TEMA 3.2

SOCKETS NO ORIENTADOS A CONEXIÓN BLOQUEANTES

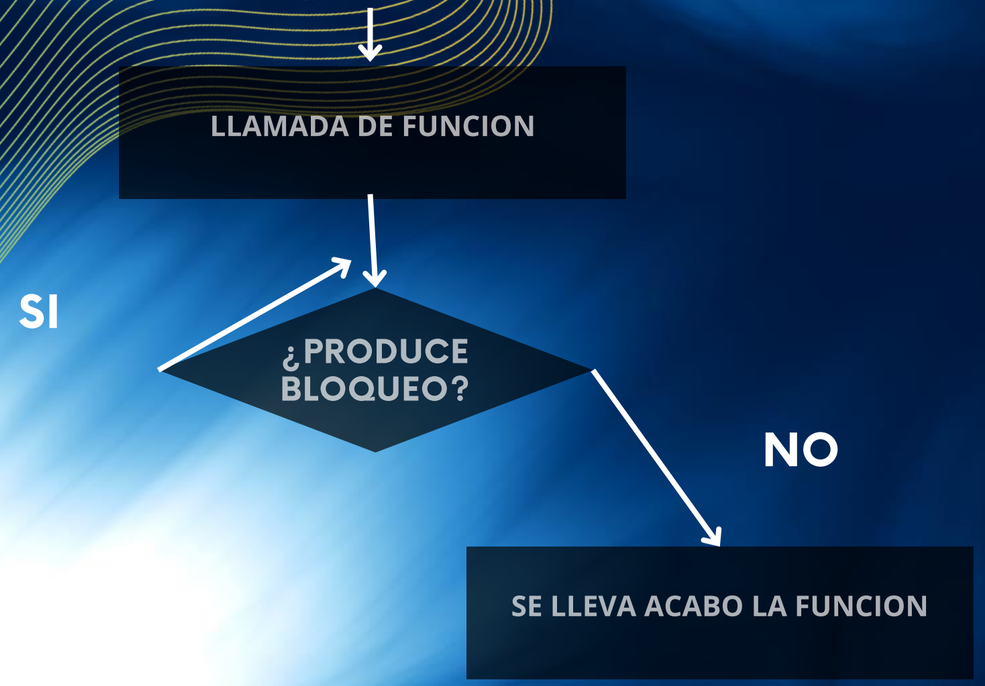
Integrantes:

- Gómez Cano Daniel Aarón
- Guzmán Jiménez Alejandra
- Rodríguez Torres Miguel Angel

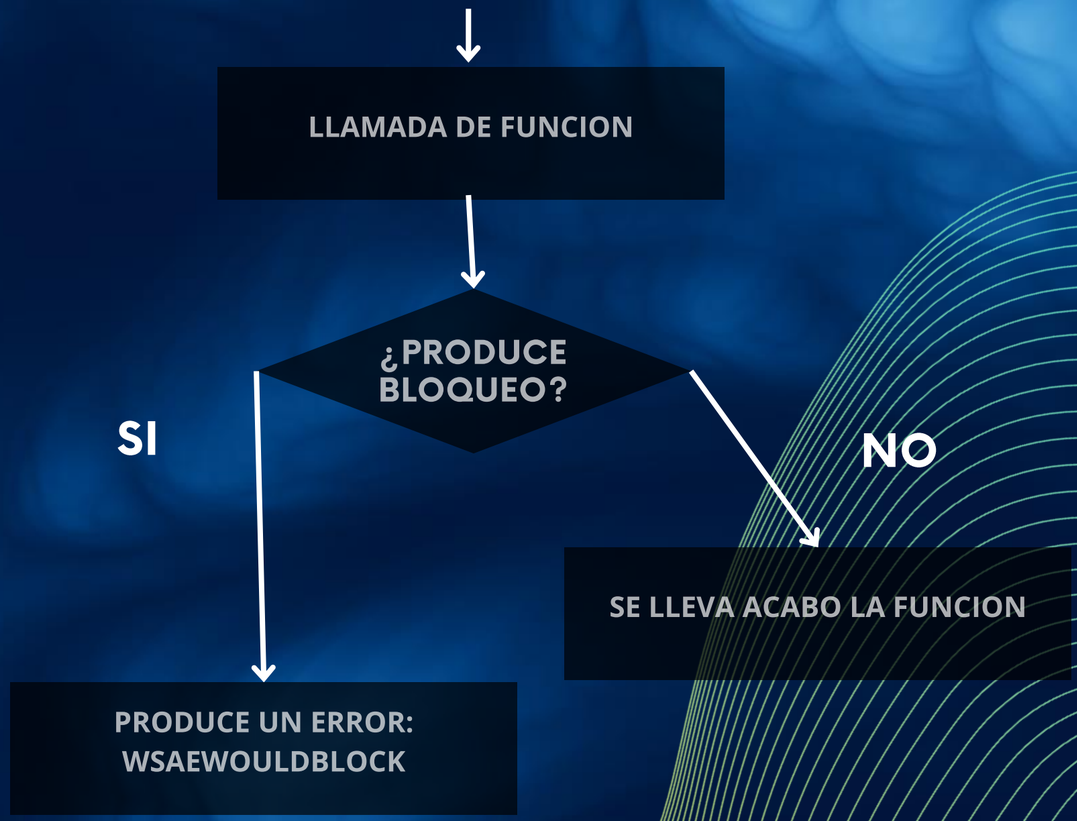
INTRODUCCIÓN

Los sockets no orientados a conexión bloqueantes permiten la comunicación entre procesos sin establecer conexiones previas, lo que los hace eficientes. Aunque detienen la ejecución hasta completar la operación de datos, simplifican el código al no requerir control de flujo. Su uso requiere considerar el impacto en el rendimiento en situaciones de alto tráfico.

FUNCION BLOQUEANTE



FUNCION NO-BLOQUEANTE



EJEMPLO SENCILLO

Suponiendo una situación en un programa enfocado hacia una aplicacion de red.

Donde el primer socket no tiene datos que leer, y un segundo socket esperando datos a ser leídos.

Por consiguiente, el programa quedaría bloqueado en el primer socket porque los datos que espera el segundo socket no serán leídos hasta que lleguen datos por el primer socket.

DISEÑO DE APLICACIONES

A la hora de diseñar aplicaciones se deberá por tanto elegir uno de los dos modelos, bloqueante o no-bloqueante.

Cuando nos decidimos por la versión bloqueante se suele decir que la aplicación diseñada usa llamadas síncronas mientras que en una versión no bloqueante se suele denominar a la aplicación diseñada como asíncrona.

CLIENTES

Cuando un socket está en modo bloqueante, las operaciones de entrada y salida (I/O) bloquearán la ejecución del programa hasta que se complete la operación.

- El cliente establece una conexión con el servidor utilizando el método **connect**().
- El cliente envía y recibe datos utilizando los métodos **send**() y **recv**().
- Si intentas leer datos y no hay datos disponibles, tu programa se detendrá y esperará hasta que lleguen los datos.
- Cuando hayas terminado de enviar y recibir datos, cerrarás la conexión utilizando el método **close**().

EJEMPLO DE PROGRAMA EN JAVA PARA CONEXIONES BLOQUEANTES (CLIENTES)

SE CONECTA A UN SERVIDOR EN UN HOST Y PUERTO ESPECÍFICOS UTILIZANDO SOCKETS BLOQUEANTES

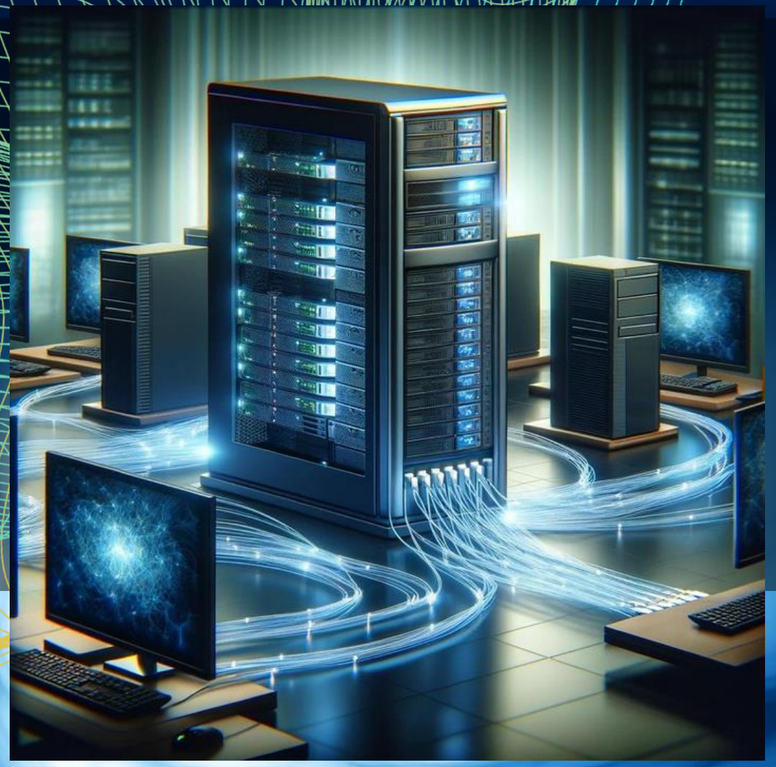
EN JAVA, UN SOCKETCHANNEL BLOQUEANTE HARÁ QUE LAS OPERACIONES DE I/O (ENTRADA/SALIDA) COMO READ() Y WRITE() SE BLOQUEEN HASTA QUE LOS DATOS ESTÉN DISPONIBLES PARA SER LEÍDOS O COMPLETAMENTE ESCRITOS. ESTO SIGNIFICA QUE EL HILO QUE EJECUTA ESTAS OPERACIONES SE DETENDRÁ EN ESTAS LLAMADAS HASTA QUE SE COMPLETE LA OPERACIÓN CORRESPONDIENTE.

```
SocketAddress rama = new InetSocketAddress("rama.poly.edu", 19);
SocketChannel client = SocketChannel.open(rama);
```

```
// Crear una conexión de SocketChannel al servidor en modo bloqueante
        SocketChannel client = SocketChannel.open();
        client.configureBlocking(true); // Asegurar que el modo bloqueante está activado
        client.connect(new InetSocketAddress(host, port));
```

EL PROGRAMA VERIFICA SI SE PROPORCIONARON ARGUMENTOS, OBTIENE LOS PUERTOS . EL CÓDIGO UTILIZA SOCKETCHANNEL QUE ES UNA FORMA DE CREAR UNA CONEXIÓN EN MODO BLOQUEANTE

INICIA UN BUCLE DONDE LEE DATOS DEL SERVIDOR A TRAVÉS DEL SOCKETCHANNEL. EL MÉTODO CLIENT.READ(BUFFER) INTENTA LEER DATOS DEL CANAL DEL SERVIDOR AL BUFFER. SIEMPRE QUE HAYA DATOS PARA LEER (CLIENT.READ(BUFFER) Y NO DEVUELVA -1, LO QUE INDICA EOF).
UNA VEZ QUE SE HAN LEÍDO DATOS EN EL BUFFER, SE INVOCA BUFFER.FLIP() PARA PREPARAR EL BUFFER PARA LA ESCRITURA. LUEGO, LOS DATOS EN EL BUFFER SE ESCRIBEN EN EL CANAL DE SALIDA (OUT.WRITE(BUFFER)), QUE EN ESTE CASO ES LA SALIDA ESTÁNDAR.



```java
// Crear una conexión de SocketChannel al servidor en modo bloqueante
SocketChannel client = SocketChannel.open();
client.configureBlocking(true); // Asegurar que el modo bloqueante está activado
client.connect(new InetSocketAddress(host, port));

while (true) {
    // Leer datos del servidor
    int bytesRead = client.read(buffer);
    if (bytesRead == -1) {
        break; // Finalizar la conexión si se alcanza el final del stream (EOF)
    }

    // Preparar el buffer para la lectura
    buffer.flip();

    // Escribir los datos en la salida estándar
    while (buffer.hasRemaining()) {
        System.out.print((char) buffer.get());
    }

    // Limpiar el buffer para la siguiente lectura
    buffer.clear();
}
```

Ejemplo de servidor

```
import java.nio.*
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;
```

Bibliotecas requeridas para el correcto funcionamiento del siguiente software

```java
public class ChargenServer{
    public static int DEFAULT_PORT = 19;

    Run | Debug
    public static void main(String [] args){
        int port;
        try{
            port = Integer.parseInt(args[0]);
        } catch (RuntimeExceptions ex) {
            port = DEFAULT_PORT;
        }
        System.out.println("Recibiendo conecci
        byte[] rotation = new byte[95*2];
        for (byte i = ' ' ; i <= '~'; i++){
            rotation[i - ' '] = i;
            rotation[i + 95 - ' '] = i;
        }
```

- Se define el puerto default.
- Busca un puerto en los argumentos de ejecución.
- Se crea un arreglo con diversos elementos ASCII.

```java
ServerSocketChannel serverChannel;
Selector selector;
try{
    serverChannel = ServerSocketChannel.open();
    ServerSocket ss = serverChannel.socket();
    InetSocketAdress address = new InetSocketAddres(port);
    ss.bind(address);
```

- **Se crean dos elementos, uno para el canal del socket y un selector para manejar diversos canales.**
- **se obtiene el socket asociado al canal.**
- **socket con la dirección**

```
        serverChannel.configureBlocking(true);
        selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    } catch (IOException ex) {
        ex.printStackTrace();
        return;
    }
```

- **IMPORTANTE: Se configura el socket para que bloquee la conexión.**
- **Se abre el selector y se asocia a la conexión creada**

```java
Set<SelectionKey> readyKeys = selector.selectedKeys();
Iterator<SelectionKey> iterator = readyKeys.iterator();
while (iterator.hasNext()) {
    SelectionKey key = iterator.next();
    iterator.remove();
```

- Se seleccionan todas las llaves seleccionadas.
- Se [illegible]ra el bucle para repetirse todas las veces que sean necesarias.

```java
try {
    if (key.isAcceptable()) {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        SocketChannel client = server.accept();
        System.out.println("Conección aceptada desde: " + client);
        client.configureBlocking(true);
        SelectionKey key2 = client.register(selector, SelectionKey.OP_WRITE);
        ByteBuffer buffer = ByteBuffer.allocate(74);
```

- Se verifica si la llave es aceptable.
- Se le informa a que cliente esta conectado.
- Se bloquea al cliente.
- Se configura al cliente para estar en modo escritura y se le asigna un buffer.

```
    buffer.put(rotation, 0, 72);
    buffer.put((byte) '\r');
    buffer.put((byte) '\n');
    buffer.flip();
    key2.attach(buffer);
```

- Se le asigna al buffer el arreglo mediante saltos de línea
- Se le asigna a la clave el buffer escrito anteriormente.

```
} else if (key.isWritable()) {
    SocketChannel client = (SocketChannel) key.channel();
    ByteBuffer buffer = (ByteBuffer) key.attachment();
    if (!buffer.hasRemaining()) {
        buffer.rewind();
        int first = buffer.get();
        buffer.rewind();
        int position = first - ' ' + 1;
        buffer.put(rotation, position, 72);
        buffer.put((byte) '\r');
        buffer.put((byte) '\n');
        buffer.flip();
    }
    client.write(buffer);
```

- Se verifica si la clave esta lista para escribir al cliente.
- Se obtiene el canal del cliente.
- Se obtiene el buffer asignado a la llave.
- Se le asigna un tamaño al buffer dentro del cliente.
- Se escribe el buffer dentro del cliente.

```
        }
    } catch (IOException ex) {
        key.cancel();
        try {
            key.channel().close();
        } catch (IOException cex) {}
    }
```

- En caso de error se cierra la clave del cliente cerrando el c

CONCLUSIONES

COMO CONCLUSIÓN TENEMOS LA FINALIDAD DEL USO DE SOCKETS NO ORIENTADOS A CONEXIONES BLOQUEANTES, COMO YA SABEMOS ESTOS SOCKETS ESTABLECERÁN UNA CONEXIÓN ENTRE EL CLIENTE Y EL SERVIDOR ANTES DE QUE SE PUEDAN ENVIAR DATOS, LO IMPORTANTE AQUÍ ES QUE ESTO SERVIRÁ PARA LAS APLICACIONES DONDE ES IMPORTANTE GARANTIZAR TANTO EL ENVÍO COMO EL RECIBIDO CORRECTO, ESTO AYUDA EN APLICACIONES WEB, BASE DE DATOS Y TAMBIÉN ES MUY ÚTIL EN CORREOS ELECTRÓNICOS Y POR ESO SU IMPORTANCIA.

REFERENCIAS

QUINTANA SUARÉZ, M. A., & FECANIN ARAUJO, M. (1996). INTRODUCCIÓN A LA PROGRAMACIÓN EN RED: SOCKET Y WINDOWS SOCKETS (1RA ED.). ISLAS CANARIAS: ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIONES. ISBN: 84-87526-39-X.
[VIII-3 - VIII-6]

Harold, E. R. (2013). Java Network Programming (4ta ed.). Sebastopol, USA: O'Reilly Media, Inc
[P247 - P359]