INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

Práctica 1

Carrito de compras

Profesor: Axel Ernesto Moreno Cervantes

Alumnos:

Jiménez Rodríguez Alejandro Martin

Núñez Ramírez Valery Aylin

Grupo: 6CM1

Introducción

El propósito de esta práctica fue **utilizar el modelo de sockets bloqueantes**, a pesar de que los WebSockets no funcionan de manera bloqueante por naturaleza. Para lograrlo, se implementó un mecanismo que simula este comportamiento, permitiendo que el servidor atienda únicamente a un cliente a la vez mientras los demás permanecen en espera.

La aplicación desarrollada consiste en una tienda en línea con carrito de compras. El cliente, implementado en HTML, CSS y JavaScript, se conecta al servidor por medio de WebSocket. Por su parte, el servidor, programado en Java, utiliza una estructura de cola FIFO junto con la designación de un "cliente activo" para reproducir el funcionamiento típico de un servidor con sockets bloqueantes tradicionales.

Esta simulación permitió observar cómo se puede adaptar un protocolo no bloqueante para imitar la atención secuencial y exclusiva de un servidor iterativo, integrando conceptos clave de comunicación en redes dentro de un entorno web moderno.

Arquitectura general

El sistema se compone de dos partes:

1. **Cliente Web (HTML + CSS + JS)**
    - Se conecta mediante WebSocket al servidor.
    - Envía solicitudes como *LISTAR_PRODUCTOS*, *BUSCAR_PRODUCTO* y *FINALIZAR_COMPRA*.
    - Recibe respuestas en formato JSON.
    - Muestra los productos, administra el carrito y genera un ticket en pantalla.
2. **Servidor WebSocket (Java)**
    - Mantiene una sola conexión activa al mismo tiempo.
    - El resto de clientes se guarda en una **cola FIFO**.
    - Solo el cliente activo puede enviar solicitudes.
    - Procesa cada petición de manera secuencial, simulando un servidor con **sockets bloqueantes**.

Simulación del comportamiento de sockets bloqueantes

En los sockets bloqueantes tradicionales, un servidor iterativo atiende a un cliente y no procesa otro hasta terminar con el actual.

Este comportamiento se simuló mediante:

- **clienteActivo** → conexión actualmente atendida.
- **colaClientes** → lista de conexiones en espera.
- **Métodos sincronizados (synchronized)** → para evitar condiciones de carrera.

Cuando un cliente se conecta:

//fragmento de servidorTienda.java

```
public void onOpen(WebSocket conn) {
    synchronized (ServidorTienda.class) {
        if (clienteActivo == null) {
            clienteActivo = conn;
            conn.send("AHORA_ACTIVO");
        } else {
            colaClientes.add(conn);
            conn.send("ESTAS_EN_ESPERA");
        }
    }
}
```

Esto significa que:

- Solo uno es el cliente activo.
- Todos los demás quedan “bloqueados” (esperando turno).

Al desconectarse el cliente activo:

//fragmento de servidorTienda.java

```
private void asignarSiguienteCliente() {
    if (!colaClientes.isEmpty()) {
```

```
            clienteActivo = colaClientes.poll();
            clienteActivo.send("AHORA_ACTIVO");
        }
    }
```

Se asigna el siguiente cliente en orden de llegada, como lo haría un servidor iterativo.

Este mecanismo es la parte **más importante de la práctica**, porque reproduce el comportamiento típico de un servidor con **sockets bloqueantes** utilizando WebSockets.

Procesamiento de solicitudes del cliente (parte crucial)

Cuando el cliente envía un mensaje, este se recibe en:

//fragmento de servidorTienda.java

```
public void onMessage(WebSocket conn, String message)
```

Aquí se filtra:

- Si el cliente **no es el activo**, se le responde con ESTAS_EN_ESPERA.
- Si **sí es el activo**, se procesa su solicitud.

La lógica principal está en:

//fragmento de servidorTienda.java

```
private void procesarSolicitud(JSONObject solicitud, WebSocket conn)
```

Este método analiza el campo "operacion" del JSON y ejecuta la acción correspondiente:

- **LISTAR_PRODUCTOS** → devuelve productos desde articulos.json.
- **BUSCAR_PRODUCTO** → filtra productos por nombre.
- **FINALIZAR_COMPRA** → calcula total, resta stock y genera un ticket.

Aquí es donde ocurre el procesamiento real de las peticiones, equivalente a la parte “bloqueante” del servidor tradicional.

Funcionamiento del cliente Web

El código JavaScript gestiona la interfaz y la comunicación con el servidor.

Conexión WebSocket

//fragmento servidor.js

```
this.ws = new WebSocket('ws://localhost:9000');
```

Mensajes importantes:

- **"ESTAS_EN_ESPERA"** → el cliente debe esperar.
- **"AHORA_ACTIVO"** → ahora puede enviar solicitudes.

Envío de solicitudes

//fragmento servidor.js

```
this.enviarSolicitud({ operacion: 'LISTAR_PRODUCTOS' });
```

El cliente genera peticiones como:

- LISTAR_PRODUCTOS
- BUSCAR_PRODUCTO
- FINALIZAR_COMPRA

Cada una envía un JSON al servidor.

Recepción de respuestas

//fragmento servidor.js

```
this.ws.onmessage = (event) => {
    this.procesarRespuesta(JSON.parse(event.data));
};
```

Aquí procesa:

- listas de productos
- tickets
- errores

Carrito de compras

El carrito usa:

//fragmento servidor.js

```
this.carrito = new Map();
```

Donde se guarda producto y cantidad.
Se actualiza dinámicamente conforme el usuario agrega o quita productos.

Flujo general del sistema

1. El cliente se conecta vía WebSocket.
2. Si no es el cliente activo → queda en espera.
3. Cuando es el activo → puede solicitar productos.
4. El servidor procesa la solicitud de forma exclusiva.
5. El cliente recibe datos y actualiza la pantalla.
6. Cuando finaliza la compra, el servidor:
    - descuenta stock
    - calcula el total
    - genera un ticket
7. El cliente muestra el ticket.
8. Al cerrar la conexión, el servidor pasa el turno al siguiente cliente.

Este ciclo reproduce el comportamiento de **sockets bloqueantes**, pero usando WebSockets modernos.

Conclusiones

Jimenez Rodriguez Alejandro Martin

La práctica permitió comprender cómo funcionan los servidores con **sockets bloqueantes** y cómo este comportamiento puede simularse aun cuando se utiliza una tecnología como WebSockets, la cual normalmente trabaja de forma no bloqueante. Mediante el uso de una cola de espera y la asignación de un cliente activo, logramos reproducir el funcionamiento secuencial de un servidor iterativo. Esto ayudó a visualizar claramente la diferencia entre un sistema que atiende múltiples clientes al mismo tiempo y uno que procesa solicitudes de uno en uno, reforzando los conceptos de concurrencia y control de acceso dentro de la materia de redes de computadoras.

Núñez Ramírez Valery Aylin

Esta práctica fue útil para entender el contraste entre los WebSockets y los **sockets bloqueantes**, así como para comprobar que es posible adaptar un protocolo no bloqueante para que funcione de manera secuencial. El uso de una estructura FIFO permitió imitar el comportamiento clásico de un servidor bloqueante, lo cual facilitó analizar cómo se gestiona el flujo de clientes y el procesamiento de sus solicitudes. En general, la actividad fortaleció la comprensión de los modelos de comunicación cliente–servidor y del control de conexiones en un entorno de red.

Bibliografía

- Tanenbaum, A. S. *Computer Networks*. 5th Edition.
- Stevens, W. Richard. *UNIX Network Programming, Volume 1: The Sockets Networking API*.
- RFC 6455 – *The WebSocket Protocol*.
- Documentación de Java WebSocket API.
- MDN Web Docs: *WebSockets API*.