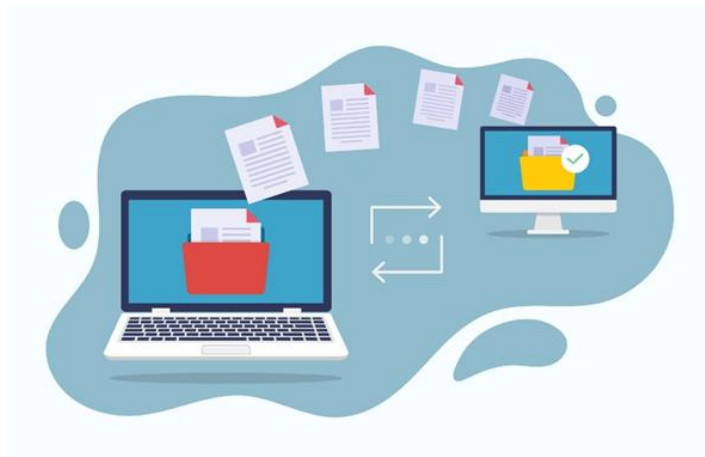


Practica 6

IMPLEMENTACIÓN DE LA APLICACIÓN ESTILO ONEDRIVE CON SOCKET NO BLOQUEANTE



MATERIA

Aplicaciones para comunicaciones de red

PROFESOR

Moreno Cervantes Axel Ernesto

GRUPO

6CM2

ALUMNOS

- Guevara Badillo Areli Alejandra
- Ramírez Martínez Alejandro

FECHA

domingo, 29 de junio de 2025

INTRODUCCIÓN

En esta práctica se desarrolló una aplicación con arquitectura cliente-servidor que simula el funcionamiento básico de una plataforma de almacenamiento en la nube, similar a OneDrive. El sistema permite al usuario gestionar archivos y carpetas de forma remota mediante una interfaz gráfica amigable, creada en Java.

A diferencia de versiones tradicionales que dependen de múltiples hilos para manejar la concurrencia, en esta implementación se utilizó un modelo que permite al servidor atender a varios clientes al mismo tiempo sin necesidad de crear un hilo por cada conexión. Esto se logró mediante un mecanismo que supervisa y gestiona de forma eficiente las operaciones de entrada y salida realizadas por los clientes.

El servidor es capaz de recibir y ejecutar instrucciones enviadas desde uno o varios clientes, permitiendo realizar acciones como transferir archivos, crear o eliminar carpetas y archivos, así como ver el contenido de directorios remotos. La comunicación entre el cliente y el servidor se basa en el envío de códigos de operación que indican claramente qué acción se desea ejecutar.

Esta práctica permite reforzar conceptos clave como la programación en red, la manipulación de archivos en Java, el manejo de múltiples conexiones simultáneas y el diseño de interfaces gráficas que mejoran la experiencia del usuario.

OBJETIVOS

- Desarrollar una aplicación distribuida de almacenamiento remoto, accesible mediante una interfaz gráfica, que permita la gestión eficiente de archivos y carpetas entre el cliente y el servidor.
- Diseñar una interfaz interactiva que permita al usuario seleccionar y navegar tanto por el sistema de archivos local como por el remoto.
- Visualizar el contenido de directorios locales y remotos directamente desde la interfaz gráfica de la aplicación.
- Implementar la transferencia bidireccional de archivos y carpetas entre el cliente y el servidor, incluyendo la preservación de la estructura de las carpetas durante el envío.
- Incorporar funcionalidades para crear, renombrar y eliminar archivos y directorios de forma remota en el servidor.
- Establecer una comunicación eficiente y concurrente entre múltiples clientes y el servidor mediante el uso de sockets TCP no bloqueantes, sin necesidad de crear un hilo por cada conexión.
- Aplicar principios de programación orientada a objetos y manejo de archivos en Java, asegurando modularidad, claridad y reutilización del código.

DESARROLLO

Este reporte se centra en la implementación de sockets no bloqueantes en el servidor de archivos, utilizando las clases `Selector`, `ServerSocketChannel` y `SocketChannel` del paquete `java.nio`. Esta técnica permite manejar múltiples conexiones de clientes con un solo hilo, mejorando la eficiencia del servidor.

Configuración del `ServerSocketChannel` (`Servidor.java`)

El punto de entrada del servidor configura los componentes principales:

```
1 public static void main(String[] args) {
2     // Configuración del shutdown hook
3     Runtime.getRuntime().addShutdownHook(new Thread(() -> {
4         running = false;
5         executorService.shutdown();
6         logger.info("Servidor deteniéndose...");
7     }));
8 }
```

```

9      try (Selector selector = Selector.open();
10          ServerSocketChannel serverChannel = ServerSocketChannel.open()) {
11
12          // Configuración del canal del servidor
13          serverChannel.configureBlocking(false);
14          serverChannel.bind(new InetSocketAddress(PUERTO));
15          serverChannel.register(selector, SelectionKey.OP_ACCEPT);
16
17          logger.info("Servidor iniciado en puerto " + PUERTO);
18
19          // Bucle principal del servidor
20          while (running) {
21              selector.select(1000); // Espera eventos con timeout de 1s
22              // ... manejo de eventos
23          }
24      }
25 }

```

También uno de los principales cambios en comparación con la practica 1 fue la implementación de un método que maneje múltiples conexiones de clientes en el servidor.

```

1 private static void acceptClientConnection(Selector selector, ServerSocketChannel serverChannel) {
2     try {
3         SocketChannel clientChannel = serverChannel.accept();
4         clientChannel.configureBlocking(false); // Configuración no bloqueante
5         SocketAddress clientAddress = clientChannel.getRemoteAddress();
6         int connId = connectionCounter.incrementAndGet();
7
8         clientChannel.register(selector, SelectionKey.OP_READ); // Registra para lectura
9         logger.info("[Conexión #" + connId + "] Cliente conectado desde: " + clientAddress);
10    } catch (IOException e) {
11        logger.log(Level.WARNING, "Error al aceptar conexión: " + e.getMessage(), e);
12    }
13 }

```

- Cada nuevo cliente se configura como no bloqueante inmediatamente después de aceptar la conexión
- Se registra para operaciones de lectura (OP_READ) en el mismo selector

```

1 while (running) {
2     try {
3         // Espera eventos con timeout de 1 segundo (no bloqueante)
4         if (selector.select(1000) > 0) {
5             Set<SelectionKey> selectedKeys = selector.selectedKeys();
6             Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
7
8             while (keyIterator.hasNext()) {
9                 SelectionKey key = keyIterator.next();
10
11                 if (key.isAcceptable()) {
12                     acceptClientConnection(selector, (ServerSocketChannel) key.channel());
13                 } else if (key.isReadable()) {
14                     handleClientRequest((SocketChannel) key.channel());
15                     key.cancel();
16                 }
17                 keyIterator.remove();
18             }
19         }
20     }
}

```

```
21 // ...
22
```

- selector.select(1000) espera eventos con un timeout de 1 segundo (no bloquea indefinidamente)
- Procesa todos los eventos pendientes antes de continuar
- key.cancel() después de manejar una lectura para evitar notificaciones repetidas

En cuanto a las solicitudes al servidor se siguen manejando las mismas que en la practica 1, haciendo uso aun del método procesarOperacion.

```
1 private static void procesarOperacion(int bandera, DataInputStream dis, DataOutputStream dos, int
  connId) throws IOException {
2     switch (bandera) {
3         case 0:
4             recibirArchivo(dis);
5             dos.writeInt(1); // Confirmación
6             break;
7         case 1:
8             enviarListaArchivos(dos);
9             break;
10        case 2:
11            enviarArchivo(dis, dos);
12            break;
13        case 5:
14            eliminarArchivos(dis);
15            dos.writeInt(1);
16            break;
17        case 6:
18            crearArchivo(dis);
19            dos.writeInt(1);
20            break;
21        case 7:
22            crearCarpeta(dis);
23            dos.writeInt(1);
24            break;
25        case 8:
26            renombrar(dis);
27            dos.writeInt(1);
28            break;
29        case 10:
30            String ruta = dis.readUTF();
31            establecerRutaRemota(ruta);
32            dos.writeInt(1);
33            break;
34        case 11:
35            enviarCarpetaCompleta(dis, dos);
36            break;
37        default:
38            logger.warning("[Conexión #" + connId + "] Operación no reconocida: " + bandera);
39            dos.writeInt(-1); // Error
40            break;
41    }
42 }
```

El cliente también emplea configuración no bloqueante para mantener la interfaz de usuario responsiva. En Cliente.java, el método conectar() establece la conexión de manera no bloqueante:

```

1 // Cliente.java - conectar()
2 SocketChannel clientChannel = SocketChannel.open();
3 clientChannel.configureBlocking(false); // Conexión no bloqueante
4 clientChannel.connect(new InetSocketAddress(HOST, PUERTO));
5
6 // Completa la conexión de manera no bloqueante
7 if (!clientChannel.finishConnect()) {
8     throw new IOException("No se pudo conectar al servidor");
9 }

```

Otro cambio importante a comparación con la práctica uno fue el cambio de todos los métodos de solicitudes del cliente a que trabajaran con sockets no bloqueantes. Aquí el ejemplo de una de las implementaciones (enviarArchivo) de este cambio.

```

1 private static void enviarArchivo(File f, JProgressBar barraProgreso) throws IOException {
2     try (SocketChannel channel = conectar();
3         FileInputStream fis = new FileInputStream(f)) {
4
5         // Envío no bloqueante de metadatos
6         ByteBuffer metaBuffer = ByteBuffer.wrap(metaOut.toByteArray());
7         while (metaBuffer.hasRemaining()) {
8             channel.write(metaBuffer);
9         }
10
11        // Envío no bloqueante del archivo
12        byte[] buffer = new byte[BUFFER_SIZE];
13        int n;
14        while ((n = fis.read(buffer)) != -1) {
15            ByteBuffer fileBuffer = ByteBuffer.wrap(buffer, 0, n);
16            while (fileBuffer.hasRemaining()) {
17                channel.write(fileBuffer);
18            }
19        }
20    }
21 }

```

- channel.write() puede devolver inmediatamente si el canal no está listo
- El bucle asegura que todos los datos se envíen eventualmente
- No bloquea el hilo durante operaciones de red lentas

Dentro de la implementación tuvimos algunos problemas ya que no todas las operaciones se benefician del modo no bloqueante. En la descarga de archivos, el cliente cambia temporalmente a modo bloqueante con timeout:

```

1 // Cliente.java - descargarArchivo()
2 try {
3     channel.configureBlocking(true); // Temporalmente bloqueante
4     channel.socket().setSoTimeout(5000);
5
6     // Operaciones de lectura...
7 } finally {
8     channel.configureBlocking(false); // Vuelve a no bloqueante
9 }

```

Esto ya que al ser una operación secuencial requiere de una transferencia completa de datos, y el modo bloqueante simplifica esta lógica de forma significativa. El timeout de 5 segundos se utilizó para evitar que el sistema quede bloqueado indefinidamente en caso de haber problemas de red.

Cambios en la Configuración Inicial

En la implementación actual (visible en ConfiguracionInicial.java), se introduce un paso de configuración previo obligatorio antes de acceder a la interfaz principal del DropBox:

```
1 // ConfiguracionInicial.java - Constructor
2 public ConfiguracionInicial() {
3     // ... configuración de la ventana
4
5     btnLocal.addActionListener(e -> {
6         Cliente.seleccionarCarpetaLocal();
7         lblLocal.setText(Cliente.getRutaActualLocal());
8         verificarConfiguracion();
9     });
10
11    btnRemota.addActionListener(e -> {
12        Cliente.seleccionarCarpetaRemota(modeloRemoto);
13        lblRemota.setText(Cliente.getRutaActualRemota());
14        verificarConfiguracion();
15    });
16
17    btnContinuar.addActionListener(e -> {
18        Dropbox dropbox = new Dropbox();
19        dropbox.setVisible(true);
20        dispose();
21    });
22 }
```

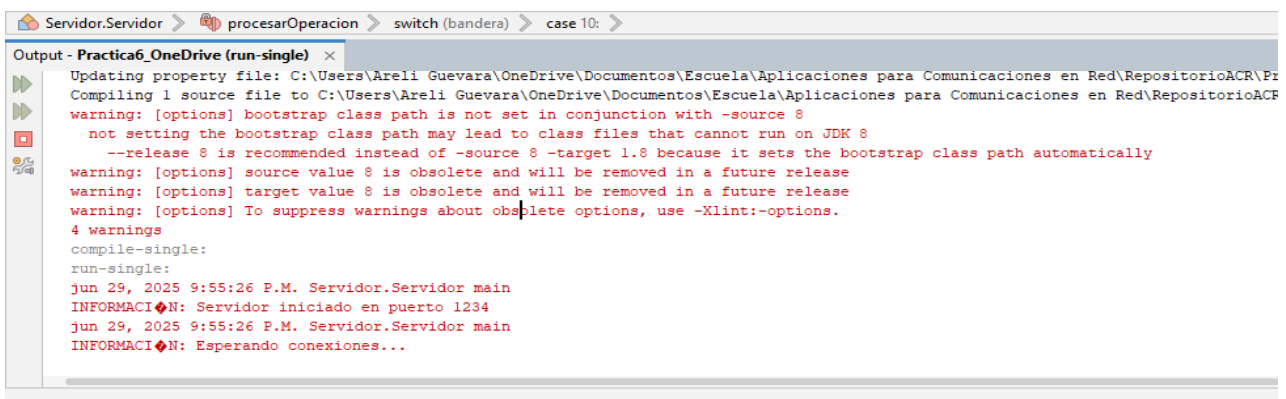
En contraste con la práctica anterior, donde:

1. **Selección dentro de la interfaz principal:** La elección de carpetas ocurría dentro de la misma ventana del DropBox, a través de menús o botones laterales
2. **Configuración opcional:** El usuario podía comenzar a usar la interfaz sin haber configurado las rutas
3. **Cambio dinámico:** Las rutas podían modificarse en cualquier momento durante la sesión

Este cambio se hizo para **anticipar y manejar los fallos de conexión** antes de que el usuario acceda al DropBox, lo cual es crítico en un sistema no bloqueante, donde las operaciones de red son asíncronas y los errores deben gestionarse sin bloquear la interfaz.

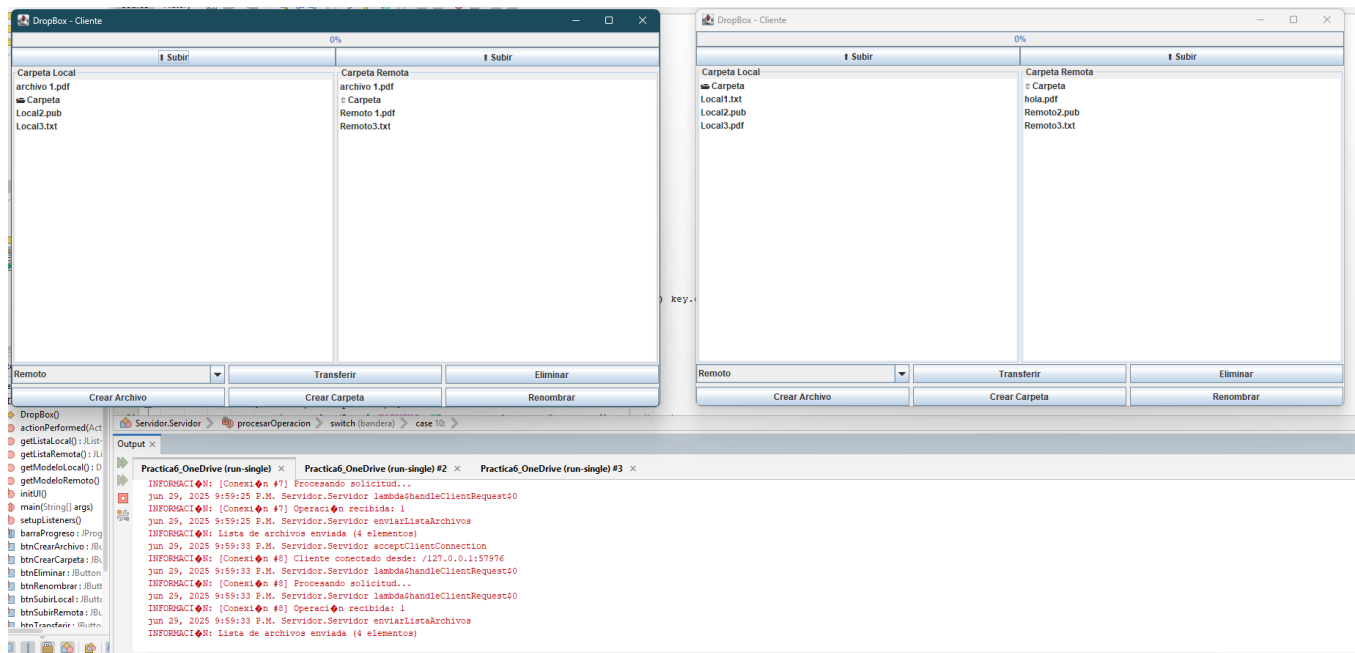
PRUEBAS Y RESULTADOS

Durante las pruebas realizadas al servidor de sockets no bloqueantes, se llevó a cabo un proceso de verificación exhaustivo para garantizar su correcto funcionamiento. En primer lugar, se inicializó el servidor en el puerto **1234**, confirmando que el servicio se activó sin errores y quedó a la espera de conexiones entrantes.



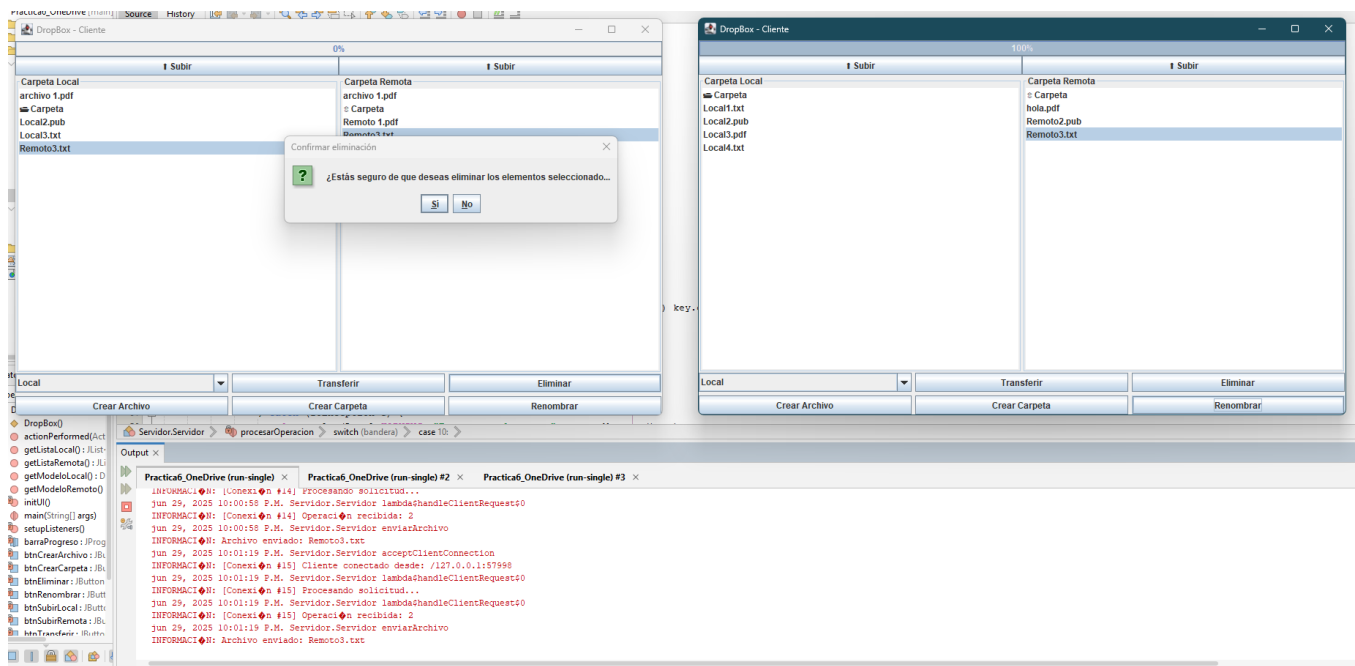
```
Output - Practica6_OneDrive (run-single) x
Updating property file: C:\Users\Areli Guevara\OneDrive\Documentos\Escuela\Aplicaciones para Comunicaciones en Red\RepositorioACR\Pr
Compiling 1 source file to C:\Users\Areli Guevara\OneDrive\Documentos\Escuela\Aplicaciones para Comunicaciones en Red\RepositorioACF
warning: [options] bootstrap class path is not set in conjunction with -source 8
not setting the bootstrap class path may lead to class files that cannot run on JDK 8
--release 8 is recommended instead of -source 8 -target 1.8 because it sets the bootstrap class path automatically
warning: [options] source value 8 is obsolete and will be removed in a future release
warning: [options] target value 8 is obsolete and will be removed in a future release
warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
4 warnings
compile-single:
run-single:
jun 29, 2025 9:55:26 P.M. Servidor.Servidor main
INFORMACION: Servidor iniciado en puerto 1234
jun 29, 2025 9:55:26 P.M. Servidor.Servidor main
INFORMACION: Esperando conexiones...
```

Una vez en operación, se establecieron dos conexiones simultáneas desde clientes distintos, cada uno configurado con rutas locales y remotas diferentes.



A continuación, se ejecutaron diversas operaciones, como la eliminación y el renombrado de archivos, así como transferencias de datos entre cliente y servidor.

Además, se comprobó que cambios como la creación de directorios o archivos que se reflejaran en tiempo real tanto en el servidor como en los clientes.



Los resultados demostraron que todas las operaciones se completaron satisfactoriamente, sin fallos en la sincronización ni interrupciones inesperadas.

CONCLUSIONES

La realización de esta práctica permitió crear una aplicación tipo OneDrive que facilita el manejo de archivos entre una computadora local y un servidor remoto. Usando Java y una interfaz gráfica sencilla, el usuario puede ver, transferir, crear, eliminar y renombrar archivos o carpetas de forma remota de manera clara y funcional.

A diferencia de otros métodos que usan varios procesos al mismo tiempo, esta aplicación emplea un solo socket no bloqueante para controlar y transferir datos, lo que permite atender a varios usuarios sin complicaciones, haciendo el sistema más eficiente.

Durante el desarrollo se fortalecieron conocimientos importantes sobre la comunicación entre computadoras, el trabajo con archivos y carpetas, y la creación de programas prácticos y fáciles de usar. Esta práctica fue una experiencia valiosa para entender cómo se pueden diseñar aplicaciones útiles para la gestión remota de archivos.