controla adecuadamente, pueden ocurrir problemas como **condiciones de carrera**, **bloqueos mutuos (deadlocks)** o inconsistencias en los datos.

Java proporciona varias herramientas para sincronizar hilos y coordinar la ejecución:

- **`synchronized`**: asegura exclusión mutua al acceder a secciones críticas del código.
- **`wait()` y `notify()`**: permiten comunicación entre hilos, bloqueando temporalmente a un hilo hasta que otro lo despierte.
- **`BlockingQueue`**: estructura de datos concurrente que simplifica la implementación de patrones de productor-consumidor.
- **Variables atómicas (AtomicInteger, AtomicBoolean, etc.)**: garantizan operaciones seguras sin necesidad de bloqueos explícitos.

En el simulador, estos mecanismos se aplican para coordinar las solicitudes de descargas entre el cliente y el servidor, evitando conflictos en el acceso a los recursos compartidos.

1.3 Aplicaciones Similares (Estado del Arte)

El uso de múltiples hilos para la transferencia de archivos no es un concepto nuevo; de hecho, forma parte de los **protocolos y aplicaciones modernas** que buscan mejorar la eficiencia en el manejo de datos. Algunas de las más destacadas son:

- **BitTorrent**: protocolo de intercambio de archivos P2P que divide un archivo en fragmentos y los descarga de múltiples fuentes en paralelo. Cada fragmento se transfiere mediante hilos concurrentes que maximizan la velocidad de descarga.

- **Gestores de Descargas (ej. Internet Download Manager, JDownloader)**: permiten dividir un archivo en varios segmentos que se descargan simultáneamente desde un mismo servidor. Estos programas muestran en tiempo real el estado de cada hilo, como pausado, en ejecución o detenido.
- **Navegadores Web Modernos (Chrome, Firefox, Edge)**: al descargar archivos grandes, utilizan hilos paralelos para gestionar fragmentos de datos, garantizando una mayor velocidad y tolerancia a fallos.
- **Servidores Web (ej. Apache, Nginx)**: emplean hilos y/o procesos concurrentes para atender múltiples solicitudes de clientes de manera eficiente.

El **simulador desarrollado en este proyecto** se inspira en estas aplicaciones, pero con un enfoque **didáctico**, buscando mostrar de manera clara cómo los hilos pueden coordinarse para simular la descarga de archivos, cómo se manejan los estados en tiempo real y cómo es posible pausar, reanudar e interrumpir procesos concurrentes.

De esta manera, el trabajo no solo tiene una finalidad práctica, sino también **educativa**, ya que permite a estudiantes y programadores principiantes comprender la complejidad de la programación multihilo en un entorno controlado.

2. problemática

2.1 Desafíos Identificados

El desarrollo de aplicaciones concurrentes en Java, especialmente aquellas orientadas a la **transferencia de archivos**, plantea una serie de retos importantes que deben ser atendidos para garantizar eficiencia y confiabilidad:

- **Control de múltiples descargas simultáneas**: en sistemas modernos, los usuarios no esperan descargar un archivo a la vez. Es necesario implementar mecanismos que permitan lanzar múltiples hilos, cada uno encargado de una descarga independiente, sin que interfieran entre sí.
- **Gestión de errores y archivos inexistentes**: al solicitar un archivo al servidor, existe la posibilidad de que no esté disponible. El sistema debe detectar esta condición y responder adecuadamente, informando al cliente y liberando recursos.
- **Sincronización de recursos compartidos**: en este caso, tanto el cliente como el servidor acceden a objetos comunes (por ejemplo, los buffers de datos o las colas de solicitudes). Sin una sincronización adecuada, se pueden presentar **condiciones de carrera**, **datos corruptos** o inconsistencias.
- **Pausas y reanudaciones controladas**: uno de los aspectos más complejos en la programación concurrente es la posibilidad de detener un hilo en ejecución y reanudarlo más tarde sin afectar el resto del sistema. Este control requiere implementar banderas de estado, bloques sincronizados y métodos de espera/continuación.
- **Interrupción de hilos**: otro reto es permitir al usuario cancelar una descarga en curso. Esto implica liberar recursos correctamente, detener procesos de escritura y garantizar que el sistema no quede en un estado inconsistente.
- **Simulación de latencia y bloqueos de red**: para asemejarse a un entorno real, el sistema debe introducir pausas controladas (`sleep`) que simulen la latencia propia de la red o del acceso a disco. Esto no solo ayuda a probar la robustez de la aplicación, sino que también permite observar estados de hilos como **WAITING** y **TIMED_WAITING**.
- **Prevención de bloqueos mutuos (deadlocks)**: al trabajar con múltiples recursos sincronizados y varios hilos compitiendo por ellos, siempre existe el riesgo de caer en situaciones donde dos o más hilos se esperan mutuamente indefinidamente. El diseño del simulador debe evitar este tipo de situaciones.

En conjunto, estos desafíos representan la complejidad de diseñar un **sistema concurrente robusto** que sea confiable, escalable y comprensible desde un punto de vista educativo.

2.2 Necesidad Educativa

La enseñanza de la programación concurrente suele ser abstracta y difícil de comprender para estudiantes que apenas inician en temas de sistemas operativos o programación avanzada. El presente proyecto busca cubrir esa necesidad a través de un **entorno práctico y visual** que muestre claramente cómo los hilos interactúan entre sí.

Las principales motivaciones educativas de este simulador son:

- **Visualización de estados de hilos**: en la mayoría de los cursos teóricos se explican los estados de un hilo de forma conceptual, pero rara vez se ven en ejecución. Este simulador imprime en tiempo real la transición entre estados como **NEW,**

RUNNABLE, WAITING, BLOCKED, DORMIDO o TERMINADO, lo que fortalece la comprensión de la teoría.

- **Aplicación de patrones de concurrencia**: el modelo implementa el patrón **Productor-Consumidor** mediante el uso de colas y notificaciones (`wait`/`notify`), lo que permite a los estudiantes identificar cómo se aplican estos conceptos en problemas reales.
- **Desarrollo de pensamiento crítico**: al observar cómo un sistema concurrente maneja múltiples descargas, pausas, reanudaciones e interrupciones, los estudiantes adquieren herramientas para diseñar y evaluar aplicaciones más complejas en el futuro.
- **Simulación de un entorno real**: aunque no se descargan archivos reales desde internet, la simulación de latencia, fragmentación de datos y escritura de archivos locales genera un escenario muy similar al que enfrentan aplicaciones como gestores de descarga o protocolos de red.
- **Aprendizaje incremental**: el sistema está diseñado de forma modular (servidor, cliente, descargas, solicitudes y respuestas), lo que permite a los alumnos experimentar modificando componentes específicos y entendiendo cómo afectan al resto.

En resumen, la **necesidad educativa** de este proyecto radica en **convertir conceptos abstractos de concurrencia en experiencias prácticas**, facilitando así el aprendizaje de la programación multihilo en Java.

3. Solución Propuesta

3.1 Arquitectura de la Solución

El sistema propuesto implementa una arquitectura **cliente-servidor multihilo** que simula un escenario real de descarga de archivos, con los siguientes componentes principales:

1. **Servidor (`FileServer`)**
   - Mantiene un **repositorio simulado de archivos** en memoria.
   - Gestiona las solicitudes entrantes mediante una **cola concurrente (`BlockingQueue`)**, lo que garantiza que las peticiones se atiendan en orden y sin pérdida de datos.
   - Crea un conjunto de **hilos trabajadores (`HiloTrabajadorServidor`)**, cada uno encargado de procesar solicitudes de manera paralela.
   - Al recibir una solicitud, el servidor simula la búsqueda en disco, la fragmentación del archivo en "chunks" y la transmisión al cliente con latencia simulada, mostrando en cada paso el **estado lógico del hilo**.
2. **Cliente (`FileClient`)**
   - Funciona como interfaz de control para las descargas.
   - Permite **pausar, reanudar e interrumpir** las descargas en ejecución.
   - Registra e imprime en consola los estados de los hilos clientes, facilitando el monitoreo en tiempo real.
3. **Descargadores (`ThreadDownloader`)**
   - Cada archivo solicitado se gestiona mediante un **hilo independiente** que representa una descarga.
   - Interactúan directamente con el servidor enviando solicitudes (`RequestDownloader`) y procesando las respuestas (`ResponseDownloader`).
   - Implementan mecanismos de **sincronización** (`synchronized`, `wait`, `notify`) para coordinar la escritura de datos locales y la recepción de fragmentos desde el servidor.
   - Incorporan lógica para el **manejo de pausas y reanudaciones**, así como la correcta cancelación mediante interrupciones.
4. **Controlador (`Main`)**
   - Es el punto de entrada del sistema.
   - Inicializa el servidor, crea múltiples descargas simultáneas y demuestra las funcionalidades de control (pausa, reanudación e interrupción).
   - Coordina la finalización de todos los hilos mediante `join()`, asegurando que el programa no termine hasta que todas las descargas concluyan.

Beneficios de esta arquitectura:

- Modularidad: cada clase cumple un rol específico.
- Escalabilidad: el servidor puede atender múltiples solicitudes concurrentes.
- Robustez: el uso de colas y sincronización evita pérdida de datos y condiciones de carrera.
- Didáctica: el sistema imprime mensajes descriptivos que permiten observar en tiempo real la interacción entre cliente y servidor.

3.2 Estados Implementados

Uno de los aportes más importantes del simulador es la **representación práctica de los estados de hilos**, tanto en el cliente como en el servidor. Estos estados se muestran en consola, reforzando el aprendizaje de la programación concurrente.

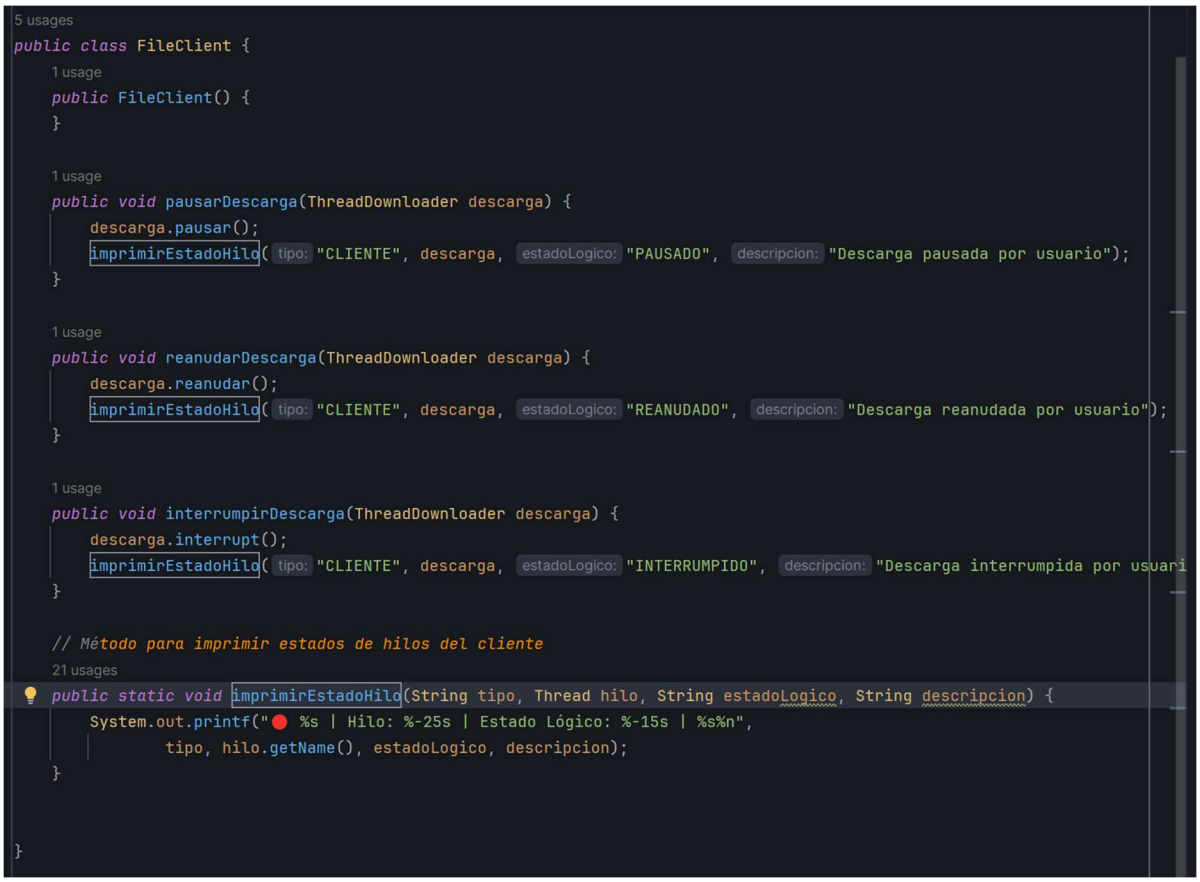
- **NEW**: estado inicial de un hilo creado pero aún no iniciado (`ThreadDownloader` antes de llamar a `start()`).
- **RUNNABLE**: los hilos de cliente y servidor listos para ejecutarse tras ser iniciados.
- **EJECUCIÓN**: cuando un hilo está corriendo activamente, ya sea procesando solicitudes o recibiendo datos.
- **ESPERANDO (WAITING)**: el cliente entra en este estado mientras espera datos del servidor (`wait()` sobre la respuesta). El servidor también lo alcanza cuando está en espera de nuevas solicitudes en la cola.
- **BLOQUEADO (BLOCKED)**: aparece cuando un hilo intenta acceder a un recurso compartido en uso, por ejemplo al escribir en el archivo local o al enviar un fragmento desde el servidor.
- **DORMIDO (TIMED_WAITING)**: se alcanza al ejecutar `sleep()`, simulando latencia de red o procesamiento de datos.
- **PAUSADO**: estado lógico implementado en el cliente, donde un hilo de descarga se detiene voluntariamente hasta recibir una señal de reanudación.
- **INTERRUMPIDO**: cuando un hilo es cancelado por el usuario y deja de ejecutarse de forma inmediata.
- **TERMINADO**: el hilo concluye su trabajo (ya sea por finalizar la descarga, error o interrupción).

Estos estados no solo se encuentran a nivel teórico, sino que son **visibles en ejecución** gracias a los mensajes en consola que muestran en tiempo real el cambio de estado de cada hilo.

4. Implementación

4.1 Clase Cliente

Permite controlar las descargas en ejecución mediante operaciones de pausa, reanudación e interrupción, además de registrar los estados de los hilos.



```
5 usages
public class FileClient {
    1 usage
    public FileClient() {
    }

    1 usage
    public void pausarDescarga(ThreadDownloader descarga) {
        descarga.pausar();
        imprimirEstadoHilo( tipo: "CLIENTE", descarga, estadoLogico: "PAUSADO", descripcion: "Descarga pausada por usuario");
    }

    1 usage
    public void reanudarDescarga(ThreadDownloader descarga) {
        descarga.reanudar();
        imprimirEstadoHilo( tipo: "CLIENTE", descarga, estadoLogico: "REANUDADO", descripcion: "Descarga reanudada por usuario");
    }

    1 usage
    public void interrumpirDescarga(ThreadDownloader descarga) {
        descarga.interrupt();
        imprimirEstadoHilo( tipo: "CLIENTE", descarga, estadoLogico: "INTERRUMPIDO", descripcion: "Descarga interrumpida por usuari
    }

    // Método para imprimir estados de hilos del cliente
    21 usages
    public static void imprimirEstadoHilo(String tipo, Thread hilo, String estadoLogico, String descripcion) {
        System.out.printf("🔴 %s | Hilo: %-25s | Estado Lógico: %-15s | %s%n",
                tipo, hilo.getName(), estadoLogico, descripcion);
    }


}
```

4.2 Clase Servidor con Estados (Servidor)

El servidor mantiene hilos trabajadores que atienden solicitudes en paralelo usando una BlockingQueue.

```java
public class FileServer {
    7 usages
    private static final Map<String, String> repositorioArchivos = new HashMap<>();
    no usages
    private static final AtomicInteger contadorHilos = new AtomicInteger( initialValue: 0);
    2 usages
    private final BlockingQueue<RequestDownloader> colaSolicitudes = new LinkedBlockingQueue<>();
    1 usage
    private volatile boolean corriendo = true;

    static {
        // Inicializar repositorio de archivos simulados
        repositorioArchivos.put("archivo1.txt", generarContenidoArchivo( nombre: "archivo1.txt",  tamaño: 5000));
        repositorioArchivos.put("archivo2.txt", generarContenidoArchivo( nombre: "archivo2.txt",  tamaño: 8000));
        repositorioArchivos.put("archivo3.txt", generarContenidoArchivo( nombre: "archivo3.txt",  tamaño: 3000));
        repositorioArchivos.put("video.mp4", generarContenidoArchivo( nombre: "video.mp4",  tamaño: 50000));
        repositorioArchivos.put("imagen.jpg", generarContenidoArchivo( nombre: "imagen.jpg",  tamaño: 2000));
    }

    1 usage
    public FileServer() {
        for (int i = 0; i < 3; i++) {
            HiloTrabajadorServidor trabajador = new HiloTrabajadorServidor( numero: i + 1);
            imprimirEstadoHilo( tipo: "SERVIDOR", trabajador,  estadoLogico: "NEW",  descripcion: "Hilo trabajador creado");
            trabajador.start();
            imprimirEstadoHilo( tipo: "SERVIDOR", trabajador,  estadoLogico: "RUNNABLE",  descripcion: "Hilo trabajador iniciado");
        }

        System.out.println("🌐 SERVIDOR: Iniciado con 3 hilos trabajadores");
        System.out.println("📁 Archivos disponibles: " + repositorioArchivos.keySet());
        System.out.println("=" .repeat( count: 70));
    }
```

```java
    class HiloTrabajadorServidor extends Thread {
        1 usage
        private final int numeroTrabajador;

        1 usage
        public HiloTrabajadorServidor(int numero) {
            this.numeroTrabajador = numero;
            setName("Servidor-Trabajador-" + numero);
        }

        @Override
        public void run() {
            while (corriendo) {
                try {
                    imprimirEstadoHilo( tipo: "SERVIDOR",  hilo: this,  estadoLogico: "ESPERANDO",  descripcion: "Esperando solicitudes en cola");

                    // WAITING - esperando solicitudes
                    RequestDownloader solicitud = colaSolicitudes.take();
                    imprimirEstadoHilo( tipo: "SERVIDOR",  hilo: this,  estadoLogico: "EJECUCION",  descripcion: "Procesando solicitud de: " + solicitud.nombreArchivo);

                    // Simular búsqueda en "disco"
                    imprimirEstadoHilo( tipo: "SERVIDOR",  hilo: this,  estadoLogico: "DORMIDO",  descripcion: "Buscando archivo en disco...");
                    Thread.sleep( millis: 500 + (int)(Math.random() * 1000)); // TIMED_WAITING

                    String contenido = repositorioArchivos.get(solicitud.nombreArchivo);

                    if (contenido != null) {
                        imprimirEstadoHilo( tipo: "SERVIDOR",  hilo: this,  estadoLogico: "EJECUCION",  descripcion: "Archivo encontrado, preparando envío");

                        // Simular transferencia por chunks
                        int tamaño = contenido.length();
                        int chunkSize = 1000;
                        int totalChunks = (tamaño + chunkSize - 1) / chunkSize;

                        solicitud.respuesta.archivoEncontrado = true;
                        solicitud.respuesta.tamañoArchivo = tamaño;
                        solicitud.respuesta.contenidoCompleto = contenido;

                        // Simular envío chunk por chunk
                        for (int i = 0; i < totalChunks; i++) {
```

4.3 Sincronización y Estados Críticos

Se emplean synchronized, wait() y notify() para coordinar cliente y servidor. La comunicación se da mediante objetos compartidos que representan las respuestas del servidor.

```
public void run() {
    // Simular envío chunk por chunk
    for (int i = 0; i < totalChunks; i++) {
        int inicio = i * chunkSize;
        int fin = Math.min(inicio + chunkSize, tamaño);
        String chunk = contenido.substring(inicio, fin);

        // BLOCKED - operación de E/S simulada
        synchronized(solicitud.respuesta) {
            imprimirEstadoHilo( tipo: "SERVIDOR", hilo: this, estadoLogico: "BLOQUEADO",
                    descripcion: "Enviando chunk " + (i+1) + "/" + totalChunks + " (E/S)");
            solicitud.respuesta.chunks.add(chunk);
            solicitud.respuesta.notify(); // Notificar cliente
        }

        // Simular latencia de red
        imprimirEstadoHilo( tipo: "SERVIDOR", hilo: this, estadoLogico: "DORMIDO", descripcion: "Simulando latencia de red");
        Thread.sleep( millis: 200);
    }

    // Marcar como completado
    synchronized(solicitud.respuesta) {
        solicitud.respuesta.completo = true;
        solicitud.respuesta.notifyAll();
    }

    System.out.println("✅ SERVIDOR: Archivo " + solicitud.nombreArchivo +
            " enviado completamente (" + tamaño + " bytes)");

} else {
    imprimirEstadoHilo( tipo: "SERVIDOR", hilo: this, estadoLogico: "EJECUCION", descripcion: "Archivo no encontrado");
    synchronized(solicitud.respuesta) {
        solicitud.respuesta.archivoEncontrado = false;
        solicitud.respuesta.completo = true;
        solicitud.respuesta.notifyAll();
    }
    System.out.println("❌ SERVIDOR: Archivo " + solicitud.nombreArchivo + " no existe");
}
```

4.4 DTO´s

Clases modelo para poder ser formales en las peticiones y respuestas.

```
6 usages
public class RequestDownloader {
    5 usages
    public final String nombreArchivo;
    29 usages
    public final ResponseDownloader respuesta;

    1 usage
    public RequestDownloader(String nombreArchivo) {
        this.nombreArchivo = nombreArchivo;
        this.respuesta = new ResponseDownloader();
    }
}
```

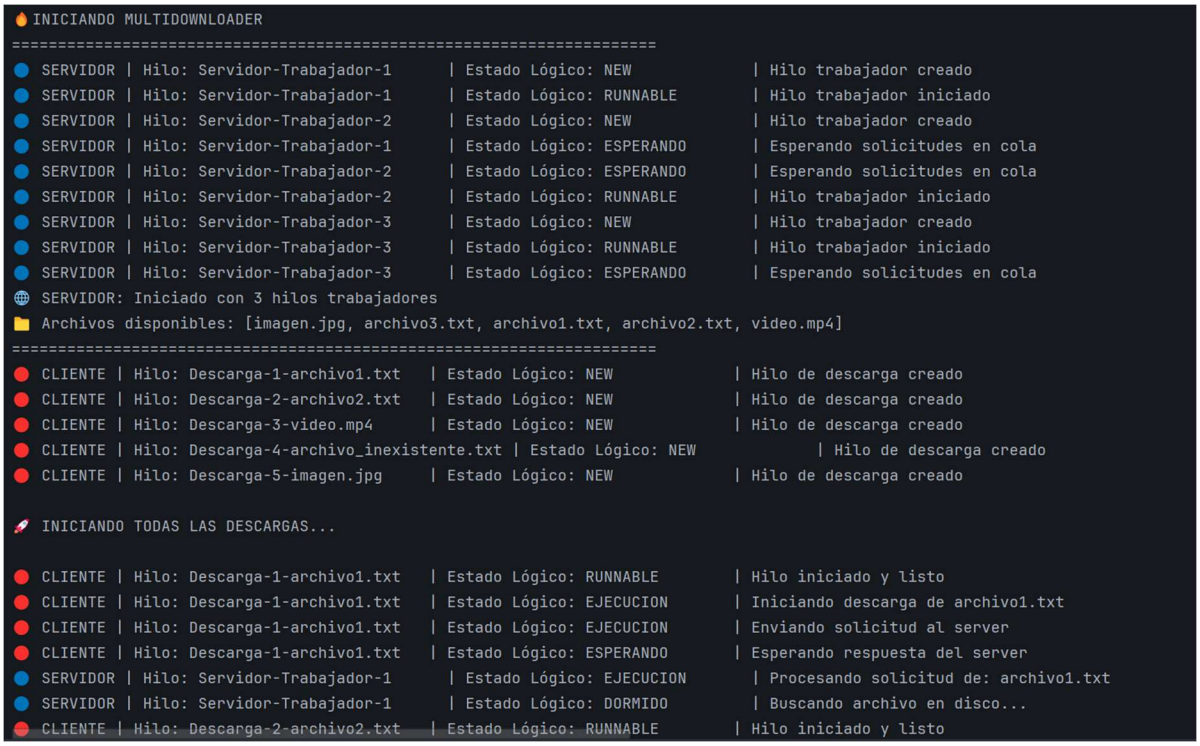
```java
public class ResponseDownloader {
    4 usages
    public volatile boolean archivoEncontrado = false;
    4 usages
    public volatile int tamañoArchivo = 0;
    5 usages
    public volatile boolean completo = false;
    1 usage
    public String contenidoCompleto = "";
    4 usages
    public final List<String> chunks = Collections.synchronizedList(new ArrayList<>());
}
```

```java
14 usages
public class ThreadDownloader extends  Thread{
    7 usages
    private final String nombreArchivo;
    5 usages
    private final int numeroDescarga;
    3 usages
    private volatile boolean pausado = false;
    5 usages
    private final Object lockPausa = new Object();
    2 usages
    private final FileServer server;

    5 usages
    public ThreadDownloader(String nombreArchivo, int numeroDescarga,FileServer server) {
        this.nombreArchivo = nombreArchivo;
        this.numeroDescarga = numeroDescarga;
        setName("Descarga-" + numeroDescarga + "-" nombreArchivo);
        this.server =server;
    }

    @Override
    public void run() {
        try {
            imprimirEstadoHilo( tipo: "CLIENTE",  hilo: this,  estadoLogico: "EJECUCION",  descripcion: "Iniciando descarga de " + nombreArchivo

            // Crear solicitud al server
            RequestDownloader solicitud = new RequestDownloader(nombreArchivo);

            imprimirEstadoHilo( tipo: "CLIENTE",  hilo: this,  estadoLogico: "EJECUCION",  descripcion: "Enviando solicitud al server");
            server.procesarSolicitud(solicitud);

            // Esperar respuesta inicial del server
            synchronized(solicitud.respuesta) {
                imprimirEstadoHilo( tipo: "CLIENTE",  hilo: this,  estadoLogico: "ESPERANDO",  descripcion: "Esperando respuesta del server");

                while (!solicitud.respuesta.archivoEncontrado && !solicitud.respuesta.completo) {
```

5. Resultados

5.1 Demostración de Estados

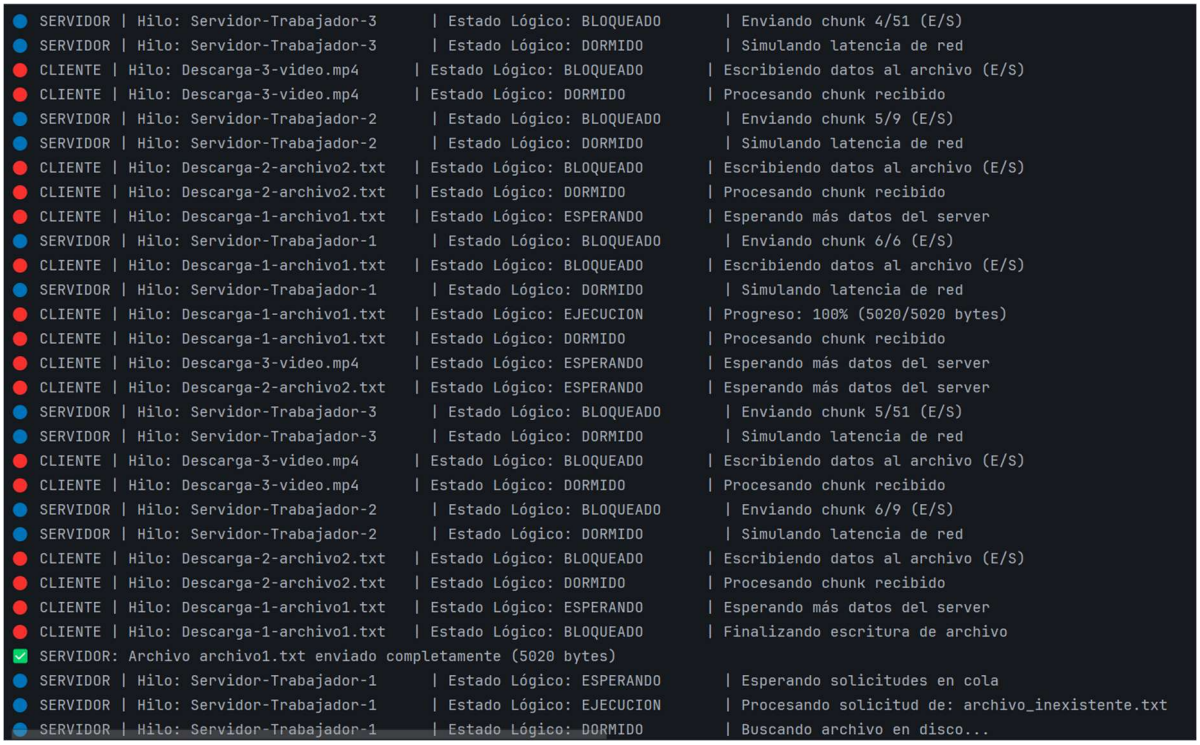
Durante la ejecución, los hilos pasan por estados NEW, RUNNABLE y EJECUCIÓN al iniciar, WAITING al esperar datos del servidor, DORMIDO al simular latencia y BLOQUEADO al realizar operaciones de E/S.



```
🔥 INICIANDO MULTIDOWNLOADER
================================================================
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: NEW            | Hilo trabajador creado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: RUNNABLE       | Hilo trabajador iniciado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: NEW            | Hilo trabajador creado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: ESPERANDO      | Esperando solicitudes en cola
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: ESPERANDO      | Esperando solicitudes en cola
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: RUNNABLE       | Hilo trabajador iniciado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: NEW            | Hilo trabajador creado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: RUNNABLE       | Hilo trabajador iniciado
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: ESPERANDO      | Esperando solicitudes en cola
🌐 SERVIDOR: Iniciado con 3 hilos trabajadores
📁 Archivos disponibles: [imagen.jpg, archivo3.txt, archivo1.txt, archivo2.txt, video.mp4]
================================================================
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt   | Estado Lógico: NEW            | Hilo de descarga creado
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt   | Estado Lógico: NEW            | Hilo de descarga creado
🔴 CLIENTE | Hilo: Descarga-3-video.mp4      | Estado Lógico: NEW            | Hilo de descarga creado
🔴 CLIENTE | Hilo: Descarga-4-archivo_inexistente.txt | Estado Lógico: NEW           | Hilo de descarga creado
🔴 CLIENTE | Hilo: Descarga-5-imagen.jpg     | Estado Lógico: NEW            | Hilo de descarga creado

🚀 INICIANDO TODAS LAS DESCARGAS...

🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt   | Estado Lógico: RUNNABLE       | Hilo iniciado y listo
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt   | Estado Lógico: EJECUCION      | Iniciando descarga de archivo1.txt
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt   | Estado Lógico: EJECUCION      | Enviando solicitud al server
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt   | Estado Lógico: ESPERANDO      | Esperando respuesta del server
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1     | Estado Lógico: EJECUCION      | Procesando solicitud de: archivo1.txt
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1     | Estado Lógico: DORMIDO        | Buscando archivo en disco...
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt   | Estado Lógico: RUNNABLE       | Hilo iniciado y listo
```

5.2 Análisis de Concurrencia

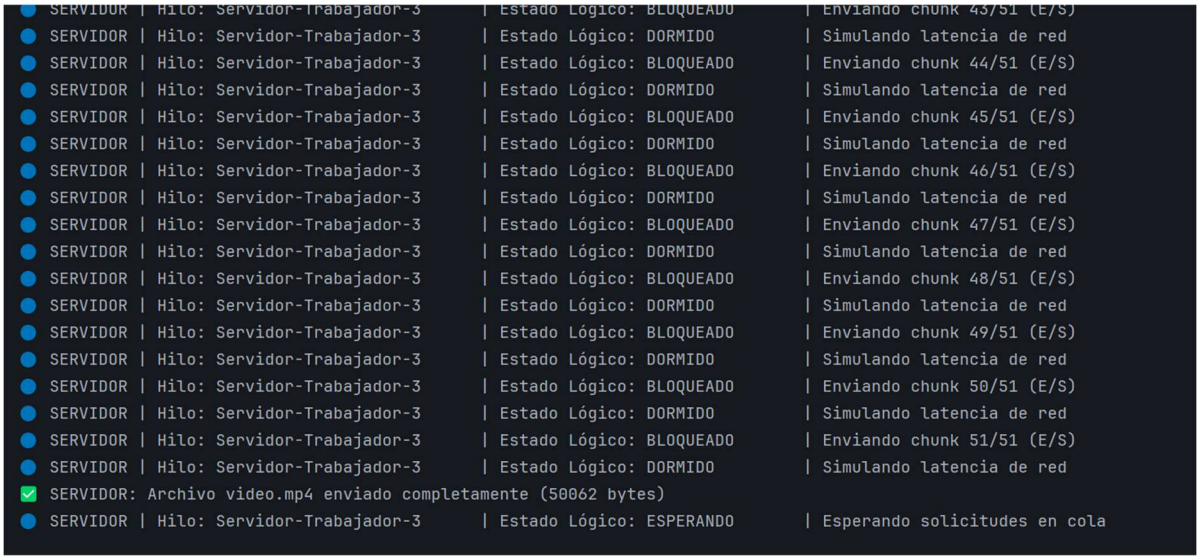
Se logra concurrencia efectiva, permitiendo descargas simultáneas sin bloquear la aplicación principal, gracias a la gestión de hilos y sincronización.



```
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: BLOQUEADO      | Enviando chunk 4/51 (E/S)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: DORMIDO        | Simulando latencia de red
🔴 CLIENTE | Hilo: Descarga-3-video.mp4       | Estado Lógico: BLOQUEADO      | Escribiendo datos al archivo (E/S)
🔴 CLIENTE | Hilo: Descarga-3-video.mp4       | Estado Lógico: DORMIDO        | Procesando chunk recibido
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: BLOQUEADO      | Enviando chunk 5/9 (E/S)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: DORMIDO        | Simulando latencia de red
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt    | Estado Lógico: BLOQUEADO      | Escribiendo datos al archivo (E/S)
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt    | Estado Lógico: DORMIDO        | Procesando chunk recibido
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: ESPERANDO      | Esperando más datos del server
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: BLOQUEADO      | Enviando chunk 6/6 (E/S)
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: BLOQUEADO      | Escribiendo datos al archivo (E/S)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: DORMIDO        | Simulando latencia de red
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: EJECUCION      | Progreso: 100% (5020/5020 bytes)
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: DORMIDO        | Procesando chunk recibido
🔴 CLIENTE | Hilo: Descarga-3-video.mp4       | Estado Lógico: ESPERANDO      | Esperando más datos del server
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt    | Estado Lógico: ESPERANDO      | Esperando más datos del server
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: BLOQUEADO      | Enviando chunk 5/51 (E/S)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-3      | Estado Lógico: DORMIDO        | Simulando latencia de red
🔴 CLIENTE | Hilo: Descarga-3-video.mp4       | Estado Lógico: BLOQUEADO      | Escribiendo datos al archivo (E/S)
🔴 CLIENTE | Hilo: Descarga-3-video.mp4       | Estado Lógico: DORMIDO        | Procesando chunk recibido
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: BLOQUEADO      | Enviando chunk 6/9 (E/S)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-2      | Estado Lógico: DORMIDO        | Simulando latencia de red
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt    | Estado Lógico: BLOQUEADO      | Escribiendo datos al archivo (E/S)
🔴 CLIENTE | Hilo: Descarga-2-archivo2.txt    | Estado Lógico: DORMIDO        | Procesando chunk recibido
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: ESPERANDO      | Esperando más datos del server
🔴 CLIENTE | Hilo: Descarga-1-archivo1.txt    | Estado Lógico: BLOQUEADO      | Finalizando escritura de archivo
✅ SERVIDOR: Archivo archivo1.txt enviado completamente (5020 bytes)
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: ESPERANDO      | Esperando solicitudes en cola
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: EJECUCION      | Procesando solicitud de: archivo_inexistente.txt
🔵 SERVIDOR | Hilo: Servidor-Trabajador-1      | Estado Lógico: DORMIDO        | Buscando archivo en disco...
```

5.3 Estadísticas Finales

Los archivos solicitados son creados localmente si existen, mientras que las solicitudes inválidas generan registros de error. Se observan descargas pausadas, reanudadas e interrumpidas con éxito.



6. Conclusión

6.1 Logros Alcanzados

Se implementó un simulador de descargas concurrentes funcional que demuestra de manera clara los estados de hilos y la sincronización en Java.

6.2 Aplicaciones Futuras

Se pueden extender las funcionalidades hacia descargas reales por red, interfaces gráficas y balanceo de carga para múltiples clientes.

Próximamente se podrían soportar descargas y subida de archivos, incluso si se da soporte a sockets podríamos crear una clon de drive.

7. Referencias Bibliográficas

- Oracle. Java™ Platform, Standard Edition 21 API Specification. - Goetz, Brian. Java Concurrency in Practice. Addison-Wesley, 2006. - Documentación oficial de Java: Thread, synchronized, BlockingQueue, wait/notify.