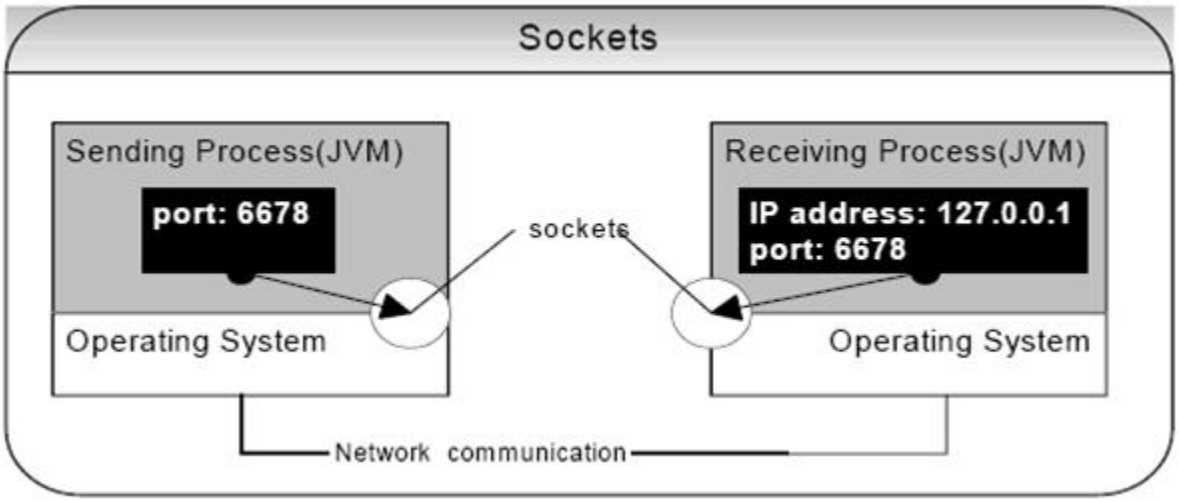
Sockets no orientados a conexiones no bloqueantes

Los sockets no orientados a conexiones usan el protocolo UDP (User Datagram Protocol). UDP es un protocolo de comunicación de red que permite enviar datagramas (paquetes de datos) sin establecer una conexión previa entre el cliente y el servidor. Los sockets no bloqueantes permiten que las operaciones de lectura y escritura no bloqueen la ejecución del programa, es decir, el programa puede seguir ejecutándose mientras espera operaciones de I/O.



Cliente	Servidor
1. **Crear el Socket:** Se crea un socket UDP que se conecta al servidor. 2. **Configurar No Bloqueante:** El socket se configura para operar en modo no bloqueante. 3. **Enviar Datos:** El cliente envía datagramas al servidor. 4. **Recibir Respuestas (Opcional):** El cliente puede recibir respuestas del servidor, dependiendo de la lógica del programa.	1. **Crear el Socket:** Se crea un socket UDP que escucha en un puerto específico. 2. **Configurar No Bloqueante:** El socket se configura para operar en modo no bloqueante. 3. **Esperar Datos:** El servidor usa un Selector para gestionar operaciones de I/O de manera no bloqueante. 4. **Recibir Datos:** El servidor recibe datagramas cuando están disponibles, procesándolos en función de la lógica del programa. 5. **Enviar Respuestas (Opcional):** El servidor puede enviar respuestas a los clientes que envían datos.

Para la presente práctica se busca mandar un archivo desde un cliente a un servidor, por ello, se debe de tener un flujo que se comentará a continuación:

Para el Cliente se envía un archivo al servidor utilizando sockets UDP no bloqueantes. Primero, abre un DatagramChannel en modo no bloqueante y configura la dirección del servidor con InetSocketAddress. Luego, utiliza un ByteBuffer para almacenar datos del archivo, que se lee mediante un ReadableByteChannel obtenido del archivo especificado en el argumento. En un bucle, el cliente lee datos del archivo al buffer, lo envía al servidor con clientChannel.send(), y luego limpia el buffer para la próxima lectura. Después de completar el envío del archivo, cierra tanto el canal de archivo como el canal UDP. Los imports utilizados son FileInputStream para leer el archivo, IOException para manejar errores de entrada/salida, ByteBuffer para manejar datos en bloques, DatagramChannel para el socket UDP, y InetSocketAddress para la dirección del servidor.

Una vez comprendido el funcionamiento de lo que va a realizar el cliente se agrega el siguiente fragmento de código al archivo Cliente.java:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.nio.channels.ReadableByteChannel;
import java.net.InetSocketAddress;

public class UDPFileClient {

    private static final int PORT = 8081;
    private static final int BUFFER_SIZE = 1024;

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java UDPFileClient <file-path>");
            return;
        }

        String filePath = args[0];

        try {
            DatagramChannel clientChannel = DatagramChannel.open();
            clientChannel.configureBlocking(false);
            InetSocketAddress serverAddress = new InetSocketAddress("localhost",
PORT);

            ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
            ReadableByteChannel fileChannel = new
FileInputStream(filePath).getChannel();

            while (fileChannel.read(buffer) > 0) {
                buffer.flip();
                clientChannel.send(buffer, serverAddress);
                buffer.clear();
            }

            fileChannel.close();
            clientChannel.close();
            System.out.println("Archivo enviado al servidor.");

        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

El servidor recibe un archivo enviado por un cliente UDP y guardarlo en el disco. Primero, abre un DatagramChannel en modo no bloqueante y lo enlaza al puerto 8081. Luego, configura un Selector para manejar operaciones de I/O de manera no bloqueante, registrando el canal con el selector para operaciones de lectura. En un bucle infinito, el selector espera eventos de lectura, y cuando hay datos disponibles, lee los datagramas recibidos a través del canal. Utiliza un ByteBuffer para almacenar temporalmente los datos y un WritableByteChannel para escribir esos datos en un archivo denominado "archivorecibido.txt". El código usa FileOutputStream para crear el canal de escritura del archivo, IOException para manejar errores de entrada/salida, ByteBuffer para manejar bloques de datos, DatagramChannel para el socket UDP, Selector y SelectionKey para la gestión eficiente de múltiples operaciones de I/O. Después de recibir y escribir los datos del archivo, imprime la dirección del cliente de origen en la consola.

El fragmento de código que detalla la lógica del servidor es el que se agregará a continuación en el archivo de Servidor.java:

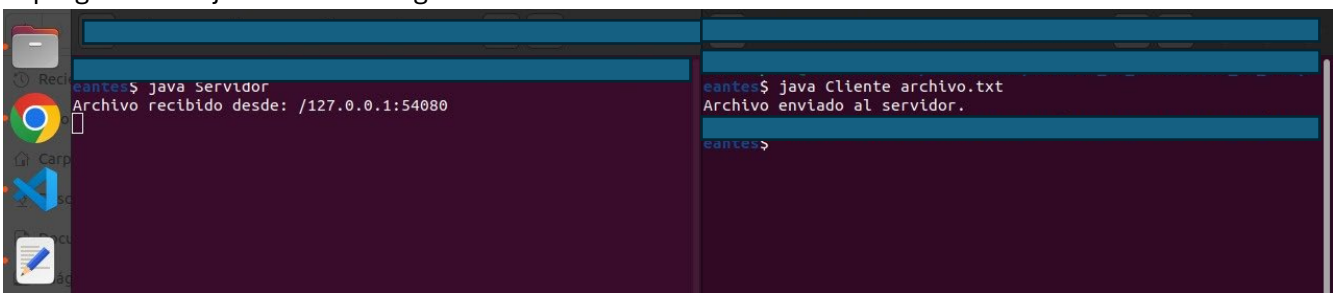
```
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.WritableByteChannel;
import java.net.InetSocketAddress;
import java.util.Iterator;

public class UDPFileServer {

    private static final int PORT = 8081;
    private static final int BUFFER_SIZE = 1024;

    public static void main(String[] args) {
        try {
            DatagramChannel serverChannel = DatagramChannel.open();
            serverChannel.configureBlocking(false);
            serverChannel.bind(new InetSocketAddress(PORT));

            Selector selector = Selector.open();
            serverChannel.register(selector, SelectionKey.OP_READ);

            ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
            WritableByteChannel fileChannel = new
FileOutputStream("archivorecibido.txt").getChannel();

            while (true) {
                selector.select();
                Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
```

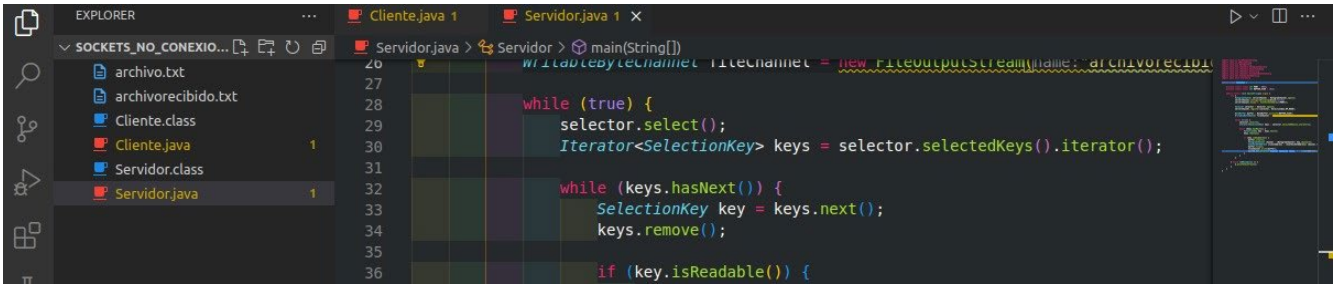
```
                while (keys.hasNext()) {
                    SelectionKey key = keys.next();
                    keys.remove();

                    if (key.isReadable()) {
                        buffer.clear();
                        DatagramChannel channel = (DatagramChannel)
key.channel();
                        InetSocketAddress clientAddress = (InetSocketAddress)
channel.receive(buffer);
                        buffer.flip();
                        fileChannel.write(buffer);
                        System.out.println("Archivo recibido desde: " +
clientAddress);
                    }
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Para compilar los archivos cliente y servidor en java se hace de la siguiente manera:

```
Javac  *.java
```

El programa en ejecución es el siguiente:




EXPLORER
SOCKETS_NO_CONEXIO...
archivo.txt
archivorecibido.txt
Cliente.class
Cliente.java 1
Servidor.class
Servidor.java 1
Cliente.java 1
Servidor.java 1
Servidor.java > Servidor > main(String[])
while (true) {
selector.select();
Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
while (keys.hasNext()) {
SelectionKey key = keys.next();
keys.remove();
if (key.isReadable()) {